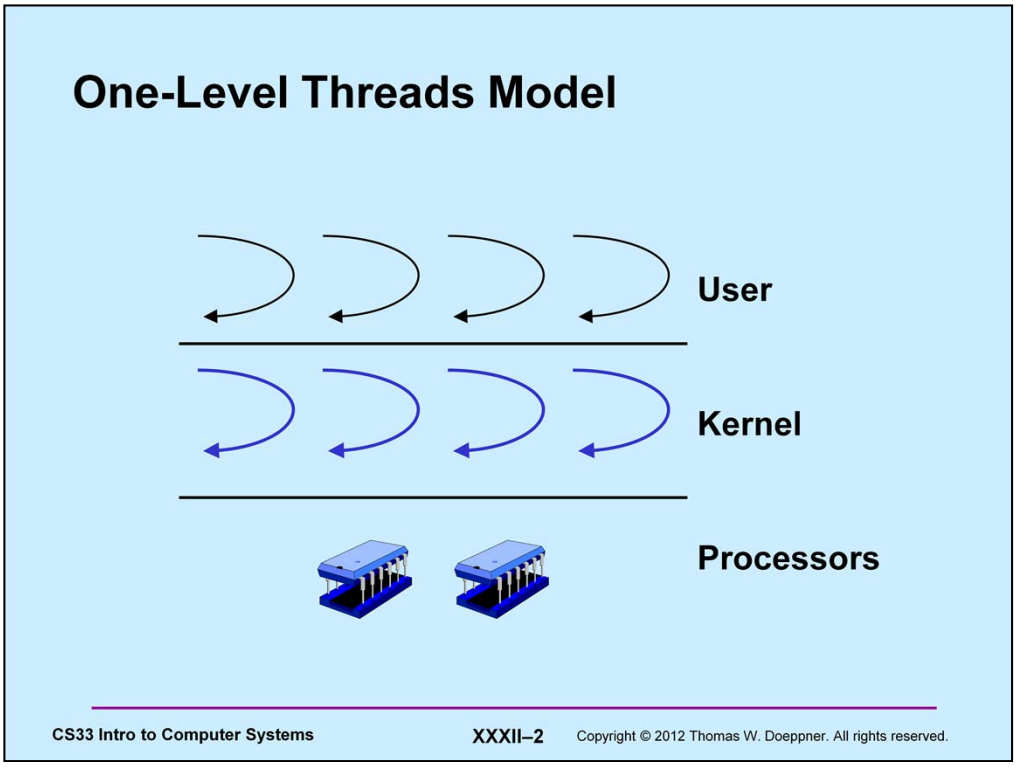
CS 33

Threads and the Operating System

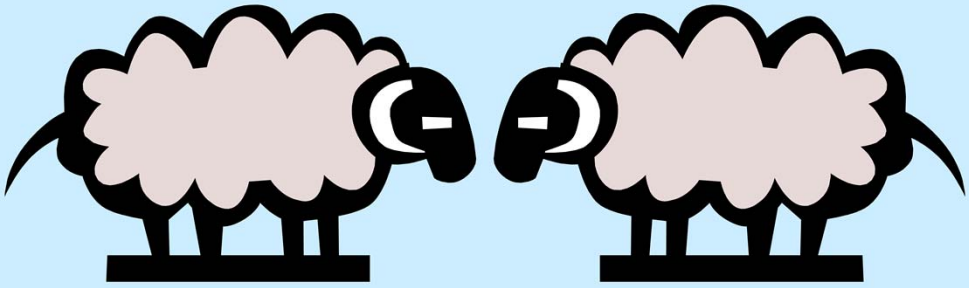


In most systems there are actually two components of the execution context: the user context and the kernel context. The former is for use when a processor is executing user code; the latter is for use when the processor is executing kernel code (on behalf of the chore). How these contexts are manipulated is one of the more crucial aspects of a threads implementation.

The conceptually simplest approach is what is known as the one-level model: each thread consists of both contexts. Thus a thread is scheduled to a processor and the processor can switch back and forth between the two types of contexts. A single scheduler in the kernel can handle all the multiplexing duties. The threading implementation in Windows is (mostly) done this way.

