Round-Robin

- Take turns
- Give each process a *time quantum*
- When the time is up, move the current process to the end of the queue

Quantum Length

- The shorter the quantum, the more responsive the system
- However, process switching is expensive (saving and reloading registers, switching virtual memory maps, flushing the cache, bookkeeping, etc.)
- Suppose the compute quantum is 4 msec and process switches take 1 msec. That's 20% overhead — too much
- Suppose we have 100 msec quanta
- If the run queue ever gets long, even cheap requests will take too long
- Need a reasonable compromise: 20-50 msec?

Priority Scheduling

- Not all processes are equally important
- Assign them *priority levels*
- Simplest version: always run the highest-priority process
- Not a good idea — what if it's CPU bound?

3 / 39

Priority Adjustments

- Periodically reduce the priority of the running process
- Eventually, it falls below the priority of the next process
- Alternative: increase the priority of non-running processes
- Called *process aging*
- Or do both

4 / 39

Dynamic Priority Adjustment

- Adjust process priority according to its recent history
- Example: increase priority of non-running processes; decrease priority of running processes, as above
- Boost priority of I/O-bound processes:
  - If process used $1/f$ of its last quantum, boost its priority proportional to f
- Use *priority classes*: have separate queues for each priority level, and run each queue round-robin; switch to lower-priority queue when this one is empty

5 / 39

Run Queues

- Each run queue is a linked list
- To raise or lower a process' priority, move it to a different list
- Two schemes for priority aging:
  - Not many processes: have a fine-grained counter for each process incremented at a clock interrupt; at some limit, increase priority
  - Lots of processes and queues: periodically, move each list to the next level up

6 / 39

Varying Quanta

- Many processes need just a little bit of CPU time
- Since process switches can be expensive, don't do them too often
- Solution to both problems: give lower priority queues longer quanta
- Top priority queue: one quantum
- Second queue: two quanta CPU allocation
- Third queue: four quanta, etc.
- Alternate solution: "short" (initial) quantum at high priority and "long" quanta at low priority thereafter

7 / 39

Helping Interactive Processes

- Look for signs of user input
- When they occur, give the process a very high priority
- Example: on CTSS, when a user typed a carriage return, the process got top priority
- Harder to do today — what's "interactive" on a networked process? For a mouse movement, which process is credited?

8 / 39

Gaming the System

- On time-sharing systems, watch for attempts to fake out the scheduler
- CTSS example: typing spurious carriage returns
- XDS 940 example: do really quick I/O operation
  - Solution: save remaining CPU quantum; when process restarts, use remainder instead of full allocation
- Not applicable on single-user machines — you're only hurting yourself
- Instead, watch for inadvertent influences on the wrong process

9 / 39

Process Priorities

- What processes should have higher priorities?
- Administrative issues
- System performance
  - Kernel processes (up to a point)
  - Interactive services processes (i.e., X server)
- Users can lower priority of their own processes, sometimes to avoid competing with themselves

10 / 39

Unix Priorities

- Tradition: lower numbers indicate higher priorities
- "Nice" value is a user-specified modifier
- A nice value of $+20$ specifies a very low priority process
- Only `root` can set negative niceness
- Default is 0
- Note: this is an API; internal metric can be different

11 / 39

Some Linux Priorities

```
UID        PRI  NI     SZ CMD
root        76   0    606 init
root        94  19      0 [ksoftirqd/0]
root        75   0      0 [khubd]
root        83   0      0 [scsi_eh_1]
root        79   0   6285 ypbind
root        75   0   1242 /usr/sbin/sshd
smb         76   0   1007 ps
```

The PRI value factors in the niceness and the process' dynamic priority

12 / 39

Shortest Process Next

- Can we emulate batch systems' shortest first algorithm?
- It's hard, because we don't have good estimates
- Instead, use historical data for a moving, decaying, average
- Let first time = T_0; second is T_1

$$
\begin{aligned}
T_2 &= \alpha T_0 + (1-\alpha)T_1 \\
T_3 &= \alpha^2 T_0 + \alpha(1-\alpha)T_1 + (1-\alpha)T_2 \\
T_4 &= \alpha^3 T_0 + \alpha^2(1-\alpha)T_1 + \alpha(1-\alpha)T_2 + \\
&\quad (1-\alpha)T_3 \\
&\cdots
\end{aligned}
$$

13 / 39

Whose History?

- Do we measure command history?
- Hard; requires a lot of kernel state
- Besides, command time can vary a lot depending on input data
- Better to measure user (or terminal) behavior

14 / 39

Guaranteed Scheduling

- For n users or process on a system, give each $1/n$ of the CPU
- Measure actual CPU usage
- Calculate process' CPU time entitlement
- Look at the ratio of the two: 0.5 means it's had only half the CPU it's entitled to, so it gets priority over a process with a ratio of 2.0

15 / 39

Lottery Scheduling

- Give each process *lottery tickets*
- Higher priority processes get more tickets; lower priority process get fewer
- At scheduling time, pick a random ticket
- The process holding that ticket gets to run
- Note: tickets can be exchanged between processes, such as between client and server

16 / 39

Fair-Share Scheduling

- In some systems, processes don't compete, users do
- We don't want to encourage forking just to get a larger share of the CPU
- Solution: make decisions based on *user* CPU consumption instead of process CPU consumption
- Priorities, etc., can still apply

17 / 39

Real-Time Scheduling 18 / 39

Real-Time Systems

- Must respond to actual clock-on-the-wall time
- A late process may be a useless process
- Two types, hard and soft
- Hard real time systems have deadlines that *must* be met; used for process control, avionics, etc.
- Soft real time tries its best, but can miss occasional deadlines
- Both depend on knowledge of processing time per request

18 / 39

Periodic Events

- Many real-time events occur with a regular frequency
- Suppose there are m events, withe event i needing C_i seconds of CPU and occurring every P_i seconds
- System works if and only if

$$\sum_{i=1}^{m} \frac{C_i}{P_i} \leq 1$$

- Watch out for process switch overhead...

