Scheduling Intricacies

Computer Science 432 — Lecture 15 — Duane Bailey

April 13, 2022

Announcements

- Office Hours: Today, 1-3:30pm, tomorrow 1-4pm, Friday 9-10:30am.
- No small group meetings this week.
- Lab 6 is out — due next Friday. Context and context-switching.
- Friday 2:35, Wege: Registration Information Session.

The Intricacies of Scheduling

- Many operating systems support the *dedicated machine abstraction*.
  - *Time* is a resource that must be shared by a machine's users.
  - The O/S gives each process a slice of time.
  - The length of this *quantum* is a parameter that can be tuned.
  - Dependent heavily on the types of jobs—the *job mix*—expected.

- We'll look at scheduler designs for xv6, Linux, and VMS. And Windows.

The xv6 Scheduler: Simplicity

- We've spent a bit of time thinking about scheduling in xv6:
  - Goal is a *teaching* operating system
  - Missing aspects: no users, no time, no challenging computation
  - Hand-rolled context switching (assembly)
  - Next selected process is simply the next process that is RUNNABLE

  - Processes are scheduled through *preemption*:
    - An external force—machine mode timer—stops ongoing processing

- Recall:
  - 3 cores, ea. w/scheduler
  - Scheduler runs forever
  - Dedicated thread
  - Every process quantum
    - Begins at swtch call
    - Ends at swtch call
  - Timer dictates quantum
  - Interval set to 100ms
  - No priority system
- *Preemptive Round-Robin*
  - Simple & fair

```
// Per-CPU process scheduler.
// Each CPU calls scheduler() after setting itself up.
// Scheduler never returns.  It loops, doing:
//  - choose a process to run.
//  - swtch to start running that process.
//  - eventually that process transfers control
//    via swtch back to the scheduler.
void
scheduler(void)
{
  struct proc *p;
  struct cpu *c = mycpu();

  c->proc = 0;
  for(;;){
    // Avoid deadlock by ensuring that devices can interrupt.
    intr_on();

    for(p = proc; p < &proc[NPROC]; p++) {
      acquire(&p->lock);
      if(p->state == RUNNABLE) {
        // Switch to chosen process.  It is the process's job
        // to release its lock and then reacquire it
        // before jumping back to us.
        p->state = RUNNING;
        c->proc = p;
        swtch(&c->context, &p->context);

        // Process is done running for now.
        // It should have changed its p->state before coming back.
        c->proc = 0;
      }
      release(&p->lock);
    }
  }
}
```

Lab 6—Little Thread Library Scheduler

- Threading library manages a small collection of threads
  - Goal is the very simplest non-preemptive scheduling.
  - Hand-rolled context switching code (assembly).
  - Each thread must voluntarily surrender the machine
    - yield & fini
    - these methods *initiate* a scheduler that runs for a short time.
  - The scheduler (you will write) must find another RUNNABLE thread
  - You want the process selection to be fair:
    - The current RUNNABLE process should be choice of last resort

- Recall:
  - 10 threads, one core
  - Scheduler runs quickly
  - Every thread run:
    - Ends at a twitch call
    - Resumes in that call
  - Friendly yields important
  - Similar to python yield
- *Non-preemptive Round-Robin*
  - Flexible & fair

```
/*
 * schedule:
 * This routine looks for another RUNNABLE thread and twitches to it.
 * If more than one routine is RUNNABLE, use an approach that leads to
 * fair behavior: Every thread should get a chance to run, occasionally.
 * It is possible that the current thread runs again.
 * If there are no threads remaining, twitch to the main thread.
 */
static void
schedule(void)
{
  // ADD CODE HERE
}
```

```
/*
 * yield:
 * For the main thread: leave state (as MONITOR) and schedule a new thread.
 * For other threads: change state to RUNNABLE and schedule a new thread.
 */
void
yield(void)
{
  // ADD CODE HERE
  schedule();
}
```

```
/*
 * fini:
 * For the main thread: leave state (as MONITOR) and schedule a new thread.
 * For other threads: change state to FREE and schedule a new thread.
 */
void
fini(void)
{
  // ADD CODE HERE
  schedule();
}
```

