Outline for Today’s Lecture

• Administrative:
  – Issues w.r.t. Program 3
    • Friday? Wed. Oct 18 11:59 pm along with HW1?
  – Program 4 is hard!
• Objective for today:
  – Continue discussion of the scheduling policies.
  – Advanced topics in scheduling.

Scheduling Algorithms (review)

• SJF - Shortest Job First
  – Provably optimal in minimizing average response time (assuming we know service times in advance). Gets long jobs out of the way of short ones.
  – Starvation possible, sacrifices “fairness”.
• FIFO, FCFS
  – Emphasis on “fairness”
  – Sacrifices average response time.
• Round Robin
  – Preemption overhead affects on throughput. The quantum size affects amount of overhead.
  – Adaptively “finds” shortest jobs without a priori information.
  – In terms of “fairness”, reduces variance in wait times.
• Priority Scheduling
  – Dangers of priority inversions.

A Simple Policy: FCFS

The most basic scheduling policy is first-come-first-served, also called first-in-first-out (FIFO).
FCFS is just like the checkout line at the QuickiMart.
Maintain a queue ordered by time of arrival.
GetNextToRun selects from the front of the queue.
FCFS with preemptive timeslicing is called round Robin.

Evaluating FCFS

How well does FCFS achieve the goals of a scheduler?
throughput. FCFS is as good as any non-preemptive policy.

fairness. FCFS is intuitively fair... sort of.

response time. Long jobs keep everyone else waiting.

Grocery store with no express line.
Preemptive FCFS: Round Robin

Preemptive timeslicing is one way to improve fairness of FCFS. If a job does not block or exit, force an involuntary context switch after each quantum of CPU time. Preempted job goes back to the tail of the ready list. If Q is infinitesimal, Round Robin is called processor sharing.

With infinitesimal Q, Round Robin is called processor sharing.

D = 5
D = 3
D = 1

R = (5+6)/5 = 5.5
R = (3+6+e)/2 = 4 + e

Evaluating Round Robin

Response time: RR reduces response time for short jobs.
- For a given load, a job’s wait time is proportional to its D.
- Fairness: RR reduces variance in wait times.
- But: RR forces jobs to wait for other jobs that arrived later.
- Throughput: RR imposes extra context switch overhead.
  - CPU is only Q/(Q+e) as fast as it was before.
  - Degrades to FCFS with large Q.

Minimizing Response Time: SJF

Shortest Job First (SJF) is provably optimal if the goal is to minimize R.
Example: express lanes at the MegaMart
Idea: get short jobs out of the way quickly to minimize the number of jobs waiting while a long job runs.
Intuition: longest jobs do the least possible damage to the wait times of their competitors.

SJF

- In preemptive case, shortest remaining time first
- In practice, we have to predict the CPU service times (computation time until next blocking).
- Favors interactive jobs, needing response, & repeatedly doing user interaction
- Favors jobs experiencing I/O bursts - soon to block, get devices busy, get out of CPU’s way
- Focus is on an average performance measure, some long jobs may starve under heavy load/constant arrival of new short jobs.
Behavior of SJF Scheduling

- With SJF, best-case $R$ is not affected by the number of tasks in the system.
  - Shortest jobs budge to the front of the line.
- Worst-case $R$ is unbounded, just like FCFS.
  - Since the queue is not "fair", starvation exists - the longest jobs are repeatedly denied the CPU resource while other more recent jobs continue to be fed.
- SJF sacrifices fairness to lower average response time.

SJF in Practice

Pure SJF is impractical: scheduler cannot predict ave. service demand ($D$) values.
However, SJF has value in real systems:

- Many applications execute a sequence of short CPU bursts with I/O in between.
  - e.g., interactive jobs block repeatedly to accept user input.
  - Goal: deliver the best response time to the user.
- e.g., jobs may go through periods of I/O-intensive activity.
  - Goal: request next I/O operation ASAP to keep devices busy and deliver the best overall throughput.
- Use adaptive internal priority to incorporate SJF into RR.
  - Weather report strategy: predict future $D$ from the recent past.

Considering I/O

In real systems, overall system performance is determined by the interactions of multiple service centers.

```
<table>
<thead>
<tr>
<th>Service Center</th>
<th>CPU</th>
<th>I/O</th>
</tr>
</thead>
<tbody>
<tr>
<td>Demand</td>
<td>2</td>
<td>4</td>
</tr>
</tbody>
</table>
```

Easy to predict $U$, $J$, $R$, and $N$ at each center: use Forced Flow Law to predict arrival rate $J$, at each center $k$, then apply Little's Law to $k$.

Then:

$$ A = S \times J $$

Bottlenecks

It is easy to see that the maximum throughput $X$ of a system is reached as $1/\tilde{X}$ approaches $D_k$ for service center $k$ with the highest demand $D_k$.