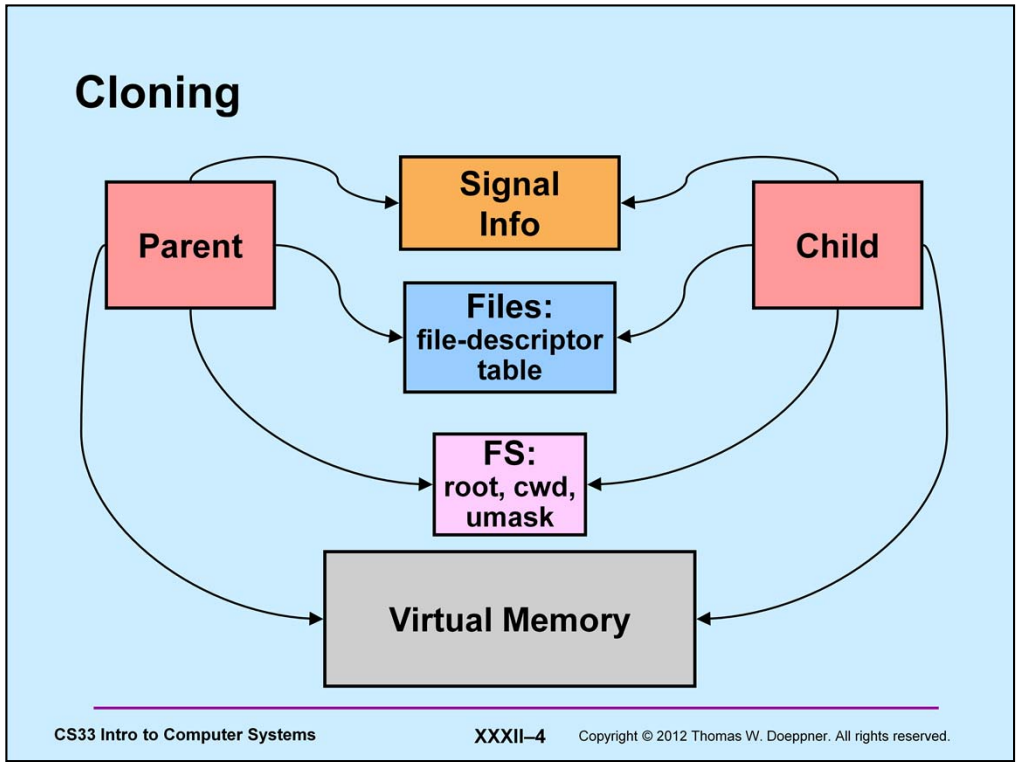
Variable-Weight Processes in Linux

- Variant of one-level model
- Portions of parent process selectively *copied* into or *shared* with child process
- Children created using *clone* system call

