Initial report – phase 2:

Pouya Sharifi 9831321

Ali Babaloo 9831322

https://github.com/pooya-sharifi/xv6_OS

1 – what is the default scheduler that is implemented in xv6?

The xv6 scheduler implements a simple scheduling policy, which runs each process in turn. This policy is called round robin. basically, it loops through all the process which are available to run (marked with the RUNNABLE) state and give access to CPU at each one of them one at a time.

2 – explain xv6 scheduling policy:

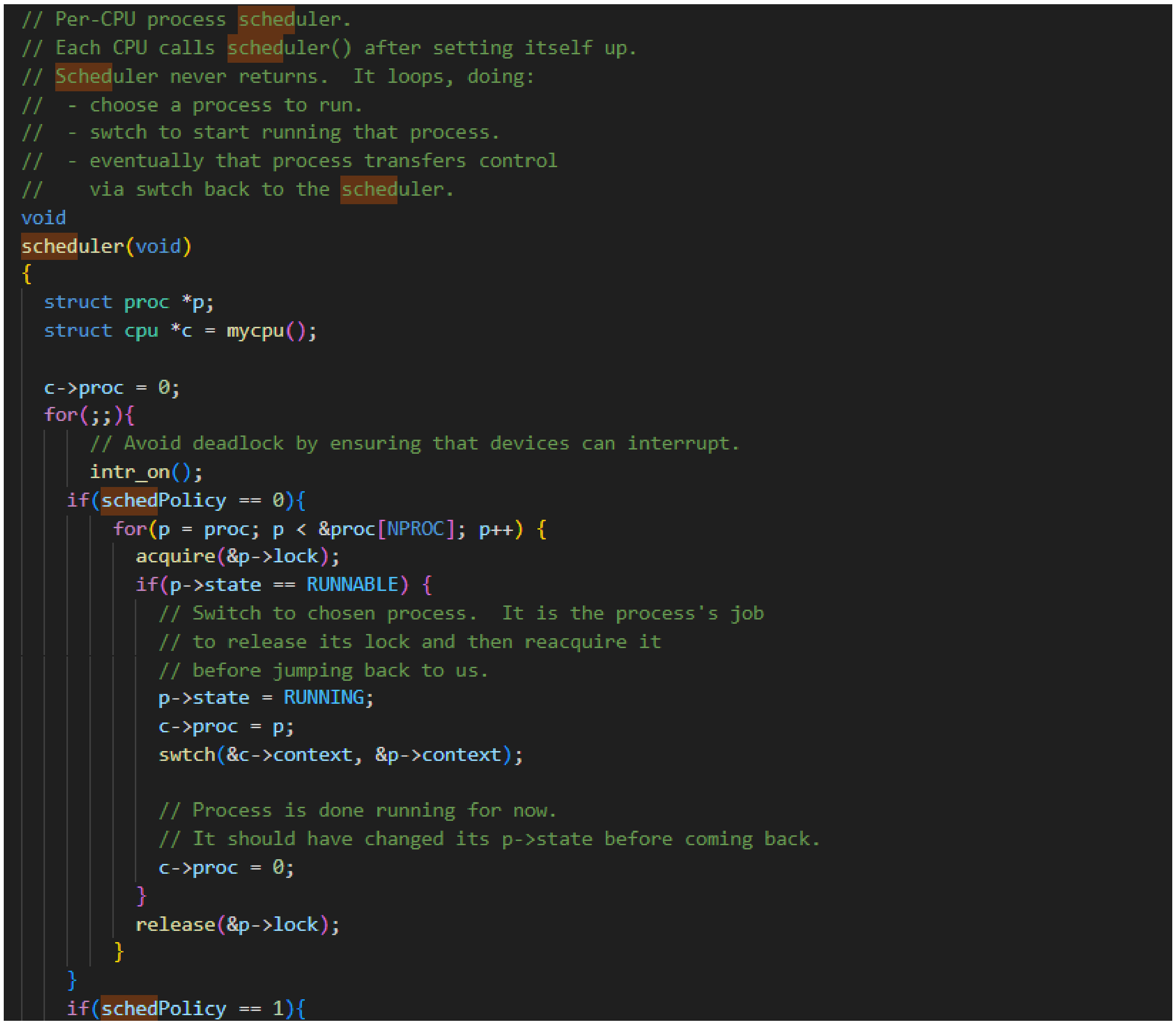
We explained main.c and main function in detail in phase 1 report. After the first process is created it calls scheduler function (. /kernel/proc.c). The scheduler loops over the process table looking for a runnable process, one that has p->state == RUNNABLE. Once it finds a process, it sets the per-CPU current process variable c->proc, marks the process as RUNNING, and then calls swtch(./kernel/proc.c) to start running it.

