Administrivia

- Assignment 1 due one week from now
- Please, please, please turn in your own work
  - Most of you would never think of cheating, and I apologize that I even have to bring this up
  - 50% of honor-code violations are in CS
  - If you are in trouble, ask for extensions, ask for help
  - But if you copy code, we have to turn it over to Judicial Affairs
  - If you copy code, re-format, re-name variables, etc., you will still be caught. See MOSS for some of theory behind this.

x86 Memory consistency correction

- Newer x86s let a processor read its own writes early
- E.g., both \( p_1 \) and \( p_2 \) can return 2:

  ```c
  int flag1 = 0, flag2 = 0;
  int p1 (void *ignored) { register int f, g; flag1 = 1; f = flag1; g = flag2; return 2*f + g; }
  int p2 (void *ignored) { register int f, g; flag2 = 1; f = flag2; g = flag1; return 2*f + g; }
  ```
  - Older CPUs would wait at “\( f = \ldots \)” until store complete
  - See [intel 3a, §8.2.3.5] for intel’s example
  - Not much loss compared to previous processors

CPU Scheduling

- The scheduling problem:
  - Have \( K \) jobs ready to run
  - Have \( N \geq 1 \) CPUs
  - Which jobs to assign to which CPU(s)

- When do we make decision?

Scheduling criteria

- Why do we care?
  - What goals should we have for a scheduling algorithm?

- Throughput – # of procs that complete per unit time
  - Higher is better

- Turnaround time – time for each proc to complete
  - Lower is better

- Response time – time from request to first response (e.g., key press to character echo, not launch to exit)
  - Lower is better

- Above criteria are affected by secondary criteria
  - CPU utilization – keep the CPU as busy as possible
  - Waiting time – time each proc waits in ready queue
Example: FCFS Scheduling

- Run jobs in order that they arrive
  - Called “First-come first-served” (FCFS)
  - E.g., Say $P_1$ needs 24 sec, while $P_2$ and $P_3$ need 3.
  - Say $P_2, P_3$ arrived immediately after $P_1$, get:

<table>
<thead>
<tr>
<th>Job</th>
<th>Arrival</th>
<th>Service Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>$P_1$</td>
<td>0</td>
<td>24</td>
</tr>
<tr>
<td>$P_2$</td>
<td>24</td>
<td>3</td>
</tr>
<tr>
<td>$P_3$</td>
<td>27</td>
<td>3</td>
</tr>
</tbody>
</table>

- Dirt simple to implement—how good is it?
- Throughput: 3 jobs / 30 sec = 0.1 jobs/sec
- Turnaround Time: $P_1: 24$, $P_2: 27$, $P_3: 30$
  - Average TT: $(24 + 27 + 30)/3 = 27$
- Can we do better?

FCFS continued

- Suppose we scheduled $P_2, P_3$, then $P_1$
  - Would get:

<table>
<thead>
<tr>
<th>Job</th>
<th>Arrival</th>
<th>Service Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>$P_2$</td>
<td>0</td>
<td>27</td>
</tr>
<tr>
<td>$P_3$</td>
<td>3</td>
<td>6</td>
</tr>
<tr>
<td>$P_1$</td>
<td>6</td>
<td>30</td>
</tr>
</tbody>
</table>

- Throughput: 3 jobs / 30 sec = 0.1 jobs/sec
- Turnaround time: $P_1: 30$, $P_2: 3$, $P_3: 6$
  - Average TT: $(30 + 3 + 6)/3 = 13$ – much less than 27
- Lesson: scheduling algorithm can reduce TT
- Minimizing waiting time can improve RT and TT
- What about throughput?

