The Scheduler in xv6

Computer Science 432 — Lecture 14 — Duane Bailey

April 11, 2022

Announcements

- Lab meeting today for Lab 6. A two-week lab thinking about context and context-switching.
- No small group meetings this week, but additional office hours.

The Dedicated Machine Abstraction

- The Big Picture.
  The *dedicated machine* abstraction allows us to pretend that we have the machine to ourselves.
  We do not.

  - Typically, there are multiple users.
  - There are processes that help to maintain the infrastructure.
  - And, of course there is the kernel.

  - We share space on the machine (caching, virtual memory, through memory managment).
  - We share time on the machine (multiplexing, through the scheduling of processes).

- Part of designing an O/S is identifying best how to support the dedicated machine abstration.

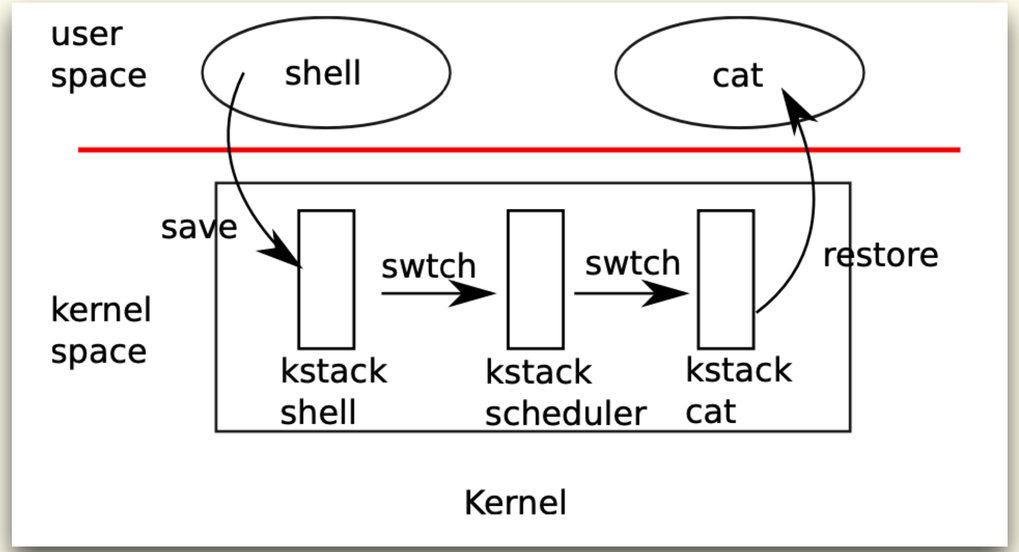
The xv6 Approach to Multiplexing

- *Time-shared* operating systems (as opposed to *real-time* operating systems [another course]):
  - Multiplex time by giving processes a fixed slice or *quantum* of time to run
    (The length is a *jiffy*, or a *tick*, or a *fuzz*.)
  - There are no "hard" deadlines; processes are switched periodically
  - Compute-bound processes use the entire quantum.
  - I/O-bound processes frequently yield the processor to allow devices to perform I/O

- Concerns & open problems
  - How do we establish the quantum?
  - How do we switch between processes?
  - How do we build a locking system that works with our approach?
  - How does multiplexing interact with multiple processors?

How Context Switching Happens

- In `xv6`, all context switches happen the same way (though for several reasons):
  - The *old* process (the one that is running out of time), switches to *its* kernel thread
  - The *old* process kernel thread switches to the *cpu-specific kernel scheduler thread*.
  - The kernel scheduler thread finds the appropriate *new* process.
  - The kernel scheduler switches to (ie. resumes) the *new* process's kernel thread.
  - The *new* process kernel thread returns to user-mode, resuming the suspended computation.



- Notice that when a process is created (how?), it starts by pretending to "resume" at the beginning of main.

Saving Registers

- What needs to be saved as part of the process's context-switch?
  - Current program counter.
  - Current stack (from stack top down). So: current stack pointer.
  - *All* general purpose registers. We make no assumptions about register saving conventions.
  - Very few `CSR` register values—most are agnostic of the process running.

