Intro to Scheduling (+ OS sync wrap)

David E. Culler
CS162 – Operating Systems and Systems Programming
Lecture 10
Sept 17, 2014

https://computing.llnl.gov/tutorials/pthreads/

Reading: A&D 7-7.1
HW 2 due wed
Proj 1 design review

Objectives

- Introduce the concept of scheduling
- General topic that applies in many context
  – rich theory and practice
- Fundamental trade-offs
  – not a simple find the "best"
  – resolution depends on context
- Ground it in OS context
- Ground implementation in Pintos
- … after synch implementation wrap-up

Recall: A Lock

- Value: FREE (0) or BUSY (1)
- A queue of waiters (threads*)
  - attempting to acquire
- An owner (thread)

semaphore has these
- value is int

- Acquire: wait till Free, take ownership, make busy
- Release: relinquish ownership, make Free, if waiter allow it to complete acquire
- Both are *atomic relative to other threads*

Recall: the "else" question ???

Locks

```
lock.Acquire();
 …
 critical section;
 …
lock.Release();
```

```
Acquire() {
  disable interrupts;
}
```

```
Release() {
  enable interrupts;
}
```

```
int value = 0;
Acquire() {
  disable interrupts;
  if (value == 1) {
    put thread on wait-queue;
    go to sleep() //??
  } else {
    value = 1;
    enable interrupts;
  }
}
```

Don't we need to do this regardless?

```
Release() {
 disable interrupts;
  if anyone on wait queue {
    take thread off wait-queue
    Place on ready queue;
  } else {
    value = 0;
  }
  enable interrupts;
}
```

If one thread in critical section, no other activity (including OS) can run!

Locks

READY → Thread B

FREE | waiters | owner

Running

Thread A

```
lock.Acquire();
 …
 critical section;
 …
lock.Release();
```

```
INIT
   int value = 0;

Acquire() {
  disable interrupts;
  if (value == 1) {
    put thread on wait-queue;
    go to sleep() //??
  } else {
    value = 1;
    enable interrupts;
  }
}

Release() {
 disable interrupts;
  if anyone on wait queue {
    take thread off wait-queue
    Place on ready queue;
  } else {
    value = 0;
  }
  enable interrupts;
}
```

```
lock.Acquire();
 …
 critical section;
 …
lock.Release();
```

Locks

READY

BUSY | waiters | owner

Running

Thread A

Thread B

```
INIT
   int value = 0;

Acquire() {
  disable interrupts;
  if (value == 1) {
    put thread on wait-queue;
    go to sleep() //??
  } else {
    value = 1;
    enable interrupts;
  }
}

Release() {
 disable interrupts;
  if anyone on wait queue {
    take thread off wait-queue
    Place on ready queue;
  } else {
    value = 0;
  }
  enable interrupts;
}
```

Thread A:

```
lock.Acquire();
 …
 critical section;
 …
lock.Release();
```

Thread B:

```
lock.Acquire();
 …
 critical section;
 …
lock.Release();
```

Locks

READY

BUSY | waiters | owner

Running

Thread A

Thread B

lock.Acquire();
…
critical section;
…
lock.Release();

```
INIT
   int value = 0;

Acquire() {
 disable interrupts;
  if (value == 1) {
    put thread on wait-queue;
    go to sleep() //??
  } else {
    value = 1;
    enable interrupts;
  }
}

Release() {
 disable interrupts;
  if anyone on wait queue {
    take thread off wait-queue
    Place on ready queue;
  } else {
    value = 0;
  }
  enable interrupts;
}
```

lock.Acquire();
…
critical section;
…
lock.Release();

Locks

READY

BUSY | waiters | owner

Running

Thread B

Thread A

```
INIT
   int value = 0;

Acquire() {
  disable interrupts;
  if (value == 1) {
    put thread on wait-queue;
    go to sleep() //??
  } else {
    value = 1;
    enable interrupts;
  }
}

Release() {
 disable interrupts;
  if anyone on wait queue {
    take thread off wait-queue
    Place on ready queue;
  } else {
    value = 0;
  }
  enable interrupts;
}
```

Thread A:

```
lock.Acquire();
 …
 critical section;
 …
lock.Release();
```

Thread B:

```
lock.Acquire();
 …
 critical section;
 …
lock.Release();
```

Locks

READY

BUSY | waiters | owner

Running

Thread B

Thread A

INIT

```
int value = 0;
```

Thread A:

```
lock.Acquire();
 …
 critical section;
 …
lock.Release();
```

```
Acquire() {
 disable interrupts;
  if (value == 1) {
    put thread on wait-queue;
    go to sleep() //??
  } else {
    value = 1;
    enable interrupts;
  }
}

Release() {
 disable interrupts;
  if anyone on wait queue {
    take thread off wait-queue
    Place on ready queue;
  } else {
    value = 0;
  }
  enable interrupts;
}
```

Thread B:

```
lock.Acquire();
 …
 critical section;
 …
lock.Release();
```

Locks

READY

BUSY | waiters | owner

Running

Thread B

Thread A

```
INIT
    int value = 0;
```

Thread A:

```
lock.Acquire();
 …
 critical section;
 …
lock.Release();
```

```
Acquire() {
  disable interrupts;
  if (value == 1) {
    put thread on wait-queue;
    go to sleep() //??
  } else {
    value = 1;
    enable interrupts;
  }
}

Release() {
 disable interrupts;
  if anyone on wait queue {
    take thread off wait-queue
    Place on ready queue;
  } else {
    value = 0;
  }
  enable interrupts;
}
```

Thread B:

```
lock.Acquire();
 …
 critical section;
 …
lock.Release();
```

Locks

READY

BUSY | waiters | owner

Running

Thread A

Thread A

```
INIT
   int value = 0;
```

```
lock.Acquire();
 …
 critical section;
 …
lock.Release();
```

```
Acquire() {
  disable interrupts;
  if (value == 1) {
    put thread on wait-queue;
    go to sleep() //??
  } else {
    value = 1;
    enable interrupts;
  }
}

Release() {
 disable interrupts;
  if anyone on wait queue {
    take thread off wait-queue
    Place on ready queue;
  } else {
    value = 0;
  }
  enable interrupts;
}
```

```
lock.Acquire();
 …
 critical section;
 …
lock.Release();
```

Locks

READY

BUSY | waiters | owner

Running

Thread A

Thread B

```
INIT
   int value = 0;

Acquire() {
  disable interrupts;
  if (value == 1) {
    put thread on wait-queue;
    go to sleep() //??
  } else {
    value = 1;
    enable interrupts;
  }
}
```

Thread A:

```
lock.Acquire();
 …
 critical section;
 …
lock.Release();
```

Thread B:

```
lock.Acquire();
 …
 critical section;
 …
lock.Release();
```

```
Release() {
 disable interrupts;
  if anyone on wait queue {
    take thread off wait-queue
    Place on ready queue;
  } else {
    value = 0;
  }
  enable interrupts;
}
```

Locks

READY

BUSY | waiters | owner

Running

Thread A

Thread B

```
INIT
   int value = 0;

Acquire() {
  disable interrupts;
  if (value == 1) {
    put thread on wait-queue;
    go to sleep() //??
  } else {
    value = 1;
    enable interrupts;
  }
}

Release() {
 disable interrupts;
  if anyone on wait queue {
    take thread off wait-queue
    Place on ready queue;
  } else {
    value = 0;
  }
  enable interrupts;
}
```

Thread A:

```
lock.Acquire();
 …
 critical section;
 …
lock.Release();
```

Thread B:

```
lock.Acquire();
 …
 critical section;
 …
lock.Release();
```

Locks

READY

BUSY | waiters | owner

Running

Thread A

Thread A

INIT

```
    int value = 0;
```

```
lock.Acquire();
 …
 critical section;
 …
lock.Release();
```

```
Acquire() {
 disable interrupts;
  if (value == 1) {
    put thread on wait-queue;
    go to sleep() //??
  } else {
    value = 1;
    enable interrupts;
  }
}
```

```
lock.Acquire();
 …
 critical section;
 …
lock.Release();
```

```
Release() {
 disable interrupts;
  if anyone on wait queue {
    take thread off wait-queue
    Place on ready queue;
  } else {
    value = 0;
  }
  enable interrupts;
}
```

Locks

READY

BUSY | waiters | owner

Thread A

Running

Thread A

```
INIT
    int value = 0;

Acquire() {
  disable interrupts;
  if (value == 1) {
    put thread on wait-queue;
    go to sleep() //??
  } else {
    value = 1;
    enable interrupts;
  }
}

Release() {
  disable interrupts;
  if anyone on wait queue {
    take thread off wait-queue
    Place on ready queue;
  } else {
    value = 0;
  }
  enable interrupts;
}
```

