5.3 Scheduling

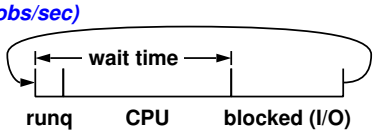
- *Goals*
- Scheduling Algorithms
- Implementation Issues
- Case Studies

Sample Sorts of Systems

- Simple batch
- Multiprogrammed batch
- Time sharing (i.e., interactive)
- Partitioned servers
- Real time *(i.e., thread must run before a deadline)*
  - *hard real time* (control) vs. *soft real time* (audio/video)
    - for hard real time system, missing deadline means *disaster*
      - e.g., controlling a nuclear power plant, landing (softly) on Mars
      - usually need specialized OS
    - for soft real time system, missing deadline *degrades quality* and user experience
      - e.g., playing streaming audio or video
      - can be supported by general purpose OS these days
- General purpose

Scheduling

- Goals (some are in conflict with one another)
  - *maximize CPU utilization*
  - *maximize throughput (jobs/sec)*



    - a thread sometimes can be classified as either *CPU-bound* or *I/O-bound*
  - *minimize wait time*
  - *minimize response time* (for interactive and real time applications)
  - *fairness*
    - approximately assign equal proportion of CPU to threads
- Scheduling is very important because its effects and how well it achieves the above goals can be *felt* by the user

5.3 Scheduling

- Goals
- *Scheduling Algorithms*
- Implementation Issues
- Case Studies

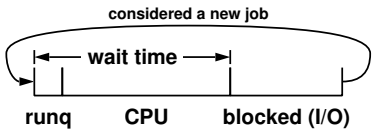
Scheduling Algorithms

Basic	Priority	Prortional Share	Real Time
FIFO	Multi-level	Lottery	EDF
SJF	Multi-level w/ Feedback*	Stride*	Rate Monotonic
SRTN			
RR			

- We will focus on how the *run queue* is managed

Scheduling Non-preemptive, Non-interactive Jobs

- Scheduling "jobs"



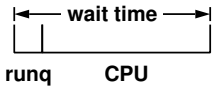
- Run one at a time
  - *no preemption*
- Running time is *known*

FIFO

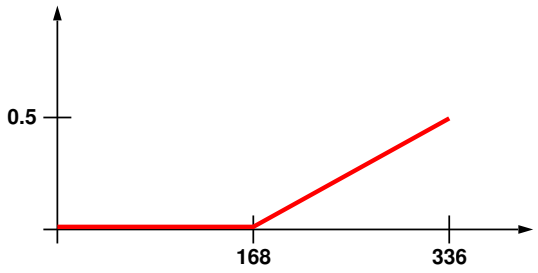
- Ex: weenix

Throughput

- "Goodness" criterion is jobs/hour
  - assuming that all jobs are sitting in the run queue at time 0



- Ex:
  - one 168-hour job
  - followed by 168 one-hour jobs



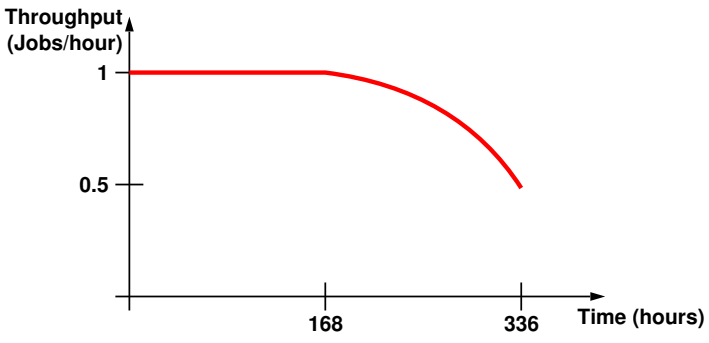
Average Wait Time

- Jobs J_i with processing times T_i for $0 \le i < n$
- *Average wait time (AWT)*
  - please note that this is *not* the same "wait time" in warmup #2
  - J_i started at time t_i
  - $t_i = \sum_{j=0}^{i-1} T_j$
  - $AWT = \sum_{i=0}^{n-1} (t_i + T_i) / n$
- For our example
  - AWT = 252 hours (with a standard deviation of 42.25 hours)
    - large average and large variation (for this example)

Shortest Job First

- $AWT = \sum_{i=0}^{n-1} (t_i + T_i) / n$
  - $t_i = \sum_{j=0}^{i-1} T_j$
- $AWT = (nT_0 + (n-1)T_1 + (n-2)T_2 + \ldots + 2T_{n-2} + T_{n-1}) / n$
- Minimized when $T_i \le T_{i+1}$ for all i
  - which is *Shortest Job First (SJF)*

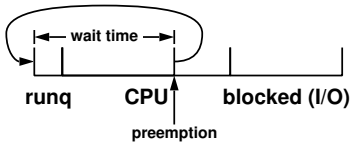
SJF and Our Example



- AWT = 86 hours (with a standard deviation of 52 hours)
- But, what if short jobs keep arriving?
  - *starvation*

Scheduling Preemptive, Non-interactive Jobs

- *Preemption:*
  - current job may be preempted by others
    - *shortest remaining time next (SRTN)*
      - *optimized throughput*
- Note: we will reserve the term "SJF" to refer to the non-preemptive case

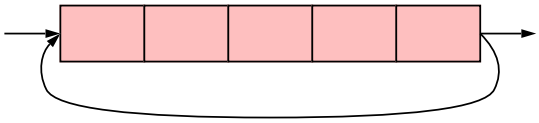


Fairness

- FIFO
  - each job eventually gets processed
- SJF and SRTN
  - a long job might have to wait indefinitely
- What's a good measure?

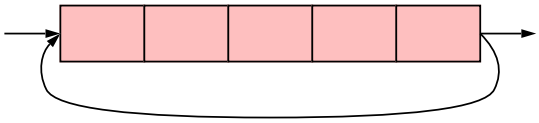
Round Robin

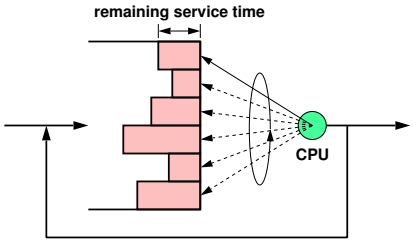
- Time-slicing
  - q = quatum / time slice



Round Robin

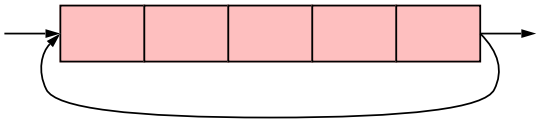
- Time-slicing
  - q = quatum / time slice





Round Robin

- Time-slicing
  - q = quatum / time slice



- Different values of q
  - q → 0: processor-sharing (idealized case)
    - not realistic
    - *translation lookaside buffer flushing* and *caching problem*
      - not enough time to achieve good hit-rate in TLB
  - q too large: some jobs appear to be not making progress

Round Robin + FIFO

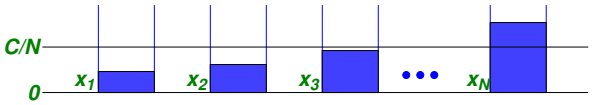
- AWT?
  - let quantum approach 0
  - 169 jobs sharing the processor
  - run at 1/169th speed for first week
  - short jobs receive one hour of processor time in 169 hours
    - different from SJF since all short jobs finish at about the same time
  - long job completes in 336 hours
  - AWT = 169.99 hours
    - recall that AWT is 252 hours for FIFO in our example and 86 hours for SJF
  - *average deviation = 1.96 hours*
    - recall that average deviation = 42.25 hours for FIFO in our example and 52 hours for SJF

Max-min Fairness

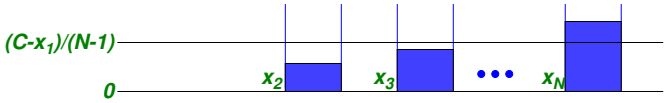
- *Max-min Fairness:* a fair service maximizes the service of the customer receiving the poorest service
- *Max-min Fairness* criterion:
  1) no user receives more than its *request*
  2) no other allocation scheme satisfying condition 1 has a high minimum allocation
  3) condition 2 remains recursively true as we remove the minimal user and reduce total resource accordingly

Max-min Fairness Example

- Total capacity *C* divided among *N* jobs
  - x_i is the request of job *i*
  - sort jobs based on x_i
  - initially, assign *C/N* to each job



  - satisfy x_1, redistribute remaining capacity evenly

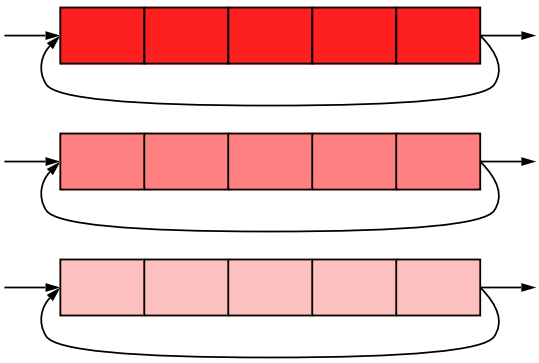


  - recursion
- This is basically *"processor sharing"*

Scheduling for Interactive Systems

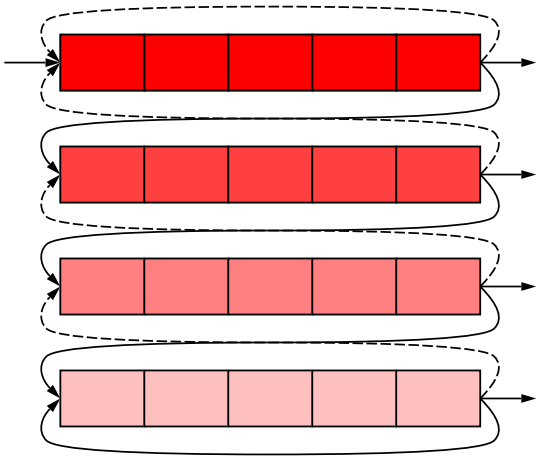
- *Length of "jobs" not known*
  - by the way, the round-robin scheduler works just fine without knowing the length of jobs
- Threads would give up the CPU voluntarily
  - they *block* for user input
- Would like to *favor interactive jobs*
  - use *priority queueing*

Round Robin with Priority



- *how* to determine priority?
  - let the threads themselves decide?

Multi-Level Feedback Queues



Multi-Level Feedback Queues

- When a thread got created, it gets highest priority
  - schedule it at the highest priority to observe what it does
    - if it *uses a full time slice*
      - *decrease its priority*
    - if it *blocks* before using up a full time slice
      - *increase its priority*
- To avoid starvation, use *aging*
  - if a job hasn't been run for a long time, increase its priority
- Clearly, not a fair scheduling algorithm

Real-Life Example

- Your iPod is broken
  - run mp3 player on your PC
- The baseball playoffs are on
  - streaming video
- An OS assignment is due
  - editor, compiler, debugger
- You've got to do everything on one computer
- Can your scheduler hack it?
- What scheduler is suitable for a general purpose system?

```
static int msync_interval(struct vm_area_struct * vma,
    unsigned long start, unsigned long end, int flags)

    int ret = 0;
    struct file * file = vma->vm_file;

    if ((flags & MS_INVALIDATE) && (vma->vm_flags & VM_LOCKED))
        return -EBUSY;

    if (file && (vma->vm_flags & VM_SHARED)) {
        ret = filemap_sync(vma, start, end-start, flags);

        if (!ret && (flags & MS_SYNC)) {
            struct address_space *mapping = file->f_mapping;
            int err;
```

Interactive Scheduling

- *Time-sliced*, *priority-based*, *preemptive*
  - e.g., multi-level feedback queues
- Priority depends on expected time to block
  - interactive threads should have high priority
  - compute threads should have low priority
- Other heuristics
  - e.g., determine priority using *long term history* (not just immediate history)
    - processor usage causes decrease
    - sleeping causes increase

Scheduling for Fairness

- If *fairness* is really important, what would you do?
- Ex: shared servers
  - you and four friends each contribute $1000 towards a server
    - you, rightfully, feel you own 20% of it
  - your friends are into threads, you're not
    - they run 5-threaded programs
    - you run a 1-threaded program
  - their programs each get 5/21 of the processor
  - your programs get 1/21 of the processor

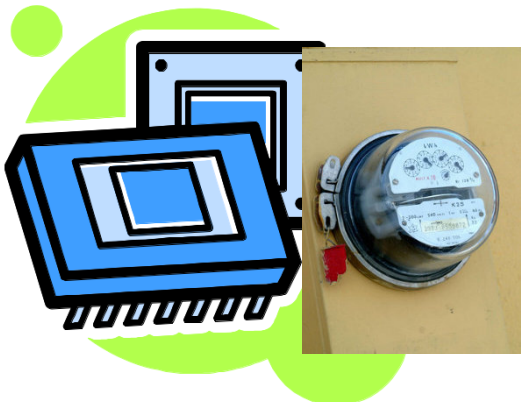
Lottery Scheduling

- 25 lottery tickets are distributed equally to you and your four friends
  - you give 5 tickets to your one thread
  - they give one ticket each to their threads
- A lottery is held for every scheduling decision
  - your thread is 5 times more likely to win than the others

Proportional-Share Scheduling

- Stride scheduling
  - 1995 paper by Waldspurger and Weihl
- Completely fair scheduling (CFS)
  - added to Linux in 2007

Metered Processors



- the textbook presented *Stride Scheduling* differently
  - as far as exam goes, stick to our presentation

Stride Scheduling Algorithm

- *Time-sliced*, *priority-based*, *preemptive*
  - every thread is assigned a priority, called a *pass* value
    - single queue, *sorted based on pass values*, *smallest first*
  - every thread is assigned a *stride* value
    - stride values are computed according to *distribution of tickets* in a lottery scheduling scheme
- In every iteration / time-slice
  1) shedule the thread with the *smallest pass value* (at the head of the queue)
  2) set the *global pass value* to be the pass value of this thread
  3) *increment* the thread's *pass* value by its *stride* value
  4) loop

Stride Scheduling Example

- Stride $\propto$ 1 / number of tickets

Thread	Tickets	Stride
A	3	
B	2	
C	1	

Stride Scheduling Example

- Stride $\propto$ 1 / number of tickets

Thread	Tickets	Stride
A	3	1/3
B	2	1/2
C	1	1

then multiply by smallest common multiplier of denominators to get *interger stride widths* (here and for *exams*)