- In general, the *What is context?* question is important, and differs based on the need.

Saving & Restoring Registers

- What needs to be saved as part of the process's context-switch?
  - Current program counter.
  - Current stack (from stack top down). So: current stack pointer.
  - *All* general purpose registers. We make no assumptions about register saving conventions.
  - But: remember, all paths to the kernel store registers in `p->trapframe`.
  - Very few CSR register values—most are agnostic of the process running.

- The items to be stored in a context switch are saved in the `context` structure in xv6.
- All context switching happens in the `swtch(old,new)` assembly.
  Because this is a subroutine call, the calling process makes no assumptions about caller-saved registers, so `swtch` only needs to save callee-saved registers.

The Path through the Scheduler

- The *scheduler* thread has its stack and context saved in the `cpu.context` field.
  - The scheduler context was set up at boot time, at the very last task of `main`.
  - When a process has to/wants to yield, it calls `yield()`, which grabs the `p->lock` and notes this process is `RUNNABLE`.
  - This calls `sched()`. This saves the process context and loads the scheduler context.

  - We're now in the scheduler.

  - When a new process is selected, it was selected from a pool of processes that had previously called `sched`.
    Thus: `swtch` moves us from the scheduler to the `sched` routine of the new process.
  - A return from `sched` returns us to `yield` which returns to `usertrap` which then calls `usertrapret`.

(A Note on Spinlocks)

- As we move through the context switch, the lock on the *old* process is held.
  - This lock is initially held by the process itself.
  - When the context switches, *the lock logically sticks with the cpu*.
  - The lock is released inside of the scheduler.

  - It is very important that the lock be held through the context switch. Why? Read the important details about *invariants* in §7.3.

The sleep/wakeup Synchronization

- As we noted in the last lecture, it's helpful to be able to hold a lock for long periods of time.
- `Sleeplocks` are the answer. They use the `sleep` & `wakeup` mechanism.
  - `Sleep` puts a process to sleep while it waits for an event involving a *channel*.
  - `Wakeup` is called when the event happens, and attempts to wake any sleepers.

How sleep Works

- `Sleep` is called when something a process needs is not available.
  - It is assumed that that determination was made by looking at a resource *while holding a lock*
  - The resource is called the *channel* and the lock is called the *condition lock*.
  - When `sleep` marks the process state as `SLEEPING` it releases the condition lock.
  - Observation: the resource evaluation and the resulting sleep appear atomic to wakeup-ers.
  - In most other ways, `sleep` works like `yield`: it calls `sched` to surrender the machine.

How wakeup Works

- `Wakeup` is called when a resource associated with a channel becomes available. E.g. `sleeplock`.
  - When `wakeup` is called, it scans the process table, looking for sleepers on the channel.
  - It grabs the process's `lock` and marks it `RUNNABLE`.
  - Since `RUNNABLE` processes are targets for scheduling, they will return from `sleep`.
  - When the return from `sleep`, they must re-check the resource
    It may have been given away, again, already!

Pipe Management

- Chapter 7 also discusses the management of pipes (kernel/pipe.c)
  - The locking mechanism, here, is essentially a counting semaphore
  - The resource counted is the number of bytes available in the pipe buffer

  - Writers produce (V) a data resource:
    - They write the buffer under the pipe's lock.
    - When the buffer is full, they wake the readers, and go to sleep.
    - If they write something new, they wake any readers.

  - Readers consume (P) the data resource:
    - The read the buffer under the pipe's lock.
    - When the buffer is empty, they go to sleep.
    - Whenever they successfully read, the wake any sleeping writers.

Parent & Child Relations

- Relations between parents and children are fraught with sleeping & waking

- Parents typically `wait` for children to finish.
  - When a child calls `exit`, it wakes its parent and then changes its state to `ZOMBIE`.
  - When a parent waits for a child to finish, it scans the process table looking for
    - Zombie children. If any, it picks one, and returns with exit status.
    - Otherwise, it `sleeps`, waiting for a child process to die.

  - If a parent dies early, its children are handed off to init, which is always waiting.