

CS162
Operating Systems and
Systems Programming
Lecture 10

Scheduling I: Concepts and Classic Policies
(Finish: Synchronization Primitives)

September 28th, 2021

Prof. Ion Stoica

<http://cs162.eecs.Berkeley.edu>

Recall: Monitors and Condition Variables

- **Monitor**: a lock and zero or more condition variables for managing concurrent access to shared data
 - Use of Monitors is a programming paradigm
 - Some languages like Java provide monitors in the language
- **Condition Variable**: a queue of threads waiting for something *inside* a critical section
 - Key idea: allow sleeping inside critical section by atomically releasing lock at time we go to sleep
 - Contrast to semaphores: Can't wait inside critical section
- Operations:
 - **Wait (&lock)**: Atomically release lock and go to sleep. Re-acquire lock later, before returning.
 - **Signal ()**: Wake up one waiter, if any
 - **Broadcast ()**: Wake up all waiters
- Rule: **Must hold lock when doing condition variable ops!**

Can we construct Monitors from Semaphores?

- Locking aspect is easy: Just use a mutex
- Can we implement condition variables this way?

```
Wait(Semaphore *thesema)  { semaP(thesema); }
Signal(Semaphore *thesema) { semaV(thesema); }
```
- Does this work better?

```
Wait(Lock *thelock, Semaphore *thesema) {
    release(thelock);
    semaP(thesema);
    acquire(thelock);
}
Signal(Semaphore *thesema) {
    semaV(thesema);
}
```

Construction of Monitors from Semaphores (con't)

- Problem with previous try:
 - P and V are commutative – result is the same no matter what order they occur
 - Condition variables are NOT commutative
- Does this fix the problem?


```
Wait(Lock *thelock, Semaphore *thesema) {  
    release(thelock);  
    semaP(thesema);  
    acquire(thelock);  
}  
Signal(Semaphore *thesema) {  
    if semaphore queue is not empty  
        semaV(thesema);  
}
```

 - Not legal to look at contents of semaphore queue
 - There is a race condition – signaler can slip in after lock release and before waiter executes semaphore.P()
- It is actually possible to do this correctly
 - Complex solution for Hoare scheduling in book
 - Can you come up with simpler Mesa-scheduled solution?

Mesa Monitor Conclusion

- Monitors represent the synchronization logic of the program
 - Wait if necessary
 - Signal when change something so any waiting threads can proceed
- Typical structure of monitor-based program:

```
lock
while (need to wait) {
    condvar.wait();
}
unlock
```




Check and/or update
state variables
Wait if necessary

do something so no need to wait

```
lock

condvar.signal();

unlock
```



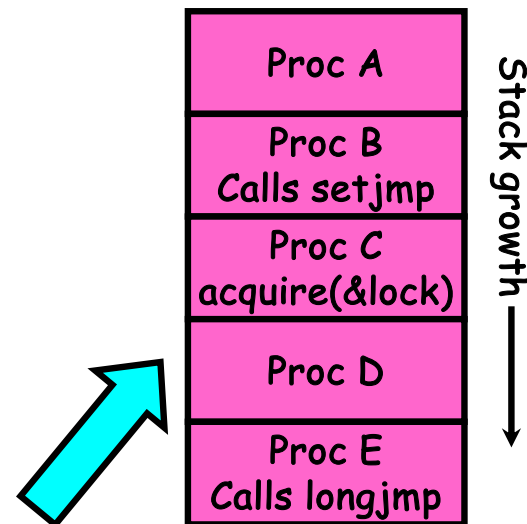
Check and/or update
state variables

C-Language Support for Synchronization

- C language: Pretty straightforward synchronization
 - Just make sure you know *all* the code paths out of a critical section

```
int Rtn() {  
    acquire(&lock);  
    ...  
    if (exception) {  
        release(&lock);  
        return errReturnCode;  
    }  
    ...  
    release(&lock);  
    return OK;  
}
```

- Watch out for `setjmp/longjmp`!
 - » Can cause a non-local jump out of procedure
 - » In example, procedure E calls `longjmp`, popping stack back to procedure B
 - » If Procedure C had `lock.acquire`, problem!



Concurrency and Synchronization in C

- Harder with more locks

```
void Rtn() {
    lock1.acquire();
    ...
    if (error) {
        lock1.release();
        return;
    }
    ...
    lock2.acquire();
    ...
    if (error) {
        lock2.release();
        lock1.release();
        return;
    }
    ...
    lock2.release();
    lock1.release();
}
```

- Is goto a solution???

```
void Rtn() {
    lock1.acquire();
    ...
    if (error) {
        goto release_lock1_and_return;
    }
    ...
    lock2.acquire();
    ...
    if (error) {
        goto release_both_and_return;
    }
    ...
release_both_and_return:
    lock2.release();
release_lock1_and_return:
    lock1.release();
}
```

C++ Language Support for Synchronization

- Languages with exceptions like C++
 - Languages that support exceptions are problematic (easy to make a non-local exit without releasing lock)
 - Consider:

```
void Rtn() {  
    lock.acquire();  
    ...  
    DoFoo();  
    ...  
    lock.release();  
}  
void DoFoo() {  
    ...  
    if (exception) throw errException;  
    ...  
}
```

- Notice that an exception in DoFoo() will exit without releasing the lock!

C++ Language Support for Synchronization (con't)

- Must catch all exceptions in critical sections
 - Catch exceptions, release lock, and re-throw exception:

```
void Rtn() {  
    lock.acquire();  
    try {  
        ...  
        DoFoo();  
        ...  
    } catch (...) {           // catch exception  
        lock.release();      // release lock  
        throw;               // re-throw the exception  
    }  
    lock.release();  
}  
void DoFoo() {  
    ...  
    if (exception) throw errException;  
    ...  
}
```

Much better: C++ Lock Guards

```
#include <mutex>
int global_i = 0;
std::mutex global_mutex;

void safe_increment() {
    std::lock_guard<std::mutex> lock(global_mutex);
    ...
    global_i++;
    // Mutex released when 'lock' goes out of scope
}
```

Python with Keyword

- More versatile than we show here (can be used to close files, database connections, etc.)

```
lock = threading.Lock()
...
with lock: # Automatically calls acquire()
    some_var += 1
...
# release() called however we leave block
```

Java synchronized Keyword

- Every Java object has an associated lock:
 - Lock is acquired on entry and released on exit from a **synchronized** method
 - Lock is properly released if exception occurs inside a **synchronized** method
 - Mutex execution of synchronized methods (beware deadlock)

```
class Account {  
    private int balance;  
  
    // object constructor  
    public Account (int initialBalance) {  
        balance = initialBalance;  
    }  
    public synchronized int getBalance() {  
        return balance;  
    }  
    public synchronized void deposit(int amount) {  
        balance += amount;  
    }  
}
```