Stride Scheduling Example

- Stride $\propto$ 1 / number of tickets

Thread	Tickets	Stride
A	3	2
B	2	3
C	1	6

Stride Scheduling Example

- Stride $\propto$ 1 / number of tickets
  - every thread is initialized with a *pass* value

Thread	Tickets	Stride
A	3	2
B	2	3
C	1	6

can start with any pass values (e.g., determined by the *current state* of the stride scheduler)

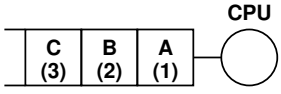
itr	A	B	C
1	1	2	3

Stride Scheduling Example

- Stride $\propto$ 1 / number of tickets
  - every thread is initialized with a *pass* value

Thread	Tickets	Stride
A	3	2
B	2	3
C	1	6

itr	A	B	C
1	1	2	3

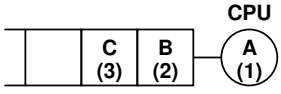


Stride Scheduling Example

- Stride $\propto$ 1 / number of tickets
  - every thread is initialized with a *pass* value

Thread	Tickets	Stride
A	3	2
B	2	3
C	1	6

itr	A	B	C
1	(1)	2	3

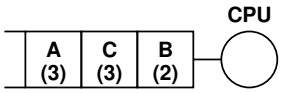


Stride Scheduling Example

- Stride $\propto$ 1 / number of tickets
  - every thread is initialized with a *pass* value

Thread	Tickets	Stride
A	3	2
B	2	3
C	1	6

itr	A	B	C
1	(1)	2	3
2	3	2	3

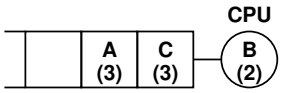


Stride Scheduling Example

- Stride $\propto$ 1 / number of tickets
  - every thread is initialized with a *pass* value

Thread	Tickets	Stride
A	3	2
B	2	3
C	1	6

itr	A	B	C
1	(1)	2	3
2	3	(2)	3

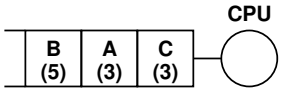


Stride Scheduling Example

- Stride $\propto$ 1 / number of tickets
  - every thread is initialized with a *pass* value

Thread	Tickets	Stride
A	3	2
B	2	3
C	1	6

itr	A	B	C
1	(1)	2	3
2	3	(2)	3
3	3	5	3

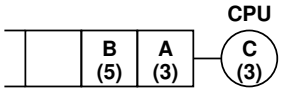


Stride Scheduling Example

- Stride $\propto$ 1 / number of tickets
  - every thread is initialized with a *pass* value

Thread	Tickets	Stride
A	3	2
B	2	3
C	1	6

itr	A	B	C
1	(1)	2	3
2	3	(2)	3
3	3	5	(3)

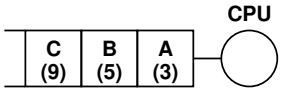


Stride Scheduling Example

- Stride $\propto$ 1 / number of tickets
  - every thread is initialized with a *pass* value

Thread	Tickets	Stride
A	3	2
B	2	3
C	1	6

itr	A	B	C
1	(1)	2	3
2	3	(2)	3
3	3	5	(3)
4	3	5	9

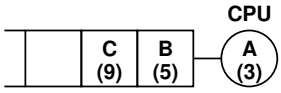


Stride Scheduling Example

- Stride $\propto$ 1 / number of tickets
  - every thread is initialized with a *pass* value

Thread	Tickets	Stride
A	3	2
B	2	3
C	1	6

itr	A	B	C
1	(1)	2	3
2	3	(2)	3
3	3	5	(3)
4	(3)	5	9

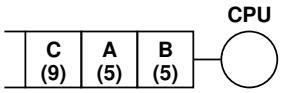


Stride Scheduling Example

- Stride $\propto$ 1 / number of tickets
  - every thread is initialized with a ***pass*** value

Thread	Tickets	Stride
A	3	2
B	2	3
C	1	6

itr	A	B	C
1	(1)	2	3
2	3	(2)	3
3	3	5	(3)
4	(3)	5	9
5	5	5	9

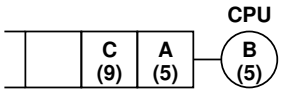


Stride Scheduling Example

- Stride $\propto$ 1 / number of tickets
  - every thread is initialized with a ***pass*** value

Thread	Tickets	Stride
A	3	2
B	2	3
C	1	6

itr	A	B	C
1	(1)	2	3
2	3	(2)	3
3	3	5	(3)
4	(3)	5	9
5	5	(5)	9

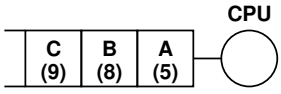


Stride Scheduling Example

- Stride $\propto$ 1 / number of tickets
  - every thread is initialized with a ***pass*** value

Thread	Tickets	Stride
A	3	2
B	2	3
C	1	6

itr	A	B	C
1	(1)	2	3
2	3	(2)	3
3	3	5	(3)
4	(3)	5	9
5	5	(5)	9
6	5	8	9

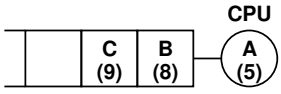


Stride Scheduling Example

- Stride $\propto$ 1 / number of tickets
  - every thread is initialized with a ***pass*** value

Thread	Tickets	Stride
A	3	2
B	2	3
C	1	6

itr	A	B	C
1	(1)	2	3
2	3	(2)	3
3	3	5	(3)
4	(3)	5	9
5	5	(5)	9
6	(5)	8	9

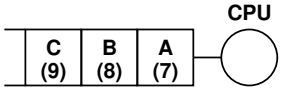


Stride Scheduling Example

- Stride $\propto$ 1 / number of tickets
  - every thread is initialized with a ***pass*** value

Thread	Tickets	Stride
A	3	2
B	2	3
C	1	6

itr	A	B	C
1	(1)	2	3
2	3	(2)	3
3	3	5	(3)
4	(3)	5	9
5	5	(5)	9
6	(5)	8	9
7	7	8	9

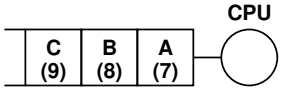


Stride Scheduling Example

- Stride $\propto$ 1 / number of tickets
  - every thread is initialized with a ***pass*** value
- Conforms to the distribution of tickets!

Thread	Tickets	Stride
A	3	2
B	2	3
C	1	6

itr	A	B	C
1	(1)	2	3
2	3	(2)	3
3	3	5	(3)
4	(3)	5	9
5	5	(5)	9
6	(5)	8	9
7	7	8	9



Stride Scheduling - Additional Details

- New thread
  - allocate the *global pass value*
    - so it gets to run first
- Thread uses less than its quantum
  - let 0 < f < 1
  - pass += f × stride
  - the result is that *interactive threads* get *higher priority*

Scheduling in Hard Real Time Systems

- Known chores and durations
  - find schedule satisfying constraints
  - uniprocessor
    - rate-monotonic scheduling of *cyclic chores*, i.e., *periodic tasks*
      - $t_i \rightarrow T_i, P_i$ where T is *execution time* and P is *period*
      - e.g., a job is required to execute 2 seconds every 6 seconds
      - deadline is implied
    - Earliest Deadline First (EDF)
      - $t_i \rightarrow d_i$ where d is a *deadline*
      - optimality proof exists under certain conditions
  - multiprocessor
    - often NP-complete ...

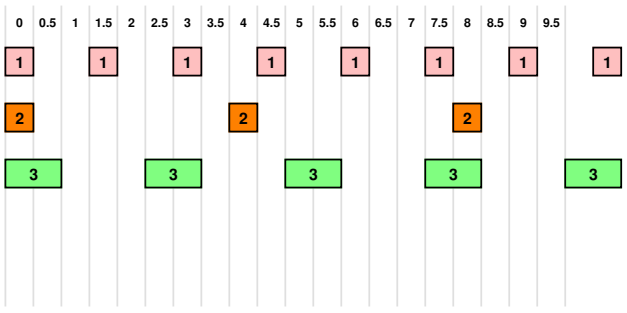
Assumptions

- Interrupts don't interfere (too much) with schedule
  - bounded interrupt delays
- Execution time really is predictable
  - what about effects of caching and paging?

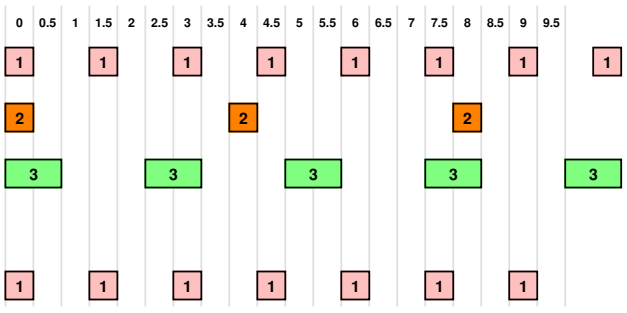
Rate-Monotonic Scheduling

- Periodic chores
  - period P_i
  - per-cycle processing time T_i ($\leq P_i$)
  - feasible if $\sum_{i=0}^{n-1} (T_i / P_i) \leq 1$
- Rate-monotonic scheduling
  - each chore *i* is handled by a thread with *priority* $1/P_i$
  - *preemptive*, *priority scheduling*
  - works when $\sum_{i=0}^{n-1} (T_i / P_i) \leq n(2^{1/n}-1)$
  - $= \ln 2$ in the limit

