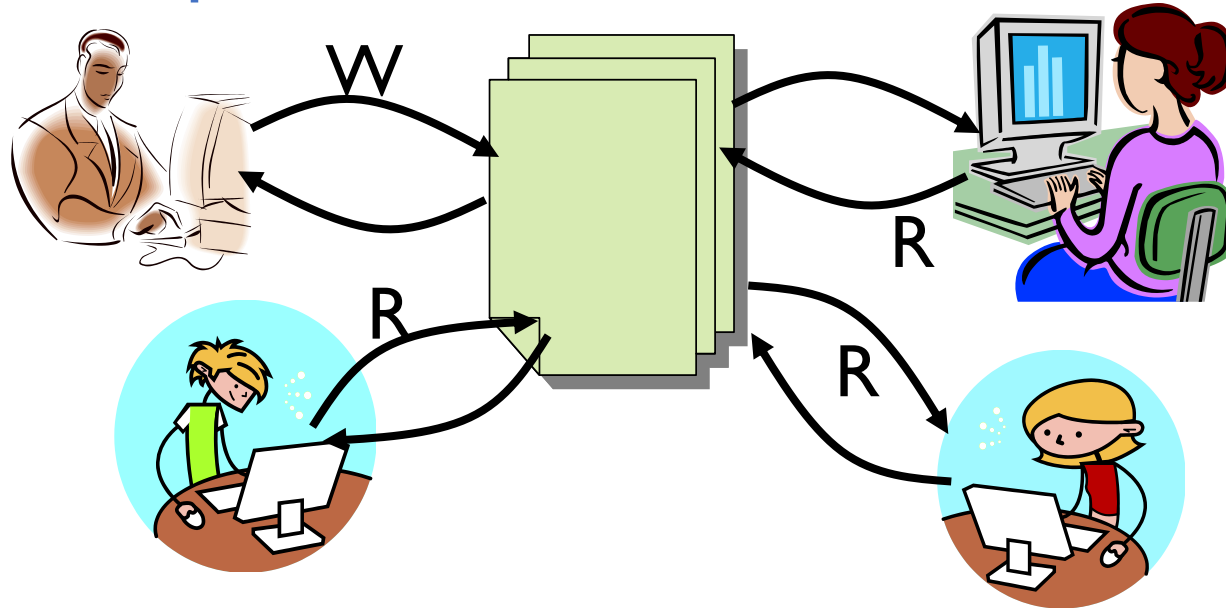# Operating Systems
(Honor Track)

Scheduling 1: Concepts and Classic Policies

Xin Jin

Spring 2023

Recap: Readers/Writers Problem

- Motivation: Consider a shared database
  - Two classes of users:
    - » Readers – never modify database
    - » Writers – read and modify database
  - Is using a single lock on the whole database sufficient?
    - » Like to have many readers at the same time
    - » Only one writer at a time

Recap: Basic Readers/Writers Solution

- Correctness Constraints:
  - Readers can access database when no writers
  - Writers can access database when no readers or writers
  - Only one thread manipulates state variables at a time
- Basic structure of a solution:
  - **`Reader()`**
    ```
    Wait until no writers
    Access database
    Check out – wake up a waiting writer
    ```
  - **`Writer()`**
    ```
    Wait until no active readers or writers
    Access database
    Check out – wake up waiting readers or writer
    ```
  - State variables (Protected by a lock called "lock"):
    » int AR: Number of active readers; initially = 0
    » int WR: Number of waiting readers; initially = 0
    » int AW: Number of active writers; initially = 0
    » int WW: Number of waiting writers; initially = 0
    » Condition okToRead = NIL
    » Condition okToWrite = NIL
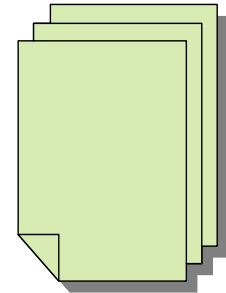
Recap: Code for a Reader

```
Reader() {
  // First check self into system
  acquire(&lock);

  while ((AW + WW) > 0) { // Is it safe to read?
    WR++;                 // No. Writers exist
    cond_wait(&okToRead,&lock);// Sleep on cond var
    WR--;                 // No longer waiting
  }

  AR++;                   // Now we are active!
  release(&lock);

  // Perform actual read-only access
  AccessDatabase(ReadOnly);

  // Now, check out of system
  acquire(&lock);
  AR--;                   // No longer active
  if (AR == 0 && WW > 0)  // No other active readers
    cond_signal(&okToWrite);// Wake up one writer
  release(&lock);
}
```

Recap: Code for a Writer

```
Writer() {
  // First check self into system
  acquire(&lock);

  while ((AW + AR) > 0) { // Is it safe to write?
    WW++;                            // No. Active users exist
    cond_wait(&okToWrite,&lock);   // Sleep on cond var
    WW--;                            // No longer waiting
  }
  AW++;                              // Now we are active!
  release(&lock);

  // Perform actual read/write access
  AccessDatabase(ReadWrite);

  // Now, check out of system
  acquire(&lock);
  AW--;                              // No longer active
  if (WW > 0){                       // Give priority to writers
    cond_signal(&okToWrite);// Wake up one writer
  } else if (WR > 0) {       // Otherwise, wake reader
    cond_broadcast(&okToRead); // Wake all readers
  }
  release(&lock);
}
```

Recap: Group Discussion

- Can readers starve? Consider Reader() entry code:

```
while ((AW + WW) > 0) {  // Is it safe to read?
  WR++;                  // No. Writers exist
  cond_wait(&okToRead,&lock);// Sleep on cond var
  WR--;                  // No longer waiting
}
AR++;                    // Now we are active!
```

