

## Operating Systems Lab CS344 Assignment 2

Group 6

Mahek Vora 200101062

Siddharth Bansal 200101093

Vani Krishna Barla 200101101

### PART A

5 new system calls namely **getNumProc**, **getMaxPID**, **getProcInfo**, **setBurstTime**, **getBurstTime** were added.

1.**getNumProc** returns the number of processes that are in the process table and not in the "UNUSED" state (either in embryo, running, runnable, sleeping, or zombie states).

2.**getMaxPID** returns the maximum value of pid from the active processes

3.**getProcInfo** returns the parent PID, the number of times the process was context switched in by the scheduler, and the process size in bytes for the process. The field contextswitches was added to proc structure and in the scheduler code, this field is incremented when a context switch occurs

```
// 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 contextCount;        // keeps tracks of the no.of context switches
    int burst_time;          // keeps track of burst time
    int run_time;           // keeps track of total time run by the process
};
```

In the scheduler code, context switches is incremented here

```
c->proc = p;
p->contextCount = p->contextCount + 1;
switchvm(p);
p->state = RUNNING;
swtch(&(c->scheduler), p->context);
switchkvm();
```

4.**setBurstTime** is used to set the burst time of the current process to the specified value, passed as a parameter, the field "burstTime" is added to the proc structure as required.

5.**getBurstTime**, returns the burst time of the process.

#### Outputs for part-A:

```
$ getNumProc
Number of active processes = 3
$ getMaxPid
MAX ID of the process running = 5
```

```
$ getProcInfo 2
The ID of the parent process is = 1
The size of the process in bytes is = 16384
The number of context switches for given process = 30
$ setBurstTime 3
Burst Time of Process is = 3
```

## PART B

### Scheduler:

The default scheduler follows round robin scheduling algorithm, the snippet below(common to all algorithms) is responsible to cause a context switch to process p. **switchvm** helps OS to load the process, the processor switches to execute it, when the process comes back to scheduler, kernel loads its memory using **switchkvm** are

```
c->proc = p;
    switchvm(p);
    p->state = RUNNING;
    p->contextCount++; //keeping count of context
    swtch(&(c->scheduler), p->context);
    switchkvm();
```

This process **p** is selected based on the scheduling algorithm chosen

## Round Robin:

The default algorithm is round robin scheduling, where we loop over the ptable once and run the runnable processes for a time-quantum amount of time. So the average overall time complexity is  $O(N)$  and thus the amortized time complexity per process in this case is  $O(1)$

## SJF: Shortest Job First

### Algorithm:

The process that has the shortest burst time from all the "RUNNABLE" processes is given the highest priority and so on.

### Code:

```
void scheduler(void)
{
    struct proc *p;
    struct cpu *c = mycpu();
    c->proc = 0;
    for(;;){
        // Enable interrupts on this processor.
        sti();
        acquire(&ptable.lock);
        int minBurst=1000;
        for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
            {
                if(p->state!=RUNNABLE)
                    continue;
                if(p->burst_time<minBurst)
                    minBurst=p->burst_time;
            }
        }
        // Loop over process table looking for process to run.

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

            // Switch to chosen process. It is the process's job
            // to release ptable.lock and then reacquire it
            // before jumping back to us.
            if(p->burst_time==minBurst)
            {
                // cprintf("process running is = %d\n",p->pid);
                c->proc = p;
            }
        }
    }
}
```

```

switchvm(p);
p->state = RUNNING;
p->contextCount++; //keeping count of context
switch(&(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);
}
}

```

We are performing a linear search to get the minimum burst time runnable process, so average time complexity per process is  $O(N)$  and overall time complexity for all processes is thus  $O(N^2)$ .

## HBSJF: Hybrid Scheduling

### Algorithm:

Given a time quantum, each process is allowed to run upto time quantum units per round, and then context switch occurs. The order in every round is determined like SJF scheduling. The detailed algorithm is as follows:

- Create a ready queue RQ where the processes get submitted.
- Set up the processes in RQ in the increasing order of burst -time of each process.
- Fix the time slice as execution time of the primary procedure lying in the Ready Queue RQ.
- Repeat the next 2 steps until the ready queue RQ gets vacant.
- Select the primary process in RQ and allocate CPU to it for unit time quantum.
- The process in ready queue RQ is to be removed if the running process' remaining burst time becomes zero, otherwise move the method which is executing to the termination of the Ready Queue RQ.