$k$ is called the bottleneck center.

Overall system throughput is limited by $D_k$ when $U_k$ approaches 1.

Example 1:

- CPU: $S = 2$
- I/O: $S = 4$

This job is I/O bound.

To improve performance, always attack the bottleneck center!

Example 2:

- CPU: $S = 4$
- I/O: $S = 4$

Demands are evenly balanced.

Will multiprogramming improve system throughput in this case?
**Bottlenecks**

It is easy to see that the maximum throughput \( X \) of a system is reached as \( 1/\tau \) approaches \( D_k \) for service center \( k \) with the highest demand \( D_k \). It is called the bottleneck center.

Overall system throughput is limited by \( k \) when \( U_k \) approaches 1.

**Example 1:**
- CPU: \( S_0 = 1 \), \( S_1 = 4 \)
- I/O: \( S_0 = 1 \), \( S_1 = 4 \)

This job is I/O bound. How much will performance improve if we double the speed of the CPU? Is it worth it?

To improve performance, attack the bottleneck center!

**Example 2:**
- CPU: \( S_0 = 4 \), \( S_1 = 4 \)
- I/O: \( S_0 = 4 \), \( S_1 = 4 \)

Demands are evenly balanced. Will multiprogramming improve system throughput in this case?

**Two Schedules for CPU/Disk**

1. **Naive Round Robin**
   - CPU busy 25/25: \( U = 100\% \)
   - Disk busy 15/25: \( U = 60\% \)

2. **Round Robin with SJF**
   - CPU busy 25/37: \( U = 67\% \)
   - Disk busy 15/37: \( U = 40\% \)

33% performance improvement

**Multilevel Feedback Queue**

Many systems (e.g., Unix variants) implement priority and incorporate SJF by using a multilevel feedback queue.

**Multilevel Feedback Queue:**

- Separate queue for each of \( N \) priority levels.
- Factor previous behavior into new job priority.
- Feedback: \( \text{Priority}_{\text{CPU-bound}} = \text{user\_base\_priority} + \left[ \text{p\_estcpu} / 4 \right] + 2 \times \text{p\_nice} \)
- GetNextToRun selects job at the head of the highest priority queue.
- Jobs holding resources.
- Jobs with high external priority.
- CPU-bound jobs.
- I/O bound jobs waiting for CPU.
- Jobs with high external priority.

**Concrete Implementation**

4.4BSD example:

multilevel feedback queues based on calculated priority, round-robin within level.

- Use quantum - reenter queue you came off of.
- Changing priority (every 4 ticks)
  \[ \text{priority} = \text{user\_base\_priority} + \left[ \text{p\_estcpu} / 4 \right] + 2 \times \text{p\_nice} \]
• \( p_{\text{estcpu}} \) is incremented each tick during which the process is found running and adjusted each second via decay filter
\[
p_{\text{estcpu}} = \frac{(2^*\text{load})}{(2^*\text{load} + 1)} p_{\text{estcpu}} + p_{\text{nice}}.
\]
Load over previous minute interval - sampled ave. of sum of lengths of run queue and short term sleep queue

• 90% of CPU utilization in any 1-sec interval is forgotten after 5 seconds.
• Upon waking from sleep, first
\[
p_{\text{estcpu}} = \frac{(2^*\text{load})}{(2^*\text{load} + 1)} p_{\text{slptime}} + p_{\text{estcpu}}
\]
and then recompute priority

Real Time Schedulers
• Real-time schedulers must support regular, periodic execution of tasks (e.g., continuous media).
• e.g. Microsoft’s Rialto scheduler [Jones97] supports an external interface for:
  – CPU Reservations
    • “I need to execute for \( x \) out of every \( y \) units.”
    • Scheduler exercises admission control at reservation time: application must handle failure of a reservation request.
  – Time Constraints
    • “Run this before my deadline at time \( T \).”

Liu and Layland (classic TR Scheduling paper)
• Hard real time - tasks executed in response to events (requests) and must be completed in some fixed time (deadline)
• Soft real time - statistical distribution of response times

Assumptions
• Tasks are periodic with constant interval between requests, \( T_i \) (request rate \( 1/T_i \))
• Each task must be completed before the next request for it occurs
• Tasks are independent
• Run-time for each task is constant (max), \( C_i \)
• Any non-periodic tasks are special
Definitions

- Deadline is time of next request
- Overflow at time $t$ if $t$ is deadline of unfulfilled request
- Feasible schedule - for a given set of tasks, a scheduling algorithm produces a schedule so no overflow ever occurs.
- Critical instant for a task - time at which a request will have largest response time.
  - Occurs when task is requested simultaneously with all tasks of higher priority

Rate Monotonic

- Assign priorities to tasks according to their request rates, independent of run times
- Optimal in the sense that no other fixed priority assignment rule can schedule a task set which can not be scheduled by rate monotonic.
- If feasible (fixed) priority assignment exists for some task set, rate monotonic is feasible for that task set.

