Abstract

These are my lecture notes and nothing more. That means that they are full of omissions and errors and were meant only as a means of helping me remember what I intended to talk about in class. They may or may not bear any resemblance to what was actually said. They may, however, be useful in providing some small insight into the author’s state of mind and approach to the course material.
# Contents

0 Syllabus

1 Lecture: Introduction and Background
   1.1 Robust Programming ................................................. 11
   1.2 Background: The Ballad of the Unknown Stuntman .................. 12
   1.3 Syllabus ............................................................. 12
   1.4 How to succeed in this course .................................... 16
   1.5 The Last Page ...................................................... 16
   1.6 Foundations .......................................................... 16
      1.6.1 What an operating system is .................................. 16
      1.6.2 Virtualization and Transparency ............................... 17
      1.6.3 What an operating system is not ................................ 17
   1.7 Major subjects for the quarter .................................... 17
   1.8 This quarter ....................................................... 18

2 Lecture: History and Definition of an O.S. 19
   2.1 Announcements ...................................................... 19
   2.2 About the lab ...................................................... 20
   2.3 About the assignment ............................................. 20
   2.4 Everything you wanted to know about C development .............. 20
   2.5 Aside: Review of 357 .............................................. 20
      2.5.1 Review of Unix IO ............................................. 20
      2.5.2 Processes, etc. ............................................... 21
      2.5.3 Compilation ................................................... 21
      2.5.4 Tool of the week: Make(1) .................................... 21
   2.6 Defining of an Operating System: The System Calls .............. 22

3 Lecture: The Process Model 23
   3.1 Announcements ...................................................... 23
   3.2 Two stories .......................................................... 24
   3.3 Defining of an Operating System: The System Calls .............. 24
   3.4 System Calls Again ................................................. 24
   3.5 System Call Mechanisms ........................................... 24
      3.5.1 How to do it ................................................... 24
   3.6 OS Pre-history: The boot process ................................ 25
      3.6.1 How it all begins (on an Intel PC with a floppy) ............ 25
      3.6.2 How it continues .............................................. 25
      3.6.3 And onwards: How does the OS get control back? .......... 25
   3.7 OS History ........................................................... 29
      3.7.1 Ancient times ................................................... 29
      3.7.2 The middle ages ............................................... 29
      3.7.3 The renaissance: families .................................... 29
      3.7.4 Modern Times ................................................... 30
### Lecture: Operating System Structures

1. **Announcements**
   
2. **From last time: History**
   - The middle ages
   - The renaissance: families
   - Modern Times

3. **From last time: The Process (Users' View)**

4. **The Operating System's view: The context**

5. **Example of a context switch**

6. **System Calls Again**

7. **System Call Mechanisms**
   - How to do it
   - And onwards

---

### Lecture: Intro. to Concurrency

1. **Announcements**

2. **Onwards**

3. **The Process Model: a little deeper**

4. **Pseudoparallelism and nondeterminism**

5. **Possible process states**

6. **Scheduling**

7. **What about IPC?**

8. **Operating System Structures**
   - Monolithic Systems
   - Layered Systems
   - Virtual Machines
   - Client-server model

9. **The Layered Architecture of Minix**

10. **Example: Description a MINIX disk interrupt**

---

### Lecture: Lightweight Processes

1. **Announcements**

2. **From last time: Client-server model**

3. **The Layered Architecture of Minix**

4. **Example: Description a MINIX disk interrupt**

5. **Once again: Processes**

6. **Lightweight Processes: Threads**
   - Nine little functions
   - Demonstration: threading in action
   - Review: context switch
   - A thread’s context: stack and registers

7. **What’s this about libraries, then?**
21.4 Page replacement algorithms ................................. 131
  21.4.1 Optimal ....................................................... 131
  21.4.2 Random ..................................................... 131
  21.4.3 Not Recently Used ......................................... 131
  21.4.4 FIFO ......................................................... 131
  21.4.5 Second Chance ............................................. 131
  21.4.6 Clock (optimized Second Chance) ......................... 131
  21.4.7 LRU ......................................................... 131
  21.4.8 Not Frequently Used ...................................... 132

22 Lecture: Wrapping up VM ................................. 133
  22.1 Announcements ................................................ 133
    22.1.1 From Last Time: Inverted page tables ................... 134
  22.2 Page replacement algorithms .................................. 134
    22.2.1 Optimal .................................................. 134
    22.2.2 Random .................................................. 134
    22.2.3 Not Recently Used ...................................... 134
    22.2.4 FIFO ...................................................... 134
    22.2.5 Second Chance .......................................... 134
    22.2.6 Clock (optimized Second Chance) ......................... 134
    22.2.7 LRU ......................................................... 135
    22.2.8 Not Frequently Used ...................................... 135
  22.3 Paging Policies (Design Issues) .......................... 135
    22.3.1 When to page ............................................. 135
    22.3.2 Page allocation ......................................... 136
    22.3.3 Page size ................................................ 136
    22.3.4 VM Tricks ............................................... 136
    22.3.5 Minix ..................................................... 136
  22.4 Filesystems .................................................. 137
    22.4.1 Some thoughts on filesystem organization ............... 137
  22.5 Filesystem Implementation Considerations ................ 137
    22.5.1 Resource Allocation: Disk Management ..................... 137
  22.6 Filesystems .................................................. 138
    22.6.1 Some thoughts on filesystem organization ............... 138
  22.7 Filesystem Implementation Considerations ................ 139

23 Lecture: Intro to Filesystems ............................ 140
  23.1 Announcements ................................................ 140
  23.2 Filesystems .................................................. 140
    23.2.1 Some thoughts on filesystem organization ............... 141
  23.3 Filesystem Implementation Considerations ................ 141
    23.3.1 Resource Allocation: Disk Management ..................... 141
  23.4 Foreshadowing: The Minix Filesystem ...................... 142
List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>A Linux open() implementation</td>
<td>26</td>
</tr>
<tr>
<td>2</td>
<td>Minix open() implementation</td>
<td>27</td>
</tr>
<tr>
<td>3</td>
<td>Minix _syscall()</td>
<td>27</td>
</tr>
<tr>
<td>4</td>
<td>Minix sendrec() abstracted</td>
<td>28</td>
</tr>
<tr>
<td>5</td>
<td>The process of a context switch</td>
<td>35</td>
</tr>
<tr>
<td>6</td>
<td>Possible states for a process</td>
<td>38</td>
</tr>
<tr>
<td>7</td>
<td>The lowest layer of the operating system</td>
<td>39</td>
</tr>
<tr>
<td>8</td>
<td>Layout of the Minix system</td>
<td>41</td>
</tr>
<tr>
<td>9</td>
<td>Layout of the Minix system</td>
<td>44</td>
</tr>
<tr>
<td>10</td>
<td>Stack development (Remember that the real stack is upside-down)</td>
<td>49</td>
</tr>
<tr>
<td>11</td>
<td>Peterson’s solution for mutual exclusion</td>
<td>56</td>
</tr>
<tr>
<td>12</td>
<td>A producer-consumer implementation with a race condition</td>
<td>57</td>
</tr>
<tr>
<td>13</td>
<td>A semaphore-based producer-consumer implementation</td>
<td>58</td>
</tr>
<tr>
<td>14</td>
<td>A monitor-based solution to the producer-consumer problem</td>
<td>60</td>
</tr>
<tr>
<td>15</td>
<td>A semaphore-based producer-consumer implementation</td>
<td>62</td>
</tr>
<tr>
<td>16</td>
<td>A monitor-based solution to the producer-consumer problem</td>
<td>64</td>
</tr>
<tr>
<td>17</td>
<td>A message-passing solution to the producer-consumer problem</td>
<td>65</td>
</tr>
<tr>
<td>18</td>
<td>A message-passing solution to the producer-consumer problem</td>
<td>68</td>
</tr>
<tr>
<td>19</td>
<td>Possible states for a process</td>
<td>68</td>
</tr>
<tr>
<td>20</td>
<td>Layout of the Minix system</td>
<td>77</td>
</tr>
<tr>
<td>21</td>
<td>How a deadlock forms</td>
<td>80</td>
</tr>
<tr>
<td>22</td>
<td>How a deadlock forms</td>
<td>83</td>
</tr>
<tr>
<td>23</td>
<td>One dimensional example of Dijkstra’s Bankers Algorithm</td>
<td>85</td>
</tr>
<tr>
<td>24</td>
<td>Another example, but one that doesn’t wedge immediately.</td>
<td>86</td>
</tr>
<tr>
<td>25</td>
<td>Resource trajectory example from Tanenbaum</td>
<td>87</td>
</tr>
<tr>
<td>26</td>
<td>Multi-dimensional Bankers Algorithm</td>
<td>89</td>
</tr>
<tr>
<td>27</td>
<td>Resource trajectory example from Tanenbaum</td>
<td>91</td>
</tr>
<tr>
<td>28</td>
<td>Multi-dimensional Bankers Algorithm</td>
<td>92</td>
</tr>
<tr>
<td>29</td>
<td>Minix IO</td>
<td>94</td>
</tr>
<tr>
<td>30</td>
<td>Minix IO</td>
<td>105</td>
</tr>
<tr>
<td>31</td>
<td>Histogram of scores for the midterm</td>
<td>115</td>
</tr>
<tr>
<td>32</td>
<td>Virtual Memory translation in the MMU (4k pages)</td>
<td>125</td>
</tr>
<tr>
<td>33</td>
<td>Virtual Memory translation in the MMU (4k pages)</td>
<td>129</td>
</tr>
<tr>
<td>34</td>
<td>Partition table entry</td>
<td>143</td>
</tr>
<tr>
<td>35</td>
<td>Anatomy of a disk</td>
<td>143</td>
</tr>
<tr>
<td>36</td>
<td>Anatomy of a filesystem</td>
<td>143</td>
</tr>
<tr>
<td>37</td>
<td>Minix superblock</td>
<td>144</td>
</tr>
<tr>
<td>38</td>
<td>Minix inode</td>
<td>145</td>
</tr>
<tr>
<td>39</td>
<td>Minix directory ent</td>
<td>145</td>
</tr>
<tr>
<td>40</td>
<td>useful Minix numbers</td>
<td>146</td>
</tr>
<tr>
<td>41</td>
<td>A minix file with 16-byte blocks and 4-byte zone numbers</td>
<td>149</td>
</tr>
<tr>
<td>42</td>
<td>Partition table entry</td>
<td>150</td>
</tr>
<tr>
<td>43</td>
<td>Anatomy of a disk</td>
<td>151</td>
</tr>
<tr>
<td>44</td>
<td>Anatomy of a filesystem</td>
<td>151</td>
</tr>
<tr>
<td>45</td>
<td>Minix superblock</td>
<td>152</td>
</tr>
<tr>
<td>46</td>
<td>Minix inode</td>
<td>152</td>
</tr>
</tbody>
</table>
List of Tables

1  Integer registers of the x86 CPU ................................. 47
0 Syllabus
cpe453 — Operating Systems I
Section 03 (Nico)
Spring 2019

Administrivia

Professor: Dr. Phillip Nico
pnico@calpoly.edu
Office: 14-205

Office Hours: in my office (14-205):
Monday: 3:10pm–4:00pm¹
Tuesday: 9:10am–10:00am¹
Wednesday: 3:10pm–4:00pm¹
Thursday: —
Friday: 3:10pm–4:00pm¹
or by appointment


You might also find the following books to be valuable resources:


Webpage: http://www.csc.calpoly.edu/~pnico/class/2019-02/cpe453

Lecture: Section 3: MWF 10:10am–11:00am 20-139

Lab: Section 4: MWF 11:10am–noon 14-255

Midterm: Wednesday, May 8th, 2019 (subject to change)

Final²: (Unreliable. Verify with the schedule of classes.)
Section 3: Monday, June 10, 2019 10:10am–1:00pm

Grading: The approximate grading breakdown (subject to change) is:
Laboratory Exercises 15%
Programming Assignments 25%
Midterm 25%
Final 35%

¹Office hours are guaranteed until the earlier of the posted end time or the time at which there are no more students.
²There is a chance that the final will be moved to one of the common final times this quarter.
Course Objectives

The purpose of this course is to gain experience with and understanding of operating systems principles and implementation. In the process, you will:

- Examine the requirements of a modern operating system, including the fundamental problems of managing concurrent processes.
- Understand the system call interface to an operating system.
- Understand how an operating system gets started (boots) and takes control of the machine.
- Understand the design and implementation of a filesystem.
- Learn a bunch of other interesting things.

Prerequisites

The prerequisites for this course are cpe 225 or cpe 233 and cpe 357. Cpe 453 assumes proficiency at writing programs in the C language and understanding of the compilation, assembly, linking and loading processes, as well as understanding of contemporary computer architecture at the instruction set level.

If you have any doubts, come talk to me.

Course Format

The course consists of three lectures and three labs a week. The labs will not meet every time—possibly not ever—but I reserve the right to call lab meetings for demos or exercises, etc. You are responsible for all material covered in either lecture or lab. If you miss a class, consult a classmate for any missed materials.

The purpose of the class is for everyone to understand about operating systems and their implementation. To this end, if you don’t understand something during class, ask. If you are confused, it is likely that a few dozen of your classmates are as well. Also, listen to others’ questions. Many times you’ll think you understand a concept until you hear someone else’s question about it. Dialogue is the best way to learn things, so don’t be afraid to speak up.

Office Hours

Office hours are as listed above or by appointment. If you are unable to come to the posted office hours, contact me and we can arrange to meet. There is no reason why any of you should be unable to see me if you need to.

Other Resources

I will maintain a class web page at http://www.csc.calpoly.edu/~pnico/class/2019-02/cpe453. On it I will keep information, assignments, announcements, etc. If there any class announcements, corrections, etc., to be made, I will post them in the Announcements section of the class web page. Please check the web page regularly. I will try to make any announcements in class as well, but I cannot guarantee it, and you don’t want to miss anything.

I usually post my lecture notes on the web. These are guaranteed to be incomplete and are not a substitute for class attendance. They will, I hope, provide a framework for your own notes and emphasize what I think is important about the class.
Depending on how much time I have, I will also maintain a FAQ on the class page of questions from office hours that seem to be of general interest.

**Laboratory Exercises and Programming Assignments**

**Programs:**
There will be a number of assignments over the quarter. These will consist of some combination of programming exercises and reports on system modifications. Together they will represent 25% of the total final grade for the course. Because the total number is not known at this point, the individual program weights will not be determined until the end of the quarter. I expect there will be around 5 assignments over the quarter, but this may vary.

Programming assignments will be distributed on the web, and each assignment will specify both when it is due and whether or not partnerships are allowed.

**Labs/Problem Sets:**
Most weeks of the quarter there will be a set of laboratory exercises intended to supplement the coursework. These will consist of written exercises and/or experimental work to be performed in the lab. Submission requirements will be posted with each lab.

**Late Policy:**
Each student will be allowed three (3) discretionary late days to be applied over the quarter. One late day will be required for each calendar day (not work day) or portion thereof after the due date. To submit work after the submission directory for an assignment has been closed, use the `latedays` program to reopen it for you. Instructions will be provided on the class web site.

Note: There may be some assignments for which the use of late days will not be allowed. This would be to facilitate the posting of solutions before the midterm or some other reason like that. It will be noted on these assignments that late days are not permitted.

If you are unable to complete an assignment by the specified time and do not have any more late days, turn in what you have for partial credit. **Late assignments will receive no credit.**

**Partnerships:**
On some of the assignments you may be working with partners. Collaboration tends to help with figuring out difficult concepts and generally makes the whole process more pleasant. A word of caution, though: While it is tempting to just divide up the work, be sure each partner understands the whole project. Concepts learned on the assignments will show up on exams which are worth far more than the individual assignments in the final analysis. Even if your partner bails you out of a tight spot, be sure you understand the work, or it will come back to haunt you. Be absolutely certain that both partners’ names appear on all assignments. Credit will be given only to students whose names appear on the assignment.

Each assignment will specify whether working with a partner is permitted.

**Submitting Written Work:**
Written work should be submitted in class on the day due or given to me in advance. If you use the CSC Department drop box (outside of 14-254) be sure that the names of the instructor, course, section, and assignment (in addition to your own name) are written clearly. Also be aware that materials dropped in the box after it has been collected for the day will be recorded the next day as late. If you are unsure whether the box has been emptied for the day, ask in the office.

For all written work, in order to facilitate grading:

- Write on only one side of the paper.
- Start a new page for a new problem if you will not be able to complete it on the current page.
- Place the problems in the order assigned.
• Fold your papers in half *lengthwise* and write your name (Last, First), the homework number, and the due date in the top right hand corner of the outside sheet.

• Do not write in red.

• Use proper English grammar and punctuation.

• Write legibly.

Neatness will not necessarily help you, but sloppiness will definitely hurt. If I can’t read it, I can’t grade it, and I will not guess at what you meant. (This also applies to exams.) These rules may seem extreme, but they are intended to make grading easier so I can return your papers more quickly.

**Submitting Programs:**
Programming assignments will be submitted online on the Computer Science Department CSL machines using the *handin* program. Instructions for submitting programs will be provided on the class web page.

When turning in programming assignments, be careful to submit your final version and to have tested it before submitting.

*Programs that fail to compile will receive no credit.*

**Exams**
There will be a midterm and a final. The midterm will be worth 25% of the final course grade and the final will be worth 35%. Exams will emphasize insight and problem solving ability rather than memorization.

**Missed Exams:** Makeup exams will only be given for the gravest of reasons. If you must miss an exam due to extreme illness, etc., contact the instructor (phone or email is fine) or leave a message with the Department of Computer Science office (805-756-2824) *before* the exam. Be sure to leave both the reason for missing the exam and how to reach you.

**Grading Policy**
The final grade for this course will be constructed of three components in the following proportions:

<table>
<thead>
<tr>
<th>Component</th>
<th>Percentage</th>
</tr>
</thead>
<tbody>
<tr>
<td>Laboratory Exercises</td>
<td>15%</td>
</tr>
<tr>
<td>Programming Assignments</td>
<td>25%</td>
</tr>
<tr>
<td>Midterm</td>
<td>25%</td>
</tr>
<tr>
<td>Final</td>
<td>35%</td>
</tr>
</tbody>
</table>

In general, letter grades will be computed according to the scale:

- Minimum A–: 90%
- Minimum B–: 80%
- Minimum C–: 70%
- Minimum D–: 60%

That is, if you score 90% on an exam you will be guaranteed at least an A– for that exam. On some assignments and exams the minimum cutoffs may (and probably will) be lowered to account for difficulty.

**Note:** I reserve the right to take other circumstances into account when assigning final grades. These include, but are not limited to, such things as substantial improvement over the quarter, significant differences between exam and homework performance, missing homeworks, etc.
Nothing would make me happier than to give everyone an A.

**Missing Assignments:** Unless special arrangements have been made in advance, a good-faith effort is required of each student for every program. Failure to submit a homework disqualifies the student from the top grade level possible for a final grade. Failure to submit two disqualifies from the top two levels, etc.

**Regrades:** In general, papers to be considered for regrades must be submitted within one week of the time they become available for picking up. The same is true for misrecorded grades. They must be reported within a week of their posting. Grade listings will be mailed out periodically as things are graded. Please check them to be sure they agree with your own records.

Collaboration and Cheating

**Policy on Collaboration**

Programming assignments in this class are intended to be demonstrations of individual or partnership abilities. To this end, programs are to be written only by the designated authors. High-level discussion of problems and problem-solving techniques, however, is beneficial to all involved. You are encouraged to discuss approaches so long as those with whom you consult are given due credit in your program headers.

- *It is never acceptable to allow someone else to have your work for reference or to refer to someone else’s work while writing your own.*

In this case, “someone else’s work” means not only other students’ programs, but also materials from any other source, including, but not limited to, the Internet, other reference books, or previous course materials. Also, “not giving your work to others” includes taking reasonable precautions to prevent them from taking it.

Collaboration that goes beyond general approaches or that is uncredited will be considered cheating. If you are unsure about what constitutes proper or improper collaboration, consult the instructor for guidance.

**Policy on Cheating**

Don’t. I consider academic dishonesty a serious offense. Any instances of cheating or plagiarism will be referred to the Office of Student Rights and Responsibilities. The Cal Poly rules and policies are listed in the catalog as well as at the OSRR web site, [http://www.osrr.calpoly.edu](http://www.osrr.calpoly.edu). The general policy, however, is very simply stated in the Campus Administrative Manual (C.A.M. 684):

- *Cheating requires an “F” course grade*

Turning in work is presumed to be a claim of authorship unless explicitly stated otherwise. If the course rules are unclear or you are unsure of how they apply, ask the instructor beforehand.

Feedback

One of the frustrations of teaching is that the instructor rarely gets any feedback on the course until the teaching evaluations at the very end when it is too late to do anything about it. If you like, dislike, or don’t understand something I’m doing with the course, please stop by my office hours, send me email, or paste together a note from newspaper clippings and drop it in my mailbox. I won’t always change things, but I will always explain why I’m doing them the way I am.

Tentative class schedule

The tentative schedule for the course is given on page 8. This is what we hope to accomplish and when we hope to accomplish it. There may be changes, but this is a rough roadmap for the quarter.
# Spring 2019 Preliminary Course Outline

(Subject to Change)

<table>
<thead>
<tr>
<th>Week</th>
<th>Lecture</th>
<th>Topic</th>
<th>Reading</th>
<th>Date</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>—</td>
<td>Cesar Chavez Birthday (observed)</td>
<td></td>
<td>April 1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>Introduction and Background</td>
<td>Chapter 1</td>
<td>April 3</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>History</td>
<td></td>
<td>April 5</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>The Process Model</td>
<td>Chapter 2</td>
<td>April 8</td>
</tr>
<tr>
<td>2</td>
<td>4</td>
<td>Processes, cont.</td>
<td></td>
<td>April 10</td>
</tr>
<tr>
<td>2</td>
<td>5</td>
<td>Operating System Structures</td>
<td></td>
<td>April 12</td>
</tr>
<tr>
<td>3</td>
<td>6</td>
<td>Concurrency</td>
<td></td>
<td>April 15</td>
</tr>
<tr>
<td>3</td>
<td>7</td>
<td>Concurrency, cont.</td>
<td></td>
<td>April 17</td>
</tr>
<tr>
<td>3</td>
<td>8</td>
<td>Even More Concurrency</td>
<td></td>
<td>April 19</td>
</tr>
<tr>
<td>4</td>
<td>9</td>
<td>Wrapping up Concurrency</td>
<td></td>
<td>April 22</td>
</tr>
<tr>
<td>4</td>
<td>10</td>
<td>Scheduling</td>
<td></td>
<td>April 24</td>
</tr>
<tr>
<td>4</td>
<td>11</td>
<td>Input and Output</td>
<td></td>
<td>April 26</td>
</tr>
<tr>
<td>5</td>
<td>12</td>
<td>Device Management</td>
<td></td>
<td>April 29</td>
</tr>
<tr>
<td>5</td>
<td>13</td>
<td>IO Systems and Deadlock</td>
<td></td>
<td>May 1</td>
</tr>
<tr>
<td>5</td>
<td>14</td>
<td>IO Systems, cont.</td>
<td></td>
<td>May 3</td>
</tr>
<tr>
<td>6</td>
<td>15</td>
<td>MINIX IO Architecture</td>
<td></td>
<td>May 6</td>
</tr>
<tr>
<td>6</td>
<td>16</td>
<td>Midterm</td>
<td></td>
<td>May 8</td>
</tr>
<tr>
<td>6</td>
<td>17</td>
<td>Memory Systems</td>
<td></td>
<td>May 10</td>
</tr>
<tr>
<td>7</td>
<td>18</td>
<td>Dynamic Segmentation</td>
<td></td>
<td>May 13</td>
</tr>
<tr>
<td>7</td>
<td>19</td>
<td>Virtual Memory</td>
<td></td>
<td>May 15</td>
</tr>
<tr>
<td>7</td>
<td>20</td>
<td>Even More Memory</td>
<td></td>
<td>May 17</td>
</tr>
<tr>
<td>8</td>
<td>21</td>
<td>Page Replacement Algorithms</td>
<td></td>
<td>May 20</td>
</tr>
<tr>
<td>8</td>
<td>22</td>
<td>Virtual Memory Wrapup</td>
<td></td>
<td>May 22</td>
</tr>
<tr>
<td>8</td>
<td>23</td>
<td>Filesystems</td>
<td></td>
<td>May 24</td>
</tr>
<tr>
<td>9</td>
<td>—</td>
<td>Memorial Day</td>
<td></td>
<td>May 27</td>
</tr>
<tr>
<td>9</td>
<td>24</td>
<td>Filesystems, cont.</td>
<td></td>
<td>May 28</td>
</tr>
<tr>
<td>9</td>
<td>25</td>
<td>Directory Schemes</td>
<td></td>
<td>May 29</td>
</tr>
<tr>
<td>9</td>
<td>26</td>
<td>Index Schemes</td>
<td></td>
<td>May 31</td>
</tr>
<tr>
<td>10</td>
<td>27</td>
<td>Other file systems</td>
<td></td>
<td>June 3</td>
</tr>
<tr>
<td>10</td>
<td>28</td>
<td>Wrapping up Filesystems</td>
<td></td>
<td>June 5</td>
</tr>
<tr>
<td>10</td>
<td>29</td>
<td>Wrapping up</td>
<td></td>
<td>June 7</td>
</tr>
<tr>
<td></td>
<td>Final Exam</td>
<td>Monday, June 10m 10:10am–1:00pm (Verify with published schedule)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

---

2Last day to drop classes is Thursday, April 11.
The Last Page

This page is so I can gather a little information about you at the beginning of the class. Please fill it out, tear it off and leave it with me on the way out.

Who are you?

Name: ____________________________
Section: __________________________
Major: ____________________________
Email: ____________________________

Class Expectations?

Please take a minute to write out what your goals and expectations are for cpe 453. What do you want to learn?

I hate to do this, but to be sure there’s no confusion on the matter...
Below, please copy the two boxed text segments from page 5 about academic dishonesty and sign the pledge (assuming you will comply, of course). Without this you will automatically receive a grade of zero for all assignments.

1)

2)

<table>
<thead>
<tr>
<th>Pledge</th>
</tr>
</thead>
<tbody>
<tr>
<td><em>I will do my own work in this class. That is, unless it is explicitly permitted by the assignment, I will neither use others’ work as my own nor make my work available for others to use. I understand that either of these actions constitutes cheating sufficient to merit a grade of F for the course.</em></td>
</tr>
</tbody>
</table>

Signature: ____________________________ Date: __________
1 Lecture: Introduction and Background

Outline:
Robust Programming
Background: The Ballad of the Unknown Stuntman
Syllabus
Preparation
How to succeed in this course
The Last Page
Foundations
  What an operating system is
  Virtualization and Transparency
  What an operating system is not
Major subjects for the quarter
This quarter

- Check on the book. (Third edition, Racoons)
- Make sure your lab accounts work.
- Assignments and labs are out
  - Coming attractions:
    | Event | Subject              | Due Date | Notes               |
    |-------|----------------------|----------|---------------------|
    | asgn5 | minget and minls     | Wed Jun 5 23:59 |
    | asgn6 | Yes, really          | Fri Jun 7 23:59 |
    | final | stuff                | Sat Jun 8 10:10 (in 03-201) |

  Use your own discretion with respect to timing/due dates.

- Office hours
  - Come
    - With high probability they are:
      Monday: 3:10pm–4:00pm\(^1\)
      Tuesday: 9:10am–10:00am\(^1\)
      Wednesday: 3:10pm–4:00pm\(^1\)
      Thursday: —
      Friday: 3:10pm–4:00pm\(^1\)
    - These may be changed as things get sorted out.
    - Also lab times
    - If you’ve been through this before unsuccessfully, tell me. I can ask you awkward questions that’ll improve your odds of never doing 453 again.
    - A note on persistence

1.1 Robust Programming

Do it.

\(^{1}\)Office hours are guaranteed until the earlier of the posted end time or the time at which there are no more students.
1.2 Background: The Ballad of the Unknown Stuntman

Usually—if all is going well, that is—we aren’t even aware of the operating system. The operating system of a computer is the group of stagehands making everything work. Without them, the show couldn’t go on, but you only notice if something doesn’t work.

“The OS is our enemy?”

Have you ever wondered why your PC won’t speak to your printer even though the printer seems to be being extremely polite?

1.3 Syllabus

- The Course Staff
  
  me: Phillip Nico
  pnico@calpoly.edu
  Office: 14-205

- About this course:
  
  What is this course about?
  Why Study OS?
  
  – Because it explains the magic
  
  What does that mean?
  
  – OS Theory:
    * Concurrent programming (deadlock avoidance)
    * Scheduling algorithms (fairness, starvation)
    * Resource allocation (e.g., Memory allocation (paging) algorithms)
  
  – OS Implementation: How it’s really done (The lab)
  
  – Stark Raving Paranoia: The OS can’t stop, can it?
  
  – Course Objectives
  
  The purpose of this course is to gain experience with and understanding of operating systems principles and implementation. In the process, you will:
  
  * Examine the requirements of a modern operating system, including the fundamental problems of managing concurrent processes.
  * Understand the system call interface to an operating system.
  * Understand how an operating system gets started (boots) and takes control of the machine.
  * Understand the design and implementation of a filesystem.
  * Learn a bunch of other interesting things.

- Prerequisites:

  **203 and (225 or 233)** Know how to program and understand architecture.
  
  **cpe 357** Be familiar with the C programming language, user-level unix, and the “user-side” of system calls.
Prerequisites. You must have satisfied the prerequisites.

