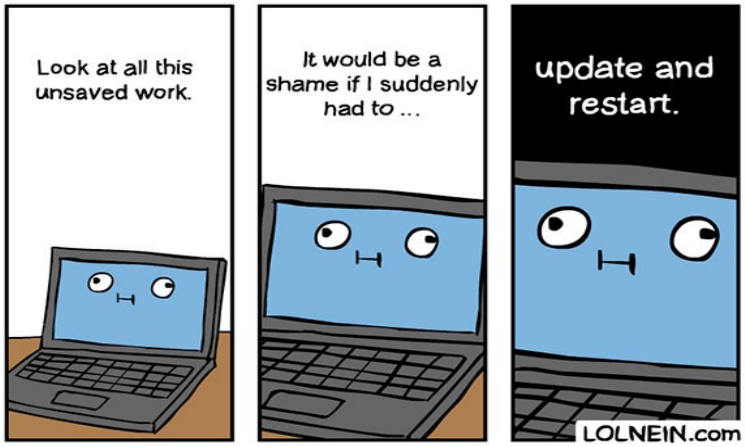
CS 370 – Project #4, Part A

Purpose: Become familiar with operating system process scheduling and process context switching through xv6
Points: 200

Introduction:

Any operating system is likely to run with more processes than the computer has CPUs, a plan is needed to time-share the CPUs among the processes. Ideally the sharing should be transparent to user processes.



Our first step will be to update a system call and implement an additional system call to obtaining basic OS scheduler performance information (average run time and average wait time).

Resources:

The following resources provide more in depth information regarding **xv6**. They include the **xv6** reference book, tools for installing, and guidance to better understand **xv6**.

1. **xv6** Reference Book: https://pdos.csail.mit.edu/6.828/2021/xv6/book-riscv-rev2.pdf
2. Lab Tools Guide: https://pdos.csail.mit.edu/6.828/2022/tools.html
3. Lab Guidance: https://pdos.csail.mit.edu/6.828/2022/labs/guidance.html

Project:

Complete the following actions.

- Add the specified new system call and associated functions.
- Add the user-level scheduler testing program.

Submission:

When complete, submit:

- A screen shot of the scheduler test program output (PNG file) via the class web page (assignment submission link) by class time.
  - The full source code will be submitted in a subsequent Part B.
- Submit a copy of the `schTest.c` program.

Submissions received after the due date/time will not be accepted.

Project Setup:

Project #4 will be done entirely in **xv6**. To get started with this project first follow the instructions below:

```
1. git clone https://github.com/mit-pdos/xv6-riscv.git proj4
2. cd proj4
3. git fetch
4. make clean
5. make qemu
```

Note: **xv6** may not compile initially as source code is missing that you will have to add throughout this project.

xv6 Scheduling Background:

Every process in **xv6** is stored in a Process Control Block (PCB) which is defined as **`struct proc`** (**`kernel/proc.h`**). An array of **`struct proc`** of size **`NPROC`** (64) is defined near the top in **`kernel/proc.c`**. This array keeps track of all **xv6** processes; at most 64 processes. Each entry in the array is a different process and unused entries are marked with the process state **`UNUSED`**. For security purposes, all functions that can access the array of processes are implemented in **`kernel/proc.c`**. No other file has access to the processes array. If you need to access the processes array, you must write a function in **`kernel/proc.c`**. You may create a system call that calls a function in **`kernel/proc.c`** (see fork/wait/exit system calls).

In **xv6**, a basic Round-Robin (RR) scheduler is implemented by default (**`kernel/proc.c`**). The RR scheduler assigns time slices to each process in equal portions. The default time slice in **xv6** is 100 ticks. Once a process begins execution it is limited by the time slice, if the process does not complete before the end of the slice the process is interrupted and placed at the end of the ready queue. The next process at the front of the ready queue is then selected and begins executing for a time slice.