### Code:

#### Ready Queue Code:

```

struct RQ{
    struct spinlock lock;

```

```

    struct proc* proc[NPROC+1];
    int length;
} ;

struct RQ RQ1;
struct RQ RQ2;

int empty(struct RQ *rq ){
    acquire(&(rq->lock));
    if(rq->length==0){
        release(&(rq->lock));
        return 1;
    }
    else{
        release(&(rq->lock));
        return 0;
    }
}

int full(struct RQ *rq){
    acquire(&(rq->lock));
    if(rq->length==NPROC){
        release(&(rq->lock));
        return 1;
    }
    else{
        release(&(rq->lock));
        return 0;
    }
}

void insert(struct proc *p,struct RQ *rq){
    if(full(rq))
        return;
    acquire(&(rq->lock));
    rq->length+=1;
    rq->proc[rq->length] = p;
    int j;
    for( j = rq->length-1;
        (j>=1)&&((rq->proc[j]->burst_time)>(p->burst_time)); j--){
        rq->proc[j+1] = rq->proc[j];
    }
    rq->proc[j+1] = p;
    release(&(rq->lock));
}

struct proc *min_process(struct RQ *rq){

```

```

    if(empty(rq))
        return 0;
    acquire(&(rq->lock));
    struct proc* minimum_process = rq->proc[1];
    if(rq->length==1)
rq->length = 0;
    else{
for(int j=2;j<=rq->length;j++){
    rq->proc[j-1] = rq->proc[j];
}
rq->length-=1;
    }
    release(&(rq->lock));

    return minimum_process;
}

```

The above code has the implementation of a data structure, RQ(ReadyQueue) (the underlying queue is a circular queue implemented on an array). Empty and full functions check if the queue is empty/full respectively.

The queue uses a sorted-insert function so all the values in the queue are sorted in the order of burst times.

The minimum burst time process is thus always at the head of the queue, and deletes the front/1st process from the ready queue.

### Scheduler Code:

```

void scheduler(void)
{
    struct proc *p;
    struct cpu *c = mycpu();
    c->proc = 0;
    for(;;){
        // Enable interrupts on this processor.
        sti();
        acquire(&ptable.lock);

        if(empty(&RQ1)){
            while(!empty(&RQ2)){
                p = min_process(&RQ2);
                insert(p, &RQ1);
            }
        }
    }
}

```

```

if(empty(&RQ1)){
    release(&ptable.lock);
    continue;
}
p = min_process(&RQ1); // O(1)
if(p->state!=RUNNABLE){
    release(&ptable.lock);
    continue;
}
c->proc = p;
switchvm(p);
p->state = RUNNING;
(p->contextCount)++;
swtch(&(c->scheduler), p->context);
switchkvm();
c->proc = 0;
release(&ptable.lock);
}
}

```

Ready queue 1 is the default ready queue and ready queue 2 will store the processes that have completed their time quantum but have not finished execution (filled by `yield_helper()`, refer below). If ready queue 1 is empty, a round is completed and we bring all processes from ready queue 2 to ready queue 1. If both the queues are empty, all processes have completed execution.

Now we take the minimum burst time process(head of queue) in  $O(1)$  time as the queue is sorted and context switches to that process. In order to maintain the sorted queue, we are performing a linear search at every iteration (like insertion sort) so the time complexity is  $O(N^2)$

In `proc.c`:

```

void yield_helper(void) {
    acquire(&ptable.lock); //DOC: yieldlock
    myproc()->state = RUNNABLE;
    insert(myproc(), &RQ2);
    sched();
    release(&ptable.lock);
}

```

In `trap.c` :

```

if(myproc() && myproc()->state == RUNNING &&
    tf->trapno == T_IRQ0+IRQ_TIMER)
{
    myproc()->run_time += 1;
    if(myproc()->burst_time){

        if(myproc()->burst_time == myproc()->run_time)
            exit();
    }
    if((myproc()->run_time)%time_quantum == 0)
        yield_helper();
}

```

In trap.c, yield\_helper() is called instead of yield() when the current process has exhausted the time quantum of the present round, to push the processes into the ReadyQueue2 if burst-time is not equal to runtime ( i.e. process still needs execution time ).

When the schedule() function is called, if the RQ1 is empty, processes in ascending order of burst times from RQ2 are inserted into RQ1. The scheduler proceeds by running the processes in RQ1 in the priority of burst time (least burst time corresponds to higher priority) for time quantum of that particular round (least burst time out of the processes present in the ready queue 1/2 ).

## Test Cases

We have three different test case files which have different burst time values for the processes. One of the files handles the case of multiple processes having the same burst time and another handles the case of multiple CPUs.