If you have not, drop the class. (Seriously.)

- Re-takes
  - If you are re-taking, perhaps talk to me.

- Enrollment:
  Current:

<table>
<thead>
<tr>
<th></th>
<th>sec 03</th>
</tr>
</thead>
<tbody>
<tr>
<td>enrolled</td>
<td>35</td>
</tr>
<tr>
<td>target</td>
<td>35</td>
</tr>
<tr>
<td>waitlist</td>
<td>4</td>
</tr>
<tr>
<td>unique</td>
<td>4</td>
</tr>
</tbody>
</table>

We're gonna try something crazy and let the registration system do its thing. There are currently 19 open seats in other sections.


Just about everything of interest ends up here, including:

- The syllabus (that you should read)
- Notes
- Assignments (read these too. It helps.)
- Labs
- Solutions
- (Probably) grades


If you are unfamiliar with the UNIX system programming environment, you might also find the following helpful:


- Also: The Minix Programmer’s Manual (online—use the version on your machine)

Note: “cmd(num)” means cmd as defined in section num of the manual.

- Grading policy:

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Labs</td>
<td>15%</td>
</tr>
<tr>
<td>Assignments</td>
<td>25%</td>
</tr>
<tr>
<td>Midterm</td>
<td>25%</td>
</tr>
<tr>
<td>Final</td>
<td>35%</td>
</tr>
</tbody>
</table>
• Assignments: (Do Them)
There will be some. A variety of programming, problem set, and report writing, totaling 25% of the final grade.
I expect there will be 5–6 assignments over the quarter, making each one worth approximately 5%
I will publish weights, but reserve the right to change them.
Asgn1 is out already. GET YOUR ACCOUNTS NOW you can even use the lab time.
Deliverable: a program or a report.
Note on assignments:
  – Do them!
  – Try to break them
    * with real tests
    * on multiple platforms (e.g. 32- and 64-bit)
  – If I provide a test harness, run it. It may not be my entire grading rig, but it’s probably a subset. (And it might be.)
  – RTFA

• Labs (6–7):
Most weeks there will be lab exercises. These will be some combination of written problems and laboratory exercises.
Deliverable: usually a report, sometimes a program.
A note on the lab period.

• Policies regarding submission of work:
  – Late Policy
  – Submitting Work (paper, online (I'll check on this)) neatness.
  – Late days
    – “Good faith effort” required on all assignments. (Missing assignment reduces maximum possible grade.)

• Exams: a midterm and a final. Emphasis on problem solving.
The midterm is tentatively scheduled for may 8 (wed)

• Grading: Policy. 1-week deadline for regrades/errors.

• Class decorum
  – I encourage interaction in class. If you’re confused, so are other people.
  – Note
    * you all belong here.
    * Remember: of course you don’t know what you are doing. That’s why you’re in the class. Speak up.
– I will not penalize you for nonattendance, but you will lose sympathy in office hours. (Find out what happened)
– Please refrain from disruptive behavior.
– In particular, this quarter I am strongly encouraging no electronic devices in class. They’re just too distracting. (http://www.yorku.ca/ncepeda/laptopFAQ.html)

Laptops hinder classroom learning for both users and nearby peers

Faria Sana, Tina Weston, and Melody Wiseheart*

General Abstract: Laptops are commonly found in university classrooms. Thus, students will inevitably use laptop applications, such as games and social networking, during class time. To investigate whether multitasking on a laptop impedes in-class learning, we conducted two experiments in a simulated university lecture setting. We found that students who multitasked on a laptop during a lecture scored lower on a test compared to students who did not multitask, and students who were in direct view of a multitasking peer scored lower on a test compared to students who were not in view of a multitasking peer. The results suggest that multitasking on a laptop is a distraction to both users and fellow students and can be detrimental to learning of classroom materials. *formerly Nicholas J Cepeda

– I try to keep the two classes in sync, but I cannot guarantee it. I do ask students to attend their own sections for exams.

• Cheating: Don’t.
  – Integrity matters
  – Partnerships and proper collaboration.
    * DO: use each other for help
    * DO NOT: simply split up the work
  – Proper Collaboration
    * general principles/approaches
    * debugging consultation
  – Improper Collaboration
    * “Here’s my code”
    * specific solutions
  – Apply the “Competition” rule
  – Credit any work that comes from anyplace else. (Not doing your homework is better than getting caught cheating. Your name isn’t Tanenbaum, is it?)

The Pledge:

– Don’t use others’ work while doing your own
– Don’t make your work available. This includes taking reasonable precautions to prevent someone else’s taking it. E.g., not leaving your program lying around or leaving it online world readable.
It is *never* acceptable to allow someone else to have your source code for reference or to refer to someone else’s code while writing your own.

**Cheating requires an “F” course grade**

Not giving your work to others includes taking reasonable precautions to prevent others taking it.

### 1.4 How to succeed in this course

- Read assignments right away so they can be percolating.
- Do discuss problems with classmates and/or me.
- Give yourself time to be stuck.
- “I need insight now” is not effective.
- When I give test harnesses, use them:
  - tryLab01 (*don’t copy it*)
  - tryAsgn1 (*don’t copy it*)
- It sounds silly to say, but: **turn stuff in.** (You can’t win if you don’t play).
- Read the textbook. Tanenbaum is a good writer.

### 1.5 The Last Page

Please fill this out as it helps me to know who you are, where you’re coming from, and what you expect from the course. It also helps me gauge enrollment.

### 1.6 Foundations

#### 1.6.1 What an operating system is

**Virtual Machine** (Extended (abstract) machine) Simplified view of the system.

Operating Systems insulate users from the complexity of the underlying system and provide functionality that doesn’t exist.

(e.g. a filesystem: do you *really* care how a file is stored?)

(Do you want to know (or care) whether the floppy motor is on? The OS does. (and you do. You just want someone else to take care of it.))

**Resource Manager** Combination mediator and traffic cop.

(Consider printer spooling.)

Typically this is possible because only the OS has access to certain instructions. (kernel or supervisor mode)

This is just protecting from accidents, let alone malice.
The operating system is the one who *really knows what’s going on.* (and makes the magic happen)
That’s all well and good, but what *is* an operating system?

**A program.** That’s it. Rather, it’s the first program that allows others to run. The things above are what makes an OS *useful.*

The OS can be considered the host of the party: it’s the first to arrive, the last to leave (after cleaning up), and mostly it tries to stay out of the way and keep the party going, but steps in where necessary to resolve conflict.

### 1.6.2 Virtualization and Transparency

From a user’s perspective this process reduces to *virtualization* and *transparency*

**Transparent** It’s there, but you can’t see it.

**Virtual** you can see it, but it isn’t there.

### 1.6.3 What an operating system is not

- Compilers
- Editors
- Command interpreters
- Web Browsers(!)

<table>
<thead>
<tr>
<th>Compilers</th>
<th>Editors</th>
<th>Shells</th>
</tr>
</thead>
<tbody>
<tr>
<td>Operating System</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Machine Instructions</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Physical Hardware</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### 1.7 Major subjects for the quarter

Three areas I really want to develop this quarter:

1. OS Theory: How an we solve various problems well?
   - Concurrent programming (deadlock avoidance)
   - Page algorithms
   - Scheduling algorithms (fairness, starvation)
2. OS Implementation: How it’s really done (The lab)
3. Stark Raving Paranoia: The OS can’t stop, can it?

---

¹This answer would be considered correct, but not complete were I to ask the same question on a midterm. :-)

17
1.8 This quarter

What we’re looking at:

• How the OS does its magic
  General areas:
    – Processes and Concurrency
    – IO
    – Memory Management
    – Filesystems
    – Security?

• This class:
  – Why study OS?
  – Why Unix? — innards are exposed
  – Why Minix? — there aren’t too many innards
    * only 30,000 lines of code (vs. ≈9 million for Linux)

• Resources:
  – the CSL machines
2 Lecture: History and Definition of an O.S.

Outline:
Announcements
About the lab
About the assignment
Everything you wanted to know about C development
Aside: Review of 357
    Review of Unix IO
    Processes, etc.
    Compilation
    Tool of the week: Make(1)
Defining of an Operating System: The System Calls

2.1 Announcements

- lab location (14-255)
- Be reading along in the book. Tanenbaum and Woodhull are good writers.
- Coming attractions:

<table>
<thead>
<tr>
<th>Event</th>
<th>Subject</th>
<th>Due Date</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>asgn5</td>
<td>minget and minls</td>
<td>Wed Jun 5 23:59</td>
<td></td>
</tr>
<tr>
<td>asgn6</td>
<td>Yes, really</td>
<td>Fri Jun 7 23:59</td>
<td></td>
</tr>
<tr>
<td>final</td>
<td>stuff</td>
<td>Sat Jun 8 10:10</td>
<td>(in 03-201)</td>
</tr>
</tbody>
</table>

Use your own discretion with respect to timing/due dates.

- Late Days (how they work)

- Beware of malloc(): there are a kajillion implementations out there, but you need to do your own.

- If you’ve been through this before unsuccessfully, tell me. I can ask you awkward questions that’ll improve your odds of never doing 453 again.

- tryLab01
  - `~pn-cs453/bin/longlines.pl`

- tryAsgn1
  - (don’t copy it)
  - Run it on a 64-bit machine (e.g. unix1–4)
    (Make sure it’s one with 32-bit libraries; unix5 doesn’t have them for whatever reason.)
  - Consider the effects of architecture (how big is an int?) and uninitialized data

- Office hours
  - Come
  - Or talk to me during lab. I’m guaranteed to be available.

- “https://www.cs.vu.nl/~ast/intel/”
2.2 About the lab

Remember man name(section)

- pipeit (ls | sort -r > outfile)

What does this mean?
- There are three processes here
- You are the plumber/reaper
- This demonstrates the process abstraction that an OS provides. All communication and synchronization takes place through the OS.

2.3 About the assignment

- malloc(3): How does it work?
- Libraries: Two forms
  - static (libmalloc.a)
  - shared object (libmalloc.so)
    * LD_LIBRARY_PATH
    * LD_PRELOAD
- Note: the order of link commands matters to gcc
- Also Note: setting environment variables:
  
  ```
  [bash] VAR=value
  export VAR
  [t]csh setenv VAR value
  ```

2.4 Everything you wanted to know about C development

- The Environment
- Linking
- Loading
- Make
- gdb

2.5 Aside: Review of 357

2.5.1 Review of Unix IO

- file descriptors
- open(2) vs. fopen(3) (and permissions)
- dup(2) and dup2(int old, int new)
- pipes, how they work
2.5.2 Processes, etc.

- **Lifecycle**
  - Birth: `fork()`
  - Death: termination (`exit()`, `_exit()`, `return`, `abort()`, `signal`)
  - Afterlife: reaping with `wait()` or `waitpid()`
    - `wait(2)`
    - `waitpid(2)`
    - `WIFEXITED()` / `WIFSIGNALED()`
    - `WEXITSTATUS()`

2.5.3 Compilation

- The compiler (`gcc`, `ACK`, `clang`)
- The linker (`ld`, `gcc`)
  - `-L` `path`
  - `-l` `name`
  - `LD_LIBRARY_PATH`
  - `LD_PRELOAD`
- The loader (`ld.so`)
- Libraries
  - Static (*.a). Made with `ar(1)`
  - Dynamic (*.so, *.dll, *.dylib) Made with the compiler
- Some thoughts on Make

2.5.4 Tool of the week: Make(1)

Make is a program to control program builds automagically, but it can be much, much more.

- Based on dependencies—there’s no need to regenerate a file if its source hasn’t changed.
  A dependency looks like: `target: source`.
  A particular target can have multiple dependency lines
- Implicit Rules—Make knows how to do certain things (compile C source, for example: if a .o file depends on a .c file if there is one of the same name in the directory).
  You can define your own if you want.
- Explicit rules: After a tab, a series of instructions for making the thing:
  ```
  foo.o: foo.c
  gcc -c -Wall -ansi -pedantic foo.c
  ```
- It supports variables. In particular CC, SHELL, FLAGS.
  `$VARNAME` to evaluate. `$` for a dollar sign
• Remember TABS
  (Quite possibly the dumbest decision since creat()).

• In rules:

  | @ | do a line silently |
  | # | comment            |
  | - | proceed even in the face of errors |

• Interfaces nicely with RCS (will check out files for you.)

• To use: make thing
  if “thing” isn’t specified, it makes the first target.

• Emacs: M-x compile

  For more info, read the man page, or the info page.

2.6 Defining of an Operating System: The System Calls

Tanenbaum set out to write a UNIX... what does that mean?
  What defines an operating system from the users’ point of view?
  The system calls.
  Just as the instruction set architecture defines an architecture, the system calls define an Operating System.
  Before we talk about IO services, implementation, etc....
  Look like C functions, but are direct requests for OS services.
  A system call is an entry into the kernel.

<table>
<thead>
<tr>
<th></th>
<th>Linux: (RH7.0) 222</th>
</tr>
</thead>
<tbody>
<tr>
<td>Solaris:</td>
<td>253</td>
</tr>
<tr>
<td>Minix:</td>
<td>53</td>
</tr>
</tbody>
</table>
3 Lecture: The Process Model

Outline:
Announcements
Two stories
Defining of an Operating System: The System Calls
System Calls Again
System Call Mechanisms
  How to do it
OS Pre-history: The boot process
  How it all begins (on an Intel PC with a floppy)
  How it continues
  And onwards: How does the OS get control back?
OS History
  Ancient times
  The middle ages
  The renaissance: families
  Modern Times

3.1 Announcements

- Coming attractions:

<table>
<thead>
<tr>
<th>Event</th>
<th>Subject</th>
<th>Due Date</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>asgn5</td>
<td>minget and minls</td>
<td>Wed</td>
<td>Jun 5 23:59</td>
</tr>
<tr>
<td>asgn6</td>
<td>Yes, really</td>
<td>Fri</td>
<td>Jun 7 23:59</td>
</tr>
<tr>
<td>final</td>
<td>stuff</td>
<td>Sat</td>
<td>Jun 8 10:10 (in 03-201)</td>
</tr>
</tbody>
</table>

Use your own discretion with respect to timing/due dates.

- Remember:
  - Lab 1 due Today
  - Asgn 1 due Friday

- For Lab01
  - 26 done (out of 35(ish))
  - tryLab01
  - Remember how late days work: latedays

- For Asgn1:
  - don't call `sbrk(2)` for every `malloc(3)` call.
  (quilting?)
  - remember how pointer arithmetic works (in the size of the pointee)
  - `uintptr_t` from `<stdint.h>`
  - **Turn in one copy if working w/a partner.**
  - About that Makefile...
3.2 Two stories

Let’s look at it again:

- Startup (Booting)
- Establishing the OS
- Switching to a user-level program
- Recovering supervisor privilege

3.3 Defining of an Operating System: The System Calls

Tanenbaum set out to write a Unix... what does that mean?

What defines an operating system from the users’ point of view?

The system calls.

Just as the instruction set architecture defines an architecture, the system calls define an Operating System.

Before we talk about IO services, implementation, etc....

Look like C functions, but are direct requests for OS services.

A system call is an entry into the kernel.

<table>
<thead>
<tr>
<th>Platform</th>
<th>Calls</th>
</tr>
</thead>
<tbody>
<tr>
<td>Linux: (RH7.0)</td>
<td>222</td>
</tr>
<tr>
<td>Solaris:</td>
<td>253</td>
</tr>
<tr>
<td>Minix:</td>
<td>53</td>
</tr>
</tbody>
</table>

3.4 System Calls Again

We said “An OS is defined by its system calls”. What does that mean?

<table>
<thead>
<tr>
<th>System Call</th>
</tr>
</thead>
<tbody>
<tr>
<td>A system call is the means by which the kernel provides access to a particular operating system service, and the services available through system calls (e.g., reading and writing the disk, allocating memory, starting new processes, etc.) are reserved to the kernel; there is no other way of doing these things.</td>
</tr>
</tbody>
</table>

3.5 System Call Mechanisms

Last time we said

- It’s all about privilege.

3.5.1 How to do it

How done? Machine dependent. usually in assembly, but hidden from the users’ view.

In any case, some sort of trap is involved to get supervisor privileges.

A trap is a software generated interrupt. Exactly what happens varies from architecture to architecture, but the result usually involves:

- saving the state of the currently executing program
• elevating of processor privilege level to “supervisor”
• transfer of control to a pre-registered routine
• *The OS does something*
• Then, usually:
  – privilege level is reduced back to user
  – original state of executing program is restored

3.6 **OS Pre-history: The boot process**

3.6.1 **How it all begins (on an Intel PC with a floppy)**

• Power on
• Boot Sector of fd0: 512b\(^2\), loaded into 0x00007c00.
• jump to 0x00007c00 . . .

3.6.2 **How it continues**

• OS Sets up whatever it needs to set up, and, depending on what sort of system it is...
• (maybe) picks a program to run
• (maybe) shifts out of supervisor mode
• (maybe) runs the program

3.6.3 **And onwards: How does the OS get control back?**

How does the OS preempt a user process? It’s not running. It takes a series of carefully-planned steps

• The O.S. runs in supervisor mode which allows it to manipulate the interrupt vector *(definition)* and register an ISR.

• Before changing privilege levels, the O.S.
  – installs ISRs for a timer and also for the system call interrupt, and
  – requests a timer interrupt for some future time.

• The O.S. then changes privilege levels and yields to the user process.

• Eventually, one of two things happens:
  1. the timer interrupts, or
  2. the process makes a system call

---

\(^2\)1440k floppy has 2 sides, 80 tracks, 18 sectors. This makes 512 bytes per sector. (1474560/80)/18 → 18432/18 → 512
```c
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <stdarg.h>
#include <errno.h>

#define SYSCALL_OPEN 0x5

/* make sure this is legitimate */
#ifndef i386
#error "This can only be compiled for an x86"
#endif

int open(const char *pathname, int flags, ...) {
    mode_t mode;
    va_list ap;
    int res;

    /* use the variable arguments support */
    va_start(ap, flags);
    mode = va_arg(ap, mode_t);  /* extract the mode from the top of the stack */
    va_end(ap);

    /* load the given values into the registers and execute
     * the given syscall. The syscall code goes into eax
     * and return value comes from there.
     */

    asm (
        "movl %0,%%eax" : : "g" (SYSCALL_OPEN) : "eax");
    asm (
        "movl %0,%%ebx" : : "g" (pathname) : "ebx");
    asm (
        "movl %0,%%ecx" : : "g" (flags) : "ecx");
    asm (
        "movl %0,%%edx" : : "g" (mode) : "edx";
    asm (
        "int $0x80" : : "eax";
    asm ("movl %0,%%eax,%%eax,%%eax,%%eax : "g" (res) :);

    if (res < 0) {
        errno = -res;
        res = -1;
    }

    return res;
}
```

Figure 1: A Linux `open()` implementation
/* . /lib/posix/_open.c */
#include <lib.h>
#include <fcntl.h>
#include <stdarg.h>
#include <string.h>

PUBLIC int open(const char *name, int flags, ...)
{
    va_list argp;
    message m;

    va_start(argp, flags);
    if (flags & O_CREAT) {
        m.m1.i1 = strlen(name) + 1;
        m.m1.i2 = flags;
        m.m1.i3 = va_arg(argp, Mode_t);
        m.m1.p1 = (char *) name;
    } else {
        _loadname(name, &m);
        m.m3.i2 = flags;
    }
    va_end(argp);
    return (syscall(FS, OPEN, &m));
}

Figure 2: Minix open() implementation

/******* lib/other/syscall.c ****************************/
PUBLIC int _syscall(who, syscallnr, msgptr)
int who;
int syscallnr;
register message *msgptr;
{
    int status;

    msgptr->m_type = syscallnr;
    status = _sendrec(who, msgptr);
    if (status != 0) {    /* 'sendrec' itself failed. */
        msgptr->m_type = status;
    }
    if (msgptr->m_type < 0) {
        errno = -msgptr->m_type;
        return(-1);
    }
    return(msgptr->m_type);
}

Figure 3: Minix _syscall()
Figure 4: Minix sendrec() abstracted
Either one causes an interrupt.

- The ISR (installed by the OS and in write-protected memory) runs with supervisor privilege. Now O.S. code is running as superman again and can decide what to do.

Note: This is all about privilege. The OS gets supervisor privilege back when the interrupt handler executes. The user process cannot stop this because the user process does not have the authority to block interrupts or to change the handler.

3.7 OS History

The history of the development of operating systems parallels the history of computer architecture. (duh?)

3.7.1 Ancient times

Vacuum tubes/relays. No programming languages. Bug meant an actual insect. (aside about Dijkstra: (paraphrased) we should eliminate the use of the word bug and replace it with error.).

“Human operating systems” rewire the machines

- programmed by the builders.
- time blocked out for exclusive access
- machines rewired with plugboards
- Eventually evolved into card-fed computers.
  (library actually meant a filing cabinet)

3.7.2 The middle ages

Transistors made machines small and reliable. OS loaded a program, then replaced itself.

Batch processing invented (IBM 1401→7094→1401)

- jobs loaded onto a tape offline
- run in the main computer
- output printed offline
- operations controlled by JCL

3.7.3 The renaissance: families

Computer families developed. Different machines of different sizes. (e.g., IBM System/360 note Brooks’ experience)

operating systems very complex, written in assembly, tied to a particular processor (family).

Innovations:

- multiprogramming (but still essentially batch systems)
- timesharing (MULTICS: Multiplexed Information and Computing Service)
• the rise of the minicomputer! (PDP-1 1961, $120,000\textsuperscript{3} vs. $2.4M\textsuperscript{4} for a 7094).
• a neglected PDP-7 and Unix: (1969)
  – History
    * Space Travel (Thompson and Ritchie)
    * Small system
  – features
    * Small system
    * modular
    * portable (Written in C!)
      · BCPL (untyped (all datatypes are bitfields))
      · B (untyped)
      · C (weakly typed)
      · Ansi C (more strongly typed) – 1989

3.7.4 Modern Times
In modern days, the rise of the PC parallels the rise of the minicomputer:
• Windows/unix (Tanenbaum vs. Torvalds?)
  The family tree:
  – AT&T
  – BSD (Berkeley Software Distributions)
  – Linux?
  – POSIX 1988
• Network operating systems. Distributed operating systems.
  one image comprises many machines. (high availability?)

Minix vs. Linux? A matter of Philosophy
• Minix
  – Designed for readability
  – Modularity
  – Message passing
  – extensibility
• linux
  – Efficiency
  – “Production” system

\textsuperscript{3}According to the CPI, this is $968,640.80 in 2016
\textsuperscript{4}$19,372,816.05 in 2016
4 Lecture: Operating System Structures

Outline:

Announcements

Two stories

OS Pre-history: The boot process
  How it all begins (on an Intel PC with a floppy)
  How it continues
  And onwards: How does the OS get control back?

From last time: History
  The middle ages
  The renaissance: families
  Modern Times

From last time: The Process (Users’ View)
The Operating System’s view: The context
Example of a context switch
System Calls Again
System Call Mechanisms
  How to do it
  And onwards

4.1 Announcements

- Coming attractions:
<table>
<thead>
<tr>
<th>Event</th>
<th>Subject</th>
<th>Due Date</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>asgn5</td>
<td>minget and minls</td>
<td>Wed Jun 5</td>
<td>23:59</td>
</tr>
<tr>
<td>asgn6</td>
<td>Yes, really</td>
<td>Fri Jun 7</td>
<td>23:59</td>
</tr>
<tr>
<td>final</td>
<td>stuff</td>
<td>Sat Jun 8</td>
<td>10:10</td>
</tr>
</tbody>
</table>

Use your own discretion with respect to timing/due dates.

- ~/nico/longlines.pl!

- lab grading scheme

- 39 submitted (out of 37(ish))

- Lab02: Why minix 3.1.8

- tryAsgn1
  - How to do alignment? Arithmetic: stdint.h and, esp., uintptr_t
  - free(NULL) called by snprintf(3). Watch for overflows
  - (also, -m32 and intel-all)
  - How to use late days
  - Don’t leave things until the last minute
  - Turn in one copy if working w/a partner.
  - Compiling on 32- vs. 64-bit architectures.
  - (and printf(3), etc, might use malloc(2). Consider puts(3) and/or write(2).)

- Asgn1 due Wednesday (reminder about late days)
4.2 From last time: History

4.2.1 The middle ages

Transistors made machines small and reliable. OS loaded a program, then replaced itself.

Batch processing invented (IBM 1401→7094→1401)

- jobs loaded onto a tape offline
- run in the main computer
- output printed offline
- operations controlled by JCL

4.2.2 The renaissance: families

Computer families developed. Different machines of different sizes. (e.g., IBM System/360 note Brooks’ experience)

- operating systems very complex, written in assembly, tied to a particular processor (family).
- Innovations:
  - multiprogramming (but still essentially batch systems)
  - timesharing (MULTICS: Multiplexed Information and Computing Service)
  - the rise of the minicomputer! (PDP-1 1961, $120,000\textsuperscript{5} vs. $2.4M\textsuperscript{6} for a 7094).
  - a neglected PDP-7 and Unix: (1969)
    - History
      * Space Travel (Thompson and Ritchie)
      * Small system
    - features
      * Small system
      * modular
      * portable (Written in C!)
        - BCPL (untyped (all datatypes are bitfields))
        - B (untyped)
        - C (weakly typed)
        - Ansi C (more strongly typed) – 1989

\textsuperscript{5}According to the CPI, this is $968,640.80 in 2016
\textsuperscript{6}$19,372,816.05 in 2016
4.2.3 Modern Times

In modern days, the rise of the PC parallels the rise of the minicomputer:

- Windows/unix (Tanenbaum vs. Torvalds?)
  The family tree:
  - AT&T
  - BSD (Berkeley Software Distributions)
  - Linux?
  - POSIX 1988

- Network operating systems. Distributed operating systems.
  one image comprises many machines. (high availability?)

Minix vs. Linux? A matter of Philosophy

- Minix
  - Designed for readability
  - Modularity
  - Message passing
  - extensibility

- linux
  - Efficiency
  - "Production" system

4.3 From last time: The Process (Users’ View)

A quick overview of the (human) users’ view of the system:
Each running program becomes a process, isolated from all other processes:

- Each process has the illusion of being alone
- Has its own memory
- Has its own scheduling time
- Has its own interface to the outside world (file descriptors)
- All interprocess (and outside world) interaction takes place through the operating system.

Processes have identity:

- User ID
- Group ID
- Process ID

Processes have resources:

- memory (address space)
- time
4.4 The Operating System’s view: The context

In the OS’s view, a process consists of some resources:

<table>
<thead>
<tr>
<th>Resource</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>registers</td>
<td>Each process gets its own copy.</td>
</tr>
<tr>
<td>address space</td>
<td>A region of memory, usually starting at address 0, corresponding to a particular process.</td>
</tr>
<tr>
<td>identity</td>
<td>uid, gid, pid, ppid, etc. All those things that determine a process’s identity and privilege.</td>
</tr>
<tr>
<td>file descriptors</td>
<td>Connections to the global file descriptor table to allow for IO</td>
</tr>
<tr>
<td>signals/masks</td>
<td>Pending notifications</td>
</tr>
</tbody>
</table>

The OS’s purpose is to keep these separate. How?

- Records kept in a **process table**
- Processes run for an allotted time *(quantum)*, or until they yield (e.g. disk wait)
- **context switch**
  - Old process suspended (timer?)
  - Old process’s registers saved (where?)
  - Old process’s memory—text, stack, data—saved in a **core image**
  - Memory set up for new process (text, stack, data)
  - Registers set up for new process
  - New process “continued” as if nothing had happened.
    Imagining blinking and discovering that it was three hours later.

4.5 Example of a context switch

How does it really happen? (with pictures, and everything.)

- Process A is running (Figure 4.5a)
- An interrupt occurs (Figure 4.5b)
- Push registers to preserve them. (Figure 4.5c)
- Save SP in the process table and switch to the operating system (kernel) stack (Figure 4.5d)
- The OS is now running:
  - Preserve Process A’s memory (if necessary)
  - Choose the next process to run.
  - Load Process B’s memory (if necessary)
- Restore B’s SP from process table. (Figure 4.5e)
- Pop B’s registers (Figure 4.5f)
- Return from interrupt (Figure 4.5g)
(a) While process A is running

(b) After the interrupt

(c) Saving A’s registers

(d) switch to the operating system (kernel) stack

(e) Restore B’s SP from process table.

(f) Pop B’s registers

(g) After return from interrupt

Figure 5: The process of a context switch
4.6 System Calls Again

We said “An OS is defined by its system calls”. What does that mean?

<table>
<thead>
<tr>
<th>System Call</th>
</tr>
</thead>
<tbody>
<tr>
<td>A system call is the means by which the kernel provides access to a particular operating system service, and the services available through system calls (e.g., reading and writing the disk, allocating memory, starting new processes, etc.) are reserved to the kernel; there is no other way of doing these things.</td>
</tr>
</tbody>
</table>

4.7 System Call Mechanisms

Last time we said

- It’s all about privilege.

4.7.1 How to do it

How done? Machine dependent, usually in assembly, but hidden from the users’ view.

In any case, some sort of trap is involved to get supervisor privileges.