Java Support for Monitors

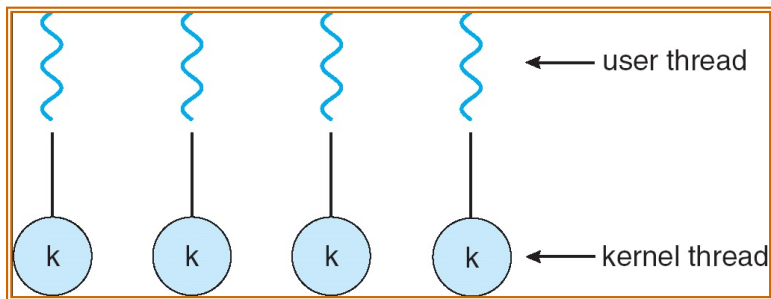
- Along with a lock, every object has a single condition variable associated with it
- To wait inside a synchronized method:
 - `void wait();`
 - `void wait(long timeout);`
- To signal while in a synchronized method:
 - `void notify();`
 - `void notifyAll();`

Announcements

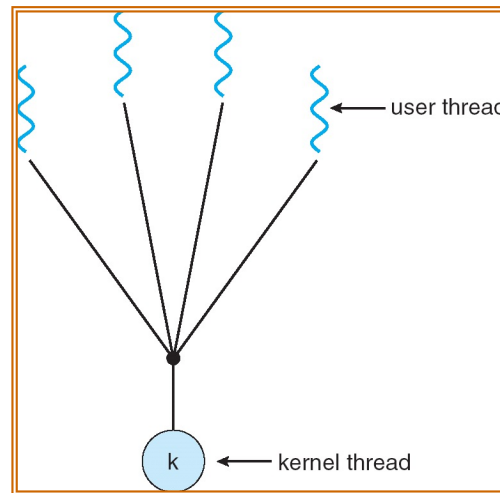
- Midterm I: Tomorrow, Wednesday, 9/29, 5-7PM
 - Video Proctored over Zoom – please read the proctoring policies carefully
 - You can have one handwritten cheat sheet
- HW 2 is due next Monday, 10/4
- Proj I due next, Wednesday, 10/6

