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
### cpe453 Midterm

<table>
<thead>
<tr>
<th></th>
<th>Possible: 50.0</th>
<th>High: 50.0</th>
<th>96.0%</th>
<th>Low: 18.0</th>
<th>38.0%</th>
<th>Mean: 41.3</th>
<th>72.8%</th>
<th>Median: 45.0</th>
<th>76.0%</th>
<th>S.D.: 9.0</th>
<th>15.2%</th>
</tr>
</thead>
</table>

<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 \textit{within} the 20ms allotted for each process.
   (and sanity check. \((20\text{ms} + 5\text{ms}) \times 50 = 1.25s\))


   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 \textit{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. \textit{How} is it possible for a process to voluntarily become the OS again? (The mechanism is the same, the system call interfaces)

   Also, interrupt \textit{≠} 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_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 */
);

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’ :-) */
);

int sys_safecopyto (  
    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? `getnucrd(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: A very narrow hurricane has washed out all but one lane of the Lake Pontchartrain Causeway. 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 `enter_bridge(int direction)` with a pre-defined integer constant indicating the direction. This will be either `NORTH` or `SOUTH`. When it wants to leave, it calls `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 `enter_bridge()` and `exit_bridge()` and whatever auxiliary data and functions you may need. Be sure to explain briefly why your solution works.

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).

For more partial credit: produce a solution that guarantees that no car will have to wait forever to cross.

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):

**Solution:**

The first two partial credit levels above can be handled by a very simple solution that uses a single semaphore for exclusive access to the bridge. Exclusive access guarantees that there will be no accidents or deadlocks, and the semaphore’s own queueing mechanisms guarantee that nobody will have to wait forever. See below. A better solution is presented on page ??.

```c
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);
}
```

A semaphore-based solution to problem 1 that allows one car at a time.

---

7Note, this problem is too hard for a midterm, but it’s a good problem to consider, so I’m leaving it here.
8Lake 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.
The full solution is much trickier: To meet all three requirements, we must not allow traffic in one direction 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 before entering the bridge. If there are, we act as if they were already on the bridge and wait. The last car crossing in the current direction wakes up all the waiting cars on the other side.

The only subtle part if the solution 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, however, 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 blocking.

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) )
        /* waiting on the other side */
        onbridge = onbridge + 1;
        /* add ourselves to the bridge */
    else
        waiting[d] = waiting[d] + 1; /* otherwise, we have to wait. */
        willwait = 1;
    }
    UP(mutex);
    if ( willwait )
        /* HERE HERE HERE */
        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<waiting[other];i++)
        /* wake up the waiting cars */
        UP(waitsem[other]);
    }
    UP(mutex);