Thread A:

```
lock.Acquire();
 …
 critical section;
 …
lock.Release();
```

Thread B:

```
lock.Acquire();
 …
 critical section;
 …
lock.Release();
```

Locks

READY

BUSY | waiters | owner

Thread B — Running

Thread A

```
INIT
  int value = 0;

Acquire() {
  disable interrupts;
  if (value == 1) {
    put thread on wait-queue;
    go to sleep() //??
  } else {
    value = 1;
    enable interrupts;
  }
}
```

Thread A:

```
lock.Acquire();
 …
 critical section;
 …
lock.Release();
```

Thread B:

```
lock.Acquire();
 …
 critical section;
 …
lock.Release();
```

```
Release() {
  disable interrupts;
  if anyone on wait queue {
    take thread off wait-queue
    Place on ready queue;
  } else {
    value = 0;
  }
  enable interrupts;
}
```

Locks

READY

BUSY | waiters | owner

Thread B — Running

Thread A

```
INIT
  int value = 0;
```

Thread A:

```
lock.Acquire();
 …
 critical section;
 …
lock.Release();
```

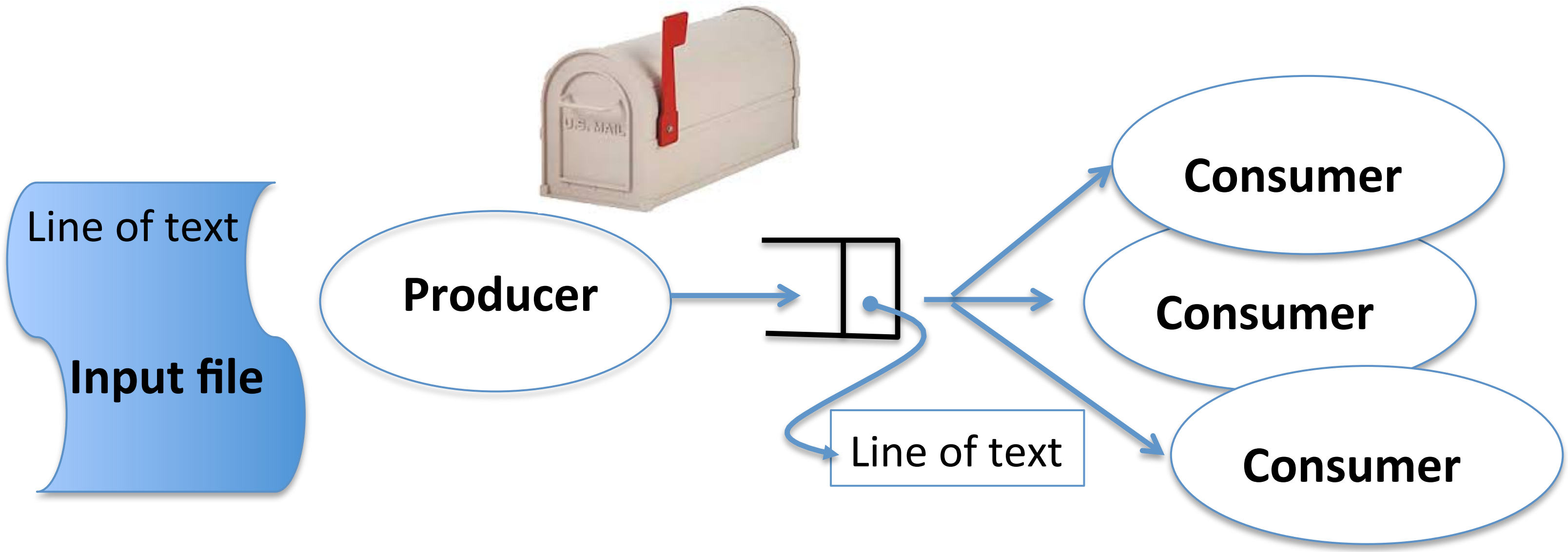
```
Acquire() {
 disable interrupts;
 if (value == 1) {
   put thread on wait-queue;
   go to sleep() //??
 } else {
   value = 1;
   enable interrupts;
 }
}
```

```
Release() {
 disable interrupts;
 if anyone on wait queue {
   take thread off wait-queue
   Place on ready queue;
 } else {
   value = 0;
 }
 enable interrupts;
}
```

Thread B:

```
lock.Acquire();
 …
 critical section;
 …
lock.Release();
```

recall: Multiple Consumers, etc.



- More general relationships require mutual exclusion
  – Each line is consumed exactly once!

Incorporate Mutex into shared object

- Methods on the object provide the synchronization
  - Exactly one consumer will process the line

```
typedef struct sharedobject {
  FILE *rfile;
  pthread_mutex_t solock;
  int flag;
  int linenum;
  char *line;
} so_t;
```

```
int waittill(so_t *so, int val) {
    while (1) {
    pthread_mutex_lock(&so->solock);
    if (so->flag == val)
        return 1; /* rtn with object locked */
    pthread_mutex_unlock(&so->solock);
  }
}
int release(so_t *so) {
  return pthread_mutex_unlock(&so->solock);
}
```

Recall: Multi Consumer

```
void *producer(void *arg) {
  so_t *so = arg;
  int *ret = malloc(sizeof(int));
  FILE *rfile = so->rfile;
  int i;
  int w = 0;
  char *line;
  for (i = 0; (line = readline(rfile)); i++) {
    waittill(so, 0);                /* grab lock when empty */
    so->linenum = i;                /* update the shared state */
    so->line = line;                /* share the line */
    so->flag = 1;                   /* mark full */
    release(so);                    /* release the loc */
    fprintf(stdout, "Prod: [%d] %s", i, line);
  }
  waittill(so, 0);                  /* grab lock when empty */
  so->line = NULL;
  so->flag = 1;
  printf("Prod: %d lines\n", i);
  release(so);              /* release the loc */
  *ret = i;
  pthread_exit(ret);
}
```

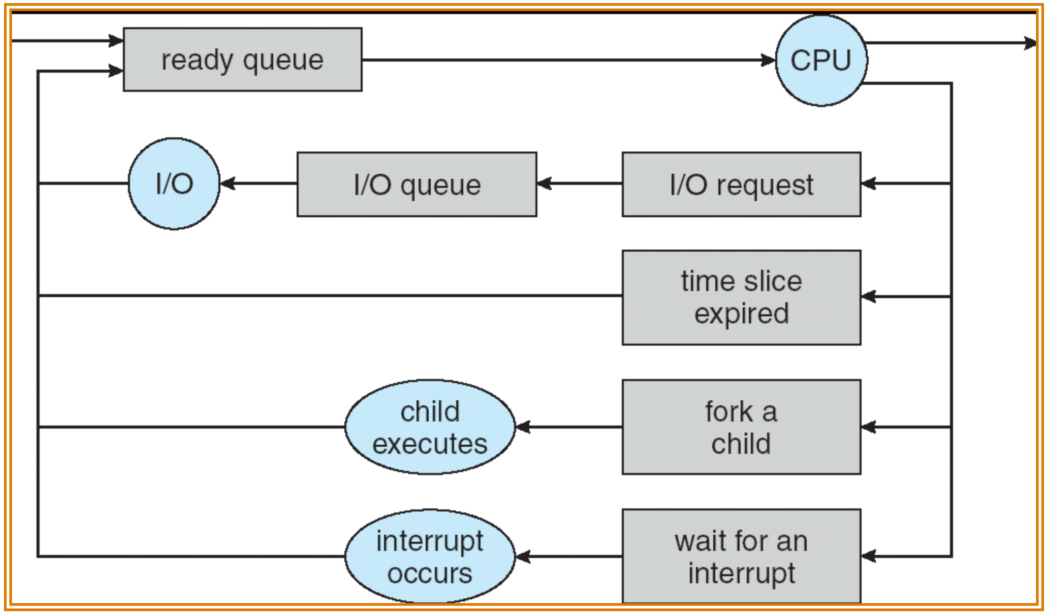
Scheduling

- the art, theory, and practice of deciding what to do next
- Ex: FIFO non-premptive scheduling
- Ex: Round-Robin
- Ex: Priority-based
- Ex: Coordinated

Definition

- Scheduling policy: algorithm for determining what to do next, when there are
  - multiple threads to run, or
  - multiple packets to send, or web requests to serve, or …
- Job or Task: unit of scheduling
  - quanta of a thread
  - program to completion
  - …
- Workload
  - Set of tasks for system to perform
  - Typically formed over time as scheduled tasks produce other tasks
- Metrics: properties that scheduling may seek to optimize

Processor Scheduling



- life-cycle of a thread
  - Active threads work their way from Ready queue to Running to various waiting queues.
- Scheduling: deciding which threads are given access to resources
- How to decide which of several threads to dequeue and run?
  - So far we have a single ready queue
  - Reason for wait->ready may make a big difference!