Recall: User/Kernel Threading Models

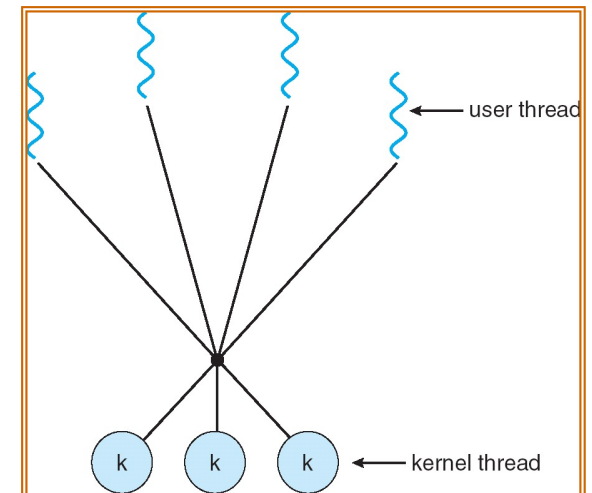
Almost all current implementations



**Simple One-to-One
Threading Model**



Many-to-One

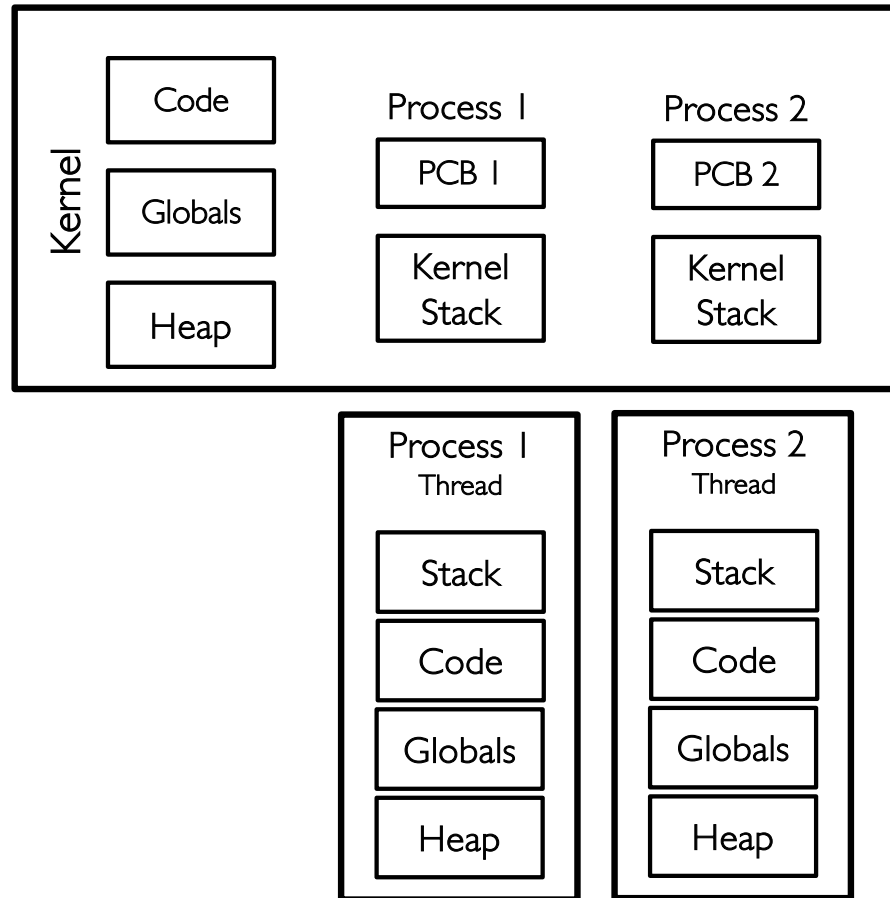


Many-to-Many

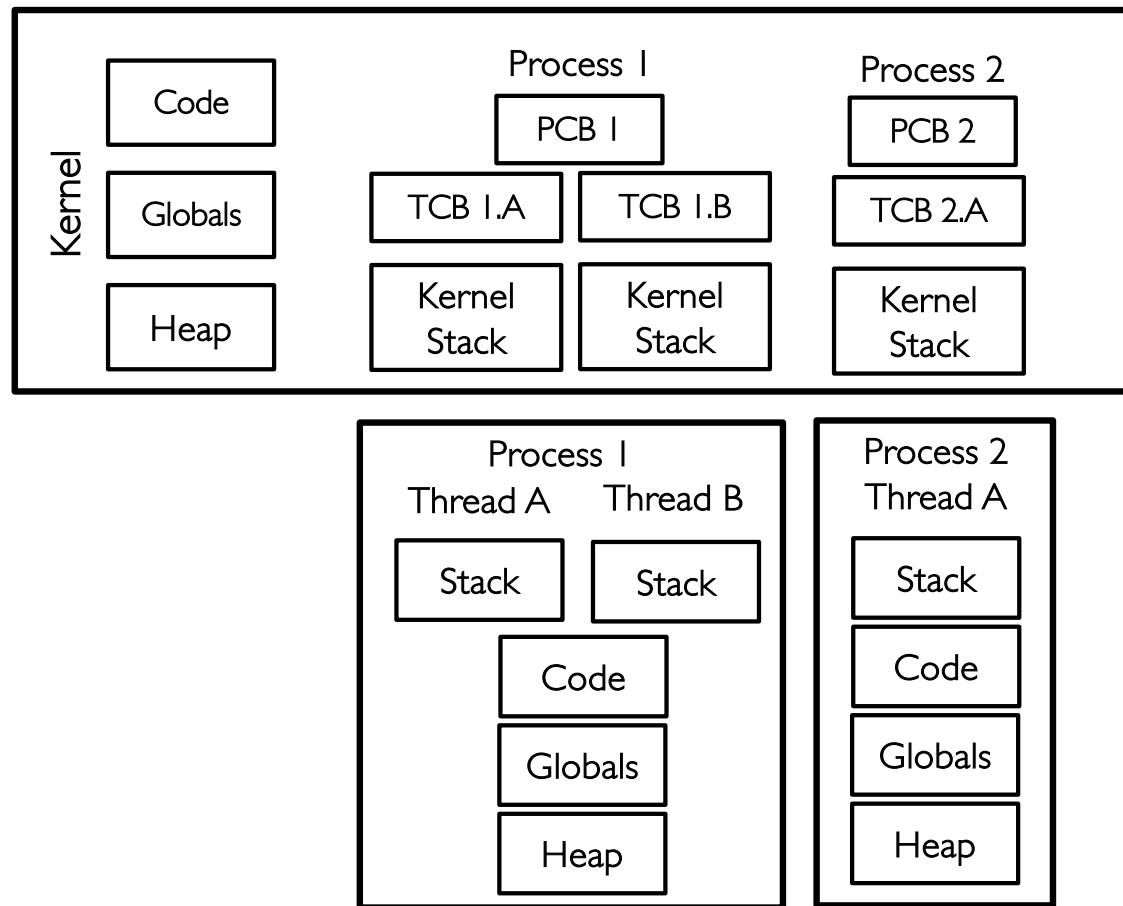
Recall: Thread State in the Kernel

- For every thread in a process, the kernel maintains:
 - The thread's TCB
 - A kernel stack used for syscalls/interrupts/traps
 - » This kernel-state is sometimes called the “kernel thread”
 - » The “kernel thread” is suspended (but ready to go) when thread is running in user-space
- Additionally, some threads just do work in the kernel
 - Still has TCB
 - Still has kernel stack
 - But not part of any process, and never executes in user mode

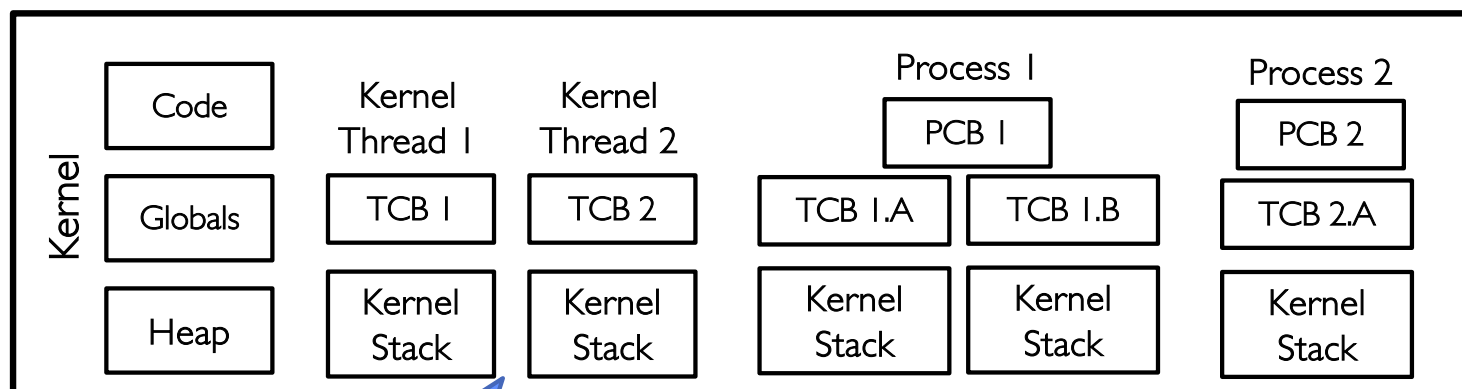
Kernel Structure So Far (1/3)



Kernel Structure So Far (2/3)

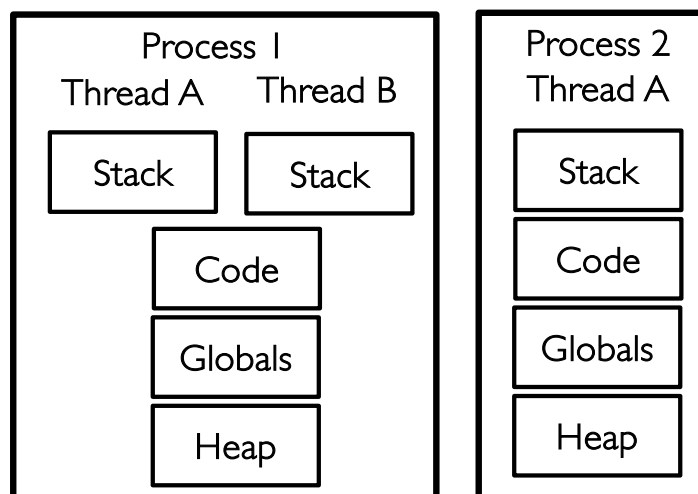


Kernel Structure So Far (3/3)



These two threads:

- Are used internally by the kernel
- Don't correspond to any particular user thread or process

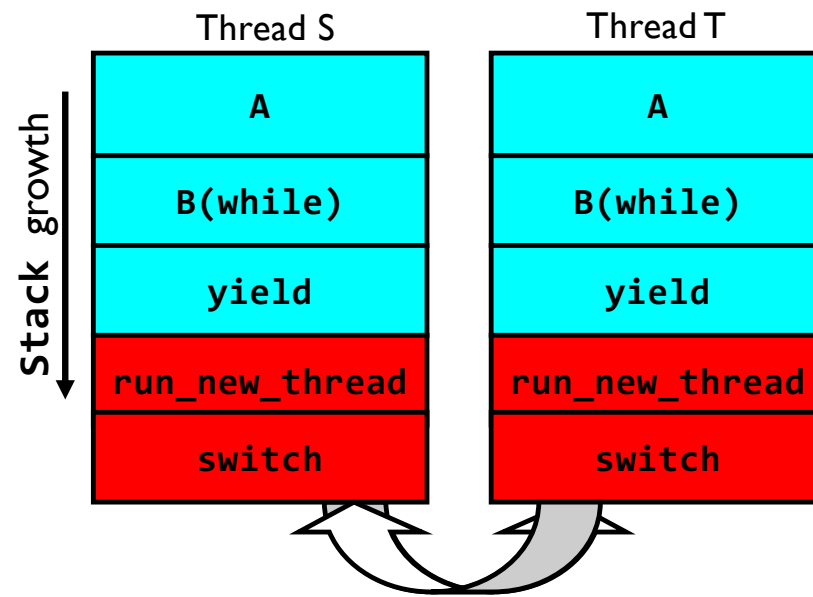


Recall: Multithreaded Stack Example

- Consider the following code blocks:

```
proc A() {  
    B();  
}  
proc B() {  
    while(TRUE) {  
        yield();  
    }  
}
```