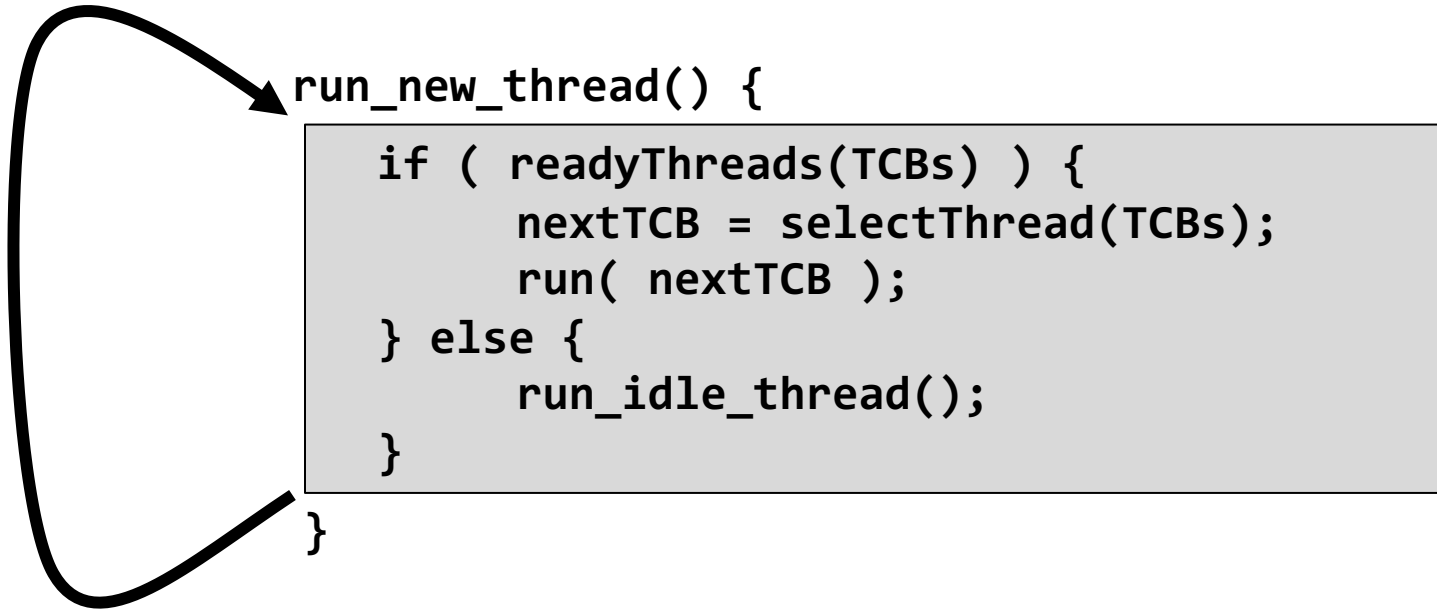- What if we erase the condition check in Reader exit?

```
AR--;                         // No longer active
if (AR == 0 && WW > 0)    // No other active readers
  cond_signal(&okToWrite);// Wake up one writer
```

- Further, what if we turn the signal() into broadcast()

```
AR--;                         // No longer active
cond_broadcast(&okToWrite); // Wake up sleepers
```

- Finally, what if we use only one condition variable (call it
  "**okContinue**") instead of two separate ones?
  - Both readers and writers sleep on this variable
  - Must use broadcast() instead of signal()

Goal for Today

```
run_new_thread() {
    if ( readyThreads(TCBs) ) {
        nextTCB = selectThread(TCBs);
        run( nextTCB );
    } else {
        run_idle_thread();
    }
}
```

- Discussion of Scheduling:
  - Which thread should run on the CPU next?
- Scheduling goals, policies
- Look at a number of different schedulers

Scheduling: All About Queues

Scheduling Assumptions

- CPU scheduling big area of research in early 70's
- Many implicit assumptions for CPU scheduling:
  - One program per user
  - One thread per program
  - Programs are independent
- Clearly, these are unrealistic but they simplify the problem so it can be solved
  - For instance: is "fair" about fairness among users or programs?
    » If I run one compilation job and you run five, you get five times as much CPU on many operating systems
- The high-level goal: Dole out CPU time to optimize some desired parameters of system

USER1	USER2	USER3	USER1	USER2

Time ⟶

Assumption: CPU Bursts
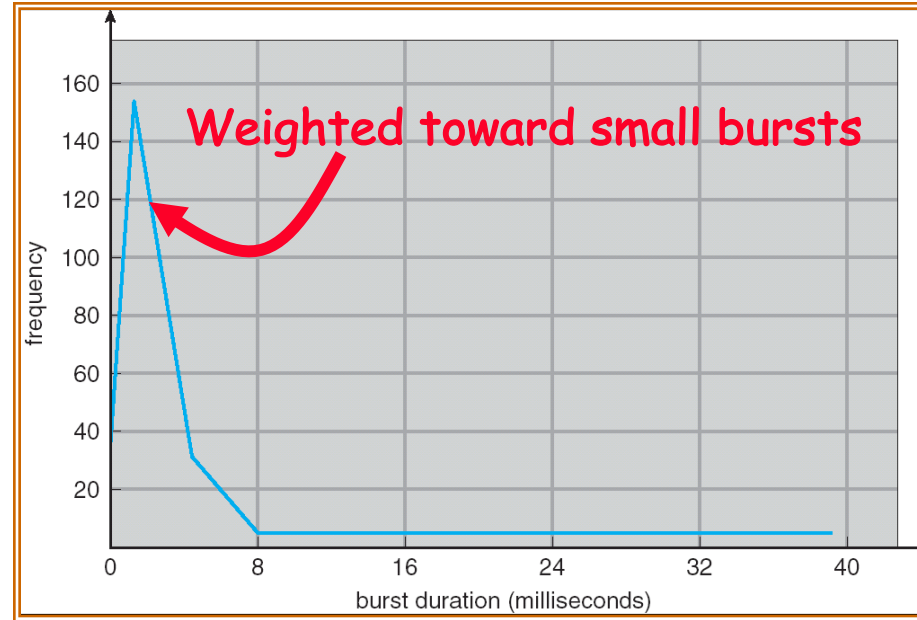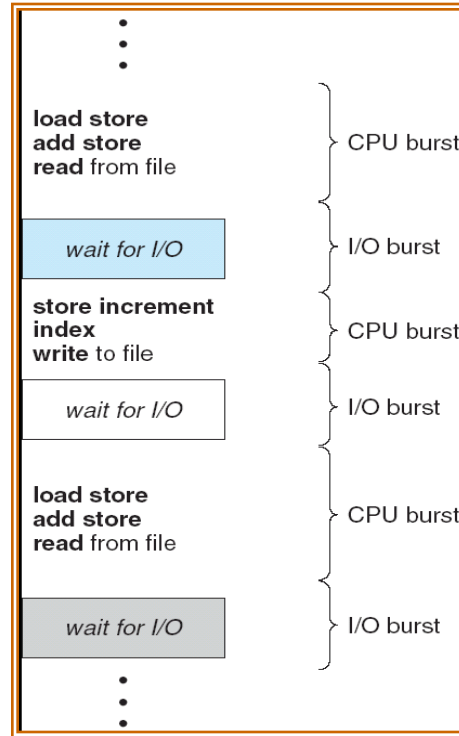
- Execution model: programs alternate between bursts of CPU and I/O
  - Program typically uses the CPU for some period of time, then does I/O, then uses CPU again
  - Each scheduling decision is about which job to give to the CPU for use by its next CPU burst
  - With time slicing, thread may be forced to give up CPU before finishing current CPU burst

