CS153
Aaron Sigal
Lab 1 Report

Overview:

- Modified the scheduler to use a round-robin system.
- Added support for priority and priority aging.
- Added support for performance-metric tracking (bonus part 3)

Infrastructure Changes

```
int             wait(int* status);                   // Lab1: updated
int             waitpid(int pid, int *status, int options); // Lab1: Added to enable waitpid
int             setpriority(int priority);           // Lab2: Added to enable setpriority
```

defs.h: necessary system-call infrastructure for setpriority

```
// Per-process state
struct proc {
  uint sz;                     // Size of process memory (bytes)
  pde_t* pgdir;                // Page table
  char *kstack;                // Bottom of kernel stack for this process
  enum procstate state;        // Process state
  int pid;                     // Process ID
  struct proc *parent;         // Parent process
  struct trapframe *tf;        // Trap frame for current syscall
  struct context *context;     // swtch() here to run process
  void *chan;                  // If non-zero, sleeping on chan
  int killed;                  // If non-zero, have been killed
  struct file *ofile[NOFILE];  // Open files
  struct inode *cwd;           // Current directory
  char name[16];               // Process name (debugging)
  int exitstatus;              // Process exit status

  int priority;               // Lab 2: The priority value. Range is 0-31

  int starttime;               // Lab 2: Tracks start time (bonus points)
  int waittime;                // Lab 2: Tracks waited time (bonus points)
};
```

proc.h: proc struct now has fields for tracking performance metrics and priority

```c
np->starttime = ticks; // Lab 2: Updates start time of new process
```

Proc.c, fork(): forked procs store start time

```c
int elapsedtime = ticks - curproc->starttime;                // Lab2: Calculate elapsed time since start
cprintf("Turnaround (elapsed) time: %d %s\n", elapsedtime, "ticks."); // Lab 2: Print (bonus)
cprintf("Wait time: %d %s\n", curproc->waittime, "ticks.");          // Lab 2: Print (bonus)
```

proc.c, exit(): exiting procs report their performance metrics

```c
// Lab 2: Sets the priority of a process. This is a system call.
int setpriority(int priority) {

  struct proc *curproc = myproc(); // Grab the current proccess

  if (!curproc) { // Check if we failed to grab.
    return -1; // Terminate
  }

  acquire(&ptable.lock);          // Get the semaphore to modify the priority of the current proccss
  curproc-> priority = priority; // Modify the priority of the current proccess
  release(&ptable.lock);          // Release the lock

  return 0;

}
```

proc.c: Definition of the setpriority system calls

```c
extern int sys_setpriority(void); // Lab 2: Added to enable setpriority
```

syscall.c: necessary system call infrastructure

```c
#define SYS_setpriority 5 // Lab 2: Added to enable setpriority
```

syscall.h: necessary system-call infrastructure

```c
// Lab 2: Created to enable setpriority
int
sys_setpriority(void)
{
  char* priority;
  if(argptr(0, &priority, 8) < 0) return -1;

  return setpriority((int)priority);
}
```

sysproc.c: System call handler for setpriority

```c
    struct proc *s;
    //Lab 2: Increase the priority of all the waiting processes. LOWER IS BETTER
    for (s = ptable.proc; s < &ptable.proc[NPROC]; s++) {
      if (s -> priority > 0) s -> priority = s -> priority - 1;
    }
    //Lab 2: Set p to the procceess with the highest priority (the lowest value) that is also ready
    for (s = ptable.proc; s < &ptable.proc[NPROC]; s++) {
      if (s -> state == RUNNABLE && s -> priority < p -> priority) p = s;
    }

    for (s = ptable.proc; s < &ptable.proc[NPROC]; s++) {
      if (s->state != RUNNABLE) continue;

        if (s != p) s -> waittime = ticks - s -> starttime; // Ensure that we aren't changing the currently-

    }

    // Switch to chosen process.  It is the process's job
    // to release ptable.lock and then reacquire it
    // before jumping back to us.
    c->proc = p;
    switchuvm(p);
    p->state = RUNNING;

    swtch(&(c->scheduler), p->context);
    switchkvm();

    if (s -> priority <= 29) {
        p -> priority += 2; // Lab 2: Decrease the priority the running proccss.
                            // By 2 since only doing 1 would cause it to
                            // slingshot back and forth, defeating the purpose of aging.
    } else if (s-> priority < 31) {
      p -> priority += 1;
    }
```