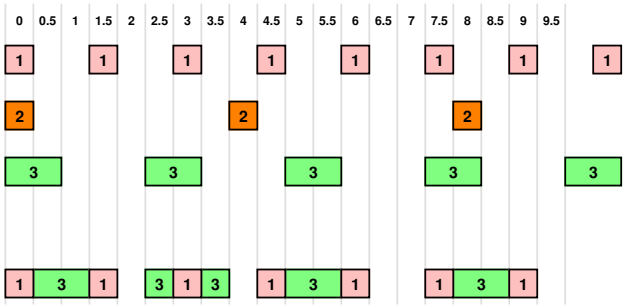
Scenario 1



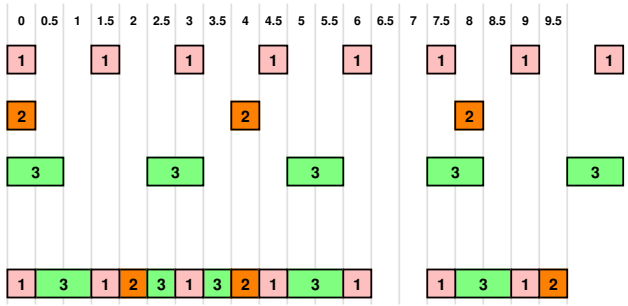
Scenario 1



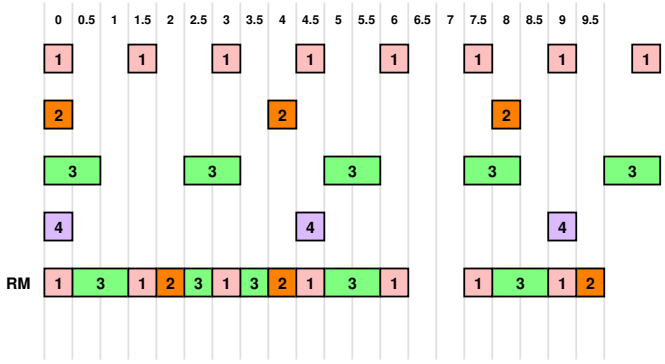
Scenario 1



Scenario 1

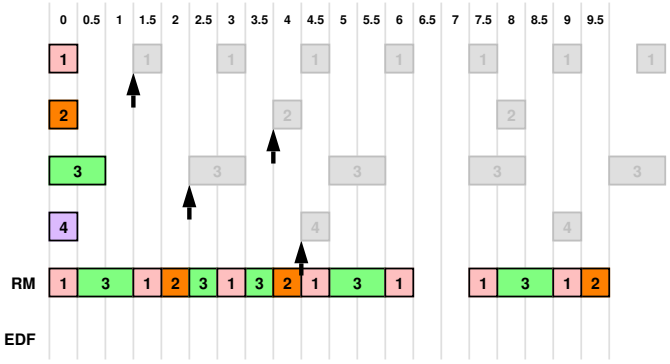


Scenario 2

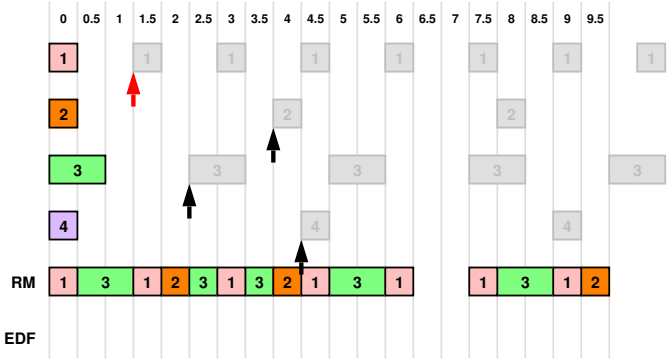


- rate-monotonic scheduler failed
- but EDF succeeds in this example

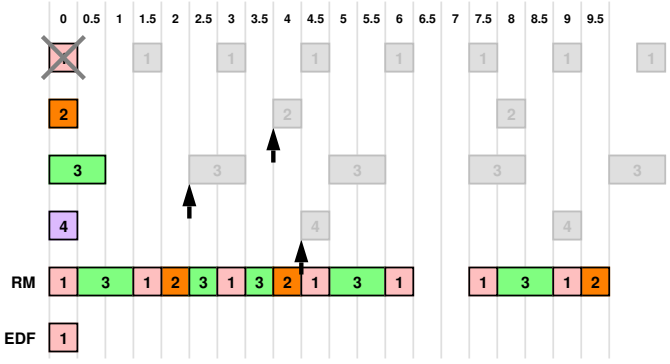
Earliest Deadline First



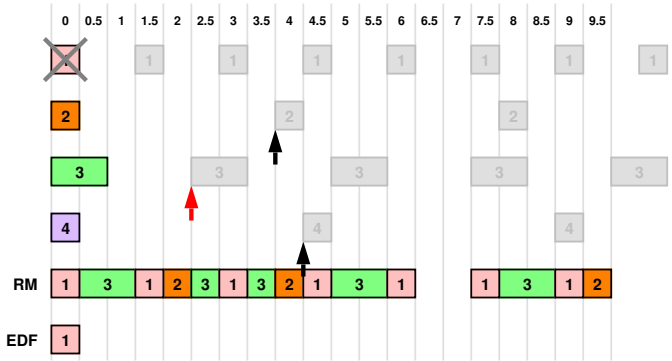
Earliest Deadline First



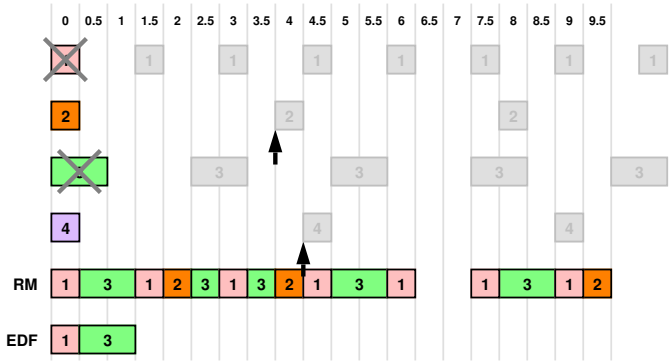
Earliest Deadline First



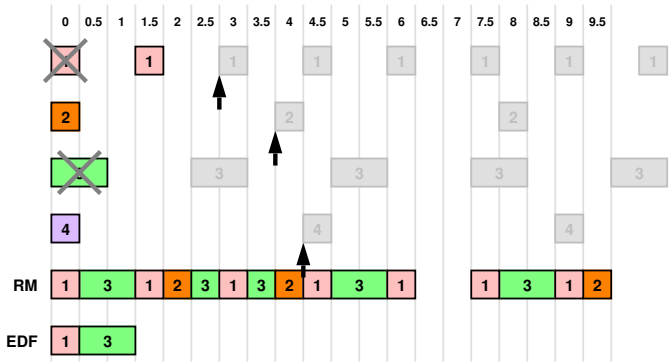
Earliest Deadline First



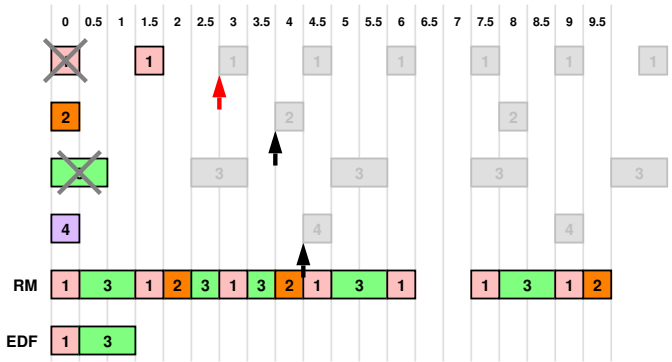
Earliest Deadline First



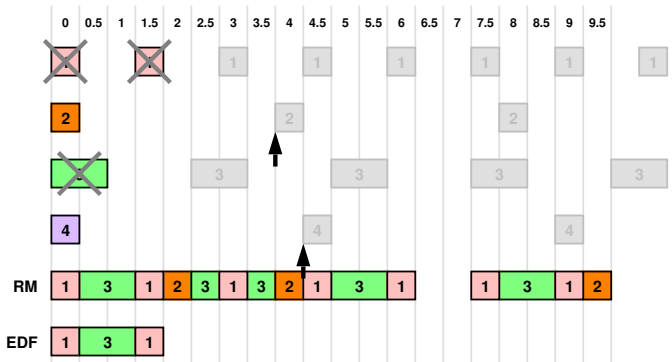
Earliest Deadline First



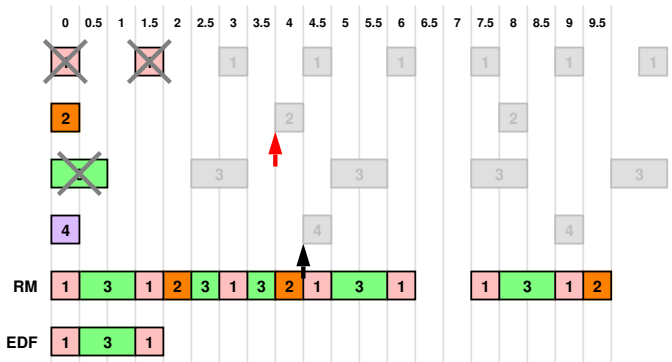
Earliest Deadline First



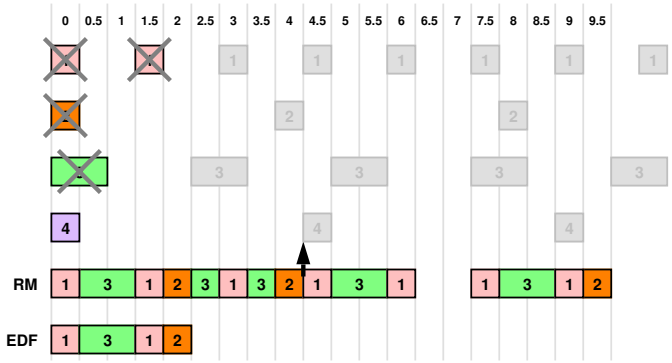
Earliest Deadline First



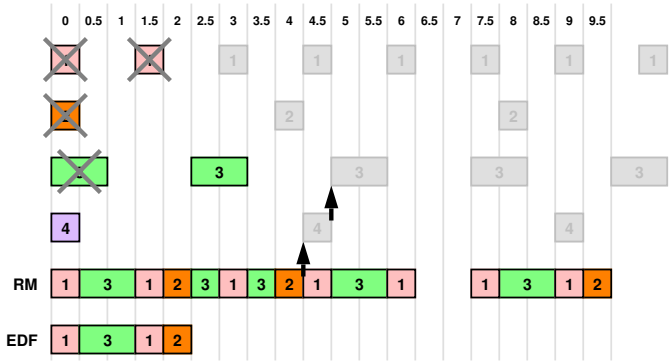
Earliest Deadline First



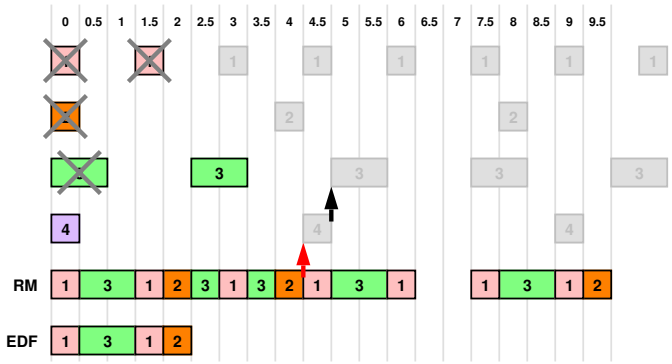
Earliest Deadline First



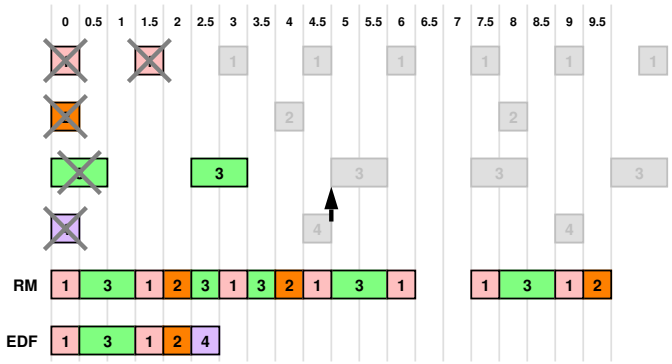
Earliest Deadline First



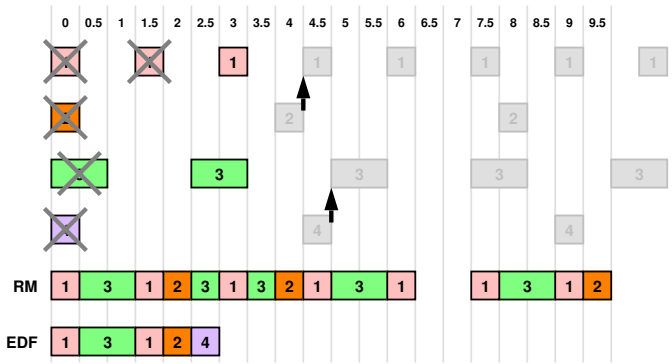
Earliest Deadline First



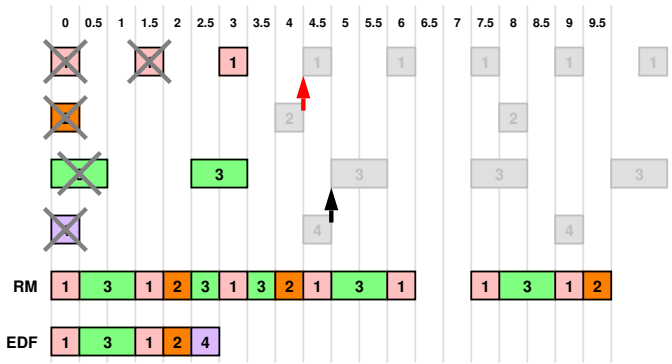
Earliest Deadline First



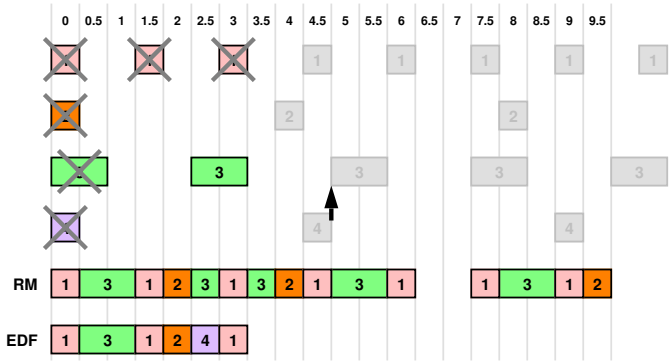
Earliest Deadline First



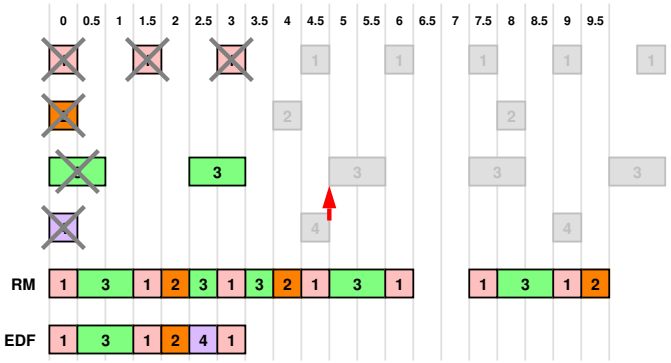
Earliest Deadline First



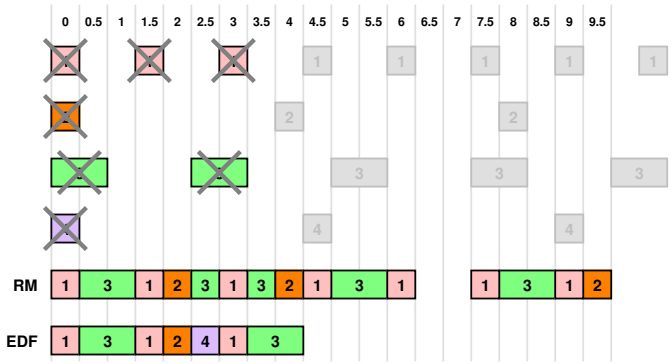
Earliest Deadline First



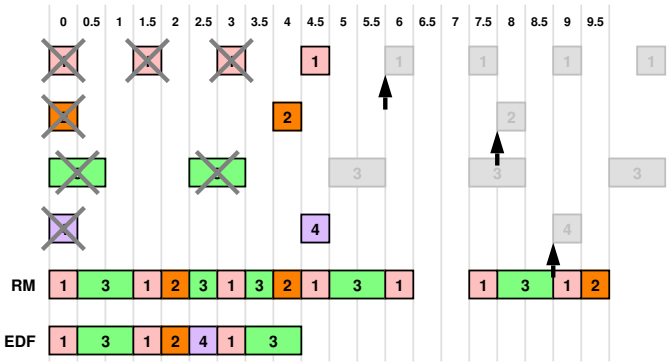
Earliest Deadline First



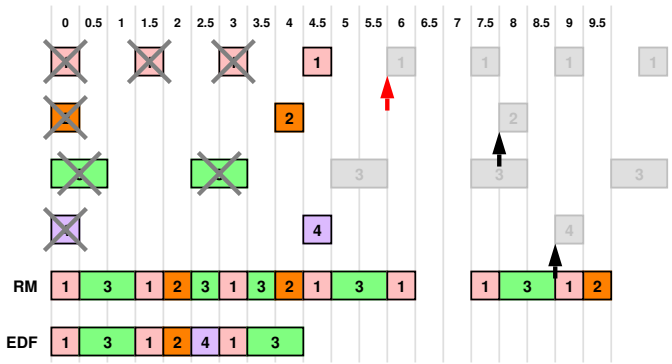
Earliest Deadline First



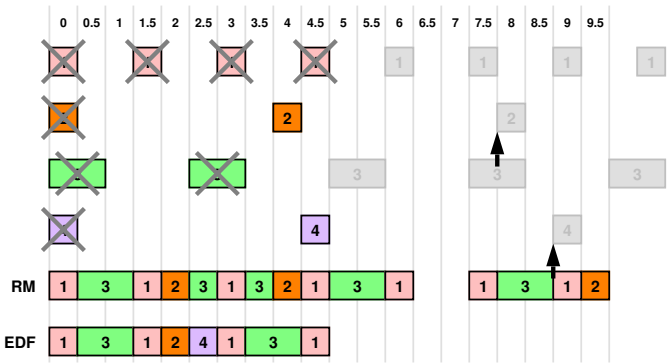
Earliest Deadline First



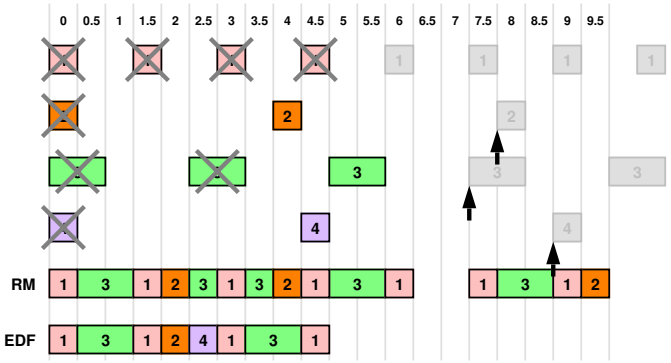
Earliest Deadline First



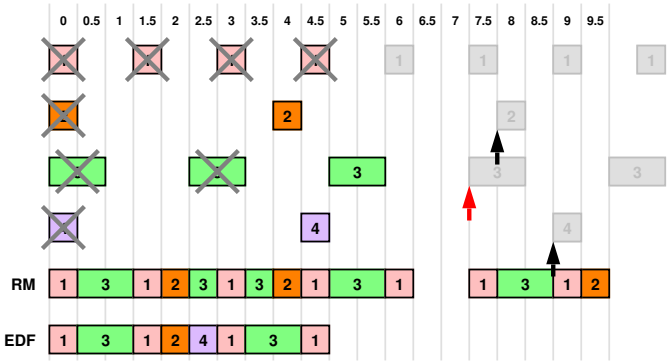
Earliest Deadline First



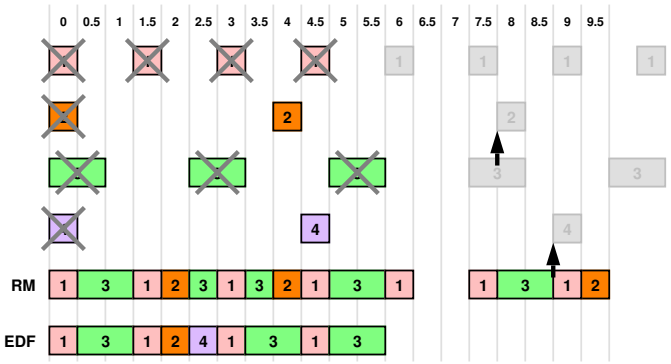
Earliest Deadline First



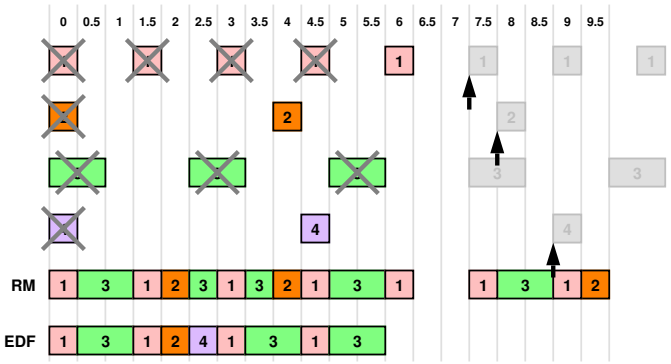
Earliest Deadline First



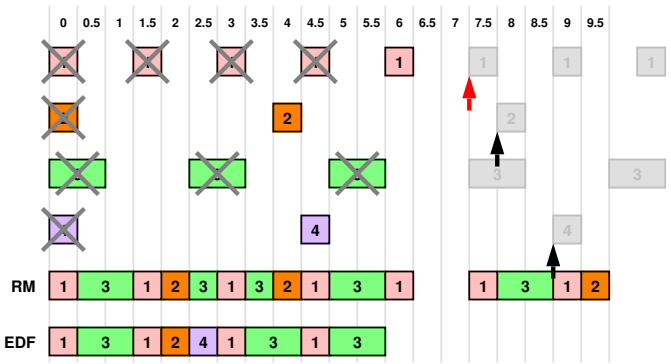
Earliest Deadline First



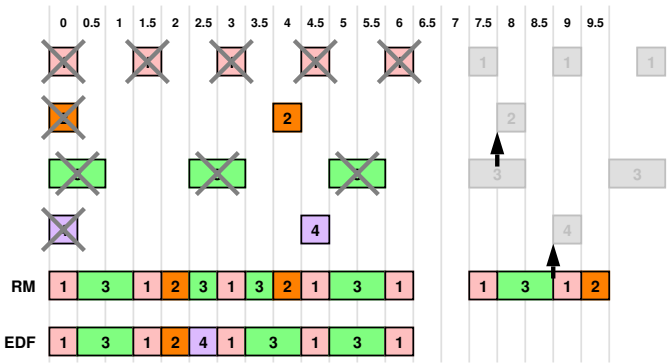
Earliest Deadline First



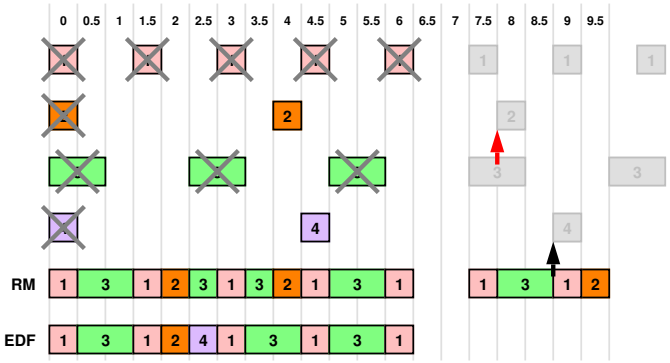
Earliest Deadline First



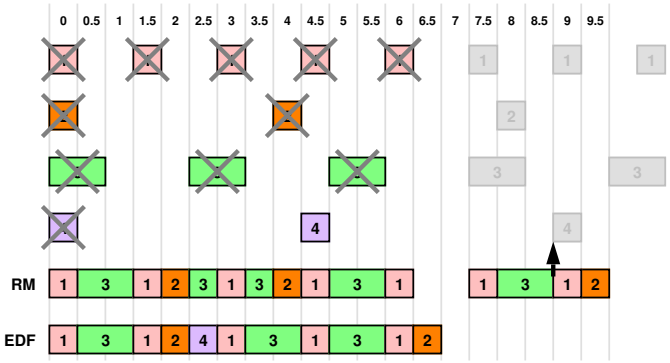
Earliest Deadline First



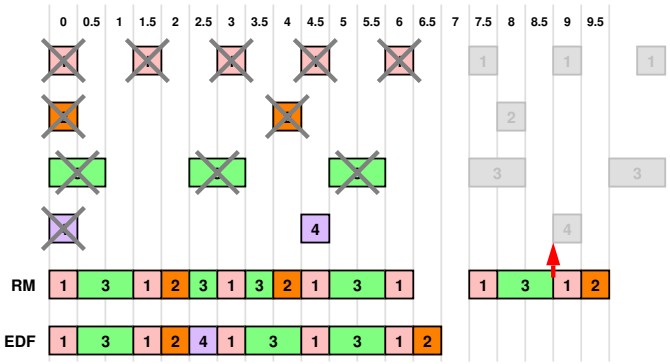
Earliest Deadline First



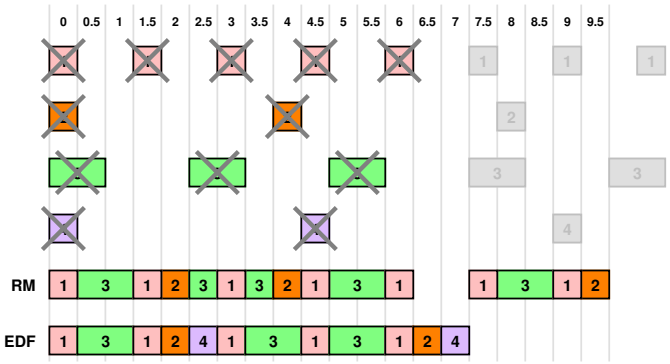
Earliest Deadline First



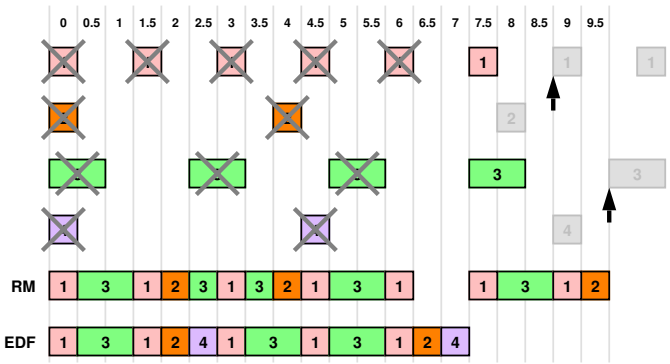
Earliest Deadline First



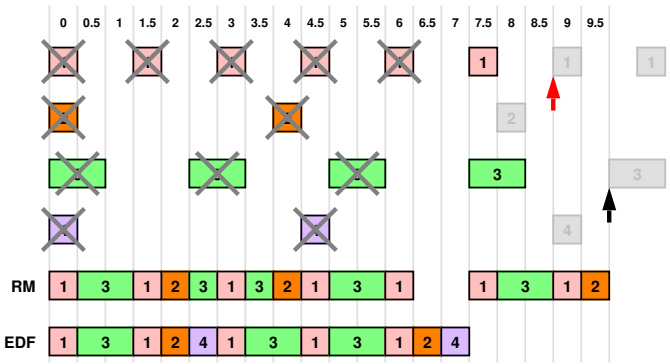
Earliest Deadline First



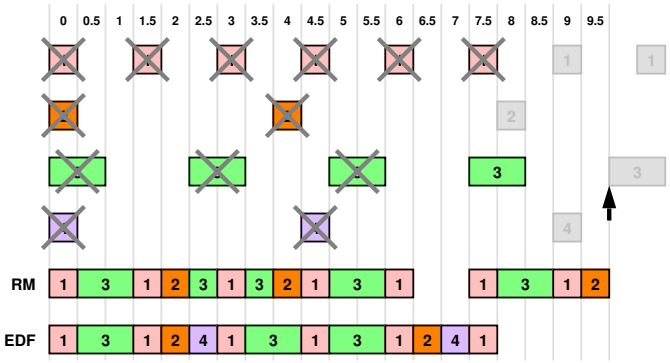
Earliest Deadline First