A *trap* is a software generated interrupt. Exactly what happens varies from architecture to architecture, but the result usually involves:

- saving the state of the currently executing program
- elevating of processor privilege level to “supervisor”
- transfer of control to a pre-registered routine
- *The OS does something*
- Then, usually:
  - privilege level is reduced back to user
  - original state of executing program is restored

4.7.2 And onwards

- How does the OS get control back?
5 Lecture: Intro. to Concurrency

Outline:

Announcements
Onwards
The Process Model: a little deeper
Pseudoparallelism and nondeterminism
Possible process states
Scheduling
What about IPC?
Operating System Structures
  Monolithic Systems
  Layered Systems
  Virtual Machines (VM/370, vmware, bochs)
  Client-server model
The Layered Architecture of Minix
Example: Description a MINIX disk interrupt

5.1 Announcements

• Coming attractions:

<table>
<thead>
<tr>
<th>Event</th>
<th>Subject</th>
<th>Due Date</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>asgn5</td>
<td>minget and minls</td>
<td>Wed Jun 5 23:59</td>
<td></td>
</tr>
<tr>
<td>asgn6</td>
<td>Yes, really</td>
<td>Fri Jun 7 23:59</td>
<td></td>
</tr>
<tr>
<td>final</td>
<td>stuff</td>
<td>Sat Jun 8 10:10</td>
<td>(in 03-201)</td>
</tr>
</tbody>
</table>

Use your own discretion with respect to timing/due dates.

• `free(NULL)` called by `snprintf(3)`. Watch for overflows
• Make troubles?
• Asgn1 due Today
• `mkfs.mfs` on 3.1.8 (Read the durned footnote)

5.2 Onwards

Last time we talked about context switching and the concept of a Process. Now that we have the general idea, let’s look at some general structures.

This week we are going to look at general OS architectures, then move into scheduling.

5.3 The Process Model: a little deeper

Now, we need to talk about some details of how it’s actually done and the issues involved.

The process is the most important concept in understanding operating systems.

The operating system provides **multiprogramming**:

**pseudoparallelism** time-sliced parallelism on a uniprocessor

(yield?, pre-empt?)
true parallelism provided by a multiprocessor

Keeping track of parallel activities is difficult, so OS designers have developed the model of the sequential process:

Each process (running program) has its own:

• IO Access: open file descriptors
• Register file
• Memory
  – Text (code) segment
  – Data (bss and heap) segment
  – Stack segment
• Program Counter

5.4 Pseudoparallelism and nondeterminism

(Ir?)regular (random) context switches mean programs can make no assumptions about:

• order of execution relative to other programs
• timing (example: a clock, or game timing)

(If timing is important, you need to get a Real-Time O.S.)

5.5 Possible process states

Not all processes can run at the same time:

At any given time, any process can be in one of three possible states: Running, Ready, or Blocked. Possible transitions between these states are shown in Figure 6.

![Figure 6: Possible states for a process](image)

In more detail, a process can be:

**Running** The process is loaded in memory and is currently executing on the cpu.

**Ready** The process is runnable, but another process has the cpu.

**Blocked** The process is waiting for some external event to enable it to run. E.g. completion of some IO event, or an alarm, or availability of more memory.
5.6 Scheduling

Transitions among the states of Figure 6 are made by the scheduler. The scheduler hides the details of starting and stopping processes:

- handles interrupts and dispatches them to the appropriate process
- chooses a process to run from among runnable processes
- Maintains the process table that contains
  - program counter
  - stack pointer
  - memory allocation information
  - status
  - open file descriptors
  - other stuff that must be saved (more record keeping: execution times, signal mask, etc.)

Minix divides its process table into separate ones for scheduling, memory, and file information, but the principle is the same.

This leads to the model of the operating system shown in Figure 7.

\[
\begin{array}{cccc}
0 & 1 & \ldots & n-2 & n-1 \\
\hline
\text{Scheduler} & & & & \\
\end{array}
\]

Figure 7: The lowest layer of the operating system

5.7 What about IPC?

Now we have separated processes so they don’t know about each other, how do they communicate? IPC must be provided by the O.S.

5.8 Operating System Structures

Now that we’ve looked at the “external” view of an operating system, how would one be built?

\[\text{Guts are so inelegant}\]
\[\text{—Patrick Beard}\]

Operating systems have to do a lot of things:

- Scheduling
- Interprocess Communication
- Memory Management
- Filesystem Management
• Device Management
• Network Management
• *Lots of other things...*

How to put one of these together?

### 5.8.1 Monolithic Systems

Provide a set of system services through a single interface (a trap handler (Supervisor Call)):

• The system is one big object
• no information hiding
• requires programmer discipline: difficult to maintain

Can be organized well, but be careful...

• Structure:
  - a main() program invoked by the trap
  - a set of service routines that carry out system calls
  - a set of utility routines

Dispatched through a call table. E.g. `open(name,flags,mode)`

```latex
\begin{verbatim}
movl $0xXXXXXXXX,%ebx ; String pointer (filename)
movl $0x2,%ecx ; flags (O_RDONLY)
movl $0x0,%edx ; mode (0)
movl $0xb,%eax ; syscall code for open()
int $0x80 ; do the call
\end{verbatim}
```

### 5.8.2 Layered Systems

Generalizes the approach above, separating tasks.

First exhibited in the THE operating system.

Technische Hogdschool [University] Eindhoven in the Netherlands, by Dijkstra in 1968

The machine: Electrologica X8, with 32K of 27-bit words (400kHz).

The OS Six layers:

<table>
<thead>
<tr>
<th>Layer</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>The Operator</td>
</tr>
<tr>
<td>4</td>
<td>User Programs</td>
</tr>
<tr>
<td>3</td>
<td>IO Management (abstract IO devices)</td>
</tr>
<tr>
<td>2</td>
<td>Operator-process communication (individual operator consoles)</td>
</tr>
<tr>
<td>1</td>
<td>Memory and drum (512K words) management</td>
</tr>
<tr>
<td>0</td>
<td>Processor allocation and multiprogramming</td>
</tr>
</tbody>
</table>

Level 3 is above level 2 because the devices might need to speak to the operator in case of malfunction.

Multics did this with concentric rings.

How did THE enforce these layers? It didn’t, but it was a good design tool.
5.8.3 Virtual Machines

(VM/370, vmware, bochs)

Go all the way: If we’re going to virtualize this machine, let’s do it:

- **vm/370**: Virtual Machine Monitor
- **Different users can run different OSs**
  - CMS: Conversational Monitor System (interactive)
  - OS/360: batch processing
  - vm/370 in itself...
- **Used by windows to run MS-DOS programs. (full emulation vs. partial emulation)**
- **VMare does the same thing completely (as do bochs and qemu)**

5.8.4 Client-server model

Mach/Minix

Pull it all out of the kernel. (Well, not all)

Servers and clients communicate via message passing.

The (now) micro-kernel does only things really requiring privilege and delegates to user-level processes.

- Separates **mechanism** from **policy**
- message-passing
- modifiable
- could be remote(?)

E.g.: Mach

5.9 The Layered Architecture of Minix

The structure of minix is as in Figure 8.

<table>
<thead>
<tr>
<th>Layer</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>4 init proc A proc B proc C ... User Processes</td>
</tr>
<tr>
<td>3</td>
<td>2 Disk Tty Ethernet Memory ... I/O Tasks</td>
</tr>
<tr>
<td>4</td>
<td>1 scheduling, interrupts, communication Clock Task System Task Process Management</td>
</tr>
</tbody>
</table>

**Figure 8**: Layout of the Minix system

**Layer 1** Sheduling, handling traps and interrupts, facilitating message passing.

The bottom is written in assembly (must be), the rest is C.

**Layer 2** IO Processes, one per device type. Compiled into the kernel, but have separate identities, (and, if supported, privilege levels)
Layer 3  Server processes: File System, Memory Manager, Network Server, etc. These are user-level processes that implement the system calls. These run with higher priority than user processes and never terminate while the system is running.

Layer 4  User processes

5.10  Example: Description a MINIX disk interrupt

- Structure of MINIX
- How a disk read happens in MINIX
6 Lecture: Lightweight Processes

Outline:

Announcements
From last time: Client-server model
The Layered Architecture of Minix
Example: Description a MINIX disk interrupt
Once again: Processes
Lightweight Processes: Threads
Introduction to Asgn2
Nine little functions
*Scheduling
Demonstration: threading in action
Review: context switch
A thread’s context: stack and registers
*Stack structure: The gcc calling convention
What’s this about libraries, then?
Race condition, defined
Critical Sections
Mutual Exclusion
Busy Waiting: Software Only
From last time: Review of Busy Waiting
Peterson’s Solution

6.1 Announcements

• Coming attractions:

<table>
<thead>
<tr>
<th>Event</th>
<th>Subject</th>
<th>Due Date</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>asgn5</td>
<td>minget and minls</td>
<td>Wed Jun 5</td>
<td>23:59</td>
</tr>
<tr>
<td>asgn6</td>
<td>Yes, really</td>
<td>Fri Jun 7</td>
<td>23:59</td>
</tr>
<tr>
<td>final</td>
<td>stuff</td>
<td>Sat Jun 8</td>
<td>10:10 (in 03-201)</td>
</tr>
</tbody>
</table>

Use your own discretion with respect to timing/due dates.

• Path to tests can be discovered \((n/2 + 1)\)

• See gradesheet snapshot

• “remember” function pointers

\[
\text{typedef void (*lwpfun)(void *);} /* type for lwp function */\]

Last time we wound up talking about virtual machines. These are great if you have a piece of software that must run with privilege but which you don’t trust. In general, though, we like features.
6.1.1 From last time: Client-server model

Mach/Minix

Pull it all out of the kernel. (Well, not all)
servers and clients communicate via message passing.
The (now) micro-kernel does only things really requiring privilege and delegates to user-level processes.

- Separates mechanism from policy
- message-passing
- modifiable
- could be remote(?)

E.g.: Mach

6.2 The Layered Architecture of Minix

The structure of minix is as in Figure 9.

<table>
<thead>
<tr>
<th>Layer</th>
<th>Processes</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>init</td>
</tr>
<tr>
<td>2</td>
<td>Disk</td>
</tr>
<tr>
<td>1</td>
<td>scheduling, interrupts, communication</td>
</tr>
</tbody>
</table>

Figure 9: Layout of the Minix system

Layer 1 Shedingling, handling traps and interrupts, facilitating message passing.
The bottom is written in assembly (must be), the rest is C.

Layer 2 I/O Processes, one per device type. Compiled into the kernel, but have separate identities, (and, if supported, privilege levels)

Layer 3 Server processes: File System, Memory Manager, Network Server, etc. These are user-level processes that implement the system calls.
These run with higher priority than user processes and never terminate while the system is running.

Layer 4 User processes

6.3 Example: Description a MINIX disk interrupt

- Structure of MINIX
- How a disk read happens in MINIX
6.4 Once again: Processes

Recall (those of you who were awake) the process of a context switch:

- Old process suspended (timer?)
- Old process’s registers saved (where?)
- Old process’s memory—text, stack, data—saved in a core image
- Memory set up for new process (text, stack, data)
- Registers set up for new process
- New process “continued” as if nothing had happened.
  Imaging blinking and discovering that it was three hours later.

6.5 Lightweight Processes: Threads

Traditional Operating Systems only support processes as above.
  Sometimes we don’t need that kind of isolation.
  Other times we don’t want that kind of isolation.
  Sometimes it’s necessary (or at least desirable) to have multiple independent threads of control
  in a single address space: filesystem cache, web browser, etc.
  Types of threads:

user-level all at the user level (advantages/disadvantages)

  POSIX P-threads, Mach C-Threads.
  - lightweight (very)
  - requires no kernel support. (if a thread blocks, the whole process blocks.)
  - has no kernel support. (relies on good behavior—the reason we have a kernel in the first place)

kernel-level With kernel support level (advantages/disadvantages)

  - More robust
  - More expensive (kernel must be aware of threads to schedule them, resulting in a “real”
    context switch.)

Unfortunately, threads changes the programming model:

- Break the model of sequential process. no more illusion of sequentiality.
- Hard to retrofit: hidden assumptions (jet engine on a car?)
- Semantic questions:
  - What about errno?
  - Signal delivery? alarms?
  - fork(): does the child have all running threads?
– fork(): what about blocked threads? (couldn’t happen w/o threads)
– Stack management: how to detect stack overflow and fix it.

• issues.
  – non-preemptive
  – preemptive

6.6 Introduction to Asgn2

6.6.1 Nine little functions

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>lwp_create(function, argument, stacksize)</td>
<td>create a new LWP</td>
</tr>
<tr>
<td>lwp_gettid()</td>
<td>return thread ID of the calling LWP</td>
</tr>
<tr>
<td>lwp_exit()</td>
<td>terminates the calling LWP</td>
</tr>
<tr>
<td>lwp_yield()</td>
<td>yield the CPU to another LWP</td>
</tr>
<tr>
<td>lwp_start()</td>
<td>start the LWP system</td>
</tr>
<tr>
<td>lwp_stop()</td>
<td>stop the LWP system</td>
</tr>
<tr>
<td>lwp_set_scheduler(scheduler)</td>
<td>install a new scheduling function</td>
</tr>
<tr>
<td>lwp_get_scheduler(scheduler)</td>
<td>find out what the current scheduler is</td>
</tr>
<tr>
<td>tid2thread(tid)</td>
<td>map a thread ID to a context</td>
</tr>
</tbody>
</table>

Scheduling

The lwp scheduler is a structure that holds pointers to five functions. These are:

void init(void) This is to be called before any threads are admitted to the scheduler. It’s to allow the scheduler to set up. This one is allowed, to be NULL, so don’t call it if it is.

void shutdown(void) This is to be called when the lwp library is done with a scheduler to allow it to clean up. This, too, is allowed, to be NULL, so don’t call it if it is.

void admit(thread new) Add the passed context to the scheduler’s scheduling pool.

void remove(thread victim) Remove the passed context from the scheduler’s scheduling pool and from the global thread list.

thread next(void) Return the next thread to be run or NO_THREAD if there isn’t one. You’ll probably have to write a helper function along the lines of tid2context() to get the actual context pointer.

Changing schedulers will involve initializing the new one, pulling out all the threads from the old one (using next() and remove()) and admitting them to the new one (with admit()), then shutting down the old scheduler.

6.6.2 Demonstration: threading in action

• nums
• randomsnakes
• hungrysnesakes
• nums with different scheduling algorithms
  -z choose the process in slot 0
• hungrysnares with different scheduling algorithms
  -z choose the process in slot 0
  -h choose the process that has eaten the most
  -l choose the process that has eaten the least

6.6.3 Review: context switch
• Old process suspended (timer?)
• Old process’s registers saved (where?)
• Old process’s memory—text, stack, data—saved in a core image
• Memory set up for new process (text, stack, data)
• Registers set up for new process
• New process “continued” as if nothing had happened.

6.6.4 A thread’s context: stack and registers

Registers The register structure of the 64-bit Intel x86_64 is shown in Table 1.

<table>
<thead>
<tr>
<th>Register</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>rax</td>
<td>General Purpose A</td>
</tr>
<tr>
<td>rbx</td>
<td>General Purpose B</td>
</tr>
<tr>
<td>rcx</td>
<td>General Purpose C</td>
</tr>
<tr>
<td>rdx</td>
<td>General Purpose D</td>
</tr>
<tr>
<td>rsi</td>
<td>Source Index</td>
</tr>
<tr>
<td>rdi</td>
<td>Destination Index</td>
</tr>
<tr>
<td>rbp</td>
<td>Base Pointer</td>
</tr>
<tr>
<td>rsp</td>
<td>Stack Pointer</td>
</tr>
<tr>
<td>r8</td>
<td>General Purpose 8</td>
</tr>
<tr>
<td>r9</td>
<td>General Purpose 9</td>
</tr>
<tr>
<td>r10</td>
<td>General Purpose 10</td>
</tr>
<tr>
<td>r11</td>
<td>General Purpose 11</td>
</tr>
<tr>
<td>r12</td>
<td>General Purpose 12</td>
</tr>
<tr>
<td>r13</td>
<td>General Purpose 13</td>
</tr>
<tr>
<td>r14</td>
<td>General Purpose 14</td>
</tr>
<tr>
<td>r15</td>
<td>General Purpose 15</td>
</tr>
</tbody>
</table>

Table 1: Integer registers of the x86 CPU
Stack structure: The gcc calling convention

The extra registers available to the x86_64 allow it to pass some parameters in registers. This makes the overall calling convention a little more complicated, but, in practice, it will be easier for your program since you won’t be passing enough parameters to push you out of the registers onto the stack.

The steps of the convention are as follows (illustrated in Figures 10a–f):

a. **Before the call** Caller places the first six integer arguments into registers %rdi, %rsi, %rdx, %rcx, %r8, and %r9. If there are more, they are pushed onto the stack in reverse order. This is shown in the figure, but you won’t encounter more in this assignment.

b. **After the call** The call instruction has pushed the return address onto the stack.

c. **Before the function body** If the function has more parameters and local variables than will fit into the registers it will execute the following two instructions to set up its frame:

   ```
   pushq %rbp
   movq %rsp,%rbp
   ```

   Then, it adjusts the stack pointer to leave room for any locals it may need.

   Your functions probably will not need to set up a frame.

d. **Before the return** If the function has set up a call frame in the previous step it needs to clean up after itself. To do this, before returning it executes a **leave** instruction. This instruction is equivalent to:

   ```
   movq %rbp,%rsp
   popq %rbp
   ```

   The effect is to rewind the stack back to its state right after the call.

   If the function did not set up a frame, its %rsp register is still pointing to the return address.

e. **After the return** After the return, the Return address has been popped off the stack, leaving it looking just like it did before the call.

   Remember, the ret instruction, while called “return”, really means “pop the top of the stack into the program counter.”

f. **After the cleanup** Finally, the caller pops off any parameters on the stack and leaves the stack is just like it was before.
Figure 10: Stack development (Remember that the real stack is upside-down)
Example (real gcc-generated code):

```c
long dummy(long x) {
    long tmp;
    tmp = x;
    return tmp;
}
```

```c
int main() {
    dummy(5);
}
```

<table>
<thead>
<tr>
<th>dummy:</th>
<th>long dummy(long x) {</th>
</tr>
</thead>
<tbody>
<tr>
<td>pushq %rbp</td>
<td>long tmp;</td>
</tr>
<tr>
<td>movq %rsp, %rbp</td>
<td>tmp = x;</td>
</tr>
<tr>
<td>movq %rdi, -24(%rbp)</td>
<td>return tmp;</td>
</tr>
<tr>
<td>movq -24(%rbp), %rax</td>
<td></td>
</tr>
<tr>
<td>movq %rax, -8(%rbp)</td>
<td></td>
</tr>
<tr>
<td>leave</td>
<td></td>
</tr>
<tr>
<td>ret</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>main:</th>
<th>int main(){}</th>
</tr>
</thead>
<tbody>
<tr>
<td>pushq %rbp</td>
<td></td>
</tr>
<tr>
<td>movq %rsp, %rbp</td>
<td></td>
</tr>
<tr>
<td>→ movl $5, %edi</td>
<td>dummy dummy()</td>
</tr>
<tr>
<td>call dummy</td>
<td># do call</td>
</tr>
<tr>
<td>leaveq</td>
<td></td>
</tr>
<tr>
<td>retq</td>
<td></td>
</tr>
</tbody>
</table>

The context type:

```c
typedef struct threadinfo_st *thread;
typedef struct threadinfo_st {
    tid_t tid;        /* lightweight process id */
    unsigned long *stack;  /* Base of allocated stack */
    size_t stacksize;     /* Size of allocated stack */
    rfile state;          /* saved registers */
    thread lib_one;       /* Two pointers reserved */
    thread lib_two;       /* for use by the library */
    thread sched_one;     /* Two more for */
    thread sched_two;     /* schedulers to use */
} context;
```

The scheduler type:

```c
/* Tuple that describes a scheduler */
typedef struct scheduler {
    void (*init)(void);  /* initialize any structures */
    void (*shutdown)(void);  /* tear down any structures */
    void (*admit)(thread new);  /* add a thread to the pool */
    void (*remove)(thread victim);  /* remove a thread from the pool */
    thread (*next)();  /* select a thread to schedule */
} *scheduler;
```
The lwp functions:

```
lwp_create(function, argument, stacksize)  create a new LWP
lwp_gettid()                                 return thread ID of the calling LWP
lwp_exit()                                   terminates the calling LWP
lwp_yield()                                  yield the CPU to another LWP
lwp_start()                                  start the LWP system
lwp_stop()                                   stop the LWP system
lwp_set_scheduler(scheduler)                 install a new scheduling function
lwp_get_scheduler(scheduler)                 find out what the current scheduler is
tid2thread(tid)                              map a thread ID to a context
```

Tools for stack manipulation:

Macros for stack manipulation: Remember, these must be called first and last.

```
GetSP(var)  Sets the given variable to the current value of the stack pointer.
SetSP(var)  Sets the stack pointer to the current value of the given variable.
```

```c
void swap_rfiles(rfile *old, rfile *new). This does two things:

1. if old != NULL it saves the current values of all 16 registers to the struct registers pointed to by old.
2. if new != NULL it loads the 16 register values contained in the struct registers pointed to by new into the registers.
```

General Notes:

- Remember to allocate enough space for your stacks
- Remember stacks are upside down
- Draw pictures!
- Don’t even think of turning the optimizer on!
- “Remember” gcc -S
- “remember” function pointers

```
typedef void (*lwpfun)(void *); /* type for lwp function */
```

- be careful of locals when climbing from one stack to another
- Careful to keep someplace to stand when free()ing de-allocated stack (you can use dummy functions in-between to be sure)
- what happens if your function returns? (“push” a frame that’ll return to lwp_exit())

6.7 What’s this about libraries, then?
7 Lecture: Concurrency and Synchronization

Outline:
- Announcements
- LWP
- Problems with parallelism: Race Conditions
  - Race condition, defined
- Critical Sections
- Mutual Exclusion
- Busy Waiting: Software Only
- From last time: Review of Busy Waiting
- Peterson’s Solution

7.1 Announcements

- Coming attractions:

<table>
<thead>
<tr>
<th>Event</th>
<th>Subject</th>
<th>Due Date</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>asgn5</td>
<td>minget and minls</td>
<td>Wed</td>
<td>23:59</td>
</tr>
<tr>
<td>asgn6</td>
<td>Yes, really</td>
<td>Fri</td>
<td>23:59</td>
</tr>
<tr>
<td>final</td>
<td>stuff</td>
<td>Sat</td>
<td>10:10 (in 03-201)</td>
</tr>
</tbody>
</table>

Use your own discretion with respect to timing/due dates.

- Warning about stack overflow/etc. E.g. WNOHANG or WUNTRACED. RTFM
- “remember” function pointers

```c
typedef void (*lwpfun)(void *); /* type for lwp function */
```

- Lab02: Choosing an emulator: Pick one that works for you.
- Be sure virtualization is turned on.

7.2 LWP

- Note that the scheduler is totally separable
- Pointers already provided
- How the scheduler works

The lwp scheduler is a structure that holds pointers to five functions. These are:

- **void init(void)** This is to be called before any threads are admitted to the scheduler. It’s to allow the scheduler to set up. This one is allowed, to be NULL, so don’t call it if it is.
- **void shutdown(void)** This is to be called when the lwp library is done with a scheduler to allow it to clean up. This, too, is allowed, to be NULL, so don’t call it if it is.
- **void admit(thread new)** Add the passed context to the scheduler’s scheduling pool and also adds it to the global `lwp.tlist`.
- **void remove(thread victim)** Remove the passed context from the scheduler’s scheduling pool and from the global thread list.
thread next(void)  Return the next thread to be run or NO_THREAD if there isn’t one. You’ll probably have to write a helper function along the lines of tid2context() to get the actual context pointer.

Changing schedulers will involve initializing the new one, pulling out all the threads from the old one (using next() and remove()) and admitting them to the new one (with admit()), then shutting down the old scheduler.

7.3  Problems with parallelism: Race Conditions

As soon as there any shared resources (even the filesystem), there can be problems.

Example: Print spooler

Consider a print spooler where files to be printed are placed in spots in an array indexed by qhead. The process to add an element looks something like: array[qhead++]=name. This breaks into three stages: read, update, write.

Consider the effect of the following interleaving.

<table>
<thead>
<tr>
<th>Process A</th>
<th>Process B</th>
</tr>
</thead>
<tbody>
<tr>
<td>(1) Read qhead</td>
<td>(2) Read qhead</td>
</tr>
<tr>
<td>(4) Update array</td>
<td>(3) Update array</td>
</tr>
<tr>
<td>(6) Increment index</td>
<td>(5) Increment Index</td>
</tr>
<tr>
<td>(7) Write qhead</td>
<td>(8) Write qhead</td>
</tr>
</tbody>
</table>

Not so good.

7.3.1 Race condition, defined

A race condition is any situation where the precise ordering of a series events affects the (correctness of the) outcome of the entire process.

The term is usually only applied where processes are reading or writing some shared data and where correctness is at stake.

Race conditions manifest themselves as nondeterministic behavior (usually leading to strange behavior at inopportune times).

7.4 Critical Sections

Critical sections require mutual exclusion.

For good solution we wish to maintain the following four principles:

1. No two processes may simultaneously enter the critical region.
2. No assumptions may be made about CPU speed of the number of CPUs
3. No process running outside of its critical section may run while another process is in its critical section.
4. No process should have to wait forever in its critical section.
7.5 Mutual Exclusion

How can we ensure mutual exclusion?

7.6 Busy Waiting: Software Only

For busy waiting to work, all contention must be among time-sliced processes or with true concurrency. (Why)

These solutions do not rely on hardware support to work correctly.

Options:

- Eliminate concurrency: That’ll fix it.
- Hope for the best?
- Disabling Interrupts
  
  **Advantages**: Foolproof?
  **Disadvantages**: Gives user (fool?) too much power. Blocks everything. (Decapitation will cure a headache.)

What about a multiprocessor?

This is used in the kernel.

- Lock Variables (Software Only)

```c
while ( TRUE ) {
    while ( lock )
        /* twiddle */;
    lock = 1;
    critical_things();
    lock = 0;
    noncritical_things();
}
```

**Advantages**: Allows finer-grained synchronization

**Disadvantages**: Doesn’t work (race-conditions)

- Strict Alternation: use a variable to say whose turn it is, have each thread set it explicitly on the way out

  **Advantages**: Works.
  **Disadvantages**: Significantly reduces parallelism. What if a process forgets? What if a process is slow?

```
<table>
<thead>
<tr>
<th>Process A</th>
<th>Process B</th>
</tr>
</thead>
<tbody>
<tr>
<td>while ( TRUE ) {</td>
<td></td>
</tr>
<tr>
<td>while ( turn != 0 )</td>
<td></td>
</tr>
</tbody>
</table>
|         /* twiddle */;
|     critical_things(); |
|     turn = 1; |
|     noncritical_things(); |
| } |
| while ( TRUE ) { |
|     while ( turn != 1 ) |
|         /* twiddle */;
|     critical_things(); |
|     turn = 0; |
|     noncritical_things(); |
| } |
```
8 Lecture: More Synchronization

Outline:
Announcements
LWP?
From Last time: Busywaiting
Onwards: Busy Waiting: With Hardware Support
Reflection
\texttt{sleep()} and \texttt{wakeup()}
Synchronization without busy waiting
Semaphores
Monitors: (Hoare 1974, Brinch Hansen 1975)
More Interprocess Communication

8.1 Announcements

- Coming attractions:

<table>
<thead>
<tr>
<th>Event</th>
<th>Subject</th>
<th>Due Date</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>asgn5</td>
<td>minget and minls</td>
<td>Wed Jun 5</td>
<td>23:59</td>
</tr>
<tr>
<td>asgn6</td>
<td>Yes, really</td>
<td>Fri Jun 7</td>
<td>23:59</td>
</tr>
<tr>
<td>final</td>
<td>stuff</td>
<td>Sat Jun 8</td>
<td>10:10 (in 03-201)</td>
</tr>
</tbody>
</table>

Use your own discretion with respect to timing/due dates.

- MEANINGFUL SYMBOLS (not: SIXTEEN)

- Don’t just click the little ’X’ to stop minix

- function pointers

- schedulers

- context structure

- tryAsgn2

- Lab02: Distributed version should work for qemu, vmware. You can install your own, of course.

8.2 LWP?

- Winding up the stack

- Function pointers

- tryAsgn2
8.3 From Last time: Busy waiting

A better solution: Peterson’s Solution.

The first safe software solution was proposed by Dekker and published by Dijkstra in 1965 (“Co-operating Sequential Processes,” in Programming Languages, London: Academic Press, 1965.)

Peterson came up with a better solution in 1981.

See Figure 11.

**Advantages:** Works. Does not require strict alternation.

**Disadvantages:** Busy waiting.

```c
#define TRUE 1
#define FALSE 0
#define N 2 /* number of processes */

int turn; /* whose turn is it */
int interested[N]; /* all values initially FALSE */

void enter_region(int self) { /* self is 0 or 1 */
    int other; /* number of the other process */

    other = 1−self;

    interested[self] = TRUE; /* show interest */
    turn = self; /* try and claim the turn */
    while ((turn==self) && (interested[other]==TRUE))
        /* dum-dee-dum */;
}

void leave_region(int self) { /* process who is leaving */
    interested[self]=FALSE;
}
```

Figure 11: Peterson’s solution for mutual exclusion

8.4 Onwards: Busy Waiting: With Hardware Support

Peterson’s soln is fairly complicated. What if we can have a little help.

- Test-and-set-lock instruction (TSL): Read a memory location and set a non-zero value to it.