In each test case file, we use fork function to create child processes which are alternatively CPU bound and I/O bound. We store the PIDs of the processes in the order they were created in the process\_ids array. The I/O processes have a for loop which calls the sleep function and the CPU processes have a large computation. The child processes terminate after this.

We store the order in which the processes terminated in the output array using the wait function.

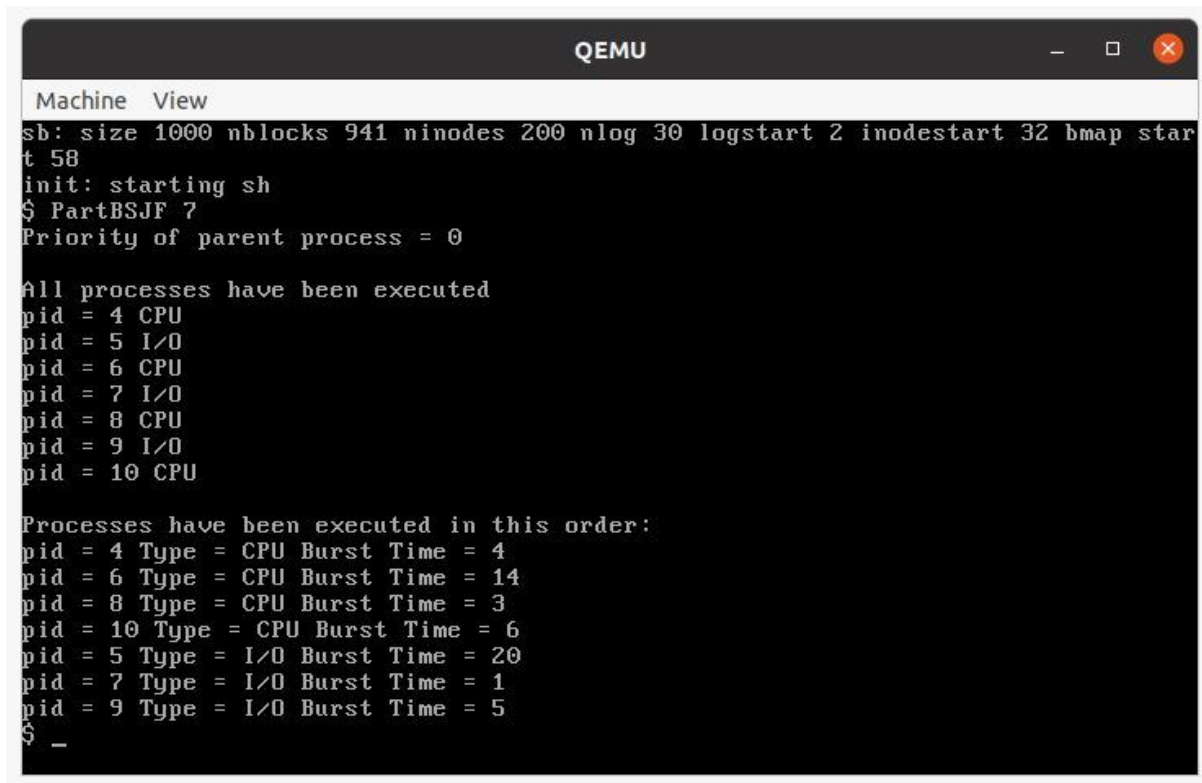
Depending on the scheduling algorithm used, the order of completion of the processes will be different for the same creation order.



After all the child processes have finished execution, we print the order in which they were created (`process_ids`) and in the order in which they finished execution (`output`)

