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:
  - 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.

- 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