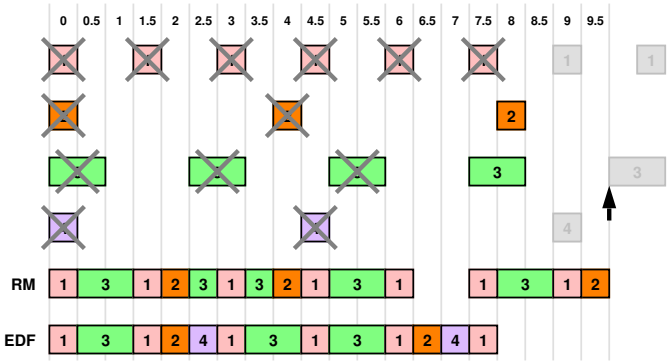
Earliest Deadline First



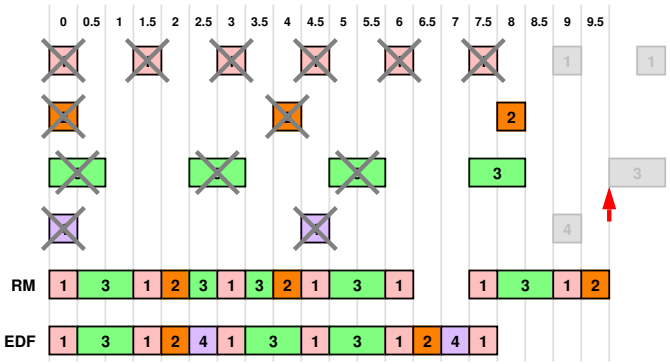
Earliest Deadline First



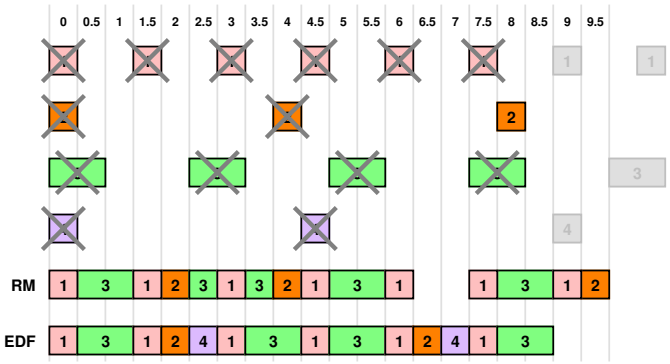
Earliest Deadline First



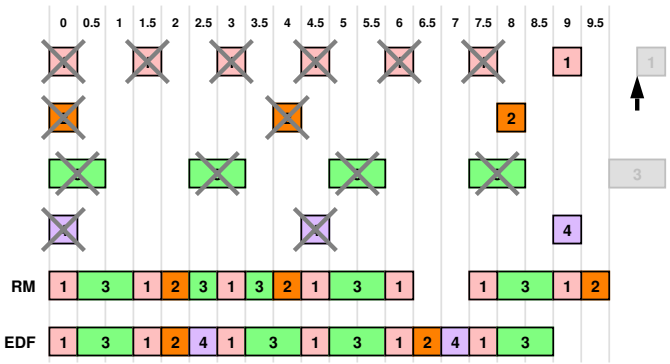
Earliest Deadline First



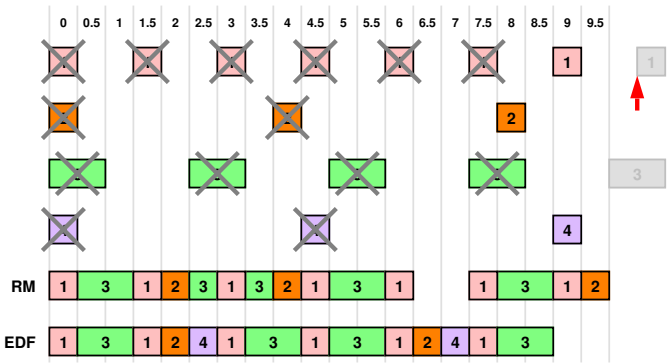
Earliest Deadline First



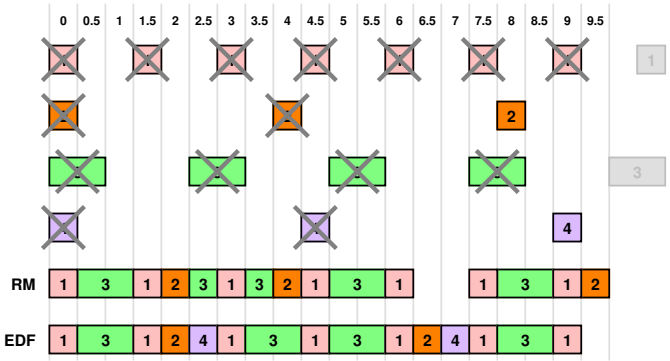
Earliest Deadline First



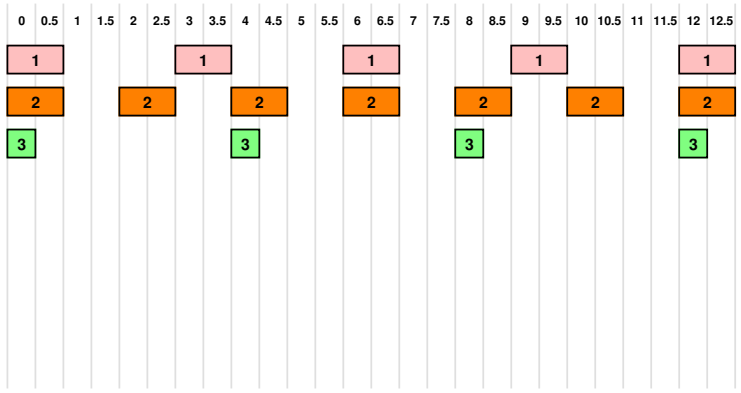
Earliest Deadline First



Earliest Deadline First

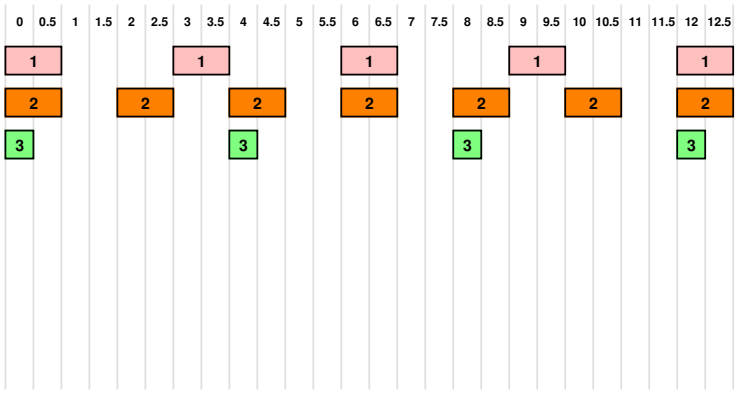


Phase Problems



- if jobs do not start in-phase, rate-monotonic scheduling still may work

Phase Problems



- see "extra slides" at the end

5.3 Scheduling

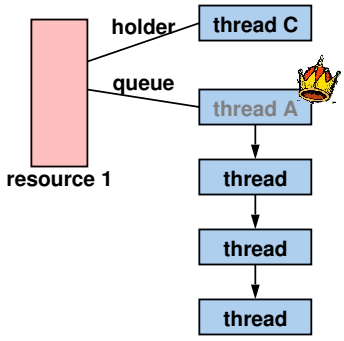
- Goals
- Scheduling Algorithms
- *Implementation Issues*
- Case Studies

Priority Problem

- High-priority thread A blocks on mutex 1
- Low-priority thread C holds mutex 1
- Thread C cannot run because medium-priority thread B is running
- A is effectively waiting at C's priority
  - *priority inversion*

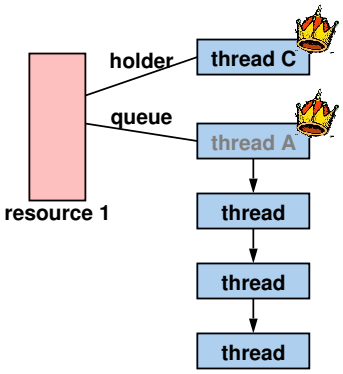
Priority Inheritance

- While A is waiting for resource held by C, it gives C its priority

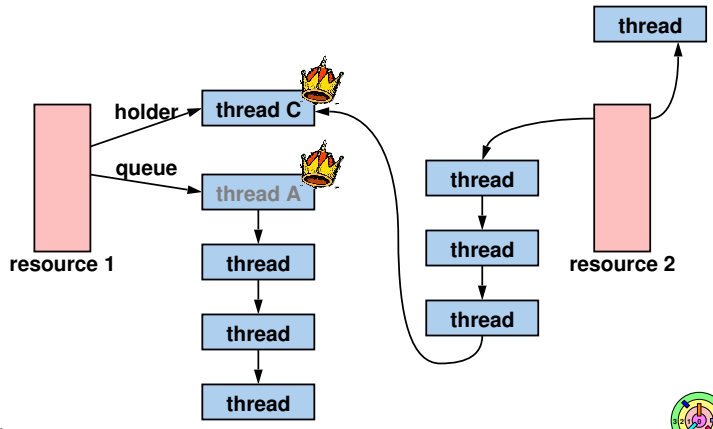


Priority Inheritance

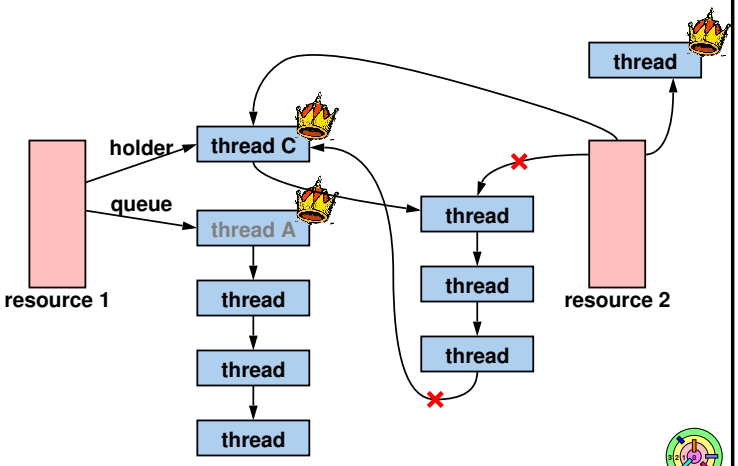
- While A is waiting for resource held by C, it gives C its priority



Cacading Inheritance



Cacading Inheritance

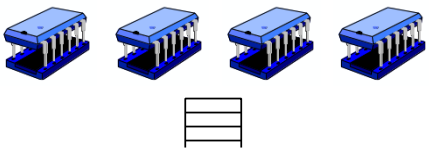


Utilizing Multiple Processors

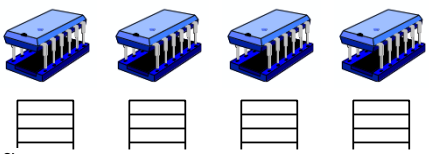
- Dedicate processors to important threads?
- Restrict interrupt handling to certain processors?
- Windows:
  - processor affinity masks
- Solaris:
  - processor sets

Cache Affinity

- After a thread has run on a particular processor, next time it runs, it would be cheaper to run it on the same processor
  - *cache affinity*

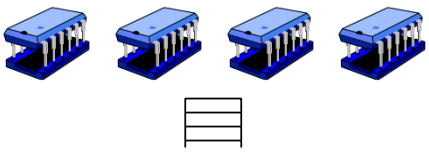


- This means that if you use a shared run queue for multiple processors, you will not be able to take advantage of cache affinity
  - therefore, you should use one run queue per processor

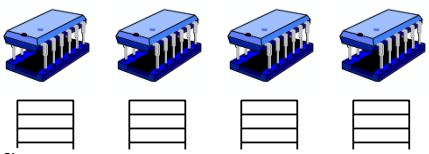


Cache Affinity

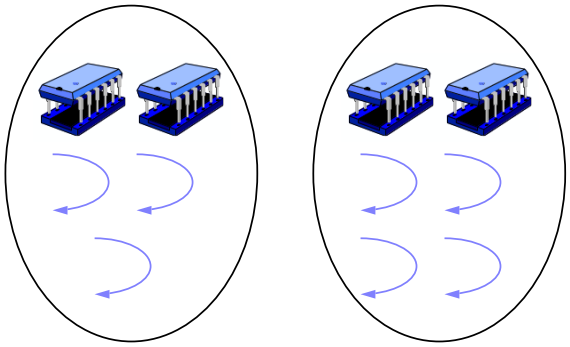
- After a thread has run on a particular processor, next time it runs, it would be cheaper to run it on the same processor
  - *cache affinity*



- This means that if you use a shared run queue for multiple processors, you will not be able to take advantage of cache affinity
  - therefore, you should use one run queue per processor