Concretely: Pintos Scheduler

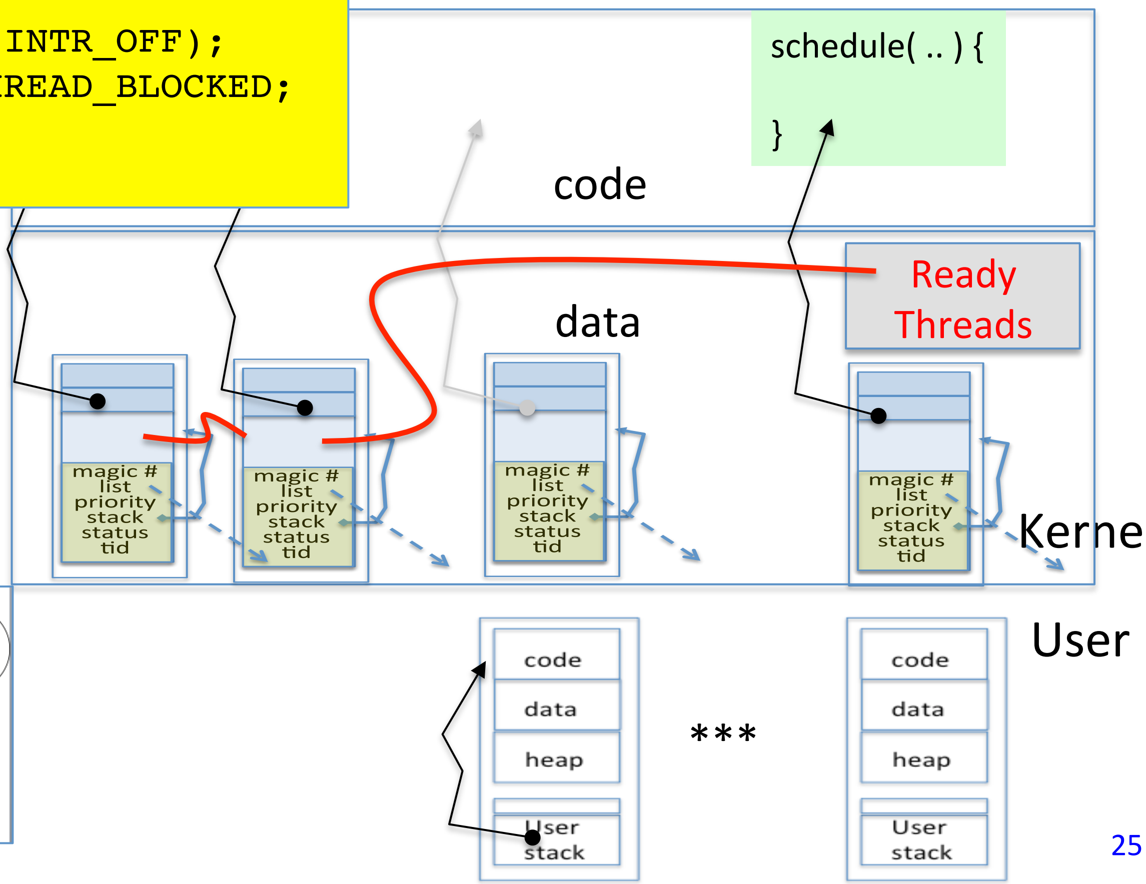
```
static void schedule (void) {
  struct thread *cur = running_thread ();
  struct thread *next = next_thread_to_run ();
  struct thread *prev = NULL;

  ASSERT (intr_get_level () == INTR_OFF);
  ASSERT (cur->status != THREAD_RUNNING);
  ASSERT (is_thread (next));

  if (cur != next)
    prev = switch_threads (cur, next);
  thread_schedule_tail (prev);
}
```

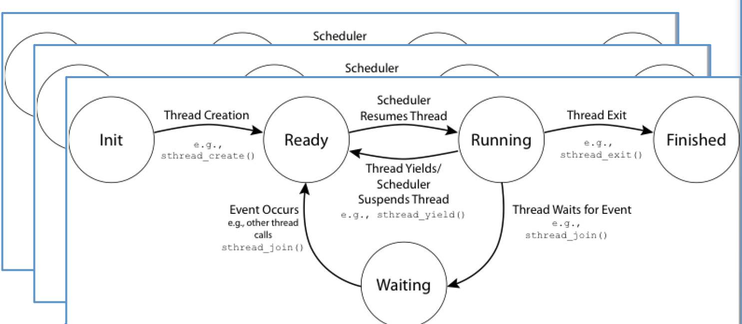
- Initially a round-robin scheduler of thread quanta
- Algorithm: next_thread_to_run

Kernel threads call into scheduler

- At various points (eg. sema_down) kernel thread must block itself
  - it calls schedule to allow next task to be selected

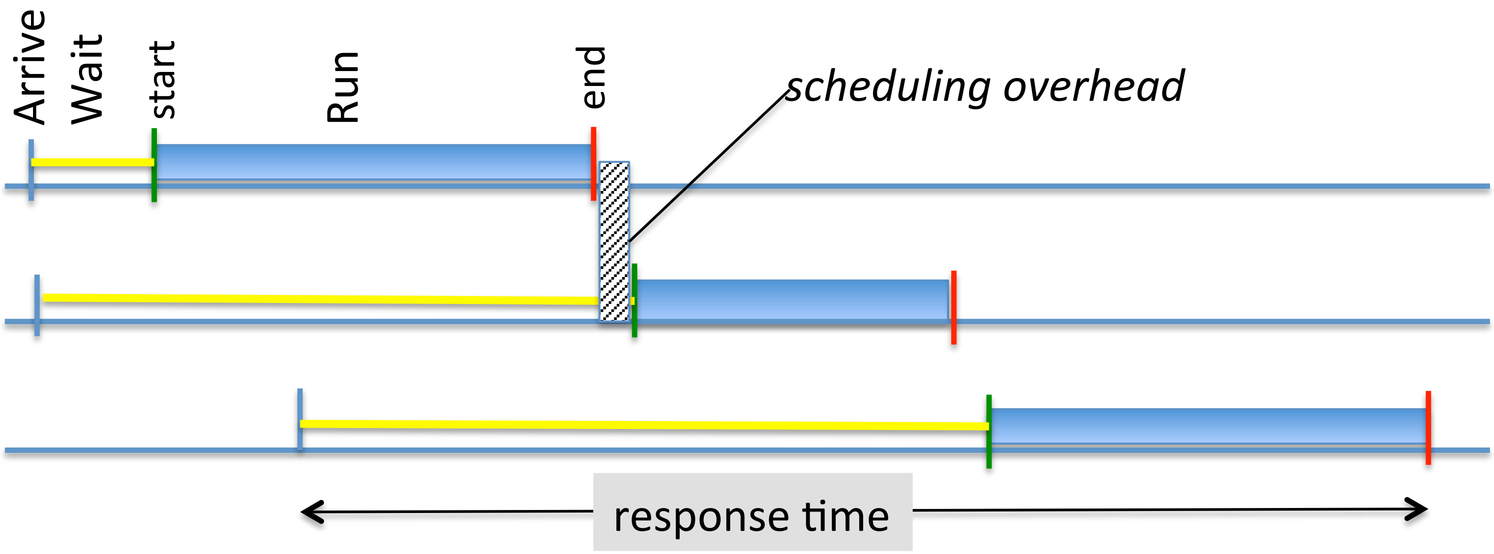
```
void thread_block (void) {
  ASSERT (!intr_context ());
  ASSERT (intr_get_level () == INTR_OFF);
  thread_current()->status = THREAD_BLOCKED;
  schedule ();
}
```





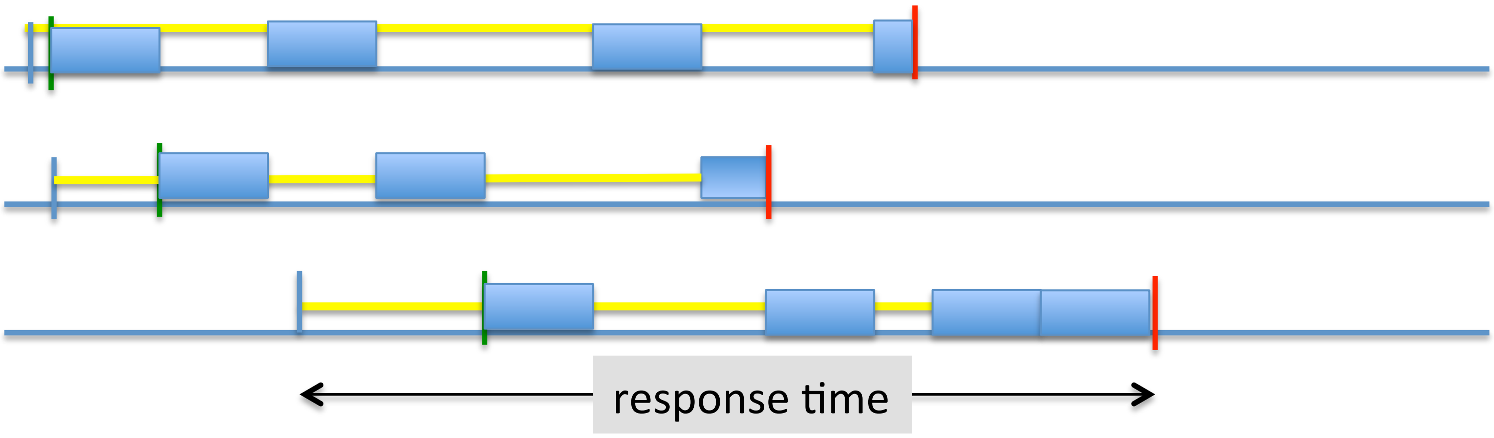
First In First Out - FCFS

- Schedule tasks in the order they arrive
  - Run until they complete or give up the processor



Round-Robin

- Each task gets a fixed amount of the resource (time quantum)
  - if does not complete, goes back into queue



- How large a time quantum?
  - Too short? Too long? Trade-offs?

