CS 344 OPERATING SYSTEMS LABORATORY - 02

GROUP 19

Drishti Chouhan (180101021)
Kousik Rajesh (180101094)
Mridul Garg (180123028)
Eklavya Jain (180123065)

PART A:

printNumProc: A user file to call the system call that gets the total number of processes.

printMaxPid: A user file to call the system call that gets the maximum PID among all the processes.

printProcInfo: A user file to call the system call that gets process information, ie., the parent PID, the number of times the process was context switched in by the scheduler, and the process size in bytes.

testBurstTime: A user file to call the system call that sets and gets the burst time of the currently scheduled process.

```
$ printNumProc
No. of Process: 3
$ printMaxPid
Maximum PID is: 5
$ printProcInfo 2
Process Info
  Parent PID is: 1
 Process Size 16384
Number of context switches 22
$ printProcInfo 100
No process found, return value = -1
$ testBurstTime
For PID = 8
Burst Time Before: 0
Burst Time After: 10
```

PART B:

Shortest Job First Scheduling Algorithm :

This scheduling algorithm chooses the job with the shortest burst time and schedules it for execution after an already scheduled job has completed.

```
void scheduler(void)
{
 struct proc *p,*shortest,*p1;
 struct cpu *c = mycpu();
 c->proc = 0;

 for (;;)
 {
   // Enable interrupts on this processor.
   sti();
   // Loop over process table looking for process to run.
   acquire(&ptable.lock);
   shortest= 0;
   int currentMin = __INT_MAX__; // CPU burst times are between 1 and 20.
   for (p1 = ptable.proc; p1 < &ptable.proc[NPROC]; p1++)
   {
     if (p1->state != RUNNABLE)  continue;

     if(p1->burst < currentMin)
```

```
      {
        shortest = p1;
        currentMin = p1->burst;
      }
      else if(p1->burst == currentMin && p1->pid<shortest->pid){
        shortest=p1;
      }
    }
    p=shortest;
    if (p != 0)
    {
      (p->count)++;
      // 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;
      p->tickcounter=0;
      // if(p->burst >=0) // The burst time we set is not equal to the execution time hence due to
mismatch the process keeps running /prematurely terminates
      // cprintf("Context switching to PID %d with remaining burst %d \n",p->pid, p->burst);
//Uncomment this line to see context switches after each quanta
      swtch(&(c->scheduler), p->context);
      switchkvm();
      c->proc = 0;
    }
    release(&ptable.lock);
  }
}
```

Design and Implementation

The scheduler loops over the process table, and schedules the shortest job for execution. If the process is not runnable, it skips that entry. In any other case, it finds and schedules the shortest job for execution.

To the proc struct defined in proc.h add two variables ***burst*** (Denoting the burst time) and ***count*** (denoting the number of context switches).

Kernel data structures: The scheduler runs over the **p-table** in order to loop through the burst times and searches for the shortest job to be scheduled.

Runtime complexity: The run time complexity of the scheduler algorithm is O(n), as a single *for loop* iterates over each process in the ptable to find the shortest job amongst the ones in the ready queue.

Corner case: When two processes have equal burst times, FCFS scheduling is implemented(the one with the lesser PID) and the process that entered the ready queue first, is scheduled.

Test cases :

1. Random burst times with CPU bound processes:

SJF Scheduler:
In the file test_scheduler.c, the function **random_computation** executes a numerical calculation and therefore is a CPU-bound process. The SJF scheduler is not affected even when the processes block and the processes are executed in the order of their execution times

2. Random burst times with I/O bound processes:
In the file test_scheduler.c, the function **random_io** executes two I/O bound processes, **printf,** and **sleep.**

```
Test 1: Pseudo-Random burst times with CPU bound processes

Children :
Child#          PID             Burst Time
0               4               9
1               5               1
2               6               4
3               7               15
4               8               20

Order of completion :
PID             Burst Time
5               1
6               4
4               9
7               15
8               20

Test 2: Pseudo-Random burst times with I/O bound processes
.....................................................
Children :
Child#          PID             Burst Time
0               9               9
1               10              1
2               11              4
3               12              15
4               13              20

Order of completion :
PID             Burst Time
10              1
11              4
9               9
12              15
13              20
```

