CSE 451: Operating Systems

Lab Section: Week 3
Today

• Last week’s quiz

• Project 2

• Scheduling
  (this will hopefully be useful for tomorrow’s quiz 😊)
Last week’s quiz

1) Define terms: (a) exception, (b) fault, (c) interrupt, (d) trap
   - see answer in slides from last week

Two reasons for losing points:
   • didn’t mention system calls!
   • these are different from Java exceptions ...
Last week’s quiz:
our “exceptions” cross user/kernel/hw boundary

Java Exceptions

User Space

Kernel Space

Hardware

divide-by-0

Process

(HW Exceptions

(language not important here; this example is C)

div-by-0 handler

segfault handler

segfault

throw E;
....
catch(e) {
  java thread
} JVM Process

JVM Runtime:
- JIT
- GC

p = NULL;
x = *p;
....

thread
2) Explain how kernel/user modes are related to privileged machine instructions
   - everyone got this right!
3) What’s the importance of separating threads from processes?

We don’t need threads for concurrency!

✔ sharing
Threads share Process’ resources

**Thread**
- stack ptr
- instruction ptr
- floating point
- etc.
- registers

**Process**
- stack (per thread)
- data
- code
- private memory (address space)
- threads (at least one)
- open files
- net connections
- etc.
- resources

**Thread**
- stack ptr
- instruction ptr
- floating point
- etc.
- registers
Project 2

• Due January 26, 11:59 pm
  - same time as Project 1
  - you can resubmit until then

• Questions?
Today

• Last week’s quiz

• Project 2

• Scheduling
  (this will hopefully be useful for tomorrow’s quiz 😊)
What real schedulers look like

scheduler.c
Scheduling happens throughout the kernel

We’ll focus on CPU scheduling
CPU scheduling in the abstract

- What might we want in a good schedule?
  - fairness (every job gets a “fair” slice)
  - priority (some jobs are more important)
  - deadlines (some jobs must finish by a certain time)
  - thread locality
CPU scheduling in the abstract

Practical issues

- new jobs are starting all the time
- how do we know how long a job will take?
  (can’t plan ahead very far)
Two decisions a scheduler makes

- When do I reschedule the CPU? (i.e., how long is the next time slice?)
- Who gets the CPU next?
When do I reschedule the CPU?

• Cooperative scheduling
  - reschedule when:
    ... a thread blocks on I/O
    ... a thread calls yield()
    ... a thread finishes
  - problem:
    ... must rely on threads to relinquish CPU (fairness)

• Preemptive scheduling
  - can reschedule at any time
  - usually at timer interrupts

✔ Batch schedulers

✔ Interactive schedulers
Who gets the CPU next?

• Many algorithms ...
  Let’s look at a few simple single-CPU algorithms

• Round robin order

  ready queue
  (cycle through using round-robin)

  blocked queue
  (e.g., waiting on I/O)

running

<table>
<thead>
<tr>
<th>T1</th>
<th>T2</th>
<th>T3</th>
<th>T4</th>
<th>T5</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>T6</th>
<th>T7</th>
</tr>
</thead>
</table>
Who gets the CPU next?

• Priority order

priority-sorted ready queue (always pick first)

running

T5 | T3 | T4 | T1 | T2

blocked queue (e.g., waiting on I/O)

T6 | T7
Who gets the CPU next?

- **Multi-level Feedback Queues**

  - Ready queue: highest priority (100) (schedule round-robin)
    - \( \uparrow \) \( T5 \) \( T3 \) \( T4 \)
  - Ready queue (level 99)
    - \( T1 \) \( T2 \)
  - Ready queue (level 98)
    - \( \cdots \) \( \cdots \)
  - Blocked queue (e.g., waiting on I/O)
    - \( \downarrow \) \( T3 \) \( T6 \)
Who gets the CPU next?

- **Multi-level Feedback Queues**

  ready queue: highest priority (100)
  (schedule round-robin)

  ready queue (level 99)

  ready queue (level 98)

  blocked queue
  (e.g., waiting on I/O)

running

periodically move old tasks “down”
The kernel needs CPU time too!

Which thread does the work of sending these packets?

**Answer:** a driver thread!

Driver threads get scheduled like any other thread.