Observations

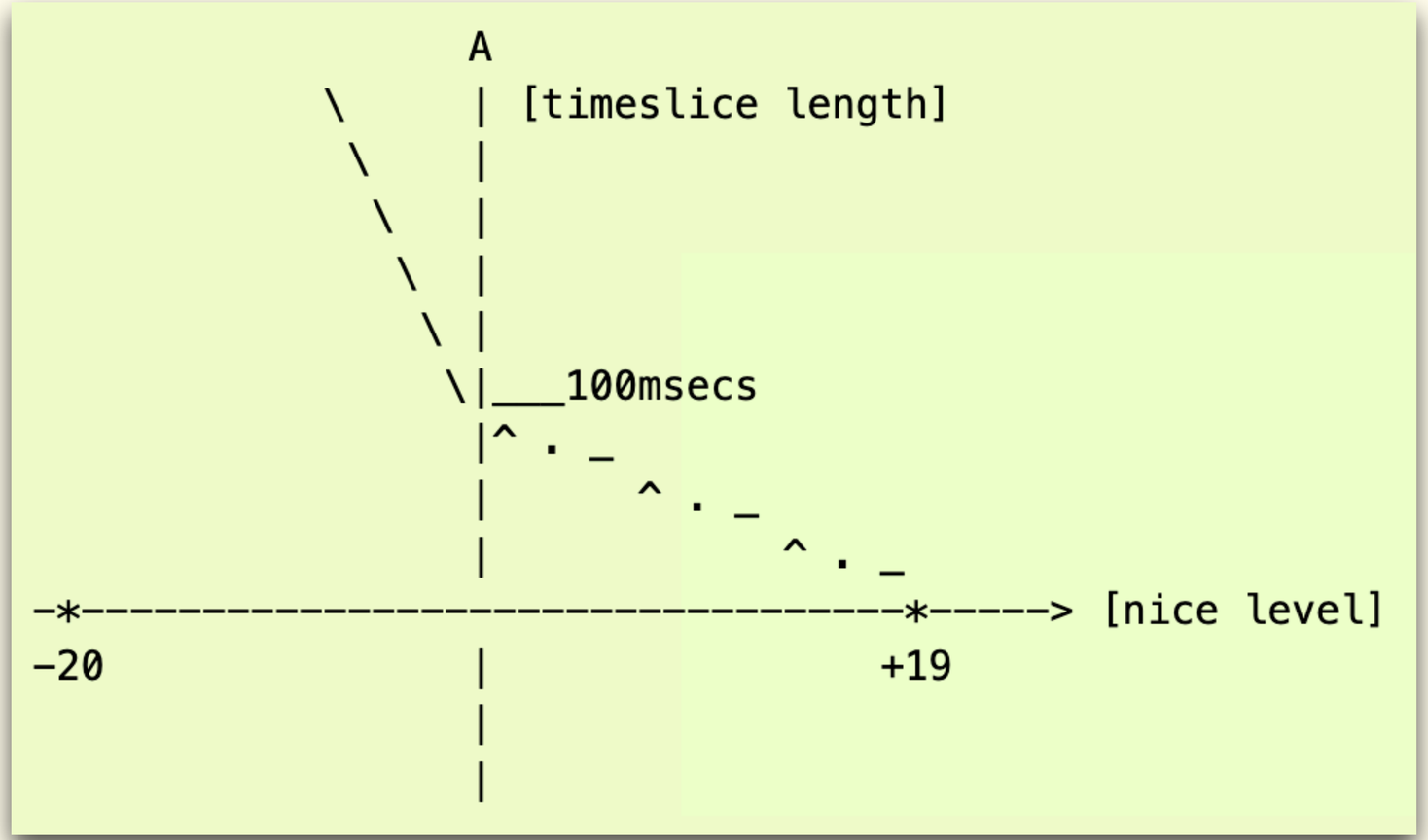
- The approach is to simplify the *concept* of scheduling:
  - There is no explicit *run queue* structure:
    - Just a list of processes / threads
    - The queue is the collection of those members that are RUNNABLE
  - Any fairness comes from carefully observing a round-robin ordering
  - Scheduling is subtly different so queue-ordering is are also subtle:
    - In xv6, scheduler is continuous, with nested loops imposing order
    - In threading, the scheduler is called many times, with a search loop

Typical Unix Details— the Linux Scheduler

- Unix scheduling must address a number of different demands:
  - Goal is handling a mix of job types.
  - Hand-rolled context switching code (assembly).
  - Each process has an associated *scheduling priority*
    - priorities range from -20 (favorable) to 20

  - You can adjust the priorities by being *nice*:
    - "Nicing" a process is the process of adjusting its priority.
    - Only the root can nice by negative values.