Scheduling Policy Goals/Criteria

- Minimize Completion Time
  - Minimize elapsed time to do an operation (or job)
  - Completion time is what the user sees:
    - » Time to echo a keystroke in editor
    - » Time to compile a program
    - » Real-time Tasks: Must meet deadlines imposed by World
- Maximize Throughput
  - Maximize operations (or jobs) per second
  - Throughput related to completion time, but not identical:
    - » Minimizing completion time will lead to more context switching than if you only maximized throughput
  - Two parts to maximizing throughput
    - » Minimize overhead (for example, context-switching)
    - » Efficient use of resources (CPU, disk, memory, etc.)
- Fairness
  - Share CPU among users in some equitable way
  - Fairness is not minimizing average completion time:
    - » Better *average* completion time by making system *less* fair

First-Come, First-Served (FCFS) Scheduling

- First-Come, First-Served (FCFS)
  - Also "First In, First Out" (FIFO) or "Run until done"
    - » In early systems, FCFS meant one program scheduled until done (including I/O)
    - » Now, means keep CPU until thread blocks
- Example:

Process	Burst Time
P_1	24
P_2	3
P_3	3

  - Suppose processes arrive in the order: P_1 , P_2 , P_3
    The Gantt Chart for the schedule is:

P_1				P_2	P_3
0			24	27	30

  - Waiting time for P_1 = 0; P_2 = 24; P_3 = 27
  - Average waiting time: (0 + 24 + 27)/3 = 17
  - Average completion time: (24 + 27 + 30)/3 = 27
- *Convoy effect:* short process stuck behind long process

FCFS Scheduling (Cont.)

- Example continued:
  - Suppose that processes arrive in order: P2 , P3 , P1
    Now, the Gantt chart for the schedule is:

P_2	P_3	P_1

  0   3   6   30

  - Waiting time for P1 = 6; P2 = 0; P3 = 3
  - Average waiting time: (6 + 0 + 3)/3 = 3
  - Average Completion time: (3 + 6 + 30)/3 = 13
- In second case:
  - Average waiting time is much better (before it was 17)
  - Average completion time is better (before it was 27)
- FIFO Pros and Cons:
  - Simple (+)
  - Head-of-line blocking: Short jobs get stuck behind long ones (-)
    » Safeway: Getting milk, always stuck behind cart full of items!
      Upside: get to read about Space Aliens!

Round Robin (RR) Scheduling

- FCFS Scheme: Potentially bad for short jobs!
  - Depends on submit order
  - If you are first in line at supermarket with milk, you don't care who is behind you, on the other hand…

- Round Robin Scheme: Preemption!
  - Each process gets a small unit of CPU time (*time quantum*), usually 10-100 milliseconds
  - After quantum expires, the process is preempted and added to the end of the ready queue.
  - n processes in ready queue and time quantum is $q \Rightarrow$
    » Each process gets $1/n$ of the CPU time
    » In chunks of at most q time units
    » No process waits more than $(n\text{-}1)q$ time units
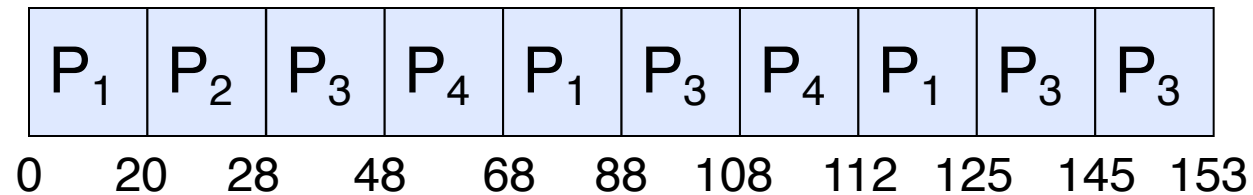
RR Scheduling (Cont.)

- Performance
  - *q* large $\Rightarrow$ FCFS
  - *q* small $\Rightarrow$ Interleaved
  - *q* must be large with respect to context switch, otherwise overhead is too high (all overhead)

Example of RR with Time Quantum = 20

- Example:

Process	Burst Time
P_1	53
P_2	8
P_3	68
P_4	24

  – The Gantt chart is:

P_1	P_2	P_3	P_4	P_1	P_3	P_4	P_1	P_3	P_3

  0   20   28   48   68   88   108   112 125   145 153

  – Waiting time for
  P_1=(68-20)+(112-88)=72
  P_2=(20-0)=20
  P_3=(28-0)+(88-48)+(125-108)=85
  P_4=(48-0)+(108-68)=88

  – Average waiting time = (72+20+85+88)/4=66¼

  – Average completion time = (125+28+153+112)/4 = 104½

- Thus, Round-Robin Pros and Cons:
 – Better for short jobs, Fair (+)
  – Context-switching time adds up for long jobs (-)

Group Discussion

- Topic: FCFS and RR
  - Is RR always better than FCFS in terms of average completion time?
  - Does a smaller quantum in RR always lead to a better average completion time?