  - scheduler may do *load balancing* occasionally



Solaris: Processor Sets



- Somewhere between the two extremes

5.3 Scheduling

- Goals
- Scheduling Algorithms
- Implementation Issues
- *Case Studies*

Linux Scheduling

- Policies mandated by POSIX
  - SCHED_FIFO (highest)
    - "real time"
    - *infinite time quantum*
  - SCHED_RR
    - "real time"
    - *adjustable time quantum*
  - SCHED_OTHER
    - "normal" scheduler
    - parameterized allocation of processor time

Linux Scheduler Evolution

- Old scheduler
  - very simple
  - poor scaling
- O(1) scheduler
  - introduced in 2.5
  - less simple
  - better scaling
- Completely fair scheduler *(CFS)*
  - even better
  - simpler in concept
  - much less so in implementation
  - based on *stride* scheduling

Old Scheduler

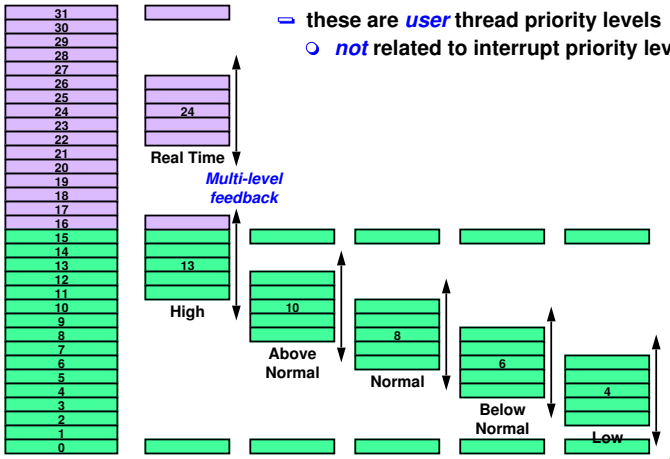
- Four per-process scheduling variables
  - *policy:* which one
  - *rt_priority:* real-time priority
    - 0 for SCHED_OTHER
    - 1 - 99 for others
  - *priority:* time-slice parameter ("nice" value)
  - *counter:* records processor consumption

- see "extra slides" at the end

Scheduling in Windows

- Handling "normal" interactive and compute-bound threads
- Real-time threads
- Multiple processors

Windows Priority Classes and Levels



- these are *user* thread priority levels
  - *not* related to interrupt priority levels

Processor Management Summary

Processor Management

API (thread, fork, exec, ...)
PCB, TCB, Locks
Scheduler
Interrupt Processing

Processor Management Summary

Processor Management

API (thread, fork, exec, ...)
PCB, TCB, Locks
Scheduler
Interrupt Processing

- Try visualizing what happens when you do "ls" in the console

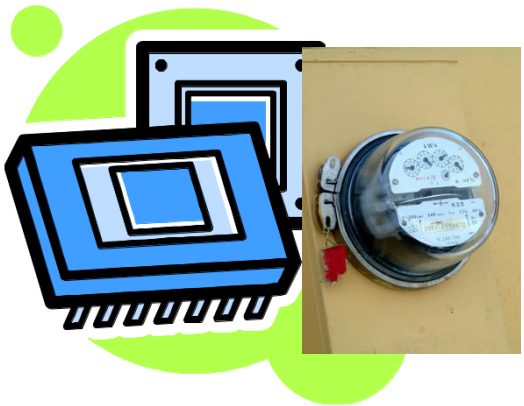
File Systems Overview

File Systems (VFS)

API (open, close, ...)
File Descriptor Tables, System File Tables
Directories, Permissions
Actual File System

Extra Slides

Metered Processors



- see "extra slides" at the end

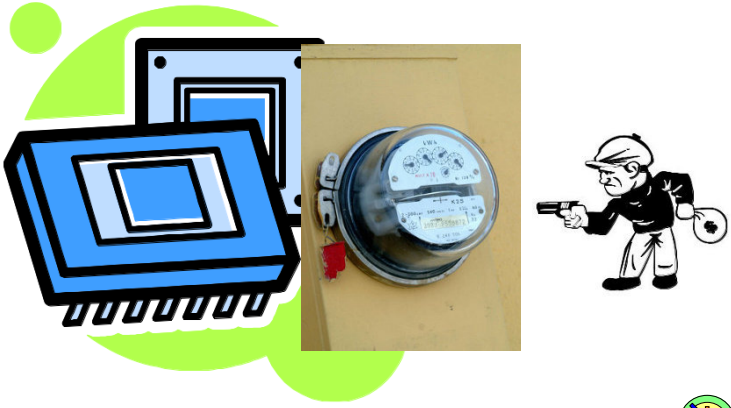
Algorithm

- Each thread has a meter, which runs only when the thread is running on the processor
- At every clock tick
  - give processor to thread that's had the least processor time as shown on its meter
  - in case of tie, thread with lowest ID wins

Algorithm

- Some threads may be more important than others
- What if new threads enter system?
- What if threads block for I/O and synchronization?

Metered Processors (Mafia Variation)



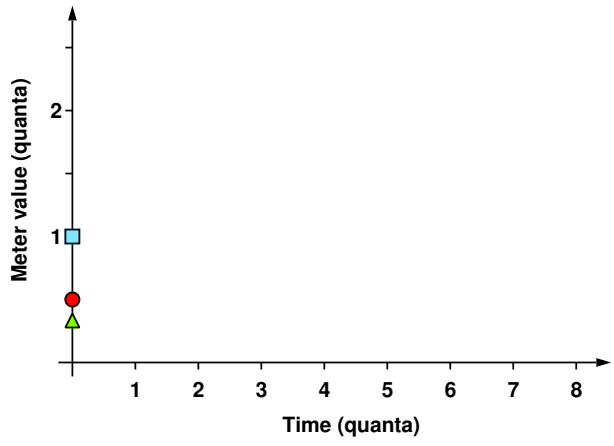
Details ...