### Round Robin algorithm (Default case)

In the Round Robin scheduler, since all the CPU processes and all the I/O processes are identical, the processes should be completed in the order they are initialized in.



```
QEMU
Machine View
sb: size 1000 nblocks 941 ninodes 200 nlog 30 logstart 2 inodestart 32 bmap star
t 58
init: starting sh
$ PartBSJF 7
Priority of parent process = 0

All processes have been executed
pid = 4 CPU
pid = 5 I/O
pid = 6 CPU
pid = 7 I/O
pid = 8 CPU
pid = 9 I/O
pid = 10 CPU

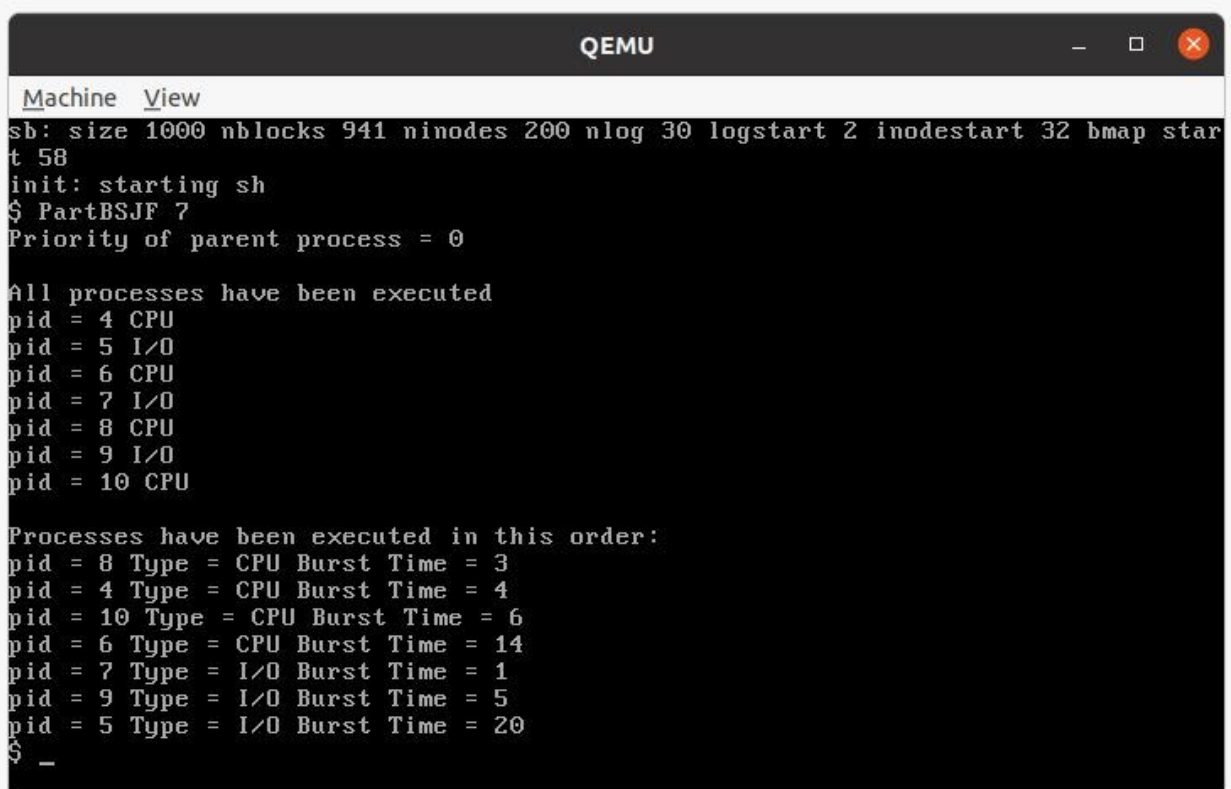
Processes have been executed in this order:
pid = 4 Type = CPU Burst Time = 4
pid = 6 Type = CPU Burst Time = 14
pid = 8 Type = CPU Burst Time = 3
pid = 10 Type = CPU Burst Time = 6
pid = 5 Type = I/O Burst Time = 20
pid = 7 Type = I/O Burst Time = 1
pid = 9 Type = I/O Burst Time = 5
$ _
```

## SJF algorithm

In Shortest Job First Scheduling, the process with the shortest burst time is completed first. The same is observed in our code.

It is different from Round robin which finishes in the same order as creation.