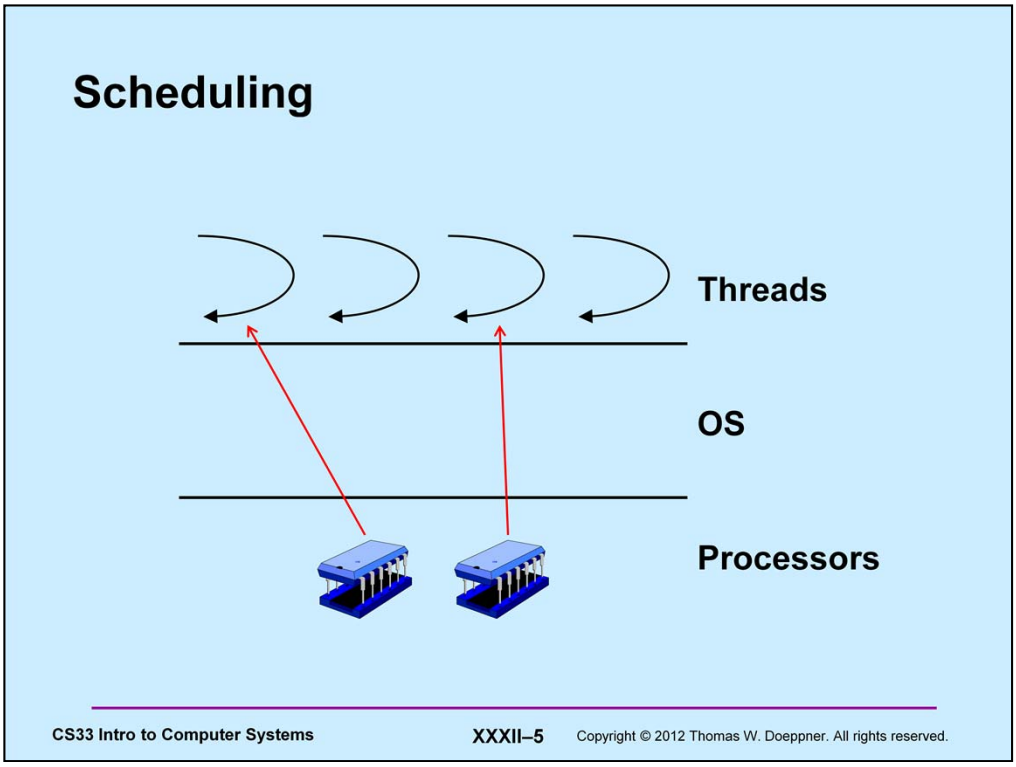


Unlike most other Unix systems, which make a distinction between processes and threads, allowing multithreaded processes, Linux maintains the one-thread-per-process approach. However, so that we can have multiple threads sharing an address space, Linux supports the *clone* system call, a variant of *fork*, via which a new process can be created that shares resources (in particular, its address space) with the parent. The result is a variant of the one-level model.

This approach is not unique to Linux. It was used in SGI's IRIX and was first discussed in early '89, when it was known as variable-weight processes. (See "Variable-Weight Processes with Flexible Shared Resources," by Z. Aral, J. Bloom, T. Doeppner, I. Gertner, A. Langerman, G. Schaffer, *Proceedings of Winter 1989 USENIX Association Meeting*.)



As implemented in Linux, a process may be created with the *clone* system call (in addition to using the *fork* system call). One can specify, for each of the resources shown in the slide, whether a copy is made for the child or the child shares the resource with the parent. Only two cases are generally used: everything is copied (equivalent to fork) or everything is shared (creating what we ordinarily call a thread, though the "thread" has a separate process ID).



The operating system is responsible for multiplexing the execution of threads on the available processors. The OS's *scheduler* is responsible for assigning threads to processors. Periodically, say every millisecond, each processor is interrupted and calls upon the OS to determine if another thread should run. If so, the current thread on the processor is preempted in favor of the next thread. Assuming all threads are treated equally, over a sufficient period of time each thread gets its fair share of available processor time. Thus, even though a system may have only one processor, all threads make process and give the appearance of running simultaneously.

Real-Life Example

- Your iPhone is broken
  - mp3 player
- You're watching a phenomenal youtube video
- You're computing a fractal model of the universe
- A 33 assignment is due
  - editor, compiler, debugger
- You've got to do everything on one computer
- Can your scheduler hack it?


```
void *Timeout_WatchDog(void *arg) {
  Timeout_t *timeout = (Timeout_t *)arg;
  pthread_barrier_wait(&timeout->client->started);
  struct timespec to;
  to.tv_sec = Timeout_wait_secs;
  to.tv_nsec = 0;
  struct timeval now;
  while(1) {
    if (nanosleep(&to, 0) == -1 ) {
      perror("nanosleep");
      exit(1);
    }
    if (timeout->active == 0) {
      pthread_cancel(timeout->client->thread);
      phread_exit(0);
    } else
      timeout->active = 0;
```


Scheduler Concerns

- Schedule threads equitably
  - this should be easy ...
- Provide good interactive response
  - what defines "interactive"?
- Make sure important things get done in a timely manner
  - this is tough ...
    - » (so tough, we won't talk about it ...)

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 (discussed soon)

The old scheduler was introduced in the early 1990s. 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.