Scheduling Metrics

- Waiting Time: time the job is waiting in the ready queue
  - Time between job's arrival in the ready queue and launching the job
- Service (Execution) Time: time the job is running
- Response (Completion) Time:
  - Time between job's arrival in the ready queue and job's completion
  - Response time is what the user sees:
    - Time to echo a keystroke in editor
    - Time to compile a program

Response Time = Waiting Time + Service Time

- Throughput: number of jobs completed per unit of time
  - Throughput related to response time, but not same thing:
    - Minimizing response time will lead to more context switching than if you only maximized throughput

Scheduling Policy Goals/Criteria

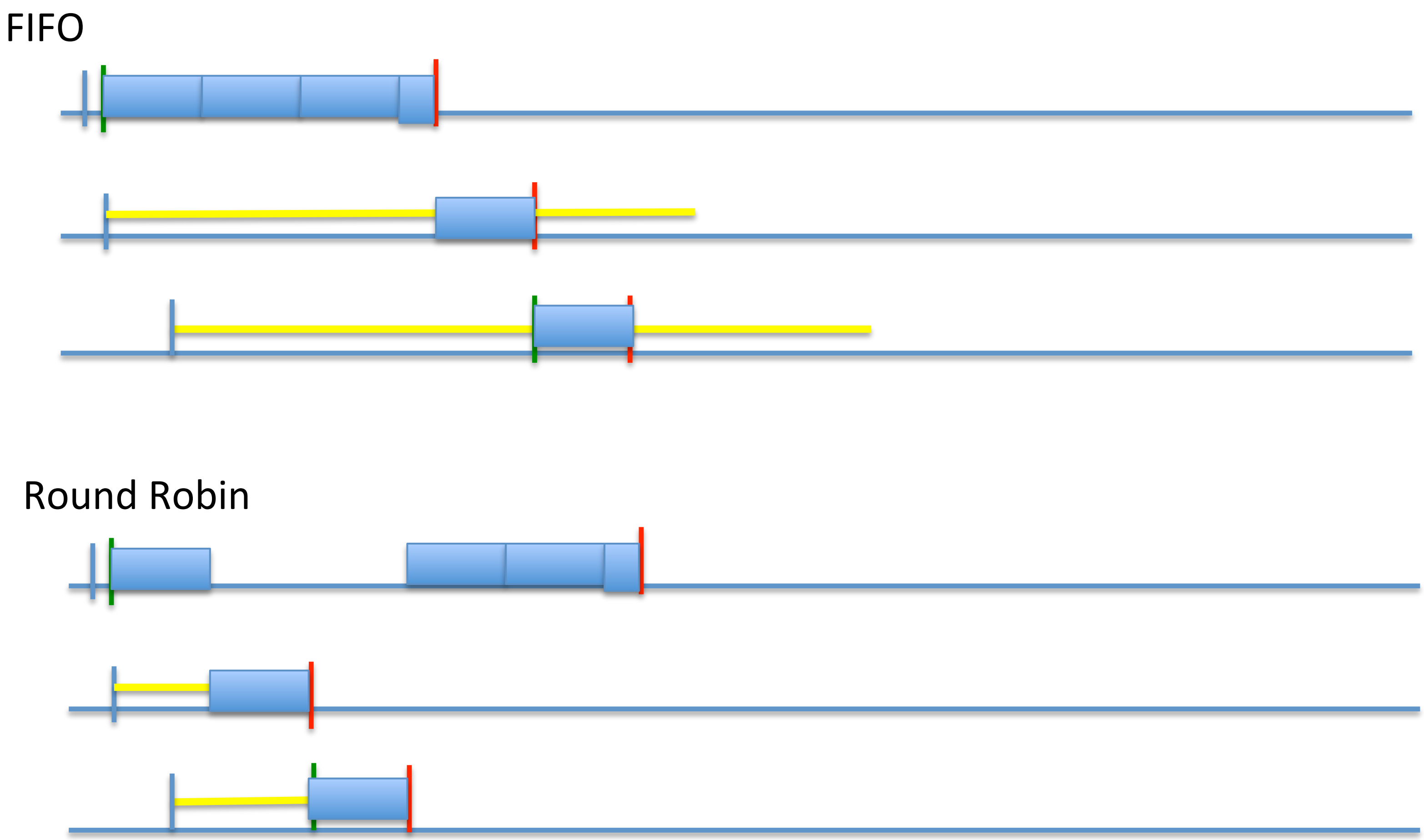
- Minimize Response Time
  - Minimize elapsed time to do an operation (or job)
- Maximize Throughput
  - Two parts to maximizing throughput
    - Minimize overhead (for example, context-switching)
    - Efficient use of resources (CPU, disk, memory, etc)
- Fairness
  - Share CPU among users in some equitable way
  - Fairness is not minimizing average response time:
    - Better *average* response time by making system *less* fair

Priority Scheduling

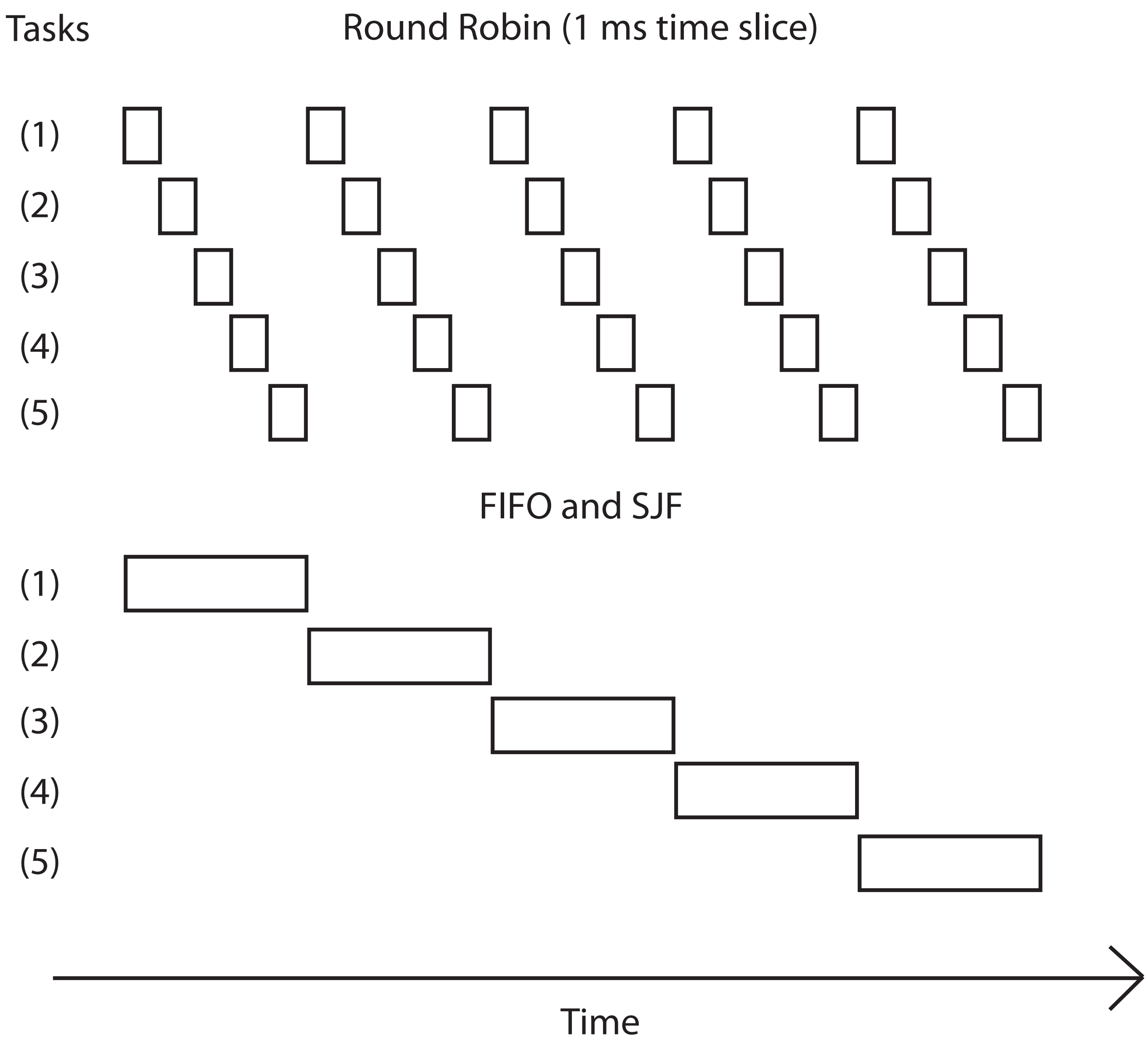
- Priorities can be a way to express desired outcome to the scheduler
  - important (high priority) tasks first, quicker, …
  - while low priority ones when resources available, …
- *Peer discussion: in groups of 2-4 come up with two ways to introduce priorities into FIFO and RR.*

- How might priorities interact positively / negatively with synchronization? With I/O ?