- Discuss in groups of two to three students
  - Each group chooses a leader to summarize the discussion
  - In your group discussion, please do not dominate the discussion, and give everyone a chance to speak

Decrease Completion Time

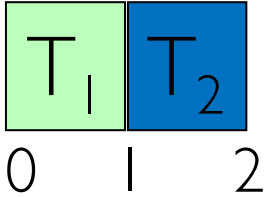- T_1: Burst Length 10

- T_2: Burst Length 1

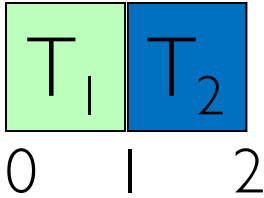- $Q = 10$
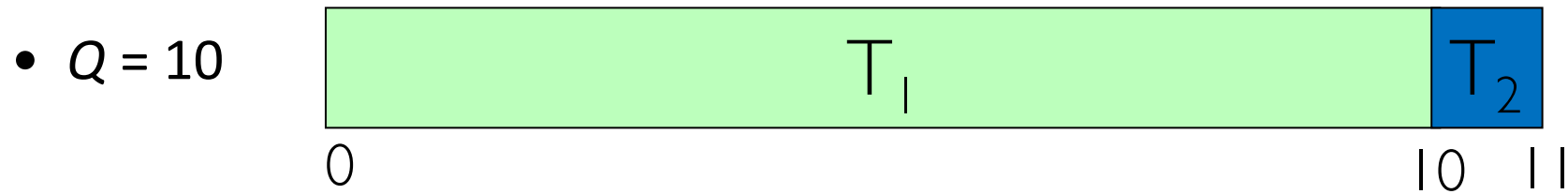
  – Average Completion Time = (10 + 11)/2 = 10.5

- $Q = 5$
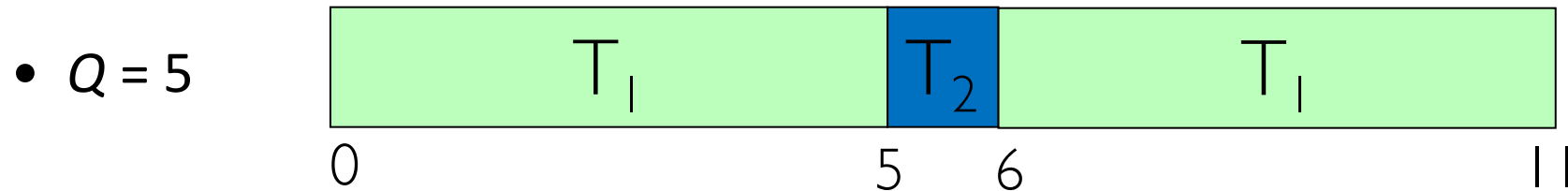
  – Average Completion Time = (6 + 11)/2 = 8.5

Same Completion Time

- T_1: Burst Length 1
- T_2: Burst Length 1

- $Q = 10$
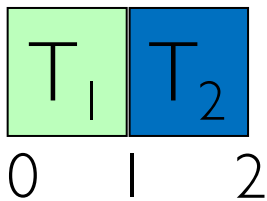
T_1	T_2

  0   1   2

  – Average Completion Time = (1 + 2)/2 = 1.5

- $Q = 1$

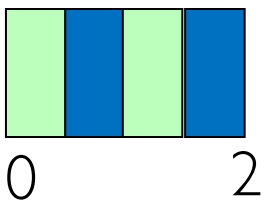 | T_1 | T_2 |
  |---|---|

  0   1   2

  – Average Completion Time = (1 + 2)/2 = 1.5

Increase Completion Time

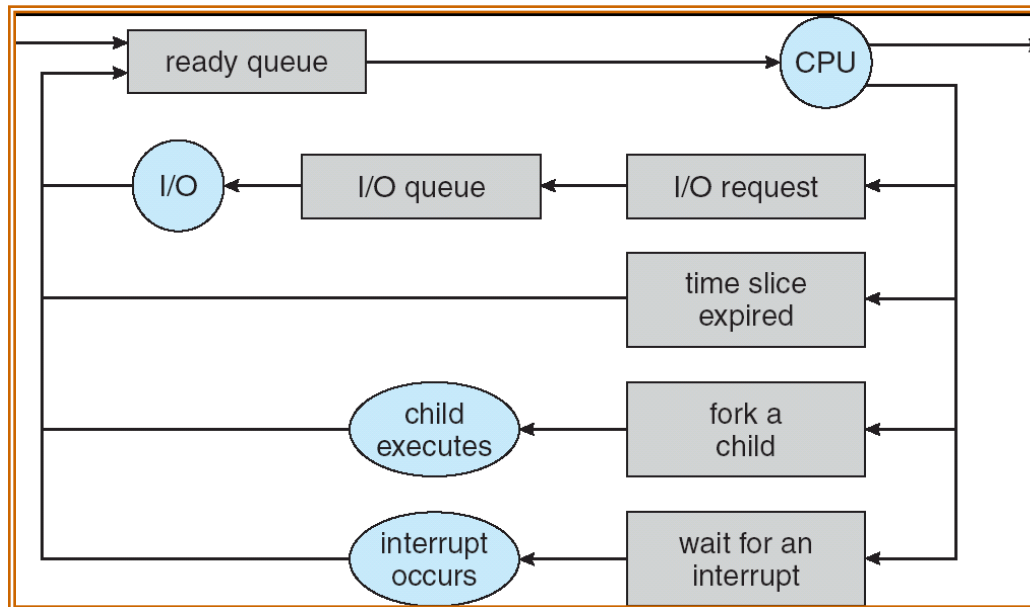- T_1: Burst Length 1
- T_2: Burst Length 1

- $Q = 1$

  - Average Completion Time = (1 + 2)/2 = 1.5

- $Q = 0.5$

 - Average Completion Time = (1.5 + 2)/2 = 1.75

How to Implement RR in the Kernel?

- FIFO Queue, as in FCFS
- But preempt job after quantum expires, and send it to the back of the queue
  - How? Timer interrupt!
  - And, of course, careful synchronization

Round-Robin Discussion

- How do you choose time slice?
  - What if too big?
    - » Waiting time suffers
  - What if infinite (∞)?
    - » Get back FIFO
  - What if time slice too small?
    - » Throughput suffers!
- Actual choices of time slice:
  - Initially, UNIX time slice one second:
    - » Worked ok when UNIX was used by one or two people.
    - » What if three compilations going on? 3 seconds to echo each keystroke!
  - Need to balance short-job performance and long-job throughput
    - » Typical time slice today is between 10ms – 100ms
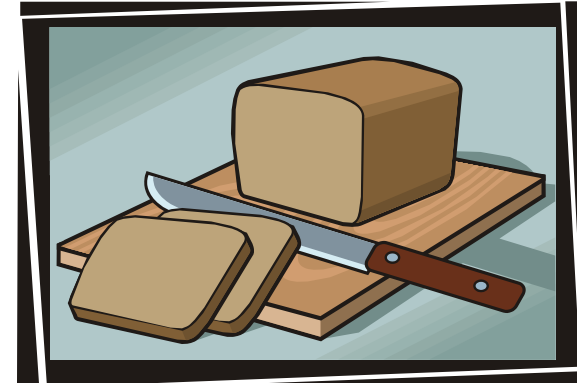
Comparisons between FCFS and Round Robin

- Assuming zero-cost context-switching time, is RR always better than FCFS?
- Simple example: 10 jobs, each take 100s of CPU time
  RR scheduler quantum of 1s
  All jobs start at the same time

- Completion Times:

| Job #	FIFO	RR
1	100	991
2	200	992
…	…	…
9	900	999
10	1000	1000

  - Both RR and FCFS finish at the same time
  - Average completion time is much worse under RR!
    » Bad when all jobs same length
- Also: Cache state must be shared between all jobs with RR but can be devoted to each job with FIFO
  - Total time for RR longer even for zero-cost switch!

