CS444 Lab 3

Implementing Simple Scheduler

What is a Scheduler?

- Decides how often processes get to run
- Key component of how OS asserts dominion of the hardware
- Quantized: runs no more often than system clock
- Apparent to user: scheduler determines how “responsive” the system feels
  - (Side note: Windows used to have a *very* messy scheduler. Anyone notice how older Windows machines, when one application was unresponsive, the whole machine felt “slow,” but it’s gotten better recently (Windows 8+)? This is due to very basic algorithmic improvements in the IO scheduler specifically, among other things.)

Different Problem Domains

- Schedulers vital to system responsiveness and performance
- Many algorithms solve many different problems
    - Batch processing: users wait for jobs to complete: minimize context switches to improve performance (mainframe)
    - Interactive: users actively interact with processes: maximize context switch opportunities to appear responsive (workstation)
    - Power-aware: limited resource environment: maximize scheduler flexibility (at cost of complexity and adjustment time) to improve resource availability (mobile phone; active area of current research/improvement)
    - Soft real-time: process needs to run continually, with guaranteed timing, but occasionally this can be sacrificed for the sake of the system (graphics processor, think “minimum FPS” for “guaranteed timing”)
    - Hard real-time: process needs to run continually, with guaranteed timing, and failure to do so will result in failure to operate (flight control and avionics package of aircraft; inconsistent scheduling means software sees outdated readings meaning plane flies into ground: human death results)

Xv6's Scheduler

- In `proc.c`, there are a few functions related to scheduling
- `scheduler` is the entrypoint to the scheduler
- `scheduler`'s job is to switch context to a process, let it run, then resume
- Currently, a very simple round-robin implementation

```
270
271   //PAGEBREAK: 42
272   // Per-CPU process scheduler.
273   // Each CPU calls scheduler() after setting itself up.
274   // Scheduler never returns.  It loops, doing:
275   //  - choose a process to run
276   //  - swtch to start running that process
277   //  - eventually that process transfers control
278   //      via swtch back to the scheduler.
279   void
280   scheduler(void)
281   {
282     struct proc *p;
283     int ran;
284
285     for(;;){
286       // Enable interrupts on this processor.
287       sti();
288       // Whether or not a process actually ran in this pass
289       ran = 0;
290
291       // Loop over process table looking for process to run.
292       acquire(&ptable.lock);
293       for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
294         if(p->state != RUNNABLE)
295           continue;
296
297         // Found a process!
298         ran = 1;
299
300         // Switch to chosen process.  It is the process's job
301         // to release ptable.lock and then reacquire it
302         // before jumping back to us.
303         proc = p;
304         switchuvm(p);
305         p->state = RUNNING;
306         swtch(&cpu->scheduler, p->context);
307         switchkvm();
308
309         // Process is done running for now.
310         // It should have changed its p->state before coming back.
311         proc = 0;
312       }
313       release(&ptable.lock);
314
315
316     // Didn't find a process this pass; halt (until the next interrupt).  We're
317     // going to hope(!) that there's a wait channel out there that will
318     // eventually cause this processor to wake. (If not, this hangs forever.)
319     // This guarantee is usually satisfied by the line clock, which the
320     // scheduler also uses.
321     if(!ran)
322       hlt();
323     }
324   }
```

Task

- Implement per-process priorities in the scheduler
- Priority is a common scheduler criterion (“task priority” on Windows, “nice” factor on everything else)
- To implement, a few things will be needed
    - Add a priority field to structure for each process
        - Processes start with priority 50, ranging from 0 to 200 inclusive
        - Processes with the same priority are scheduled round-robin
        - 0 is the highest priority; 200 is the lowest
    - Add a system call that takes one argument (`setpriority`)
        - Set current process priority to given value
            - If the new priority is lower than old, yield
    - Finally, make `sched` respect priorities: modified algorithm

Modified Algorithm

- Loop forever:
  - Lock the process table
  - Loop circularly through each process, starting at last scheduled
    - If process runnable, check if priority lower than last lowest
      - Save process id and new lowest priority
- Switch context to saved process
- Unlock the process table
- If no process ran, halt (until interrupt)

Necessary Changes

- `syscall.h`: define new syscall number
- `user.h`: declare userspace syscall function
- `usys.S`: implement userspace syscall function
- `syscall.c`: add handler to syscall table
- `sysproc.c`: implement syscall behavior
- `proc.h`: add priority field to PCB
- `proc.c`: set default priority in `allocproc`, alter scheduler to implement modified algorithm