Old Scheduler

- Three per-process scheduling variables
  - *policy*: which one
    - » we look at only the default policy
  - *priority*: time-slice parameter ("nice" value)
  - *counter*: records processor consumption

We start by discussing the old scheduler. Three variables are used for scheduling processes, as shown in the slide. The first two are inherited from a process's parent. Using (privileged) system calls, one can change the first two. The second, *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:

      counter = counter/2 + priority

The default scheduling policy 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 default 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.

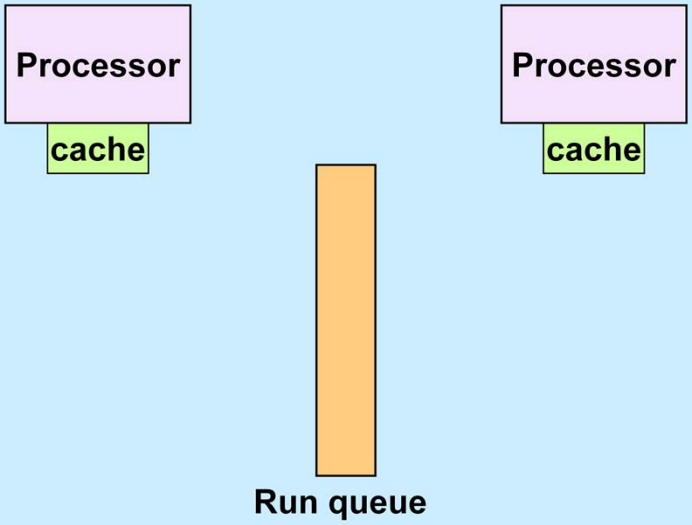
Sleeping processes are given a "boost" when they wake up. The rationale is that we want to favor interactive and I/O intensive requests (the latter don't use much processor time and thus let's quickly get them back to waiting for an I/O operation to complete). To implement this, at the end of each cycle,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

Old Scheduler: Who's Next?

- Run queue searched beginning to end
  - new arrivals go to front
- Next running process is first process with highest "goodness"
  - *counter* for default 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 PC there will be few, but on a busy server there could be many.

Diagram



Old Scheduler: Problems

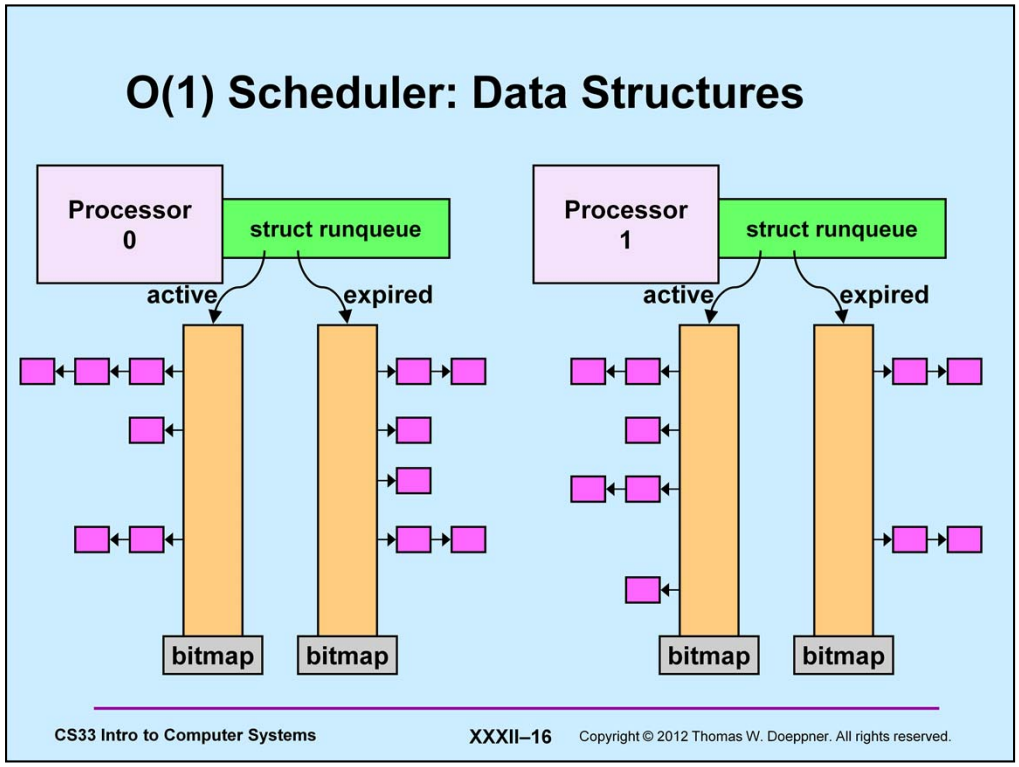
- O(n) execution
- Poor interactive performance with heavy loads
- Contention for run-queue lock
- Processor affinity
  - cache "footprint"