Earlier Example with Different Time Quantum

Best FCFS:	P$_2$ [8]	P$_4$ [24]	P$_1$ [53]	P$_3$ [68]

0   8   32   85   153
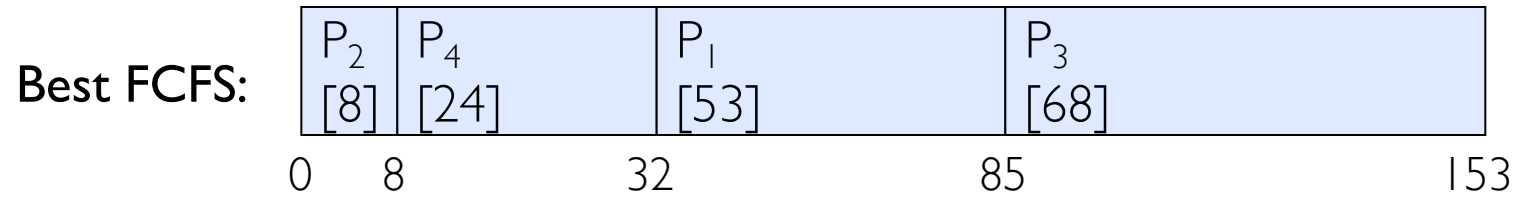
	Quantum	P$_1$	P$_2$	P$_3$	P$_4$	Average
Wait Time	Best FCFS	32	0	85	8	31¼
	Q = 1	84	22	85	57	62
	Q = 5	82	20	85	58	61¼
	Q = 8	80	8	85	56	57¼
	Q = 10	82	10	85	68	61¼
	Q = 20	72	20	85	88	66¼
	Worst FCFS	68	145	0	121	83½
Completion Time	Best FCFS	85	8	153	32	69½
	Q = 1	137	30	153	81	100½
	Q = 5	135	28	153	82	99½
	Q = 8	133	16	153	80	95½
	Q = 10	135	18	153	92	99½
	Q = 20	125	28	153	112	104½
	Worst FCFS	121	153	68	145	121¾

Handling Differences in Importance: Strict Priority Scheduling

Priority 3	→	Job 1	→	Job 2	→	Job 3
Priority 2	→	Job 4				
Priority 1						
Priority 0	→	Job 5	→	Job 6	→	Job 7
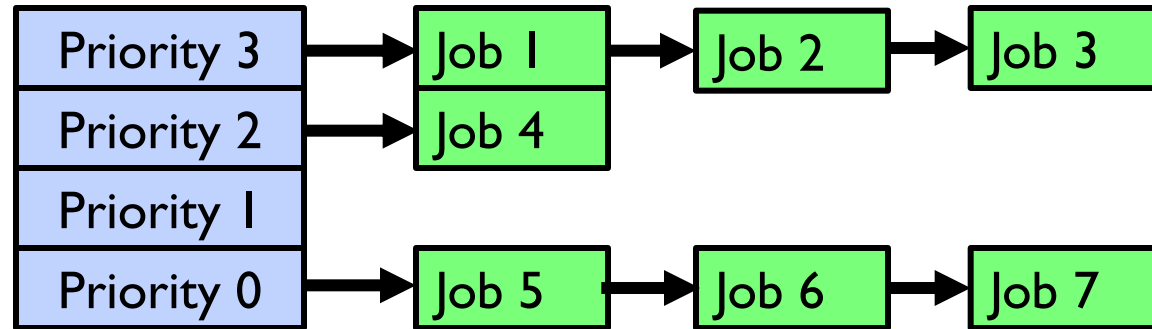
- Execution Plan
  - Always execute highest-priority runable jobs to completion
  - Each queue can be processed in RR with some time-quantum
- Problems:
  - Starvation:
    - » Lower priority jobs don't get to run because higher priority jobs
  - Deadlock: Priority Inversion
    - » Happens when low priority task has lock needed by high-priority task
    - » Usually involves third, intermediate priority task preventing high-priority task from running
- How to fix problems?
  - Dynamic priorities: adjust base-level priority up or down based on heuristics about interactivity, locking, burst behavior, etc…

Scheduling Fairness

- What about fairness?
  - Strict fixed-priority scheduling between queues is unfair (run highest, then next, etc):
    - » long running jobs may never get CPU
    - » Urban legend: In Multics, shut down machine, found 10-year-old job $\Rightarrow$ Ok, probably not...
  - Must give long-running jobs a fraction of the CPU even when there are shorter jobs to run
  - Tradeoff: fairness gained by hurting avg completion time!

Scheduling Fairness

- How to implement fairness?
  - Could give each queue some fraction of the CPU
    - » What if one long-running job and 100 short-running ones?
    - » Like express lanes in a supermarket—sometimes express lanes get so long, get better service by going into one of the other lines
  - Could increase priority of jobs that don't get service
    - » What is done in some variants of UNIX
    - » This is ad hoc—what rate should you increase priorities?
    - » And, as system gets overloaded, no job gets CPU time, so everyone increases in priority$\Rightarrow$Interactive jobs suffer

What if we Knew the Future?

