Scheduling
(continued)

New (Stride Scheduling) Algorithm

- Each thread has a *(possibly crooked)* meter that runs only when the thread is running on the processor
- Each thread's meter is initialized as 1/bribe
- 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

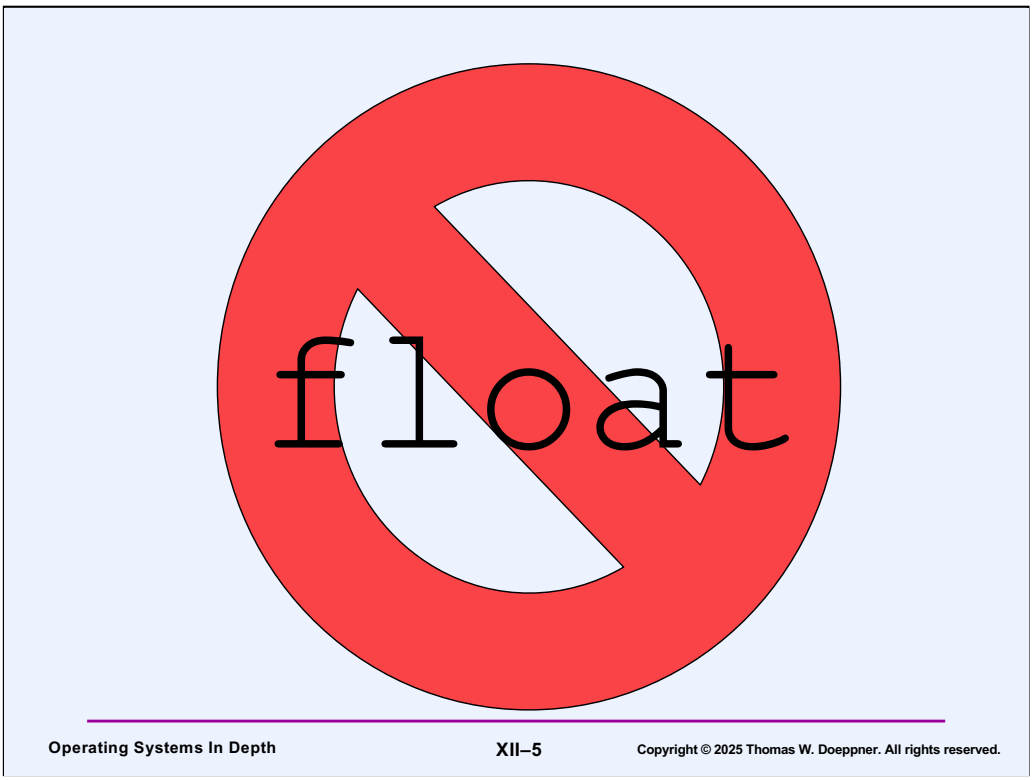
- Three threads
  - T_1 has one ticket: meter_rate = 1
  - T_2 has two tickets: meter_rate = 1/2
  - T_3 has three tickets: meter_rate = 1/3
  - six total tickets
- After 6 clock ticks
  - T_1's meter incremented by 1 once
  - T_2's meter incremented by 1/2 twice
  - T_3's meter incremented by 1/3 three times
- Each meter shows increase of 1
  - what "1" means depends on the bribe

Assuming six total tickets, after six clock ticks each thread sees its meter being increased by 1. Thus, after each cycle, all threads have the same values shown on their meters, but what these values mean in terms of time is proportional to the number of tickets they've purchased.

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/bribe;
  InsertQueue(t);
}
```


InsertQueue is a function that places the thread on the run queue. Let's assume that the run queue is implemented as a data structure that allows one to perform operations such as inserting a thread and extracting the thread with the smallest metered time in O(log n) time. In CFS (in Linux), this queue is implemented as a red-black tree.



The code in the preceding slide used floating-point arithmetic. Because of precision problems and the fact that floating-point arithmetic might be considerably slower than fixed-point arithmetic, we don't want to use it. Instead, we'll do everything with scaled fixed-point arithmetic, as shown in the next slide.

More Details (revised)

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

const long BigInt = 2^50;

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


So that we don't have to use floating point, we use scaled fixed-point arithmetic, as shown in the slide. So, rather than set meters to 1/bribe, they're set to BigInt/bribe – thus the values are scaled by BigInt.

So as to make the rest of our discussion easier to follow, we'll stick with floating point.

More Details (continued)

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

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


On each clock tick, we adjust the current thread's **metered_time** to account for the processor time it just used, adjusted according to its bribe. If this thread is no longer the thread that has used the least (bribed) processor time, we switch to the thread that has the least processor time.

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 total (normalized) processor time: *TotalTime*
    - measured by a "fixed" meter going at the rate of 1/TotalBribe
- New thread
  1) pays bribe, gets meter
  2) `metered_time` initialized to `TotalTime+meter_rate`

To handle the addition of a new thread, we must initialize its **metered_time** to an appropriate value. The idea is to set its **metered_time** to what it would have been if the thread had been running since the beginning of time. Note that, despite the crookedness, all threads running since the beginning of time will have the same value for **metered_time** give or take their **meter_rate** — this is the whole point of the algorithm.

Consider an additional meter on the system that runs at the rate **1/TotalBribe**, and whose value we call **TotalTime**. Thus at every clock interrupt its value is incremented by **1/TotalBribe** — its value goes up by one when the meters of all the active threads have each gone up by one. And thus this meter provides a tight lower bound on the values of all the meters of threads that have been running since the beginning of time.

So, when we add a new thread to the system, we initialize its meter with the value **TotalTime** plus the new thread's **meter_rate**, so that it competes with the other threads for processor time as if it had been competing since the beginning of time.

Example (continued)

- Three threads
  - T_1 has one ticket: meter_rate = 1
  - T_2 has two tickets: meter_rate = 1/2
  - T_3 has three tickets: meter_rate = 1/3
  - TotalBribe = 6
- Assume one clock interrupt/second
  - at every interrupt: TotalTime += 1/6
- After 6 seconds
  - T_1's meter incremented by 1 once
  - T_2's meter incremented by 1/2 twice
  - T_3's meter incremented by 1/3 three times
  - TotalTime incremented by 1/6 six times

TotalTime thus can be used to show what would be on a thread's meter if it has been running since the beginning of time. Note that it's not strictly necessary to maintain it separately, since its value will be what's on each thread's meter (after subtracting off the meter rate) at the end of each cycle.

What's Going On ...

- Assume T clock interrupts/second
  - every TotalBribe seconds
    - TotalTime incremented by T
    - each thread's metered_time incremented by T
- TotalTime · TotalBribe
  = actual total processor time
- metered_time · bribe
  = actual time used by thread
- Initialize meter so it appears as if new thread has been getting its share of processor time since beginning of time

Recall that after TotalBribe clock ticks, each thread's meter will be incremented by one. Thus, at T clock interrupts/second, after TotalBribe seconds (and thus T*TotalBribe clock ticks), each thread's meter is incremented by T.

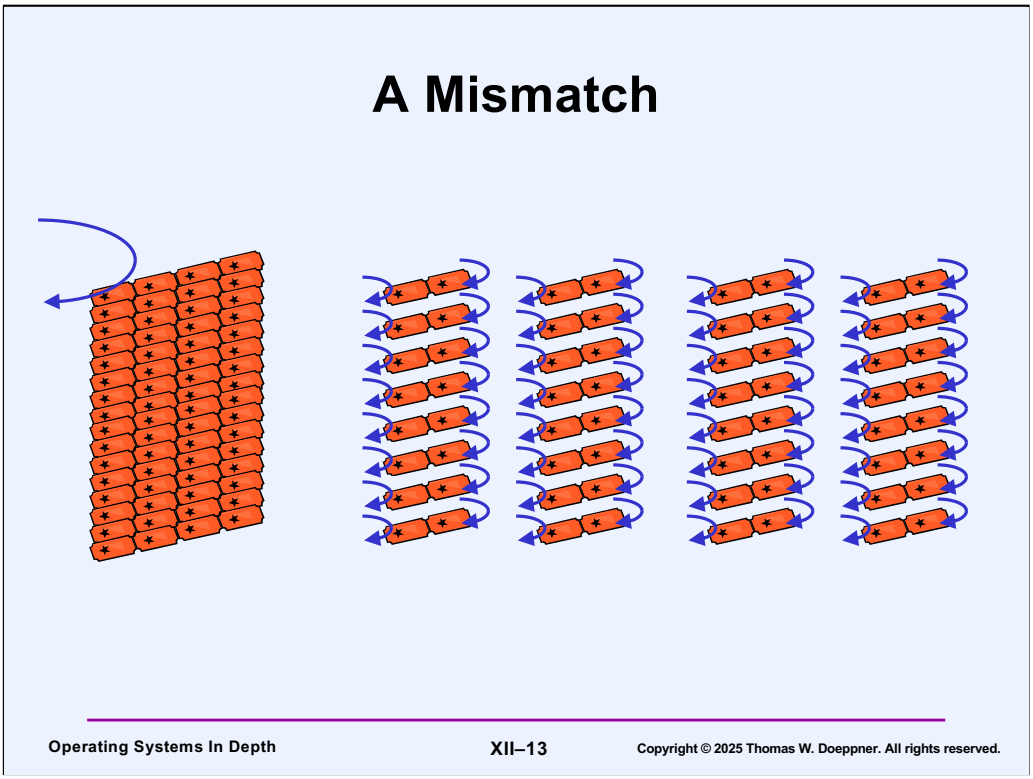
Revised Details

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

  TotalTime += 1/TotalBribe;
  CurrentThread->metered_time +=
      CurrentThread->meter_rate;
  InsertQueue(CurrentThread);
  NextThread =
      PullSmallestThreadFromQueue();
  if (NextThread != CurrentThread)
    SwitchTo(NextThread);
}
```