This function is in charge of choosing which process to run next. A process that wants to give up the CPU must acquire its own process lock p->lock, release any other locks it is holding, update its own state (p->state), and then call sched. You can see this sequence in yield (./kernel/proc.c), sleep and exit. sched double-checks some of those requirements and then checks an implication: since a lock is held, interrupts should be disabled. Finally, sched calls swtch to save the current context in p->context and switch to the scheduler context in cpu->context. swtch returns on the scheduler's stack as though scheduler's swtch had returned (kernel/proc.c). The scheduler continues its for loop, finds a process to run, switches to it, and the cycle repeats.

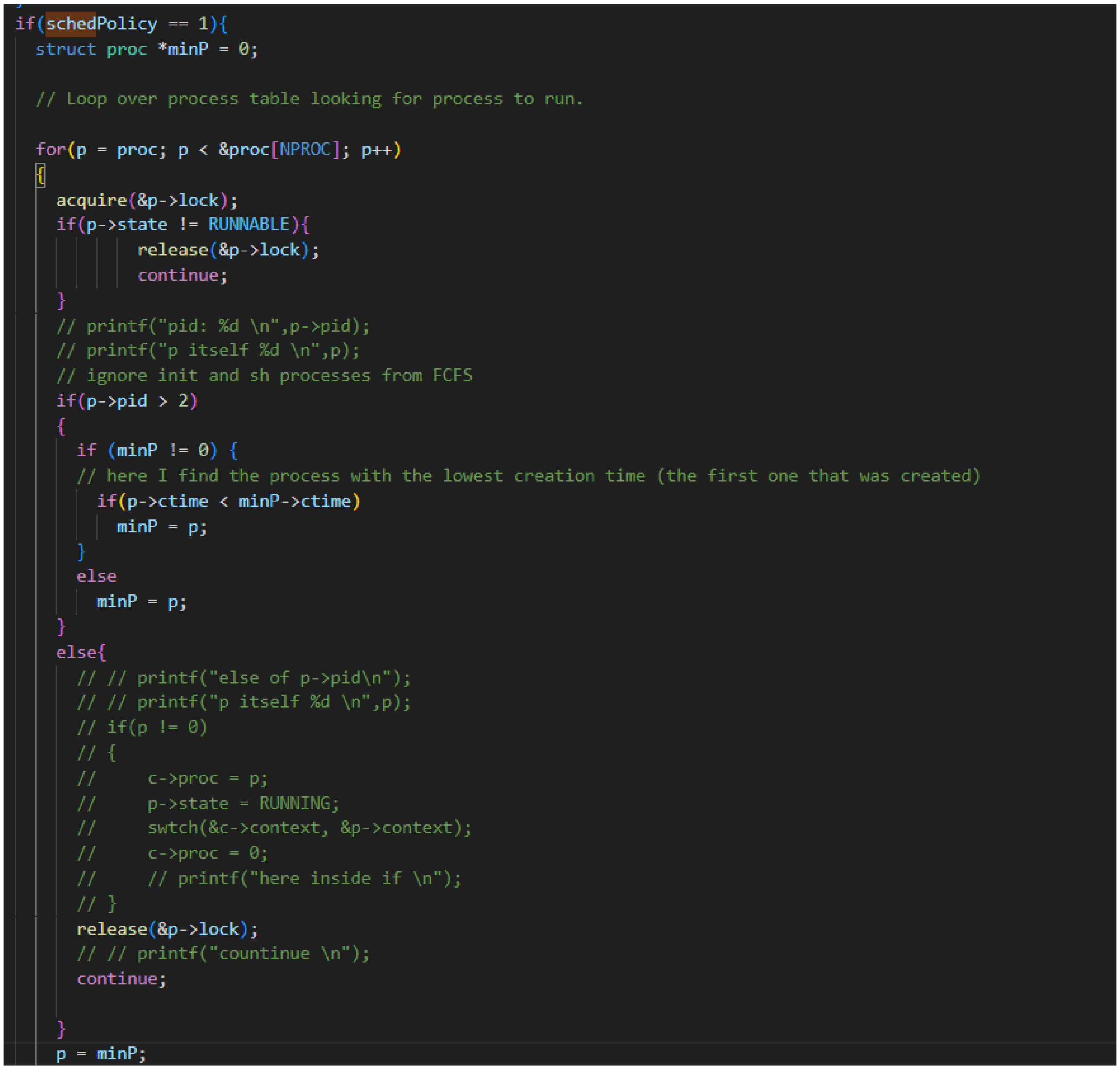
So in short, a process is running until a trap happens in kerneltrap function (./kerne/trap.c) then this function checks if the trap is a timer interrupt and then it calls yield function. This function change process state from RUNNING to RUNNABLE and calls sched function and it saves and restore of its kernel thread and intena then it switches context from current process to CPU's context to call the scheduler by using swtch function then scheduler iterates over all processes until it finds new process with RUNNABLE state and this loop happens forever.