- Could we always mirror best FCFS?
- Shortest Job First (SJF):
  - Run whatever job has least amount of computation to do
  - Sometimes called "Shortest Time to Completion First" (STCF)
- Shortest Remaining Time First (SRTF):
  - Preemptive version of SJF: if job arrives and has a shorter time to completion than the remaining time on the current job, immediately preempt CPU
  - Sometimes called "Shortest Remaining Time to Completion First" (SRTCF)
- These can be applied to whole program or current CPU burst
  - Idea is to get short jobs out of the system
  - Big effect on short jobs, only small effect on long ones
 - Result is better average completion time

Discussion

- SJF/SRTF are the best you can do at minimizing average completion time
  - Provably optimal (SJF among non-preemptive, SRTF among preemptive)
  - Since SRTF is always at least as good as SJF, focus on SRTF

- Comparison of SRTF with FCFS
  - What if all jobs the same length?
    » SRTF becomes the same as FCFS (i.e. FCFS is best can do if all jobs the same length)
  - What if jobs have varying length?
    » SRTF: short jobs not stuck behind long ones

Example to illustrate benefits of SRTF

A or B

C

C's I/O   C's I/O   C's I/O
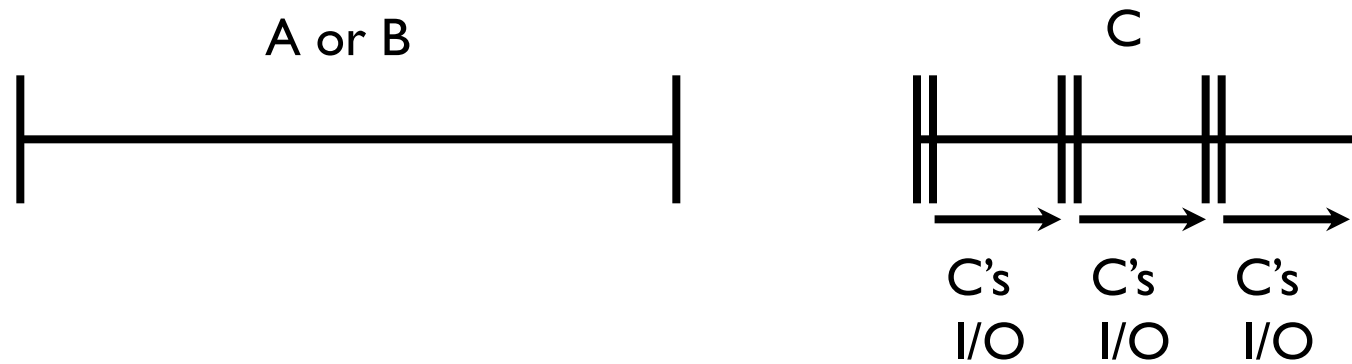
- Three jobs:
  - A, B: both CPU bound, run for week
    C: I/O bound, loop 1ms CPU, 9ms disk I/O
  - If only one at a time, C uses 90% of the disk, A or B could use 100% of the CPU
- With FCFS:
  - Once A or B get in, keep CPU for two weeks
- What about RR or SRTF?
  - Easier to see with a timeline

SRTF Example continued:
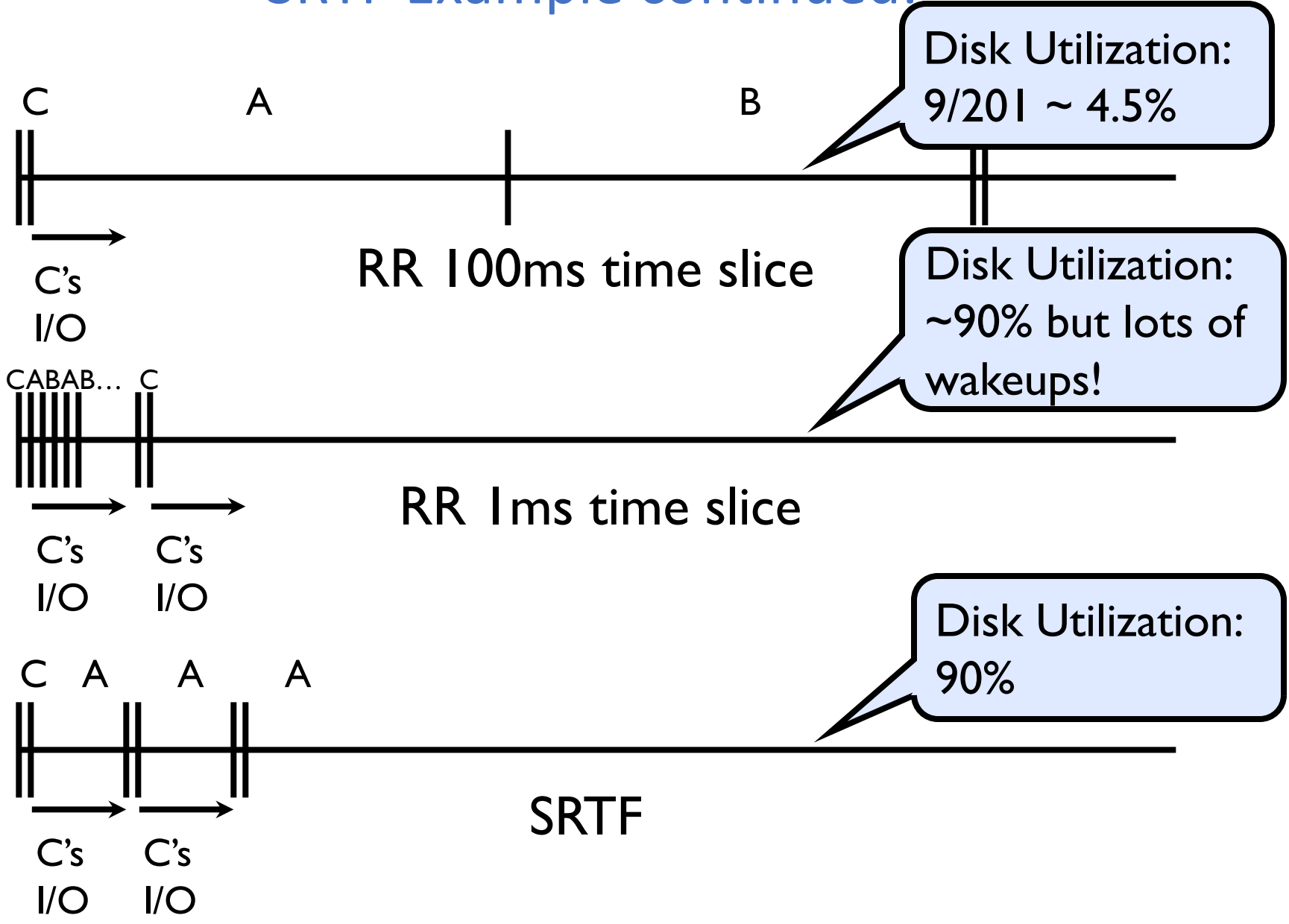
SRTF Further discussion