- Each thread pays a bribe
  - the greater the bribe, the slower the meter runs
  - to simplify bribing, you buy "tickets"
  - one ticket is required to get a fair meter
  - two tickets get a meter running at half speed
  - three tickets get a meter running at 1/3 speed
  - etc.

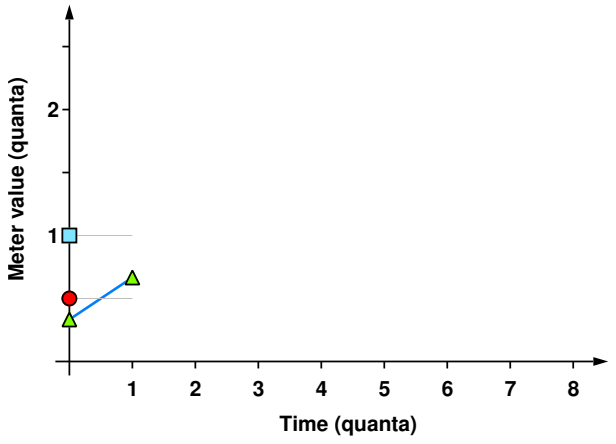
New Algorithm

- Each thread has a (possibly crooked) meter, which runs only when the thread is running on the processor
- At every clock tick
  - give processor to thread that's had the least processor time as shown on its meter
  - in case of tie, thread with lowest ID wins

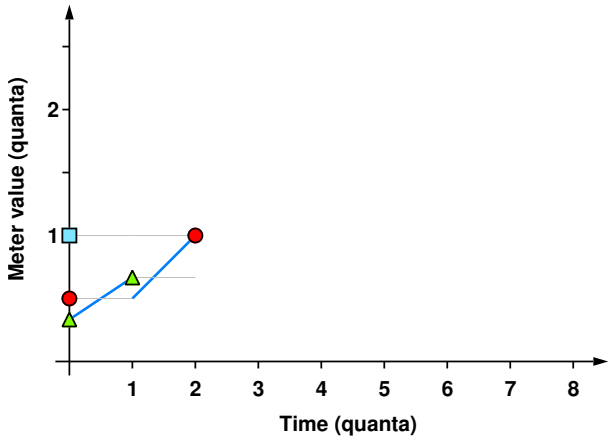
Example



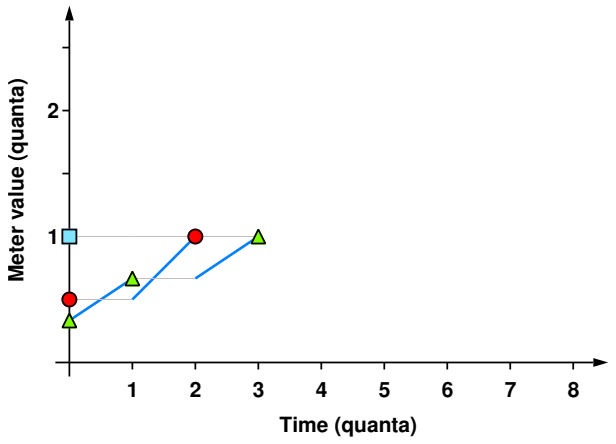
Example



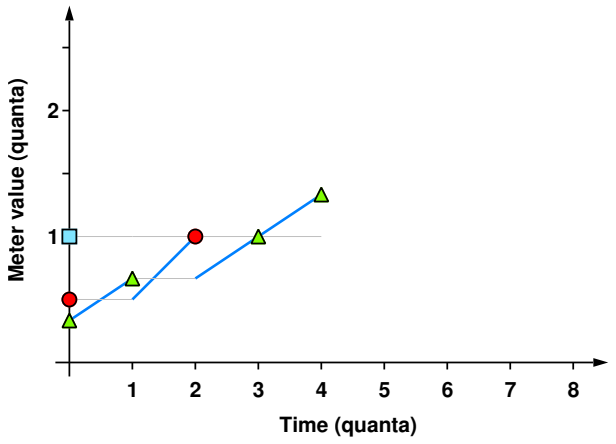
Example



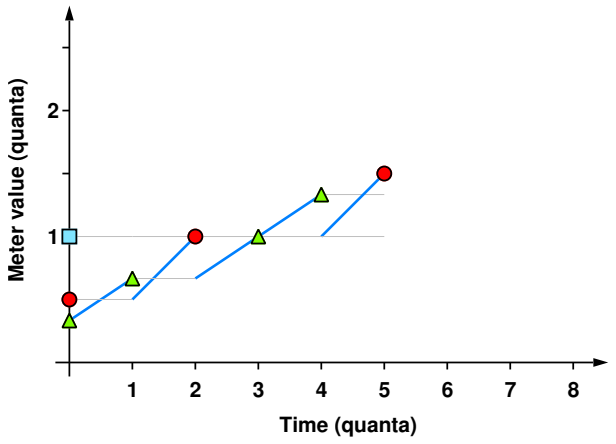
Example



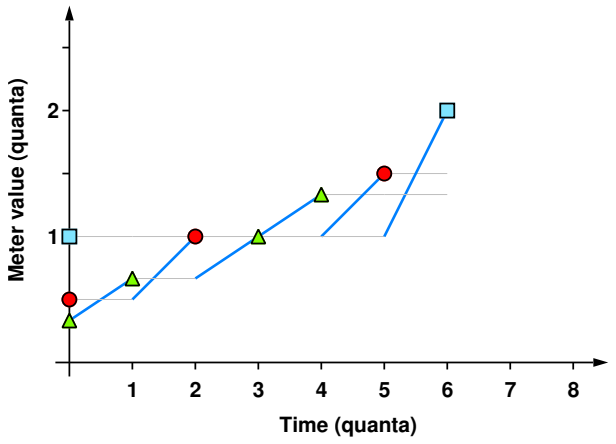
Example



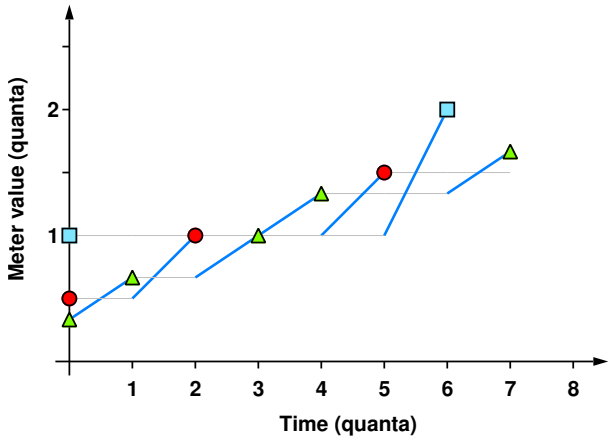
Example



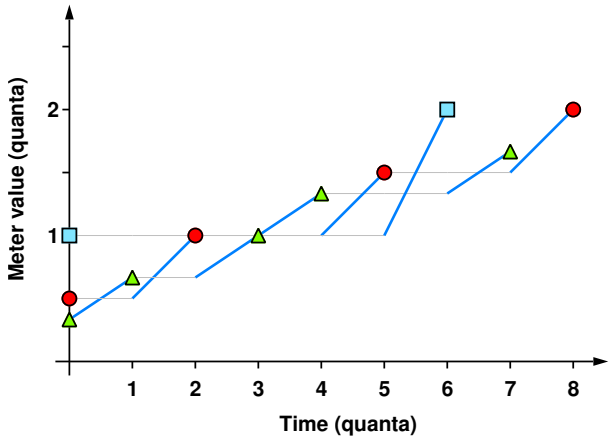
Example



Example

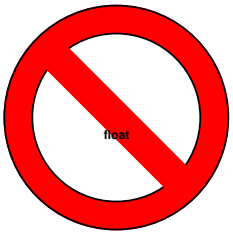


Example



More Details

```
typedef struct {
  float bribe, meter_rate, metered_time;
} thread_t;

void thread_init(thread_t *t, float bribe) {
  if (bribe < 1)
    abort();
  t->bribe = bribe;
  t->meter_rate = t->metered_time = 1.0/bribe;
  InsertQueue(t);
}
```

More Details (revised)

```
typedef struct {
  long long bribe, meter_rate, metered_time;
} thread_t;

const long long BigInt = 2^20;

void thread_init(thread_t *t, long bribe) {
  if (bribe < 1)
    abort();
  t->bribe = bribe;
  t->meter_rate = t->metered_time = BigInt/bribe;
}
```


More Details (continued)

```
void OnClockTick() {
  thread_t *NextThread;

  CurrentThread->metered_time +=
      CurrentThread->meter_rate;
  InsertQueue(CurrentThread);
  NextThread = PullSmallestThreadFromQueue();
  if (NextThread != CurrentThread)
    SwitchTo(NextThread);
}
```

Handling New Threads

- It's time to get an accountant ...
  - keep track of total bribes
  - TotalBribe = total number of tickets in use
  - keep track of actual (normalized) processor time: `TotalTime`
  - measured by a "fixed" meter going at the rate of 1/TotalBribe
    - BigInt/TotalBribe when we convert from floating point
- New thread
  - pays bribe, gets meter
  - `metered_time` initialized to `TotalTime+meter_rate`

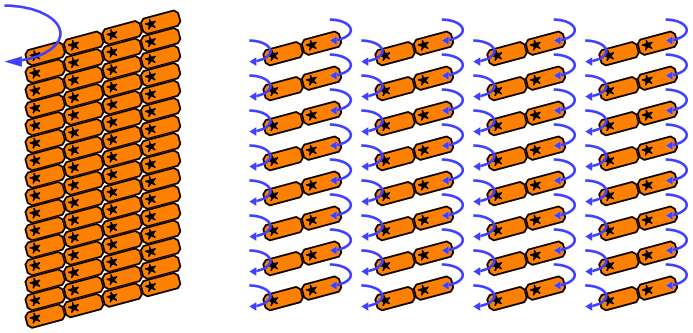
Revised Details

```
void OnClockTick() {
  thread_t *NextThread;

  TotalTime += BigInt/TotalBribe;
  CurrentThread->metered_time +=
      CurrentThread->meter_rate;
  InsertQueue(CurrentThread);
  NextThread =
      PullSmallestThreadFromQueue();
  if (NextThread != CurrentThread)
    SwitchTo(NextThread);
}
```

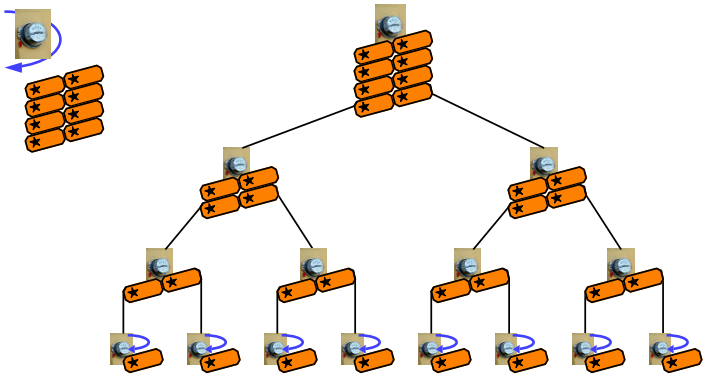
Thread Leaves, then Returns

```
void ThreadDepart(thread_t *t) {
  t->remaining_time =
      t->metered_time - TotalTime;
  // remaining_time is a new component
}

void ThreadReturn(thread_t *t) {
  t->metered_time =
      TotalTime + t->remaining_time;
}
```

A Mismatch



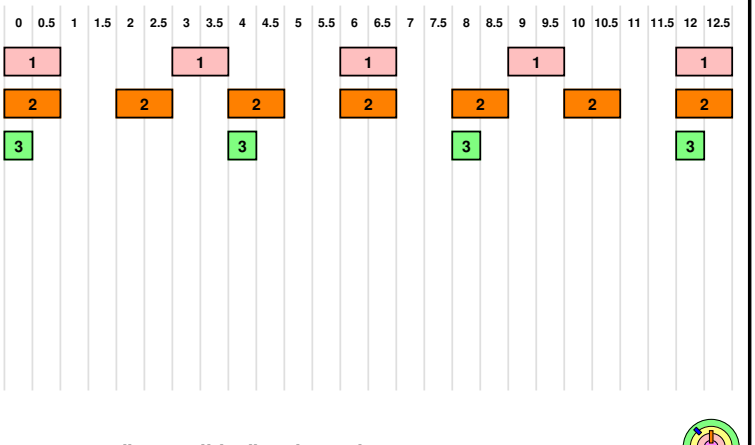
Hierarchical Stride Scheduling



Making It Work: How Many Tickets?