Switching from one process to another involves saving the old process's CPU registers, and restoring the previously-saved registers of the new process; saving and restoring the stack pointer and program counter means the CPU will switch stacks and switch what code it is executing; this is called a ***context switch***.

The ***context switch*** function performs the saves and restores for a process context switch. This function doesn't directly know about processes; it just saves and restores register sets, called **`contexts`**. When it is time for a process to give up the CPU, the context switch occurs to saving the context and returning to the scheduler context. Each context is contained in a **`struct context`** (**`kernel/proc.h`**), itself contained in a process's **`struct proc`** or a CPU's **`struct cpu`**. Context switch takes two arguments: **`struct context *old`** and **`struct context *new`**. It saves the current registers in old, loads registers from new, and returns.

Implement New System Call – `waitStats()`

The existing **`wait()`** system call waits for a child process to complete. You should review the existing wait system call to obtain a basic idea of how it works. From there, we will add a new version of the wait system call, **`waitStats()`**, that will provide some basic information about the total run time and wait time for that process.

FILE: **`proc.h`**

In order to track each process, we will add some new variables into the process control block. These variables are as follows:

```
int  timeOfCreation;
uint64  runTime;
uint64  startTime;
uint64  numScheduled;
uint64  sleepTime;
uint64  totalRunTime;
uint64  endTime;
```

FILE: **`sysproc.c`**

We will add two new functions as follows:

```
// return how many clock tick interrupts have occurred since start.
uint64 sys_uptime(void)
{
  uint xticks;

  acquire(&tickslock);
  xticks = ticks;
  release(&tickslock);
  return xticks;
}

uint64 sys_waitStats(void)
{
  uint64    addr, addr1, addr2;
  uint wtime=0, rtime=0;

  argaddr(0, &addr);
  argaddr(1, &addr1);
  argaddr(2, &addr2);

  int ret = waitStats(addr, &wtime, &rtime);
  struct proc* p = myproc();
  if (copyout(p->pagetable, addr1,(char*)&wtime, sizeof(int)) < 0)
    return -1;
  if (copyout(p->pagetable, addr2,(char*)&rtime, sizeof(int)) < 0)
    return -1;
  return ret;
}
```

FILE: **trap.c**
In the clockintr() function, insert a call to the **updateTime()** function. This must be done inside the lock, after the **ticks** variable has been updated.

FILE: **syscall.c** and **syscall.h**
Add the new syscall for **waitStats()** in a similar manner as before.

FILE: **usys.pl** and **user.h**
Add the new syscall for **waitStats()** in a similar manner as before.

FILE: **proc.c**
Add the following two functions to support the new system call.

```
void updateTime() {
  struct proc *p;

  for (p = proc; p < &proc[NPROC]; p++) {
    acquire(&p->lock);

    if (p->state == RUNNING) {
      p->runTime++;
      p->totalRunTime++;
    }

    if (p->state == SLEEPING)
      p->sleepTime++;

    release(&p->lock);
  }

}

int waitStats(uint64 addr, uint* rtime, uint* wtime) {
  struct proc *np;
  int havekids, pid;
  struct proc *p = myproc();

  acquire(&wait_lock);

  for(;;){
    // Scan through table looking for exited children.
    havekids = 0;
    for(np = proc; np < &proc[NPROC]; np++) {
      if(np->parent == p){
        // make sure the child isn't still in exit() or swtch().
        acquire(&np->lock);

        havekids = 1;
        if(np->state == ZOMBIE) {
          // Found one.
```

```
            pid = np->pid;
            *rtime = np->totalRunTime;
            *wtime = np->endTime - np->timeOfCreation - np->totalRunTime;
            if(addr != 0 && copyout(p->pagetable, addr, (char *)&np->xstate,
                                    sizeof(np->xstate)) < 0) {
              release(&np->lock);
              release(&wait_lock);
              return -1;
            }
            freeproc(np);
            release(&np->lock);
            release(&wait_lock);
            return pid;
          }
          release(&np->lock);
        }
      }

      // No point waiting if we don't have any children.
      if(!havekids || p->killed) {
        release(&wait_lock);
        return -1;
      }

      // Wait for a child to exit.
      sleep(p, &wait_lock);  //DOC: wait-sleep
    }
  }
```

In addition, the new variables added the process control block need to be initialized to 0 in the **`allocproc()`** function, except for the **`timeOfCreation`** which should be set to **`ticks`**. Finally, the **`endTime`** variable must be set in the **`exit()`** function (to **`ticks`**) which represents the final ending time for that process.