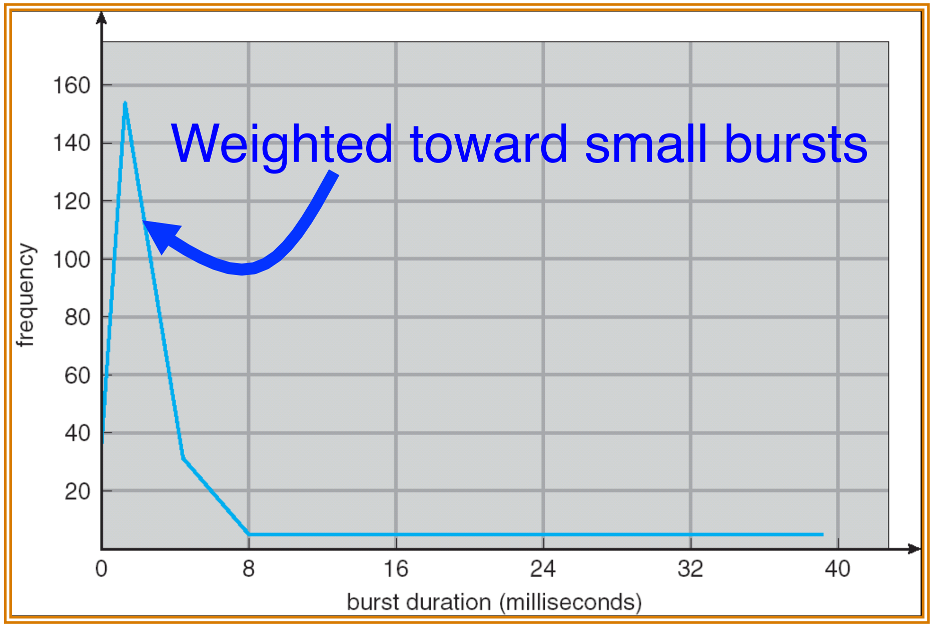
Round Robin vs FIFO

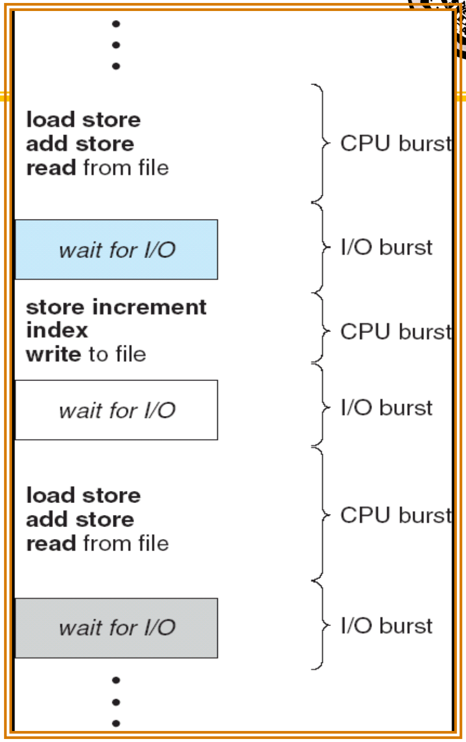


Round Robin vs. FIFO



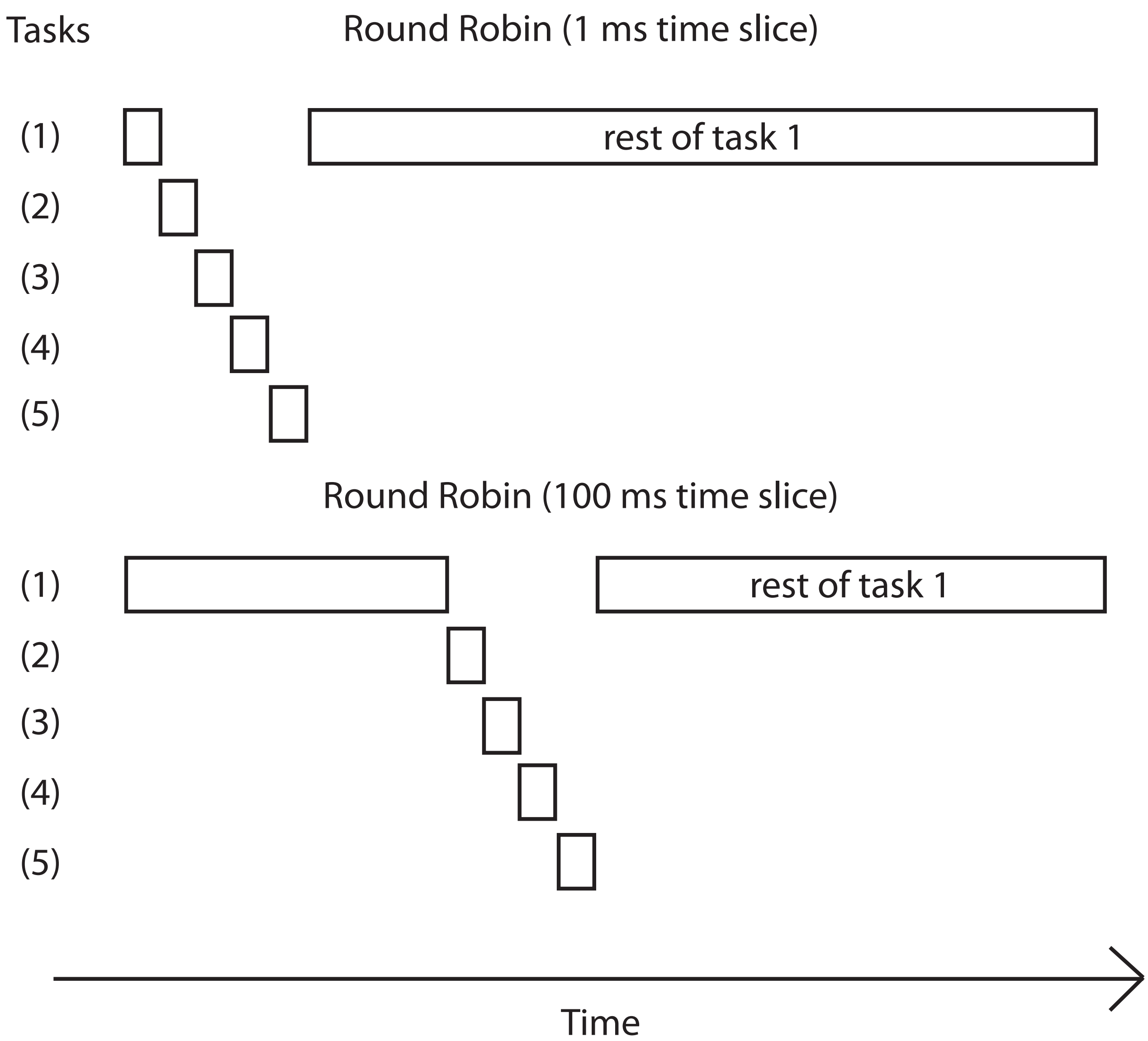
CPU Bursts



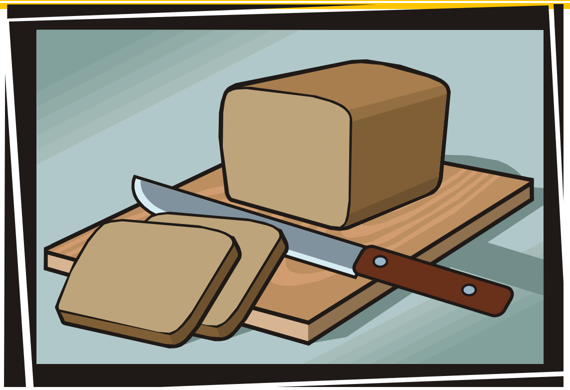


- Programs alternate between bursts of CPU and I/O
  - Program typically uses the CPU for some period of time, then does I/O, then uses CPU again
  - Each scheduling decision is about which job to give to the CPU for use by its next CPU burst
  - With timeslicing, thread may be forced to give up CPU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