View CPU and I/O devices the same

- CPU is one of several devices needed by users’ jobs
  - CPU runs compute jobs, Disk drive runs disk jobs, etc.
  - With network, part of job may run on remote CPU
- Scheduling 1-CPU system with $n$ I/O devices like scheduling asymmetric $n + 1$-CPU multiprocessor
  - Result: all I/O devices + CPU busy $\Rightarrow n+1$ fold speedup!
  - Overlap them just right? throughput will be almost doubled

Bursts of computation & I/O

- Jobs contain I/O and computation
  - Bursts of computation
    - Then must wait for I/O
- To Maximize throughput
  - Must maximize CPU utilization
  - Also maximize I/O device utilization
- How to do?
  - Overlap I/O & computation from multiple jobs
    - Means response time very important for I/O-intensive jobs: I/O device will be idle until job gets small amount of CPU to issue next I/O request

Histogram of CPU-burst times

- What does this mean for FCFS?

FCFS Convoy effect

- CPU bound jobs will hold CPU until exit or I/O (but I/O rare for CPU-bound thread)
  - long periods where no I/O requests issued, and CPU held
  - Result: poor I/O device utilization
- Example: one CPU-bound job, many I/O bound
  - CPU bound runs (I/O devices idle)
  - CPU bound blocks
  - I/O bound job(s) run, quickly block on I/O
  - CPU bound runs again
  - I/O completes
  - CPU bound job continues while I/O devices idle
- Simple hack: run process whose I/O completed?
  - What is a potential problem?
 SJF Scheduling

- **Shortest-job first (SJF)** attempts to minimize TT
  - Schedule the job whose next CPU burst is the shortest
- **Two schemes:**
  - **Non-preemptive** – once CPU given to the process it cannot be preempted until completes its CPU burst
  - **Preemptive** – if a new process arrives with CPU burst length less than remaining time of current executing process, preempt (Know as the Shortest-Remaining-Time-First or SRTF)
- **What does SJF optimize?**
  - Gives minimum average waiting time for a given set of processes

**Examples**

<table>
<thead>
<tr>
<th>Process</th>
<th>Arrival Time</th>
<th>Burst Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>P₁</td>
<td>0.0</td>
<td>7</td>
</tr>
<tr>
<td>P₂</td>
<td>2.0</td>
<td>4</td>
</tr>
<tr>
<td>P₃</td>
<td>4.0</td>
<td>1</td>
</tr>
<tr>
<td>P₄</td>
<td>5.0</td>
<td>4</td>
</tr>
</tbody>
</table>

**Non-preemptive**

<table>
<thead>
<tr>
<th>P₁</th>
<th>P₂</th>
<th>P₃</th>
<th>P₄</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>3</td>
<td>7</td>
<td>8</td>
</tr>
<tr>
<td>2</td>
<td>4</td>
<td>11</td>
<td>16</td>
</tr>
</tbody>
</table>

**Preemptive**

<table>
<thead>
<tr>
<th>P₁</th>
<th>P₂</th>
<th>P₃</th>
<th>P₄</th>
<th>P₅</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>2</td>
<td>4</td>
<td>5</td>
<td>11</td>
</tr>
<tr>
<td>7</td>
<td>8</td>
<td>12</td>
<td>16</td>
<td></td>
</tr>
</tbody>
</table>

**Drawbacks?**

**Exp. weighted average example**

**Round robin (RR) scheduling**

- **Solution to fairness and starvation**
  - Preempt job after some time slice or quantum
  - When preempted, move to back of FIFO queue
  - (Most systems do some flavor of this)
- **Advantages:**
  - Fair allocation of CPU across jobs
  - Low average waiting time when job lengths vary
  - Good for responsiveness if small number of jobs
- **Disadvantages?**
RR disadvantages

- Varying sized jobs are good
  ...but what about same-sized jobs?
- Assume 2 jobs of time=100 each:
  - What is average completion time?
  - How does that compare to FCFS?