For example, the process with pid 8 was finished after the execution for the processes pid 4 and pid 6 in round robin scheduling even though its burst time was less than both of them while in SJF scheduler, it was executed first.



```
QEMU
Machine View
sb: size 1000 nblocks 941 ninodes 200 nlog 30 logstart 2 inodestart 32 bmap star
t 58
init: starting sh
$ PartBSJF 7
Priority of parent process = 0

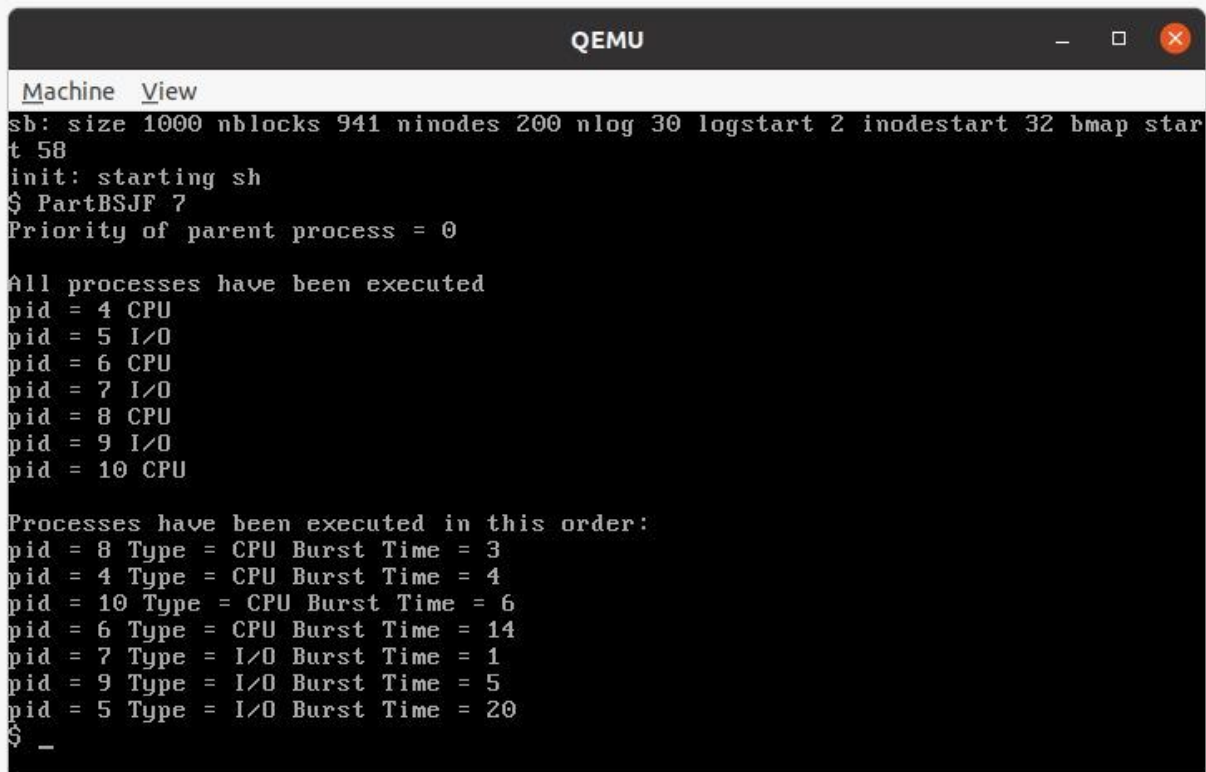
All processes have been executed
pid = 4 CPU
pid = 5 I/O
pid = 6 CPU
pid = 7 I/O
pid = 8 CPU
pid = 9 I/O
pid = 10 CPU

Processes have been executed in this order:
pid = 8 Type = CPU Burst Time = 3
pid = 4 Type = CPU Burst Time = 4
pid = 10 Type = CPU Burst Time = 6
pid = 6 Type = CPU Burst Time = 14
pid = 7 Type = I/O Burst Time = 1
pid = 9 Type = I/O Burst Time = 5
pid = 5 Type = I/O Burst Time = 20
$ _
```

## Hybrid algorithm

In hybrid algorithm, the processes are executed in round robin fashion where priority is determined as  $\text{priority} = \text{burst time}$ .

This algorithm avoids starvation for larger processes and gives equal opportunities to every process, the lowest burst time process is executed 1st in every round until its completion and thus there isn't any difference in the output of hybrid and SJF algorithms.



```
QEMU
Machine View
sb: size 1000 nblocks 941 ninodes 200 nlog 30 logstart 2 inodestart 32 bmap star
t 58
init: starting sh
$ PartBSJF 7
Priority of parent process = 0

All processes have been executed
pid = 4 CPU
pid = 5 I/O
pid = 6 CPU
pid = 7 I/O
pid = 8 CPU
pid = 9 I/O
pid = 10 CPU

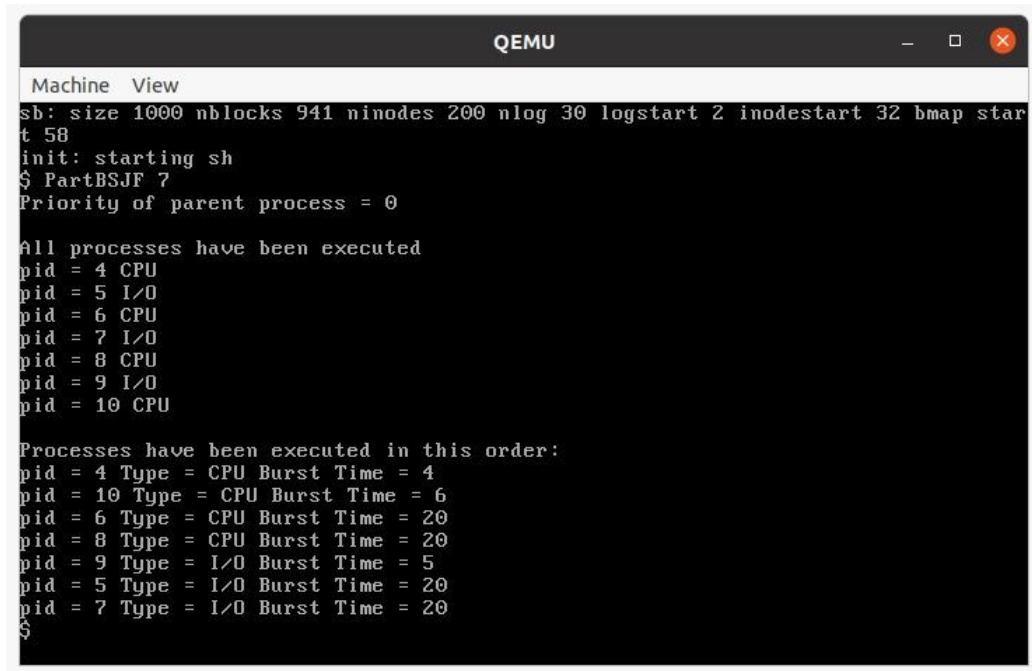
Processes have been executed in this order:
pid = 8 Type = CPU Burst Time = 3
pid = 4 Type = CPU Burst Time = 4
pid = 10 Type = CPU Burst Time = 6
pid = 6 Type = CPU Burst Time = 14
pid = 7 Type = I/O Burst Time = 1
pid = 9 Type = I/O Burst Time = 5
pid = 5 Type = I/O Burst Time = 20
$ _
```