Round Robin Slice



Round-Robin Discussion

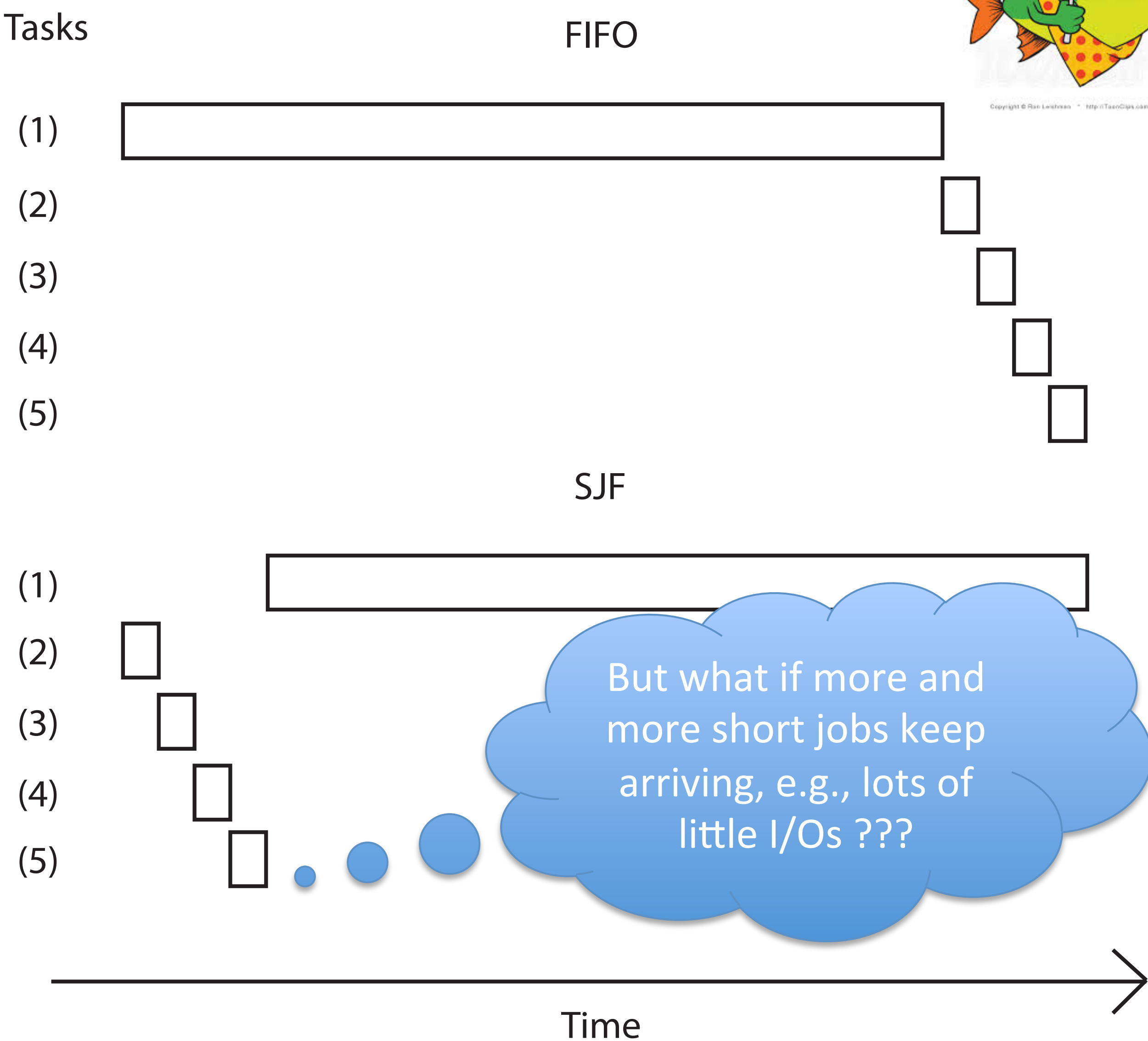


- How do you choose time slice?
  - What if too big?
    - Response time suffers
  - What if infinite (∞)?
    - Get back FCFS/FIFO
  - What if time slice too small?
    - Throughput suffers!
- Actual choices of timeslice:
  - Initially, UNIX timeslice one second:
    - Worked ok when UNIX was used by one or two people.
    - What if three compilations going on? 3 seconds to echo each keystroke!
  - In practice, 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

What if we Knew the Future?

- Shortest Job First (SJF):
  - Run whatever job has the least amount of computation to do
- 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
    - but how do you now???
- 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
- Want a simple approximation to SRTF …

FIFO vs. SJF

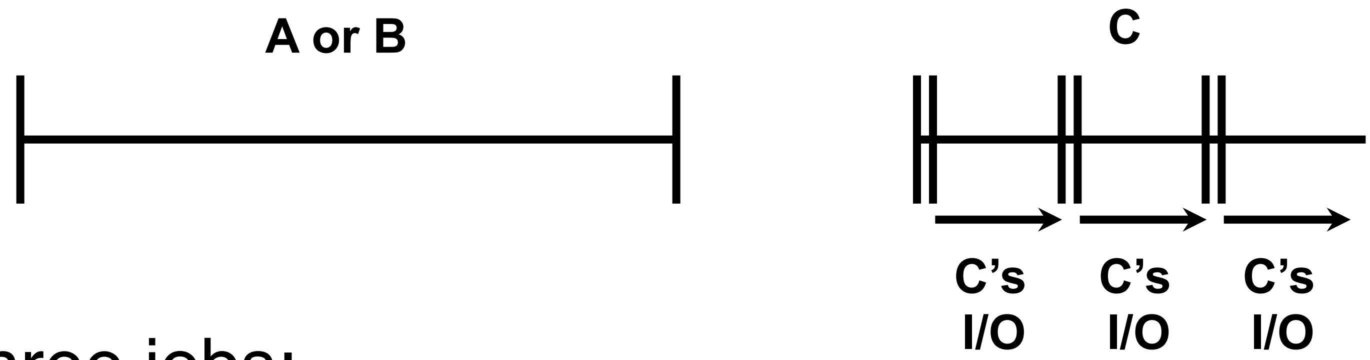

Tasks
FIFO
(1)
(2)
(3)
(4)
(5)
SJF
(1)
(2)
(3)
(4)
(5)
But what if more and more short jobs keep arriving, e.g., lots of little I/Os ???
Time

Discussion

- SJF/SRTF are best 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

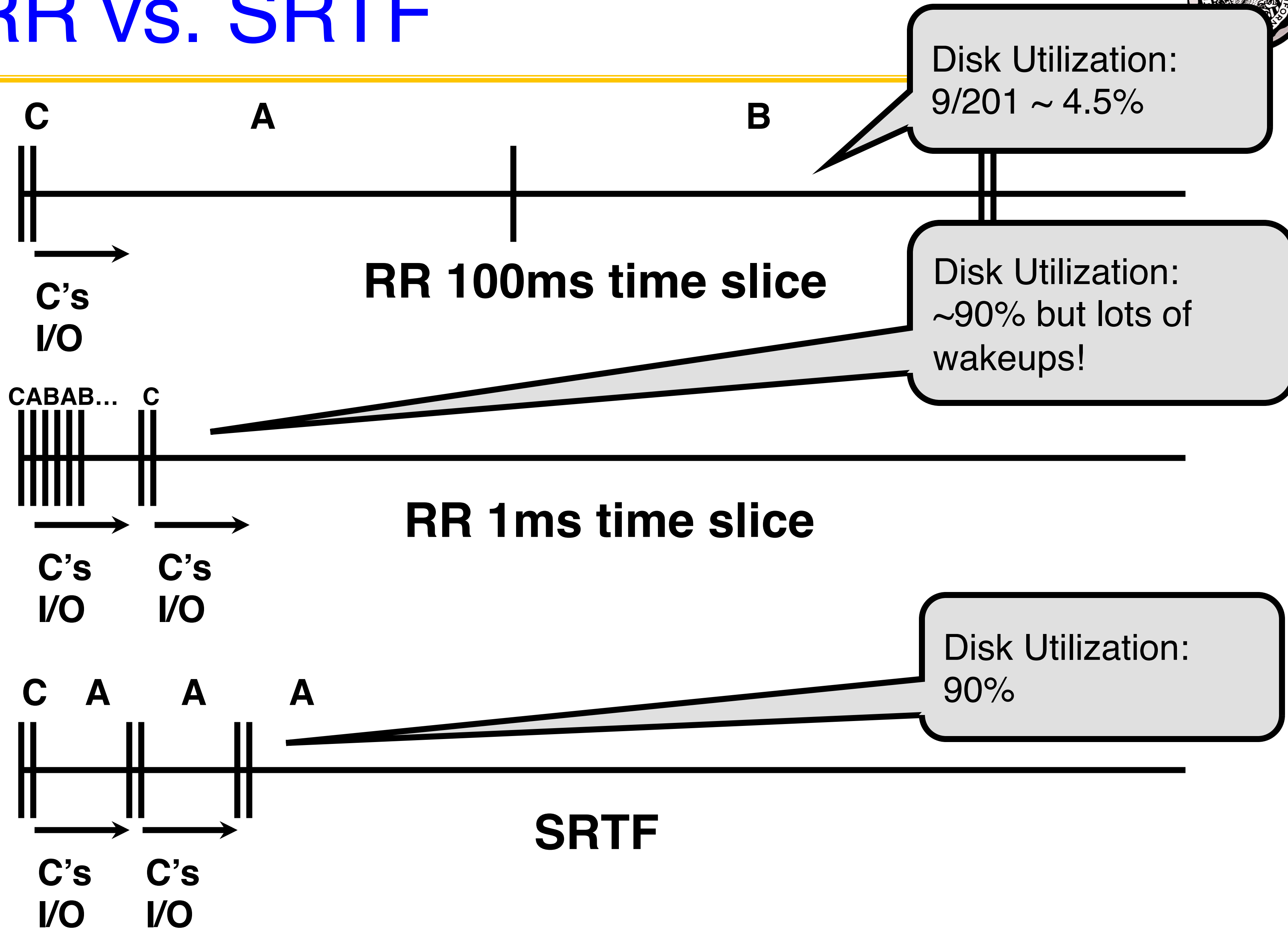
- Comparison of SRTF with FCFS and RR
  - What if all jobs the same length?
    - SJF becomes the same as FCFS (i.e., FCFS is best can do if all jobs the same length)
  - What if jobs have varying length?
    - SRTF (and RR): short jobs not stuck behind long ones