proc.c, scheduler(): 3 for loops pass over the proc table to raise the priority of the waiting procs, set the current proc to the highest-priority waiting proc, and update the wait-times of all waiting procs. This is done in order. Finally, the priority of the running proc is reduced.

```c
int setpriority(int); // Lab 2: Added declaration
```

User.h: necessary system-call infrastructure

```
sb: size 1000 nblocks 941 ninodes 200 nlog 30 logstart 2 inodestart 32 bmap start 58
init: starting sh
$ lab2 2

 This program tests the correctness of your lab#2

  Part c) testing that wait and waitpid interact correctly with the roundrobin scheduler:

 The is child with PID# 4
Turnaround (elapsed) time: 1 ticks.
Wait time: 0 ticks.

 The is child with PID# 5
Turnaround (elapsed) time: 2 ticks.
Wait time: 1 ticks.

 The is child with PID# 6
Turnaround (elapsed) time: 0 ticks.

 The is child with PID# 7
Turnaround (elapsed) time: 1 ticks.
Wait time: 0 ticks.

 The is child with PID# 8
Turnaround (elapsed) time: 2 ticks.
Wait time: 1 ticks.
Wait time: 2 ticks.

 This is the parent: Now waiting for child with PID# 7

 This is the partent: Child# 7 has exited

 This is the parent: Now waiting for child with PID# 5

 This is the partent: Child# 5 has exited

 This is the parent: Now waiting for child with PID# 6

 This is the partent: Child# 6 has exited

 This is the parent: Now waiting for child with PID# 4

 This is the partent: Child# 4 has exited

 This is the parent: Now waiting for child with PID# 8

 This is the partent: Child# 8 has exited
Turnaround (elapsed) time: 35 ticks.
Wait time: 7 ticks.
$
```

Test, instance 1

```
$ lab2 2

 This program tests the correctness of your lab#2

  Part c) testing that wait and waitpid interact correctly with the roundrobin scheduler:


 The is child with PID# 11
Turnaround (elapsed) time: 1 ticks.
Wait time: 0 ticks.

 The is child with PID# 12
Turnaround (elapsed) time: 1 ticks.
Wait time: 1 ticks.

 The is child with PID# 13

 The is child with PID# 14
Turnaround (elapsed) time: 2 ticks.
Wait time: 1 ticks.
 The is child with PID# 10
Turnaround (elapsed) time: 2 ticks.
Wait time: 2 ticks.
Turnaround (elapsed) time: 2 ticks.
Wait time: 2 ticks.

 This is the parent: Now waiting for child with PID# 13

 This is the partent: Child# 13 has exited

 This is the parent: Now waiting for child with PID# 11

 This is the partent: Child# 11 has exited

 This is the parent: Now waiting for child with PID# 12

 This is the partent: Child# 12 has exited

 This is the parent: Now waiting for child with PID# 10

 This is the partent: Child# 10 has exited

 This is the parent: Now waiting for child with PID# 14

 This is the partent: Child# 14 has exited
Turnaround (elapsed) time: 32 ticks.
Wait time: 5 ticks.
$
```

Test, instance 2

The two above screenshots demonstrate the performance-tracking capabilities of my lab as well as the functionality of the round-robin scheduler. Not only does each processes get a turn, it doesn't favor any process specifically. Note that the **parent** processes necessarily take longer than their children because **they are using waitPID on them.** This was to verify that round-robin scheduling plays nicely with my wait, waitPID, and exit implementations.