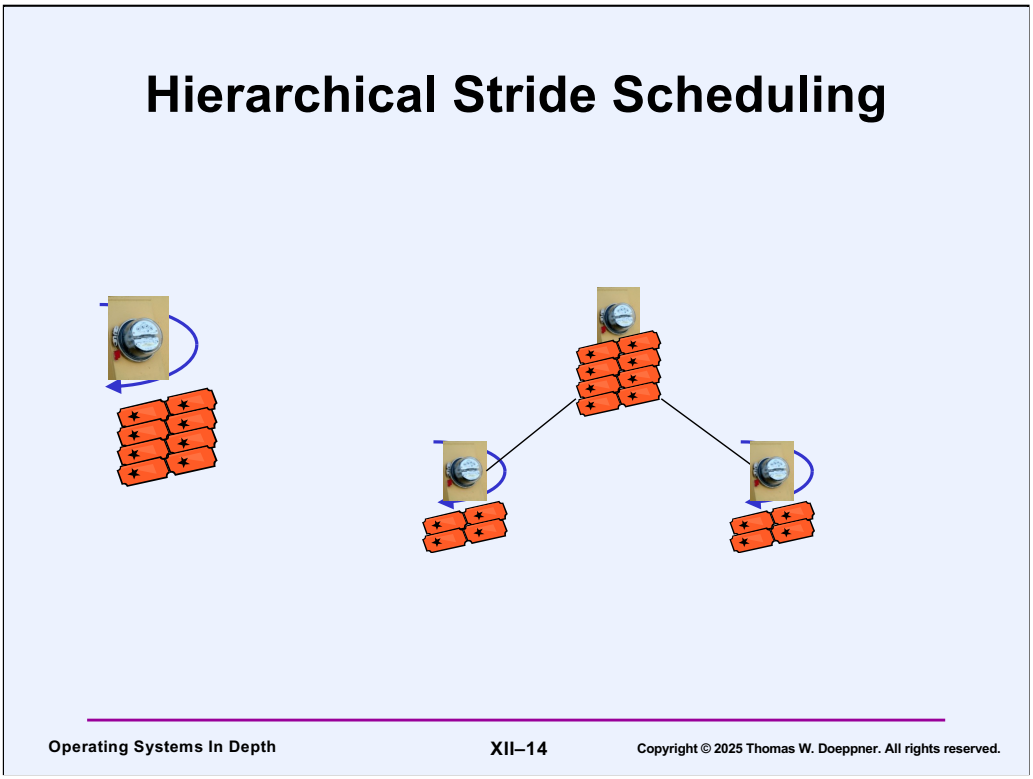
Thread Leaves, then Returns

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

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


When a thread becomes non-runnable, perhaps because it's waiting for I/O, we subtract its tickets from **TotalBribe**, since it's no longer contending for processor time. Thus **TotalTime** is now incremented at a somewhat faster rater, since fewer threads are competing for the processor. We also keep track, in **remaining_time**, of how much greater the thread's **metered_time** was than **TotalTime** when it became nonrunnable. When the thread becomes runnable again, we'd like to set its meter to the value it would have had if it had been runnable all this time. This turns out to be the value of **TotalTime** when it becomes runnable, plus its saved **remaining_time**.



Here we have one thread with 64 tickets and 64 threads each with one ticket. What the stride scheduler will do with them is to first run the thread with 64 tickets for 64 clock ticks, then run each of the other threads for one clock tick each. Perhaps this is what is desired, but what might be better is for the 64-ticket thread to run for every other time quantum over the next 128 clock ticks, alternating with the other threads, each of which executes once during the 128 clock ticks.



Here's a possible extension of stride scheduling, shown with an example in which we have one single-threaded process and one two-threaded process. The single-threaded process, on the left, has one meter and has purchased eight tickets. The two-threaded process has purchased eight tickets as well, but it has divvyed them out so that each thread gets four. This is represented by giving the process, as a whole, its own meter, but also giving each thread a meter.