## Corner Cases

Same Burst Time:

Whenever multiple processes with same burst time are in the ready queue, the scheduler picks the process which arrived first(or has the least PID in our case). Our implementation works for this case.

SJF scheduling:

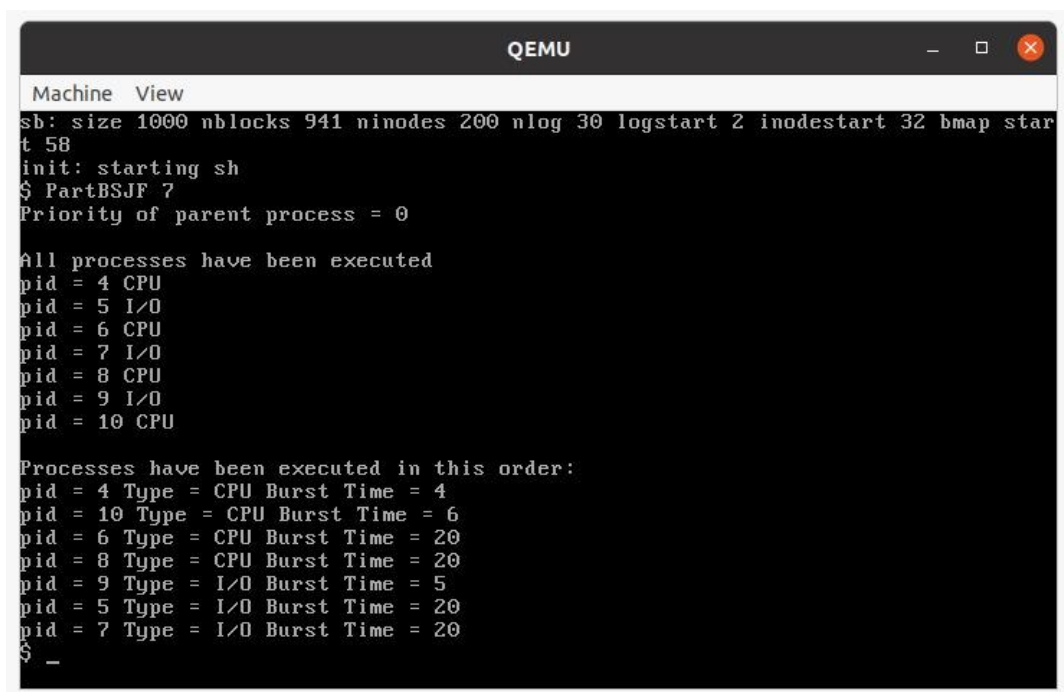


```
QEMU
Machine View
sb: size 1000 nblocks 941 ninodes 200 nlog 30 logstart 2 inodestart 32 bmap star
t 58
init: starting sh
$ PartBSJF 7
Priority of parent process = 0

All processes have been executed
pid = 4 CPU
pid = 5 I/O
pid = 6 CPU
pid = 7 I/O
pid = 8 CPU
pid = 9 I/O
pid = 10 CPU

Processes have been executed in this order:
pid = 4 Type = CPU Burst Time = 4
pid = 10 Type = CPU Burst Time = 6
pid = 6 Type = CPU Burst Time = 20
pid = 8 Type = CPU Burst Time = 20
pid = 9 Type = I/O Burst Time = 5
pid = 5 Type = I/O Burst Time = 20
pid = 7 Type = I/O Burst Time = 20
$
```