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 MP 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.

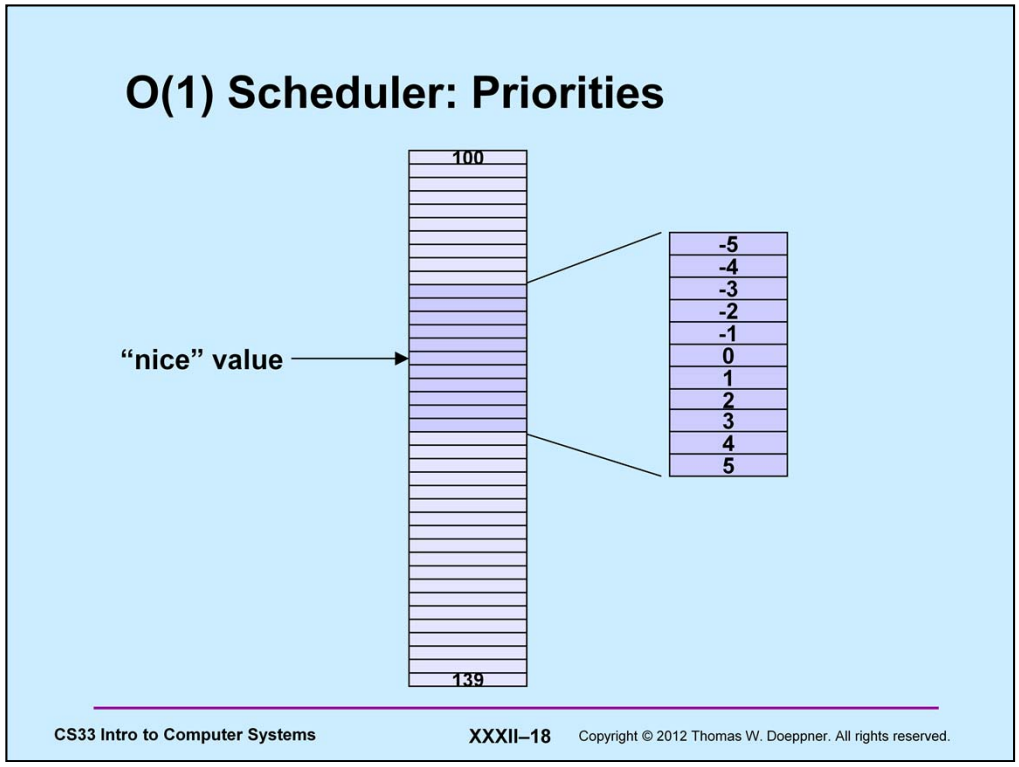


The O(1) scheduler associates a pair of queues with each processor via a per-processor *struct runqueue*. Each queue is actually an array of lists of processes, one list for each possible priority value. There are 140 priorities, running from 0 to 139; "good" priorities have low numbers; "bad" priorities 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 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
    - » 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 either the active or 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
  - higher priority processes get longer time quanta
- Time-slice expiration
  - interactive processes rejoin active queue
    - » unless processes have been on expired queue too long

When a process completes its time slice, it is inserted into either its processor's active queue or its processor's expired queue, depending on its priority. The intent is that interactive and real-time processes get another time slice on the processor, while other processes have to wait a bit on the inactive queue. When there are no processes remaining in the active queue, the two queues are switched. Of course, if there are interactive processes, 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 completing their time slices 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.