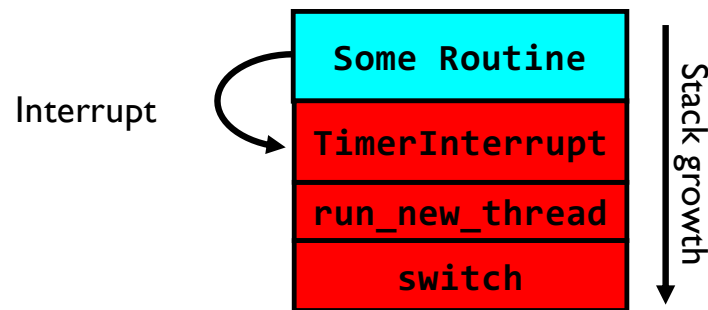
- Suppose we have 2 threads:
 - Threads S and T



Thread S's switch returns to Thread T's (and vice versa)

Use of Timer Interrupt to Return Control

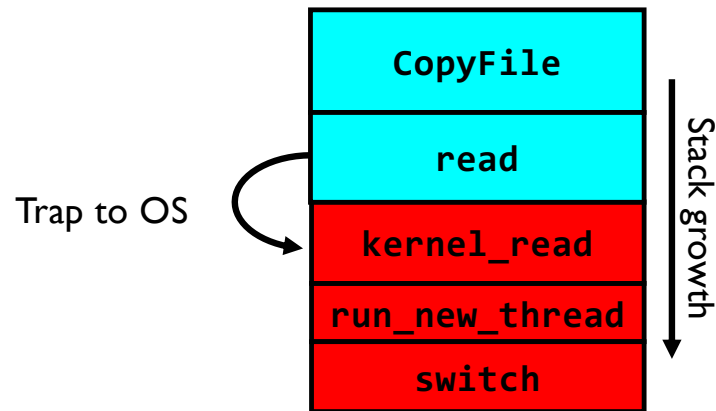
- Solution to our dispatcher problem
 - Use the timer interrupt to force scheduling decisions



- Timer Interrupt routine:

```
TimerInterrupt() {  
    DoPeriodicHouseKeeping();  
    run_new_thread();  
}
```

What happens when thread blocks on I/O?



- What happens when a thread requests a block of data from the file system?
 - User code invokes a system call
 - Read operation is initiated
 - Run new thread/switch
- Thread communication similar
 - Wait for Signal/Join
 - Networking

Recall: Famous Quote WRT Scheduling: Dennis Richie

Dennis Richie,
Unix V6, slp.c:

```
2230  /*
2231  * If the new process paused because it was
2232  * swapped out, set the stack level to the last call
2233  * to savu(u_ssav). This means that the return
2234  * which is executed immediately after the call to aretu
2235  * actually returns from the last routine which did
2236  * the savu.
2237  *
2238  * You are not expected to understand this.
2239  */
```

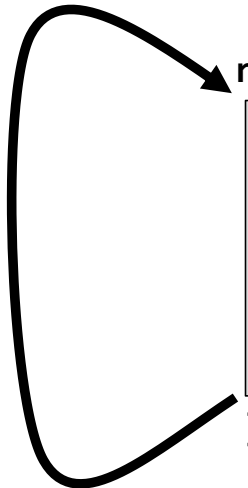
"If the new process paused because it was swapped out, set the stack level to the last call to savu(u_ssav). This means that the return which is executed immediately after the call to aretu actually returns from the last routine which did the savu."

"You are not expected to understand this."

Source: Dennis Ritchie, Unix V6 slp.c (context-switching code) as per The Unix Heritage Society(tuhs.org); gif by Eddie Koehler.

