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</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.

- 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:
  `strcat()` is reentrant.
  `strtok()` is not.

12.2.1 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.

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()`ing)
2. When an IO interrupt occurs
3. When a timer interrupt occurs

12.2.3 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.
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** Sheduling, 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 1</th>
<th>immediate (so no queue)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Level 2</td>
<td>run to completion/block</td>
</tr>
<tr>
<td>Level 3</td>
<td>run to completion/block</td>
</tr>
<tr>
<td>Level 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:

\[
\begin{array}{l}
\text{Synchronous}\\
\text{send(dest, &message);} \\
\text{receive(source, &message);} \\
\text{send_rec(dest_src, &message);} \\
\text{Asynchronous}\\
\text{notify(dest);} \\
\end{array}
\]

Messages (other than \texttt{notify()}) communicated via \texttt{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),
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.*