SJF SCHEDULER

DEFAULT ROUND-ROBIN SCHEDULER

```
Test 1: Pseudo-Random burst times with CPU bound processes
Using Round Robin scheduling

Children :
Child#          PID             Burst Time
0               4               9
1               5               1
2               6               4
3               7               15
4               8               20

Order of completion :
PID             Burst Time
7               15
8               20
4               9
5               1
6               4

Test 2: Pseudo-Random burst times with I/O bound processes
Using Round Robin scheduling
.....................................................
Children :
Child#          PID             Burst Time
0               9               9
1               10              1
2               11              4
3               12              15
4               13              20

Order of completion :
PID             Burst Time
11              4
10              1
9               9
13              20
12              15
```

Default Scheduler(Round Robin):
The default round robin scheduler doesn't take burst times into account when scheduling the processes. Therefore, order of completion is random and not dependent on the burst times.

3. Increasing Burst times

Since the test case contains processes with increasing burst times, they will be scheduled in ascending order of their burst times.

4. Decreasing burst times

The test contains processes with decreasing burst times, but they will be scheduled in the shortest job first order.

```
Test 3: Increasing burst times with both CPU and I/O bound processes
.....................................................................
Children :
Child#          PID             Burst Time
0                14             1
1                15             5
2                16             8
3                17             10
4                18             13

Order of completion :
PID             Burst Time
14              1
15              5
16              8
17              10
18              13

Test 4: Decreasing burst times with both CPU and I/O bound processes
.....................................................................
Children :
Child#          PID             Burst Time
0                19             17
1                20             14
2                21             12
3                22             11
4                23             9

Order of completion :
PID             Burst Time
23              9
22              11
21              12
20              14
19              17
```

SJF SCHEDULER

DEFAULT ROUND ROBIN SCHEDULER

```
Test 3: Increasing burst times with both CPU and I/O bound processes
Using Round Robin scheduling
.....................................................................
Children :
Child#          PID             Burst Time
0                14             1
1                15             5
2                16             8
3                17             10
4                18             13

Order of completion :
PID             Burst Time
18              13
17              10
15              5
14              1
16              8

Test 4: Decreasing burst times with both CPU and I/O bound processes
Using Round Robin scheduling
.....................................................................
Children :
Child#          PID             Burst Time
0                19             17
1                20             14
2                21             12
3                22             11
4                23             9

Order of completion :
PID             Burst Time
22              11
23              9
19              17
21              12
20              14
```

Default Scheduler(Round Robin):
The default round robin scheduler doesn't take burst times into account when scheduling the processes. Therefore, order of completion is random and not dependent on the burst times.

BONUS

To the proc struct defined in proc.h, add another parameter **tickcounter** to the struct proc This denotes the number of CPU cycles the process has been in the running state since it was initialized.

// trap.c

In trap.c we change the conditional to include a check for when **tickcounter** becomes equal to **QUANTA** in which case we decrement the burst, reset tickcounter and call yield() to call scheduler again

```
// For hybrid scheduling
 if (myproc() && myproc()->state == RUNNING && tf->trapno == T_IRQ0 + IRQ_TIMER && inctickcounter()
== QUANTA){

   decBurst();
   cprintf("Interrupt %d with burst %d \n", myproc()->pid, myproc()->burst);
   myproc()->tickcounter=0;
   yield();
 }
 // For SJF scheduling
 // if (myproc() && myproc()->state == RUNNING && tf->trapno == T_IRQ0 + IRQ_TIMER)
 // {
 //   yield();
 // }
```

// proc.c

In proc.c, we define the scheduler function, which is a hybrid of SJF and default round-robin. The algorithm chooses the process with the least burst time and executes the process for a fixed time quanta till there is an interrupt called by trap.c when the time quanta expires.