Earliest Deadline First

- Dynamic algorithm
- Priorities are assigned to tasks according to the deadlines of their current request
- With EDF there is no idle time prior to an overflow
- For a given set of $m$ tasks, EDF is feasible if $C_1/T_1 + C_2/T_2 + \ldots + C_m/T_m \leq 1$
- If a set of tasks can be scheduled by any algorithm, it can be scheduled by EDF
**Proportional Share**

Goals: to integrate real-time and non-real-time tasks, to police ill-behaved tasks, to give every process a well-defined share of the processor.

- Each client, \( i \), gets a weight \( w_i \).
- Instantaneous share \( f_i(t) = \frac{w_i}{\sum w_j} \)
- Service time (\( t \) constant in interval)
  \[ S_i(t_0, t_1) = f_i(t) \Delta t \]
- Set of active clients varies \( \Rightarrow f_i \) varies over time

**Lottery Scheduling**

Waldspurger and Weihl (OSDI 94)

- Goal: responsive control over the relative rates of computation
- Claims:
  - Support for modular resource management
  - Generalizable to diverse resources
  - Efficient implementation of proportional-share resource management: consumption rates of resources by active computations are proportional to relative shares allocated

**Basic Idea**

- Resource rights are represented by lottery tickets
  -Give \( W_p \) "lottery tickets" to each process \( p \).
  -abstract, relative (vary dynamically wrt contention), uniform (handle heterogeneity)
  -responsiveness: adjusting relative # tickets gets immediately reflected in next lottery
- At allocation time: hold a lottery; Resource goes to the computation holding the winning ticket.
  -GetNextToRun selects "winning ticket" randomly.

**Fairness**

- Expected allocation is proportional to # tickets held - actual allocation becomes closer over time.
- Number of lotteries won by client
  \[ E[w] = n p \text{ where } p = t/T \]
- Response time (# lotteries to wait for first win)
  \[ E[n] = 1/p \]
Example List-based Lottery

T = 20

10 2 5 1 2

Summing: 10 12 17

Random(0, 19) = 15

Bells and Whistles

- Ticket transfers - objects that can be explicitly passed in messages
  - Can be used to solve priority inversions
- Ticket inflation
  - Create more - used among mutually trusting clients to dynamically adjust ticket allocations
- Currencies - "local" control, exchange rates
- Compensation tickets - to maintain share
  - use only $f$ of quantum, ticket inflated by $1/f$ in next

Kernel Objects

1000 base

ticket

1000 currency

name

200 Active amount

Issued tickets

Backing tickets

Currency name

Exchange rate: 1 bob = 20 base
Example List-based Lottery

\( T = 3000 \text{ base} \)

\[
\begin{array}{c}
\text{10 task2} & \text{2bob} & \text{5 task3} & \text{1} & \text{2bob} \\
\text{base}
\end{array}
\]

Random(0, 2999) = 1500

Compensation

- A holds 400 base, B holds 400 base
- A runs full 100msec quantum, B yields at 20msec
- B uses 1/5 allotted time
  Gets compensation ticket valued at \( 400/(1/5) = 2000 \text{ base at next lottery} \)

Ticket Transfer

- Synchronous RPC between client and server
- create ticket in client’s currency and send to server to fund it’s currency
- on reply, the transfer ticket is destroyed

Control Scenarios

- Dynamic Control
  Conditionally and dynamically grant tickets
  Adaptability
- Resource abstraction barriers supported by currencies. Insulate tasks.
Other Kinds of Resources

Control relative waiting times for mutex locks.
- Mutex currency funded out of currencies of waiting threads
- Holder gets inheritance ticket in addition to its own funding, passed on to next holder (resulting from lottery) on release.

Dynamic Voltage Scaling
(Weiser, Demers, Shenker)

- Energy/time $\propto$ Voltage$^2$
- Voltage scheduling - transition times of $\sim$10μs (according to Weiser, Pering)
- Intuitive goal - fill “soft idle” times with slow computation
- MIPJ - metric MIPS/Watts

Relationships

- Power (watts) = Voltage (volts) * Current (amps)
- Power (watts) = Energy (Joules) / Time (sec)
- Energy (Joules) = Power (watts) * Time (sec)
- Energy (Joules) = Voltage (volts) * Charge (coulombs)
- Current (amps) = Voltage (volts) / Resistance (ohms)

Interval Scheduling
(adjust clock based on past window, no process reordering involved)