Included by Ali R. Butt in CS3204 from Virginia Tech

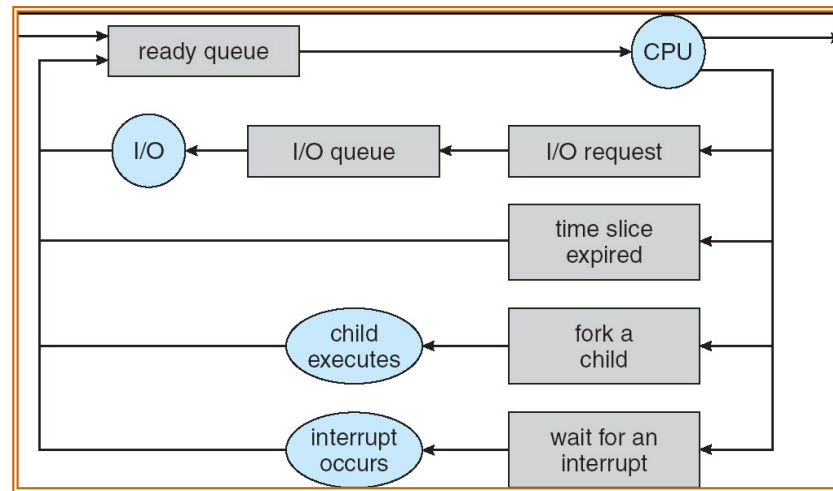
Goal for Today



```
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

Recall: Scheduling



- Question: How is the OS to decide which of several tasks to take off a queue?
- **Scheduling**: deciding which threads are given access to resources from moment to moment
 - Often, we think in terms of CPU time, but could also think about access to resources like network BW or disk access

Scheduling: All About Queues

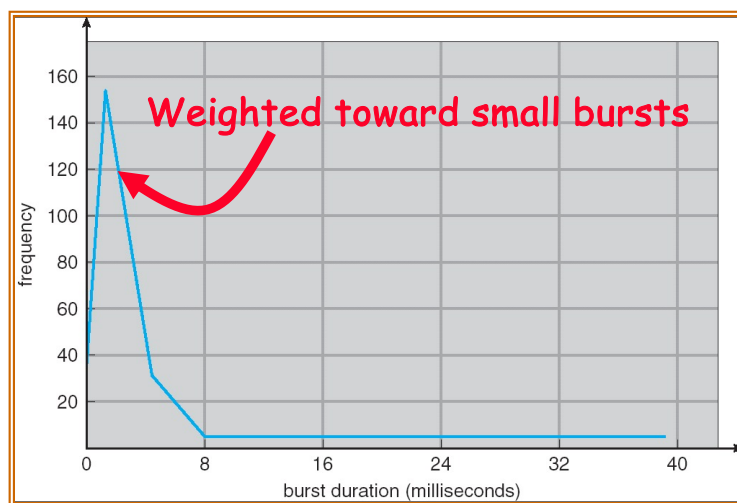
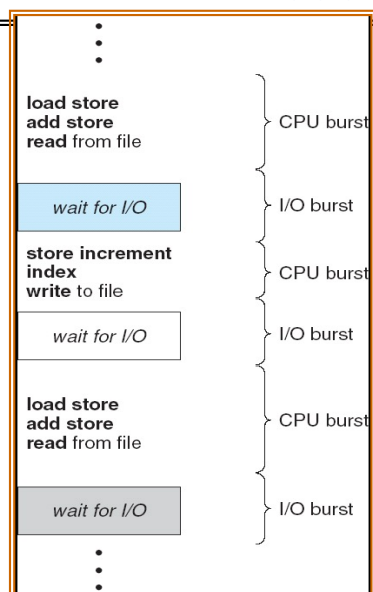


Scheduling Assumptions

- 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



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 timeslicing, thread may be forced to give up CPU before finishing current CPU burst

Scheduling Policy Goals/Criteria

- Minimize Response Time
 - Minimize elapsed time to do an operation (or job)
 - Response 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 response time, but not identical:
 - » Minimizing response 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 response time:
 - » Better average response time by making system less fair

First-Come, First-Served (FCFS) Scheduling

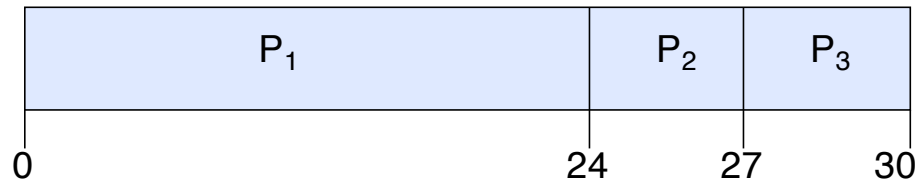
- First-Come, First-Served (FCFS)
 - Also “First In, First Out” (FIFO) or “Run until done”
 - » In early systems, FCFS meant one program scheduled until done (including I/O)
 - » Now, means keep CPU until thread blocks



- Example:

Process	Burst Time
P_1	24
P_2	3
P_3	3

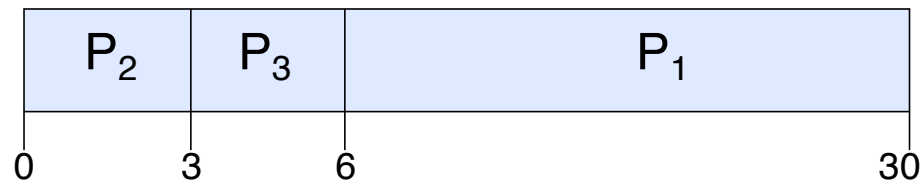
 - Suppose processes arrive in the order: P_1, P_2, P_3
 - The Gantt Chart for the schedule is:



- 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:



- 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 (+)
 - 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 $(n-1)q$ time units



RR Scheduling (Cont.)

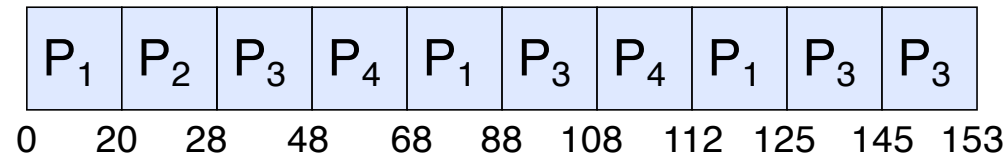
- Performance
 - q large \Rightarrow FCFS
 - q small \Rightarrow Interleaved (really small \Rightarrow hyperthreading?)
 - q must be large with respect to context switch, otherwise overhead is too high (all overhead)

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:



- Waiting time for

$$P_1 = (68 - 20) + (112 - 88) = 72$$

$$P_2 = (20 - 0) = 20$$

$$P_3 = (28 - 0) + (88 - 48) + (125 - 108) = 85$$

$$P_4 = (48 - 0) + (108 - 68) = 88$$

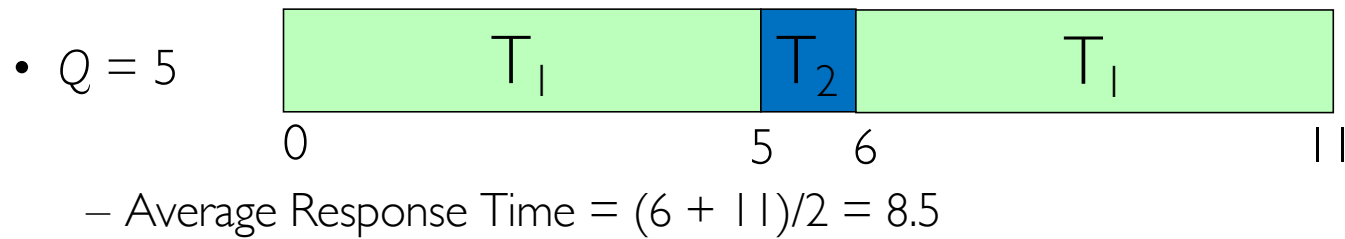
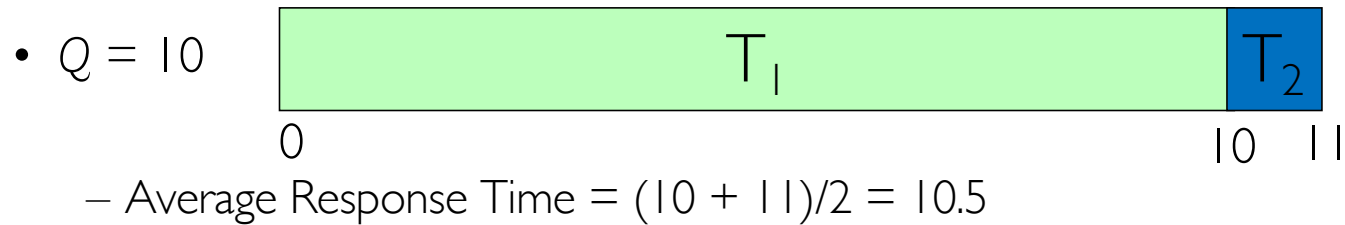
