CS 344 : OPERATING SYSTEMS LAB
LAB 2

GROUP NUMBER : 12

GROUP MEMBERS:

MOHAMMAD HUMAM KHAN (180123057)

KARTIKEYA SINGH (180123021)

SIDDHARTHA JAIN (180101078)

AB SATYAPRAKASH (180123062)

** xv6 runs on multiprocessors, and allows multiple CPUs to execute concurrently inside the kernel. These multiple CPUs operate on a single address space and share data structures among them. So, locking is used for process synchronization. The **Process table** is a shared structure among all system calls and CPUs, so it is in the critical section of every process accessing it. A process which wants to access a shared structure (Process table) has to first acquire a lock through **acquire()** function in the entry section of the program code, to make sure only one CPU holds the lock for accessing shared structure (process table) in the critical section of the code. At last after completing operation in the critical section, the process has to release the lock through **release()** function in the exit section of the program code, thus giving other CPUs a fair chance to access the critical section. So, in one line, locks provide process synchronization for better functioning of the Operating System.*

PART-A

1. getNumProc() and getMaxPid()

Code:

```
int
sys_getNumProc(void){
    struct proc* p;
    acquire(&ptable.lock);
    int numProc = 0;
    for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
      if(p->state != UNUSED)
        numProc++;
    }
    release(&ptable.lock);
    return numProc;
}

int
sys_getMaxPid(void){
  struct proc* p;
  acquire(&ptable.lock);
  int maxPid = -1;
  for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
    if((p->pid) > maxPid)
      maxPid = (p->pid);
  }
```

```
  release(&ptable.lock);
  return maxPid;
}
```

Here we implemented the above system calls and ran the test case to check the correctness. The total number of active processes are 3 viz. ***init, sh, partA_test1*** at the time of execution of test case program ***partA_test1.c***. The maximum PID is 3 initially, but as we run the user test program for multiple times we can see that MaxPID increases because after termination of previously run test programs, the entry remains in the process table and is not removed. So when we loop over the process table for getting MaxPID, these processes are also present and hence MaxPID increases every time we run the user test program.

```
init: starting sh
$ partA_test1
Total Number of Active Processes: 3
Maximum PID: 3
$ partA_test1
Total Number of Active Processes: 3
Maximum PID: 4
$ partA_test1
Total Number of Active Processes: 3
Maximum PID: 5
```

2. getProcInfo(pid, &processInfo)

<u>Code:</u>

```
int
sys_getProcInfo(int pid,struct processInfo* procInfo){
  argint(0,&pid);
  argptr(1,(char**)&procInfo,sizeof(procInfo));
  struct proc* p;
  acquire(&ptable.lock);
  int success = -1;
  for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
    if((p->pid) == pid){
      success = 0;
      procInfo->ppid = p->parent->pid;
```

```
      procInfo->psize = p->sz;
      procInfo->numberContextSwitches = p->numContextSwitch;
    }
  }
  release(&ptable.lock);
  return success;
}
```

```
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;     // switch() 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 numContextSwitch;        // Number of context switches
  int burstTime;               // CPU burst time of the process
};
```

In this question we implemented the system call **getProcInfo(pid, &processInfo).** As can be seen the system call takes 2 arguments, **pid** an integer, which is process ID and **processInfo** a structure, containing information related to process.To use the functionality of processInfo, we include the corresponding structure in **user.h** and header files in **proc.c**.

To calculate the number of context switches of the process, we add a new field in the **struct proc** (which contains all the information about the process) called

numContextSwitch. This is a counter that is initialized to 0 when the process is brought to the **EMBRYO state** for the first time by **allocproc()** and increases every time the process makes a context switch (when a process moves from a RUNNABLE to RUNNING state).

```
init: starting sh
$ partA_test2
PID     PPID     SIZE     Number of Context Switches
1       443       12288    27
2       1        16384    16
3       2        12288    10
```

3. set_burst_time(n) & get_burst_time()

__**Code:**__

```
int
sys_set_burst_time(int n){
  argint(0,&n);
  struct proc* p;
  acquire(&ptable.lock);
  int success = -1;
  for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
    if((p->state == RUNNING)){
      p->burstTime = n;
      success = 0;
      break;
    }
  }
  release(&ptable.lock);
  yield();
  return success;
}

int
sys_get_burst_time(){
  struct proc* p;
  int burstTime = -1;
  acquire(&ptable.lock);
  for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
```

```
    if((p->state) == RUNNING){
      burstTime = p->burstTime;
      break;
    }
  }
  release(&ptable.lock);
  return burstTime;
}
```

