OS-LAB-3 Report

1)

Switching between kernel threads. If a kernel thread calls sched() or yield() or sleep(), the kernel saves that kernel thread's registers, picks another runnable kernel thread, and restores its registers -- restoring ESP and EIP cause a stack switch and control switch. scheduler() also switches user segments; this has no immediate effect, it only affects what happens on a future return from kernel to user.

Every xv6 process has its own kernel stack and register set, as we saw in Chapter2. Each CPU has a separate scheduler thread for use when it is executing the scheduler rather than any process's kernel thread. Switching from one thread to another involves saving the old thread's CPU registers, and restoring previouslysaved registers ofthe new thread; the fact that %esp and %eip are saved and restored means that theCPU will switch stacks and switch what code it is executing.

swtch doesn't directly know about threads; it just saves and restores register sets,called contexts. When it is time for the process to give up the CPU, the process's kernel thread will call swtch to save its own context and return to the scheduler context.Each context is represented by a struct context*, a pointer to a structure stored on the kernel stack involved. Swtch takes two arguments: struct context **old andstruct context *new. It pushes the current CPU register onto the stack and saves thestack pointer in *old. Then swtch copies new to %esp, pops previously saved registers, and returns.

Instead of following the scheduler into swtch, let's instead follow our user processback in. We saw in Chapter 3 that one possibility at the end of each interrupt is thattrap calls yield. Yield in turn calls sched, which calls swtch to save the currentcontext in proc->context and switch to the scheduler context previously saved incpu->scheduler (2516).

Swtch (2702) starts by loading its arguments off the stack into the registers %eaxand %edx (2709-2710); swtch must do this before it changes the stack pointer and can nolonger access the arguments via %esp. Then swtch pushes the register state, creating acontext structure on the current stack. Only the callee-save registers need to be saved;the convention on the x86 is that these are %ebp, %ebx, %esi, %ebp, and %esp.Swtch pushes the first four explicitly (2713-2716); it saves the last implicitly as thestruct context* written to *old (2719). There is one more important register: theprogram counter %eip was saved by the call instruction that invoked swtch and ison the stack just above %ebp. Having saved the old context, swtch is ready to restorethe new one. It moves the pointer to the new context into the stack pointer (2720).The new stack has the same form as the old one that swtch just left—the new stackwas the old one in a previous call to swtch—so swtch can invert the sequence to restore the new context. It pops the values for %edi, %esi, %ebx, and %ebp and thenreturns (2723-2727). Because swtch has changed the stack pointer, the values restoredand the instruction address returned to are the ones from the new context.

In our example, sched called swtch to switch to cpu->scheduler, the per-CPUscheduler context. That context had been saved by scheduler's call to swtch (2478).When the swtch we have been tracing returns, it returns not to sched but to scheduler, and its stack pointer points at the current CPU's scheduler stack, not initproc'skernel stack.

---

2)

In contrast to the previous O(1) scheduler used in older Linux 2.6 kernels, the CFS scheduler implementation is not based on run queues. Instead, a red–black tree implements a "timeline" of future task execution. Additionally, the scheduler uses nanosecond granularity accounting, the atomic units by which an individual process' share of the CPU was allocated (thus making redundant the previous notion of timeslices). This precise knowledge also means that no specific heuristics are required to determine the interactivity of a process, for example.[2]

Like the old O(1) scheduler, CFS uses a concept called "sleeper fairness", which considers sleeping or waiting tasks equivalent to those on the runqueue. This means that interactive tasks which spend most of their time waiting for user input or other events get a comparable share of CPU time when they need it.

The data structure used for the scheduling algorithm is a red-black tree in which the nodes are scheduler-specific `sched_entity` structures. These are derived from the general `task_struct` process descriptor, with added scheduler elements.

The nodes are indexed by processor "*execution time*" in nanoseconds.[3]

A "*maximum execution time*" is also calculated for each process. This time is based upon the idea that an "ideal processor" would equally share processing power amongst all processes. Thus, the *maximum execution time* is the time the process has been waiting to run, divided by the total number of processes, or in other words, the *maximum execution time* is the time the process would have expected to run on an "ideal processor".

When the scheduler is invoked to run a new process, the operation of the scheduler is as follows:

1. The leftmost node of the scheduling tree is chosen (as it will have the lowest spent *execution time*), and sent for execution.
2. If the process simply completes execution, it is removed from the system and scheduling tree.
3. If the process reaches its *maximum execution time* or is otherwise stopped (voluntarily or via interrupt) it is reinserted into the scheduling tree based on its new spent *execution time*.
4. The new leftmost node will then be selected from the tree, repeating the iteration.

If the process spends a lot of its time sleeping, then its spent time value is low and it automatically gets the priority boost when it finally needs it. Hence such tasks do not get less processor time than the tasks that are constantly running.

---

4)

Processes scheduled under one of the real-time policies (SCHED_FIFO, SCHED_RR) have a sched_priority value in the range 1 (low) to 99 (high). (As the numbers imply, real-time threads always have higher priority than normal threads.)

"Real time" (for a process) refers to the scheduling algorithm, or the thinking the kernel does when it decides which process gets to run. A real time process will *preempt* all other processes (of lesser scheduling weight) when an interrupt is received and it needs to run.

A program that just accepts user input is going to go to sleep (block) while waiting for the input, or even between keystrokes (depending). Such a program does not need to have such a high scheduling priority. RT processes should *need* to run ahead of every other process on the system. This could be because the process is critical to some crucial goal, or high performance timers are needed (in which case, you'd want a real time OS, which standard Linux is *not*).

---

5)

In Linux kernel, the scheduler is invoked by periodic timer interrupt. This is called periodic scheduling which is essential for preempting tasks that have consumed more CPU cycles in order to offer other tasks on run-queue a fair chance to utilize the CPU.

The scheduler is also invoked by kernel functions that block the current task, and this allows the scheduler to decide on which task on the run-queue should run and context switch to that task.

7)

Load balancing is an expensive procedure on today's systems, both computationwise, because it requires iterating over dozens of runqueues, and communication-wise, because it involves modifying remotely cached data structures, causing extremely expensive cache misses and synchronization. As a result, the scheduler goes to great lengths to avoid executing the load-balancing procedure often. At the same time, not executing it often enough may leave runqueues unbalanced. When that happens, cores might become idle when there is work to do, which hurts performance. So in addition to periodic loadbalancing, the scheduler also invokes "emergency" load balancing when a core becomes idle, and implements some load-balancing logic upon placement of newly created or newly awoken threads. These mechanisms should, in theory, ensure that the cores are kept busy if there is work to do.

The load tracking metric. A strawman load-balancing algorithm would simply ensure that each runqueue has roughly the same number of threads. However, this is not necessarily what we want. Consider a scenario with two run queues, where one queue has some number of high-priority threads and another queue has the same number of lowpriority threads. Then high-priority threads would get the same amount of CPU time as low-priority threads. That is not what we want. One idea, then, is to balance the queues based on threads' weights, not their number.

thread does not need a whole core. To achieve this goal, CFS balances runqueues not just based on weights, but based on a metric called load, which is the combination of the thread's weight and its average CPU utilization. If a thread does not use much of a CPU, its load will be decreased accordingly.