- Unix quanta:
  - prio 0: 100ms jiffy
  - negative priorities
    - longer jiffies
    - slope is significant
    - target: real-time
      e.g. audio
  - positive priorities
    - shorter jiffies
    - shallow slope
    - target: compute bound
- *Preemptive Priority Scheduler*
  - Resilient for job mixes
  - Always evolving.



kernel.org

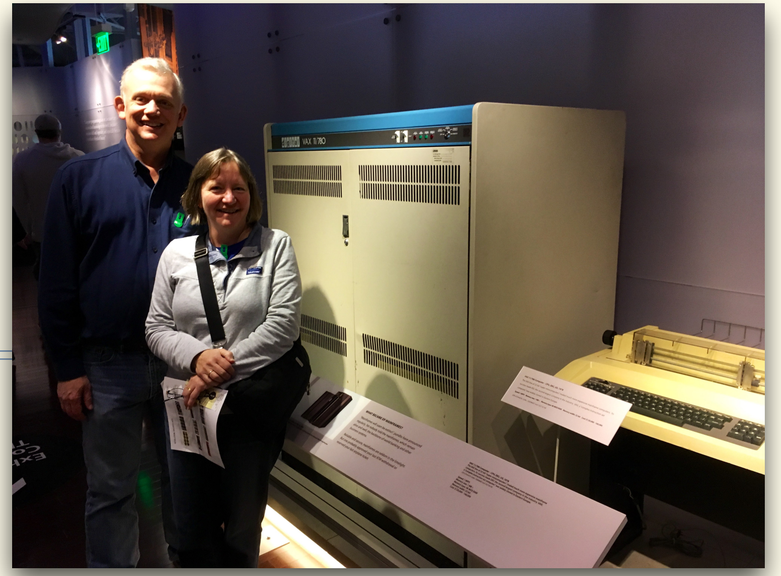
Details

- The Linux scheduler is called the *sched_other* or *normal* scheduler.
  - The highest priority (niceness) processes get the largest time slices.
  - Each process has a *counter* that keeps track of its remaining slice time.

  - There is a single *run queue* of processes that are runnable.
  - The queue is ordered by the time slice counter.

  - When the runnable processes run out of time
    - Each runnable process gets its counter reset.
    - Tasks not in the run queue get counter reset *plus half of current counter.*
    - Thus interactive tasks (which block on I/O) get a priority boost.