**Advantages:** Atomic access guarantees simple correctness.

**Disadvantages:** Requires hardware support, still busy waiting

<table>
<thead>
<tr>
<th>enter:</th>
<th>tsl r1,lock</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>cmp r1,#0</td>
</tr>
<tr>
<td></td>
<td>jne enter</td>
</tr>
<tr>
<td></td>
<td>ret</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>leave:</th>
<th>move lock,#0</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>ret</td>
</tr>
</tbody>
</table>
8.5 Reflection

The fundamental busywaiting problem: Wasting time.

That is, *the waiting process can actively prevent the event for which it is waiting.*
(feeding the cat?) Remember the four desired principles:

1. No two processes may simultaneously enter the critical region.
2. No assumptions may be made about CPU speed of the number of CPUs
3. No process running outside of its critical section may run while another process is in its critical section.
4. No process should have to wait forever in its critical section.

Reconsider the problems with busywaiting:

- Except on a multiprocessor, busywaiting *must* waste time.
- Consider the priority inversion problem: a low-priority process is holding a lock that a high-priority process needs, but the low priority process can’t run while the high-priority one is waiting.

8.5.1 sleep() and wakeup()

**sleep()** goes to sleep until awakened

**wakeup(process)** wakes up a sleeping process

```
#define N 100;
int count=0;

void producer(void) {
    while(TRUE) {
        produce();
        if ( count == N ) /* full */
            sleep();
        enter_item();
        count = count + 1;
        if ( count == 1 ) /* empty */
            wakeup (consumer);
    }
}

void consumer(void) {
    while(TRUE) {
        if ( count == 0 ) /* empty */
            sleep();
        remove_item();
        count = count - 1;
        if ( count == N-1 ) /* full */
            wakeup (producer);
        consume();
    }
}
```

Figure 12: A producer-consumer implementation with a race condition

Race conditions still: Wakeup might be missed:

When the buffer is empty:

1. Scheduler interrupt consumer() after the count==0 test, but before the sleep().
2. producer() produces an item, notes that count is now 1, and wake()s consumer()
3. the \texttt{wake()} is lost because \texttt{consumer()} is not asleep.

4. \texttt{consumer()} is scheduled, then goes to sleep.

5. \texttt{producer()} continues to produce until the buffer is full, then goes to sleep.

Nobody ever wakes up.

8.6 Synchronization without busy waiting

8.7 Semaphores

Generalized \texttt{sleep()} and \texttt{wakeup()} using a counter.

\begin{verbatim}
  p(counter) down(counter) decrement counter; wait if zero
  v(counter) up(counter) increment counter; wakeup others if was zero
\end{verbatim}

Must be \textit{atomic}:

\begin{itemize}
  \item usually done as a system call with disabled interrupts—note this is not the same as a long-term busywait.
  \item LWP can turn off signals to achieve the same effect.
\end{itemize}

Semaphores can be used for both (And these are \textbf{different})

\begin{itemize}
  \item mutual exclusion (binary semaphore)
  \item synchronization (initially \textit{N} for producer-consumer)
\end{itemize}

Figure 13 shows a solution to the producer-consumer problem using semaphores.

Still must be careful: \textbf{if you lock in the wrong order, you can deadlock.} consider the effect of reversing the downs in figure 13.

\begin{verbatim}
#define N 100
semaphore mutex = 1;
semaphore empty = N;
semaphore full = 0;

void producer(void) {
  while(TRUE) {
    produce();
    down(&empty);
    down(&mutex);
    enter_item();
    up(&mutex);
    up(&full);
  }
}

void consumer(void) {
  while(TRUE) {
    down(&full);
    down(&mutex);
    remove_item();
    up(&mutex);
    up(&empty);
    consume();
  }
}
\end{verbatim}

Figure 13: A semaphore-based producer-consumer implementation
8.7.1 Monitors: (Hoare 1974, Brinch Hansen 1975)

Recall the sensitivity of semaphores to ordering we saw in the example of the other time: If the producer and consumer lock mutex and full/empty in the wrong order, they will deadlock.

Monitors are a higher-level synchronization mechanism requiring programming-language support.

- A monitor is region of code where only one process can be active at a time. (enforced by the language).
- Communication between processes is done via condition variables with the primitives:

  - wait(condition) wait until a signal. A process that blocks in the monitor releases other processes to enter.
  - Note that this will be ok, because the release is voluntary.
  - signal(condition) send a signal. To enforce the exclusion principle, a process that signals is required to leave the monitor immediately.
  - A signal wakes one process waiting on that condition variable.

  These are very like sleep and wakeup, but automagically synchronized so counters are unnecessary.

A monitor-based solution to the producer-consumer problem is shown in Figure 14.

8.8 More Interprocess Communication

Ok, so, looking at our mechanisms that work

<table>
<thead>
<tr>
<th>Mechanism</th>
<th>Work</th>
</tr>
</thead>
<tbody>
<tr>
<td>Spin Locks:</td>
<td>Waste Time</td>
</tr>
<tr>
<td>Semaphores:</td>
<td>Complicated (easy to get wrong)</td>
</tr>
<tr>
<td>Monitors:</td>
<td>Require language support</td>
</tr>
</tbody>
</table>

So what else can we do?
monitor ProducerConsumer
    condition full,empty;
    integer count;

    procedure enter()
    begin
        if count = N then
            wait(full);
            enter_item();
            count := count + 1;
        if count = 1 then
            signal(empty);
    end;

    procedure remove()
    begin
        if count = 0 then
            wait(empty);
            remove_item();
            count := count - 1;
        if count = N-1 then
            signal(full);
    end;

    count := 0;
end monitor

procedure consumer()
begin
    while TRUE do
    begin
        ProducerConsumer.remove();
        consume_item();
    end
end;

procedure producer()
begin
    while TRUE do
    begin
        produce_item();
        ProducerConsumer.enter();
    end
end;

Figure 14: A monitor-based solution to the producer-consumer problem.
9 Lecture: Even More Concurrency

Outline:
Announcements
Synchronization without busy waiting
Sleep and Wakeup
Semaphores
  Monitors: (Hoare 1974, Brinch Hansen 1975)
More Interprocess Communication
  Further generalization: message passing
Robust Programming Clinic
Classic IPC Problems

9.1 Announcements

• Coming attractions:

<table>
<thead>
<tr>
<th>Event</th>
<th>Subject</th>
<th>Due Date</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>asgn5</td>
<td>minget and minls</td>
<td>Wed Jun 5 23:59</td>
<td></td>
</tr>
<tr>
<td>asgn6</td>
<td>Yes, really</td>
<td>Fri Jun 7 23:59</td>
<td></td>
</tr>
<tr>
<td>final</td>
<td>stuff</td>
<td>Sat Jun 8 10:10 (in 03-201)</td>
<td></td>
</tr>
</tbody>
</table>

Use your own discretion with respect to timing/due dates.

• Don’t just click the little ’X’ to stop minix
• tryAsgn2
• Call stacks, Algol 60
• how to do exit()
• Wind up a stack.

9.2 Synchronization without busy waiting

9.3 Sleep and Wakeup

A nice idea, but broken.
9.4 Semaphores

Generalized \texttt{sleep()} and \texttt{wakeup()} using a counter.

\begin{verbatim}
  p(counter)  down(counter)  decrement counter; wait if zero
  v(counter)  up(counter)     increment counter; wakeup others if was zero
\end{verbatim}

Must be \textbf{atomic}:

- usually done as a system call with disabled interrupts—note this is not the same as a long-term busywait.
- LWP can turn off signals to achieve the same effect.

Semaphores can be used for both (And these are \textbf{different})

- mutual exclusion (binary semaphore)
- synchronization (initially N for producer-consumer)

Figure 15 shows a solution to the producer-consumer problem using semaphores. Still must be careful: \textbf{if you lock in the wrong order, you can deadlock}. consider the effect of reversing the downs in figure 15.

```c
#define N 100
semaphore mutex = 1;
semaphore empty = N;
semaphore full = 0;

void producer(void) {
  while(TRUE) {
    produce();
    down(&empty);
    down(&mutex);
    enter_item();
    up(&mutex);
    up(&full);
  }
}

void consumer(void) {
  while(TRUE) {
    down(&full);
    down(&mutex);
    remove_item();
    up(&mutex);
    up(&empty);
    consume();
  }
}
```

Figure 15: A semaphore-based producer-consumer implementation

9.4.1 Monitors: \textbf{(Hoare 1974, Brinch Hansen 1975)}

Recall the sensitivity of semaphores to ordering we saw in the example of the other time: If the producer and consumer lock \texttt{mutex} and \texttt{full/empty} in the wrong order, they will deadlock. Monitors are a higher-level synchronization mechanism requiring programming-language support.

- A monitor is region of code where only one process can be active at a time. (enforced by the language).
• Communication between processes is done via **condition variables** with the primitives:

  - **wait(condition)** wait until a signal. A process that blocks in the monitor releases other processes to enter.
    Note that this will be ok, because the release is *voluntary*.

  - **signal(condition)** send a signal. To enforce the exclusion principle, a process that signals is required to leave the monitor immediately.
    A signal wakes one process waiting on that condition variable.

  These are very like sleep and wakeup, but automagically synchronized so counters are unnecessary.

A monitor-based solution to the producer-consumer problem is shown in Figure 16.

### 9.5 More Interprocess Communication

Ok, so, looking at our mechanisms that work

- **Spin Locks**: Waste Time
- **Semaphores**: Complicated (easy to get wrong)
- **Monitors**: Require language support

So what else can we do?

#### 9.5.1 Further generalization: message passing

*If you need me, just whistle.*

Monitors require language support, a tricky thing in the “real world”. Message passing gets much of the same safety using operating system support.

Processes communicate via two primitive functions:

- **send(dst, message)**
- **receive(src, message)**

Where do they get sent? (How do we address them?)

Options:

- individual processes
- mailboxes (possibly multiple consumers)

In either case, there may or may not be buffers. If there is a buffer, the OS has to manage this buffer (a meta-producer-consumer problem :). If not, a sender is stopped until there is a receiver: rendezvous

Issues with message passing,

- messages can be lost (acknowledgement)
- messages can be duplicated (sequence numbers)
- messages can be forged (authentication)
- performance:
monitor ProducerConsumer
condition full, empty;
integer count;

procedure enter()
begin
    if count = N then
        wait(full);
        enter_item();
        count := count + 1;
    if count = 1 then
        signal(empty);
end;

procedure remove()
begin
    if count = 0 then
        wait(empty);
        remove_item();
        count := count - 1;
    if count = N - 1 then
        signal(full);
end;

count := 0;
end monitor

procedure consumer()
begin
    while TRUE do
    begin
        ProducerConsumer.remove();
        consume_item();
    end
end;

procedure producer()
begin
    while TRUE do
    begin
        produce_item();
        ProducerConsumer.enter();
    end
end;

Figure 16: A monitor-based solution to the producer-consumer problem.
– Copying overhead (copy, copy, copy)
– Transmission latency (You may as well do it yourself)

But it can elegantly solve some problems, as with the producer-consumer solution given in Figure 17. Message passing producer-consumer: send empty messages and send back full ones.

```c
#define N 100

void producer(void) {
    message m;
    thing item;
    while(TRUE) {
        produce(&item);
        receive(consumer,&m);
        build_message(&m,item)
        send(consumer,&m);
    }
}

void consumer(void) {
    message m;
    thing item;
    int i;
    /* send N empties */
    for(i=0;i<N;i++)
        send(producer,&m);
    /* go to work */
    while(TRUE) {
        receive(producer,&m);
        extract_item(&m,item);
        send(producer,&m);
        consume(&item);
    }
}
```

Figure 17: A message-passing solution to the producer-consumer problem.

### 9.6 Classic IPC Problems

- **Producer/Consumer (been there, done that?)**
- **Dining Philosophers (1965):** synchronization rite of passage.
  Competing access for limited resources.
  Issues:
  - deadlock
  - starvation
- **Readers/Writers: e.g. a database.**
  Issues:
  - Many readers can work simultaneously, but only one writer.
  - Starvation: If we let all the readers by (no problem), the writer may starve.
  - perhaps an aging scheme?
10 Lecture: Scheduling

Outline:
Announcements
We talked a lot about LWP
   From Last Time: Further generalization: message passing
Robust Programming Clinic
Classic IPC Problems
So what: Scheduling
   Process States
   Policy vs. Mechanism
   Process types
   When to schedule
   Evaluation Criteria
   Non-preemptive scheduling: run-to-completion
   Preemptive scheduling

10.1 Announcements

- Coming attractions:

<table>
<thead>
<tr>
<th>Event</th>
<th>Subject</th>
<th>Due Date</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>asgn5</td>
<td>minget and minls</td>
<td>Wed</td>
<td>Jun 5 23:59</td>
</tr>
<tr>
<td>asgn6</td>
<td>Yes, really</td>
<td>Fri</td>
<td>Jun 7 23:59</td>
</tr>
<tr>
<td>final</td>
<td>stuff</td>
<td>Sat</td>
<td>Jun 8 10:10</td>
</tr>
</tbody>
</table>

Use your own discretion with respect to timing/due dates.

- tryAsgn2

- Reminder about SPC Meeting

- Old exams on the web site (warning...)

10.2 We talked a lot about LWP

List of stuff to talk about wrt asgn02:

- Draw stack
- pointer arithmetic in size of pointee
- Allocate enough stack
- Do atomic swaps (w/\texttt{swap_rfiles()})
- Be paranoid
10.2.1 From Last Time: Further generalization: message passing

*If you need me, just whistle.*

Monitors require language support, a tricky thing in the “real world”. Message passing gets much of the same safety using operating system support.

Processes communicate via two primitive functions:

- send(dst, message)
- receive(src, message)

Where do they get sent? (How do we address them?)

Options:

- individual processes
- mailboxes (possibly multiple consumers)

In either case, there may or may not be buffers. If there is a buffer, the OS has to manage this buffer (a meta-producer-consumer problem :). If not, a sender is stopped until there is a receiver: rendezvous

Issues with message passing:

- messages can be lost (acknowledgement)
- messages can be duplicated (sequence numbers)
- messages can be forged (authentication)
- performance:
  - Copying overhead (copy, copy, copy)
  - Transmission latency (You may as well do it yourself)

But it can elegantly solve some problems, as with the producer-consumer solution given in Figure 18. Message passing producer-consumer: send empty messages and send back full ones.

10.3 Classic IPC Problems

- Producer/Consumer (been there, done that?)
  
  Competing access for limited resources.
  
  Issues:
  
  - deadlock
  - starvation

- Readers/Writers: e.g. a database.
  
  Issues:
  
  - Many readers can work simultaneously, but only one writer.
  - Starvation: If we let all the readers by (no problem), the writer may starve.
  - perhaps an aging scheme?


```c
#define N 100

void producer(void) {
    message m;
    thing item;
    while(TRUE) {
        produce(&item);
        receive(consumer,&m);
        build_message(&m,item)
        send(consumer,&m);
    }
}

void consumer(void) {
    message m;
    thing item;
    int i;

    /* send N empties */
    for(i=0;i<N;i++)
        send(producer,&m);

    /* go to work */
    while(TRUE) {
        receive(producer,&m);
        extract_item(&m,item);
        send(producer,&m);
        consume(&item);
    }
}
```

Figure 18: A message-passing solution to the producer-consumer problem.

10.4 So what: Scheduling

We talked a lot about running along until we block, then pick who gets to run, but how is this done?

10.4.1 Process States

Remember at any given time, any process can be in one of three possible states: Running, Ready, or Blocked. Possible transitions between these states are shown in Figure 19.

![Possible states for a process](image)

Figure 19: Possible states for a process

10.4.2 Policy vs. Mechanism

policy How we want things to behave. (graduate students should finish within 7 years.)
mechanism How we’re going to make it happen. (cut off funding. (vs. throwing them out.))

We’re pretty clear on how it happens (particularly after asgn2), but what is it we want to do anyway?

10.4.3 Process types
Processes are usually roughly categorized into one of two different types

IO Bound characterized by short bursts of computation before blocking on IO (or a semaphore)
- Might want to give priority because they can get done and go back to sleep. (hide IO latency)
- Also, more likely to be interactive.

Compute Bound characterized by long bursts of computation before blocking on IO (or a semaphore)

These are dynamic. A process may move back and forth.

10.4.4 When to schedule
Scheduling is mandatory in two cases:
1. When a process exits
2. When a process blocks

It might be desirable under a few other conditions:
1. When a new process is created
   (Consider the situation of parent and child after fork(jing))
2. When an IO interrupt occurs
3. When a timer interrupt occurs

10.4.5 Evaluation Criteria
What makes a good algorithm?

Fairness Make sure each process gets its fair share

Efficiency/Utilization keep the CPU busy 100 percent of the time

Response Time minimize response time for interactive users

Turnaround minimize turnaround time for batch users

Throughput maximize the number of jobs processed per time.

Faster, better, cheaper, choose two.
10.4.6 Non-preemptive scheduling: run-to-completion

- Run to completion/blockage

Examples:

1. FCFS

2. Shortest Job First:
   - Provably optimal
   - Problem: Starvation

Add an aging function?

10.4.7 Preemptive scheduling

**Round Robin**  Every (runnable) process goes in turn.
11 Lecture: Scheduling, this time for real

Outline:
- Announcements
- Classic IPC Problems
  - From Last time: Process types
  - When to schedule
  - Evaluation Criteria
  - Non-preemptive scheduling: run-to-completion
  - Preemptive scheduling

11.1 Announcements

- Coming attractions:

<table>
<thead>
<tr>
<th>Event</th>
<th>Subject</th>
<th>Due Date</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>asgn5</td>
<td>minget and minls</td>
<td>Wed</td>
<td>5 Jun 23:59</td>
</tr>
<tr>
<td>asgn6</td>
<td>Yes, really</td>
<td>Fri</td>
<td>7 Jun 23:59</td>
</tr>
<tr>
<td>final</td>
<td>stuff</td>
<td>Sat</td>
<td>8 Jun 10:10</td>
</tr>
</tbody>
</table>

Use your own discretion with respect to timing/due dates.

- Old exams on the web site (warning...) 

- Midterm end of next week.

- Asgn2 is due today. Do it.

- Cleaning up after yourself.
  - `ps -u username`
  - `killall -u username -r pn-cs453/lib`

- Style guide:
  - Clean build
  - Magic numbers
  - Long lines (`pnico/bin/longlines.pl`)
  - **not checking error returns**

These will always cost points unnecessarily.

11.2 Classic IPC Problems

- Producer/Consumer (been there, done that?)

  Competing access for limited resources.
  Issues:
  - deadlock
– starvation

• Readers/Writers: e.g. a database.

Issues:

– Many readers can work simultaneously, but only one writer.
– Starvation: If we let all the readers by (no problem), the writer may starve.
– perhaps an aging scheme?

11.2.1 From Last time: Process types

Processes are usually roughly categorized into one of two different types

**IO Bound** characterized by short bursts of computation before blocking on IO (or a semaphore)

– Might want to give priority because they can get done and go back to sleep. (hide IO latency)
– Also, more likely to be interactive.

**Compute Bound** characterized by long bursts of computation before blocking on IO (or a semaphore)

These are dynamic. A process may move back and forth.

11.2.2 When to schedule

Scheduling is mandatory in two cases:

1. When a process exits
2. When a process blocks

It might be desirable under a few other conditions:

1. When a new process is created
   (Consider the situation of parent and child after fork()ing)
2. When an IO interrupt occurs
3. When a timer interrupt occurs

11.2.3 Evaluation Criteria

What makes a good scheduling algorithm?

**Fairness** Make sure each process gets its fair share

**Efficiency/Utilization** keep the CPU busy 100 percent of the time

**Response Time** minimize response time for interactive users

**Turnaround** minimize turnaround time for batch users

**Throughput** maximize the number of jobs processed per time.
11.2.4 Non-preemptive scheduling: run-to-completion

- Run to completion/blockage

Examples:
1. FCFS
2. Shortest Job First:
   - Provably optimal
   - Problem: Starvation
   Add an aging function?

11.2.5 Preemptive scheduling

Round Robin  Every ( runnable) process goes in turn.

Priority Scheduling  Give each job (or class of jobs) a priority ( rank, price, etc) and choose the most important one.

   Idea: Break priorities into classes: IO Bound first, then compute-bound. Why?
   Example: CTSS (Compatible Time Sharing System):
      - large quantum for CPU-bound jobs:
      - Processes that use the whole quantum move down.
      - Processes doing IO move up to the top again.

Other possibilities for granting priority:
- Process already in memory?
- Locks held/needed
- Resource requirements
- Shortest remaining work next (past performance might be a good predictor of future results?)

Guaranteed/Lottery  Give each process its due.

Real Time  A whole other course
12 Lecture: MINIX IO Architecture

Outline:
Announcements
Aside: Reentrancy
   From Last time: Process types
   When to schedule
   Evaluation Criteria
   Non-preemptive scheduling: run-to-completion
   Preemptive scheduling
Minix architecture revisited
Minix 2 Scheduling
Minix 3 Scheduling
   Three-level scheduling
Scheduling Example
Minix IPC
Minix structures
Where do we go from here?
Input/Output
Devices
Deadlock and its avoidance
   Deadlock Avoidance Methods

12.1 Announcements

- Coming attractions:

<table>
<thead>
<tr>
<th>Event</th>
<th>Subject</th>
<th>Due Date</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>asgn5</td>
<td>minget and minls</td>
<td>Wed Jun 5 23:59</td>
<td></td>
</tr>
<tr>
<td>asgn6</td>
<td>Yes, really</td>
<td>Fri Jun 7 23:59</td>
<td></td>
</tr>
<tr>
<td>final</td>
<td>stuff</td>
<td>Sat Jun 8 10:10 (in 03-201)</td>
<td></td>
</tr>
</tbody>
</table>

Use your own discretion with respect to timing/due dates.

- Precision of expression is important
- Midterm is a week from Wednesday
- Old exams to web (and warning)
- Just a reminder that you should be reading T&W. Remember that chapter 2 reading is slow going.

12.2 Aside: Reentrancy

Reentrant code is code that written such that a single copy in memory can be shared between many applications at once. That is, a reentrant function is one where it is possible to safely have more than one activation at the same time.

What that means:

- no self-modifying code
• no static variables (except those that actually pertain to truly global state).

Examples:
\texttt{strcat()} is reentrant.
\texttt{strtok()} is not.

12.2.1 From Last time: Process types

Processes are usually roughly categorized into one of two different types

\textbf{IO Bound} characterized by short bursts of computation before blocking on IO (or a semaphore)

• Might want to give priority because they can get done and go back to sleep. (hide IO latency)
• Also, more likely to be interactive.

\textbf{Compute Bound} characterized by long bursts of computation before blocking on IO (or a semaphore)

These are dynamic. A process may move back and forth.

12.2.2 When to schedule

Scheduling is mandatory in two cases:

1. When a process exits
2. When a process blocks

It might be desirable under a few other conditions:

1. When a new process is created
   (Consider the situation of parent and child after fork(jing))
2. When an IO interrupt occurs
3. When a timer interrupt occurs

12.2.3 Evaluation Criteria

What makes a good scheduling algorithm?

\textbf{Fairness} Make sure each process gets its fair share

\textbf{Efficiency/Utilization} keep the CPU busy 100 percent of the time

\textbf{Response Time} minimize response time for interactive users

\textbf{Turnaround} minimize turnaround time for batch users

\textbf{Throughput} maximize the number of jobs processed per time.
12.2.4 Non-preemptive scheduling: run-to-completion

- Run to completion/blockage

Examples:

1. FCFS
2. Shortest Job First:
   - Provably optimal
   - Problem: Starvation

Add an aging function?

12.2.5 Preemptive scheduling

**Round Robin** Every (Runnable) process goes in turn.

**Priority Scheduling** Give each job (or class of jobs) a priority (rank, price, etc) and choose the most important one.

- Idea: Break priorities into classes: IO Bound first, then compute-bound. Why?
- Example: CTSS (Compatible Time Sharing System):
  - large quantum for CPU-bound jobs:
  - Processes that use the whole quantum move down.
  - Processes doing IO move up to the top again.

Other possibilities for granting priority:

- Process already in memory?
- Locks held/needed
- Resource requirements
  - Shortest remaining work next (past performance might be a good predictor of future results?)

**Guaranteed/Lottery** Give each process its due.

**Real Time** A whole other course

12.3 Minix architecture revisited

The structure of minix is as in Figure 20.

**Layer 1** Shceduling, handling traps and interrupts, facilitating message passing.

The bottom is written in assembly (must be), the rest is C. Clock and System Tasks exist here.
Layer 2  IO Processes, one per device type. Compiled into the kernel, but have separate identities, (and, if supported, privilege levels)

Layer 3  Server processes: File System, Memory Manager, Network Server, etc. These are user-level processes that implement the system calls.
These run with higher priority than user processes and never terminate while the system is running.

Layer 4  User-level processes.

### 12.4 Minix 2 Scheduling

Three-level priority scheduling system:

<table>
<thead>
<tr>
<th>Level</th>
<th>Priority</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>immediate (so no queue)</td>
</tr>
<tr>
<td>2</td>
<td>run to completion/block</td>
</tr>
<tr>
<td>3</td>
<td>run to completion/block</td>
</tr>
<tr>
<td>4</td>
<td>Round-robin. 100ms quantum</td>
</tr>
</tbody>
</table>

Remember the idle process

### 12.5 Minix 3 Scheduling

Sixteen-level priority scheduling system:
• Variable quantum size set in the process table

• A process that uses its entire quantum will be given another one and sent to the end of the queue.

• Potential change of priority level for processes using complete quanta:

  – If it uses its whole quantum and is its own successor: drops a level.
  – If it was not its own predecessor, rises a level, capped in process table.

• Processes that block with time remaining in their quantum are moved to the head of the queue when they return, but only with the remaining part of their quanta.

12.5.1 Three-level scheduling

For any scheduling scheme, it might be worthwhile to treat processes in memory and processes on disk differently.

Processes can be scheduled at three different locations:

admission The system can decide whether or not to admit new jobs, and what to do with them.

memory Constraints are different for jobs in memory and on disk

disk (above)

12.6 Scheduling Example

Perhaps?

12.7 Minix IPC

Three primitives of Minix IPC:

<table>
<thead>
<tr>
<th>Synchronous</th>
</tr>
</thead>
<tbody>
<tr>
<td>send(dest, &amp;message);</td>
</tr>
<tr>
<td>receive(source, &amp;message);</td>
</tr>
<tr>
<td>send_rec(dest_src, &amp;message);</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Asynchronous</th>
</tr>
</thead>
<tbody>
<tr>
<td>notify(dest);</td>
</tr>
</tbody>
</table>

Messages (other than notify()) communicated via rendezvous semantics: The sender waits until the receiver gets the message. Notify is asynchronous.

Why?

• simplifies buffer management.

• more predictable: there is no question of a program behaving differently given a different buffer size (out of its control)

(How would one find out the buffer size?)
12.8 Minix structures
To prepare for the next part of the course, read Sections 2.5–2.6 (this time for real):

- Understand the basic structure of the OS
- Be careful and read critically: There is much to learn from observing OS code. There are also places where OS writers do things that ordinary C programmers should not. (“advanced strategy” — G. Fink)
  
  Remember: design considerations for an OS are different from “mortal programs”. (e.g. consistency above all else panic(())
- Read the code; you will eventually anyhow.

12.9 Where do we go from here?
Three major operational areas for any operating system: (specializations of magic?)

**IO** How does information get in and out of the system and get where it’s going.

**Memory Management** Making sure that things are where programs expect them to be.

**Filesystem** Making sure things stay where put and that everything operates efficiently.

12.10 Input/Output
Without IO, there’s no real point in doing the computation. It’s also complicated.
As always, it’s all about abstraction: keep the dirty machine details hidden.

12.11 Devices
The Unix/Minix modes, devices are classified as:

- block devices (e.g. disks)
- character devices (e.g. ttys)


12.12 Deadlock and its avoidance
Since we’re talking about sharing, we have to worry about deadlock.

Occasionally a process must be given exclusive access to some resource, e.g.:

- a scanner,
- cdrom,
- tape drive,
- mouse,
- printer(lpd is a user process),
• terminal (usually unnoticed), etc.

Imagine two processes:

• copy tape to cdrom
• copy cdrom to tape

oops. (see Fig. 21 for a resource graph description)

A set of processes is deadlocked if each process in the set is waiting for an event that only another process in the set can cause.

To model deadlocks, we need to consider abstract resources of two types:

preemptable resource one that can be taken away from the process owning it with no ill effects. (e.g. the CPU, memory)

nonpreemptable resource one that can not be taken away from the process holding it without causing an error. (e.g. a printer once it’s started to print.)

To use a resource:

1. request the resource
2. use the resource
3. release the resource.

Conditions required for deadlock: (Coffman, et al., 1971)

1. Mutual exclusion processes don’t share
2. Hold and wait processes can hold a resource and make other requests
3. No pre-emption processes can’t be forced to surrender a resource
4. Circular wait the circle must exist.
12.12.1 Deadlock Avoidance Methods

- **Don’t** the ostrich approach

- **Detection and recovery** discover deadlocks and kill jobs until it’s gone. (carefully, or heuristically)

- **Prevention** place restrictions on accesses so that deadlock can never occur. (e.g. order resources)

- **Avoidance** Be very careful in resource allocation so it doesn’t happen.

What does Unix do?