– Average waiting time = $(72 + 20 + 85 + 88) / 4 = 66\frac{1}{4}$

– Average completion time = $(125 + 28 + 153 + 112) / 4 = 104\frac{1}{2}$

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

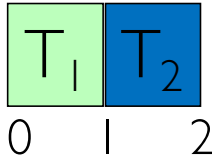
Decrease Response Time

- T_1 : Burst Length 10
- T_2 : Burst Length 1

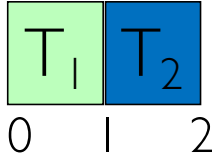


Same Response Time

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

- $Q = 10$ 

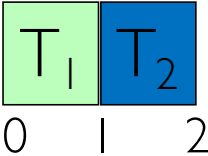
– Average Response Time = $(1 + 2)/2 = 1.5$

- $Q = 1$ 

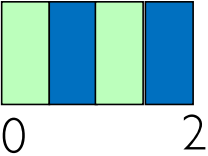
– Average Response Time = $(1 + 2)/2 = 1.5$

Increase Response Time

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

- $Q = 1$ 

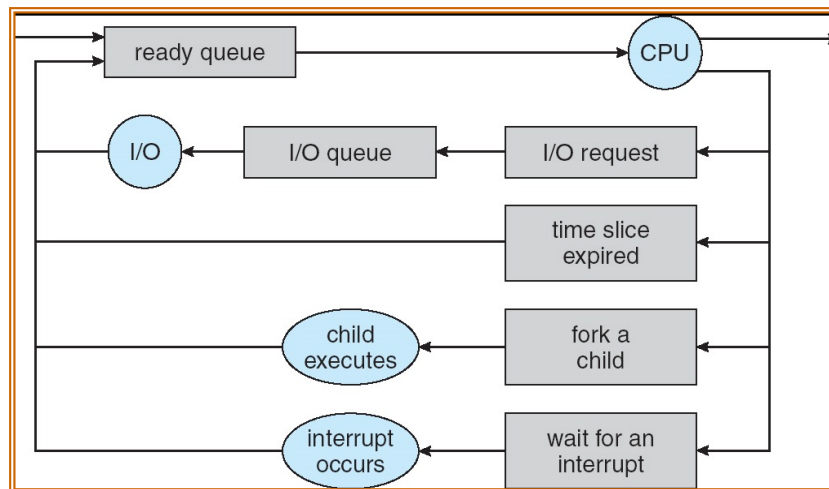
– Average Response Time = $(1 + 2)/2 = 1.5$

- $Q = 0.5$ 

– Average Response Time = $(1.5 + 2)/2 = 1.75$

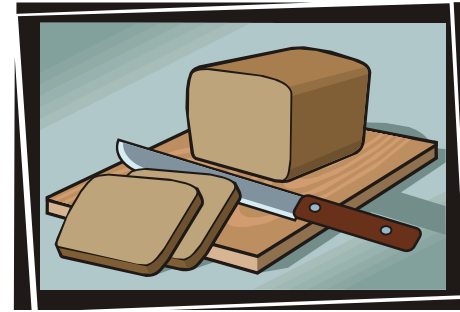
How to Implement RR in the Kernel?

- 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?
 - » Response time suffers
 - What if infinite (∞)?
 - » Get back FIFO
 - What if time slice too small?
 - » Throughput suffers!
- Actual choices of timeslice:
 - Initially, UNIX timeslice 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
 - » Typical context-switching overhead is 0.1ms – 1ms
 - » Roughly 1% overhead due to context-switching



Comparisons between FCFS and Round Robin

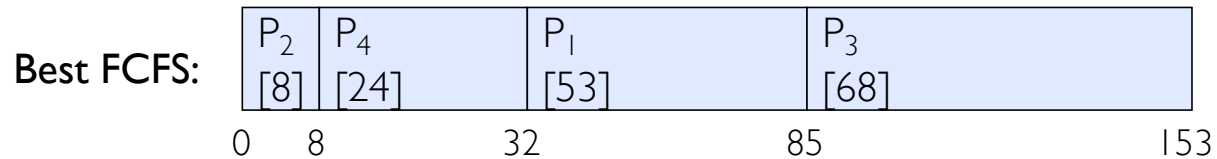
- Assuming zero-cost context-switching time, is RR always better than FCFS?
- Simple example: 10 jobs, each take 100s of CPU time
RR scheduler quantum of 1s
All jobs start at the same time

- Completion Times:

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

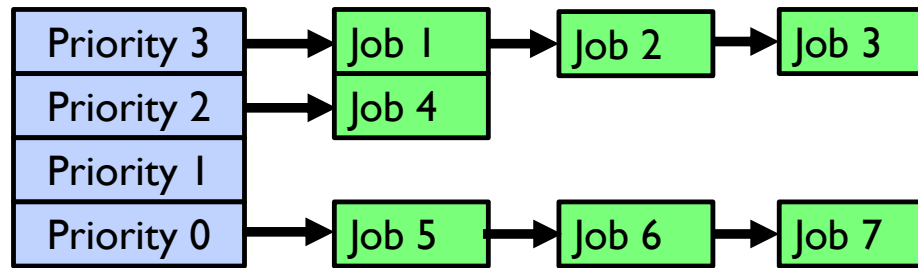
- Both RR and FCFS finish at the same time
 - Average response time is much worse under RR!
 - » Bad when all jobs same length
- Also: Cache state must be shared between all jobs with RR but can be devoted to each job with FIFO
 - Total time for RR longer even for zero-cost switch!

Earlier Example with Different Time Quantum



	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



- Execution Plan
 - Always execute highest-priority runnable 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 response 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⇒Interactive jobs suffer

What if we Knew the Future?

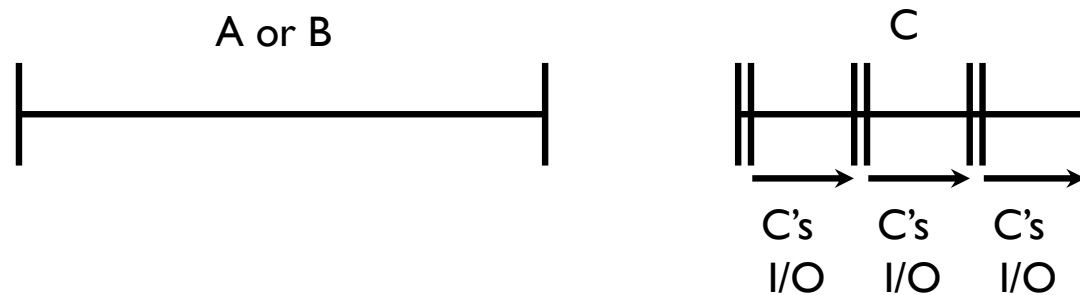
- Could we always mirror best FCFS?
- Shortest Job First (SJF):
 - Run whatever job has least amount of computation to do
