Scheduling 1: Concepts and Classic Policies

Lecture 9

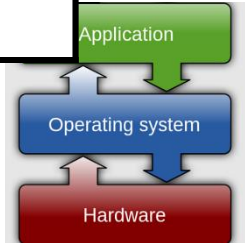
Hartmut Kaiser

https://teaching.hkaiser.org/spring2025/csc4103/

Recall: “Too Much Milk”

- Analogy between problems in OS and problems in real life
- Example: People need to coordinate:

Time	Person A	Person B
3:00	Look in Fridge. Out of milk	
3:05	Leave for store	
3:10	Arrive at store	Look in Fridge. Out of milk
3:15	Buy milk	Leave for store
3:20	Arrive home, put milk away	Arrive at store
3:25		Buy milk
3:30		Arrive home, put milk away



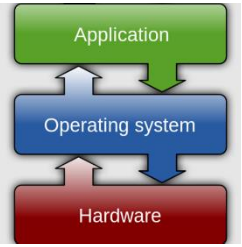
Recall: Single-Core Lock Implementation

```
int value = FREE;

Acquire() {
  disable interrupts;
  if (value == BUSY) {
    put thread on wait queue;
    run_new_thread();
    // Enable interrupts?
  } else {
    value = BUSY;
  }
  enable interrupts;
}
```

```
Release() {
  disable interrupts;
  if (anyone on wait queue) {
    take thread off wait queue;
    Place on ready queue;
  } else {
    value = FREE;
  }
  enable interrupts;
}
```

- Key idea: maintain a lock variable (**`value`**) and disable interrupts only during operations on that variable



Recall: Re-enable Interrupts when Waiting

- In scheduler, since interrupts are disabled when switching threads:
  - Responsibility of the next thread is to re-enable interrupts
  - When the sleeping thread wakes up, returns to acquire and re-enables interrupts

<u>Thread A</u>

```
...
disable ints
call run_new_thread
```

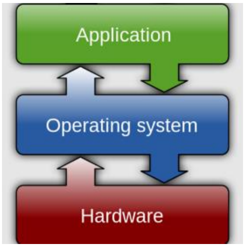
context switch →

<u>Thread B</u>

```
run_new_thread return
enable ints
...
disable int
call run_new_thread
```

← context switch

```
run_new_thread return
enable ints
...
```

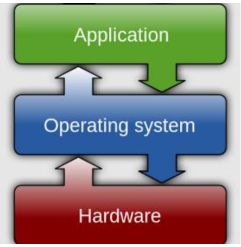


Recall: Spinlock

- Spinlock implementation:

```
int value = 0; // Free
Acquire() {
  while (test&set(value)) {}; // spin while busy
}
Release() {
  value = 0;                  // atomic store
}
```

- Spinlock doesn't put the calling thread to sleep—it just busy waits
  - When might this be preferable?



Recall: Multi-Core Lock Implementation

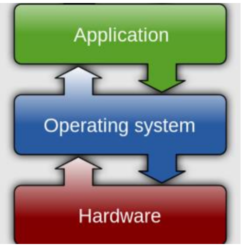
- Can we build test&set locks without busy-waiting?
  - Can't entirely, but can minimize!
  - Idea: only busy-wait to atomically check lock value

```
int guard = 0;
int value = FREE;

Acquire() {
    // Short busy-wait time
    while (test&set(guard)) /**/;
    if (value == BUSY) {
        put thread on wait queue;
        run_new_thread() & guard = 0;
    } else {
        value = BUSY;
        guard = 0;
    }
}
```

```
Release() {
    // Short busy-wait time
    while (test&set(guard)) /**/;
    if anyone on wait queue {
        take thread off wait queue
        Place on ready queue;
    } else {
        value = FREE;
    }
    guard = 0;
}
```

- Note: sleep has to be sure to reset the guard variable
  - Why can't we do it just before or just after the sleep?



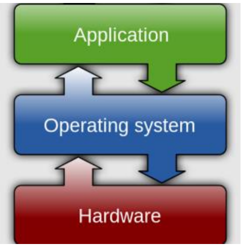
Recall: test&test&set

- A better spinlock solution:

```
int mylock = 0; // Free
Acquire() {
  do {
    while(mylock) /**/;       // Wait until might be free
  } while(test&set(&mylock)); // exit if get lock
}

Release() {
  mylock = 0;
}
```

- Explanation:
  - Wait until lock might be free (only reading – stays in cache)
  - Then, try to grab lock with test&set
  - Repeat if fail to actually get lock
- Busy-Waiting: no longer impacts other processors!

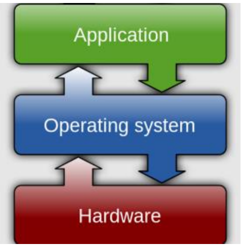


Recall: Linux futex: Fast Userspace Mutex

```
#include <linux/futex.h>
#include <sys/time.h>

int futex(int *uaddr, int futex_op, int val,
      const struct timespec *timeout );
```

- uaddr points to a 32-bit value in user space
- futex_op
  - FUTEX_WAIT – if val == *uaddr sleep till FUTEX_WAIT
    - Atomic check that condition still holds
  - FUTEX_WAKE – wake up at most val waiting threads
  - FUTEX_FD, FUTEX_WAKE_OP, FUTEX_CMP_REQUEUE
- timeout
  - ptr to a timespec structure that specifies a timeout for the op



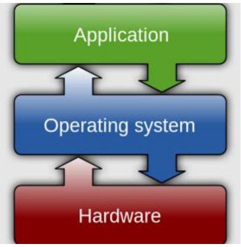
Recall: Userspace Locks with futex

```
int value = 0; // free
bool maybe_waiters = false;

Acquire() {
    while (test&set(value)) {
        maybe_waiters = true;
        futex(&value, FUTEX_WAIT, 1);
        // futex: sleep if lock is acquired
        maybe_waiters = true;
    }
}

Release() {
    value = 0;
    if (maybe_waiters) {
        maybe_waiters = false;
        futex(&value, FUTEX_WAKE, 1);
        // futex: wake up a sleeping thread
    }
}
```

- This is syscall-free in the uncontended case
  - Temporarily falls back to syscalls if multiple waiters, or concurrent acquire/release
- But it can be considerably optimized!
  - See "Futexes are Tricky" by Ulrich Drepper

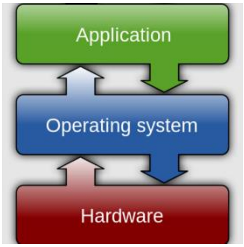


Scheduling

Today: Scheduling

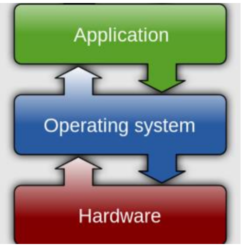
```
if ( readyProcesses(PCBs) ) {
        nextPCB = selectProcess(PCBs);
        run( nextPCB );
} else {
        run_idle_process();
}
```

- Scheduling: Mechanism for deciding which processes/threads receive the CPU
- Lots of different scheduling policies provide …
  - Fairness or
  - Realtime guarantees or
  - Latency optimization or …



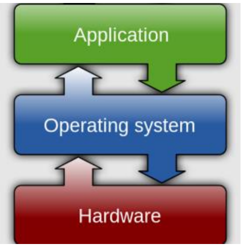
Scheduling Opportunities

- Every “yield”
- Every timer tick (interrupt)
- But also:
  - Every syscall
  - Every interrupt (even if not due to timer)
- Whenever you enter the kernel, for any reason…
- The kernel could switch the running thread at any of these times!