Not a thing. The Unix way (as with many other operating systems) is to hope it doesn’t happen, and if it does, it expects some higher being (super user) to fix it. Think *Deus extra machina*. 
13 Lecture: Managing Multiple Resources

Outline:
- Announcements
- Where do we go from here?
- Input/Output
- Devices
- Deadlock and its avoidance
  - Deadlock Avoidance Methods
  - Banker’s Algorithm (Dijkstra, 1965)
  - Single Resource
  - Managing Multiple Resources
    - Resource Trajectories
  - Managing Multiple Resources, cont.
    - Multi-way banker’s
  - Wrapping up deadlock avoidance

13.1 Announcements
- Coming attractions:

<table>
<thead>
<tr>
<th>Event</th>
<th>Subject</th>
<th>Due Date</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>asgn5</td>
<td>minget and minls</td>
<td>Wed Jun 5</td>
<td>23:59</td>
</tr>
<tr>
<td>asgn6</td>
<td>Yes, really</td>
<td>Fri Jun 7</td>
<td>23:59</td>
</tr>
<tr>
<td>final</td>
<td>stuff</td>
<td>Sat Jun 8</td>
<td>10:10 (in 03-201)</td>
</tr>
</tbody>
</table>

Use your own discretion with respect to timing/due dates.

- `memmove(3)` vs. assignment

- Precision of expression is important

- github

13.2 Where do we go from here?

Three major operational areas for any operating system: (specializations of magic?)

IO   How does information get in and out of the system and get where it’s going.

Memory Management  Making sure that things are where programs expect them to be.

Filesystem  Making sure things stay where put and that everything operates efficiently.

13.3 Input/Output

Without IO, there’s no real point in doing the computation. It’s also complicated.
As always, it’s all about abstraction: keep the dirty machine details hidden.
13.4 Devices

The Unix/Minix modes, devices are classified as:

- block devices (e.g. disks)
- character devices (e.g. ttys)


13.5 Deadlock and its avoidance

Since we’re talking about sharing, we have to worry about deadlock. Occasionally a process must be given exclusive access to some resource, e.g.:

- a scanner,
- cdrom,
- tape drive,
- mouse,
- printer(lpd is a user process),
- terminal(usually unnoticed), etc.

Imagine two processes:

- copy tape to cdrom
- copy cdrom to tape

oops. (see Fig. 22 for a resource graph description)

A set of processes is deadlocked if each process in the set is waiting for an event that only another process in the set can cause.

Figure 22: How a deadlock forms

To model deadlocks, we need to consider abstract resources of two types:
**preemptable resource** one that can be taken away from the process owning it with no ill effects.
(e.g. the CPU, memory)

**nonpreemptable resource** one that cannot be taken away from the process holding it without causing an error. (e.g. a printer once it’s started to print.)

To use a resource:

1. request the resource
2. use the resource
3. release the resource.

**Conditions required for deadlock:** (Coffman, et al., 1971)

1. **Mutual exclusion** processes don’t share
2. **Hold and wait** processes can hold a resource and make other requests
3. **No pre-emption** processes can’t be forced to surrender a resource
4. **Circular wait** the circle must exist.

**13.5.1 Deadlock Avoidance Methods**

- **Don’t** the ostrich approach
- **Detection and recovery** discover deadlocks and kill jobs until it’s gone. (carefully, or heuristically)
- **Prevention** place restrictions on accesses so that deadlock can never occur. (e.g. order resources)
- **Avoidance** Be very careful in resource allocation so it doesn’t happen.

**What does Unix do?**

Not a thing. The Unix way (as with many other operating systems) is to hope it doesn’t happen, and if it does, it expects some higher being (super user) to fix it. Think *Deus extra machina.*

**13.6 Banker’s Algorithm** (Dijkstra, 1965)

Keep track of the current state of resource allocation and only approve resources that could not make any one customer’s requests unsatisfiable. (approve if there is a path that gets all customers to their credit limits.)

Each process must publish its maximum requirements up front.
13.6.1 Single Resource

Analogous to a line of credit: Bankers keep on hand much less than the sum of the maxima. The OS can, too.

A state is considered **safe** if there exists a sequence of states that leads to all customers getting their maximum loans. (resources)

The process:

1. Choose a process whose resource requirements can be satisfied. (It doesn’t matter which, because it always increases the resource pool.)

2. Assume its resources are released (because it’s finished)

3. Repeat until all processes terminate or there are no more satisfiable processes.

If all processes can terminate, the state **safe**. If not, it is **unsafe** and the resource request must be delayed.

Consider the following example in Figure 23 (from Tanenbaum).

<table>
<thead>
<tr>
<th>Name</th>
<th>Used</th>
<th>Max.</th>
<th>Needed</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>0</td>
<td>6</td>
<td>6</td>
</tr>
<tr>
<td>B</td>
<td>0</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>C</td>
<td>0</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>D</td>
<td>0</td>
<td>7</td>
<td>7</td>
</tr>
</tbody>
</table>

Available: 10

Safe: any order.

<table>
<thead>
<tr>
<th>Name</th>
<th>Used</th>
<th>Max.</th>
<th>Needed</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>1</td>
<td>6</td>
<td>5</td>
</tr>
<tr>
<td>B</td>
<td>1</td>
<td>5</td>
<td>4</td>
</tr>
<tr>
<td>C</td>
<td>2</td>
<td>4</td>
<td>2</td>
</tr>
<tr>
<td>D</td>
<td>4</td>
<td>7</td>
<td>3</td>
</tr>
</tbody>
</table>

Available: 2

Safe: \( C \rightarrow D \rightarrow B \rightarrow A \)

<table>
<thead>
<tr>
<th>Name</th>
<th>Used</th>
<th>Max.</th>
<th>Needed</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>1</td>
<td>6</td>
<td>5</td>
</tr>
<tr>
<td>B</td>
<td>2</td>
<td>5</td>
<td>3</td>
</tr>
<tr>
<td>C</td>
<td>2</td>
<td>4</td>
<td>2</td>
</tr>
<tr>
<td>D</td>
<td>4</td>
<td>7</td>
<td>3</td>
</tr>
</tbody>
</table>

Available: 1

Unsafe: Nobody

Figure 23: One dimensional example of Dijkstra’s Bankers Algorithm

13.7 Managing Multiple Resources

But multiple types of resources are a problem...
Initial condition:

<table>
<thead>
<tr>
<th>Name</th>
<th>Used</th>
<th>Max.</th>
<th>Needed</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>0</td>
<td>10</td>
<td>10</td>
</tr>
<tr>
<td>B</td>
<td>0</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>C</td>
<td>0</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>D</td>
<td>0</td>
<td>9</td>
<td>9</td>
</tr>
<tr>
<td>E</td>
<td>0</td>
<td>3</td>
<td>3</td>
</tr>
</tbody>
</table>

Available: 11

Safe: any order.

In progress:

<table>
<thead>
<tr>
<th>Name</th>
<th>Used</th>
<th>Max.</th>
<th>Needed</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>2</td>
<td>10</td>
<td>8</td>
</tr>
<tr>
<td>B</td>
<td>1</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>C</td>
<td>2</td>
<td>4</td>
<td>2</td>
</tr>
<tr>
<td>D</td>
<td>2</td>
<td>9</td>
<td>7</td>
</tr>
<tr>
<td>E</td>
<td>2</td>
<td>3</td>
<td>1</td>
</tr>
</tbody>
</table>

Available: 2

Safe: \(C(4) \rightarrow B(5) \rightarrow E(7) \rightarrow D(9) \rightarrow A(11)\)

After A requests one more unit:

<table>
<thead>
<tr>
<th>Name</th>
<th>Used</th>
<th>Max.</th>
<th>Needed</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>3</td>
<td>10</td>
<td>7</td>
</tr>
<tr>
<td>B</td>
<td>1</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>C</td>
<td>2</td>
<td>4</td>
<td>2</td>
</tr>
<tr>
<td>D</td>
<td>2</td>
<td>9</td>
<td>7</td>
</tr>
<tr>
<td>E</td>
<td>2</td>
<td>3</td>
<td>1</td>
</tr>
</tbody>
</table>

Available: 1

Unsafe: \(B(2) \rightarrow C(4) \rightarrow E(6) \rightarrow \text{Nobody}\)

Figure 24: Another example, but one that doesn’t wedge immediately.
13.7.1 Resource Trajectories

The example from Tanenbaum, p.249 is shown in Figure 25.
Yuck.

13.8 Managing Multiple Resources, cont.

13.8.1 Multi-way bankers’

Here we simply do a vectorized version of the Banker’s Algorithm:

- Maintain a table of held resources
- Maintain a table of maximum requests
- Maintain a table of remaining requests
- Maintain a vectors of allocated, free, total resources

If there does not exist a row less than the available vector, the system will deadlock, else:

1. Choose a process whose resource requirements can be satisfied. (It doesn’t matter which, because it always increases the resource pool.)
2. Assume its resources are released (because it’s finished)
3. Repeat until all processes terminate or there are no more satisfiable processes.
If all processes can terminate, the state safe. If not, it is unsafe and the resource request must be delayed.

See the example in Figure 26. Manipulations:

1. Initial situation

2. B requests an instance of $R_2$ (printer?): (safe: D, then A or E, then...)

3. E requests the last $R_2$ printer: unsafe

13.9 Wrapping up deadlock avoidance

Are any of these any good?

- Processes rarely know their resource requirements in advance.
- Processes come and go.

This is hard.

If you have a good idea, you can be famous like Dijkstra. :)

What does Unix do?

Not a thing. The Unix way (as with many other operating systems) is to hope it doesn’t happen, and if it does, it expects some higher being (super user) to fix it. Think Deus extra machina.
<table>
<thead>
<tr>
<th>$R_0$</th>
<th>$R_1$</th>
<th>$R_2$</th>
<th>$R_3$</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>3</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>B</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>C</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>D</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>E</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>$R_0$</th>
<th>$R_1$</th>
<th>$R_2$</th>
<th>$R_3$</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>B</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>C</td>
<td>3</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>D</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>E</td>
<td>2</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

(a) Initial Configuration

(b) After B requests an instance of $R_2$
(Safe: $D \rightarrow A \rightarrow E \rightarrow \ldots$)

<table>
<thead>
<tr>
<th>$R_0$</th>
<th>$R_1$</th>
<th>$R_2$</th>
<th>$R_3$</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>3</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>B</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>C</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>D</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>E</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>$R_0$</th>
<th>$R_1$</th>
<th>$R_2$</th>
<th>$R_3$</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>B</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>C</td>
<td>3</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>D</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>E</td>
<td>2</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

(b) After E requests an instance of $R_2$
(Unsafe: No process can be satisfied)

Figure 26: Multi-dimensional Bankers Algorithm
14 Lecture: IO Processing

Outline:
- Announcements
- Managing Multiple Resources
  - Resource Trajectories
- Managing Multiple Resources, cont.
  - Multi-way bankers’
- Wrapping up deadlock avoidance
- Back to I/O
- Devices
  - Low level considerations: timing, interleaving, etc.
  - Accessing a device: Device Controllers
- Reading from a device: DMA vs. Programmed IO
- Goals of IO Software
- IO Software
- Minix IO Structure
- Minix IPC
  - Interrupt Handlers
  - Simple
  - complex

14.1 Announcements

- Coming attractions:

<table>
<thead>
<tr>
<th>Event</th>
<th>Subject</th>
<th>Due Date</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>asgn5</td>
<td>minget and minls</td>
<td>Wed Jun 5 23:59</td>
<td></td>
</tr>
<tr>
<td>asgn6</td>
<td>Yes, really</td>
<td>Fri Jun 7 23:59</td>
<td></td>
</tr>
<tr>
<td>final</td>
<td>stuff</td>
<td>Sat Jun 8 10:10 (in 03-201)</td>
<td></td>
</tr>
</tbody>
</table>

Use your own discretion with respect to timing/due dates.

- ABSTRACTION
- Lab03 due Monday
  - no late days
  - I will release lab solns right after
- Exam Wednesday
  - Bring questions Monday; I’ll leave time.

14.2 Managing Multiple Resources

But multiple types of resources are a problem...

14.2.1 Resource Trajectories

The example from Tanenbaum, p.249 is shown in Figure 27.
Yuck.
14.3 Managing Multiple Resources, cont.

14.3.1 Multi-way bankers’

Here we simply do a vectorized version of the Banker’s Algorithm:

- Maintain a table of held resources
- Maintain a table of maximum requests
- Maintain a table of remaining requests
- Maintain a vectors of allocated, free, total resources

If there does not exist a row less than the available vector, the system will deadlock, else:

1. Choose a process whose resource requirements can be satisfied. (It doesn’t matter which, because it always increases the resource pool.)

2. Assume its resources are released (because it’s finished)

3. Repeat until all processes terminate or there are no more satisfiable processes.

If all processes can terminate, the state **safe**. If not, it is **unsafe** and the resource request must be delayed.

See the example in Figure 28. Manipulations:
Figure 28: Multi-dimensional Bankers Algorithm
1. Initial situation

2. B requests an instance of $R_2$ (printer?): (safe: D, then A or E, then...)

3. E requests the last $R_2$ printer: unsafe

14.4 Wrapping up deadlock avoidance

Are any of these any good?

- Processes rarely know their resource requirements in advance.
- Processes come and go.

This is hard.

If you have a good idea, you can be famous like Dijkstra. :)

What does Unix do?

Not a thing. The Unix way (as with many other operating systems) is to hope it doesn’t happen, and if it does, it expects some higher being (super user) to fix it. Think Deus extra machina.

14.5 Back to I/O

Without IO, there’s no real point in doing the computation. It’s also complicated.

As always, it’s all about abstraction: keep the dirty machine details hidden.

14.6 Devices

The Unix/Minix modes, devices are classified as:

- block devices (e.g. disks)
- character devices (e.g. ttys)


14.6.1 Low level considerations: timing, interleaving, etc.

Isn’t it nice the controller can take care of it?

(under each level is another nice level of abstraction)

14.6.2 Accessing a device: Device Controllers

Mercifully the OS talks to device controllers not the actual devices. Isn’t abstraction great?

Device controllers abstract away much of the complexity. Accessed via:

**Memory-Mapped IO** Device control registers are mapped into memory. (creates “holes” in memory)

**IO Ports** Device control registers must be accessed through special instructions. (convenient, but complicates the CPU).

Either way, we set a value in some register, then the controller does the thing, then it sets a register to tell us that it’s done it.
14.7 Reading from a device: DMA vs. Programmed IO

**Standard (“Programmed IO”)** Controller interrupts, CPU copies data from controller to memory.

**DMA** Controller copies data to memory, then interrupts.

At least we have interrupts (Think about the world if we didn’t. We’d just have to check again and again, called “polling”).

14.8 Goals of IO Software

- **abstraction** Abstract away the complexity to achieve device independence.
- **organization/standardization** It should be easy to name something. “/etc/motd”, rather than /dev/hda1/... or, worse, (3,1)
- **error handling** Fix it ASAP, or report it in a useful way.
- **synchronization** Make operations make sense. (e.g. sharing a printer.)

14.9 IO Software

As always, it’s all about abstraction: (note the correspondence to the layers in Minix)

1. Interrupt handlers (bottom)
2. Device drivers — knows about the device
3. Device-independent OS software — does not know about the device
4. User-level software (top)

14.10 Minix IO Structure

We discussed principles of resource management, but how does this really work in the Minix environment?

The IO architecture of MINIX is shown in Figure 29

| User Processes | make IO call; format IO: spooling |
| Device-independent software | Naming, protection, blocking, buffering, allocation |
| Device drivers | set up device registers; check status |
| Interrupt Handlers | wake up driver when IO completed |
| Hardware | perform IO |

Figure 29: Minix IO

Corresponds to:
- user processes
- servers
- tasks
- process management
14.11 Minix IPC

Three primitives of Minix IPC:

**Synchronous**
- `send(dest, &message);`
- `receive(source, &message);`
- `send_rec(dest_src, &message);`

**Asynchronous**
- `notify(dest);`

Messages (other than `notify()`) communicated via *rendezvous* semantics: The sender waits until the receiver gets the message. Notify is asynchronous.

Why?
- simplifies buffer management.
- more predictable: there is no question of a program behaving differently given a different buffer size (out of its control)
  (How would one find out the buffer size?)

14.11.1 Interrupt Handlers

- Disk
- Clock

Do a little work, then *tell someone.*

Can be simple or complex: (not always the way you’d think)

14.11.2 Simple

- disk: handler acknowledges interrupt and passes the word on to the device driver.

  why? These calls are infrequent, long term, and involve a lot of work.

14.11.3 complex

- clock: counts ticks `pending_ticks` and only wakes up handler if necessary. (Clock frequency 60hz, quantum 100ms, 6/schedule interval.) (See comment in `clock.c`)

  wakes up driver if tty event or SCHED_RATE

  why? These calls are frequent, high overhead, not much to be done.

  * `pending_ticks`:
  * This is protected by explicit locks in `clock.c`. Don’t
  * update realtime directly, since there are too many
  * references to it to guard conveniently.
  * `lost_ticks`:
  * Clock ticks counted outside the clock task.
  * `sched_ticks, prev_ptr`:
  * Updating these competes with similar code in `do_clocktick()`.
  * No lock is necessary, because if bad things happen here
  * (like `sched_ticks` going negative), the code in `do_clocktick()`
  * will restore the variables to reasonable values, and an
  * occasional missed or extra `sched()` is harmless.
  * Are these complications worth the trouble? Well, they make the system 15% faster on a 5MHz 8088, and make task debugging much easier since there are
  * no task switches on an inactive system.
• tty: (keyboard) each key event causes an interrupt (down/up) keep track of events (bochs?)
  More importantly: IO Processing state machine
15 Lecture: More IO Processing

Outline:
Announcements
Midterm Review
  What we’ve done
  Areas
  What do you want to talk about?
Intermission: Robust Programming
  Defensive Programming
  Information Hiding
  Expect the Impossible
Principles
  lab03/asgn4: Systems Programming Review

15.1 Announcements

• Coming attractions:

<table>
<thead>
<tr>
<th>Event</th>
<th>Subject</th>
<th>Due Date</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>asgn5</td>
<td>minget and minls</td>
<td>Wed Jun 5 23:59</td>
<td></td>
</tr>
<tr>
<td>asgn6</td>
<td>Yes, really</td>
<td>Fri Jun 7 23:59</td>
<td></td>
</tr>
<tr>
<td>final</td>
<td>stuff</td>
<td>Sat Jun 8 10:10 (in 03-201)</td>
<td></td>
</tr>
</tbody>
</table>

Use your own discretion with respect to timing/due dates.

• Midterm Wednesday
  – will be on chapters 1, 2, and 3.1–3.3 (Deadlocks)
  – Probably 6-7 questions.
    * Short answer for breadth
    * long answer for depth

• Size_t is unsigned

• Rounding up.

• COMPLEXITY is the enemy (asgn1)

15.2 Midterm Review

Review:

• Timesharing vs. multiprogramming

We spent the whole time talking about the midterm
15.2.1 What we’ve done
Introduction and Background
History and Definition of an O.S.
The Process Model
Operating System Structures
Intro. to Concurrency
Concurrency and Synchronization
Lightweight Processes
Concurrency
More Concurrency
Even More Concurrency
Scheduling
Scheduling, this time for real
MINIX IO Architecture
I/O and Deadlock
IO Processing
IO Processing

15.2.2 Areas
• Jokes
• Basic OS Knowledge
• Concurrency/Synchronization Knowledge
• threads/programming
• Systems Programming
• Synchronization
• Scheduling
• IO Handling

15.2.3 What do you want to talk about?
15.3 Intermission: Robust Programming

Overview:

- Defensive programming
- Information hiding
- Expect the impossible

15.3.1 Defensive Programming

- **Paranoia.** Not enough, but it’s a good place to start.
  - Don’t trust anything you don’t generate!
  - Whenever someone uses your program or library routine, assume they will try to break it.
  - When you call another function, check that it succeeds.
  - Most importantly, assume that your own code will have problems, and program defensively, so those problems can be detected as quickly as possible.

- **Stupidity.**
  - Assume that the caller or user is an idiot, and cannot read any manual pages or documentation.
  - Assume that if documentation exists it will:
    1. not be read, and
    2. be wrong
  - Program so that the code you write will handle incorrect, bogus, and malformed inputs and parameters in a reasonable fashion,
  - “reasonable” being defined by the environment. For example, if you print an error message, the message should be self-explanatory and not require the user to look up error codes in a manual.

Part of the problem is that in a week, you most likely will have forgotten the details of your program, and may call it incorrectly or give it bogus input. This programming style is also a form of defensive programming.

15.3.2 Information Hiding

<table>
<thead>
<tr>
<th>Principle of Least Privilege</th>
</tr>
</thead>
<tbody>
<tr>
<td>Every program and every user of the system should operate using the least set of privileges necessary to complete the job.</td>
</tr>
<tr>
<td>(defined in [?])</td>
</tr>
</tbody>
</table>

need to know.
• **Dangerous Implements.** A “dangerous implement” is anything that your routines expect to remain consistent across calls.

For example, in the standard I/O library, the contents of the FILE structure for allocated files is expected to remain fixed across calls to the standard I/O library. That makes it a “dangerous implement.”

- Users might accidentally (or deliberately) modify the data in that data structure, causing your library functions to fail—badly. Never return pointers or indices into arrays; always hide the true addresses (or indices) by using something called a “token.”
- Hiding data structures also makes your program or library more modular

### 15.3.3 Expect the Impossible

• **Can’t happen.** “As the old saw goes, “never say ‘never.’”’ Impossible cases are rarely that; most often, they are merely highly unlikely.”

• example: expansion of function return values

### 15.4 Principles

<table>
<thead>
<tr>
<th>Principle of Fail-Safe Defaults</th>
</tr>
</thead>
<tbody>
<tr>
<td>Base access decisions on permission rather than exclusion.</td>
</tr>
<tr>
<td>(defined in [?])</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Principle of Least Privilege</th>
</tr>
</thead>
<tbody>
<tr>
<td>Every program and every user of the system should operate using the least set of privileges necessary to complete the job.</td>
</tr>
<tr>
<td>(defined in [?])</td>
</tr>
</tbody>
</table>

Also “Principle of least astonishment”

### 15.4.1 lab03/asgn4: Systems Programming Review

• User IDs (*getuid(2), geteuid(2)*)

• Unix authentication `char *crypt(const char *key, const char *salt);`

  Passwd entry:
  
  pnico:x:uid:gid:GECOS:/vm/pnico:/bin/tcsh

• Password storage *getpwnam(2), getpwuid(2), getspnam(2), getpuid(2)*

• File Information: *stat(2), lstat(2), fstat(2)*

• Meanings of permissions bits:
  
  - `S_IRUSR`, etc.
  
  - `suid` (and meaning thereof)

• Changing them: *chmod(2), fchmod(2)*
• File times: st_atime, st_mtime, st_ctime. (Aside utime(2))
• File ownership: chown(2), fchown(2)
• Terminals: tcgetattr(2), tcsetattr(2).
16 Lecture: Midterm

Outline:
Announcements
Midterm

16.1 Announcements

- Coming attractions:

<table>
<thead>
<tr>
<th>Event</th>
<th>Subject</th>
<th>Due Date</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>asgn5</td>
<td>minget and minls</td>
<td>Wed Jun 5</td>
<td>23:59</td>
</tr>
<tr>
<td>asgn6</td>
<td>Yes, really</td>
<td>Fri Jun 7</td>
<td>23:59</td>
</tr>
<tr>
<td>final</td>
<td>stuff</td>
<td>Sat Jun 8</td>
<td>10:10</td>
</tr>
</tbody>
</table>

Use your own discretion with respect to timing/due dates.

- Good luck on the midterm

16.2 Midterm

The midterm was this day.
17 Lecture: Canceled

Outline:
- Devices
  - Low level considerations: timing, interleaving, etc.
  - Accessing a device: Device Controllers
- Reading from a device: DMA vs. Programmed IO
  - Goals of IO Software
  - IO Software
- Minix IO Structure
  - Minix IPC
    - Interrupt Handlers
      - Simple
      - complex
- Structure of Minix: Device Tasks (Drivers)
  - Driver Behavior
    - Example: Disk seek optimization

• Coming attractions:

<table>
<thead>
<tr>
<th>Event</th>
<th>Subject</th>
<th>Due Date</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>asgn5</td>
<td>minget and minls</td>
<td>Wed Jun 5</td>
<td>23:59</td>
</tr>
<tr>
<td>asgn6</td>
<td>Yes, really</td>
<td>Fri Jun 7</td>
<td>23:59</td>
</tr>
<tr>
<td>final</td>
<td>stuff</td>
<td>Sat Jun 8</td>
<td>10:10 (in 03-201)</td>
</tr>
</tbody>
</table>

Use your own discretion with respect to timing/due dates.

• Building assignments. I had to build several asgn2 and asgn3s by hand.

• Note labs/assignments

• Exams hopefully Friday, although it is hard to tell.

I got sick

17.1 Devices

The Unix/Minix modes, devices are classified as:

- block devices (e.g. disks)
- character devices (e.g. ttys)


17.1.1 Low level considerations: timing, interleaving, etc.

Isn’t it nice the controller can take care of it?
  (under each level is another nice level of abstraction)
17.1.2 Accessing a device: Device Controllers

Mercifully the OS talks to device controllers not the actual devices. Isn’t abstraction great?

Device controllers abstract away much of the complexity. Accessed via:

**Memory-Mapped IO**  Device control registers are mapped into memory. (creates “holes” in memory)

**IO Ports**  Device control registers must be accessed through special instructions. (convenient, but complicates the CPU).

Either way, we set a value in some register, then the controller does the thing, then it sets a register to tell us that it’s done it.

17.2 Reading from a device: DMA vs. Programmed IO

**Standard (“Programmed IO”)**  Controller interrupts, CPU copies data from controller to memory.

**DMA**  Controller copies data to memory, then interrupts.

At least we have interrupts (Think about the world if we didn’t. We’d just have to check again and again, called “polling”).

17.2.1 Goals of IO Software

**abstraction**  Abstract away the complexity to achieve device independence.

**organization/standardization**  It should be easy to name something. “/etc/motd”, rather than /dev/hda1/..., or, worse, (3,1)

**error handling**  Fix it ASAP, or report it in a useful way.

**synchronization**  Make operations make sense. (e.g. sharing a printer.)

17.2.2 IO Software

As always, it’s all about abstraction: (note the correspondence to the layers in Minix)

1. Interrupt handlers (bottom)
2. Device drivers — knows about the device
3. Device-independent OS software — does not know about the device
4. User-level software (top)

17.3 Minix IO Structure

We discussed principles of resource management, but how does this really work in the Minix environment?

The IO architecture of MINIX is shown in Figure 30

Corresponds to:

- user processes
17.4 Minix IPC

Three primitives of Minix IPC:

<table>
<thead>
<tr>
<th>Synchronous</th>
<th>Asynchronous</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>send(dest, &amp;message);</code></td>
<td><code>notify(dest);</code></td>
</tr>
<tr>
<td><code>receive(source, &amp;message);</code></td>
<td><code>send_rec(dest_src, &amp;message);</code></td>
</tr>
</tbody>
</table>

Messages (other than `notify()`) communicated via rendezvous semantics: The sender waits until the receiver gets the message. Notify is asynchronous.

Why?

- simplifies buffer management.
- more predictable: there is no question of a program behaving differently given a different buffer size (out of its control)

(How would one find out the buffer size?)

17.4.1 Interrupt Handlers

- Disk
- Clock

Do a little work, then tell someone.

Can be simple or complex: (not always the way you’d think)

17.4.2 Simple

- disk: handler acknowledges interrupt and passes the word on to the device driver.

  why? These calls are infrequent, long term, and involve a lot of work.
17.4.3 complex

- clock: counts ticks \texttt{pending_ticks} and only wakes up handler if necessary. (Clock frequency 60hz, quantum 100ms, 6/schedule interval. (See comment in \texttt{clock.c})

  wakes up driver if tty event or \texttt{SCHED\_RATE}

  why? These calls are frequent, high overhead, not much to be done.

  * \texttt{pending_ticks}:
  
  \begin{itemize}
  \item This is protected by explicit locks in \texttt{clock.c}. Don't
  \item update realtime directly, since there are too many
  \item references to it to guard conveniently.
  \item \texttt{lost_ticks}:
  \item Clock ticks counted outside the clock task.
  \item \texttt{sched_ticks}, \texttt{prev_ptr}:
  \item Updating these competes with similar code in \texttt{do\_clocktick()}.
  \item No lock is necessary, because if bad things happen here
  \item (like \texttt{sched_ticks} going negative), the code in \texttt{do\_clocktick()}
  \item will restore the variables to reasonable values, and an
  \item occasional missed or extra \texttt{sched()} is harmless.
  \item Are these complications worth the trouble? Well, they make the system 15% faster on a 5MHz 8088, and make task debugging much easier since there are
  \item no task switches on an inactive system.
  \end{itemize}

- tty: (keyboard) each key event causes an interrupt (down/up) keep track of events (bochs?)

  More importantly: IO Processing state machine

17.5 Structure of Minix: Device Tasks (Drivers)

Device drivers are independent processes.

Drivers are all structured as:

\begin{verbatim}
Initialization
while(TRUE) {
  receive(any,message)
  switch(mess.m_type) {
    ...
  }
  send(mess.m_source,reply);
}
\end{verbatim}

17.6 Driver Behavior

Minix’s device drivers are very simple so that we can get out heads around them. What would we do if we were working on a “real” system.

17.6.1 Example: Disk seek optimization

Device driver behavior can be very complex for optimization:

Consider the issues involved in disk access: seek time, rotational latency, transfer time...