The scheduler selects the process that has the least time on its meter. If it selects the single-threaded process, then that process's thread runs for a clock tick and its meter is incremented by 1/8. If the scheduler selects the two-threaded process, it then chooses the thread of the process that has the least time on its meter. After a clock tick expires, 1/4 is added to the thread's meter and 1/8 is added to the process's meter. Thus the scheduler will alternate between the two processes, but each time it chooses the two-threaded process, it will alternate between threads.

Quiz 1

Recall our discussion about handling interactive jobs in early Unix. The later BSD scheduler would give priority to threads that were mainly idle, thus favoring interactive jobs.

Does stride scheduling favor such interactive jobs? I.e., if a waiting thread suddenly receives input, will it get to run sooner than other runnable threads of the same priority?

a) yes

b) no

Latency

- Some threads may require low latency
  - should run very soon after becoming runnable
  - could give them higher priorities
    - not good if the thread runs for a long time
  - a possibility: give such threads lower latency in exchange for running for shorter periods of time

EEVDF (1)

- **"Earliest Eligible Virtual Deadline First"**
- **Assume all threads have same priority**
- **If there are *n* runnable threads, each gets a (1/n)-second execution slot each second**

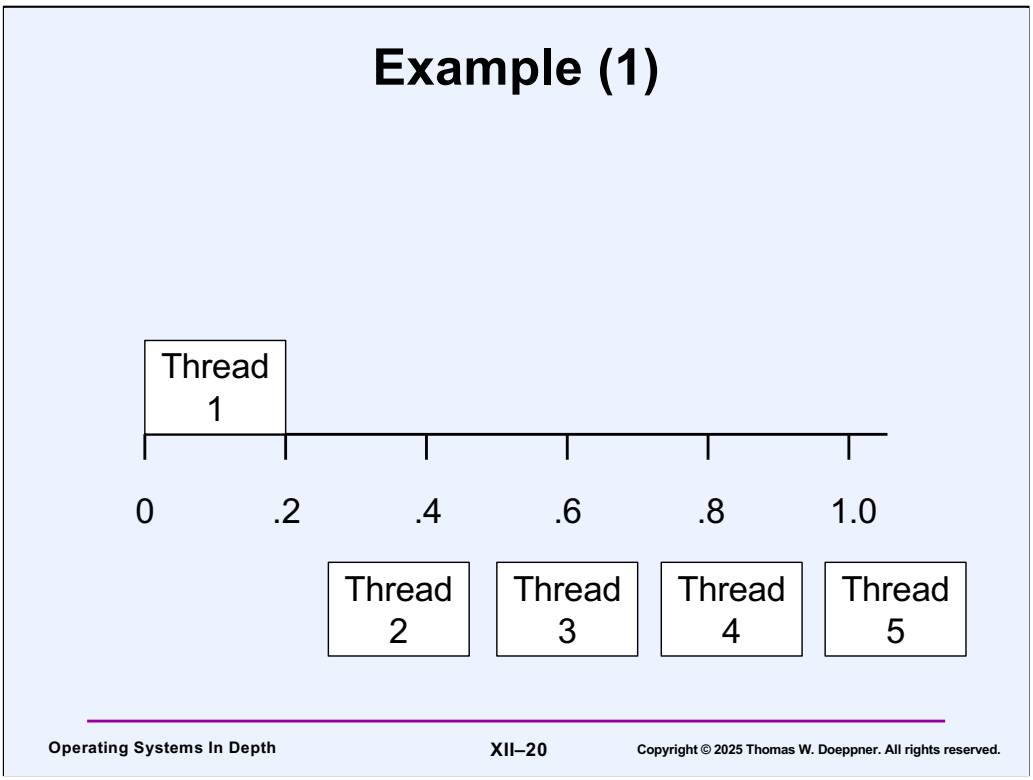
A description of EEVDF can be found at https://lwn.net/Articles/925371/. A scheduler based on these ideas replaced CFS in Linux version 6.6 (August 2023), but not all distributions of Linux are using such versions.

EEVDF (2)

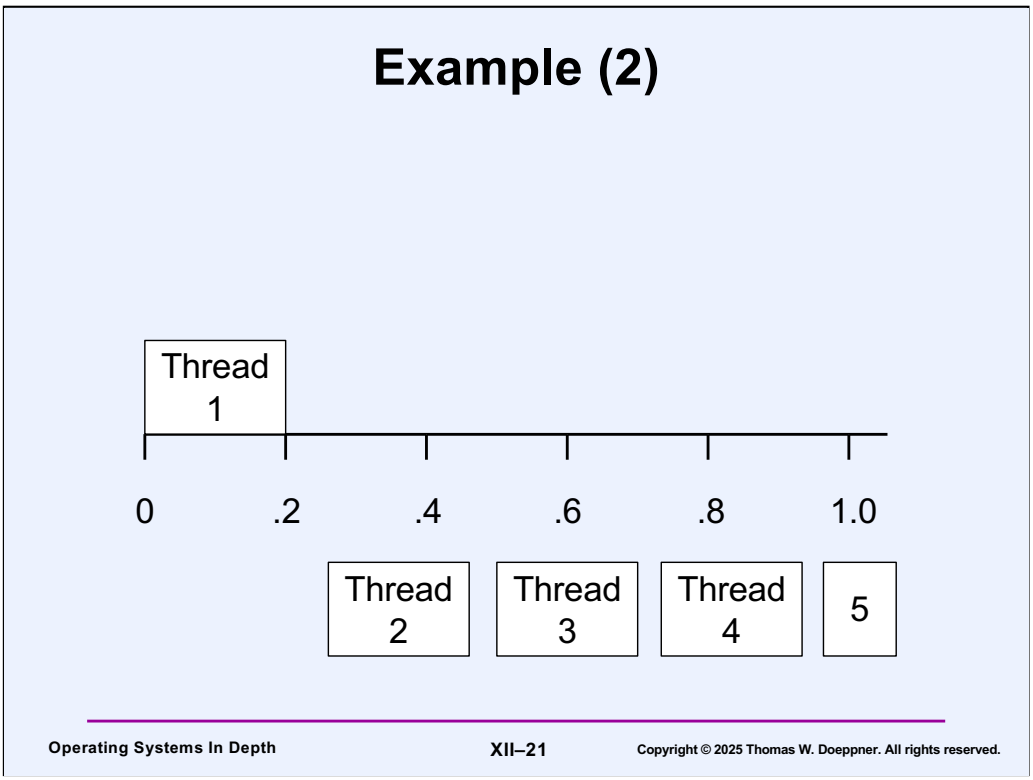
- Implemented by keeping track of how much time a thread has actually run during the period
  - *lag* is how much much more time till it reaches its quota
  - *eligible* for running if lag is positive
  - *virtual deadline* is earliest time by which it can achieve its quota
    - assuming it starts now and runs for 1/n second
  - next thread to run is the one with the *earliest virtual deadline*