FCFS Scheduler: To implement the FCFS (First-Come-First-Serve) scheduler, we introduced modifications to the existing scheduling policy. We added a for loop that iterates over all processes and selects the one with the smallest creation time (ctime). We store this process in a variable called minp and execute it. However, since the documentation advises against using the FCFS policy for the first two processes, we ignore them initially and execute them at the end.

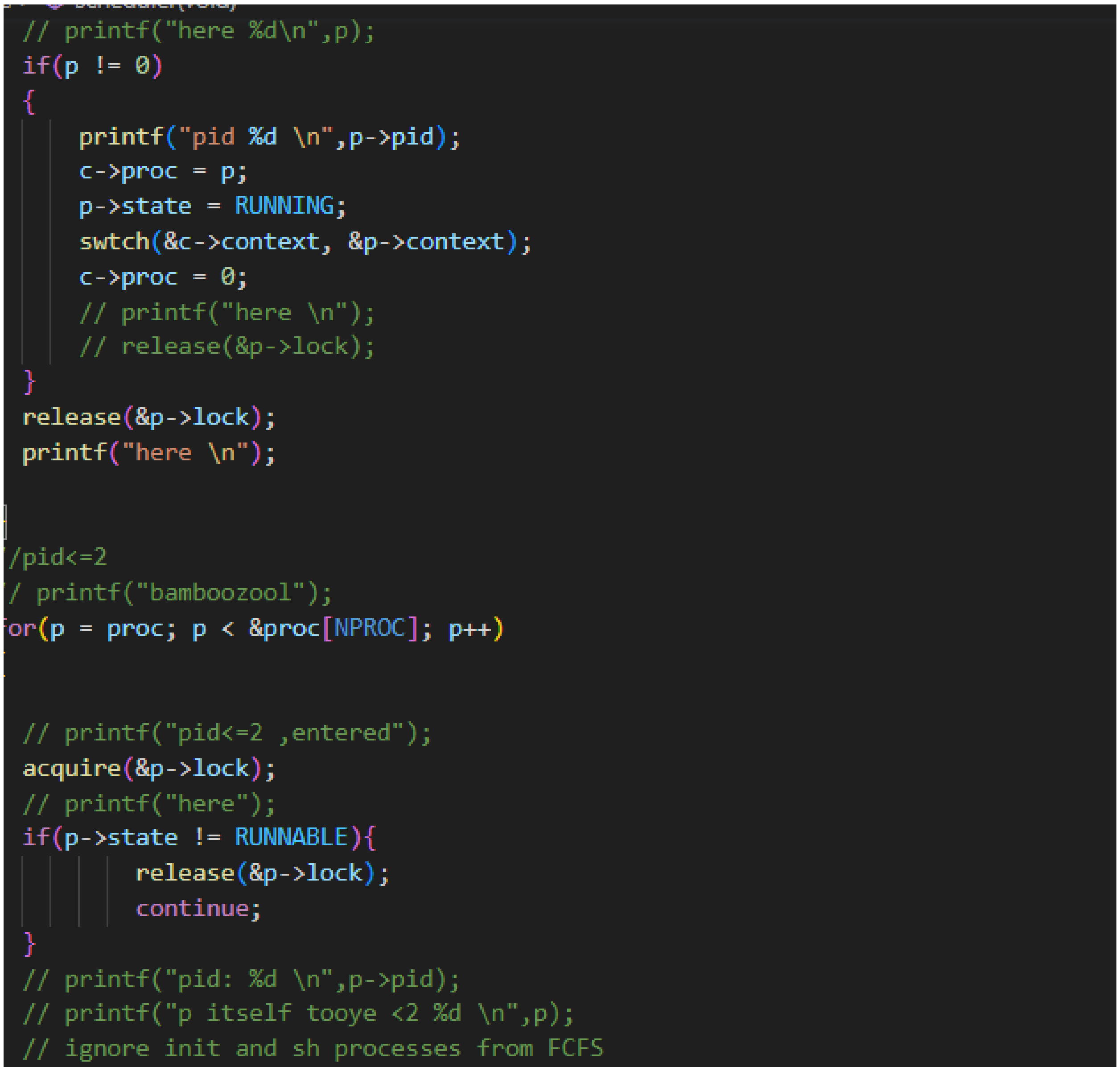
To enable users to switch between scheduling policies, we implemented a system call named "changePolicy." When the argument is set to 0, it functions as a Round Robin scheduler, while setting it to 1 invokes the FCFS scheduler described above.



