Room Etiquette

- Do not go into room until previous class is dismissed
  - That is, until you see folks coming out
- Going into the room before that disrupts the class, disadvantaging them
- Also, please be respectful to members of the class and the instructor
  - Too much nastiness in the world today!
  - No excuse for being rude

What Does *perror*(3) Mean?

- This is the library function *perror* in section 3 of the Linux manual
- To see it, type

```
man 3 perror
```

  to the Linux shell

Process Scheduling

Round Robin

- Designed especially for time sharing
  - Uses *quantum*, typically between 1/60 sec and 1 sec
- Processes kept in a queue
- As each process is preempted, it moves to the rear of the queue
- All new arrivals come in at the rear of the queue

Example

- Using our previous jobs with a quantum of 5:

time	0	5	10	13	18	23	28	33	35	40	45	47	52	57	61
proc	A	B	C	D	E	A	B	D	E	B	E	B	B	B	
rem	5	24	0	2	7	0	19	0	2	14	0	9	4	0	

Round Robin

- Decision mode: preemptive (at quantum)
- Priority function: $p(a, r, t) = c$
- Arbitration rule: cyclic

Process	Service time	Arrival time	Start	Finish	T	W	R
A	10	0	…	28	28	18	2.8
B	29	1	…	61	60	31	2.1
C	3	2	…	13	11	8	3.7
D	7	3	…	35	28	21	4.0
E	12	4	…	47	43	35	3.6
mean					34	22.6	3.2

Variants

- Round Robin, but adjust quantum periodically.
  - *example*: after every process switch, the quantum becomes q/n, where n is the number of processes in the ready list
  - Few ready processes means that each gets a long quantum, minimizing process switches.
  - Lots of ready processes means that this algorithm gives more processes a shot at the CPU over a fixed period of time, at the price of more process switching
  - Processes needing a small amount of CPU time get a quantum fairly soon, and hence *may* finish sooner.
- Round Robin, but give the current process an extra quantum when a new process arrives
  - This reduces process switching in proportion to the number of processes arriving.

Multi-Level Feedback Queue

- Goal is to optimize turnaround time , make a system feel responsive to interactive users
- Problems:
  - Reducing turnaround time means running SJF algorithm, but do not know that time in advance
  - Round Robin great at reducing response tim, terrible at reducing turnaround time

Solution: Multiple Queues!

- MLFB uses multiple queues, each with its own priority
  - Each queue uses round robin, with processes going on the end until they are moved to next higher queue
- Rule: given processes A, B, the MLFQ:

  If priority(A) > priority(B), then A runs

  If priority(A) = priority(B), then A, B run in round robin

  If priority(B) > priority(A), then B runs
- Entering processes go into the highest priority queue
- If process blocks, it reenters the scheduler at prescribed level
  - Usually same of higher priority ones
- Some systems: periodically move processes to highest priority queue

Results

- CPU-bound jobs drop in priority after some number of quanta
- I/O bound jobs are on the top, as this gives interactive users quick response
- If a process changes from a CPU-bound process to an I/O-bound process, its priority changes accordingly (but it may change slowly)
- So it is *adaptive*, adapting to the process mix, rather than keeping conetant how each process is handled

Multi-Level Feedback Queue

- Decision mode: preemptive (at quantum)
- Priority function: $p(a, r, t) = n - i$, where I satisfies both $0 \le i < n$ and $T_0(2^i - 1) \le a < T_0(2^{i+1} - 1)$, where $T_p = 2^p T_0$
- Arbitration rule: cyclic or chronological within queues

Below: quantum = 1, $n = 2$, $T_0 = 2$, $T_1 = 4$, $T_2 = 8$

Process	Service time	Arrival time	Start	Finish	T	W	R
A	10	0	…	38	38	28	3.8
B	29	1	…	61	60	31	2.1
C	3	2	…	13	11	8	3.7
D	7	3	…	30	27	20	3.9
E	12	4	…	44	40	28	3.3
mean					35.2	23	3.4

Other Issues

- Some questions:
  - How many queues should there be?
  - How big should the quantum be for each queue?
  - How and when should you move a process to a higher queue?
- No set answers; you learn from experience
  - Most use different quanta for levels; the lower the priority, the longer the quantum as CPU-bound processes tend to drop
  - Some quanta set by table; others by formula
  - Some systems allow users to advise on priority
  - Some reserve highest levels for operating system work

External Priority Methods

- Scheduling depends upon external factors such as amount paid
- User buys a particular response ratio
- Process must finish by a certain time
- Groups of users are allocated unequal blocks of time based on some criteria
  - Importance of work
  - Funding
  - Others . . .

Modified Round Robin

- Set the quantum independently for each *process*
- The quantum is based on an external priority for the process
    - High priority work gets a longer quantum than normal processes
    - The more you pay, the longer the quantum

VAX//VMS Scheduler

- 32 priority levels
  - 0-15 for regular processes
  - 16-31 for real-time processes
  - The higher the number, the higher the priority
- Real-time processes have fixed priority
- Regular process priority is dynamic

Assignment of Priorities

- At process creation, assign a *base* priority
  - This is process' minimum priority
- System events alter current priority of the process
  - Each event has an associated priority increment
  - Example: terminal read > terminal write > disk I/O
  - When awakened by system event, increment added to priority
- On pre-emption due to quantum expiration, priority decreased by 1
- Similar to a MLFB scheme, with two major differences:
  - Processes need not start at the highest level (they start at the base priority level)
  - Quanta are associated with each *process*, not level

Worst Service Next

- After each quantum, compute *suffering function* based on:
  - How long the process has been waiting
  - How many times has it been pre-empted
  - How much user is paying
  - How much time and resources it is expected to use
- Process with greatest suffering goes next