Summary of Updated Files

As was done in project 1, you will be updating the following files:

`Makefile`
`user/user.h`
`user/usys.pl`
`kernel/syscall.h`
`kernel/syscall.c`
`kernel/sysproc.c`
`kernel/defs.h`
`kernel/proc.h`
`kernel/proc.c`
`kernel/trap.c`

Pay close attention to user space vs kernel space.

Scheduler Test

Once the new system call has been added, create a small user-space program **`schTest`** that will create a set of children using ***fork()***. The children will be used to simulate CPU-bound and I/O bound processes. The number of CPU-bound and I/O processes should be defined parameters. The schTest program should take two command line argument "**`-c <int>`**" and "**`-i <int>`**", specifying the number of CPU bound processes and I/O bound processes, respectively.

To create a CPU-bound process, we will imitate useful work by iterating as follows:

```
for (volatile int i = 0; i < 1000000000; i++)
     {}                                          // CPU bound process
```

To create an I/O-bound process, we will imitate useful I/O work by sleeping as follows:

```
sleep(200);
```

Each process should print:

```
printf("\nProcess %d finished", n);
```

Before the final **`exit()`** system call.

Normally, the parent process would execute a **`wait()`** system call for each child to wait for the child to complete. This function blocks the calling process until one of its child processes exits. We will need to call wait for each child created.

Thw new **`waitStats()`** will return some basic scheduler statistics; average run time and average wait time. It will be called as follows:

```
for(n=MAX_PROC; n > 0; n--) {
     if(waitStats(0, &rTime, &wTime) >= 0) {
          totRunTime += rTime;
          totWaitTime += wTime;
     }
}
```

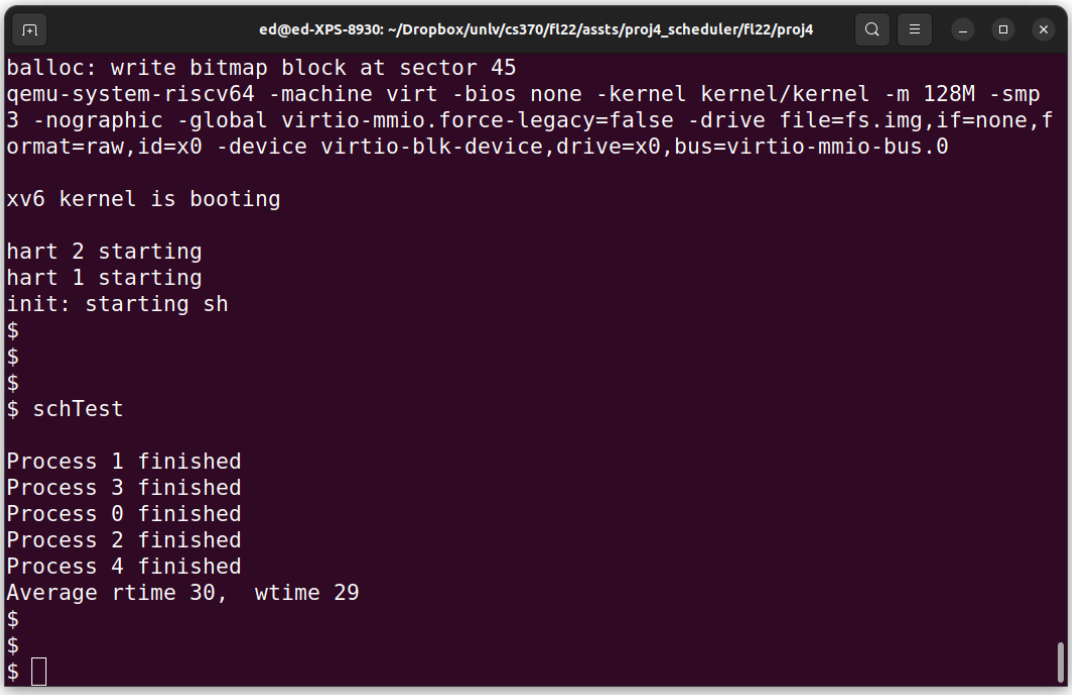
After the loop completes, the average run time and average wait time should be printed.