Proportional-Share Scheduling

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

The Stride scheduling paper is Stride Scheduling: Deterministic Proportional-Share Resource Management, Technical Memorandum MIT/LCS/TM-528, MIT Laboratory for Computer Science, June 1995.

There really isn't any documentation for CFS. It's making its first appearance in Linux 2.6.23. Some information may be found at http://kerneltrap.org/node/8059.

CFS and stride scheduling are pretty much the same thing. The presentation here is based on stride scheduling, though using different terminology than did the original paper.

Metered Processors

To measure the usage of a processor, let's assume the existence of a meter.

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

Assuming all threads are equal, all started at the same time, and all run forever, the intent is to share the processor equitably. Note that has the time between clock ticks approaches zero, each thread gets 1/n of total processor time, where n is the number of threads.

Issues

- 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
(Rhode Island Variation)

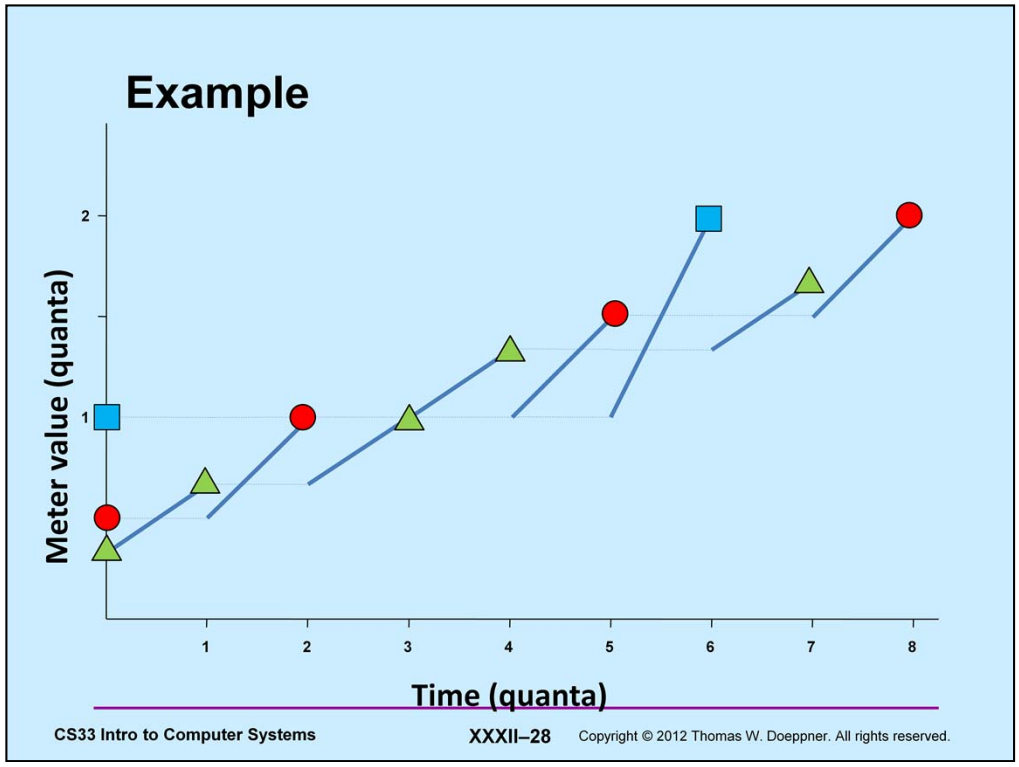
Let's now assume that meters can be "fixed" so that they run more slowly than they should. Thus a thread with a fixed meter gets charged for less processor time than it has actually used.

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



The slide illustrates the execution of three threads using stride scheduling. Thread 1 (labeled with a triangle) has a paid a bribe of three tickets. Thread 2 (labeled with a circle) has paid a bribe of two tickets, and thread three (labeled with a square) had paid only one ticket. The thicker lines indicate when a thread is running. Their slopes are proportional to the meter rates (and inversely proportional to the bribe).

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 routine 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.

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 actual (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, since 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. This value would be that obtained if there were only one thread in the system, which had purchased all the tickets in use. Thus it would be executing at all clock ticks, and its *metered_time* would be incremented by *1/TotalBribe* at each tick. So, we'll keep track of what this value would be and use it to initialize each new thread's *metered_time*.