EEVDF (3)

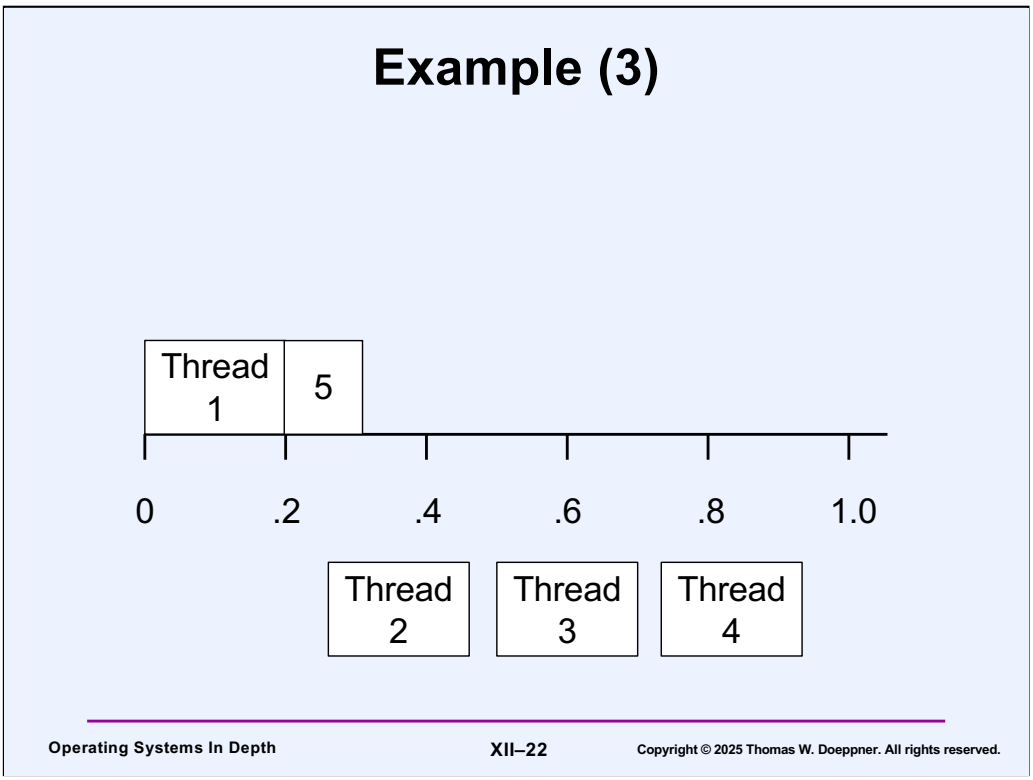
- If a thread requires low latency, it will be given a shorter time slice (say 1/(2n))
  - its virtual deadline thus comes sooner than those for other threads
  - it runs for correspondingly shorter periods of time
  - for any scheduling interval it gets the same total amount of time as other threads
    - but in smaller chunks



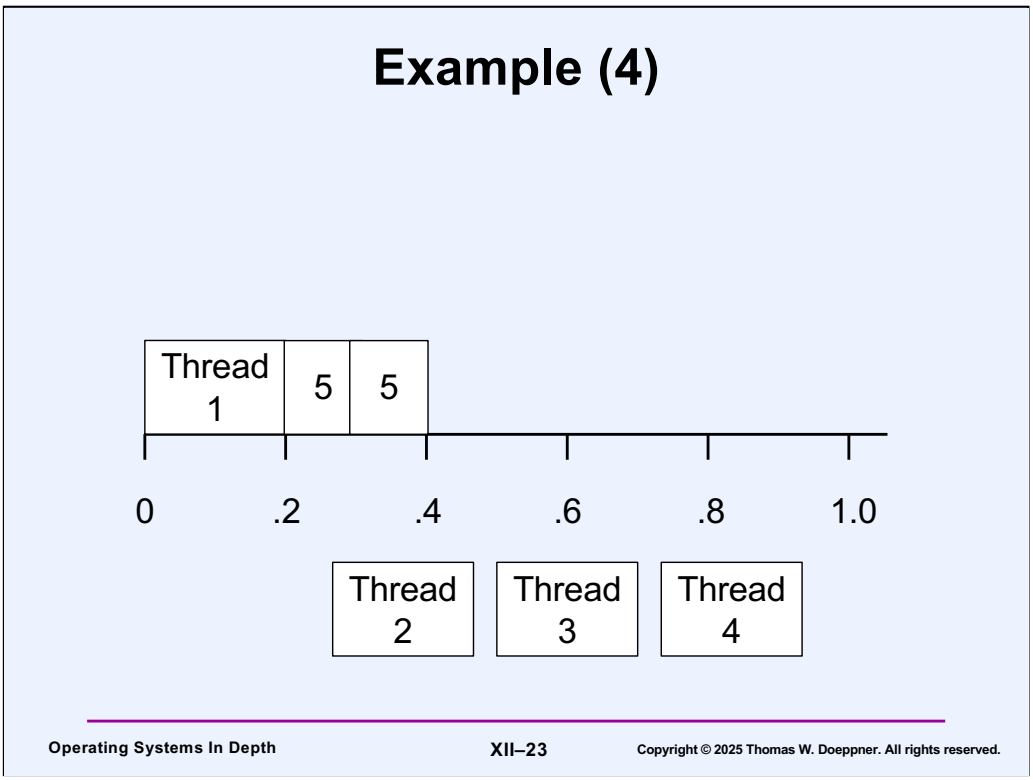
Thread 1 has a time slice of .2 seconds and has just completed a time slice. Its lag is 0, and thus it won't be eligible until the end of the one-second scheduling interval. The other threads have positive lags of .2 seconds each, so one of them will be chosen to run next



Suppose thread 5 has a time slice of .1 seconds (all the others have a time slice of .2 seconds). Thus its virtual deadline is .3, which is earlier than that of the other threads. And thus it will be scheduled immediately after thread 1.



Thread 5 has now completed its time slice, and its lag is .1. Since it still has the smallest positive lag, it will be scheduled (again) next.



Thread 5 completes its second time slice. Its lag is now 0, thus threads 2, 3, and 4 will run before threads 1 and 5 run again.