Versatile O/S Scheduling—VMS

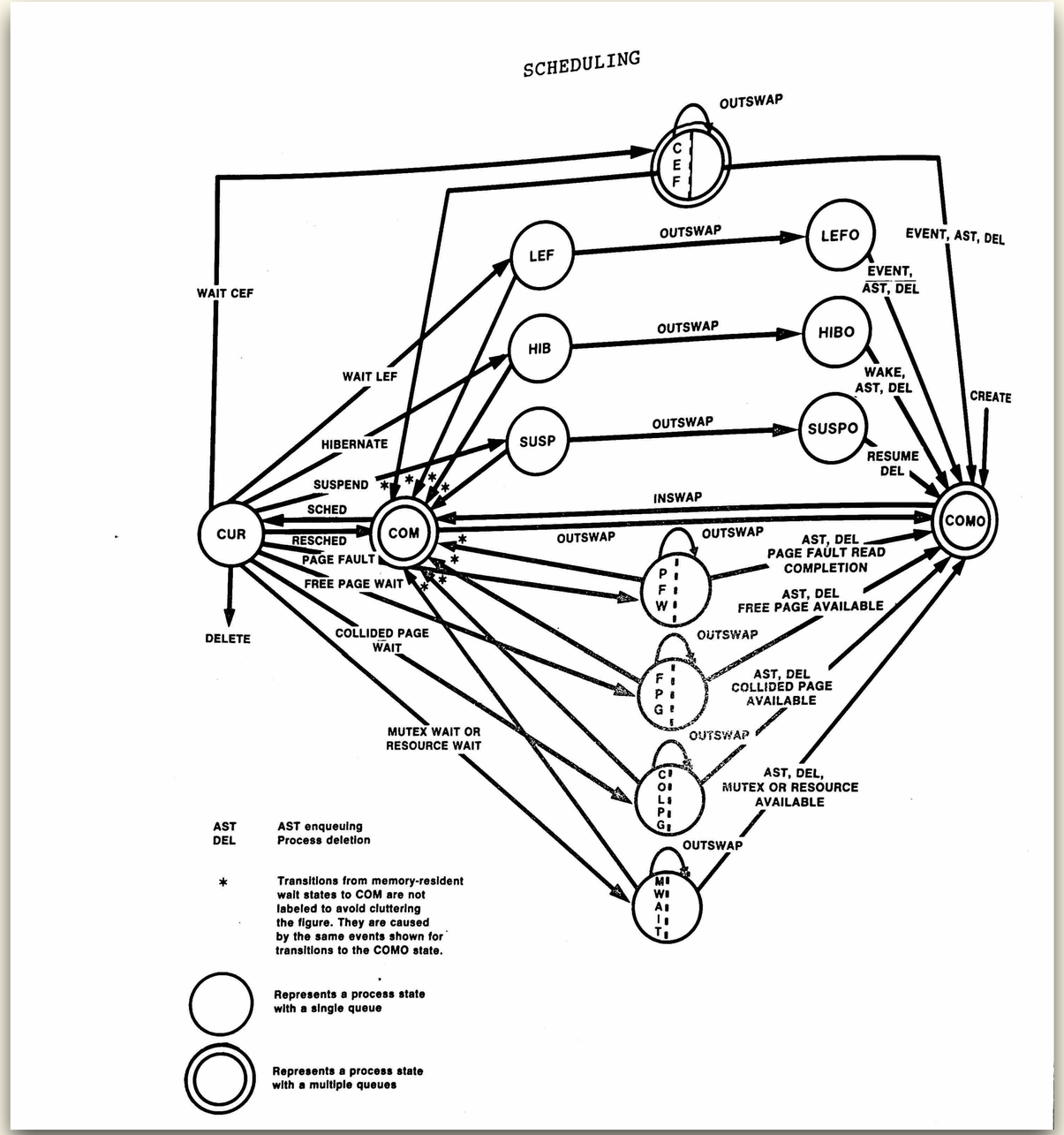


- VMS: a very popular commercial O/S from DEC
  Chief engineer: David Cutler.
  - Runs on specific machine: DEC VAX, the SPEC definition CPU power 1
    - CISC with dedicated context switching instructions
  - Leveraged significant experience with real-time O/S
  - Processes have *dynamic* process priorities.
    - Priorities range from 0 to 31
    - When process computes, its priority drops
    - When a process yields, its priority increases

- VMS priority cycling:
  - 0-15 are user processes
    - Typical: 4-8
    - NULL: 0
  - 16-31 are *real-time*
    - Run for many quanta only preempted by higher priority process
  - *Round-robin* within same prio.

- Load & Save Process Context
  - *Complex* instructions
  - Each moves 2 dozen registers!
  - SVPCTX in *many* places.
  - LDPCTX used *once*.

- *Preemptive Priority Scheduler*
  - Peak of scheduling complexity



Details

- Processes are primarily identified by their priority:
  - There are 32(!) different run queues: 0 has just the null/idle
  - Accessed in round-robin order.
  - Law of the Jungle: when a higher priority process is runnable it preempts.

  - Priorities 0-15 are time-sliced for *normal* time-sharing processes:
    - Rescheduling (using entire quantum) causes a lowering of priority
    - Waiting processes get a boost in priority.
  - Priorities 16-31 are for *real time* processes:
    - Priorities never change
    - Governed entirely by Law of the Jungle

Windows Scheduling

- Windows is another Dave Cutler design….
- Very similar to VMS:
  - 32 priorities (0 is lowest, 31 highest)
  - There are sub-priorities within each main priority.
  - No notion of real-time processes (this is good for a desktop machine)
    - But: 0-15 can get boosts, while 16-31 never do.
  - Generally, it appears that “Law-of-the-Jungle” style scheduling applies

- A fairly complex preemptive scheduler for interactive computing.