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);
}
```

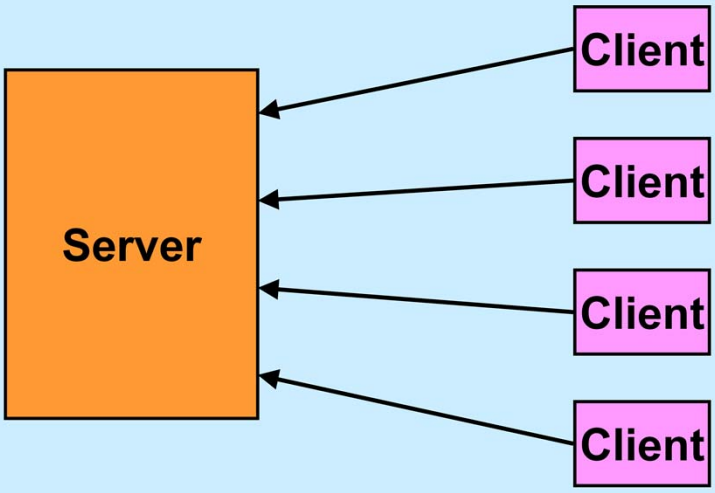

Thread Leaves, then Returns

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

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


When a thread leaves the system temporarily, perhaps to wait for I/O, we keep track of how much its *metered_time* differs from *TotalTime* (what the "fair" value would be). When the thread resumes execution, it gets credit (or debit) for this time. Note that this isn't necessarily the best thing to do. In CFS, a returning thread gets a slight bonus for having been sleeping (and thus gets the processor sooner than it would have otherwise) — this bonus is in the form of a smaller *metered_time* than it would have been given in the code in the slide.

Designing a Fast Server



Scenario

- N-processor shared-memory multiprocessor
- Multiple high-speed network interfaces
- Thousands of concurrent clients
  - requests range from trivial to lengthy, pure compute to I/O intensive

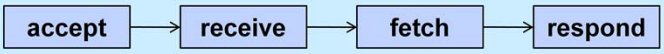
Handling a Request

1) Accept connection
2) Receive request
3) Fetch data from file system
4) Send response

This is, of course, highly simplified!

Implementation

- Threads
  - one thread per client-request
    - » 10,000 concurrent requests = 10,000 threads
- Events
  - represent each client-request with a control block
  - steps form a pipeline



  - transitions between steps are events
  - have a relatively small number of threads to handle the events

A Thread/Connection Approach

- Use a two-level threads implementation with multiple kernel threads
- Reduce stack size to a minimum
  - grow stacks dynamically when necessary
- Capriccio does this
  - supports up to 100,000 concurrent threads
  - Knot, a web server handling tens of thousands of concurrent clients, built on top of Capriccio
  - performed better than Apache

Capriccio and Knot were research projects at the University of California, Berkeley. A paper describing Capriccio is "Capriccio: Scalable Threads for Internet Services," presented at SOSP 2003. A copy can be found at http://www.stanford.edu/class/cs240/readings/capriccio-sosp-2003.pdf. A paper describing Knot, its use of Capriccio, and its performance is "Why Events Are A Bad Idea (for high-concurrency servers)," presented at HotOS 2003. A copy can be found at http://research.microsoft.com/apps/pubs/default.aspx?id=101665.

Programming with Events …

- Use the pipeline approach with a small number of threads
  - researchers at University of Waterloo (Canada) did this and produced a better-performing server than Knot
  - so far, theirs is the final word

The Waterloo work is described in "Comparing the Performance of Web Server Architectures," presented at EuroSys 2007. A copy can be found at http://www.cs.uwaterloo.ca/~brecht/papers/getpaper.php?file=eurosys-2007.pdf.

Summing Up

- Writing highly concurrent programs is generally easier using strictly threads than it is using an event-based strategy
- For very high levels of concurrency, one can generally produce better-performing applications using an event-based strategy augmented with threads

The End

Well, not quite …

Database is due on 12/18.

We'll do our best to get everything else back next week.

Happy coding and happy holidays!