set_burst_time(n) is used to set the burst time of the process to a user input value.To implement this, we include a new field in the **proc (struct)** (containing all the information about the process), called **burstTime (int)**, which is used to store the burst time. **yield()** is called at the end of **set_burst_time(n)** so that after running this system call the schedule yield() is called at the end of set_burst_time(n) so that after running this system call the scheduler is invoked again and hence scheduling happens again according to new burst times.r is invoked again and hence scheduling happens again according to new burst times.

```
init: starting sh
$ partA_test3
Burst Time: 4
Burst Time: 5
Burst Time: 6
Burst Time: 7
Burst Time: 8
Burst Time: 9
Burst Time: 10
Burst Time: 11
Burst Time: 12
```

get_burst_time() is added to get the burst time of the process at any instant. It is just a helper function, to check whether the burst time of the process is correctly set or not. In the later part of the assignment, it is used to check the correctness of the shortest job first (SJF) scheduling algorithm.

PART-B : SJF SCHEDULING ALGORITHM

Initially in xv6, Round-Robin Scheduler is implemented which preempts the process after 1 clock cycle i.e. the value of time quantum is equal to 1 clock cycle. This part is implemented in trap.c file in line no. 103-107 where we call yield() function which forces the process to give up CPU. At the end of each interrupt, trap calls yield. Yield in turn calls sched, which calls **swtch()** to save the current context in proc->context and switch to the scheduler context previously saved in cpu->scheduler.

```
// Force process to give up CPU on clock tick.
  // If interrupts were on while locks held, would need to check
nlock.
  // if(myproc() && myproc()->state == RUNNING &&
  //    tf->trapno == T_IRQ0+IRQ_TIMER)
  //   yield();
```

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

  for(;;){
    // Enable interrupts on this processor.
    sti();
    // Loop over the process table looking for the process to run.
    acquire(&ptable.lock);
    p = 0;
    int minTime = 5000; // CPU burst times are between 1 and 20.

    for(temp = ptable.proc; temp < &ptable.proc[NPROC]; temp++){
      if((temp->state == RUNNABLE) && (temp->burstTime) < minTime){
        minTime = temp->burstTime;
        p = temp;
      }
    }
    if(p!=0){
      (p->numContextSwitch)++;
```

```
      // Switch to the 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();
      // Process is done running for now.
      // It should have changed its p->state before coming back.
      c->proc = 0;
    }
    release(&ptable.lock)
  }
}
```

EXPLANATION OF SCHEDULING POLICY

The Scheduling policy implemented is Shortest Job First (SJF) where the scheduler chooses the process with minimum burst available in READY QUEUE (state = RUNNABLE) and schedules it for execution. In xv6, there is no concept of burst time as such, so we have to add an extra field in struct proc to keep account of burst time of each process set by the user. Now our scheduling algorithm will use these burst times set by the user to schedule processes for execution.

IMPLEMENTATION DETAILS OF THE SCHEDULER

1. The scheduler first calls **sti() to enable interrupts**. Interrupts are enabled in every iteration of the loop on an idling CPU as there might be no **RUNNABLE** process because the processes (ex: the shell) are waiting for I/O. If the scheduler left interrupts disabled all the time, the I/O would never arrive.
2. Now the scheduler **acquires locks** for reading the process table and finding a process to run.

3. We initialize a variable **p = 0** i.e. a process structure initialized with NULL. This variable stores the process with minimum burst time found while iterating over the process table.
4. Now we initialize a variable **minTime = 5000** because we have to find the process with minimum burst time, so we initialize it with a value greater than all possible burst time of processes.
5. Then we iterate over the process table to check for two conditions:
    - ❖ Whether the process is **RUNNABLE** i.e. in the Ready Queue or not. Only processes that are in the ready queue are scheduled.
    - ❖ Whether the burst time of this process is less than the minimum burst time of a process found till now or not. If the burst time of this process is less than minTime, then the value of minTime and p are updated.
6. After exiting the loop, if the value of p is not zero i.e. there is some process that needs CPU for execution. Then the number of context switches for the process is incremented.
7. Now the value of per-CPU variable proc is set back to p i.e. **c->proc = p** is executed and the variable indicating the current executing process on cpu is set to current scheduled process p.
8. Then **switchuvm(p)** is called which sets up the process's kernel stack and switches to its page table.
9. Then the process state is marked as **RUNNING.**
10. Then **swtch()** function is called which performs the job of context-switching i.e. save current registers (including where to continue on scheduler) and load process's registers, handing the cpu over to the process.
11. Now control is transferred back to the scheduler and hence it switches back to kernel stacks and page table.
12. After this the per-CPU variable proc is set back to 0.
13. After this the lock is released so that other CPUs can also access the process table.

TIME COMPLEXITY OF SCHEDULING ALGORITHM

SJF Scheduling Algorithm works in O(N) time where N is the number of processes in the process table. In xv6 the maximum size of the process table is 64, So the algorithm essentially works in constant time. Each time the scheduler runs to find the process with minimum burst time, it loops over all the processes present in the process table. So the time complexity of the algorithm becomes O(N).

HANDLING CORNER CASES

- **CASE-I : DEFAULT BURST TIME**

  The default burst time of each process is set to 0 when the process is brought to EMBRYO state in allocproc(). Now for all user processes the range of values in which the user sets the burst time is greater than or equal to 1. This makes sure that system processes like **init** and **sh** are scheduled before any user processes for execution. This ensures smooth working of the system.

- **CASE-II : EQUAL BURST TIMES**

  In case when burst times of two processes become equal, FCFS Scheduling policy is followed i.e. the process that arrives first is scheduled before the process that arrives later.

EXPLANATION OF OUTPUT OF TEST CASES

Note: ***Please run all test cases in a separate qemu terminal, else due to constraints of qemu, when the number of processes increases it may lead to erroneous results.***

- **CASE-I : BURST TIMES IN RANDOM ORDER**

  In this test case we allocated random burst times to child processes created using fork(). Then we introduced a dummy delay so that all child processes could be created and scheduled at the same time. On running our scheduler we get the following results. It can be concluded that SJF schedules processes according to their burst times while Round-Robin scheduler gives chance to every process for a particular time quantum.