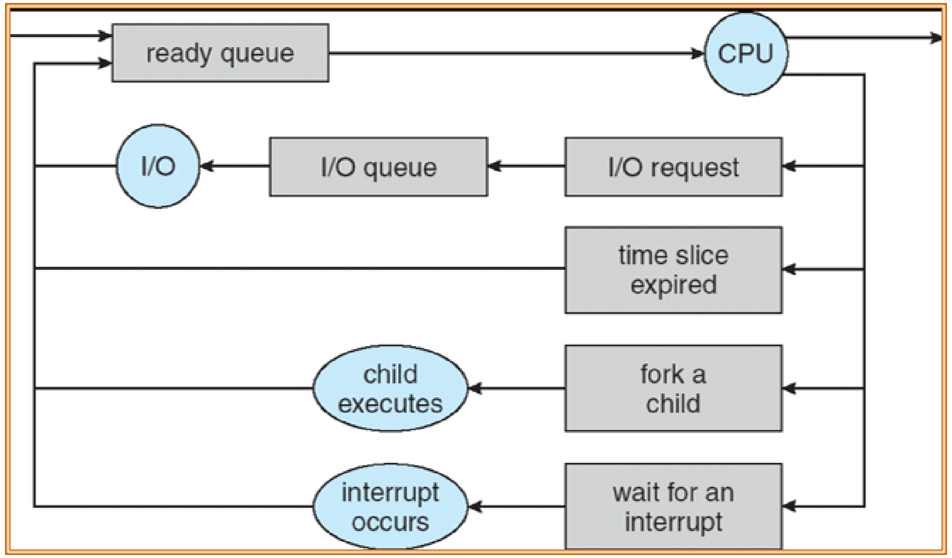


Broader Take on Scheduling

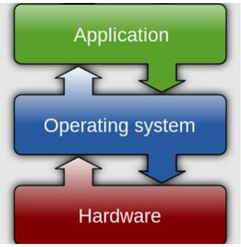
- Scheduling: deciding which threads are given access to resources from moment to moment
  - Often, we think in terms of CPU time, but could also think about access to resources like network BW or disk access



Scheduling: All About Queues

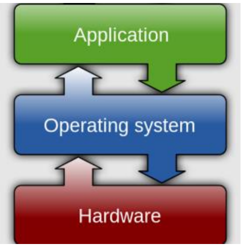


Useful formulation of scheduling: How is the OS to decide which of several tasks to take off a queue?

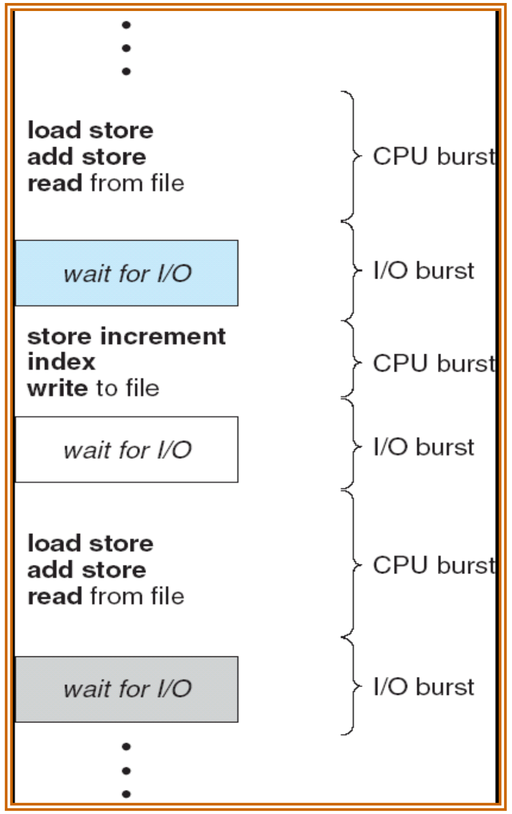


Scheduling: All About Trade-Offs

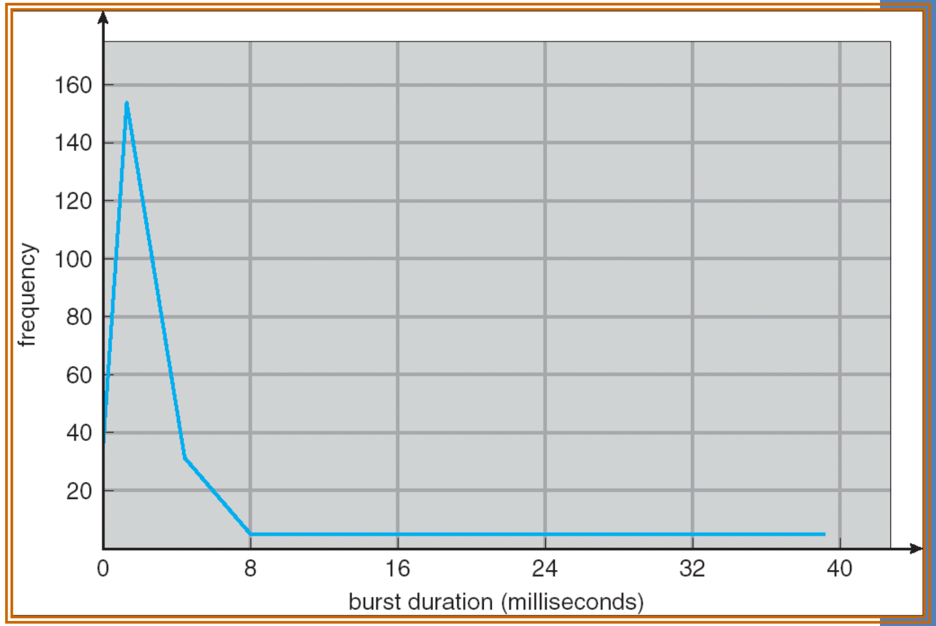
- Individuals care about getting their task done quickly
- System cares about overall efficiency
  - Utilize multiple HW resources well, low overhead, …
- Huge variation in job characteristics
- Fairness???
- What is our utility function?