- Starvation
  - SRTF can lead to starvation if many small jobs!
  - Large jobs never get to run

- Somehow need to predict future
  - How can we do this?
  - Some systems ask the user
    » When you submit a job, have to say how long it will take
    » To stop cheating, system kills job if takes too long
  - But: hard to predict job's runtime even for non-malicious users

- Bottom line, can't really know how long job will take
  - However, can use SRTF as a yardstick
    for measuring other policies
  - Optimal, so can't do any better

- SRTF Pros & Cons
  - Optimal (average completion time) (+)
  - Hard to predict future (-)
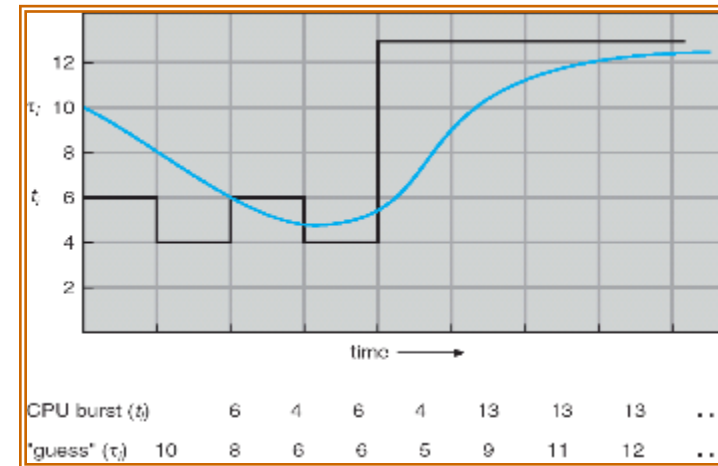 - Unfair (-)

RESEARCH DEPT.

Predicting the Length of the Next CPU Burst

- **Adaptive:** Changing policy based on past behavior
  - CPU scheduling, in virtual memory, in file systems, etc
  - Works because programs have predictable behavior
    - » If program was I/O bound in past, likely in future
    - » If computer behavior were random, wouldn't help
- Example: SRTF with estimated burst length
  - Use an estimator function on previous bursts:
    Let t_{n-1}, t_{n-2}, t_{n-3}, etc. be previous CPU burst lengths.
    Estimate next burst $\tau_n = f(t_{n-1}, t_{n-2}, t_{n-3}, \ldots)$
  - Function f could be one of many different time series estimation schemes (Kalman filters, etc)
  - For instance, exponential averaging
    $\tau_n = \alpha t_{n-1} + (1-\alpha)\tau_{n-1}$
    with $(0 < \alpha \leq 1)$

| CPU burst (t_i) | | 6 | 4 | 6 | 4 | 13 | 13 | 13 | ... |
| "guess" (τ_i) | 10 | 8 | 6 | 6 | 5 | 9 | 11 | 12 | ... |

Lottery Scheduling

- Yet another alternative: Lottery Scheduling
  - Give each job some number of lottery tickets
  - On each time slice, randomly pick a winning ticket
  - On average, CPU time is proportional to number of tickets given to each job
- How to assign tickets?
  - To approximate SRTF, short running jobs get more, long running jobs get fewer
  - To avoid starvation, every job gets at least one ticket (everyone makes progress)
- Advantage over strict priority scheduling: behaves gracefully as load changes
  - Adding or deleting a job affects all jobs proportionally, independent of how many tickets each job possesses

- Lottery Scheduling Example
  - Assume short jobs get 10 tickets, long jobs get 1 ticket

| # short jobs/ # long jobs	% of CPU each short jobs gets	% of CPU each long jobs gets
1/1	91%	9%
0/2	N/A	50%
2/0	50%	N/A
10/1	9.9%	0.99%
1/10	50%	5%

  - What if too many short jobs to give reasonable completion time?
    » If load average is 100, hard to make progress
    » One approach: log some user out

Multi-Level Feedback Scheduling


Long-Running Compute Tasks Demoted to Low Priority
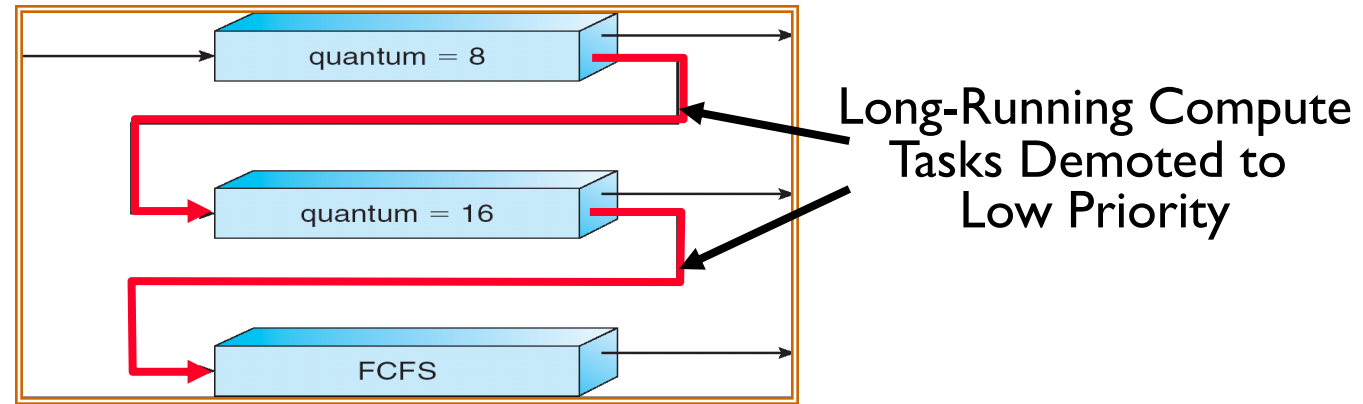
- Another method for exploiting past behavior (first use in CTSS)
  - Multiple queues, each with different priority
    » Higher priority queues often considered "foreground" tasks
  - Each queue has its own scheduling algorithm
    » e.g. foreground – RR, background – FCFS
    » Sometimes multiple RR priorities with quantum increasing exponentially (highest:1ms, next: 2ms, next: 4ms, etc)
- Adjust each job's priority as follows (details vary)
  - Job starts in highest priority queue
  - If timeout expires, drop one level
  - If timeout doesn't expire, push up one level (or to top)