Determining the Time Slice

- Linux defines a new scheduling parameter
  - latency-nice
    - settable by a system call
  - the lower the value, the less latency a thread has, and thus the shorter its time slices

Real-Time Scheduling

- Jobs j_i arrive at times t_i with deadlines d_i
  - find schedule satisfying constraints
  - does schedule exist?
  - if so, what is it?
  - in general, j_i, t_i, and d_i are not known ahead of time

Uniprocessor

- Earliest-deadline first
  - optimal
- Rate-monotonic scheduling of cyclic chores
  - easy

Multiprocessor

- Earliest-deadline first and rate-monotonic still work (non-optimally)
- Finding optimal schedule is NP-complete
  - equivalent to bin-packing problem

Assumptions

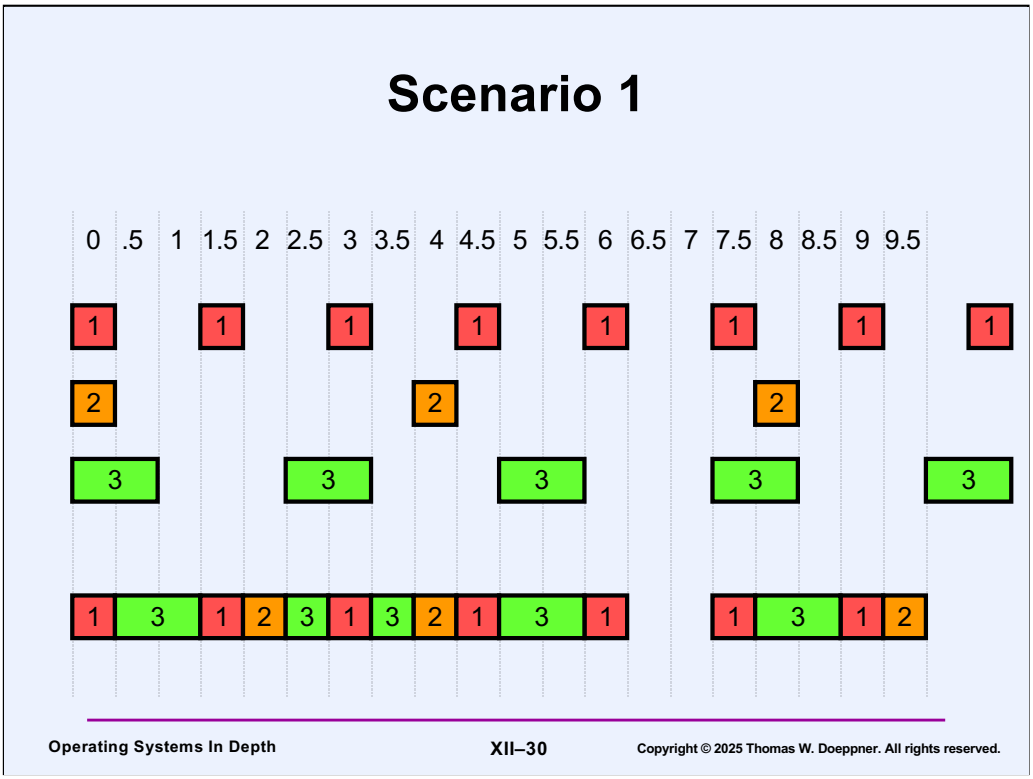
- Interrupts don't interfere (too much) with schedule
  - bounded interrupt delays
- Execution time really is predictable
  - what about effects of caching and paging?

Rate-Monotonic Scheduling

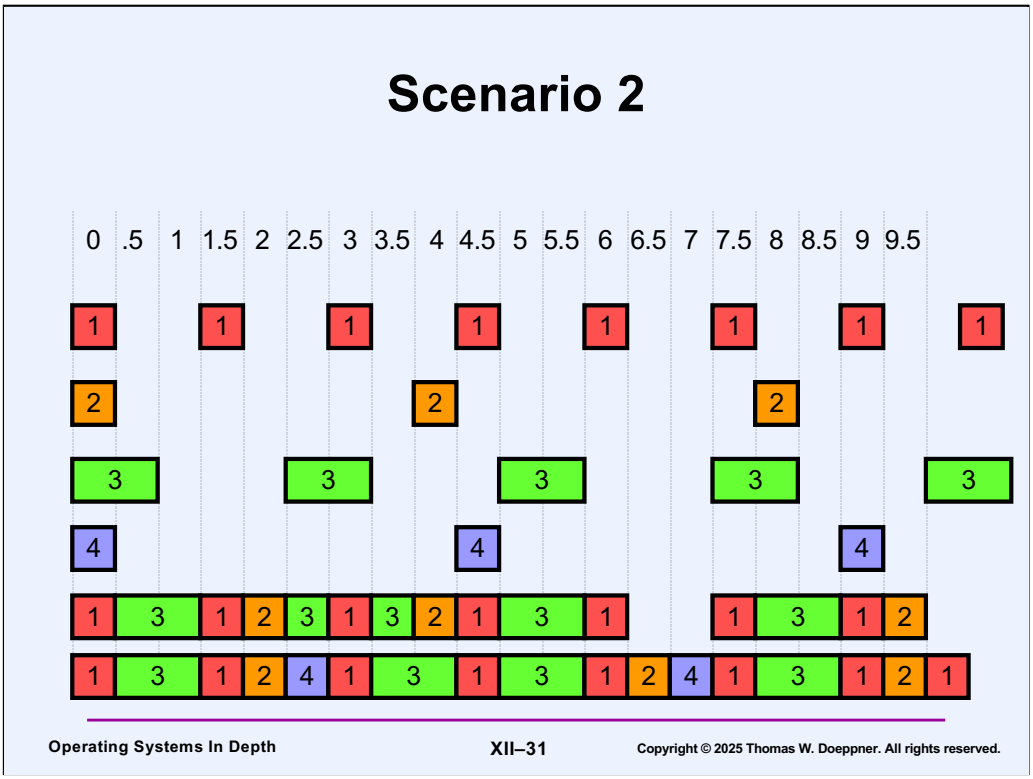
- **Periodic chores**
  - **period P_i**
  - **per-cycle processing time T_i (<= P_i)**
  - **feasible if $sum(T_i/P_i) <= 1$**
- **Rate-monotonic scheduling**
  - **each chore *i* is handled by a thread with priority $1/P_i$**
  - **preemptive, priority scheduling**
  - **works when $sum(T_i/P_i) <=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

Consider a situation in which we have a number of chores, each of which must be performed periodically. Can we easily compute a schedule for completing them so that all deadlines are met? (The deadline in this case is that each chore must be completed before its next period begins.) Clearly no such schedule can exist if the sum of the chores' duty cycles (ratio of per-cycle processing time to the length of the period) is greater than one. Though there are reasonably efficient algorithms that solve this scheduling problem in all cases (for example, shortest-completion-time first) we look at an algorithm that solves the problem in the restricted case that the sum of the duty cycles doesn't exceed the figure given in the slide.