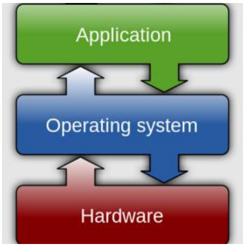


CPU and I/O Bursts



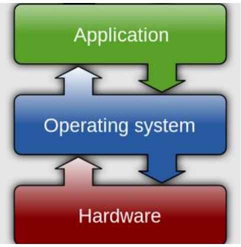
- Programs alternate between bursts of CPU and I/O activity
- Scheduler: Which thread (CPU burst) to run next?
- Interactive programs vs. Compute Bound vs. Streaming





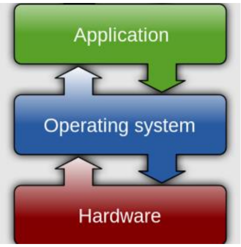
Evaluating Schedulers

- Response Time (ideally low)
  - What user sees: from keypress to character on screen
  - Or completion time for non-interactive
- Throughput (ideally high)
  - Total operations (jobs) per second
  - Overhead (e.g. context switching), artificial blocks
- Fairness
  - Fraction of resources provided to each
  - May conflict with best average throughput or response time



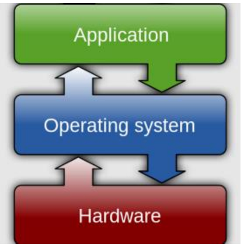
Discussion: Scheduling Assumptions

- Equal or variable job length?
- Run to completion vs. preemption?
- Arrival time (at once vs varied)?
- Resources: CPU(s), I/O, Network, …?
- Advanced Knowledge of Job characteristics or needs
  - Off-line scheduling is given the entire collection of tasks and computes a schedule
  - On-line scheduling makes decisions as tasks arrive



Scheduling Assumptions

- Many implicit assumptions needed to make the problem solvable
- 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

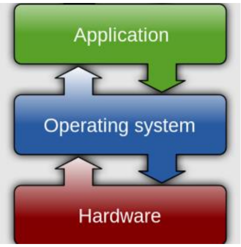


First-Come, First-Served Scheduling (FCFS)

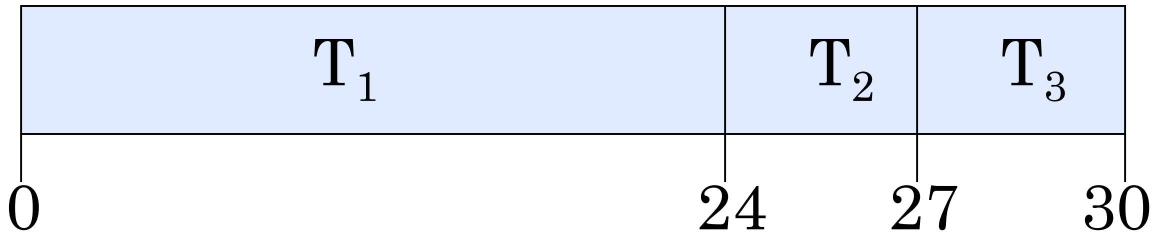
- Also: “First In First Out” (FIFO)
- Example:

Process	Burst Time
T1	24
T2	3
T3	3

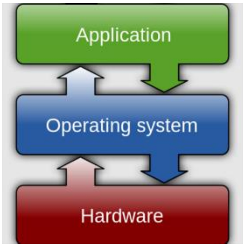
- Arrival Order: T1, T2, then T3 (all arrive at time 0)



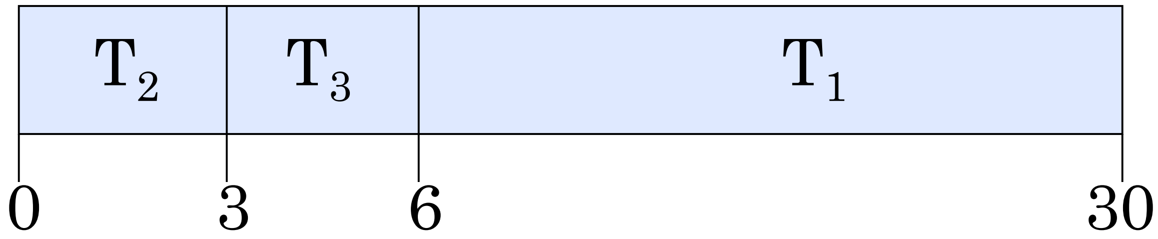
First-Come, First-Served Scheduling (FCFS)



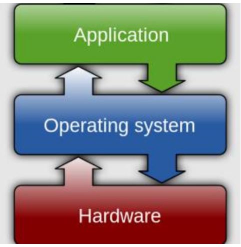
- Response Times: T1 = 24, T2 = 27, T3 = 30
  - Average Response Time = (24+27+30) / 3 = 27
- Waiting times: T1 = 0, T2 = 24, T3 = 27
  - Average Wait Time = (0 + 24 + 27) / 3 = 17
- Convoy Effect: Short processes stuck behind long processes
  - If T2, T3 arrive any time < 24, they must wait



Slightly Different Arrival Order?

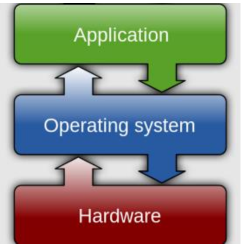


- T2 < T3 < T1
- Response Time: T1 = 30, T2 = 3, T3 = 6
  - Average Response Time = (30 + 3 + 6) / 3 = 13
  - Versus 27 with T1 < T2 < T3
- Waiting Time: T1 = 6, T2 = 0, T3 = 3
  - Average Waiting Time = (6+0+3) / 3 = 3
  - Versus 17 with T1 < T2 < T3