Context switch costs

- What is the cost of a context switch? (recall from Lecture 2)

- Brute CPU time cost in kernel
  - Save and restore registers, etc.
  - Switch address spaces (expensive instructions)
- Indirect costs: cache, buffer cache, & TLB misses

Time quantum

- Typical values: 10–100 msec

Two-level scheduling

- Switching to swapped out process very expensive
  - Swapped out process has most pages on disk
  - Will have to fault them all in while running
  - One disk access costs ~10ms. On 1GHz machine, 10ms = 10 million cycles!
- Context-switch-cost aware scheduling
  - Run in-core subset for “a while”
  - Then swap some between disk and memory
- How to pick subset? How to define “a while”?
  - View as scheduling memory before CPU
  - Swapping in process is cost of memory “context switch”
  - So want “memory quantum” much larger than swapping cost
Priority scheduling

- Associate a numeric priority with each process
  - E.g., smaller number means higher priority (Unix/BSD)
  - Or smaller number means lower priority (Pintos)
- Give CPU to the process with highest priority
  - Can be done preemptively or non-preemptively
- Note SJF is a priority scheduling where priority is the predicted next CPU burst time
- Starvation – low priority processes may never execute
- Solution?
  - Aging - increase a process’s priority as it waits

Multilevel feedback queues (BSD)

- Every runnable process on one of 32 run queues
  - Kernel runs process on highest-priority non-empty queue
  - Round-robs among processes on same queue
- Process priorities dynamically computed
  - Processes moved between queues to reflect priority changes
  - If a process gets higher priority than running process, run it
- Idea: Favor interactive jobs that use less CPU

Sleeping process increases priority

- p_estcpu not updated while asleep
  - Instead p_slptime keeps count of sleep time
- When process becomes runnable
  \[ p_{estcpu} \leftarrow \left( \frac{2 \cdot \text{load}}{2 \cdot \text{load} + 1} \right)^p_{slptime} \times p_{estcpu} \]
  - Approximates decay ignoring nice and past loads
- Previous description based on [McKusick]²

Process priority

- p_nice – user-settable weighting factor
- p_estcpu – per-process estimated CPU usage
- Incremented whenever timer interrupt found proc. running
- Decayed every second while process runnable
  \[ p_{estcpu} \leftarrow \left( \frac{2 \cdot \text{load}}{2 \cdot \text{load} + 1} \right) p_{estcpu} + p_{nice} \]
- Load is sampled average of length of run queue plus short-term sleep queue over last minute
- Run queue determined by \( p_{usrpri}/4 \)
  \[ p_{usrpri} \leftarrow 50 + \left( \frac{p_{estcpu}}{4} \right) + 2 \cdot p_{nice} \]
  (value clipped if over 127)

Pintos notes

- Same basic idea for second half of project 1
  - But 64 priorities, not 128
  - Higher numbers mean higher priority
  - Okay to have only one run queue if you prefer
    (less efficient, but we won’t deduct points for it)
- Have to negate priority equation:
  \[ \text{priority} = 63 - \left( \frac{\text{recent_cpu}}{4} \right) - 2 \cdot \text{nice} \]

²See library.stanford.edu for off-campus access
Limitations of BSD scheduler

- Hard to have isolation / prevent interference
  - Priorities are absolute
- Can’t donate priority (e.g., to server on RPC)
- No flexible control
  - E.g., In monte carlo simulations, error is \( \frac{1}{\sqrt{N}} \) after \( N \) trials
  - Want to get quick estimate from new computation
  - Leave a bunch running for a while to get more accurate results
- Multimedia applications
  - Often fall back to degraded quality levels depending on resources
  - Want to control quality of different streams

Real-time scheduling

- Two categories:
  - Soft real time—miss deadline and CD will sound funny
  - Hard real time—miss deadline and plane will crash
- System must handle periodic and aperiodic events
  - E.g., procs A, B, C must be scheduled every 100, 200, 500 msec, require 50, 30, 100 msec respectively
  - Schedulable if \( \sum \frac{CPU}{period} \leq 1 \) (not counting switch time)
- Variety of scheduling strategies
  - E.g., first deadline first (works if schedulable)