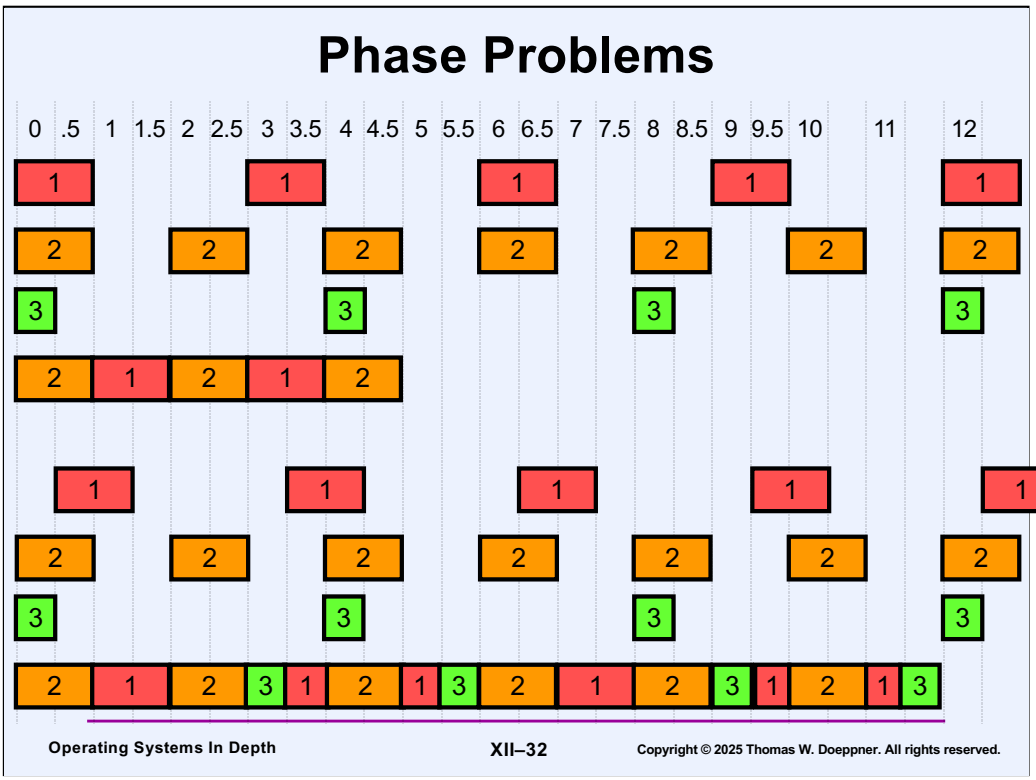
Rate-monotonic scheduling involves assigning each chore's thread a priority proportional to its frequency (or inversely proportional to the length of its period). The threads are scheduled according to priority, with higher-priority threads preempting the execution of lower-priority threads. This algorithm is particularly nice because it can be built on top of the priority-based schedulers of commercial operating systems.



The slide shows a successful application of rate-monotonic scheduling. The top three rows in the figure show three cyclic chores. The first occurs every 1.5 seconds and requires .5 seconds. The second occurs every 4 seconds and requires .5 seconds. The third occurs every 2.5 seconds and requires 1 second. The fourth row shows the schedule obtained using rate-monotonic scheduling.



Here rate-monotonic scheduling doesn't work, but earliest-deadline-first does. We've added one more cyclic chore to the example of the previous slide, this one requiring .5 seconds every 4 seconds. The fifth row is the beginning of a schedule using rate-monotonic scheduling, but we can't complete the new chore within its period. However, by using **earliest deadline first**, we are able to satisfy the deadline, as shown in the bottom row.



This slide shows the effect of phase on rate-monotonic scheduling. The top three lines show three chores. The first requires 1 second every 3 seconds, the second requires 1 second every 2 seconds, and the third requires .5 seconds every 4 seconds. The fourth line shows what happens when rate-monotonic scheduling is used: the third chore can't make its deadline even once.

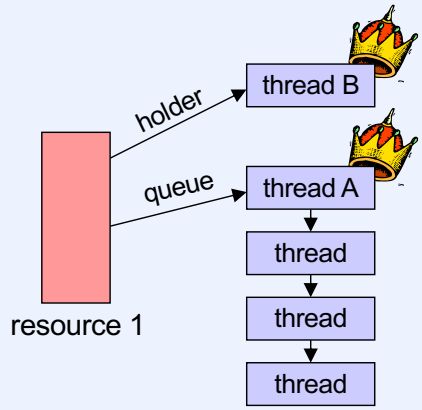
In the bottom half of the figure, we've started the first chore a half-second after the others. The last line shows the rate-monotonic schedule: all three chores consistently meet their deadlines.

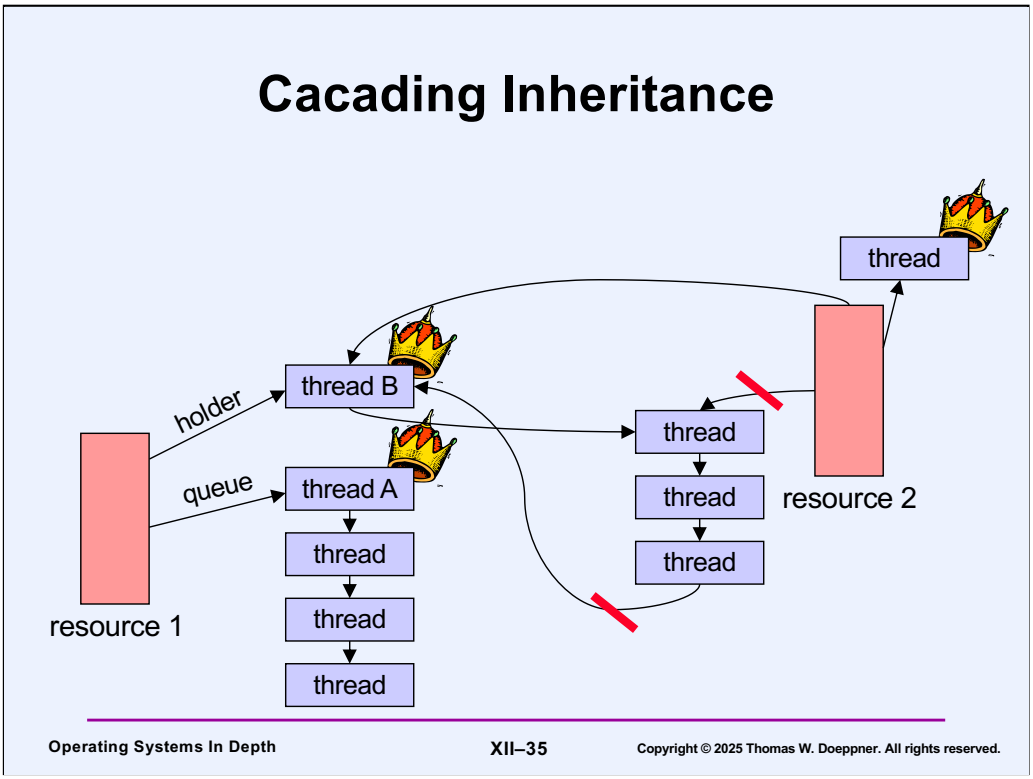
Priority Problem

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

Priority Inheritance

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





Thread B might, in turn, be waiting on another mutex. So, thread B's (newly acquired) high priority is given to the holder of that mutex, and B moves to the front of the queue for that mutex.