Candidate algorithms:

- First come, first served
• Shortest seek first
• elevator algorithm
• Further: pick up "blocks of opportunity"; try to predict
18 Lecture: More Memory

Outline:
Announcements
Something completely different: integrity
  Motivation
  rules (again)
  Academic Misconduct
Harms
Consider the /dev/Secret assignment
System Event Framework: SEF
On to Memory
Why care?
Memory Management Strategies
  Multiprogramming with fixed partitions
Important concerns with multiprogramming
  Relocation
  Protection
Dynamic Segmentation/Swapping
Memory allocation strategies

18.1 Announcements

- Coming attractions:

<table>
<thead>
<tr>
<th>Event</th>
<th>Subject</th>
<th>Due Date</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>asgn5</td>
<td>minget and minls</td>
<td>Wed Jun 5 23:59</td>
<td></td>
</tr>
<tr>
<td>asgn6</td>
<td>Yes, really</td>
<td>Fri Jun 7 23:59</td>
<td></td>
</tr>
<tr>
<td>final</td>
<td>stuff</td>
<td>Sat Jun 8 10:10</td>
<td></td>
</tr>
</tbody>
</table>

  Use your own discretion with respect to timing/due dates.

- Be aware of which version of minix you’re running (and that it’s probably different from the 3.3 the tutorial is written for)

- Minix booting info: be specific:
  
c0d0p0s0> image=/boot/image/testimage
c0d0p0s0> boot

  Also: make hdboot (read usage(8))

- RTFM/RTFB/RTFA

- Remember, your SecretKeepers shouldn’t interact with stdin or stdout (stderr, is ok for error messages). They only interact with the filesystem through messages.

- Midterms friday (probably)
18.2 Consider the /dev/Secret assignment

- Message types (caller pid in IO_ENDPT, sender pid in m_source):

<table>
<thead>
<tr>
<th>Type</th>
<th>Meaning</th>
<th>Returns</th>
</tr>
</thead>
<tbody>
<tr>
<td>HARD_INT</td>
<td>Interrupt notification</td>
<td></td>
</tr>
<tr>
<td>DEV_GATHER_S</td>
<td>Read from device (COUNT, ADDRESS)</td>
<td>adjust iov_bytes</td>
</tr>
<tr>
<td>DEV_SCATTER_S</td>
<td>Write to device (COUNT, ADDRESS)</td>
<td>adjust iov_bytes</td>
</tr>
<tr>
<td>DEV_IOCTL</td>
<td>ioctl() operation on device</td>
<td>EINVAL</td>
</tr>
<tr>
<td>DEV_OPEN</td>
<td>Open operation on device</td>
<td>OK</td>
</tr>
<tr>
<td>DEV_CLOSE</td>
<td>Close operation on device</td>
<td>OK</td>
</tr>
<tr>
<td>CANCEL</td>
<td>Abort in-progress IO operation</td>
<td>EINTR</td>
</tr>
<tr>
<td>SYS_SIG</td>
<td>MINIX is shutting down</td>
<td></td>
</tr>
</tbody>
</table>

Look at a driver’s header to see what the fields look like (e.g. tty)

- Response:
  The task must send a m_type == TASK_REPLY message with the current process number in REP_PROC_NR and the resulting status in REP_STATUS.

- Kernel to user space and vice-versa. Remember, the kernel exists in virtual space, too.

```c
int sys_datacopy(
    int src_proc, /* pid of source process, or "SELF" */
    vir_bytes src_vir, /* virtual address of source buffer */
    int dst_proc, /* pid of destination process, or "SELF" */
    vir_bytes dst_vir, /* virtual address of destination buffer */
    phys_bytes bytes /* bytes to copy */
);```

```c
int sys_safecopyfrom (
    endpoint_t source, /* source process */
    cp_grant_id_t grant, /* source buffer */
    vir_bytes grant_offset, /* offset in source buffer (for block devs) */
    vir_bytes my_address, /* virtual address of destination buffer */
    size_t bytes, /* bytes to copy */
    int my_seg /* memory segment (It's 'D' :-/) */
);```

```c
int sys_safecopeto (  
    endpoint_t source, /* destination process */
    cp_grant_id_t grant, /* destination buffer */
    vir_bytes grant_offset, /* offset in destination buffer (for block devs) */
    vir_bytes my_address, /* virtual address of source buffer */
    size_t bytes, /* bytes to copy */
    int my_seg /* memory segment (It's 'D' :-/) */
);```

- Response:
  The task must send a m_type == TASK_REPLY message with the current process number in REP_PROC_NR and the resulting status in REP_STATUS.
• How to find out who is calling you? `getnucred(endpoint_t)`.
  As of Minix 3.1.8 only reliable in `open()`.

18.3 System Event Framework: SEF

• Because there are so many drivers of the same form, `driver.c` provides the System Event Framework, a skeleton driver with callbacks for the customizable parts:

```c
PRIVATE struct driver secret_tab =
{
  secret_name,
  secret_open,
  secret_close,
  secret_ioctl,
  secret_prepare,
  secret_transfer,
  nop_cleanup,
  secret_geometry,
  nop_alarm,
  nop_cancel,
  nop_select,
  nop_ioctl,
  do_nop,
};
```

• Lots of useful things are to be found in `com.h` and `ipc.h`.
• Note that the exact callbacks and names differ among versions of minix.

18.4 On to Memory

Parkenson’s law:

Programs expand to fill the memory available to hold them.

Remember this “law” is not inviolate.
How much memory is enough? *A little bit more.*
It is the memory manager’s function to manage the memory hierarchy:

• registers
• cache(s)
• memory
• disk
• tape
• carved stone tablets

and hide the complexity from the users.
Note: as size increases, cost, speed, and volatility decrease.
This can range from the simple to the incredibly complex.
18.5 Why care?

Consider:

- Embedded applications
- Performance
- Price

18.6 Memory Management Strategies

- Monoprogramming: one at a time
- Multiprogramming with fixed partitions

18.6.1 Multiprogramming with fixed partitions

Subdivide memory and let multiple programs run

- multiple queues based on size
- single queue
  
  optimization issues:
  
  – pick the biggest job that fits (uses memory well, but risks starvation)
  – solve this with an aging function?

  This approach was used by OS/360, called MFT (Multiprogramming with a Fixed Number of Tasks)

18.7 Important concerns with multiprogramming

Two issues:

- Relocation
- Protection

18.7.1 Relocation

Linking and loading assume addresses for program optimizations.

- use only relative addressing?
- re-link on loading? (does not solve protection issues)
  
  *we do this all the time, though, with shared libraries.*

- use base registers
18.7.2 Protection

Programs shouldn’t stomp on each other:

- go back to one at a time? (Coming down from the trees in the first place was a bad move?)
- IBM 360: PSW had a 4-bit key, and each 2K of memory also had a 4-bit key. All memory references were checked against the key.
- CDC 6600: base and limit registers:
  - All memory accesses have base register added to them
  - All memory accesses are checked to be below the limit.
- 8088 implemented base registers.
- A better plan came along later (virtual memory, but much more complicated).

18.8 Dynamic Segmentation/Swapping

Regardless of how we solve the protection and relocation problems, the fixed-size segments have to go.

Dynamic number and size of memory partitions—we can allocate exactly what we need (if we can figure it out).

- flexible
- more complicated (no such thing as a free lunch)

Problems:

- fragmentation (Leads to the same partitioning problems as before. Jobs can’t find spaces big enough to run.)
  Fragmentation occurs when the manner in which a subdividable resource (typically storage) is allocated leads to unusable portions.
  - internal fragmentation occurs when the allocated block is larger than can be used by the process. The remaining unused space inside the block is lost to the system.
  - external fragmentation occurs when the spaces between allocated blocks become too small to be usefully allocated to another process, even though sufficient total quantities of the resource are available.

What about fragmentation?

* Compaction?
  We can move processes, but this is computationally expensive and we may have to re-link (w/o base registers)?
  What about pointers in this environment?

- process growth
  - padding We can pad processes—leave extra space around them “just in case”.
  - swapping bump one process to disk and restart later.
  - divide divide the process into stack and heap parts that each grow in different directions.
18.9 Memory allocation strategies

**bitmaps** Simple manipulation (look for an appropriate block)
   but painfully slow

**lists** keep a list of adjacent (process/hole) blocks. Merge holes as possible

   Either way, options for picking a location:

   - first fit
   - next fit
   - best fit oddly enough, worse than first fit. (leaves little holes)
   - worst fit
19 Lecture: Midterm Post-Mortem

Outline:
   Announcements
   Something completely different: integrity rules (again)
   Motivation
   The Midterm
   General thoughts
   Consider the /dev/Secret assignment
   System Event Framework: SEF

19.1 Announcements
   • Coming attractions:

<table>
<thead>
<tr>
<th>Event</th>
<th>Subject</th>
<th>Due Date</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>asgn5</td>
<td>minget and minls</td>
<td>Wed Jun 5 23:59</td>
<td></td>
</tr>
<tr>
<td>asgn6</td>
<td>Yes, really</td>
<td>Fri Jun 7 23:59</td>
<td></td>
</tr>
<tr>
<td>final</td>
<td>stuff</td>
<td>Sat Jun 8 10:10</td>
<td>(in 03-201)</td>
</tr>
</tbody>
</table>

Use your own discretion with respect to timing/due dates.

• Be aware of which version of minix you’re running (and that it’s probably different from the 3.3 the tutorial is written for)

• Minix booting info: be specific:
  
  c0d0p0s0> image=/boot/image/testimage
  c0d0p0s0> boot

  Also: make hdboot (read usage(8))

19.2 The Midterm
<table>
<thead>
<tr>
<th>Grade</th>
<th>Cutoff</th>
<th>Percent</th>
</tr>
</thead>
<tbody>
<tr>
<td>A−</td>
<td>43.75</td>
<td>87.5%</td>
</tr>
<tr>
<td>B−</td>
<td>37.50</td>
<td>75.0%</td>
</tr>
<tr>
<td>C−</td>
<td>31.25</td>
<td>62.5%</td>
</tr>
<tr>
<td>D−</td>
<td>25.00</td>
<td>50.0%</td>
</tr>
</tbody>
</table>

Figure 31: Histogram of scores for the midterm
Overall:

- This was not a hard exam:

<table>
<thead>
<tr>
<th>Problem</th>
<th>Description</th>
<th>Source</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Process states</td>
<td>previous exam</td>
</tr>
<tr>
<td>2</td>
<td>Time for C.S./utilization</td>
<td>previous exam</td>
</tr>
<tr>
<td>3</td>
<td>Preemption</td>
<td>prev exam/class announcement</td>
</tr>
<tr>
<td>4</td>
<td>Busywaiting</td>
<td>previous exam</td>
</tr>
<tr>
<td>5</td>
<td>Synchronization</td>
<td>previous exam</td>
</tr>
<tr>
<td>6</td>
<td>Sched.</td>
<td>hw/soln/previous exam</td>
</tr>
</tbody>
</table>

- Final redemption plan.

The exam questions:

1. c’mon

2. The context switch has to happen within the 20ms allotted for each process.
   (and sanity check. \((20\text{ms} + 5\text{ms}) \times 50 = 1.25\text{s}\))

3. Preemption: Detail is important: “the OS does . . . ”? It’s not running.
   The most important concepts to capture here are:
   - The O.S. runs in supervisor mode which allows it to manipulate the interrupt vector (definition) and register an ISR.
   - Before changing privilege levels, the O.S.
     - installs ISRs for a timer and also for the system call interrupt, and
     - requests a timer interrupt (not signal) for some future time.
   - The O.S. then changes privilege levels and yields to the user process.
   - Eventually, one of two things happens:
     (a) the timer interrupts, or
     (b) the process makes a system call
   Either one causes an interrupt.
   - The ISR (installed by the OS and in write-protected memory) runs with supervisor privilege. Now O.S. code is running as superman again and can decide what to do.

Note: This is all about privilege. The OS gets supervisor privilege back when the interrupt handler executes. The user process cannot stop this because the user process does not have the authority to block interrupts or to change the handler.

“the process will voluntarily give up control” doesn’t actually solve the problem. How is it possible for a process to voluntarily become the OS again? (The mechanism is the same, the system call interfaces)
Also, Interrupt ≠ signal

4. Busywaiting. Yes, you’re waiting on something, but it’s in a particular way.

5. Synchronization
• Don’t use software lock variables. You know they don’t work
• Plan out your moves; you need somewhere to stand.
• Generally, directly interrogating a semaphore leads to a race condition so it’s pretty meaningless.
• Don’t deadlock

6. scheduling
• Be careful...

19.3 General thoughts

Fundamentals:
• Think, then answer
• Be clear in your answers. I can’t be inside your head.
• Mechanisms? Show understanding. (“the OS will reclaim control.” How?? It’s not running.)
• You only get to answer an exam question once!

Remember, you had most of the questions in advance.
19.4 Consider the /dev/Secret assignment

- Message types (caller pid in IO_ENPT, sender pid in m_source):

<table>
<thead>
<tr>
<th>Type</th>
<th>Meaning</th>
<th>Returns</th>
</tr>
</thead>
<tbody>
<tr>
<td>HARD_INT</td>
<td>Interrupt notification</td>
<td></td>
</tr>
<tr>
<td>DEV_GATHER_S</td>
<td>Read from device (COUNT, ADDRESS)</td>
<td>adjust iov, bytes</td>
</tr>
<tr>
<td>DEV_SCATTER_S</td>
<td>Write to device (COUNT, ADDRESS)</td>
<td>adjust iov, bytes</td>
</tr>
<tr>
<td>DEV_IOCTL</td>
<td>ioctl() operation on device</td>
<td>EINVAL</td>
</tr>
<tr>
<td>DEV_OPEN</td>
<td>Open operation on device</td>
<td>OK</td>
</tr>
<tr>
<td>DEV_CLOSE</td>
<td>Close operation on device</td>
<td>OK</td>
</tr>
<tr>
<td>CANCEL</td>
<td>Abort in-progress IO operation</td>
<td>EINTR</td>
</tr>
<tr>
<td>SYS_SIG</td>
<td>MINIX is shutting down</td>
<td></td>
</tr>
</tbody>
</table>

Look at a driver's header to see what the fields look like (e.g. tty)

- Response:

The task must send a m_type == TASK_REPLY message with the current process number in REP_PROC_NR and the resulting status in REP_STATUS.

- Kernel to user space and vice-versa. Remember, the kernel exists in virtual space, too.

```c
int sys_datalcopy(
    int src_proc, /* pid of source process, or "SELF" */
    vir_bytes src_vir, /* virtual address of source buffer */
    int dst_proc, /* pid of destination process, or "SELF" */
    vir_bytes dst_vir, /* virtual address of destination buffer */
    phys_bytes bytes /* bytes to copy */
);
```

```c
int sys_safe_copyfrom (
    endpoint_t source, /* source process */
    cp_grant_id_t grant, /* source buffer */
    vir_bytes grant_offset, /* offset in source buffer (for block devs) */
    vir_bytes my_address, /* virtual address of destination buffer */
    size_t bytes, /* bytes to copy */
    int my_seg /* memory segment (It's 'D' :-): */
);
```

```c
int sys_safe_copyto (
    endpoint_t source, /* destination process */
    cp_grant_id_t grant, /* destination buffer */
    vir_bytes grant_offset, /* offset in destination buffer (for block devs) */
    vir_bytes my_address, /* virtual address of source buffer */
    size_t bytes, /* bytes to copy */
    int my_seg /* memory segment (It's 'D' :-): */
);
```

- Response:

The task must send a m_type == TASK_REPLY message with the current process number in REP_PROC_NR and the resulting status in REP_STATUS.
• How to find out who is calling you? `getnucred(endpoint_t)`.
   As of Minix 3.1.8 only reliable in `open()`.

19.5 System Event Framework: SEF

• Because there are so many drivers of the same form, `driver.c` provides the System Event Framework, a skeleton driver with callbacks for the customizable parts:

```c
PRIVATE struct driver secret_tab =
{
    secret_name,
    secret_open,
    secret_close,
    secret_ioctl,
    secret_prepare,
    secret_transfer,
    nop_cleanup,
    secret_geometry,
    nop_alarm,
    nop_cancel,
    nop_select,
    nop_ioctl,
    do_nop,
};
```

• Lots of useful things are to be found in `com.h` and `ipc.h`.

• Note that the exact callbacks and names differ among versions of minix.
1. (20) Consider the following situation\textsuperscript{7}: A very narrow hurricane has washed out all but one lane of the Lake Pontchartrain Causeway\textsuperscript{8}. Given that it is a very large lake, going around is impractical, so it is necessary to come up with a system to keep the bridge open. The conditions:

- Cars arrive at random intervals from either the north or south.
- The remains of the bridge are only one car wide and cars cannot back up. That is, a car that meets another car is stuck forever.
- Whenever a car wants to enter the bridge, it calls the function \texttt{enter\_bridge(int direction)}
  with a pre-defined integer constant indicating the direction. This will be either \texttt{NORTH} or \texttt{SOUTH}. When it wants to leave, it calls \texttt{exit\_bridge(int dir)}.

Using semaphores and the C-like syntax used for semaphore examples in class and in Tanenbaum and Woodhull, develop a solution to the problem. Implement \texttt{enter\_bridge()} and \texttt{exit\_bridge()} and whatever auxiliary data and functions you may need. \textbf{Be sure to explain briefly why your solution works.}

\textbf{For partial credit:} produce a solution that allows cars to cross the bridge without risking meeting another car on the way (and getting stuck forever).

\textbf{For more partial credit:} produce a solution that guarantees that no car will have to wait forever to cross.

\textbf{For full credit:} produce a solution that does all of the above and allows multiple cars travelling in the same direction to be on the bridge at a time. (It is 24 miles long, after all.)

Write your code below (and on the next page if necessary):

\textbf{Solution:}

\begin{verbatim}
semaphore mutex; /* initialized to 1 */

void enter_bridge(direction d) {
    /* make sure nobody else is on the bridge */
    DOWN(mutex);
}

void exit_bridge(direction d) {
    /* indicate that the bridge is free */
    UP(mutex);
}
\end{verbatim}

A semaphore-based solution to problem 1 that allows one car at a time.

\textsuperscript{7}Note, this problem is too hard for a midterm, but it’s a good problem to consider, so I’m leaving it here.

\textsuperscript{8}Lake Pontchartrain is a lake 41 miles long and 24 miles wide north of New Orleans, La. The causeway is the bridge that spans the “short” direction and is one of the two roads out of the city.
To meet all three requirements, we
must not allow traffic in one direc-
tion to hold up traffic in the other
direction forever, but we can’t just
take turns, because a car may never
arrive on the other side, and we still
have to allow multiple cars to cross
in the same direction.
The technique is to check to see if
there are cars on the other side be-
fore entering the bridge. If there are,
we act as if they were already on the
bridge and wait. The last car cross-
ing in the current direction wakes
up all the waiting cars on the other
side.
The only subtle part if the solu-
tion is the question of what happens
if enter bridge() is interrupted at
line 41 (marked at right). Could it
cause problems if a process were to
decide to wait until cars have cleared
the bridge, but be interrupted before
it an down waitsem?
Consider: The only way a car(C)
could decide to wait is if there is
at least one car on the bridge, and
before the C releases mutex, it has
already been added to the count of
waiting cars. So, the worst that
could happen during an interruption
is that the only car on the bridge
could leave. On its way out, how-
ever, it would put all waiting cars
onto the bridge and up waitsem the
appropriate number of times. So, all
that will happen is that when C does
get to run and downs waitsem, the
operation will succeed without block-
ing.

```c
typedef int direction;
#define NORTH 0
#define SOUTH 1

semaphore mutex; /* initialized to 1 */
semaphore waitsem[2]; /* initialized to 0,0 */

direction bridgedir = NORTH; /* direction cars are currently moving */
                      /* how it is initialized doesn’t matter */
int onbridge = 0; /* there are no cars initially on the bridge */
int waiting[2] = {0,0}; /* nobody is waiting in either direction */

direction otherdir(direction d) {
    direction ret;
    if ( d == NORTH )
        ret = SOUTH;
    else
        ret = NORTH;
    return ret;
}

void enter_bridge(direction d) {
    direction other = otherdir(d);
    int willwait = 0;

    DOWN(mutex);
    if ( onbridge == 0 ) {
        /* there’s nobody on the bridge, take it */
        onbridge = 1;
        bridgedir = d;
        waiting[d] = 0;
    } else if ( (onbridge > 0) &&
                (bridgedir == d) &&
                (waiting[other] == 0) ) {
        /* it’s going our way, and nobody’s */
        onbridge = onbridge + 1; /* add ourselves to the bridge */
    } else {
        waiting[d] = waiting[d] + 1; /* otherwise, we have to wait. */
        willwait = 1;
    }
    UP(mutex);
}

```

```c
/* HERE HERE HERE */
if ( willwait ) /* if we have to wait, wait */
    DOWN(waitsem[d]);
}

void exit_bridge(direction d) {
    /* indicate that the bridge is free */
    direction other = otherdir(d);

