Real-World Scheduling
Linux supports three scheduling policies, as shown in the slide. What’s normally used is SCHED_OTHER (blame POSIX for the name). The other policies are somewhat inaccurately called “real-time” policies. Super-user privileges are required to set the scheduling policy.
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.
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.
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.
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.)
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.
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”).
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.
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.
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.
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.
Scheduling in Windows

- Handling “normal” interactive and compute-bound threads
- Real-time threads
- Multiple processors
Threads are assigned base priorities, chosen from the six ranges shown in the slide (usually the middle of a range). Their current priority is some value equal to or greater than their base (always equal to the base for real-time threads). When waking up from a sleep, a thread’s current priority is set to its base priority plus some wait-specific value (usually in the range 1 to 6, depending on what sort of event it was waiting for). A thread’s current priority is decremented by one, but to no less than the base, each time a quantum expires on it. Threads of the current window get three times the time quantum of other threads, thus giving them a greater portion of available processor time.
Windows employs an array of dispatch queues, one for each priority level.
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
When a thread is created and first made runnable, its creator (running in kernel mode) puts it in the deferred ready state and enqueues it in the deferred ready queue associated with the current processor. It’s also randomly assigned an ideal processor, on which it will be scheduled if available. This helps with load balancing. Its creator (or, later, the thread itself) may also give it an affinity mask indicating the set of processors on which it may run.

Each processor, each time it completes the handling of the pending DPC requests, checks its deferred ready queue. If there are any threads in it, it processes them, assigning them to processors. This works as follows. The DPC handler first checks to see if there are any idle processors that the thread can run on (if it has an affinity mask, then it can run only on the indicated processors). If there are any acceptable idle processors, preference is given first to the thread’s ideal processor, then to the last processor on which it ran (to take advantage of the thread’s cache footprint). The thread is then put in the standby state and given to the selected processor as its next thread. The processor would be currently running its idle thread, which repeatedly checks for a standby thread. Once found, the processor switches to the standby thread.

If there are no acceptable idle processors, then the thread is assigned to its ideal processor. The DPC handler checks to see if the thread has a higher priority than what’s running on that processor. If so, it puts the thread in the standby state, and sends the processor an interrupt. When the processor returns from its interrupt handler it will notice the higher-priority thread in standby state and switch to it, after first putting its current thread on its deferred ready list. Otherwise, the DPC handler puts the thread in the ready state and puts it in one of the ideal processors’ ready queues according to the thread’s priority.

When a thread completes its time quantum (which is dealt with by a DPC), the processor searches its ready queues for a thread of equal or higher priority and switches to it, if it finds one. Otherwise it continues with the current thread.

An executing thread might perform some sort of blocking operation, putting itself in a wait queue. Its processor then searches its ready queues for the next thread to run.

When a thread is made runnable after it has been in the wait state, it’s put into the deferred ready queue of the processor on which the thread doing the unwait operation was running.
Recent processor designs put a memory controller inside of each processor chip, with separate memory attached to each chip. Thus access to the attached memory is relatively quick. To reach memory attached to other chips from a particular chip, the request must be routed through one or more other processor chips. Thus accessing other memory is more time-consuming than accessing local memory. Such systems are referred to as non-uniform memory access (NUMA) systems. What’s shown here is the Intel Core 7 with Intel’s QuickPath Interconnect. More information can be found at http://www.intel.com/content/www/us/en/io/quickpath-technology/quick-path-interconnect-introduction-paper.html.
How Linux deals with these concerns is discussed at http://lwn.net/Articles/80911/.  

Scheduling Concerns

- Hyperthreads
  - two instruction streams sharing same functional units and same L1 cache
- How long does cache footprint matter?
  - what cache parameters are important?
- When is it a good idea to put a thread on:
  - a different core?
  - a different NUMA node?
Since hyper threads share the L1 cache (and also L2 and L3 caches), there’s no real cost in switching a thread from one hyperthread to another in the same core. However, note that if two threads are running concurrently on two hyperthreads of the same core, each will most likely run substantially slower than if it were running on the core without a competing hyperthread (this is because the two hyperthreads on the core share the same set of functional units).
If a thread has run recently on one core, then it's clearly better to run it on that core again than to move it to another core, due to its footprint in its L1 cache. But suppose both hyperthreads of the other core (sharing the same L2 cache) are idle, and one hyperthread of our thread's original core is busy, but the other hyperthread is idle. Does it now make sense to run our thread on the core?

Suppose we create a new thread in a multithreaded process. If all cores have at least one busy hyperthread, is it better to run the new thread on the same core as its creator, or on a different core? Similarly, suppose rather than creating a new thread on a multithreaded process, a thread calls fork, thus creating a new (single-threaded) process. Should this new thread run on its creator's core or on a different core?
Now we have threads running on separate NUMA nodes. It is clearly very expensive to move a thread from one node to another, if its address space remains in the memory of the of the original node. If a thread calls exec, does it now make sense to move it to different NUMA node? (Consider both the user and kernel portions of its address space.)
Quiz 1

We have a system comprised of two NUMA nodes, each with four cores, each with two hyperthreads. The first node has four threads of different processes running on it; the other node is completely idle. One of the threads on the first node calls `pthread_create`.

a) The new thread should be assigned to an idle hyperthread on the first node

b) The new thread should be assigned to a hyperthread on the other node

c) The new thread should be assigned to a hyperthread on the same node and some other thread should move to the other node
Quiz 2

We have a system comprised of two NUMA nodes, each with four cores, each with two hyperthreads. The first node has four threads of different processes running on it; the other node is completely idle. One of the threads on the first node calls fork.

a) The new thread should be assigned to an idle hyperthread on the first node

b) The new thread should be assigned to a hyperthread on the other node

c) The new thread should be assigned to a hyperthread on the same node and some other thread should move to the other node
Quiz 3

We have a system comprised of two NUMA nodes, each with four cores, each with two hyperthreads. The first node has four threads of different processes running on it; the other node is completely idle. One of the threads on the first node calls exec.

a) The thread should be assigned to an idle hyperthread on the first node

b) The thread should be assigned to a hyperthread on the other node

c) The thread should be assigned to a hyperthread on the same node and some other thread should move to the other node