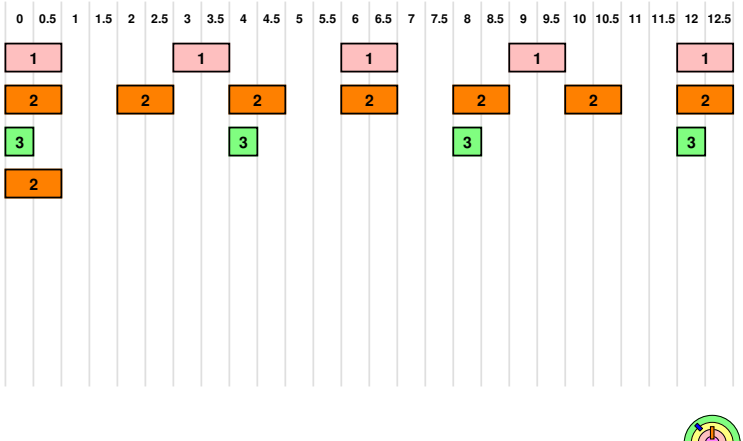
- MP3 player requires 1/4 of processor time
  - 2 tickets
  - guaranteed 1/4 of processor time
- Streaming video player requires 3/8 of processor time
  - 3 tickets
  - guaranteed 3/8 of processor time
- OS project can use all of processor time, but usually much less
  - 3 tickets
  - guaranteed 3/8 of processor time