Scheduling Details
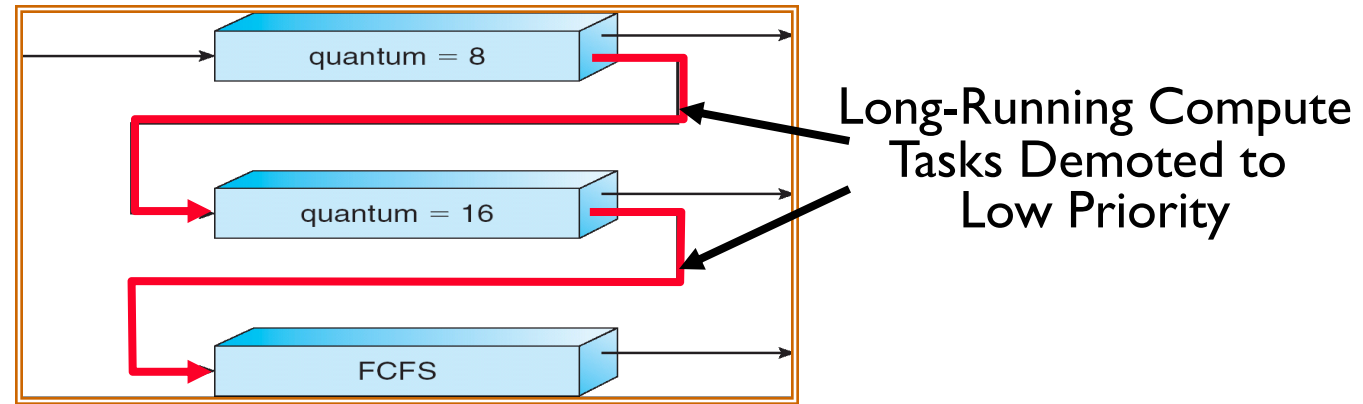
- Result approximates SRTF:
  - CPU bound jobs drop like a rock
  - Short-running I/O bound jobs stay near top
- Scheduling must be done between the queues
  - Fixed priority scheduling:
    » serve all from highest priority, then next priority, etc.
  - Time slice:
    » each queue gets a certain amount of CPU time
    » e.g., 70% to highest, 20% next, 10% lowest

Scheduling Details
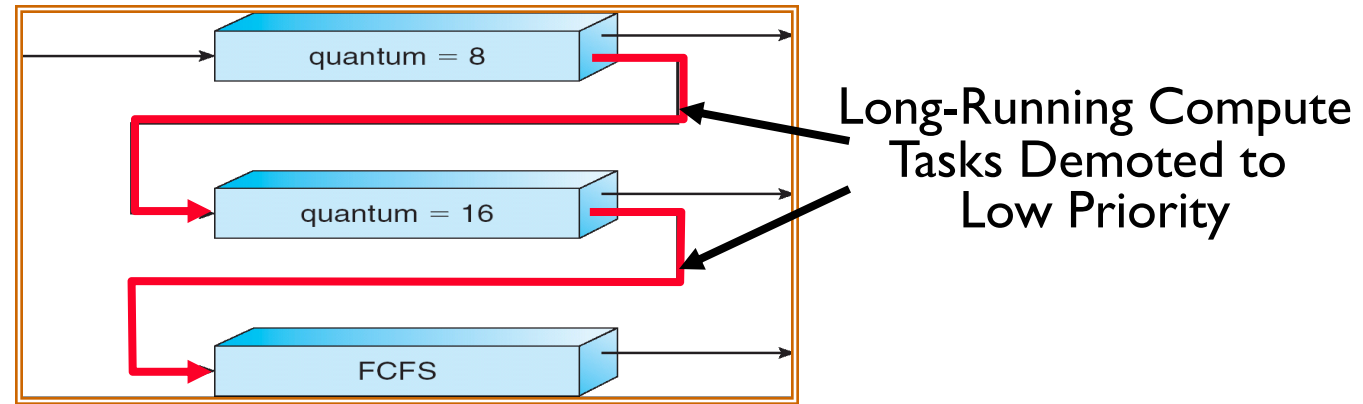
- Countermeasure: user action that can foil intent of the OS designers
  - For multilevel feedback, put in a bunch of meaningless I/O to keep job's priority high
  - Of course, if everyone did this, wouldn't work!
- Example of Othello program:
  - Playing against competitor, so key was to do computing at higher priority than the competitors.
    » Put in printf's, run much faster!

How to Handle Simultaneous Mix of Diff Types of Apps?

- Consider mix of interactive and high throughput apps:
  - How to best schedule them?
  - How to recognize one from the other?
    - » Do you trust app to say that it is "interactive"?
  - Should you schedule the set of apps identically on servers, workstations, pads, and cellphones?
- For instance, is Burst Time (observed) useful to decide which application gets CPU time?
  - Short Bursts $\Rightarrow$ Interactivity $\Rightarrow$ High Priority?
- Assumptions encoded into many schedulers:
  - Apps that sleep a lot and have short bursts must be interactive apps – they should get high priority
  - Apps that compute a lot should get low(er?) priority, since they won't notice intermittent bursts from interactive apps
- Hard to characterize apps:
  - What about apps that sleep for a long time, but then compute for a long time?
  - Or, what about apps that must run under all circumstances (say periodically)

Conclusion

- **Round-Robin Scheduling:**
  - Give each thread a small amount of CPU time when it executes; cycle between all ready threads
  - Pros: Better for short jobs
- **Shortest Job First (SJF)/Shortest Remaining Time First (SRTF):**
  - Run whatever job has the least amount of computation to do/least remaining amount of computation to do
  - Pros: Optimal (average completion time)
  - Cons: Hard to predict future, Unfair
- **Lottery Scheduling:**
  - Give each thread a priority-dependent number of tokens
    (short tasks$\Rightarrow$more tokens)
- **Multi-Level Feedback Scheduling:**
  - Multiple queues of different priorities and scheduling algorithms
  - Automatic promotion/demotion of process priority in order to approximate SJF/SRTF