```
init: starting sh
$ test2 10
Increasing burst times
Burst times of parent process = 2

All children completed
Child 0.    pid 4    burst time = 3
Child 1.    pid 5    burst time = 4
Child 2.    pid 6    burst time = 5
Child 3.    pid 7    burst time = 6
Child 4.    pid 8    burst time = 7
Child 5.    pid 9    burst time = 8
Child 6.    pid 10    burst time = 9
Child 7.    pid 11    burst time = 10
Child 8.    pid 12    burst time = 11
Child 9.    pid 13    burst time = 12

Exit order
pid 6   burst time = 5
pid 8   burst time = 7
pid 4   burst time = 3
pid 5   burst time = 4
pid 10   burst time = 9
pid 12   burst time = 11
pid 9   burst time = 8
pid 11   burst time = 10
pid 7   burst time = 6
pid 13   burst time = 12
```

```
init: starting sh
$ test2 10
Increasing burst times
Burst times of parent process = 2

All children completed
Child 0.    pid 4    burst time = 3
Child 1.    pid 5    burst time = 4
Child 2.    pid 6    burst time = 5
Child 3.    pid 7    burst time = 6
Child 4.    pid 8    burst time = 7
Child 5.    pid 9    burst time = 8
Child 6.    pid 10    burst time = 9
Child 7.    pid 11    burst time = 10
Child 8.    pid 12    burst time = 11
Child 9.    pid 13    burst time = 12

Exit order
pid 4   burst time = 3
pid 5   burst time = 4
pid 6   burst time = 5
pid 7   burst time = 6
pid 8   burst time = 7
pid 9   burst time = 8
pid 10   burst time = 9
pid 11   burst time = 10
pid 12   burst time = 11
pid 13   burst time = 12
```

➢ **<u>CASE-II</u> : <u>BURST TIMES IN INCREASING ORDER</u>**

```
init: starting sh
$ test1 10
Random burst times
Burst times of parent process = 2

All children completed
Child 0.    pid 4    burst time = 5
Child 1.    pid 5    burst time = 17
Child 2.    pid 6    burst time = 9
Child 3.    pid 7    burst time = 3
Child 4.    pid 8    burst time = 7
Child 5.    pid 9    burst time = 10
Child 6.    pid 10    burst time = 8
Child 7.    pid 11    burst time = 15
Child 8.    pid 12    burst time = 16
Child 9.    pid 13    burst time = 4

Exit order
pid 8   burst time = 7
pid 4   burst time = 5
pid 5   burst time = 17
pid 6   burst time = 9
pid 7   burst time = 3
pid 9   burst time = 10
pid 10   burst time = 8
pid 11   burst time = 15
pid 12   burst time = 16
pid 13   burst time = 4
```

```
init: starting sh
$ test1 10
Random burst times
Burst times of parent process = 2

All children completed
Child 0.    pid 4    burst time = 5
Child 1.    pid 5    burst time = 17
Child 2.    pid 6    burst time = 9
Child 3.    pid 7    burst time = 3
Child 4.    pid 8    burst time = 7
Child 5.    pid 9    burst time = 10
Child 6.    pid 10    burst time = 8
Child 7.    pid 11    burst time = 15
Child 8.    pid 12    burst time = 16
Child 9.    pid 13    burst time = 4