Linux Scheduling

- Policies
  - SCHED_FIFO
    - "real time"
    - infinite time quantum
  - SCHED_RR
    - "real time"
    - adjustable time quantum
  - SCHED_DEADLINE
    - earliest-deadline first
  - SCHED_OTHER
    - "normal" scheduler
    - parameterized allocation of processor time

Linux currently supports four scheduling policies, as shown in the slide. The first three policies are somewhat inaccurately called "real-time" policies. What's normally used is SCHED_OTHER (blame POSIX for the name). Super-user privileges are required to set the scheduling policy.

Linux Scheduler Evolution

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

The old scheduler is essentially unchanged since the early days of Linux. It's very simple and works reasonably well on lightly loaded uniprocessors. The O(1) scheduler, introduced in release 2.5, is considerably more sophisticated. CFS is based on stride scheduling, but shares much with the implementation of the O(1) scheduler. We don't cover the EEVDF scheduler here – not enough is known about its Linux implementation.

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**
  - ***counter*: records processor consumption**

We start by discussing the old scheduler. Four variables are used for scheduling processes, as shown in the slide. The first three are inherited from a process's parent. Using (privileged) system calls, one can change the first two. The third, **priority**, can be worsened with non-privileged system calls, but can be improved only with a privileged system call.

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

Implementing time slicing requires the support of the clock-interrupt handler. Each process's **counter** variable is initialized to the process's **priority**. At each clock "tick", **counter** is decremented by one. When it reaches zero, the process's time slice is over and it must relinquish the processor. (How **counter** gets a positive value again is discussed in the next slide.) In current versions of Linux, "HZ" is 1000.

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 *counter*s for runnable processes are all 0**
  - **– sleeping processes get "boost" at wakeup**
    - **- at beginning of each cycle, for each process (runnable or not):**

      counter = counter/2 + priority

SCHED_OTHER is a throughput policy, which means that all processes are guaranteed to make progress, though some (those with better priority) make faster progress (get a higher percentage of processor cycles) than others. It's easiest to understand if we assume a fixed number of processes, all of which use the SCHED_OTHER policy and all of which remain runnable. Scheduling is based on cycles, the length of which (measured in ticks) is the sum of the (runnable) processes' priorities. In each cycle, each process thus gets **priority** ticks of processor time, for that process's value of **priority**. This is done by setting each process's *counter* to **priority** when the cycle starts.

Sleeping processes are given a "boost" when they wake up. The rationale is that we want to favor interactive and I/O intensive requests (the latter don't use much processor time and thus let's quickly get them back to waiting for an I/O operation to complete). To implement this, at the end of each cycle, all processes (not just all runnable processes) have their **counters** set as shown in the slide. (Thus, the maximum value of **counter** is twice the process's priority.)

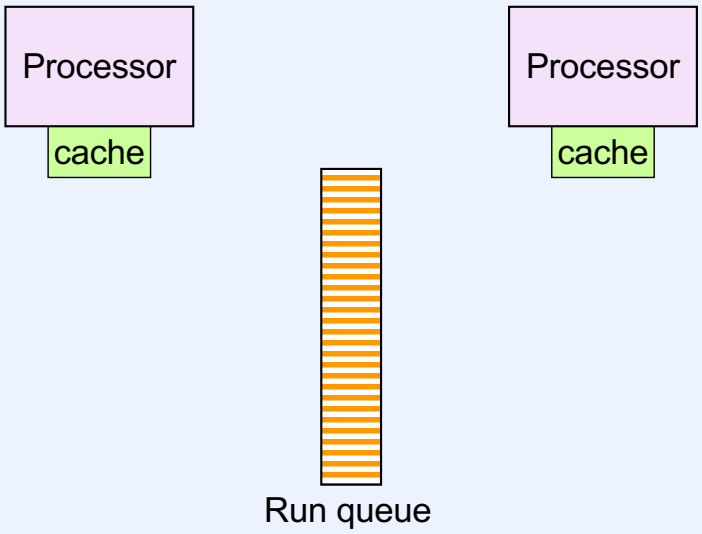
Note that in this scheduler, numerically high priorities provide a process with more computing time than do numerically low priorities.

Old Scheduler: Who's Next?

- Run queue searched beginning to end
  - new arrivals go to beginning
  - 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

When a process gives up the processor, it executes (in the kernel) a routine called **schedule** which determines the next process to run, as shown in the slide. Note that all runnable processes are examined so as to find the "best" one. This is not a good strategy if there are a lot of them; on a workstation there will be few, but on a busy server there could be many.

Diagram



Old Scheduler: Problems

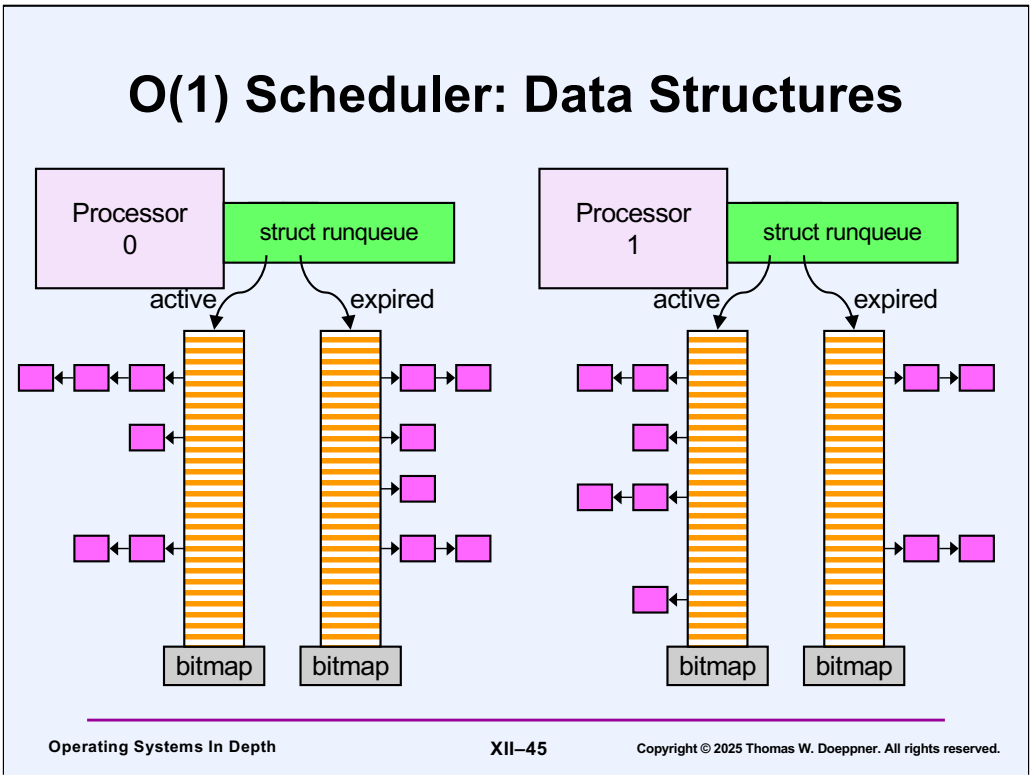
- O(n) execution
- Poor interactive performance with heavy loads
- SMP contention for run-queue lock
- SMP affinity
  - cache "footprint"

The main problems with the old scheduler are summarized in the slide. As we've seen, the scheduler must periodically examine all processes, as well as examining all runnable processes for each scheduling decision. When a system is running with a heavy load, interactive processes will get a much smaller percentage of processor cycles than they do under a lighter load. Since there's one run queue feeding all processors, all must contend for its lock. Finally, though some attempt is made at dealing with processor-affinity issues, processes still tend to move around among available processors (and thus losing any advantage of their "cache footprint").