Multiprocessor scheduling issues

- Must decide on more than which processes to run
  - Must decide on which CPU to run which process
- Moving between CPUs has costs
  - More cache misses, depending on arch more TLB misses too
- Affinity scheduling—try to keep threads on same CPU
  - But also prevent load imbalances
  - Do cost-benefit analysis when deciding to migrate

Multiprocessor scheduling (cont)

- Want related processes scheduled together
  - Good if threads access same resources (e.g., cached files)
  - Even more important if threads communicate often, otherwise must context switch to communicate
- Gang scheduling—schedule all CPUs synchronously
  - With synchronized quanta, easier to schedule related processes/threads together

Thread scheduling

- With thread library, have two scheduling decisions:
  - Local Scheduling – Thread library decides which user thread to put onto an available kernel thread
  - Global Scheduling – Kernel decides which kernel thread to run next
- Can expose to the user
  - E.g., pthread_attr_setscope allows two choices
    - PTHREAD_SCOPE_SYSTEM – thread scheduled like a process (effectively one kernel thread bound to user thread – Will return ENOTSUP in user-level pthreads implementation)
    - PTHREAD_SCOPE_PROCESS – thread scheduled within the current process (may have multiple user threads multiplexed onto kernel threads)

Thread dependencies

- Say \( H \) at high priority, \( L \) at low priority
  - \( L \) acquires lock \( l \)
  - Scene 1: \( H \) tries to acquire \( l \), fails, spins. \( L \) never gets to run.
  - Scene 2: \( H \) tries to acquire \( l \), fails, blocks. \( M \) enters system at medium priority. \( L \) never gets to run.
  - Both scenes are examples of priority inversion
- Scheduling = deciding who should make progress
  - Obvious: a thread’s importance should increase with the importance of those that depend on it.
  - Naïve priority schemes violate this
**Fair Queuing (FQ)**

- **Digression: packet scheduling problem**
  - Which network packet should router send next over a link?
  - Problem inspired some algorithms we will see next week
  - Plus good to reinforce concepts in a different domain...

- **For ideal fairness, would send one bit from each flow**
  - In weighted fair queuing (WFQ), more bits from some flows

**Packet scheduling**

- **Differences from CPU scheduling**
  - No preemption or yielding—must send whole packets
    - Thus, *can’t send one bit at a time*
  - But know how many bits are in each packet
    - Can see the future and know how long packet needs link

- **What scheduling algorithm does this suggest?**

**Priority donation**

- Say higher number = higher priority
- **Example 1:** *L* (prio 2), *M* (prio 4), *H* (prio 8)
  - *L* holds lock *l*
  - *M* waits on *l*, *L*'s priority raised to *L‘* = \(\max(M, L) = 4\)
  - Then *H* waits on *l*, *L*'s priority raised to \(\max(H, L’) = 8\)

- **Example 2: Same threads**
  - *L* holds lock *l*, *M* holds lock *l2*
  - *M* waits on *l*, *L*'s priority now *L‘* = 4 (as before)
  - Then *H* waits on *l2*, *M‘*s priority goes to *M‘* = \(\max(H, M) = 8\), and *L*'s priority raised to \(\max(M’, L’) = 8\)

- **Example 3:** *L* (prio 2), *M1*, . . . , *M1000* (all prio 4)
  - *L* has *l*, and *M1*, . . . , *M1000* all block on *l*. *L*'s priority is \(\max(L, M1, . . . , M1000) = 4\).

**FQ Algorithm**

- Suppose clock ticks each time a bit is transmitted
- Let \(P_i\) denote the length of packet *i*
- Let \(S_i\) denote the time when start to transmit packet *i*
- Let \(F_i\) denote the time when finish transmitting packet *i*
  - \(F_i = S_i + P_i\)

- **When does router start transmitting packet *i***?
  - If arrived before router finished packet *i – 1* from this flow, then immediately after last bit of *i – 1* \((F_{i-1})\)
  - If no current packets for this flow, then start transmitting when arrives (call this \(A_i\))

- **Thus:** \(F_i = \max(F_{i-1}, A_i) + P_i\)