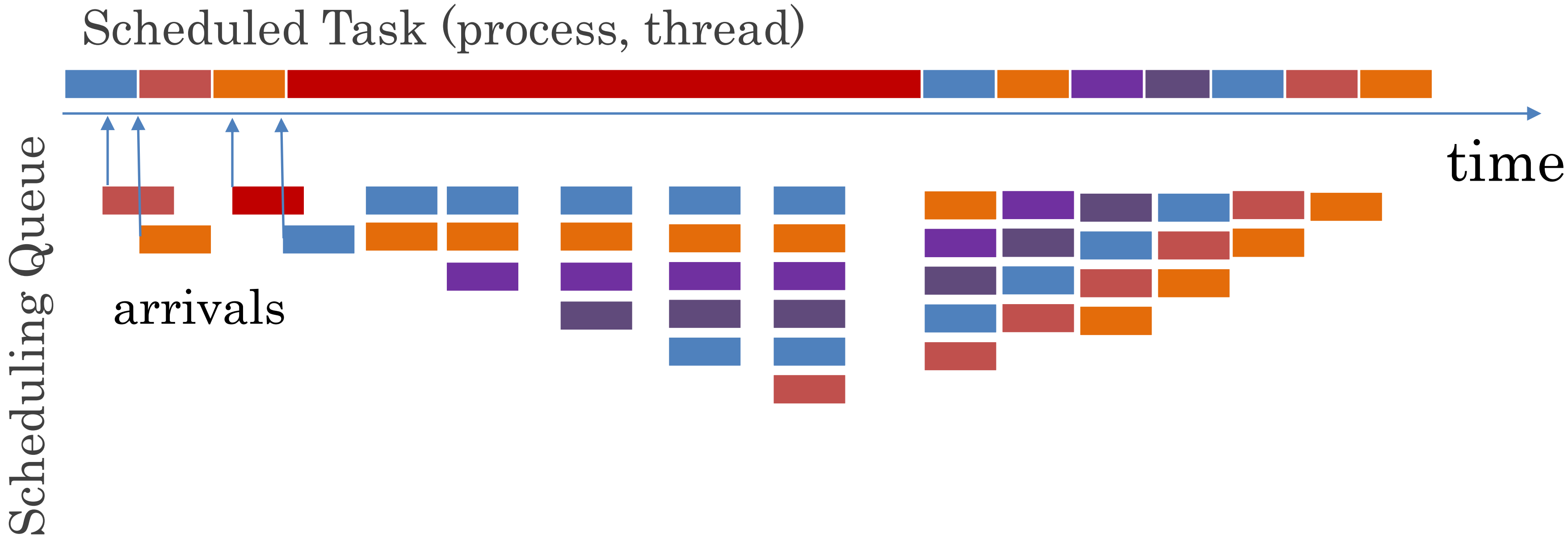


How to Implement FCFS in the Kernel?

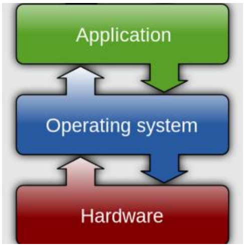
- Comes down to scheduling queue data structure
  - FIFO
  - E.g., push_front, pop_back



Convoy Effect



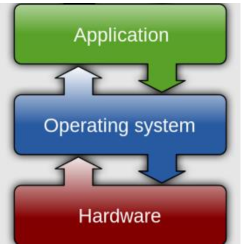
- With FCFS non-preemptive scheduling, convoys of small tasks tend to build up when a large one is running.



First-Come, First-Serve Scheduling

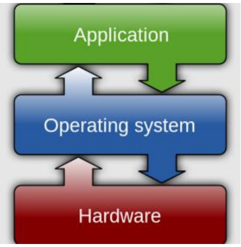
- FCFS Scheme: Potentially bad for short jobs!
  - Depends on submit order
  - If you are first in line at supermarket with milk only, you don't care who is behind you, on the other hand…

- Idea: What if we preempted long-running jobs to give shorter jobs a chance to run?



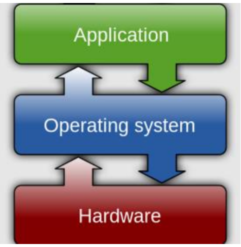
Announcements

- Project 1 due next Monday, March 24
  - Let me know if you need an extension
- Assignment 2 due Monday, April 7
- Midterm 1 is still being graded – apologies!
  - Expect grades to be available by the weekend

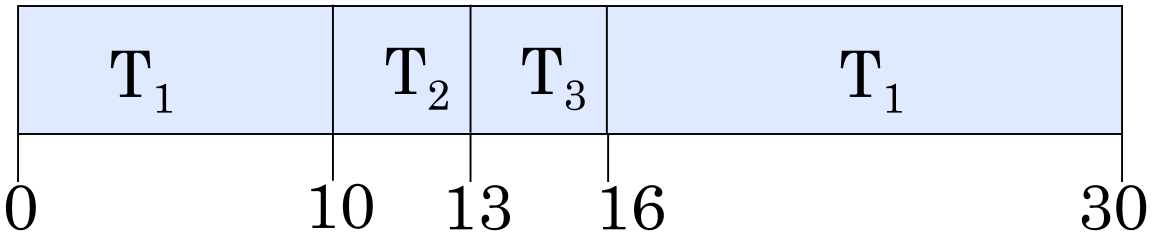


Round-Robin Scheduling (RR)

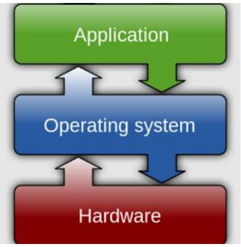
- Give out small units of CPU time ("time quantum")
  - Typically 10 – 100 milliseconds
- When quantum expires, preempt, and schedule
  - Round Robin: add to end of the queue
- Each of N processes gets ~1/N of CPU (in window)
  - With quantum length Q ms, process waits at most (N-1)*Q ms to run again
- Downside: More context switches



Example From Earlier ($Q = 10$)



- Regardless of arrival order, short jobs gets a chance early
- Much less sensitive to arrival order
- How much context switch overhead?



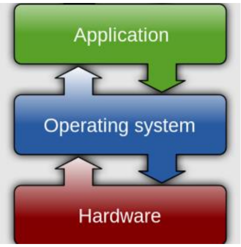
Another Example ($Q = 20$)

Task	Burst Time
T1	53
T2	8
T3	68
T4	24

T_1	T_2	T_3	T_4	T_1	T_3	T_4	T_1	T_3	T_3

0 20 28 48 68 88 108 112 125 145 153

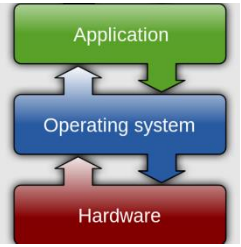
- Avg. response time = (125+28+153+112) / 4 = 104.5
- Waiting times:
  - T1 = (68-20)+(112-88) = 72
  - T2 = (20-0) = 20
  - T3 = (28-0)+(88-48)+(125-108) = 85
  - T4 = (48-0)+(108-68) = 88
- Average waiting time = (72+20+85+88) / 4 = 66.25
- And don't forget context switch overhead!



Round-Robin Quantum

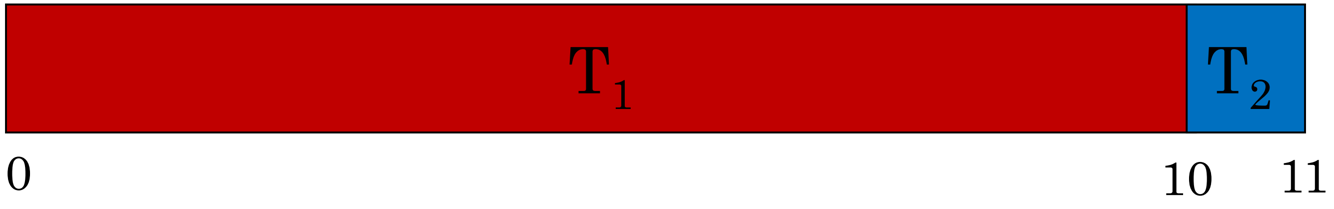
- Assume that context switch overhead is 0
- What happens when we decrease Q?

- Avg. response time always decreases or stays the same?
- Avg. response time always increases or stays the same?
- Avg. response time can increase, decrease, or stays the same?