Be sure to use parameters for the number of processes that will be CPU-bound and the number of processes that will be I/O-bound and the maximum number of processes (CPU-bound plus I/O bound). The specific choice for the appropriate total number of processes will vary between machines and you should try a number of different values.

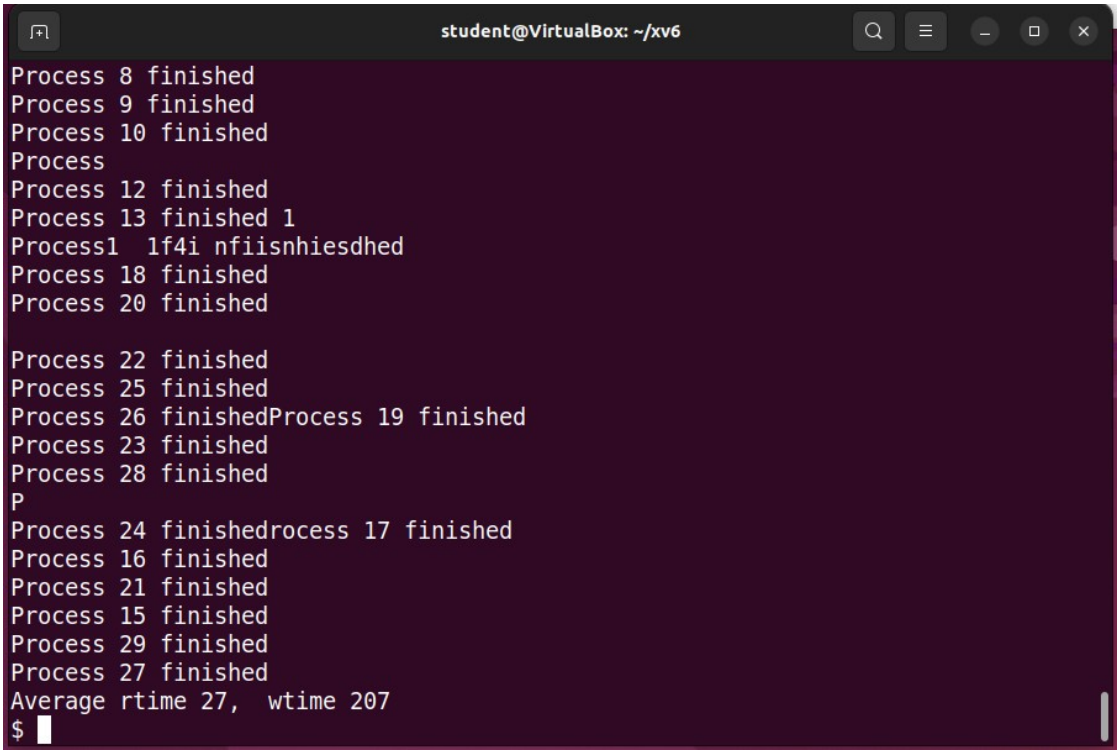
As before, the **`makefile`** will need to be updated for the new user-level program.

Example Execution

Below is an example with 5 CPU-bound processes and no I/O bound processes.



Below is an example with a mix of CPU-bound and I/O-bound processes.



The average run time and average wait times will vary significantly between different machines.