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

The O(1) scheduler deals with all the concerns dealt with by the old one along with some additional concerns, as shown in the slide.

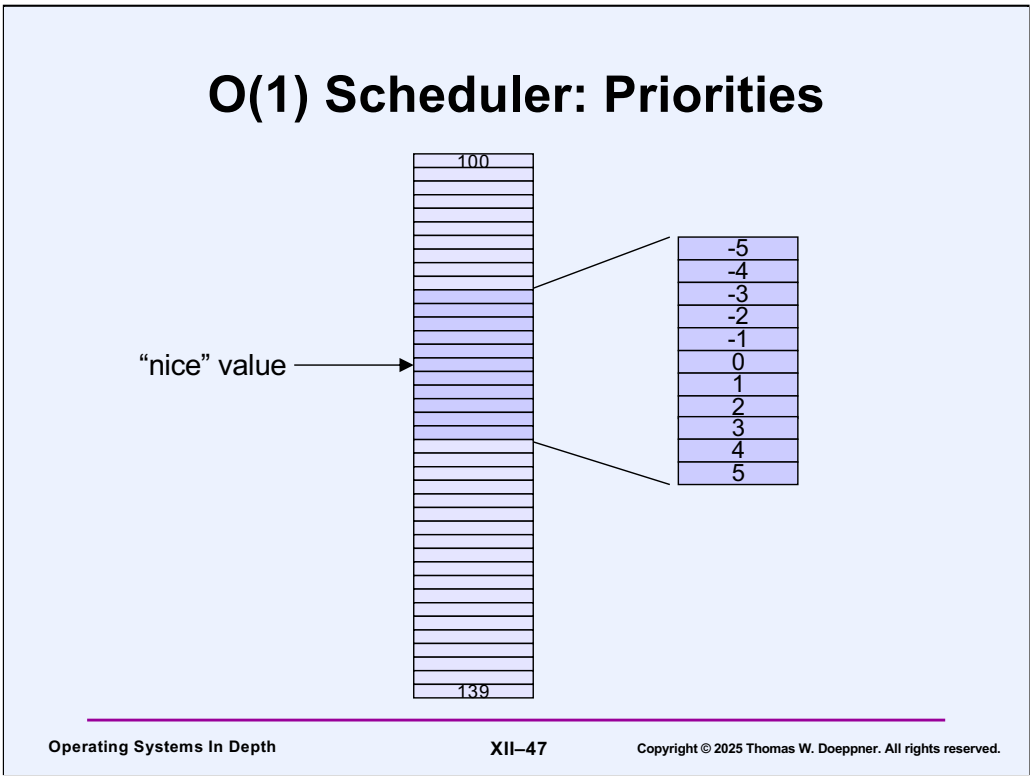


The O(1) scheduler associates a pair of queues with each processor via a per-processor **struct runqueue**. Each queue is actually an array of lists of processes, one list for each possible priority value. There are 140 priorities, running from 0 to 139; "good" priorities have low numbers; "bad" priorities have high numbers. Real-time priorities run from 0 to 99; normal priorities run from 100 to 139. Associated with each queue is a 140-element bit map indicating which priority values have a non-empty list of processes.

O(1) Scheduler: Queues

- Two queues per processor
  - active: processes with remaining time slice
  - expired: processes whose time slices expired
  - each queue is an array of lists of processes of the same priority
    - bitmap indicates which priorities have processes
  - processors scheduled from private queues
    - infrequent lock contention
    - good affinity

When processes become runnable they are assigned time slices (based on their priority) and put on some processor's **active** queue. When a processor needs a process to run, it chooses the highest priority process on its active queue (this can be done quickly (and in time bounded by a constant) by scanning the queue's bit vector to find the first non-empty priority level, then selecting the first process from the list at that priority). When a process completes its time slice, it goes back to the expired queue of its processor (as explained shortly). If it blocks for some reason and later wakes up, it will generally go back to the active queue of the processor it last was on. Thus, processes tend to stay on the same processor (providing good use of the cache footprint). Since processors rarely access other processors' queues, there is very little lock contention.



The values over which a process's priority may range are determined by its "nice" value, settable by a system call, and are within a range of +5 and -5. The default is a nice value of 0, in which case the process's priority ranges from 115 through 125. Within the range, the priority is determined by how much time the process has been sleeping in the recent past. In no case will a non-real-time process's priority be less than 100 or greater than 139.

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**
  - **interactive processes are defined as those whose priority is above a certain threshold**
- **Time-slice expiration**
  - **normal processes join expired queue**
  - **real-time join active queue**

When a process completes its time slice, it is inserted into its processor's expired queue, unless it's a real-time process, in which case it rejoins the active queue. The intent is that real-time processes get another time slice on the processor, while other processes have to wait a bit on the expired queue. When there are no processes remaining in the active queue, the two queues are switched. Of course, if there are interactive processes that often block, then resume execution (and thus their time slices never expire), the active queue might never empty out. So, if the processes in the expired queue have been waiting too long (how long this is depends on the number of runnable processes on the queue), interactive processes go to the expired queue rather than the active queue. Runnable real-time processes never go to the expired queue: they are always in the active queue (and always have priority over non-real-time processes). Thus two queues are employed as a means to guarantee that, in the absence of real-time processes, all processes get some processor time. Lower priority processes will remain on the active queue until all higher-priority processes have moved to the expired queue.

The net effect is similar to the old scheduler: in the absence of real-time processes, processes get the processor in proportion to their priority. However, interactive processes (those that have recently woken up) get extra time slices.

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**

Since processors schedule strictly from their own private queues, load balancing is an issue (it wasn't with the old scheduler, since there was only one global queue serving all processors). Each processor checks its queues for emptiness every millisecond. If empty, it calls a load balancing routine to find the processor with the largest queues and then transfers processes from that processor to the idle one until they are no longer imbalanced (they are considered balanced if they are no more the 25% different in size). Similarly, each processor checks the other processors' queues every 250 milliseconds. If an imbalance is found (and it's not just a momentary imbalance but has been that way since the last time the processor's queues were examined), then load balancing is done.