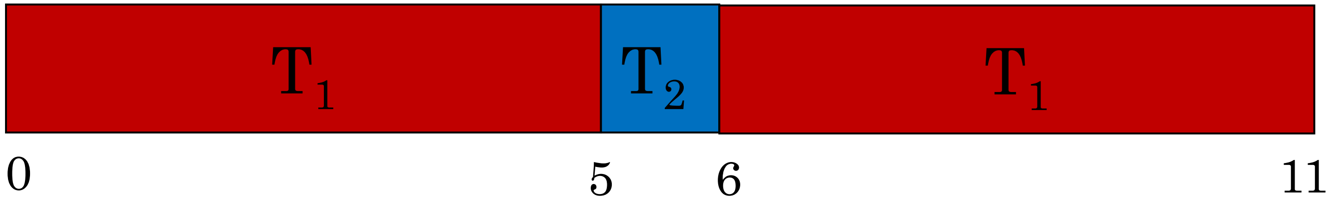


Decrease Response Time

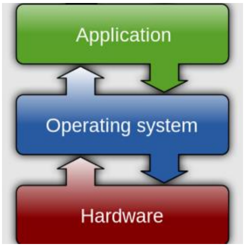
- T1: Burst Length 10
- T2: Burst Length 1
- Q = 10



- Average Response Time = (10 + 11)/2 = 10.5
- Q = 5

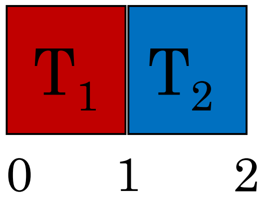


- Average Response Time = (6 + 11)/2 = 8.5

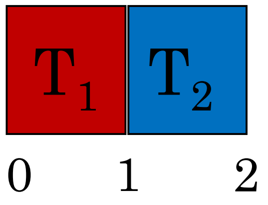


Same Response Time

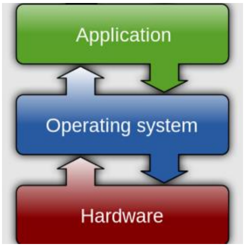
- T1: Burst Length 1
- T2: Burst Length 1
- Q = 10



- Average Response Time = (1 + 2)/2 = 1.5
- Q = 1

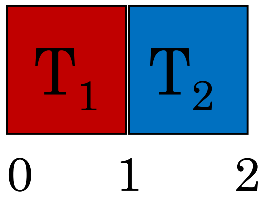


- Average Response Time = (1 + 2)/2 = 1.5

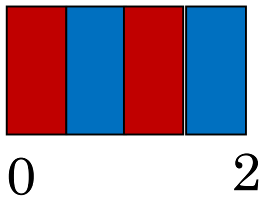


Increase Response Time

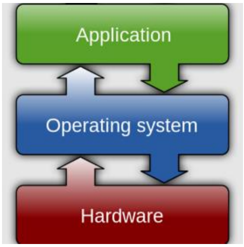
- T1: Burst Length 1
- T2: Burst Length 1
- Q = 1



- Average Response Time = (1 + 2)/2 = 1.5
- Q = 0.5



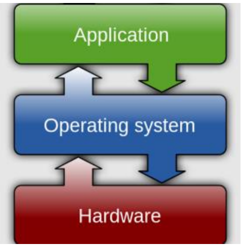
- Average Response Time = (1.5 + 2)/2 = 1.75



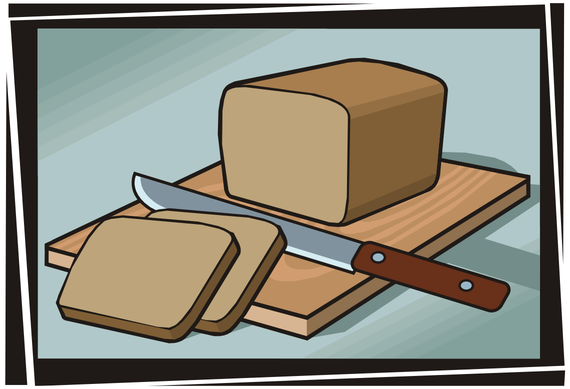
How to Implement RR in the Kernel?

- Round Robin – what is it?
- 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

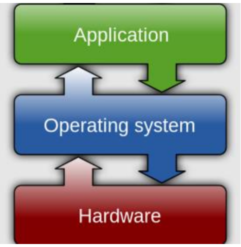
Project 2: Scheduling



Discussion: Round-Robin Scheduling

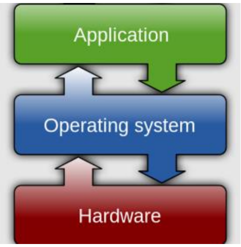


- How to choose the time quantum?
  - Too big? RR approaches FCFS
  - Too small? Throughput suffers (due to context switches)
- Actual choices of timeslice:
  - Initially, in UNIX timeslice was one second:
    - Worked ok when UNIX was used by one or two people.
    - When might this perform poorly?
  - 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

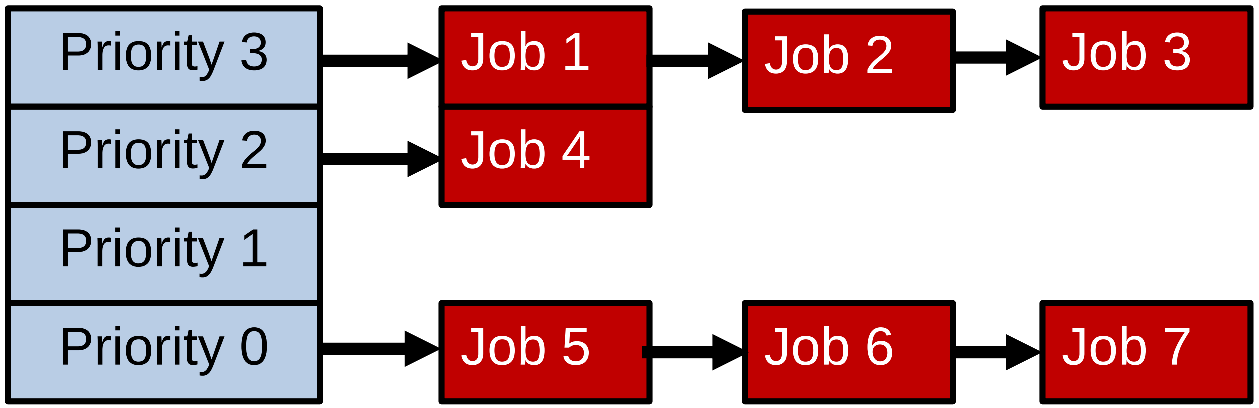


Priority

- Interactive vs. compute bound



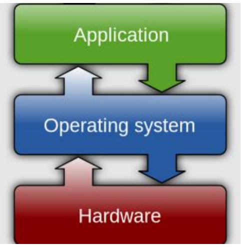
Priority Scheduler



- Something gives jobs (processes) priority
  - Usually the user sets it explicitly, perhaps based on $ rate
- Always run the ready thread with highest priority
  - Low priority thread might never run!
  - Starvation

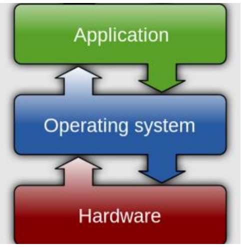
How to Implement Priority Scheduling in the Kernel?