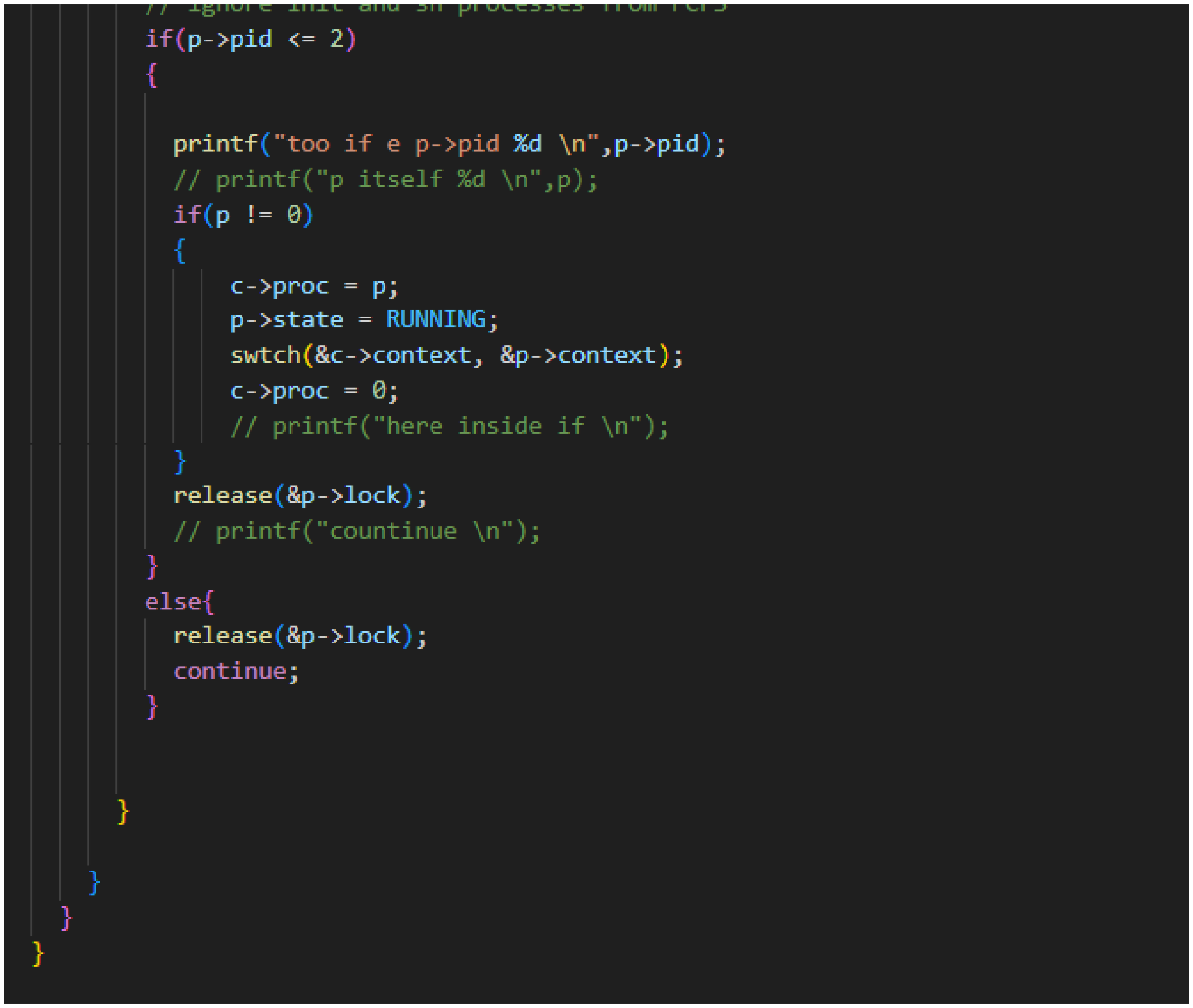
```
// Per-CPU process scheduler.
// Each CPU calls scheduler() after setting itself up.
// Scheduler never returns.  It loops, doing:
//  - choose a process to run.
//  - swtch to start running that process.
//  - eventually that process transfers control
//    via swtch back to the scheduler.
void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

  c->proc = 0;
  for(;;){
    // Avoid deadlock by ensuring that devices can interrupt.
    intr_on();
   if(schedPolicy == 0){
     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);
     }
   }
   if(schedPolicy == 1){
```



```
if(schedPolicy == 1){
  struct proc *minP = 0;

  // Loop over process table looking for process to run.

  for(p = proc; p < &proc[NPROC]; p++)
  {
    acquire(&p->lock);
    if(p->state != RUNNABLE){
            release(&p->lock);
            continue;
    }
    // printf("pid: %d \n",p->pid);
    // printf("p itself %d \n",p);
    // ignore init and sh processes from FCFS
    if(p->pid > 2)
    {
      if (minP != 0) {
      // here I find the process with the lowest creation time (the first one that was created)
        if(p->ctime < minP->ctime)
          minP = p;
      }
      else
        minP = p;
    }
    else{
      // // printf("else of p->pid\n");
      // // printf("p itself %d \n",p);
      // if(p != 0)
      // {
      //     c->proc = p;
      //     p->state = RUNNING;
      //     swtch(&c->context, &p->context);
      //     c->proc = 0;
      //     // printf("here inside if \n");
      // }
      release(&p->lock);
      // // printf("countinue \n");
      continue;

    }
    p = minP;
```