    DOWN(mutex);
    onbridge = onbridge-1; /* decrement the counter */
    if ( onbridge == 0 ) {
        /* if we were the last */
        if ( waiting[other] != 0 ) {
            /* if someone is waiting to go the other way, put them */
            /* on the bridge, then wake those cars up */
            onbridge = waiting[other];
            bridgedir = other;
            waiting[other] = 0;
        }
        for(i=0;i<wating[other];i++) { /* wake up the waiting cars */
            UP(waitsem[other]);
        }
    }
    UP(mutex);
}
```
20 Lecture: Virtual Memory

Outline:
Announcements
The CSU
From last time: Dynamic segmentation
Memory allocation strategies
Dealing with memory size
Virtual Memory (Fotheringham, 1961)
What it’s all about
Translation Logistics
Where to keep the page table?
Possible approaches to dealing with size
Translation Lookaside Buffers
What about really big memories?
Limited memory size
Inverted page tables

20.1 Announcements

- Coming attractions:

<table>
<thead>
<tr>
<th>Event</th>
<th>Subject</th>
<th>Due Date</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>asgn5</td>
<td>minget and minls</td>
<td>Wed Jun 5</td>
<td>23:59</td>
</tr>
<tr>
<td>asgn6</td>
<td>Yes, really</td>
<td>Fri Jun 7</td>
<td>23:59</td>
</tr>
<tr>
<td>final</td>
<td>stuff</td>
<td>Sat Jun 8</td>
<td>10:10 (in 03-201)</td>
</tr>
</tbody>
</table>

Use your own discretion with respect to timing/due dates.

- Final Redemption Plan
- Common Final: Saturday, June 8th, 10:10-1:00
- Banquets
- be aware of which version of minix you’re running
- Minix booting info: be specific:
  - `c0d0p0s0> image=/boot/image/testimage`
  - `c0d0p0s0> boot`
  - Also: `make hdboot`
- RTFM/RTFB/RTFA
- Remember, your SecretKeepers shouldn’t interact with stdin or stdout (stderr, is ok for error messages). They only interact with the filesystem through messages.
- Can you make the system busy?
20.2 From last time: Dynamic segmentation

More complicated (no such thing as a free lunch)

Problems:

- **fragmentation** (Leads to the same partitioning problems as before. Jobs can’t find spaces big enough to run.)

  Fragmentation occurs when the manner in which a subdividable resource (typically storage) is allocated leads to unusable portions.

  **internal** fragmentation occurs when the allocated block is larger than can be used by the process. The remaining unused space inside the block is lost to the system.

  **external** fragmentation occurs when the spaces between allocated blocks become too small to be usefully allocated to another process, even though sufficient total quantities of the resource are available.

What about fragmentation?

  - Compaction?
    
    We can move processes, but this is computationally expensive and we may have to re-link (w/o base registers)?

  What about pointers in this environment?

- **process growth**

  **padding** We can pad processes—leave extra space around them “just in case”.

    - Inward-facing (neat, bounded)
    - Outward facing (allows for swapping)

  **swapping** bump one process to disk and restart later.

  **divide** divide the process into stack and heap parts that each grow in different directions.

20.3 Memory allocation strategies

  **bitmaps** Simple manipulation (look for an appropriate block)

    but painfully slow

  **lists** keep a list of adjacent (process/hole) blocks. Merge holes as possible

    Either way, options for picking a location:

    - first fit
    - next fit
    - best fit oddly enough, worse than first fit. (leaves little holes)
    - worst fit
20.4 Dealing with memory size

And now we return to the original problem: Too little memory.
But we don’t really use all of memory at once, do we?
All of these solutions are based on what would come to be known as the principle of **locality of reference** and of a **working set** (Denning, 1968, 1980).

- overlays
- virtual memory

20.5 Virtual Memory (Fotheringham, 1961)

**transparent** It’s there, but you can’t see it.

**virtual** You can see it, but it isn’t there.

20.5.1 What it’s all about

**The way it works**

- Programs exist in a *virtual address space*
- Virtual memory is divided into *pages*
- Physical memory is divided into *frames*
- Non-resident pages are stored in backing store (*swap space*) and brought in as needed.
- Virtual addresses are translated into physical addresses by the MMU (Memory Management Unit) using *page tables*

- Page tables contain:
  - virtual page number (index)
  - cache enabled (C)
  - modified bit (M)
  - referenced bit (R)
  - protection bits (rwx?)
  - present bit (P)
  - physical frame number

- Attempting to reference an unmapped page causes a *page fault*.

VM translation is illustrated in Figure 32
What it provides

1. expanded memory
2. relocation
3. isolation

The last one, isolation is important to consider: no process can refer to another process's memory. So how can processes communicate?

Only through the kernel because the kernel has access to physical memory. (At least part of it—the pager—must have access. (How does it do that? It’s using VM hardware, too.))

![Virtual Memory translation in the MMU (4k pages)](image)

20.6 Translation Logistics

Consider: page tables are large and must be accessed quickly:

For 4KB Pages, 27 bits per entry:

<table>
<thead>
<tr>
<th>C</th>
<th>M</th>
<th>R</th>
<th>r</th>
<th>w</th>
<th>x</th>
<th>P</th>
<th>physical page number</th>
</tr>
</thead>
</table>

At one word (32 bits) per Page table entry:

<table>
<thead>
<tr>
<th>Virtual Memory size</th>
<th>4Gb ($2^{32}$)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Page Size:</td>
<td>4k ($2^{12}$)</td>
</tr>
<tr>
<td>No. Pages:</td>
<td>1048576 ($2^{20}$)</td>
</tr>
<tr>
<td>P.T. Size</td>
<td>4Mb</td>
</tr>
</tbody>
</table>
Remember, that’s **per process** because every process has its own virtual address space.

### 20.6.1 Where to keep the page table?
- In the MMU? (fast translation, slow loading)
  With a 400 MHz memory bus, to move 4Mb would take 0.0026s, and that’s just to move the data ignoring any instruction overhead. Of course, you’d have to move the old table out, too, so, that’d be 5.2 msec per context switch. Given that a quantum is 100msec, that’s a minimum of 5% overhead!
- in Memory (slower translation, possibly faster loading)

### 20.6.2 Possible approaches to dealing with size
- restrict virtual memory
- page the page tables (multi-level page tables?)
- Even so, many references per instruction:
  TLB: a small, associative page-table cache.

### 20.6.3 Translation Lookaside Buffers
- Small (32–64 entry) associative cache of page table entries
- Management (hardware vs. software.)
  **hardware management** A TLB miss causes an ordinary page lookup and the MMU replaces a TLB entry.
  **software management** A TLB miss causes a trap. This has to be handled efficiently, but with reasonably large TLBs the miss rate can be kept low.
  Why do this? This leads to a much simpler (i.e. faster) TLB.

  Idea: Keep a cache of recently evicted TLB entries around and check them first. (Consider the case where a multi-level page table must be searched.)
21 Lecture: More VM

Outline:
- Announcements
  - From last time: Virtual Memory (Fotheringham, 1961)
    - What it’s all about
  - Translation Logistics(HERE)
- Where to keep the page table?
- Possible approaches to dealing with size
- Translation Lookaside Buffers
- What about really big memories?
- Limited memory size
- Inverted page tables
- Page replacement algorithms
  - Optimal
  - Random
  - Not Recently Used
  - FIFO
  - Second Chance
  - Clock (optimized Second Chance)
  - LRU
  - Not Frequently Used

21.1 Announcements

- Coming attractions:

<table>
<thead>
<tr>
<th>Event</th>
<th>Subject</th>
<th>Due Date</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>asgn5</td>
<td>minget and minls</td>
<td>Wed Jun 5</td>
<td>23:59</td>
</tr>
<tr>
<td>asgn6</td>
<td>Yes, really</td>
<td>Fri Jun 7</td>
<td>23:59</td>
</tr>
<tr>
<td>final</td>
<td>stuff</td>
<td>Sat Jun 8</td>
<td>10:10</td>
</tr>
</tbody>
</table>

| Notes | (in 03-201) |

Use your own discretion with respect to timing/due dates.

- Remember, your SecretKeepers shouldn’t interact with stdin or stdout (stderr, is ok for error messages). They only interact with the filesystem through messages.

21.2 From last time: Virtual Memory (Fotheringham, 1961)

**transparent** It’s there, but you can’t see it.

**virtual** You can see it, but it isn’t there.

21.2.1 What it’s all about

The way it works

- Programs exist in a *virtual address space*
- Virtual memory is divided into *pages*
- Physical memory is divided into *frames*
• Non-resident pages are stored in backing store (swap space) and brought in as needed.

• Virtual addresses are translated into physical addresses by the MMU (Memory Management Unit) using page tables

• Page tables contain:
  – virtual page number (index)
  – cache enabled (C)
  – modified bit (M)
  – referenced bit (R)
  – protection bits (rwx?)
  – present bit (P)
  – physical frame number

• Attempting to reference an unmapped page causes a page fault.

VM translation is illustrated in Figure 33

What it provides

1. expanded memory
2. relocation
3. isolation

The last one, isolation is important to consider: no process can refer to another process’s memory. So how can processes communicate?

Only through the kernel because the kernel has access to physical memory. (At least part of it—the pager—must have access. (How does it do that? It’s using VM hardware, too.))

21.3 Translation Logistics(HERE)

Consider: page tables are large and must be accessed quickly:

For 4KB Pages, 27 bits per entry:

<table>
<thead>
<tr>
<th>C</th>
<th>M</th>
<th>R</th>
<th>r</th>
<th>w</th>
<th>x</th>
<th>P</th>
<th>physical page number</th>
</tr>
</thead>
</table>

At one word (32 bits) per Page table entry:

<table>
<thead>
<tr>
<th>Virtual Memory size:</th>
<th>4Gb (2^{32})</th>
</tr>
</thead>
<tbody>
<tr>
<td>Page Size:</td>
<td>4k (2^{12})</td>
</tr>
<tr>
<td>No. Pages:</td>
<td>1048576 (2^{20})</td>
</tr>
<tr>
<td>P.T. Size</td>
<td>4MB</td>
</tr>
</tbody>
</table>

Remember, that’s per process because every process has its own virtual address space.
Figure 33: Virtual Memory translation in the MMU (4k pages)
21.3.1 Where to keep the page table?

- In the MMU? (fast translation, slow loading)
  With a 400 MHz memory bus, to move 4MB would take 0.0026s, and that’s just to move the data ignoring any instruction overhead. Of course, you’d have to move the old table out, too, so, that’d be 5.2 msec per context switch. Given that a quantum is 100msec, that’s a minimum of 5% overhead!
- in Memory (slower translation, possibly faster loading)

21.3.2 Possible approaches to dealing with size

- restrict virtual memory
- page the page tables (multi-level page tables?)
- Even so, many references per instruction:
  TLB: a small, associative page-table cache.

21.3.3 Translation Lookaside Buffers

- Small (32–64 entry) associative cache of page table entries
- Management (hardware vs. software.)
  
  **hardware management** A TLB miss causes an ordinary page lookup and the MMU replaces a TLB entry.
  
  **software management** A TLB miss causes a trap. This has to be handled efficiently, but with reasonably large TLBs the miss rate can be kept low.
  Why do this? This leads to a much simpler (i.e. faster) TLB.
  
  Idea: Keep a cache of recently evicted TLB entries around and check them first. (Consider the case where a multi-level page table must be searched.)

21.3.4 What about really big memories?

<table>
<thead>
<tr>
<th>Memory size:</th>
<th>$2^{41}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Page Size:</td>
<td>4k ($2^{12}$)</td>
</tr>
<tr>
<td>No. Pages:</td>
<td>$2^{52}$ entries, 4,503,599,627,370,496</td>
</tr>
<tr>
<td>P.T. Size</td>
<td>8B * $2^{44}$ is a lot (36,028,797,018,963,968 ≈ 36PB)</td>
</tr>
</tbody>
</table>

21.3.5 Limited memory size

Always an option
  (Modern linux only uses 47 bits of it.)

21.3.6 Inverted page tables

Keep track of the frames, not the pages: Slow, but bounded in size. (For 1 Gb, must search 256k page table entries, 1MB on every memory reference?)
  
  The TLB saves us.
  
  Example: Order card catalog in the library by call number (where it is stored) vs. title (what it is).
21.4 Page replacement algorithms


21.4.1 Optimal

Have a list of future references and evict the page that won’t be used for the longest time.

21.4.2 Random

Pick a page, any page

21.4.3 Not Recently Used

Divide the pages into categories based on R(reference) and M(modify) bits: (Clear R on clock tick)

1. not referenced, not modified
2. not referenced, modified
3. referenced, not modified
4. referenced, modified

Evict a random page from the lowest numbered category.
Rationale?

21.4.4 FIFO

Pure round-robin first in, first out. Evict the oldest page in memory. (Problem: might be main())

21.4.5 Second Chance

Like FIFO, but if the head of the queue was referenced, clear the referenced bit and send it to the back of the list.
Reasonable, but has a lot of manipulation.

21.4.6 Clock (optimized Second Chance)

Second chance, but moves a pointer rather than manipulating the list.

21.4.7 LRU

Least Recently Used — best approximator of Optimal, but difficult:

- requires list manipulation

- can be approximated by a counter that’s updated in the page table at every reference. (requires hardware)
21.4.8 Not Frequently Used

Simulate LRU: At every clock tick, add the Reference bit into a counter in the page table. Replace the page with the lowest counter on a page fault.

Problem: NFU Never forgets.

Soln: Modify NFU with aging: Right-shift the counter and add the bit in at the high end.
# Lecture: Wrapping up VM

**Outline:**

- **Announcements**
  - From Last Time: Inverted page tables
  - Page replacement algorithms
    - Optimal
    - Random
    - Not Recently Used
    - FIFO
    - Second Chance
    - Clock (optimized Second Chance)
    - LRU
    - Not Frequently Used
- **Paging Policies (Design Issues)**
  - When to page
  - Page allocation
  - Page size
  - VM Tricks
  - Minix

**Filesystems**

- Some thoughts on filesystem organization

**Filesystem Implementation Considerations**

These are explored in the following three subsections

**Unix Filsystem Structure**

- Resource Allocation: Disk Management

**Filesystems**

- Some thoughts on filesystem organization

**Filesystem Implementation Considerations**

These are explored in the following three subsections

**Unix Filsystem Structure**

- Resource Allocation: Disk Management

## 22.1 Announcements

- Coming attractions:

<table>
<thead>
<tr>
<th>Event</th>
<th>Subject</th>
<th>Due Date</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>asgn5</td>
<td>minget and minls</td>
<td>Wed Jun 5</td>
<td>23:59</td>
</tr>
<tr>
<td>asgn6</td>
<td>Yes, really</td>
<td>Fri Jun 7</td>
<td>23:59</td>
</tr>
<tr>
<td>final</td>
<td>stuff</td>
<td>Sat Jun 8</td>
<td>10:10 (in 03-201)</td>
</tr>
</tbody>
</table>

Use your own discretion with respect to timing/due dates.

- Asgn5 is supposed to work *outside* of minix

- Last chance to check for conflicts on the Final


22.1.1 From Last Time: Inverted page tables
Keep track of the frames, not the pages: Slow, but bounded in size. (For 1 Gb, must search 256k page table entries, 1MB on every memory reference?)
  The TLB saves us.
  Example: Order card catalog in the library by call number (where it is stored) vs. title (what it is).

22.2 Page replacement algorithms

22.2.1 Optimal
Have a list of future references and evict the page that won’t be used for the longest time.

22.2.2 Random
Pick a page, any page

22.2.3 Not Recently Used
Divide the pages into categories based on R(reference) and M(modify) bits: (Clear R on clock tick)
  1. not referenced, not modified
  2. not referenced, modified
  3. referenced, not modified
  4. referenced, modified

Evict a random page from the lowest numbered category.
Rationale?

22.2.4 FIFO
Pure round-robin first in, first out. Evict the oldest page in memory. (Problem: might be main())

22.2.5 Second Chance
Like FIFO, but if the head of the queue was referenced, clear the referenced bit and send it to the back of the list.
  Reasonable, but has a lot of manipulation.

22.2.6 Clock (optimized Second Chance)
Second chance, but moves a pointer rather than manipulating the list.
22.2.7 LRU
Least Recently Used — best approximator of Optimal, but difficult:

- requires list manipulation
- can be approximated by a counter that’s updated in the page table at every reference. (requires hardware)

22.2.8 Not Frequently Used
Simulate LRU: At every clock tick, add the Reference bit into a counter in the page table. Replace the page with the lowest counter on a page fault.

Problem: NFU Never forgets.
Sohn: Modify NFU with aging: Right-shift the counter and add the bit in at the high end.

22.3 Paging Policies (Design Issues)
We’ve spent a lot of time discussing which page to replace, but let’s consider larger issues:

- When to page
- How many frames should a process get

22.3.1 When to page
Definitions:

locality of reference the principle that holds that a page that is referenced is likely to be referenced again. (Past performance is an indicator of future behavior.)

Working Set the set of pages a process needs to run (on average)

The working set has to be determined experimentally: set a threshold above which some proportion of references hit the WS.

Considerations:

thrashing if a process spends most of its time paging, it is said to be thrashing. (Forward progress of only a few instructions before having a page fault. The job will never finish.)

Example: 6-page working set, FIFO, in 5-pages of memory.

Techniques:

demand paging Only bring in a page on a page fault (page nothing in in advance).

prepaging Make sure a process’s working set is in memory before allowing it to run.

swapping sometimes there are too many processes for the resources, and some must be paged out to disk.
22.3.2 Page allocation

**local** Each process is given some proportion of memory and that’s what it gets. Page replacement is done from w/in that segment.

- **Pros:** Fair
- **Cons:** Fair whether or not this makes sense. (Big processes and small processes get the same resources.) Could cause thrashing when other segments are hardly used.

**global** Each page replacement is taken from all pages.

- **Pros:** gives better balance and tuning over growing and shrinking working sets.
- **Cons:** Might allow communication between processes that shouldn’t. Also, a process’s working set may shrink too small. (1 page may not be enough to execute an instruction.)

How to determine how many pages to give to each process?
- **Historical:** Measure working set by number of pages referenced within some threshold time.
- **Page fault Frequency:** Notice the frequency of page faults. If it is high, allocate more pages. If low, take some back. If all processes are high, swap somebody out.

22.3.3 Page size

need to balance *internal fragmentation* against paging overhead. (A big page costs about as much to load as a small one because of seek and rotation latencies.)

22.3.4 VM Tricks

- Fork and exec
- Shared text (read only)
- Shared memory.
- Memory mapped file IO

22.3.5 Minix

What does Minix do?

`chmem` `[+][-]=` *amount file*
22.4 Filesystems

The last thing we need to make a computer system useful is large-scale storage. This brings us to filesystems. If a disk is just a large, slow, random access memory (it is), why do we handle files in a different way.

Filesystems are just like memories, but:

- very large
- long term (persistent)
- Often files (and disks) must be shared

So... what is a file anyway? A collection of data indexable somehow

- named?
- typed vs. untyped
- stream based or record based
- random access vs. sequential access (why would it matter)

22.4.1 Some thoughts on filesystem organization

What do we need, and how do we do it? (and where did we put it?)

We have to reconcile a clean, abstract view of storage with the realities of blocks on a disk.

22.5 Filesystem Implementation Considerations

resource allocation How to best use the available resources

performance

reliability (consistency)

These are explored in the following three subsections

22.5.1 Resource Allocation: Disk Management

The concerns here are:

File structure (Sequential, random access, record based, stream based?)

Disk block size (Fragmentation vs. overhead)

Allocation Management

- File structure

There are different strategies for allocating blocks to files:
| contiguous          | – pro Great performance from contiguous reads |
|                    | – con requires knowledge of file size in advance (or copy) |
|                    | – con fragmentation. |
|                    | – con de-frags/copies are expensive |
| linked list        | – pro no fragmentation, simple |
|                    | – con slow: each read must be completed before another can start. |
|                    | – con peculiar block size \((2^n - 1)\) |
|                    | – con one bad block will lose the rest of the file. (similar problem w/compressed FS) |
| indexed            | Solves boths problems. Random access on the disk, but allows readahead. (Indexing can be done in memory.) |
| i-node index       | a good variant—indexes the file, not the disk |

### 22.6 Filesystems

The last thing we need to make a computer system useful is large-scale storage.

This brings us to filesystems. If a disk is just a large, slow, random access memory (it is), why do we handle files in a different way?

Filesystems are just like memories, but:

- very large
- long term (persistent)
- Often files (and disks) must be shared

So…what is a file anyway? A collection of data indexable somehow

- named?
- typed vs. untyped
- stream based or record based
- random access vs. sequential access (why would it matter)

#### 22.6.1 Some thoughts on filesystem organization

What do we need, and how do we do it? (and where did we put it?)

We have to reconcile a clean, *abstract* view of storage with the realities of blocks on a disk.
22.7 Filesystem Implementation Considerations

resource allocation How to best use the available resources

performance

reliability (consistency)

These are explored in the following three subsections
23 Lecture: Intro to Filesystems

Outline:
- Announcements
- Filesystems
  - Some thoughts on filesystem organization
- Filesystem Implementation Considerations
  - These are explored in the following three subsections
    - Unix Filsystem Structure
      - Resource Allocation: Disk Management
    - Foreshadowing: The Minix Filesystem
      - *Partitions and Subpartitions
      - *Links
      - *Data structures

23.1 Announcements

- Coming attractions:
  
<table>
<thead>
<tr>
<th>Event</th>
<th>Subject</th>
<th>Due Date</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>asgn5</td>
<td>minget and minls</td>
<td>Wed Jun 5</td>
<td>23:59</td>
</tr>
<tr>
<td>asgn6</td>
<td>Yes, really</td>
<td>Fri Jun 7</td>
<td>23:59</td>
</tr>
<tr>
<td>final</td>
<td>stuff</td>
<td>Sat Jun 8</td>
<td>10:10 (in 03-201)</td>
</tr>
</tbody>
</table>

  Use your own discretion with respect to timing/due dates.

- Asgn5
  
  - `minget` and `minls` must run outside of minix
  - `tryAsgn5`
  - Test Images
    
    * There are many disk images in: `~pn-cs453/Given/Asgn5/Images`
    * `Notes` describes what’s in each.
    * Remember that `minix` types (e.g. `ino_t`) may be sized differently from their `linux` counterparts. Use `stdint.h` types.
  
  - `minls -v -v`
    
    - sectors, blocks, and zones, oh! my!

- Check for error return values!

- `pwdauth`

23.2 Filesystems

The last thing we need to make a computer system useful is large-scale storage.

This brings us to filesystems. If a disk is just a large, slow, random access memory (it is), why do we handle files in a different way?

Filesystems are just like memories, but:

- very large
• long term (persistent)

• Often files (and disks) must be shared

So... what is a file anyway? A collection of data indexable somehow

• named?

• typed vs. untyped

• stream based or record based

• random access vs. sequential access (why would it matter)

23.2.1 Some thoughts on filesystem organization

What do we need, and how do we do it? (and where did we put it?)

We have to reconcile a clean, abstract view of storage with the realities of blocks on a disk.

23.3 Filesystem Implementation Considerations

resource allocation How to best use the available resources

performance

reliability (consistency)

These are explored in the following three subsections

23.3.1 Resource Allocation: Disk Management

The concerns here are:

File structure (Sequential, random access, record based, stream based?)

Disk block size (Fragmentation vs. overhead)

Allocation Management

• File structure

There are different strategies for allocating blocks to files:
23.4 Foreshadowing: The Minix Filesystem

- Remember: These types are as defined by Minix, not any other *-ix.
- Also: struct __attribute__((__packed__)) partition

Partitions and Subpartitions

Any disk can have up to four primary partitions. The information for these partitions is stored in the partition table, located at address 0x1BE on the disk. The structure of a partition table entry is given in Figure 34. The fields we will be interested in are: type, because it says whether this is a MINIX partition, and lFirst and size. lFirst gives the first absolute9 sector number of the partition and lFirst + size − 1 gives last.

Note, for the CHS form of the partition description only the bottom 6 bits of the sector field are the sector. The top two bits of the sector field are prepended to the cylinder to form a 10-bit cylinder value.

A valid partition table contains a signature: 0x55 in byte 510, and 0xAA in byte 511. You must check the partition table for validity before proceeding.

Each partition is like a complete disk of its own and could include a (sub)partition table of its own, with the same structures at the same positions relative to the beginning of the containing partition. Once you have chased down the right partition, it’s necessary to navigate the filesystem.

See Figures 35–36.

---

9That is, even for the subpartition table, the sector numbers are relative to the beginning of the disk, not the partition.
<table>
<thead>
<tr>
<th>Type</th>
<th>Name</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>uint8_t</td>
<td>bootind</td>
<td>Boot magic number (0x80 if bootable)</td>
</tr>
<tr>
<td>uint8_t</td>
<td>start_head</td>
<td>Start of partition in CHS</td>
</tr>
<tr>
<td>uint8_t</td>
<td>start_sec</td>
<td></td>
</tr>
<tr>
<td>uint8_t</td>
<td>start_cyl</td>
<td></td>
</tr>
<tr>
<td>uint8_t</td>
<td>type</td>
<td>Type of partition (0x81 is MINIX)</td>
</tr>
<tr>
<td>uint8_t</td>
<td>end_head</td>
<td>End of partition in CHS</td>
</tr>
<tr>
<td>uint8_t</td>
<td>end_sec</td>
<td></td>
</tr>
<tr>
<td>uint8_t</td>
<td>end_cyl</td>
<td></td>
</tr>
<tr>
<td>uint32_t</td>
<td>lFirst</td>
<td>First sector (LBA addressing)</td>
</tr>
<tr>
<td>uint32_t</td>
<td>size</td>
<td>size of partition (in sectors)</td>
</tr>
</tbody>
</table>

Figure 34: Partition table entry

<table>
<thead>
<tr>
<th>MBR</th>
<th>Partition 1</th>
<th>Partition 2</th>
<th>Partition 3</th>
<th>Unused</th>
</tr>
</thead>
</table>

Figure 35: Anatomy of a disk

Figure 36: Anatomy of a filesystem
Links

A *nix filesystem is held together by links.

- A directory is a regular file containing links
  - Sort of. A directory has special restrictions on who can write it.
- A link is an entry in a directory associating a name with an i-node
- All paths start with the root directory. (inode 1)
- Inodes index file zones (numbering starts at 1)
- Zones are “out there” on the disk, numbering starts at the beginning of the filesystem
  - Zone zero is special (it’s a hole)

Data structures

See Figures 37–40.

```
struct superblock {
    /* Minix Version 3 Superblock */
    /* this structure found in fs/super.h */
    /* in minix 3.1.1 */
    /* on disk. These fields and orientation are non-negotiable */
    uint32_t ninoide; /* number of inodes in this filesystem */
    uint16_t pad1; /* make things line up properly */
    int16_t i_blocks; /* # of blocks used by inode bit map */
    int16_t z_blocks; /* # of blocks used by zone bit map */
    uint16_t firstdata; /* number of first data zone */
    int16_t log_zone_size; /* log2 of blocks per zone */
    int16_t pad2; /* make things line up again */
    uint32_t max_file; /* maximum file size */
    uint32_t zones; /* number of zones on disk */
    int16_t magic; /* magic number */
    int16_t pad3; /* make things line up again */
    uint16_t blocksize; /* block size in bytes */
    uint8_t subversion; /* filesystem sub-version */
}
```

Figure 37: Minix superblock

144
```c
#define DIRECT_ZONES 7

struct inode {
    uint16_t mode;  /* mode */
    uint16_t links; /* number or links */
    uint16_t uid;
    uint16_t gid;
    uint32_t size;
    int32_t atime;
    int32_t mtime;
    int32_t ctime;
    uint32_t zone[DIRECT_ZONES];
    uint32_t indirect;
    uint32_t two_indirect;
    uint32_t unused;
};

Figure 38: Minix inode
```

```c
#ifndef DIRSZ
#define DIRSZ 60
#endif

struct fileent {
    uint32_t ino;
    char name[DIRSZ];
};

Figure 39: Minix directory ent
```
/ * constants */
#define PTABLE_OFFSET 0x1BE
#define PMAGIC510 0x55
#define PMAGIC511 0xAA
#define MINIXPART 0x81

#define MIN_MAGIC 0x4d5a /* the minix magic number */
#define MIN_MAGIC_REV 0x5a4d /* the minix magic number reversed */
#define MIN_MAGIC_OLD 0x2468 /* the v2 minix magic number */
#define MIN_MAGIC_REV_OLD 0x6824 /* the v2 magic number reversed */

#define MIN_ISREG(m) (((m)&0170000)==0100000)
#define MIN_ISDIR(m) (((m)&0170000)==0040000)
#define MIN_IRUSR 0400
#define MIN_IWUSR 0200
#define MIN_IXUSR 0100
#define MIN_IRGRP 0040
#define MIN_IWGRP 0020
#define MIN_IXGRP 0010
#define MIN_IROTH 0004
#define MIN_IWOTH 0002
#define MIN_IXOTH 0001

/* we have an endian problem */

Figure 40: useful Minix numbers
24 Lecture: Filesystems, cont.

Outline:
- Announcements
- Filesystem Implementation Considerations
- From Last time: File structure
- Directory Structures
- Foreshadowing: The Minix Filesystem
  - Partitions and Subpartitions
  - Links
  - Data structures

24.1 Announcements

- Coming attractions:

<table>
<thead>
<tr>
<th>Event</th>
<th>Subject</th>
<th>Due Date</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>asgn5</td>
<td>minget and minls</td>
<td>Wed Jun 5</td>
<td>23:59</td>
</tr>
<tr>
<td>asgn6</td>
<td>Yes, really</td>
<td>Fri Jun 7</td>
<td>23:59</td>
</tr>
<tr>
<td>final</td>
<td>stuff</td>
<td>Sat Jun 8</td>
<td>10:10  (in 03-201)</td>
</tr>
</tbody>
</table>

Use your own discretion with respect to timing/due dates.

- minget and minls must run outside of minix
- minget and minls are the same program
- minget -v -v
- tryAsgn5 exists

24.2 Filesystem Implementation Considerations

resource allocation How to best use the available resources

performance

reliability (consistency)

24.3 From Last time: File structure

There are different strategies for allocating blocks to files:
| contiguous       | pro  | Great performance from contiguous reads |
|                 | con  | requires knowledge of file size in advance (or copy) |
|                 | con  | fragmentation. |
|                 | con  | de-frags/copies are expensive |
| linked list     | pro  | no fragmentation, simple |
|                 | con  | slow: each read must be completed before another can start. |
|                 | con  | peculiar block size ($2^n - 1$) |
|                 | con  | one bad block will lose the rest of the file. (similar problem w/compressed FS) |
| indexed         |      | Solves boths problems. Random access on the disk, but allows readahead. (Indexing can be done in memory.) |
| i-node index    |      | a good variant—indexes the file, not the disk |

Figure 41 shows the logical structure of a file in a minix system given 16-byte disk blocks and 4-byte zone numbers. (A 16-byte block, is, of course, ridiculous, but it fits on a page.)

### 24.4 Directory Structures

Now that we have a file system, how do we find a file in it?

- Flat structure?
- Hierarchical?

Also, what do we want in it:

- Everything?
- Just files?

Examples:

**CP/M** (CP/M == Control Program/Monitor, FWIW)

<table>
<thead>
<tr>
<th>Owner</th>
<th>Name</th>
<th>Extension</th>
<th>extent</th>
<th>reserved</th>
<th>block count</th>
<th>block numbers</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>8</td>
<td>3</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>16</td>
</tr>
</tbody>
</table>

Note
Figure 41: A minix file with 16-byte blocks and 4-byte zone numbers
• No precise byte count
• No directory hierarchy

**MS-DOS FAT-16**

<table>
<thead>
<tr>
<th>Name</th>
<th>Extension</th>
<th>attributes</th>
<th>reserved</th>
<th>Time</th>
<th>Date</th>
<th>First Block</th>
<th>Size</th>
</tr>
</thead>
</table>

Note

• Precise byte count
• Block No. is an index into a file allocation table containing a linked list.
• directory hierarchy: A file can be a directory

**UNIX**

Directories only associate names with i-node numbers. All other information is included in the i-node.

**MINIX**

Just like unix, but some details...

### 24.5 Foreshadowing: The Minix Filesystem

- Remember: These types are as defined by Minix, not any other *-ix.
- Also: `struct __attribute__((packed)) partition`

#### Partitions and Subpartitions

Any disk can have up to four *primary partitions*. The information for these partitions is stored in the *partition table*, located at address `0x1BE` on the disk. The structure of a partition table entry is given in Figure 42. The fields we will be interested in are: `type`, because it says whether this is a MINIX partition, and `lFirst` and `size`. `lFirst` gives the first absolute sector number of the partition and `lFirst + size − 1` gives last.

<table>
<thead>
<tr>
<th>Type</th>
<th>Name</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>uint8_t</code></td>
<td>bootind</td>
<td>Boot magic number (0x80 if bootable)</td>
</tr>
<tr>
<td><code>uint8_t</code></td>
<td>start_head</td>
<td>Start of partition in CHS</td>
</tr>
<tr>
<td><code>uint8_t</code></td>
<td>start_sec</td>
<td></td>
</tr>
<tr>
<td><code>uint8_t</code></td>
<td>start_cyl</td>
<td></td>
</tr>
<tr>
<td><code>uint8_t</code></td>
<td>type</td>
<td>Type of partition (0x81 is MINIX)</td>
</tr>
<tr>
<td><code>uint8_t</code></td>
<td>end_head</td>
<td>End of partition in CHS</td>
</tr>
<tr>
<td><code>uint8_t</code></td>
<td>end_sec</td>
<td></td>
</tr>
<tr>
<td><code>uint8_t</code></td>
<td>end_cyl</td>
<td></td>
</tr>
<tr>
<td><code>uint32_t</code></td>
<td>lFirst</td>
<td>First sector (LBA addressing)</td>
</tr>
<tr>
<td><code>uint32_t</code></td>
<td>size</td>
<td>size of partition (in sectors)</td>
</tr>
</tbody>
</table>

Figure 42: Partition table entry

---

That is, even for the subpartition table, the sector numbers are relative to the beginning of the disk, not the partition.
Note, for the CHS form of the partition description only the bottom 6 bits of the sector field are the sector. The top two bits of the sector field are prepended to the cylinder to form a 10-bit cylinder value.

A valid partition table contains a signature: 0x55 in byte 510, and 0xAA in byte 511. You must check the partition table for validity before proceeding.

Each partition is like a complete disk of its own and could include a (sub)partition table of its own, with the same structures at the same positions relative to the beginning of the containing partition. Once you have chased down the right partition, it’s necessary to navigate the filesystem.

See Figures 43-44.

<table>
<thead>
<tr>
<th>MBR</th>
<th>Partition 1</th>
<th>Partition 2</th>
<th>Partition 3</th>
<th>Unused</th>
</tr>
</thead>
</table>

![Diagram of disk anatomy](image)

**Figure 43: Anatomy of a disk**

<table>
<thead>
<tr>
<th>Boot</th>
<th>Superblock</th>
<th>i-map</th>
<th>z-map</th>
<th>i-table</th>
<th>directory and data blocks</th>
</tr>
</thead>
</table>

**Figure 44: Anatomy of a filesystem**

**Links**

A *nix filesystem is held together by links.

- A directory is a regular file containing links
  - Sort of. A directory has special restrictions on who can write it.
- A link is an entry in a directory associating a name with an i-node
- All paths start with the root directory. (inode 1)
- Inodes index file zones (numbering starts at 1)
- Zones are “out there” on the disk, numbering starts at the beginning of the filesystem
  - Zone zero is special (it’s a *hole*).

**Data structures**

See Figures 45-48.
struct superblock { /* Minix Version 3 Superblock
   * this structure found in fs/super.h
   * in minix 3.1.1
*/
    /* on disk. These fields and orientation are non-negotiable */
    uint32_t ninodes; /* number of inodes in this filesystem */
    uint16_t pad1;   /* make things line up properly */
    int16_t i_blocks; /* # of blocks used by inode bit map */
    int16_t z_blocks; /* # of blocks used by zone bit map */
    uint16_t firstdata; /* number of first data zone */
    int16_t log_zone_size; /* log2 of blocks per zone */
    int16_t pad2; /* make things line up again */
    uint32_t max_file; /* maximum file size */
    uint32_t zones; /* number of zones on disk */
    int16_t magic; /* magic number */
    int16_t pad3; /* make things line up again */
    uint16_t blocksize; /* block size in bytes */
    uint8_t subversion; /* filesystem sub-version */
};

Figure 45: Minix superblock

#define DIRECT_ZONES 7

struct inode {
    uint16_t mode; /* mode */
    uint16_t links; /* number or links */
    uint16_t uid;
    uint16_t gid;
    uint32_t size;
    int32_t atime;
    int32_t mtime;
    int32_t ctime;
    uint32_t zone[DIRECT_ZONES];
    uint32_t indirect;
    uint32_t two间接;
    uint32_t unused;
};

Figure 46: Minix inode
```c
#ifndef DIRSIZ
#define DIRSIZ 60
#endif

struct fileent {
    uint32_t ino;
    char     name[DIRSIZ];
};

Figure 47: Minix directory ent

/* constants */
#define PTABLE_OFFSET 0x1BE
#define PMAGIC510 0x55
#define PMAGIC511 0xAA
#define MINIXPART 0x81

#define MIN_MAGIC 0x4d5a /* the minix magic number */
#define MIN_MAGIC_REV 0x5a4d /* the minix magic number reversed */
#define MIN_MAGIC_OLD 0x2468 /* the v2 minix magic number */
#define MIN_MAGIC_REV_OLD 0x6824 /* the v2 magic number reversed */