- Scheduling queue data structure determines next thread of those in the ready queue(s)
  - Kernel prefers threads with more urgent priority
- Why might a thread not be in the ready queue?
  - Waiting on I/O
  - Locks?

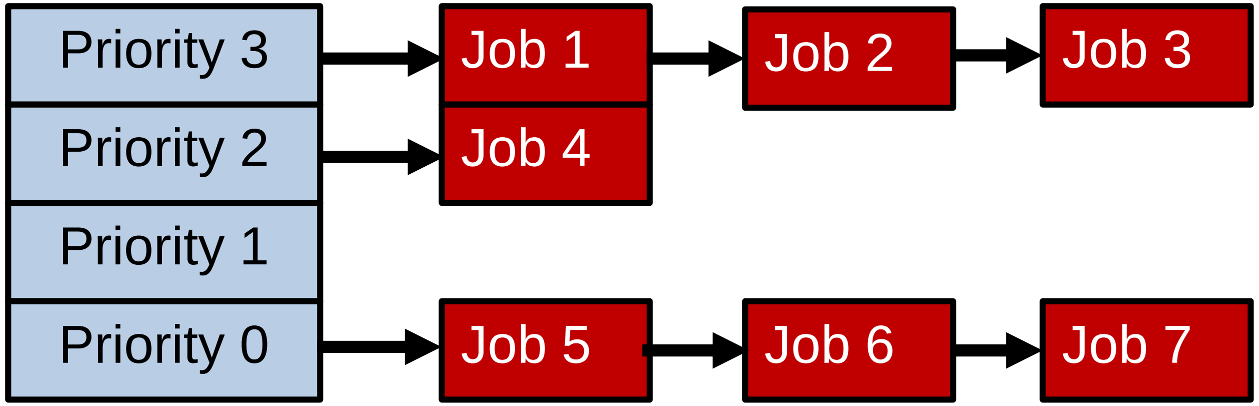


Adaptive Scheduling

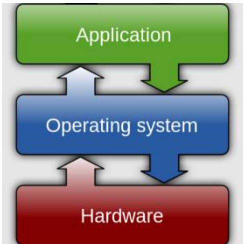
- Modern schedulers use knowledge about program to make better scheduling decisions
- Provided by the user (servers vs. background)
- Estimate future based on the past



Policy Based on Priority Scheduling

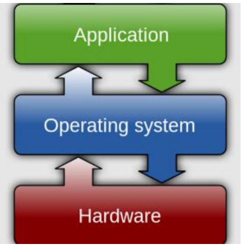


- Systems may try to set priorities according to some policy goal
- Example: Give interactive higher priority than long calculation
  - Prefer jobs waiting on I/O to those consuming lots of CPU
- Try to achieve fairness: elevate priority of threads that don’t get CPU time (ad-hoc, bad if system overload)



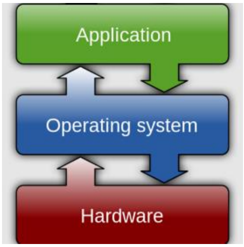
Adaptive Scheduling

- How can we adapt the scheduling algorithm based on threads' past behavior?
- Two steps:
  - Based on past observations, predict what threads will do in the future.
  - Make scheduling decisions based on those predictions.
- Start with the second step. Suppose we knew the workload in advance. What should the scheduler do?



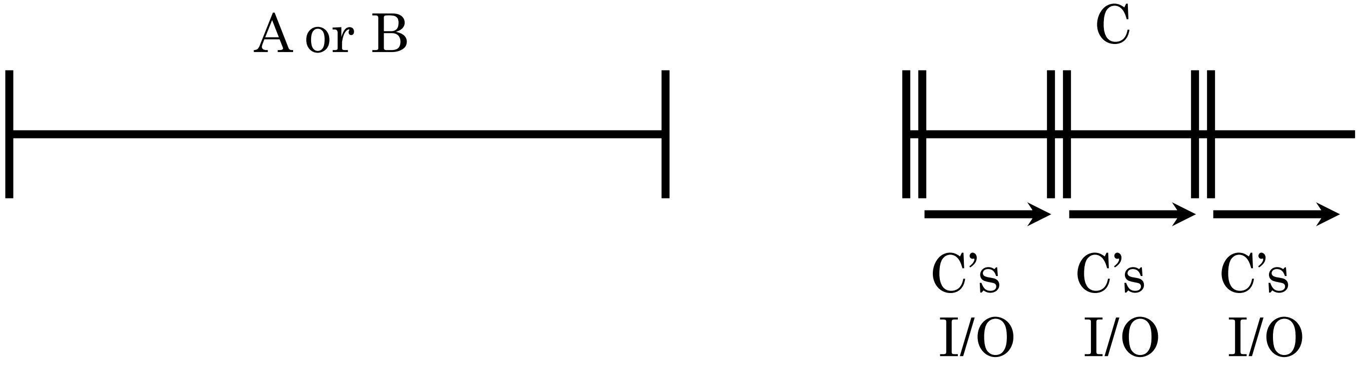
What if we knew how long each CPU burst will be, in advance?

- Key Idea: remove convoy effect
  - Short jobs always stay ahead of long ones
- Non-preemptive: Shortest Job First
  - Like FCFS if we always chose the best possible ordering
- Preemptive Version: Shortest Remaining Time First
  - If a job arrives and has shorter time to completion than current job, immediately preempt CPU
  - Sometimes called "Shortest Remaining Time to Completion First"

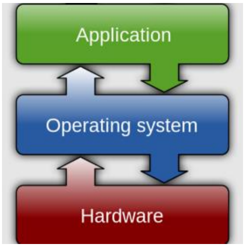


SRTF Example (Shortest Remaining Time First)