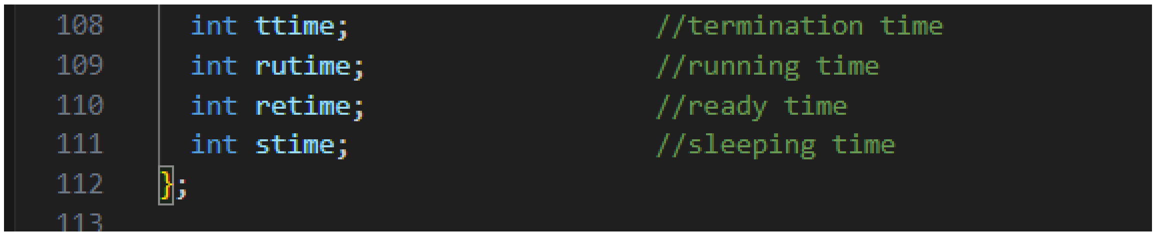


```
 // printf("here %d\n",p);
 if(p != 0)
 {
     printf("pid %d \n",p->pid);
     c->proc = p;
     p->state = RUNNING;
     swtch(&c->context, &p->context);
     c->proc = 0;
     // printf("here \n");
     // release(&p->lock);
 }
 release(&p->lock);
 printf("here \n");


/pid<=2
/ printf("bamboozool");
or(p = proc; p < &proc[NPROC]; p++)


 // printf("pid<=2 ,entered");
 acquire(&p->lock);
 // printf("here");
 if(p->state != RUNNABLE){
         release(&p->lock);
         continue;
 }
 // printf("pid: %d \n",p->pid);
 // printf("p itself tooye <2 %d \n",p);
 // ignore init and sh processes from FCFS
```



```
        // ignore init and sh processes from fcfs
        if(p->pid <= 2)
        {

          printf("too if e p->pid %d \n",p->pid);
          // printf("p itself %d \n",p);
          if(p != 0)
          {
              c->proc = p;
              p->state = RUNNING;
              swtch(&c->context, &p->context);
              c->proc = 0;
              // printf("here inside if \n");
          }
          release(&p->lock);
          // printf("countinue \n");
        }
        else{
          release(&p->lock);
          continue;
        }


      }

    }
  }
}
```