#define MIN_ISREG(m) (((m)&0170000)==0100000)
#define MIN_ISDIR(m) (((m)&0170000)==0040000)
#define MIN_IRUSR 0400
#define MIN_IWUSR 0200
#define MIN_IXUSR 0100
#define MIN_IRGRP 0040
#define MIN_IWGRP 0020
#define MIN_IXGRP 0010
#define MIN_IROTH 0004
#define MIN_IWOTH 0002
#define MIN_IXOTH 0001

Figure 48: useful Minix numbers
```
25 Lecture: More Filesystems

Outline:
Announcements
Example: The Minix Filesystem
Directory Structures
Foreshadowing: The Minix Filesystem
  *Partitions and Subpartitions
  *Links
  *Data structures
Filesystem Implementation Considerations
Disk block Size
Disk block Management
Filesystem Performance

25.1 Announcements

- Coming attractions:

<table>
<thead>
<tr>
<th>Event</th>
<th>Subject</th>
<th>Due Date</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>asgn5</td>
<td>minget and minls</td>
<td>Wed</td>
<td>23:59</td>
</tr>
<tr>
<td>asgn6</td>
<td>Yes, really</td>
<td>Fri</td>
<td>23:59</td>
</tr>
<tr>
<td>final</td>
<td>stuff</td>
<td>Sat</td>
<td>10:10 (in 03-201)</td>
</tr>
</tbody>
</table>

Use your own discretion with respect to timing/due dates.

- Asgn5
  - tryAsgn5
  - minls -v -v
    - sectors, blocks, and zones, oh! my!

- Check for error return values!

- primitive != simple
  - not supported != not installed

- minget and minls must run outside of minix

- I hate to have to say it, but the solutions manual is not an acceptable source when doing Lab05.

- A note on asgn6

25.2 Directory Structures

Now that we have a file system, how do we find a file in it?

- Flat structure?
- Hierarchical?

Also, what do we want in it:
• Everything?
• Just files?

Examples:

**CP/M** *(CP/M == Control Program/Monitor, FWIW)*

```
1 8 3 1 2 1 16
Owner Name Extension extent reserved block count block numbers
```

Note

• No precise byte count
• No directory hierarchy

**MS-DOS FAT-16**

```
8 3 1 10 2 2 2 4
Name Extension attributes reserved Time Date First Block Size
```

Note

• Precise byte count
• Block No. is an index into a file allocation table containing a linked list.
• directory hierarchy: A file can be a directory

**UNIX**

Directories only associate names with i-node numbers. All other information is included in the i-node.

**MINIX**

Just like unix, but some details...

### 25.3 Foreshadowing: The Minix Filesystem

• Remember: These types are as defined by Minix, not any other *-ix.

• Also: `struct _attribute_ ((__packed__)) partition`

**Partitions and Subpartitions**

Any disk can have up to four *primary partitions*. The information for these partitions is stored in the *partition table*, located at address **0x1BE** on the disk. The structure of a partition table entry is given in Figure 49. The fields we will be interested in are: *type*, because it says whether this is a MINIX partition, and *lFirst* and *size*. *lFirst* gives the first absolute\(^\text{11}\) sector number of the partition and *lFirst + size − 1* gives last.

Note, for the CHS form of the partition description only the bottom 6 bits of the sector field are the sector. The top two bits of the sector field are prepended to the cylinder to form a 10-bit cylinder value.

A valid partition table contains a *signature*: **0x55** in byte 510, and **0xAA** in byte 511. You must check the partition table for validity before proceeding.

\(^\text{11}\)That is, even for the subpartition table, the sector numbers are relative to the beginning of the disk, not the partition.
<table>
<thead>
<tr>
<th>Type</th>
<th>Name</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>uint8_t</td>
<td>bootind</td>
<td>Boot magic number (0x80 if bootable)</td>
</tr>
<tr>
<td>uint8_t</td>
<td>start_head</td>
<td>Start of partition in CHS</td>
</tr>
<tr>
<td>uint8_t</td>
<td>start_sec</td>
<td></td>
</tr>
<tr>
<td>uint8_t</td>
<td>start_cyl</td>
<td></td>
</tr>
<tr>
<td>uint8_t</td>
<td>type</td>
<td>Type of partition (0x81 is MINIX)</td>
</tr>
<tr>
<td>uint8_t</td>
<td>end_head</td>
<td>End of partition in CHS</td>
</tr>
<tr>
<td>uint8_t</td>
<td>end_sec</td>
<td></td>
</tr>
<tr>
<td>uint8_t</td>
<td>end_cyl</td>
<td></td>
</tr>
<tr>
<td>uint32_t</td>
<td>lFirst</td>
<td>First sector (LBA addressing)</td>
</tr>
<tr>
<td>uint32_t</td>
<td>size</td>
<td>size of partition (in sectors)</td>
</tr>
</tbody>
</table>

Figure 49: Partition table entry

Each partition is like a complete disk of its own and could include a (sub)partition table of its own, with the same structures at the same positions relative to the beginning of the containing partition. Once you have chased down the right partition, it’s necessary to navigate the filesystem. See Figures 50–51.

---

**Links**

A *nix filesytem is held together by links.

- A directory is a regular file containing links
  - Sort of. A directory has special restrictions on who can write it.
- A link is an entry in a directory associating a name with an i-node
- All paths start with the root directory. (inode 1)
- inodes index file zones (numbering starts at 1)
- Zones are “out there” on the disk, numbering starts at the beginning of the filesystem
  - Zone zero is special (it’s a hole)

Data structures

See Figures 52–55.

```c
struct superblock {
    /* Minix Version 3 Superblock
    * this structure found in fs(super.h
    * in minix 3.1.1
    */
    /* on disk. These fields and orientation are non-negotiable */
    uint32_t ninodes; /* number of inodes in this filesystem */
    uint16_t pad1;   /* make things line up properly */
    int16_t i_blocks; /* # of blocks used by inode bit map */
    int16_t z_blocks; /* # of blocks used by zone bit map */
    uint16_t firstdata; /* number of first data zone */
    int16_t log_zone_size; /* log2 of blocks per zone */
    int16_t pad2;   /* make things line up again */
    uint32_t max_file; /* maximum file size */
    uint32_t zones; /* number of zones on disk */
    int16_t magic; /* magic number */
    int16_t pad3;   /* make things line up again */
    uint16_t blocksize; /* block size in bytes */
    uint8_t subversion; /* filesystem sub-version */
}
```

Figure 52: Minix superblock
#define DIRECT_ZONES 7

struct inode {
  uint16_t mode;  /* mode */
  uint16_t links; /* number or links */
  uint16_t uid;
  uint16_t gid;
  uint32_t size;
  int32_t atime;
  int32_t mtime;
  int32_t ctime;
  uint32_t zone[DIRECT_ZONES];
  uint32_t indirect;
  uint32_t two间接;
  uint32_t unused;
};

Figure 53: Minix inode

#ifndef DIRSIZ
#define DIRSIZ 60
#endif

struct fileent {
  uint32_t ino;
  char name[DIRSIZ];
};

Figure 54: Minix directory ent
/* constants */
#define PTABLE_OFFSET 0x1BE
#define PMAGIC510 0x55
#define PMAGIC511 0xAA
#define MINIXPART 0x81

#define MIN_MAGIC 0x4d5a    /* the minix magic number */
#define MIN_MAGIC_REV 0x5a4d /* the minix magic number reversed*/
#define MIN_MAGIC_OLD 0x2468 /* the v2 minix magic number */
#define MIN_MAGIC_REV_OLD 0x6824 /* the v2 magic number reversed*/

#define MIN_ISREG(m) (((m)&0170000)==0100000) /* we have an endian problem */
#define MIN_ISDIR(m) (((m)&0170000)==0040000)
#define MIN_IRUSR 0400
#define MIN_IWUSR 0200
#define MIN_IXUSR 0100
#define MIN_IRGRP 0040
#define MIN_IWGRP 0020
#define MIN_IXGRP 0010
#define MIN_IROTH 0004
#define MIN_IWOTH 0002
#define MIN_IXOTH 0001

Figure 55: useful Minix numbers
25.4 Filesystem Implementation Considerations

resource allocation How to best use the available resources

performance

reliability (consistency)

25.5 Disk block Size

How big should it be: tradeoff between index size and seek time vs. internal fragmentation.
   Median Unix file size is 1k.
   DOS FAT filesystem: Fixed number of blocks regardless of size. (FAT-16 has 16 bits of block address, so 64k blocks. This means that for a 1GB filesystem, the block size is 16k!)
   Note: Where’d zones come from?

25.6 Disk block Management

Somehow we must keep track of allocated and free blocks:

linked-list faster, but uses more disk space

bitmaps less space, but slower allocation if not held in memory

25.7 Filesystem Performance

- Clever disk allocation — clever layout can make for more efficient disk accesses. (e.g., i-nodes to the middle, blocks allocated in rings, etc.)

- Buffer cache—with clever caching, one may never (hardly ever) have to access the disk. This risks inconsistencies in case of failure, however.
   Which block to replace? Depends. Misses are infrequent, so LRU is possible, but undesirable (because the most important things are the most likely to be lost)

- Log Structured Filesystems (Rosenblum and Ousterhour 1991)
   Don’t write back dirty blocks immediately, just accumulate diffs.

- Journaling Filesystems (NTFS, ext3)
   1. Write what you’re going to do
   2. Do it
   3. Write that you’ve done it. (or erase the entry)

Now, if there’s a crash, you can just reissue any commands that didn’t complete.
26 Lecture: Filesystems, cont.

Outline:
- Announcements
- We talked a lot about asgn5
- Filesystem Performance
- Filesystem Reliability
  - Filesystem Consistency (fsck)

26.1 Announcements
- Coming attractions:

<table>
<thead>
<tr>
<th>Event</th>
<th>Subject</th>
<th>Due Date</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>asgn5</td>
<td>minget and minls</td>
<td>Wed Jun 5 23:59</td>
<td></td>
</tr>
<tr>
<td>asgn6</td>
<td>Yes, really</td>
<td>Fri Jun 7 23:59</td>
<td></td>
</tr>
<tr>
<td>final</td>
<td>stuff</td>
<td>Sat Jun 8 10:10</td>
<td>(in 03-201)</td>
</tr>
</tbody>
</table>

Use your own discretion with respect to timing/due dates.

- Remember to turn in lab5
- Remember to turn in asgn5
- Remember to do asgn6
- Late day reminder
- Finals week office hours: Tuesday 9–11, or email me.

26.2 We talked a lot about asgn5
- `getopt(3)` is your friend
- CHS in the partition table (top two bits of sector become MSBs of the cylinder)
- The disk is a file (`read(2)`, `lseek(2)`, etc. Or stdio)
- Remember: These types are as defined by Minix, not any other *-ix.
- Also: `struct __attribute__ ((__packed__)) partition`
- Don’t get fancy. Use the right types.

26.3 Filesystem Performance
- Clever disk allocation — clever layout can make for more efficient disk accesses. (e.g., i-nodes to the middle, blocks allocated in rings, etc.)
- Buffer cache — with clever caching, one may never (hardly ever) have to access the disk. This risks inconsistencies in case of failure, however.
  Which block to replace? Depends. Misses are infrequent, so LRU is possible, but undesirable (because the most important things are the most likely to be lost)
• Log Structured Filesystems (Rosenblum and Ousterhout 1991)
  Don’t write back dirty blocks immediately, just accumulate diffs.

• Journaling Filesystems (NTFS, ext3)
  1. Write what you’re going to do
  2. Do it
  3. Write that you’ve done it. (or erase the entry)

Now, if there’s a crash, you can just reissue any commands that didn’t complete.

26.4 Filesystem Reliability
Which strikes more fear into your heart: a blown CPU or a blown hard drive?
The PC can be replaced—it’s just time, money, and aggravation—but what about that customer list?

• Backups—don’t leave home without one.

• RAID—1988

  (SIGMOD == Special Interest Group for Management Of Data)
  Why aren’t we all doing this?

• Limit Damage sync(1,2), update keep damage to within 30s.

• Consistency checks—keep your filesystem in tip-top shape. See below.

26.4.1 Filesystem Consistency (fsck)
Every “reasonable” filesystem has some method of monitoring its own self-consistency. (Even chkdsk).

• Keeping track of disk blocks: No matter what allocation policy is used for allocating disk blocks, it is necessary to keep track of which ones have been allocated:
  
  linked list keep a list of blocks that list unallocated blocks. When you need one, search the list for an appropriate amount of space.
  
  bitmap Do the same thing with a bitmap. Uses less memory, but may require more searching as the disk fills up.

  do the same thing for i-nodes if your filesystem is such as has them.

• checking them for consistency:
1. create an array of counters for allocated and free blocks:

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td>allocated:</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>free:</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

2. Sweep through the filesystem and free list and increment the appropriate counter each time a block is found.

If all goes well, each block will be found in one of the lists.

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td>allocated:</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>free:</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

If not, there is a problem:

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td>allocated:</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>2</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>free:</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Possible problems:

<table>
<thead>
<tr>
<th>problem</th>
<th>solution</th>
</tr>
</thead>
<tbody>
<tr>
<td>missing block</td>
<td>place in free list</td>
</tr>
<tr>
<td>allocated block in free list</td>
<td>remove from free list</td>
</tr>
<tr>
<td>block allocated twice</td>
<td>copy the block to a free one and hope things work out ok.</td>
</tr>
</tbody>
</table>

3. Do the same thing for reference counts from directories to i-nodes.

Possible problems:

<table>
<thead>
<tr>
<th>problem</th>
<th>solution</th>
</tr>
</thead>
<tbody>
<tr>
<td>referenced i-node has incorrect link count</td>
<td>correct the count</td>
</tr>
<tr>
<td>unreferenced i-node has nonzero link count</td>
<td>lost+found</td>
</tr>
</tbody>
</table>
27 Lecture: Filesystem Wrapup

Outline:
- Announcements
- Filesystems, one more time...
- How does NTFS Work
  - Problems
  - Basics
  - Structure
- Next Time

27.1 Announcements
- Coming attractions:

<table>
<thead>
<tr>
<th>Event</th>
<th>Subject</th>
<th>Due Date</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>asgn5</td>
<td>minget and minls</td>
<td>Wed</td>
<td>Jun 5 23:59</td>
</tr>
<tr>
<td>asgn6</td>
<td>Yes, really</td>
<td>Fri</td>
<td>Jun 7 23:59</td>
</tr>
<tr>
<td>final</td>
<td>stuff</td>
<td>Sat</td>
<td>Jun 8 10:10 (in 03-201)</td>
</tr>
</tbody>
</table>

Use your own discretion with respect to timing/due dates.

- Memory != disk
- Evaluations: do ’em, but actually at the end.
- Note: Saturday’s final is not in the usual place.
- Remember to do asgn5 and asgn6 (zero and fifteen (one secret), so far).

27.2 Filesystems, one more time...

27.3 How does NTFS Work

The New Technology File System was designed for Windows NT (duh) to overcome some of the weaknesses of FAT-16 and FAT-32 which were becoming too limiting on large disks.

Consider:
- a 1TB filesystem has $2^{40} = 1099511627776$ bytes

- a 16-bit block index (FAT-16) allowed $2^{16} = 65536$ disk blocks, for a minimum block size of $2^{24} = 16$MB

- The 32-bit block index (FAT-32) allowed $2^{32} = 4294967296$ disk blocks for a minimum of 256B on the same system.

But, this, too, is getting close to wearing out. A 4TB disk would have a 1KB min block size, is a 16TB disk unthinkable? (on Amazon for approx $1,200).
27.3.1 Problems
Among others:

- Size
- Security
- collation order (case?)

27.3.2 Basics

- sectors are still 512B
- sectors are gathered into *clusters* (blocks) Size 512B–64KB depending on volume size
- the default cluster size is 4kB
- file names are limited to 255 characters
- File names are unicode (ϕιλǫ) and support case-sensitivity (collation, etc.)
  (And that’s just Greek. Consider adding Hebrew, Arabic, Chinese, etc.)
- Path names are limited to 32,767 characters (or 64KB with a null-terminator)
- NTFS uses 64-bit disk addresses

27.3.3 Structure

**Master File Table (MFT)**

- Is itself a file, and can be stored anywhere on the disk
- Contains a 1024-byte entry for every file on the system
- Allocated in large hunks to limit fragmentation (initial reservation is approx. 12.5% of the disk to allow for growth)
- A file is a collection of *attributes*
- Each MFT record is 1024 bytes (up to a maximum of $2^{48}$ records)
- If all attributes don’t fit, more MFT records are allocated (like CP/M)
- The first 16 entries are reserved for metadata. See Figure 56 (taken from Figure 11-41 in Tananbaum’s Modern Operating Systems)
- Initially indexed in boot sector
<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Standard information</td>
<td>Flags, POSIX times, etc.</td>
</tr>
<tr>
<td>File name</td>
<td>unicode name, may be repeated</td>
</tr>
<tr>
<td>Security descriptor</td>
<td>(obsolete)</td>
</tr>
<tr>
<td>Attribute List</td>
<td>Location of other MFT records</td>
</tr>
<tr>
<td>Object ID</td>
<td>64-bit file identifier unique to this volume</td>
</tr>
<tr>
<td>Reparse point</td>
<td>used for mounting and symlinks</td>
</tr>
<tr>
<td>Volume Name</td>
<td>volume name (used only in $Volume)</td>
</tr>
<tr>
<td>Volume Information</td>
<td>volume version (used only in $Volume)</td>
</tr>
<tr>
<td>Index root</td>
<td>used for directories</td>
</tr>
<tr>
<td>Index allocation</td>
<td>used for very large directories</td>
</tr>
<tr>
<td>Bitmap</td>
<td>used for very large directories</td>
</tr>
<tr>
<td>Logged utility stream</td>
<td>Controls logging to $LogFile</td>
</tr>
<tr>
<td>Data</td>
<td>stream data (may be repeated)</td>
</tr>
</tbody>
</table>

Figure 57: The NTFS Attributes
Attributes

- Located in each MFT record
- 16 attribute types
- Attributes that fit in the record itself are called resident
- Attributes that are stored in other disk blocks are called nonresident
- A file's data is just another (repeatable) attribute
  (Originally introduced so NTFS servers could support MacIntosh clients)
- Valid attributes are seen in figure 57 (also “borrowed” from Tanenbaum)

Data allocation

- There's a file allocation bitmap
- Blocks (clusters) are allocated in runs whenever possible.
- Holes are supported

Data stream records look like:

<table>
<thead>
<tr>
<th>header</th>
<th>run 1</th>
<th>run 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>offset, end + 1</td>
<td>block, length</td>
<td>block, length</td>
</tr>
</tbody>
</table>

The header contains the offset of the first block in the file and the offset of the first block that this run doesn’t cover (thus end + 1). Each run consists of a block (cluster) number and a length. The sum of the lengths of the run records had better add up to end + 1 – start.

E.g. a five block file in two runs:

| 0,5 | 3,3 | 10,2 |

For to support holes, you add more data stream records. An 11-block file with a hole:

| 0,5 | 3,3 | 10,2 | 10,11 | 12,1 |

Really big (or fragmented) files

- Large directories are real data structures.

Compression?

- in 16 block chunks

27.4 Next Time

- Bring questions/discussion items (Last chance . . .)
- Official Class Photo
28 Lecture: Security/Wrapping Up

Outline:
Announcements
Final Exam
Structure
Wrapup
  What are Operating Systems?
  Major subject areas for this course
  Why is this relevant to you?
Official Class Photo
  What do you want to talk about?
And now for something completely different: Security
What it is
Where do vulnerabilities come from?
Real nature of the work: That’s funny
What to do
Demonstrations
Other useful topics
  VM Tricks
Floating Point

28.1 Announcements

- Coming attractions:

<table>
<thead>
<tr>
<th>Event</th>
<th>Subject</th>
<th>Due Date</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>asgn5</td>
<td>minget and minls</td>
<td>Wed Jun 5</td>
<td>23:59</td>
</tr>
<tr>
<td>asgn6</td>
<td>Yes, really</td>
<td>Fri Jun 7</td>
<td>23:59</td>
</tr>
<tr>
<td>final</td>
<td>stuff</td>
<td>Sat Jun 8</td>
<td>10:10 (in 03-201)</td>
</tr>
</tbody>
</table>

Use your own discretion with respect to timing/due dates.

- I’ll plan on being at at least the beginning of lab Friday.
- Finals Week Office Hours
  Monday 10:00am–noon
- Remember to do asgn5 and asgn6 (two and five, so far).
- No class next time.
- Plans:
  - Answer questions
  - Class picture
- **Note:** Saturday’s final is not in the usual place.
- Don’t confuse memory and disk
28.2 Final Exam

28.2.1 Structure

Final: Chapters 1–5.
- short answer for breadth
- longer questions for depth

Emphasis on understanding: I might ask you what, but I’m more likely to ask why?, how?, or what if?.

28.3 Wrapup

28.3.1 What are Operating Systems?

resource allocation/performance/abstraction/reliability/tradeoffs

It all comes down to balancing:

1. Resource allocation
   - How to make the most of what you have
   - How to keep different processes from conflicting
   - How to make more than you have (faking it...)

2. Performance
   - Tuning the system for good performance while still remaining correct
   - scheduling, synchronization, resource allocation

3. Abstraction
   - Complexity management is key

4. Reliability
   - In the face of accident
   - In the face of malice

5. Tradeoffs
   *Faster, better, cheaper, choose two.*

Sounds like computer science, doesn’t it?

28.3.2 Major subject areas for this course

**OS Structure and History** What are the issues w/OS. Why? What are they (and what are they not (almost everything))

**Processes** What is a process, what do they need, how are they scheduled? What are the pitfalls (deadlock?)

**synchronization methods**
**Input/Output** How does an OS communicate with the outside world (and how do device drivers work)?

**Memory Systems** How to solve the problem of multiple programs at once and not enough memory:

- Swapping,
- segmentation,
- virtual memory systems,
- paging algorithms.

**Filesystems** How to organize a large amount of persistent data:

- resource allocation
- performance
- consistency

**Security** Think about what's going on.

### 28.3.3 Why is this relevant to you?

You're not going to go out of here and write operating systems, so why study them:

- you will write large, concurrent systems
- all the principles here affect how programs behave and how they should be written
- consciousness of these issues will make you a better computer scientist; possibly a better human being. :)

### 28.4 Official Class Photo

In which this class is immortalized on the web.

### 28.4.1 What do you want to talk about?
28.5 And now for something completely different: Security

Security is not an add-on feature, it must be built in. It is a state of mind and a design principle. (In fact, we’ve been talking about it all along.)

28.6 What it is

...and nothing else.

We’re really talking about Information Assurance.

- Requires looking at things differently. A thing may be exactly what it seems...and something else.
  - buffer overflow
  - drawer of old souveniers, etc.

- Computer security is System security (Bad guys don’t have to play by the rules.) Ex:
  - The Gordian Knot
  - Indiana Jones and the pistol
  - Fights in New Jersey?
  - the Maginot line
  - stealing a whole ATM
  - Manufacture of RFID passports outsourced

<table>
<thead>
<tr>
<th>Principle of Easiest Penetration (Pfleeger)</th>
</tr>
</thead>
<tbody>
<tr>
<td>An intruder must be expected to use any available means of penetration. The penetration may not necessarily be by the most obvious means, nor is it necessarily the one against which the most solid defense has been installed.</td>
</tr>
</tbody>
</table>

- Unfortunately, the good guys do have to play by the rules (this is harder)

- The threats are real (though I shouldn’t have to tell y’all this)
  - 11/08 Acroread vulnerability: Javascript buffer overflow
  - New RFID Passports?
    Contains in digital form (RFID):
    * date of birth,
    * gender,
    * birthplace
    * issuance and expiration dates
    * photograph
    Outsourced to Thailand?

- There isn’t a hacker behind every rock and tree, but there is behind every other rock and tree.
• By the way, what are we protecting?

Think in terms of three security Services:

<table>
<thead>
<tr>
<th>confidentiality</th>
<th>will those things that are supposed to be kept secret in fact be kept secret?</th>
</tr>
</thead>
<tbody>
<tr>
<td>integrity</td>
<td>Can we rely on the validity of the information provided.</td>
</tr>
<tr>
<td>availability</td>
<td>Will we be able to get to it when we need it.</td>
</tr>
</tbody>
</table>

One must think not in terms of how things are supposed to be used, but how they can fail.

Consider:

[…] Security engineering involves making things not happen. It involves figuring out how things fail, and then preventing those failures.

In many ways this is similar to safety engineering. Safety is another engineering requirement that isn’t simply a “feature.” But safety engineering involves making sure things do not fail in the presence of random faults: it’s about programming Murphy’s computer, if you will. Security engineering involves making sure things do not fail in the presence of an intelligent and malicious adversary who forces faults at precisely the worst time and in precisely the worst way. Security engineering involves programming Satan’s computer.

And Satan’s computer is hard to test.


Complete essay available at:

http://www.counterpane.com/crypto-gram-9911.html#WhyComputersareInsecure

So, with regard regard to each of CIA, we want to look at:

• Threats circumstances that have the potential to cause harm (“badness”)
  This is intentionally left vague: value will vary from system to system and environment.

• Vulnerabilities are weaknesses in the system that could be exploited to cause loss or harm

• Controls (or countermeasures) are actions, devices, techniques, etc., that remove or reduce a vulnerability.
  Controls need to be proportional. It’s only necessary to mitigate the vulnerability to the point where it’s not worth bothering. Anything else is wasted. (e.g. Greg’s bike)

Remember, systems are vulnerable where they are weak (not where we expect attacks).

<table>
<thead>
<tr>
<th>Principle of Easiest Penetration (Pfleeger)</th>
</tr>
</thead>
<tbody>
<tr>
<td>An intruder must be expected to use any available means of penetration. The penetration may not necessarily be by the most obvious means, nor is it necessarily the one against which the most solid defense has been installed.</td>
</tr>
</tbody>
</table>

Example:

• xkcd example
• Small: U.C.D.
• Large: Maginot line
28.7  Where do vulnerabilities come from?

- Careless programmers hall of shame: the dumbest mistake in the world, buffer overflow.
  - `gets()`
  - `strcat()`
  - `strcpy()`
  - `scanf()`

Don’t let this happen to you.
Moral: Don’t ever assume anything about the data you are receiving or the environment.

- Careless sysadmins/designers
  - writeable `/etc/passwd`
  - writeable root directory

- Poor/sloppy design
  - Bad design: Dumb VM tricks: TENEX password flaw.
    Combine:
    * Password protection on individual files.
    * User-visible paging statistics.
    * Short-circuit password evaluation.
  - old (possibly apocryphal) vulnerability in `su`

  **The setup** `su` assumed that if it was unable to open the passwd file, that the system was experiencing a catastrophic failure and allowed root access.

  **The exploit** open random files until the file descriptor table fills up and `exec()` `su`...

  - `mkdir`
    `mkdir` used to be a `mknod` followed by a `chown`. On a slow system there would be a race condition that would allow a user to capture ownership of any file:

    \begin{tabular}{|l|l|}
    \hline
    mkdir & attack \\
    \hline
    mknod foo & rm foo \\
    chown user foo & ln -s /etc/passwd foo \\
    \hline
    \end{tabular}

  - root mail exploit

- Then there are dumb users:
  - my passwd is “pizza”
  - it’s on the bulletin board
  - it’s always logged in
  - social engineering.

Remember, systems are vulnerable where they are weak (not where we expect attacks).
28.8 Real nature of the work: That’s funny

- sonic attack?

28.9 What to do

- **Principle of Least Privilege** Don’t ever use more privilege for any operation that is necessary.
  
  Consider role-based mechanisms vs. only root/user

- **Check parameters and enforce boundaries** Make sure that failures cannot propagate.
  
  - Internal: Consider the tty driver’s behavior:
    
    ```c
    if (numap(m_ptr->PROC_NR, (virtbytes) m_ptr->ADDRESS, m_ptr->COUNT) == 0) {
      r = EFAULT;
    } else {
    ...
    }
    ```

  - External: See Fig 58
  
  - If I gave you a gun and told you it was unloaded, would you believe me?
  
    - Check return values: Matt B.’s example

- **Do something reasonable** (Principle of least astonishment)

- **Clear memory pages/disk blocks** Information is persistent even when “undefined”.

- **Consider Security/Usability Tradeoffs** (If a PC is buried in concrete, it’s more secure, but decidedly less useful.)

28.10 Demonstrations

(Postponed due to uncooperative demo platform.)

- Stack smashing

- \%n conversion
Principles:

1. **Principle of Least Privilege (f)**
   Every program and every user of the system should operate using the least set of privileges necessary to complete the job.

2. **Principle of Economy of Mechanism (a)**
   Keep the design as simple and small as possible.

3. **Principle of Open Design (d)**
   Design should not be secret. The mechanisms should not depend on the ignorance of potential attackers, but rather on the possession of specific, more easily protected, keys or passwords.

4. **Principle of Complete Mediation (c)**
   Every access to every object must be checked for authority. Both direct access, and attempts to circumvent access control should be considered.

5. **Principle of Fail-Safe Defaults (b)**
   Base access decisions on permission rather than exclusion.

6. **Principle of Separation of Privilege (e)**
   When feasible, a protection mechanism that requires two keys to unlock it is more robust and flexible than one that allows access to the presenter of only a single key.

7. **Principle of Least Common Mechanism(g)**
   Minimize the amount of mechanism common to more than one user and depended on by all users.

8. **Principle of Psychological Acceptability (h)**
   (Pfeeger: “Ease of Use”) It is essential that the human interface be designed for ease of use, so that users routinely and automatically apply the protection mechanisms.