19 / 39

Aperiodic Events

- Other events don't happen on a regular schedule
- The basic constraint is the same: total load must be less than total capacity
- Engineering such systems is harder

20 / 39

Mixing Real-Time and Non-Realtime Processes

- Some systms have both kinds of processes (and schedulers)
- General strategy: give real-time processes priority over all conventional processes
- Schedule them round-robin or possibly even nonpreemptively

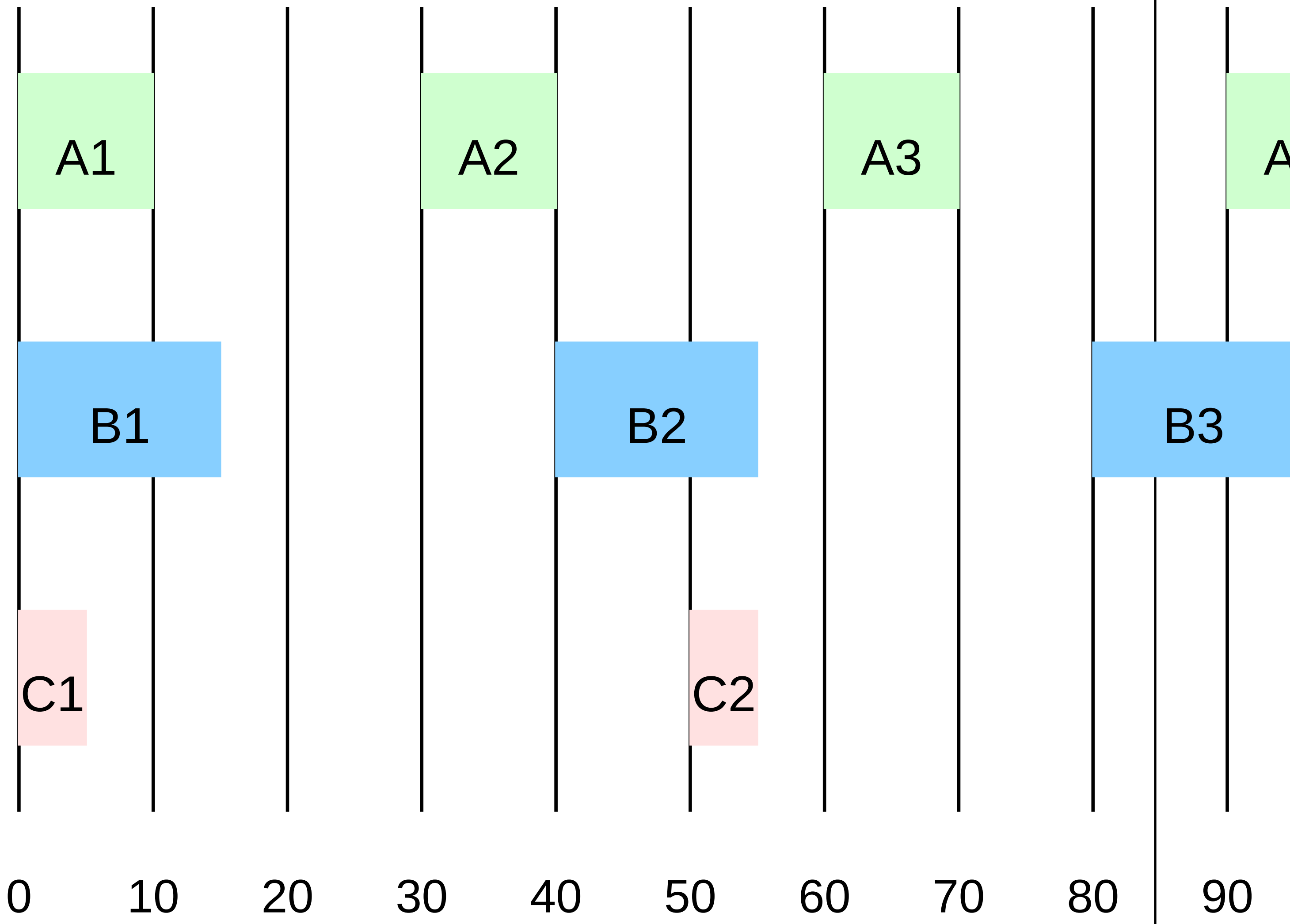
21 / 39

Multiple Real-Time Processes

- A runs every 30 msec; each time it needs 10 msec of CPU time
- B runs 25 times/sec for 15 msec
- C runs 20 times/sec for 5 msec
- For our equation, A uses $10/30$ of the CPU, B uses $15/40$, and C uses $5/50$; that's about 81%

22 / 39

Diagram



$A1$ must finish before $A2$ starts, $B1$ before $B2$,. . .

Other Issues

- Some real-time systems permit preemption; some do not
- Desirability depends on system type (text's discussion is for multimedia system, which are usually preemptible
- May have aperiodic processes in the mix
- Static or dynamic scheduling

24 / 39