Additionally, we needed to declare variables in proc.h to track various process timing information, including ttime (termination time), rutime (running time), retime (ready time), and stime (sleeping time). These variables are initialized at allocproc and wait. To keep these values up to date, we introduced an "updateproc" function that is called for every tick increment.



```
  int ttime;                        //termination time
  int rutime;                       //running time
  int retime;                       //ready time
  int stime;                        //sleeping time
};
```

after that we had to implement a series of system calls to return these variables, like getttime and getctime which we will implement like the previous system calls we coded.

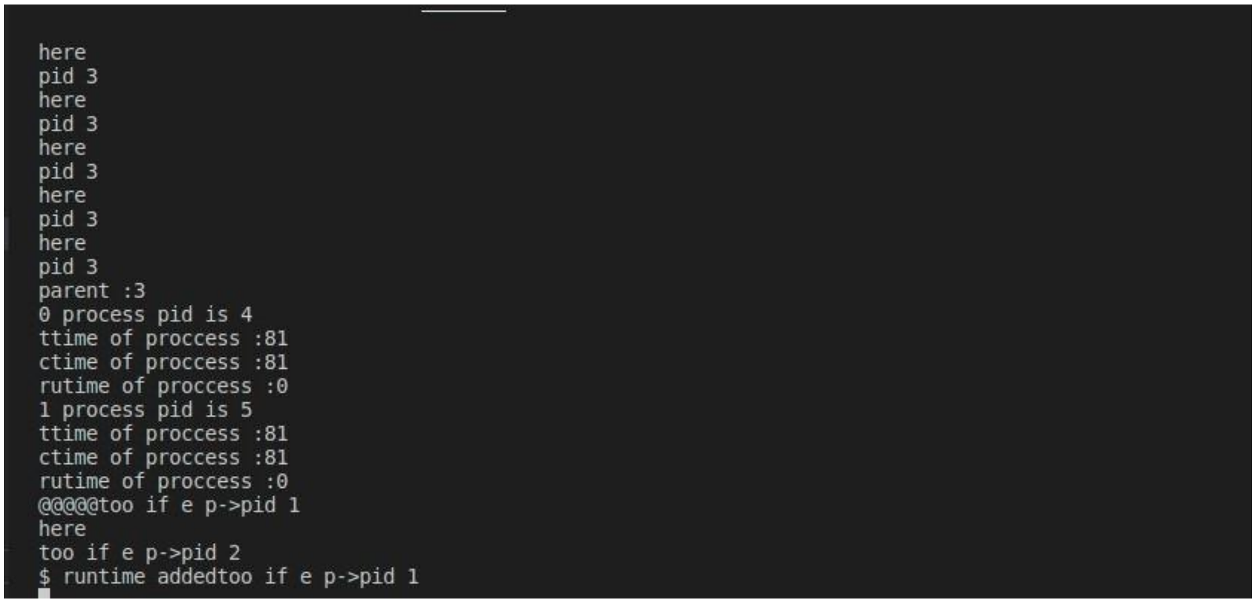
Then we have to write the test section:

to do so first we make a file in the user folder and we also added a new entry to the makefile :

$U/_sched_test\

then inside this sched_test file we fork a couple of process where they add 2 arrays with the size of 10000.

while they do that the parent waits on them and then uses the system calls that were previously mentioned to gather the information that is needed to compare the two policies.



```
here
pid 3
here
pid 3
here
pid 3
here
pid 3
here
pid 3
parent :3
0 process pid is 4
ttime of proccess :81
ctime of proccess :81
rutime of proccess :0
1 process pid is 5
ttime of proccess :81
ctime of proccess :81
rutime of proccess :0
@@@@@too if e p->pid 1
here
too if e p->pid 2
$ runtime addedtoo if e p->pid 1
```

Result of sched_test.c

We also use the changePolicy() system call to switch between these policies.