We also define two functions in proc.c, **decBurst,** and **inctickcounter:**
decBurst: Decreases the burst time of the running process after every time-slice.
inctickcounter: For every clock cycle that the process is in the running state, the function increments the tickcounter parameter by 1 clock cycle.

```
void decBurst(void)
{
 acquire(&ptable.lock);
    mycpu()->proc->burst = mycpu()->proc->burst < 0 ? -1 : mycpu()->proc->burst - QUANTA;
 release(&ptable.lock);
}

int inctickcounter()
{
 int res;
 acquire(&ptable.lock);
 res = ++mycpu()->proc->tickcounter;
 release(&ptable.lock);
 return res;
}
```

// param.h

Define **QUANTA,** the time-quantum for the round-robin preemptive scheduling.

```
#define  QUANTA       4      // Quanta for hybrid scheduling
```

// output

HYBRID SCHEDULING

```
Test 1: Pseudo-Random burst times with CPU bound processes
Context switching to PID 4 with remaining burst 0
Context switching to PID 5 with remaining burst 0
Context switching to PID 6 with remaining burst 0
Context switching to PID 7 with remaining burst 0
Context switching to PID 8 with remaining burst 0
Context switching to PID 5 with remaining burst 1
Context switching to PID 3 with remaining burst 0
Context switching to PID 6 with remaining burst 4
Context switching to PID 6 with remaining burst 0
Context switching to PID 3 with remaining burst 0
Context switching to PID 4 with remaining burst 9
Context switching to PID 4 with remaining burst 5
Context switching to PID 4 with remaining burst 1
Context switching to PID 3 with remaining burst 0
Context switching to PID 7 with remaining burst 15
Context switching to PID 7 with remaining burst 11
Context switching to PID 7 with remaining burst 7
Context switching to PID 7 with remaining burst 3
Context switching to PID 3 with remaining burst 0
Context switching to PID 8 with remaining burst 17
Context switching to PID 8 with remaining burst 13
Context switching to PID 8 with remaining burst 9
Context switching to PID 8 with remaining burst 5
Context switching to PID 8 with remaining burst 1
Context switching to PID 3 with remaining burst 0

Children :
Child#          PID             Burst Time
0               4               9
1               5               1
2               6               4
3               7               15
4               8               17

Order of completion :
PID             Burst Time
5               1
6               4
4               9
7               15
8               17
Context switching to PID 2 with remaining burst 0
```

1. The first 5 lines denote the creation of 5 child processes with PID 4,5,6,7 and 8 with **burst initialized by 0.** Then the process with the minimum burst time, i.e., PID 5 is scheduled.
2. It **runs for one time-slice** and then **preempted**. It's burst time becomes less than equal to zero and **does not need to be scheduled again.** Now the control goes to the parent process (PID 3).
3. Then the context is switched to PID 6 with burst time 4. **After one quantum has completed, PID 6 is preempted. But now again it has the least burst time and hence again context is switched.** Now, the context switches to the parent (PID 3).
4. Now the scheduler chooses the process with PID 4, i.e., the next shortest process, **which runs for one time quantum. This reduces its burst time to 5, which again is the shortest process among all the processes and is thereby scheduled again. The same thing happens one more time and this process is finally completed after three quanta.** Further, control is passed to the parent (PID 3).
5. The next shortest process is PID 7, **which runs in a similar manner as the process with PID 4. It takes 4 quanta to complete its execution.**
6. Similarly, **the process with PID 8 takes 5 quanta to exhaust its burst time.**

Assumptions / Implementation details:

1. All processes from PID 4-8 are introduced simultaneously at t=0.
2. Following a hybrid scheduling algorithm. The smallest process is picked and executed for 1 quantum and its burst is decremented by the number of cycles in 1 quantum after which it is preempted and a new process with the smallest burst is scheduled by the scheduler. This is done repeatedly till all the processes have terminated (exhausted their burst time, or spent burst time amount of time in the scheduler).

NOTES:

- Every time ptable is accessed acquire and release have been used to ensure good locking discipline
- The system call **set_burst_time** has a call to yield() so that the scheduler is called when burst time is set
- The burst-time we set is not equal to the actual execution time

---