Phase Problems

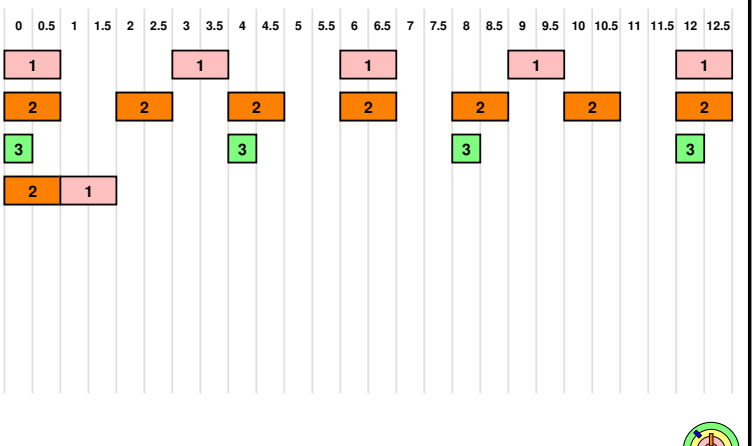


see "extra slides" at the end

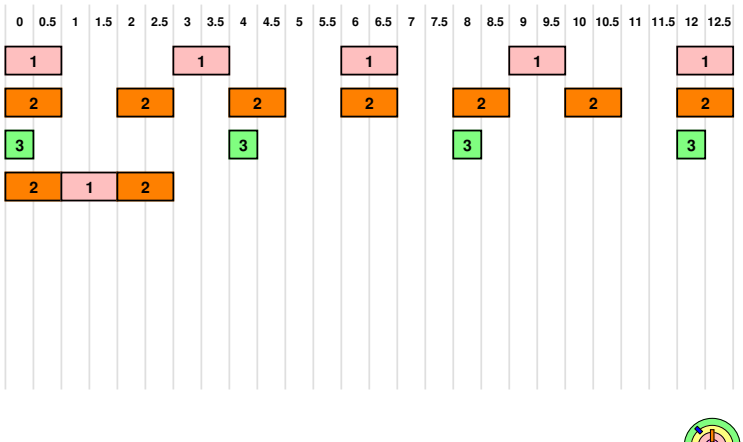
Phase Problems



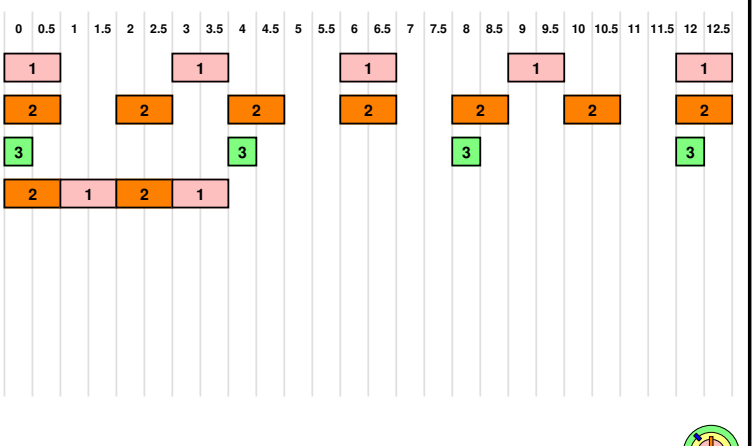
Phase Problems



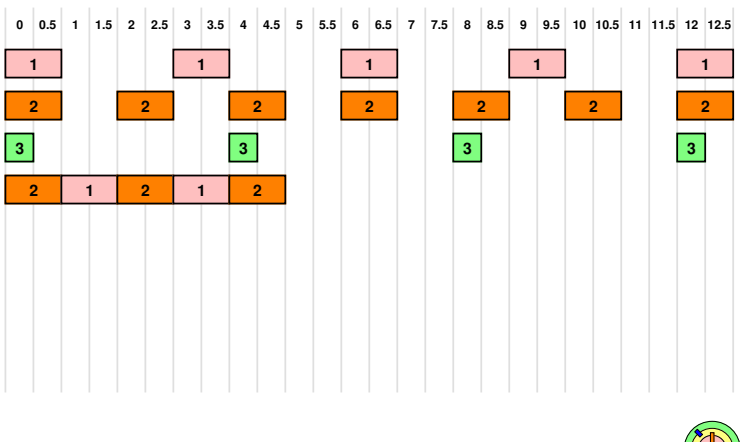
Phase Problems



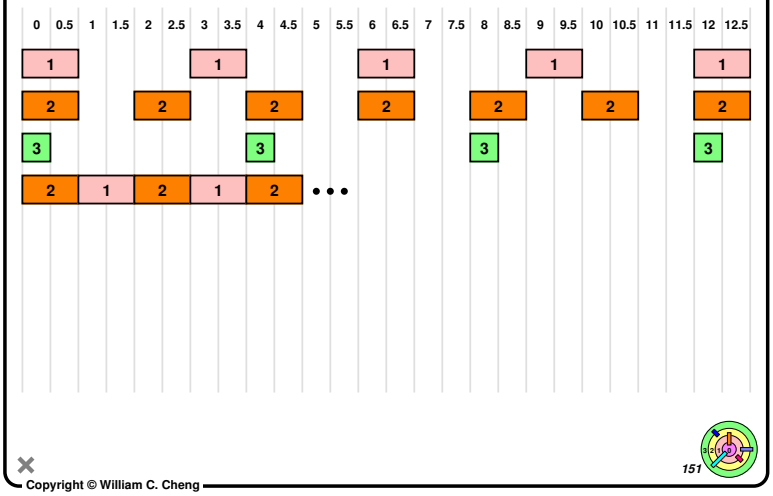
Phase Problems



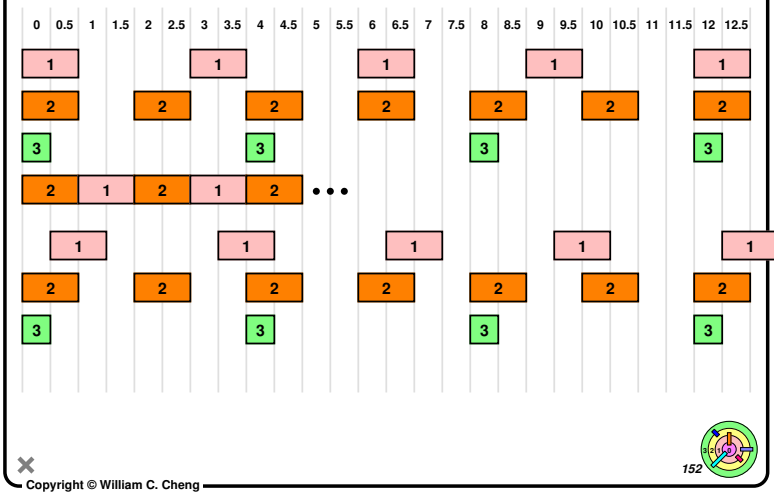
Phase Problems



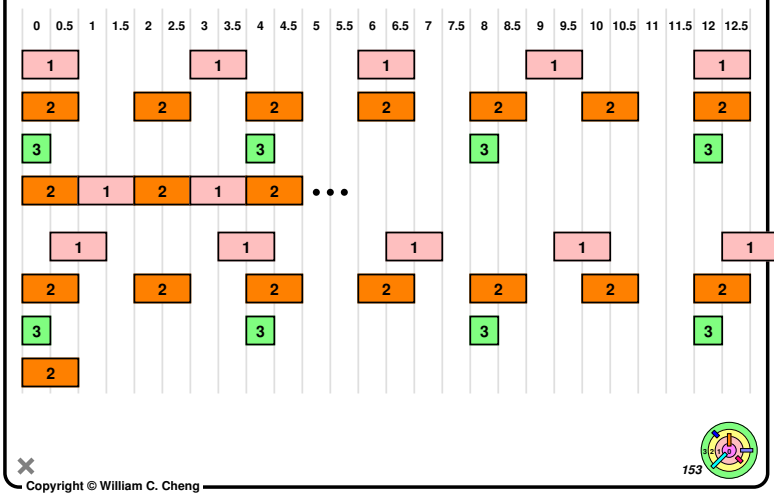
Phase Problems



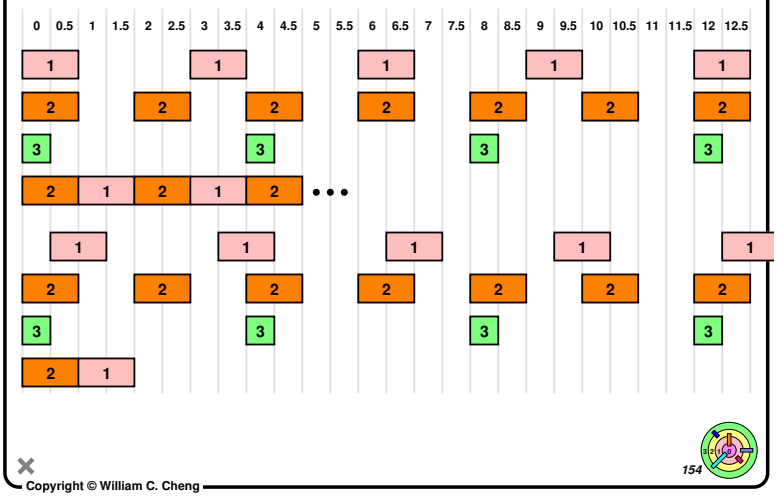
Phase Problems



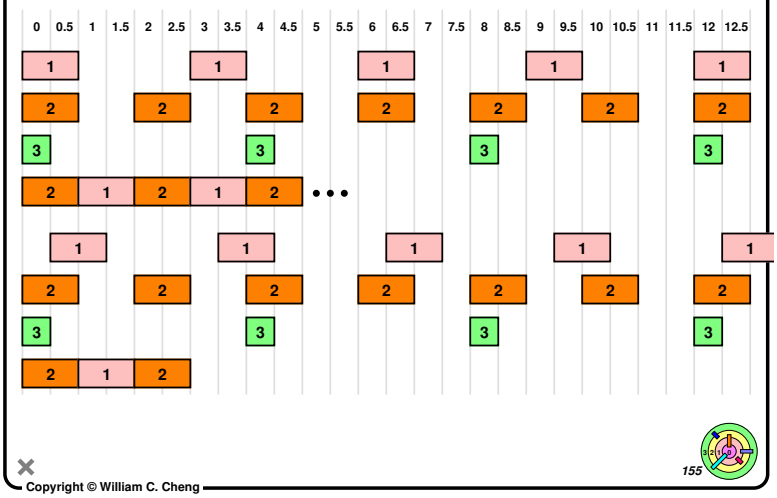
Phase Problems



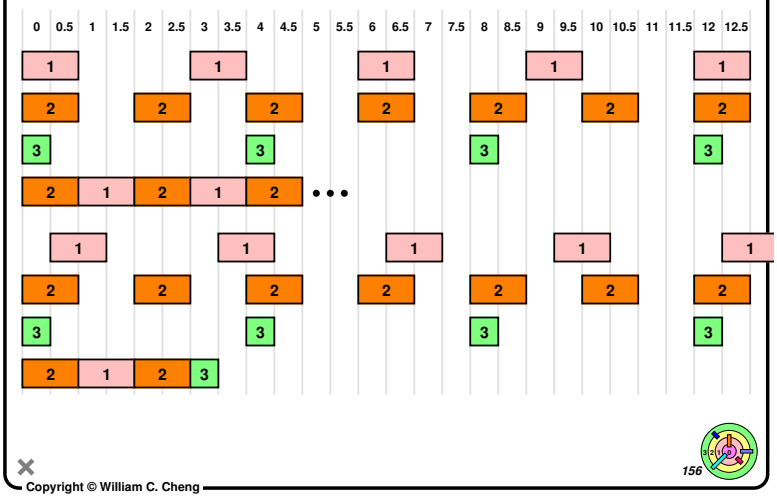
Phase Problems



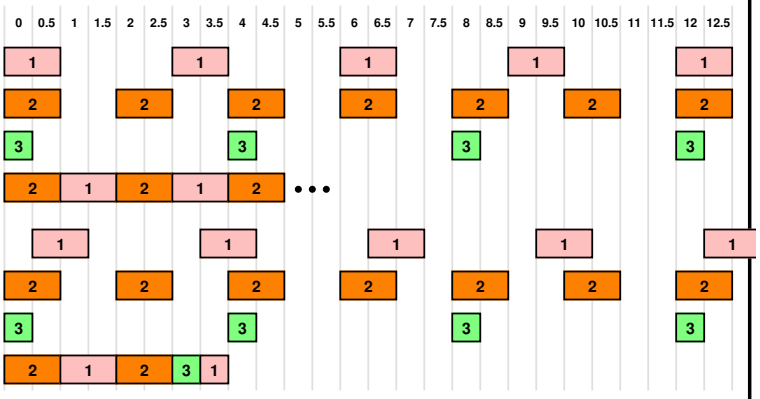
Phase Problems



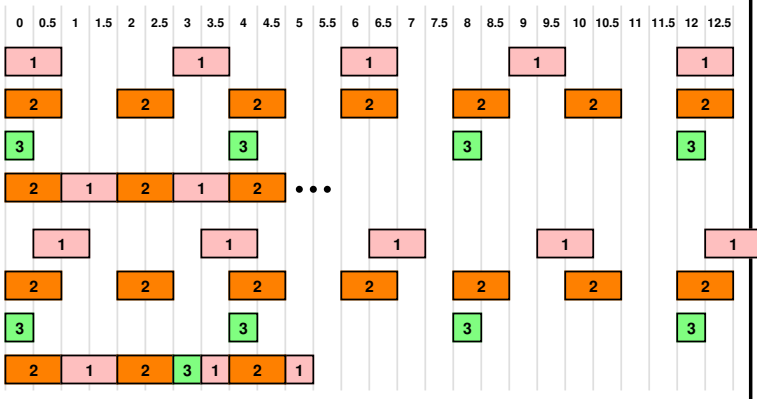
Phase Problems



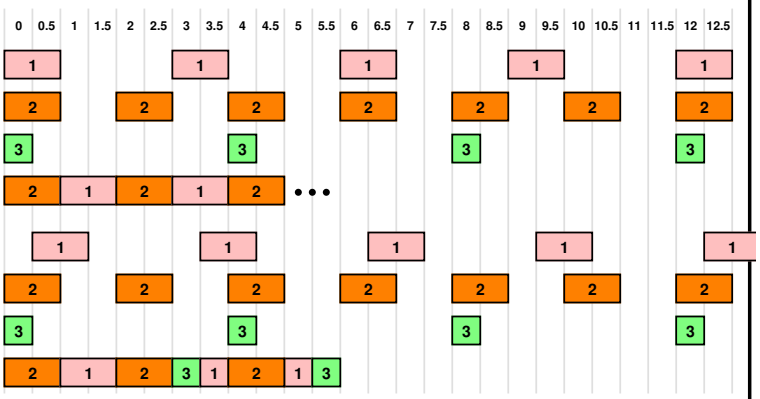
Phase Problems



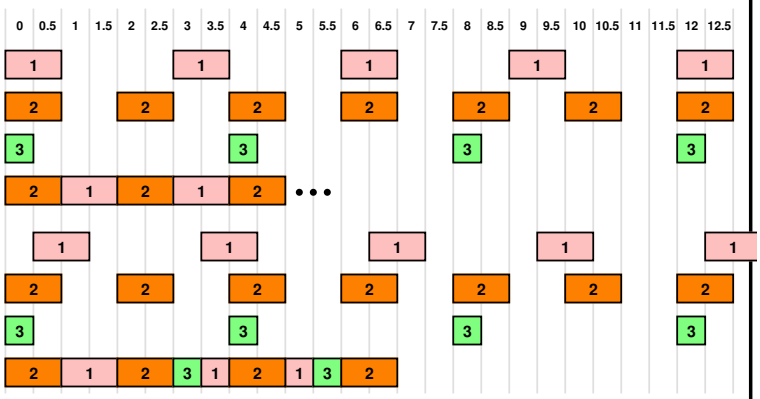
Phase Problems



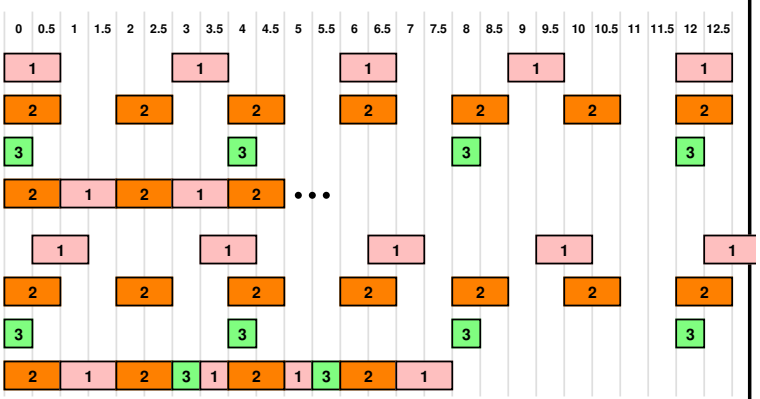
Phase Problems



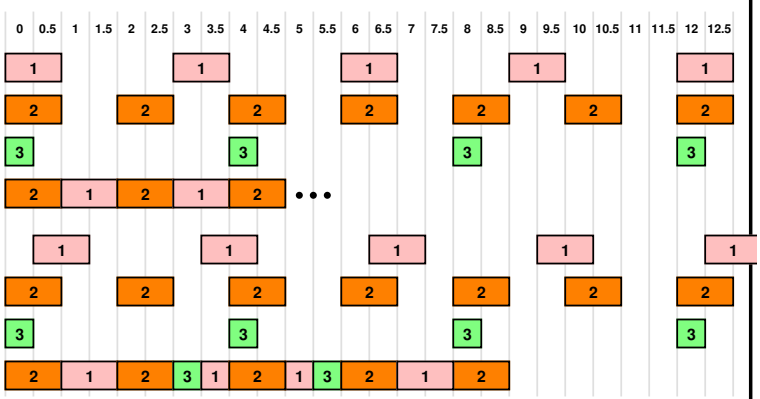
Phase Problems



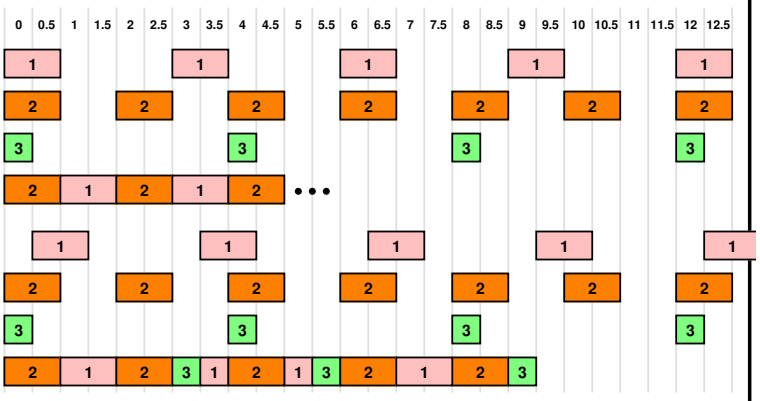
Phase Problems



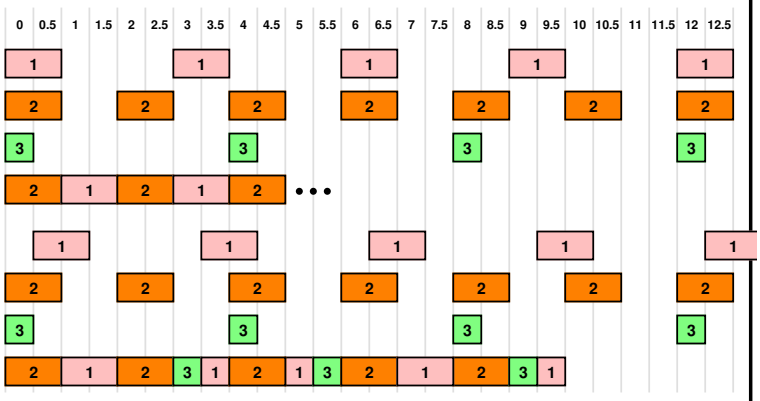
Phase Problems



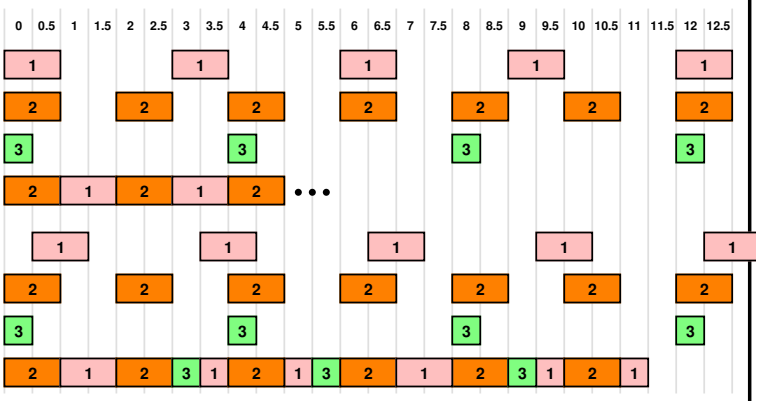
Phase Problems



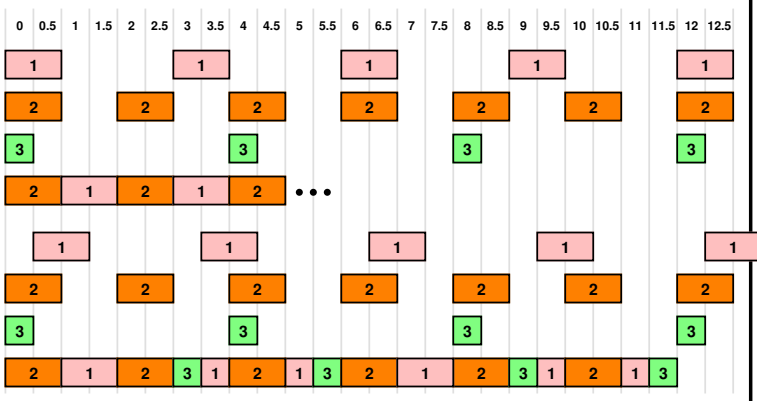
Phase Problems



Phase Problems



Phase Problems



Old Scheduler

- Four per-process scheduling variables
  - *policy:* which one
  - *rt_priority:* real-time priority
    - 0 for SCHED_OTHER
    - 1 - 99 for others
  - *priority:* time-slice parameter ("nice" value)
  - *counter:* records processor consumption

- see "extra slides" at the end

Old Scheduler: Time Slicing

- Clock "ticks" HZ times per second
  - interrupt/tick
- Per-process *counter*
  - current process's is decremented by one each tick
  - time slice over when counter reaches 0

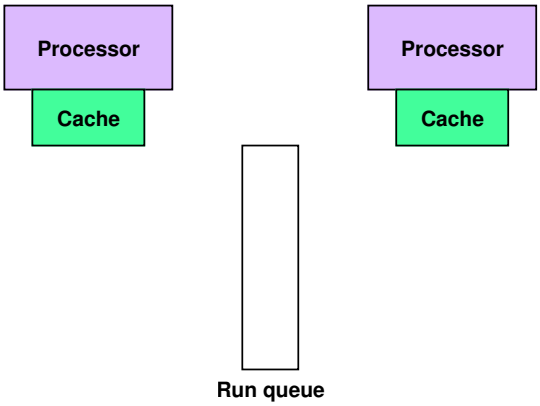
Old Scheduler: Throughput

- Scheduling cycle
  - length, in "ticks", is sum of priorities
  - each process gets *priority* ticks/cycle
    - *counter* set to *priority*
    - cycle over when *counters* for runnable processes are all 0
  - sleeping processes get "boost" at wakeup
    - at beginning of each cycle, for each process:
      - counter = counter/2 + priority

Old Scheduler: Who's Next?

- Run queue searched beginning to end
  - new arrivals go to front
  - SCHED_RR processes go to end at completion of time slices
- Next running process is first process with highest "goodness"
  - 1000 + rt_priority for SCHED_FIFO and SCHED_RR processes
  - *counter* for SCHED_OTHER processes

Diagram



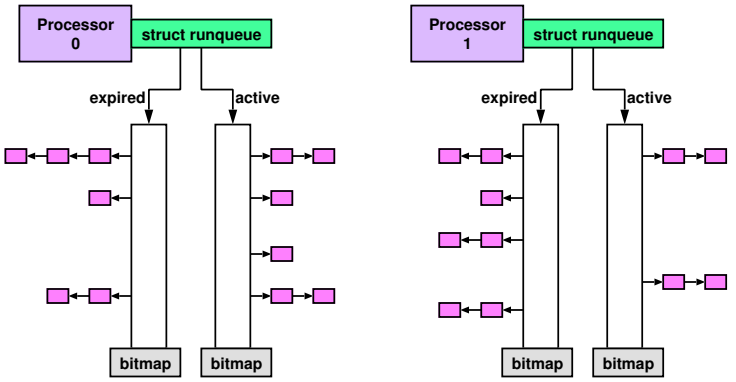
Old Scheduler: Problems

- O(n) execution
- Poor interactive performance with heavy loads
- SMP contention for run-queue lock
- SMP affinity
  - cache "footprint"
- How well does it handle the real-life example?

O(1) Scheduler

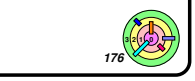
- All concerns of old scheduler plus:
  - efficient, scalable execution
  - identify and favor interactive processes
  - good SMP performance
    - minimal lock overhead
    - processor affinity

O(1) Scheduler: Data Structures

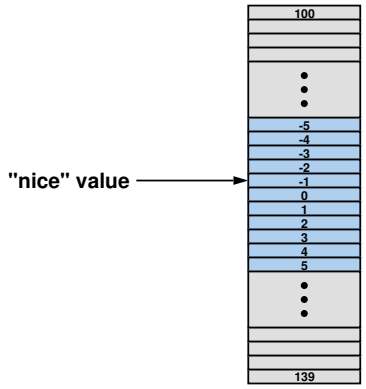


O(1) Scheduler: Queues

- Two queues per processor
  - active: processes with remaining time slice
  - expired: processes with no more time slice
  - each queue is an array of lists of processes of the same priority
    - bitmap indicates which priorities have processes
  - processors scheduled from private queues



O(1) Scheduler: Priorities



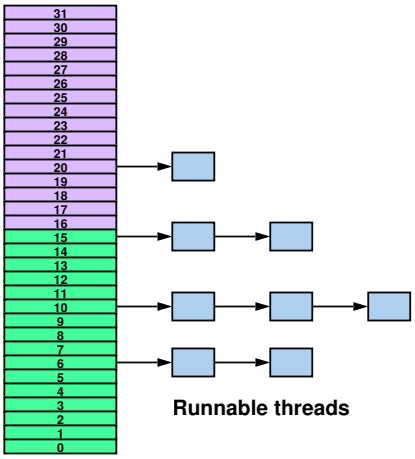
O(1) Scheduler: Actions

- Process switch
  - pick best priority from active queue
    - if empty, switch active and expired
  - new process's time slice is function of its priority
- Wake up
  - priority is boosted or dropped depending on sleep time
  - higher priority processes get longer time quanta
- Time-slice expiration
  - interactive processes rejoin active queue
    - unless processes have been on expired queue too long

O(1) Scheduler: Load Balancing

- Processors with empty queues steal from busiest processor
  - checked every millisecond
- Processors with relatively small queues also steal from busiest processor
  - checked every 250 milliseconds

Uniprocessor Windows



Improving Real Time

- Multimedia applications need 80% of processor time
- Make sure normal applications get at least 20%
- How?
- Windows solution: MMCSS
  - multimedia class scheduler service
  - dynamically manage multimedia threads
    - run at real-time priority 80% of time
    - run at normal priority 20% of time

Which Processor?

- Newly created thread assigned *ideal processor*
  - randomly chosen
- May also set *affinity mask*
  - may be scheduled only on processors in mask
- Scheduling decision:
  - if idle processors available
    - first preference: ideal processor
    - second preference: most recent processor
  - otherwise
    - joins run queue of ideal processor

Some Details ...