Exit order
pid 7   burst time = 3
pid 13   burst time = 4
pid 4   burst time = 5
pid 8   burst time = 7
pid 10   burst time = 8
pid 6   burst time = 9
pid 9   burst time = 10
pid 11   burst time = 15
pid 12   burst time = 16
pid 5   burst time = 17
```

In this test case we allocated burst times to child processes in increasing order created using fork(). Then we introduced a dummy delay so that all child processes could be created and scheduled at the same time. On running our scheduler we get the following results. It can be concluded that SJF schedules processes according to their burst times while Round-Robin scheduler gives chance to every process for a particular time quantum.

- **CASE-III : BURST TIMES IN DECREASING ORDER**

```
init: starting sh
$ test3 10
Decreasing burst times
Burst times of parent process = 2

All children completed
Child 0.    pid 4    burst time = 20
Child 1.    pid 5    burst time = 19
Child 2.    pid 6    burst time = 18
Child 3.    pid 7    burst time = 17
Child 4.    pid 8    burst time = 16
Child 5.    pid 9    burst time = 15
Child 6.    pid 10    burst time = 14
Child 7.    pid 11    burst time = 13
Child 8.    pid 12    burst time = 12
Child 9.    pid 13    burst time = 11

Exit order
pid 4   burst time = 20
pid 5   burst time = 19
pid 6   burst time = 18
pid 8   burst time = 16
pid 7   burst time = 17
pid 10   burst time = 14
pid 9   burst time = 15
pid 13   burst time = 11
pid 12   burst time = 12
pid 11   burst time = 13
```

```
init: starting sh
$ test3 10
Decreasing burst times
Burst times of parent process = 2

All children completed
Child 0.    pid 4    burst time = 20
Child 1.    pid 5    burst time = 19
Child 2.    pid 6    burst time = 18
Child 3.    pid 7    burst time = 17
Child 4.    pid 8    burst time = 16
Child 5.    pid 9    burst time = 15
Child 6.    pid 10    burst time = 14
Child 7.    pid 11    burst time = 13
Child 8.    pid 12    burst time = 12
Child 9.    pid 13    burst time = 11

Exit order
pid 13   burst time = 11
pid 12   burst time = 12
pid 11   burst time = 13
pid 10   burst time = 14
pid 9   burst time = 15
pid 8   burst time = 16
pid 7   burst time = 17
pid 6   burst time = 18
pid 5   burst time = 19
pid 4   burst time = 20
```

In this test case we allocated burst times to child processes in decreasing order created using fork(). Then we introduced a dummy delay so that all child processes could be created and scheduled at the same time. On running our scheduler we get the following results. It can be concluded that SJF schedules processes according to their burst times while Round-Robin scheduler gives chance to every process for a particular time quantum.

- **CASE-IV : I/O BOUND PROCESS**

In this test case we allocated burst times in increasing order to child processes created using fork(). Then we introduced a dummy delay which performs a long burst of I/O operations and then a short burst of CPU operations. This makes sure that all child processes could be created and scheduled at the same time. On running our scheduler we get the following results. It can be concluded that SJF schedules processes according to their burst times while Round-Robin scheduler gives chance to every process for a particular time quantum.

```
All children completed
Child 0.    pid 4    burst time = 3
Child 1.    pid 5    burst time = 4
Child 2.    pid 6    burst time = 5
Child 3.    pid 7    burst time = 6
Child 4.    pid 8    burst time = 7
Child 5.    pid 9    burst time = 8
Child 6.    pid 10    burst time = 9
Child 7.    pid 11    burst time = 10
Child 8.    pid 12    burst time = 11
Child 9.    pid 13    burst time = 12

Exit order
pid 5   burst time = 4
pid 4   burst time = 3
pid 7   burst time = 6
pid 6   burst time = 5
pid 12   burst time = 11
pid 9   burst time = 8
pid 10   burst time = 9
pid 13   burst time = 12
pid 8   burst time = 7
pid 11   burst time = 10
```

```
All children completed
Child 0.    pid 4    burst time = 3
Child 1.    pid 5    burst time = 4
Child 2.    pid 6    burst time = 5
Child 3.    pid 7    burst time = 6
Child 4.    pid 8    burst time = 7
Child 5.    pid 9    burst time = 8
Child 6.    pid 10    burst time = 9
Child 7.    pid 11    burst time = 10
Child 8.    pid 12    burst time = 11
Child 9.    pid 13    burst time = 12

Exit order
pid 4   burst time = 3
pid 5   burst time = 4
pid 6   burst time = 5
pid 7   burst time = 6
pid 8   burst time = 7
pid 9   burst time = 8
pid 10   burst time = 9
pid 11   burst time = 10
pid 12   burst time = 11
pid 13   burst time = 12
```