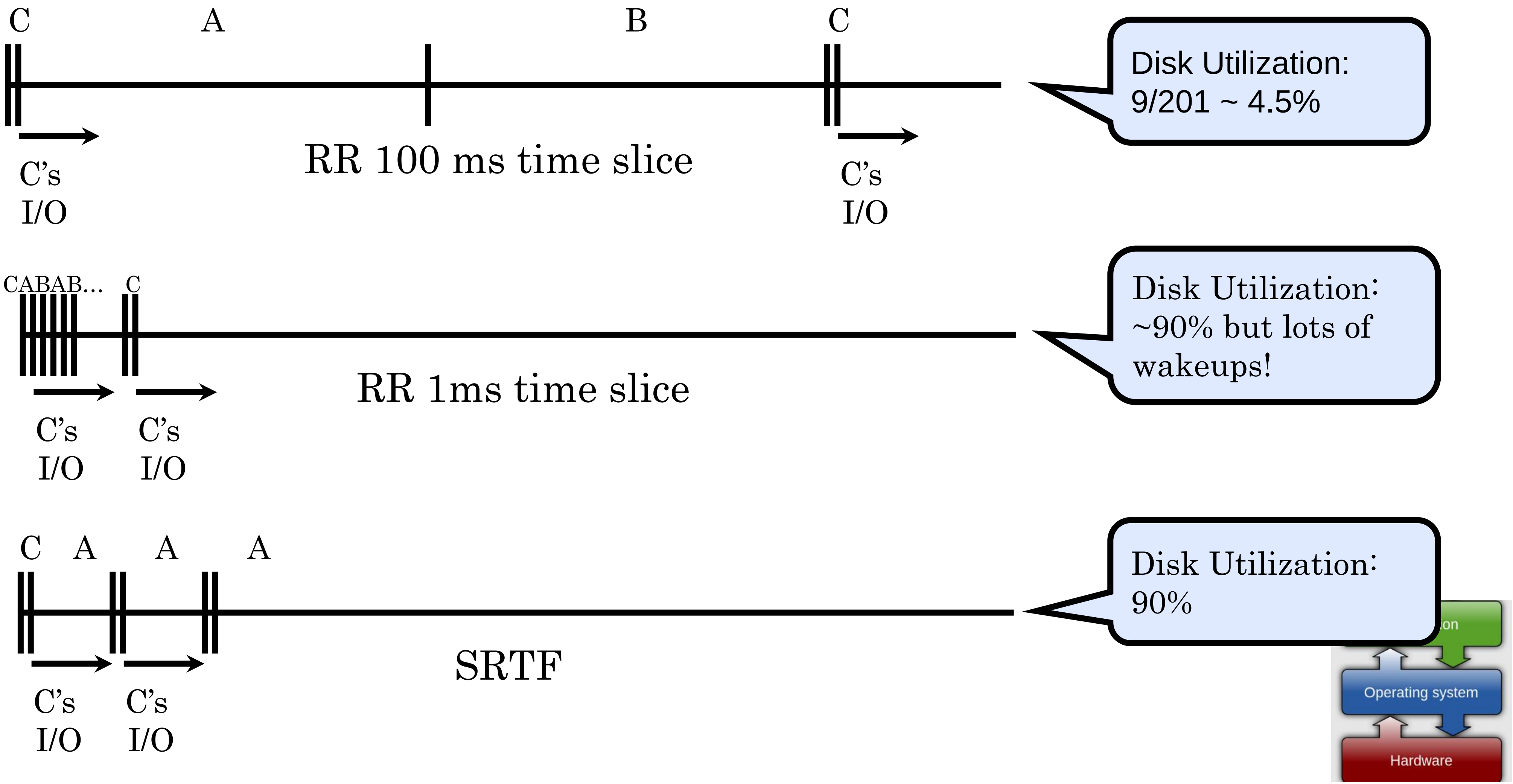
- Three jobs in system
  - A and B are CPU calculations that take a week to run
  - C: Continuous loop of 1ms CPU time, 9ms of I/O time



- FCFS? A or B starve C
  - I/O throughput problem: lose opportunity to do work for C while CPU runs A or B

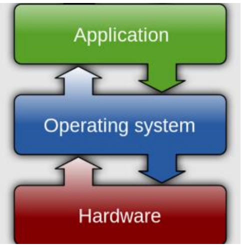


SRTF Example
(Shortest Remaining Time First)



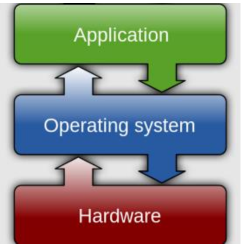
Discussion: SJF and SRTF

- Provably Optimal with respect to Response Time
- But Starvation is possible
  - What if new short jobs keep arriving?
- But: Need to predict the future!
  - Ask the user when they submit the job? How to prevent cheating?
  - SRTF useful as a benchmark to measure other policies?



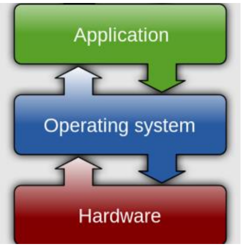
Adaptive Scheduling

- How can we adapt the scheduling algorithm based on threads' past behavior?
- Two steps:
  - Based on past observations, predict what threads will do in the future.
  - Make scheduling decisions based on those predictions.
- Now, let's look at the first step. How can we predict future behavior from past behavior?

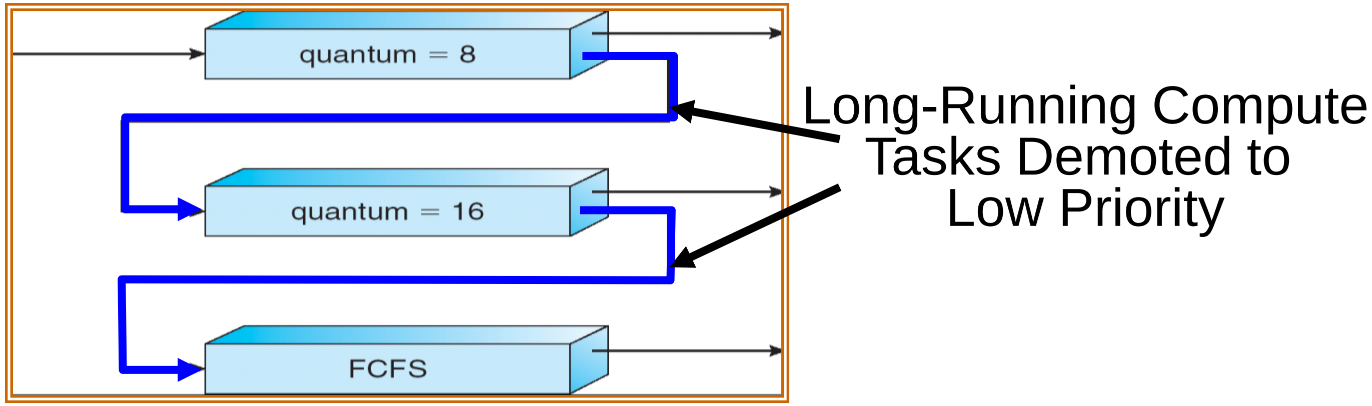


Predicting Future Behavior

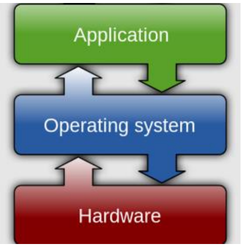
- Consider Round-Robin Scheduling
- If process exhausts quantum, it has to be preempted
  - Consuming all of the CPU time it can: “CPU-Bound”
  - Likely to remain CPU-Bound
- If process blocks on I/O before quantum exhausted
  - Short CPU bursts, just to initiate I/O: “I/O-Bound”
  - Often interactive tasks
  - Likely to remain I/O-Bound and/or Interactive