Example to illustrate benefits of SRTF



- Three jobs:
  - A,B: CPU bound, each run for a 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 use 100% of the CPU
- With FIFO:
  - Once A or B get in, keep CPU for one week each
- What about RR or SRTF?
  - Easier to see with a timeline

RR vs. SRTF



C
A
B
Disk Utilization:
9/201 ~ 4.5%
C's
I/O
RR 100ms time slice
CABAB… C
Disk Utilization:
~90% but lots of
wakeups!
C's
I/O
C's
I/O
RR 1ms time slice
C A A A
Disk Utilization:
90%
C's
I/O
C's
I/O
SRTF

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: even non-malicious users have trouble predicting runtime of their jobs
- Bottom line, can't really know how long job will take
  - However, can use SRTF as a yardstick for measuring other policies
  - Optimal => Practical approximations?
- SRTF Pros & Cons
  - Optimal (average response time) (+)
  - Hard to predict future (-)
  - Unfair (-)

Summary

- Scheduling: selecting a process from the ready queue and allocating the CPU to it
- FCFS Scheduling:
  - Run threads to completion in order of submission
  - Pros: Simple (+)
  - Cons: Short jobs get stuck behind long ones (-)
- 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 (+)
  - Cons: Poor when jobs are same length (-)
- Shortest Remaining Time First (SRTF):
  - Run whatever job has the least remaining amount of computation to do
  - Pros: Optimal (average response time)
  - Cons: Hard to predict future, Unfair

Backup Detail on Scheduling Trade-Offs

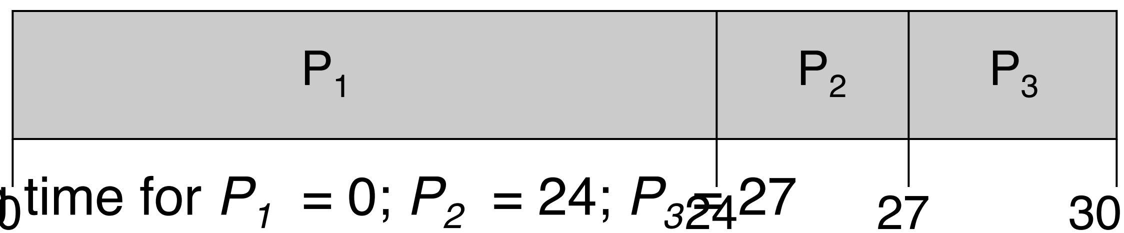
First-Come, First-Served (FCFS) Scheduling

- 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

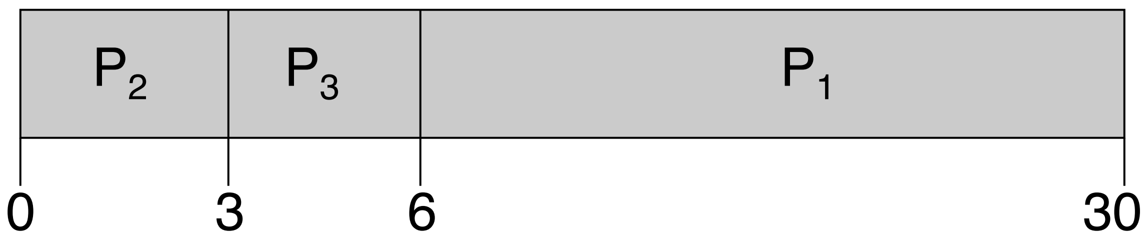
  - Suppose processes arrive in the order: P_1 , P_2 , P_3
    The Gantt Chart for the schedule is:



  - 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 behind long process

FCFS Scheduling (Cont.)

- Example continued:
  - Suppose that processes arrive in order: P_2 , P_3 , P_1
    Now, the Gantt chart for the schedule is:



  - Waiting time for $P_1 = 6; P_2 = 0; P_3 = 3$
  - 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)
- FCFS Pros and Cons:
  - Simple (+)
  - Short jobs get stuck behind long ones (-)
    - Safeway: Getting milk, always stuck behind cart full of small items

Round Robin (RR)

- 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
  - 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
- 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	Remaining Time
P_1	53	53
P_2	8	8
P_3	68	68
P_4	24	24

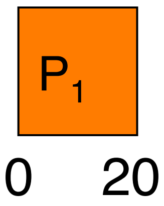
  - The Gantt chart is:

Example of RR with Time Quantum = 20

- Example:

Process	Burst Time	Remaining Time
P_1	53	33
P_2	8	8
P_3	68	68
P_4	24	24

  – The Gantt chart is:

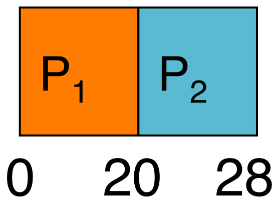


Example of RR with Time Quantum = 20

- Example:

Process	Burst Time	Remaining Time
P_1	53	33
P_2	8	0
P_3	68	68
P_4	24	24

  - The Gantt chart is:

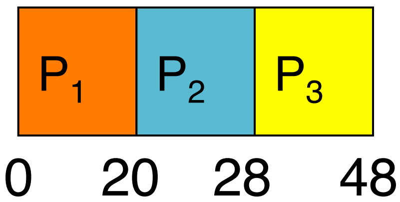


Example of RR with Time Quantum = 20

- Example:

Process	Burst Time	Remaining Time
P_1	53	33
P_2	8	0
P_3	68	48
P_4	24	24

  - The Gantt chart is:

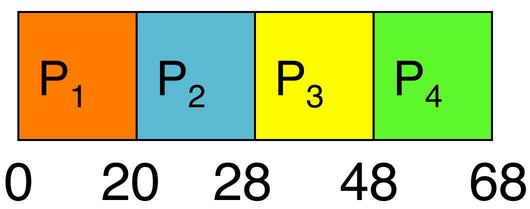


Example of RR with Time Quantum = 20

- Example:

Process	Burst Time	Remaining Time
P_1	53	33
P_2	8	0
P_3	68	48
P_4	24	4

  – The Gantt chart is:

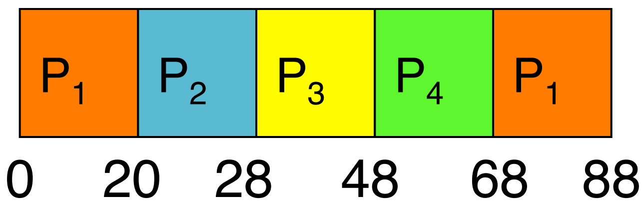


Example of RR with Time Quantum = 20

- Example:

Process	Burst Time	Remaining Time
P_1	53	13
P_2	8	0
P_3	68	48
P_4	24	4

  – The Gantt chart is:

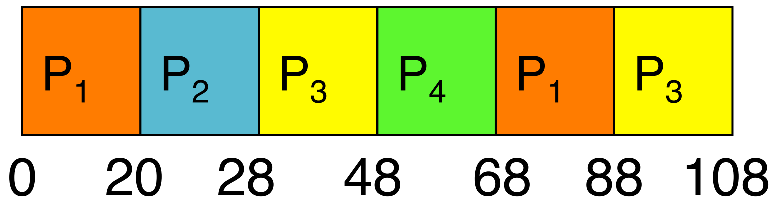


Example of RR with Time Quantum = 20

- Example:

Process	Burst Time	Remaining Time
P_1	53	13
P_2	8	0
P_3	68	28
P_4	24	4

  - The Gantt chart is:

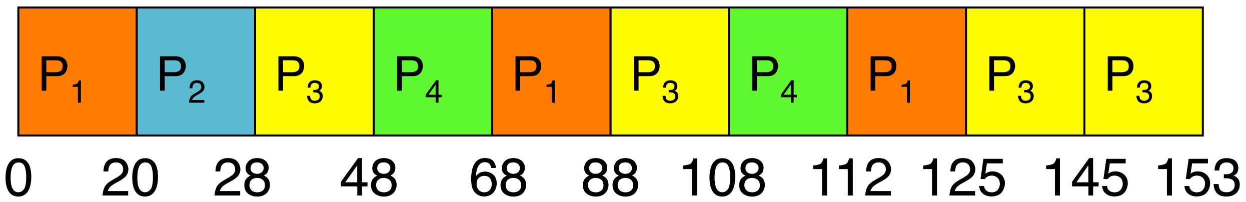


Example of RR with Time Quantum = 20

- Example:

Process	Burst Time	Remaining Time
P_1	53	0
P_2	8	0
P_3	68	0
P_4	24	0

  – The Gantt chart is:

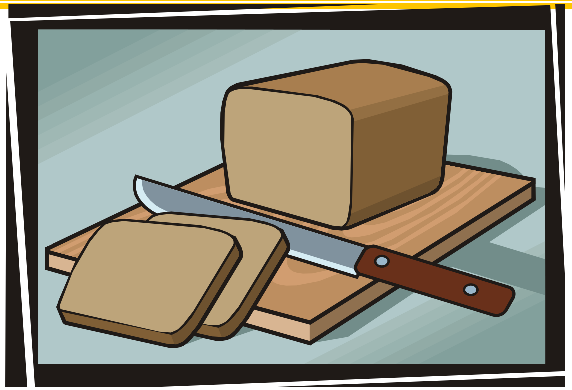


  – Waiting time for P_1=(68-20)+(112-88)=72
    P_2=(20-0)=20
    P_3=(28-0)+(88-48)+(125-108)=85
    P_4=(48-0)+(108-68)=88
  – Average waiting time = (72+20+85+88)/4=66¼
  – Average completion time = (125+28+153+112)/4 = 104½

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

Round-Robin Discussion

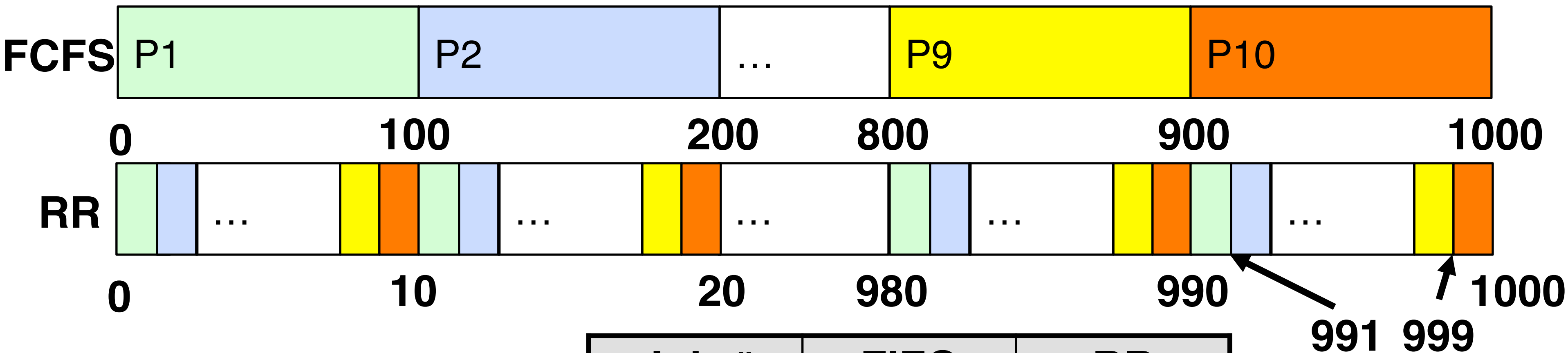
- How do you choose time slice?
  - What if too big?
    - Response time suffers
  - What if infinite (∞)?
    - Get back FCFS/FIFO
  - What if time slice too small?
    - Throughput suffers!



- 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!
  - In practice, 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 takes 100s of CPU time
  RR scheduler quantum of 1s
  All jobs start at the same time

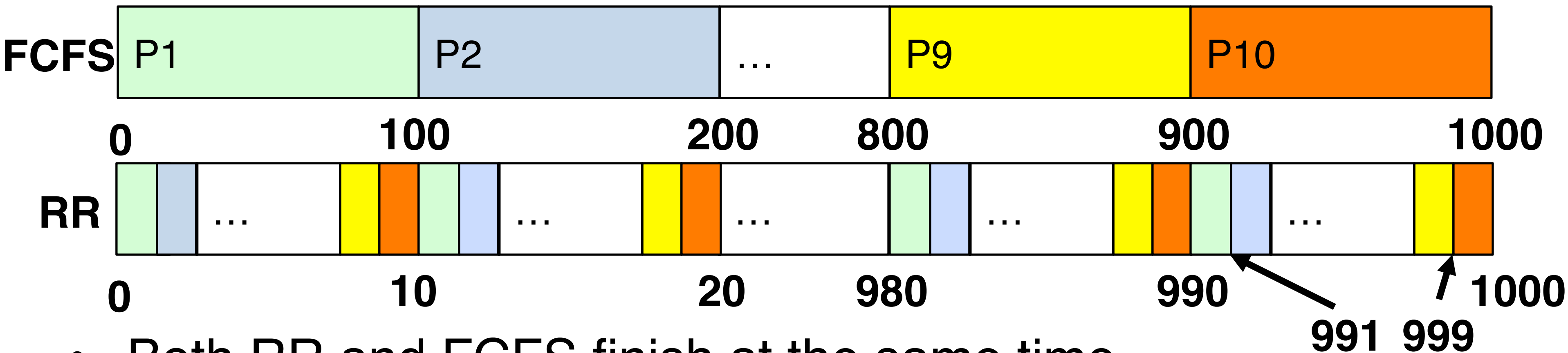


- Completion Tim
- FIFO average 5
- RR average 995.5!

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

Comparisons between FCFS and Round Robin

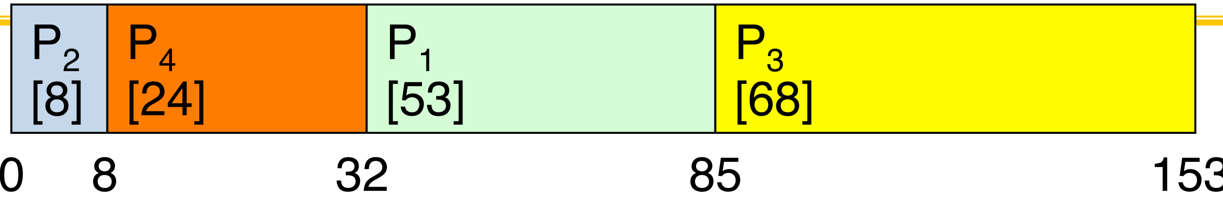
- Assuming zero-cost context-switching time, is RR always better than FCFS?
- Simple example: 10 jobs, each takes 100s of CPU time
  RR scheduler quantum of 1s
  All jobs start at the same time



- 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 FCFS
  – Total time for RR longer even for zero-cost switch!

Earlier Example with Different Time Quantum

Best FCFS:



	Quantum	P_1	P_2	P_3	P_4	Average
Wait Time	Best FCFS	32	0	85	8	31¼
Completion Time	Best FCFS	85	8	153	32	69½

Earlier Example with Different Time Quantum

Worst FCFS: P_3 [68] | P_1 [53] | P_4 [24] | P_2 [8]

0 68 121 145 153

	Quantum	P_1	P_2	P_3	P_4	Average
Wait Time	Best FCFS	32	0	85	8	31¼
	Worst FCFS	68	145	0	121	83½
Completion Time	Best FCFS	85	8	153	32	69½
	Worst FCFS	121	153	68	145	121¾

Earlier Example with Different Time Quantum

Worst FCFS:

P_3 [68]	P_1 [53]	P_4 [24]	P_2 [8]

0 68 121 145 153

P_1	P_2	P_3	P_4	P_1	P_3	P_4	P_1	P_3	P_4	P_1	P_3	P_1	P_3	P_1	P_3	P_1	P_3	P_3	P_3

0 8 **16** 24 32 40 48 56 64 72 **80** 88 96 104 112 120 128 **133** 141 149 **153**

	Quantum	P_1	P_2	P_3	P_4	Average
Wait Time	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