Guaranteed Response Ratio

- User buys a guaranteed response ratio
- Like Worst Service Next
- Suffering function takes into account difference between the guaranteed response ratio and the actual current response ratio

Deadline Scheduling

- Each process specifies:
  - How long it will run (usually an overestimate by person submitting job)
  - When it must be finished by
- System does one of two things:
  - Accepts the job and schedules the process to meet both the time required for the process to execute and when it needs to finish by
  - Rejects the job, because it cannot be completedby the deadline

Fair Share Scheduler

- Allocate blocks of CPU time to a particular set of processes
  - Within each group, use a standard schedule
  - Allocate CPU proportionally to each group
- Example:
  - Process p_1 in group 1; processes p_2, p_3 in group 2; processes p_4, p_5, p_6 in group 3; processes p_7, p_8, p_9, p_{10} in group 4
  - Regular scheduler: give each process 10% of CPU time
  - Fair share scheduler: give each *group* 25% of CPU time
    - p_1 gets 25%
    - p_2, p_3 get 25%/2 = 12.50%
    - p_4, p_5, p_6 get 25%/3 = 8.33%
    - p_7, p_8, p_9, p_{10} get 25%/4 = 6.25%

Example from UNIX Fair Share Scheduler

- Assume 3 processes
- Group 1 has process A, group 2 has processes B, C
- Internal priority function:

  priority = (recent CPU usage)/2 + (group CPU usage)/2 + threshhold

  (threshhold is 60 for user processes)
- Decay function:

  decay of CPU usage = (CPU usage)/

  This decrements the current CPU usage of processes not run

  It effectively raises the process priority

Real-Life Example

- Quantum is 1 second
- The lower the number, the higher the priority

A runs for 1 second

Decay applied to CPU and group CPU usage; both become 30

A's new priority is 30/2 + 30/2 + 60 = 90

B, C both have priority 60, so one of them goes

B runs for 1 second

Decay applied to CPU and group CPU usage

A's CPU time is now 15, group 1's in 15, B's is 30, group 2's is 30

A's new priority is 15/2 + 15/2 + 60 = 74 (note integer division)

B's new priority is 30/2 + 30/2 + 60 = 90

C's new priority is 0/2 + 30/2 + 60 = 75

Real-Life Example

A runs for 1 second

Decay applied to CPU and group CPU usage

A’s CPU time is now (15+60)/2 = 37, group 1’s is 37, B’s is 15, group 2’s is 15

A’s new priority is 37/2 + 37/2 + 60 = 97 (note integer division)

B’s new priority is 15/2 + 15/2 + 60 = 75

C’s new priority is 0/2 + 15/2 + 60 = 67

C runs for 1 second

Decay applied to CPU and group CPU usage

A’s CPU time is now 37/2 = 18, group 1’s is 18, B’s is 7, group 2’s is 37; C’s is 30

A’s new priority is 18/2 + 18/2 + 60 = 69 (note integer division)

B’s new priority is 7/2 + 37/2 + 60 = 81

C’s new priority is 30/2 + 37/2 + 60 = 93

Real-Life Example

- So now A runs
- Note group 1 (A) gets 50% of the CPU, group 2 (B, C) gets 50% of the CPU
  - In group 2, B gets 25% and C gets 25% (equally split)

Lottery Scheduling

- Idea: hold lottery to determine which process runs next
  - Processes that are to run more often get more chances to win the lottery
- Tickets represent share of CPU the process should receive
  - A has 75 tickets, B has 25; then A gets CPU 75% of the time, B 25% of the time
- How it works
  - Say there are 100 tickets; A has tickets 0-74, B 75-99
  - Scheduler picks random number between 0 and 99 inclusive
  - If it’s between 0, 74 inclusive, run A; otherwise run B

Example: From Above

time	num	proc	time	num	proc
0	79	B	10	82	B
1	68	A	11	45	A
2	69	A	12	94	B
3	28	A	13	27	A
4	75	B	14	12	A
5	94	B	15	15	A
6	68	A	16	29	A
7	28	A	17	43	A
8	15	A	18	76	B
9	40	A	19	95	B

- 100 tickets
- A has tickets 0 to 74 (75% of all tickets)
- B has tickets 75 to 99 (25% of all tickets)
- So A should run 75% of the time, B 25% of the time

- In table: *num* is random number between 0 and 99 inclusive; *proc* is process

- A runs 13 times, B runs 7 times
- So A runs 65% of the time, B runs 35% of the time

Implementation

- Keep processes in a list
- Scheduler generates random number between 0 and number of tickets (less 1)
- Scheduler walks list, adding up numbers
- When sum exceeds random number, that's the process that runs

Example

- 5 processes
  - A has 30 tickets
  - B has 25 tickets
  - C has 10 tickets
  - D has 55 tickets
  - E has 6 tickets
- Scheduler generates 78
- Cumulative sum exceeds 78 at D (65 < 78 <120), so D runs

process	num tickets	cum sum
A	30	30
B	25	55
C	10	65
D	55	120
E	6	126

Compensation Tickets

- I/O bound processes block often, using less than a full quantum, so are likely to get less than their expected share of CPU
- Process that uses a fraction *f* of its CPU quantum can be given a *compensation ticket*
  - Ticket inflates value by 1/*f* until process gets CPU
- These favor I/O-bound and interactive processes, helping them get their fir share of CPU

Example

- Quantum is 150ms
- Process blocks for I/O after 50ms
  - f = 50ms/150ms = 1/3
- Value of all the process' tickets are multiplied by 1/f, or 3
- After process gets CPU, original values restored

Problem

- How are tickets distributed among the processes?
  - Give each user some number of tickets, and user distributes them among their processes
  - An open problem
- Guarantees are probabilistic, not deterministic
- High response time variability