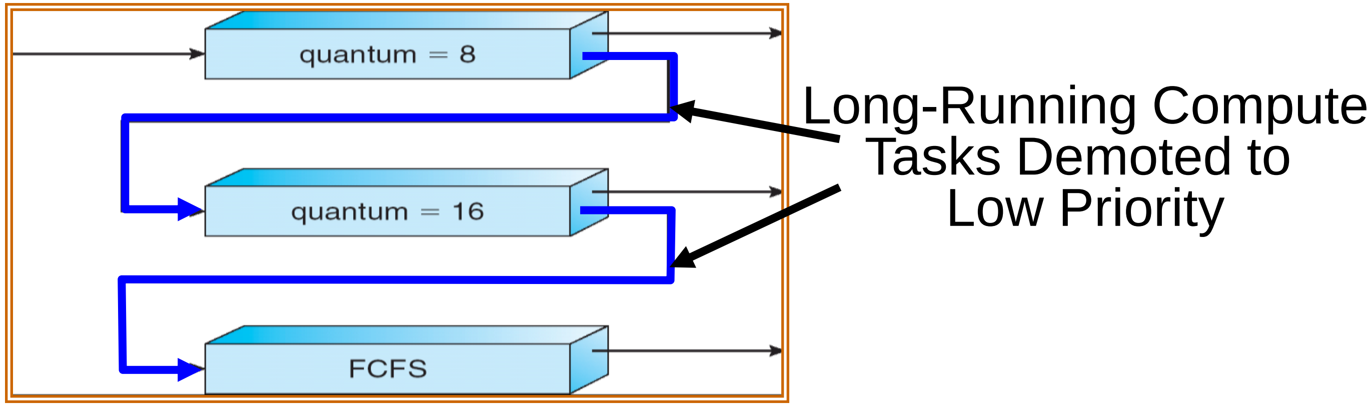
Multi-Level Feedback Queue (MLFQ)



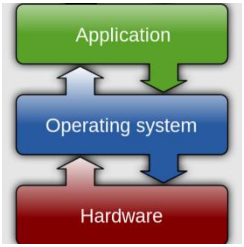
- Multiple queues, each of different priority
  - Round Robin within each queue
  - Different quantum length for each queue
- Favor I/O-bound jobs for interactivity
  - Get click or kick off I/O transfer
- Low overhead for CPU bound



Multi-Level Feedback Queue (MLFQ)

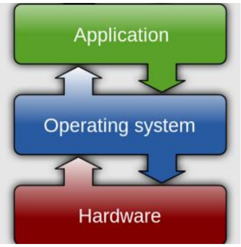


- Intuition: approximate SRTF by setting priority level proportional to burst length
- Job Exceeds Quantum: Drop to lower queue
- Job Doesn't Exceed Quantum: Raise to higher queue



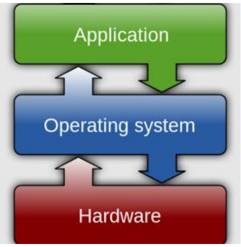
Multi-Level Feedback Queue

- Approximates Shortest Remaining Time First
  - CPU-bound have lowest priority (run last)
  - I/O-bound (short CPU bursts) have highest priority (run first)
- Low overhead
  - Easy to update priority of a job
  - Easy to find next ready task to run
- Can a process cheat?
  - Yes, add meaningless I/O operations (but has a cost)

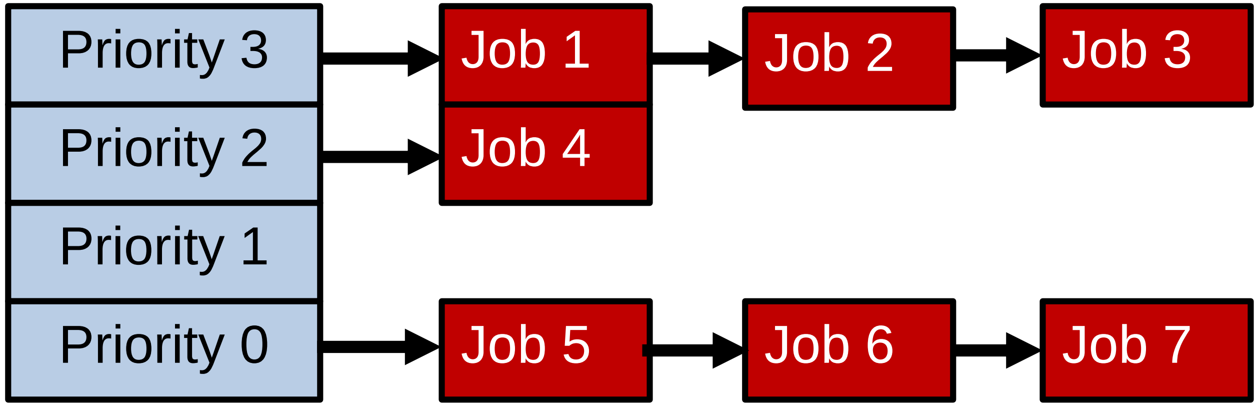


How to Implement MLFQ in the Kernel?

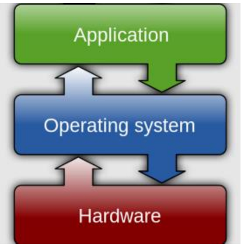
- We could explicitly build the queue data structures
- Or, we can leverage priority-based scheduling!



Recall: Policy Based on Priority Scheduling

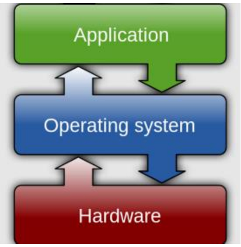


- Systems may try to set priorities according to some policy goal
- Example: Give interactive higher priority than long calculation
  - Prefer jobs waiting on I/O to those consuming lots of CPU
- Try to achieve fairness: elevate priority of threads that don't get CPU time (ad-hoc, bad if system overload)

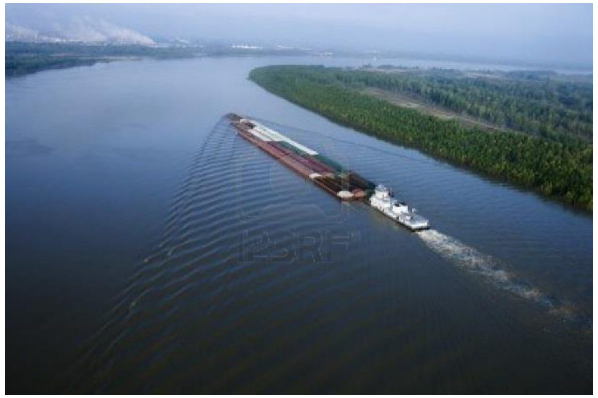


Conclusion

- First-Come First-Served: Simple, vulnerable to convoy effect
- Round-Robin: Fixed CPU time quantum, cycle between ready threads
- Priority: Respect differences in importance
- Shortest Job/Remaining Time First: Optimal for average response time, but unrealistic
- Multi-Level Feedback Queue: Use past behavior to approximate SRTF and mitigate overhead



LSU CENTER FOR COMPUTATION & TECHNOLOGY



STE||AR GROUP