 - Sometimes called “Shortest Time to Completion First” (STCF)
- Shortest Remaining Time First (SRTF):
 - Preemptive version of SJF: if job arrives and has a shorter time to completion 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 response time



Discussion

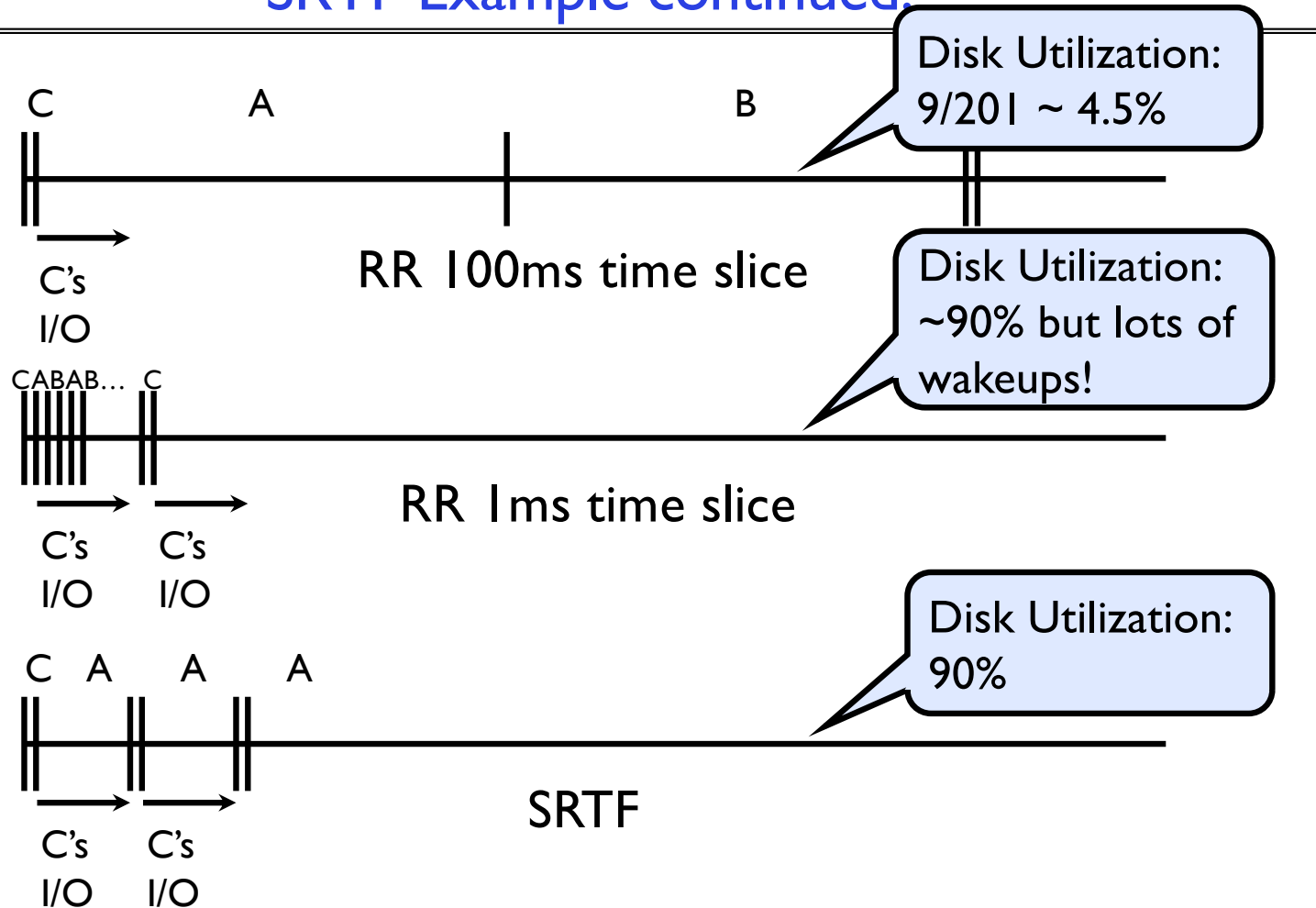
- SJF/SRTF are the best you can do at minimizing average response time
 - Provably optimal (SJF among non-preemptive, SRTF among preemptive)
 - Since SRTF is always at least as good as SJF, focus on SRTF
- Comparison of SRTF with FCFS
 - What if all jobs the same length?
 - » SRTF becomes the same as 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



- 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

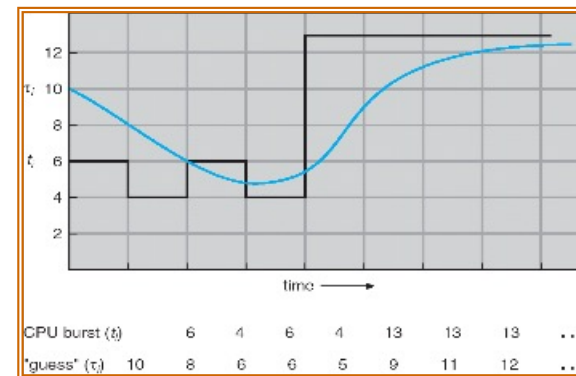
- Starvation
 - SRTF can lead to starvation if many small jobs!
 - Large jobs never get to run
- Somehow need to predict future
 - How can we do this?
 - Some systems ask the user
 - » When you submit a job, have to say how long it will take
 - » To stop cheating, system kills job if takes too long
 - 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 response time) (+)
 - Hard to predict future (-)
 - Unfair (-)



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 $\tau_n = f(t_{n-1}, t_{n-2}, t_{n-3}, \dots)$
 - 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 $(0 < \alpha \leq 1)$



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 (Cont.)

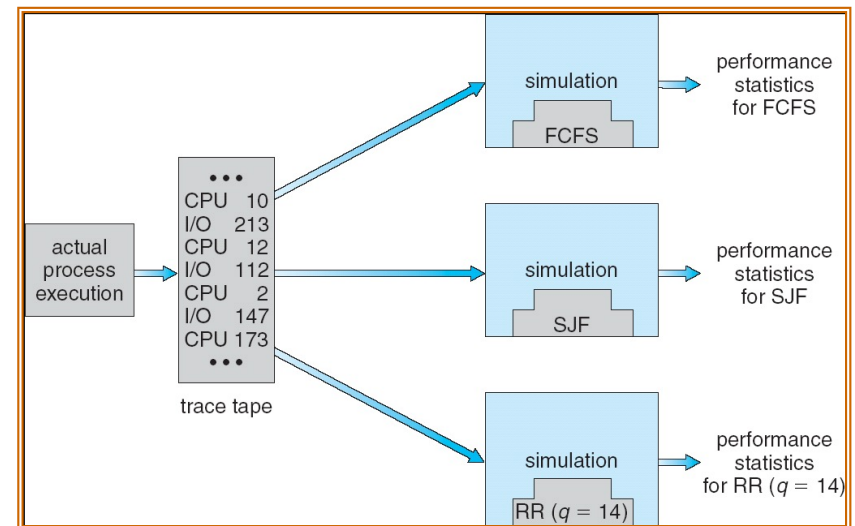
- Lottery Scheduling Example
 - Assume short jobs get 10 tickets, long jobs get 1 ticket

# 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 response time?
 - » If load average is 100, hard to make progress
 - » One approach: log some user out

How to Evaluate a Scheduling algorithm?

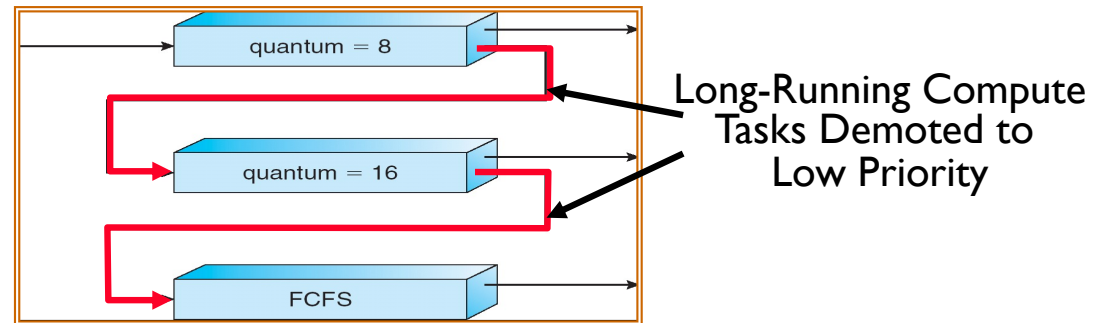
- Deterministic modeling
 - takes a predetermined workload and compute the performance of each algorithm for that workload
- Queueing models
 - Mathematical approach for handling stochastic workloads
- Implementation/Simulation:
 - Build system which allows actual algorithms to be run against actual data
 - Most flexible/general



How to Handle Simultaneous Mix of Diff Types of Apps?

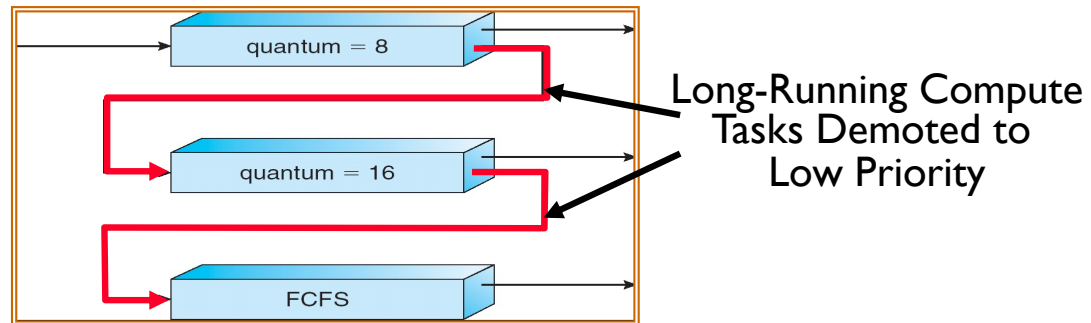
- Consider mix of interactive and high throughput apps:
 - How to best schedule them?
 - How to recognize one from the other?
 - » Do you trust app to say that it is “interactive”?
 - Should you schedule the set of apps identically on servers, workstations, pads, and cellphones?
- For instance, is Burst 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)

Multi-Level Feedback Scheduling



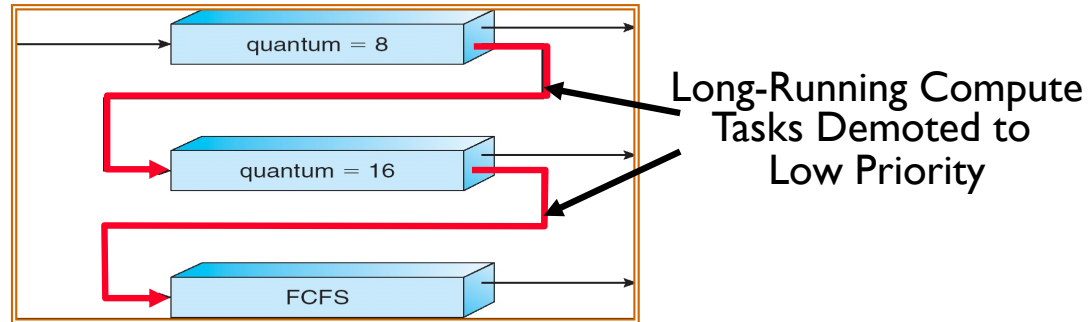
- Another method for exploiting past behavior (first use in CTSS)
 - Multiple queues, each with different priority
 - » Higher priority queues often considered “foreground” tasks
 - Each queue has its own scheduling algorithm
 - » e.g. foreground – RR, background – FCFS
 - » Sometimes multiple RR priorities 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 the competitors.
 - » Put in printf's, ran much faster!

So, Does the OS Schedule Processes or Threads?

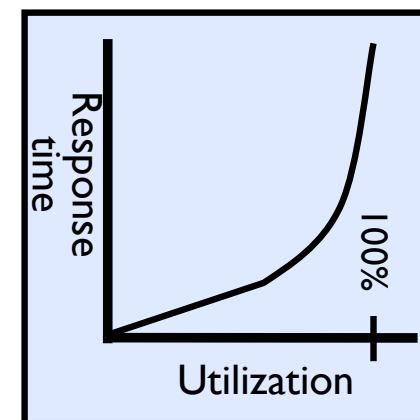
- Many textbooks use the “old model”—one thread per process
- Usually it's really: **threads** (e.g., in Linux)
- One point to notice: switching threads vs. switching processes incurs different costs:
 - Switch threads: Save/restore registers
 - Switch processes: Change active address space too!
 - » Expensive
 - » Disrupts caching

Multi-Core Scheduling

- Algorithmically, not a huge difference from single-core scheduling
- Implementation-wise, helpful to have *per-core* scheduling data structures
 - Cache coherence
- *Affinity scheduling*: once a thread is scheduled on a CPU, OS tries to reschedule it on the same CPU
 - Cache reuse

A Final Word On Scheduling

- When do the details of the scheduling policy and fairness really matter?
 - When there aren't enough resources to go around
- When should you simply buy a faster computer?
 - (Or network link, or expanded highway, or ...)
 - One approach: Buy it when it will pay for itself in improved response time
 - » Perhaps you're paying for worse response time in reduced productivity, customer angst, etc...
 - » Might think that you should buy a faster X when X is utilized 100%, but usually, response time goes to infinity as utilization \Rightarrow 100%
- An interesting implication of this curve:
 - Most scheduling algorithms work fine in the “linear” portion of the load curve, fail otherwise
 - Argues for buying a faster X when hit “knee” of curve



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 response time)
 - Cons: Hard to predict future, Unfair
- **Multi-Level Feedback Scheduling:**
 - Multiple queues of different priorities and scheduling algorithms
 - Automatic promotion/demotion of process priority in order to approximate SJF/SRTF
- **Lottery Scheduling:**
 - Give each thread a priority-dependent number of tokens (short tasks \Rightarrow more tokens)