Hybrid Scheduling:



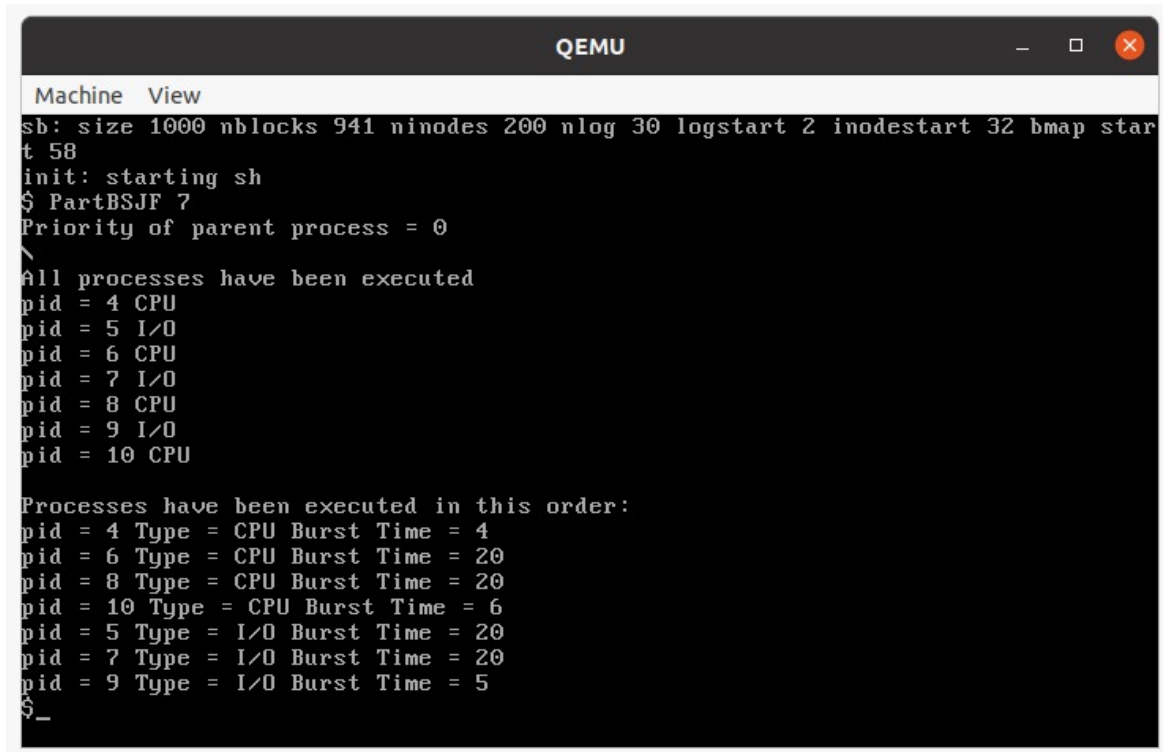
```
QEMU
Machine View
sb: size 1000 nblocks 941 ninodes 200 nlog 30 logstart 2 inodestart 32 bmap star
t 58
init: starting sh
$ PartBSJF 7
Priority of parent process = 0

All processes have been executed
pid = 4 CPU
pid = 5 I/O
pid = 6 CPU
pid = 7 I/O
pid = 8 CPU
pid = 9 I/O
pid = 10 CPU

Processes have been executed in this order:
pid = 4 Type = CPU Burst Time = 4
pid = 10 Type = CPU Burst Time = 6
pid = 6 Type = CPU Burst Time = 20
pid = 8 Type = CPU Burst Time = 20
pid = 9 Type = I/O Burst Time = 5
pid = 5 Type = I/O Burst Time = 20
pid = 7 Type = I/O Burst Time = 20
$ -
```

Multiple CPUs:

SJF scheduling:

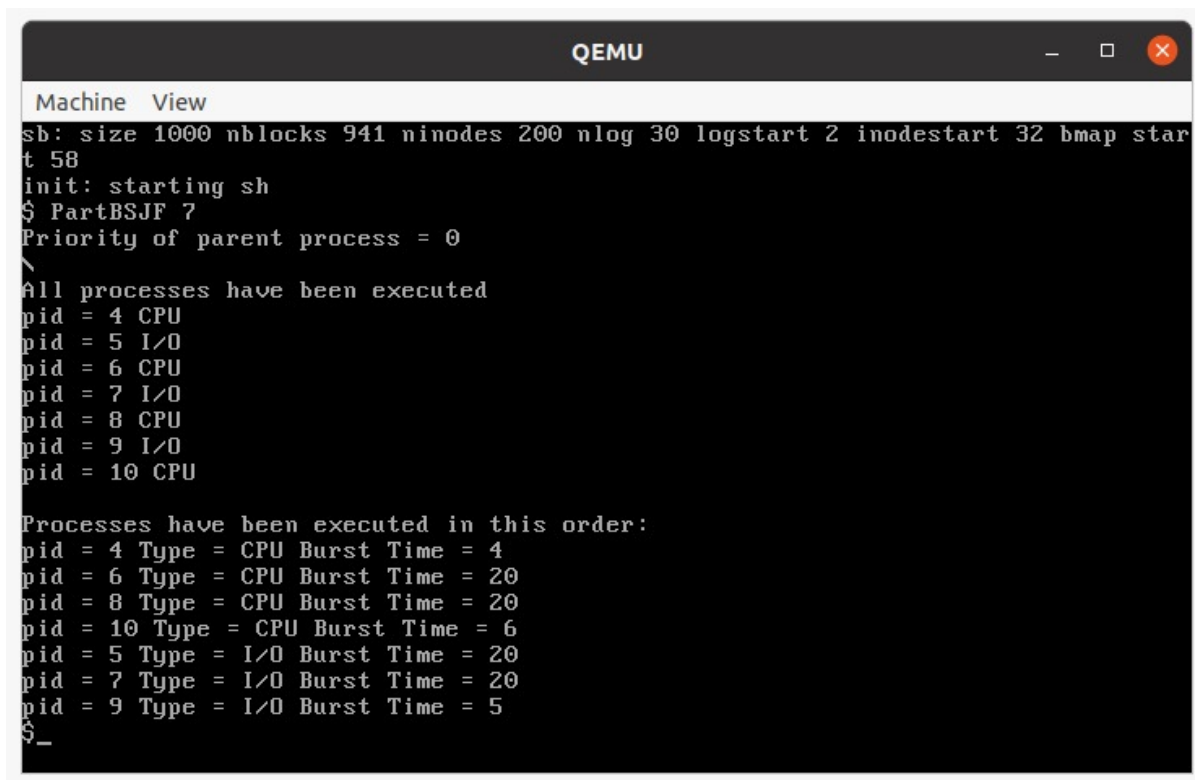


The screenshot shows a QEMU terminal window with a black background and white text. The window title is "QEMU". The text inside the terminal shows the execution of a program that simulates SJF scheduling on 10 processes. The processes are identified by their PID and type (CPU or I/O) and their burst times. The execution order is listed, showing that processes are executed in ascending order of their burst times. The processes are: pid = 4 CPU (burst time 4), pid = 6 CPU (burst time 20), pid = 8 CPU (burst time 20), pid = 10 CPU (burst time 6), pid = 5 I/O (burst time 20), pid = 7 I/O (burst time 20), and pid = 9 I/O (burst time 5). The terminal also shows the initial state of the system, including the size of the sb, the number of blocks, nodes, and log entries, and the fact that all processes have been executed.

```
Machine View
sb: size 1000 nblocks 941 ninodes 200 nlog 30 logstart 2 inodestart 32 bmap star
t 58
init: starting sh
$ PartBSJF 7
Priority of parent process = 0
\
All processes have been executed
pid = 4 CPU
pid = 5 I/O
pid = 6 CPU
pid = 7 I/O
pid = 8 CPU
pid = 9 I/O
pid = 10 CPU

Processes have been executed in this order:
pid = 4 Type = CPU Burst Time = 4
pid = 6 Type = CPU Burst Time = 20
pid = 8 Type = CPU Burst Time = 20
pid = 10 Type = CPU Burst Time = 6
pid = 5 Type = I/O Burst Time = 20
pid = 7 Type = I/O Burst Time = 20
pid = 9 Type = I/O Burst Time = 5
$ _
```

Hybrid scheduling



The screenshot shows a QEMU terminal window with a black background and white text. The window title is "QEMU". The text inside the terminal shows the execution of a program that simulates hybrid scheduling on 10 processes. The processes are identified by their PID and type (CPU or I/O) and their burst times. The execution order is listed, showing that processes are executed in ascending order of their burst times. The processes are: pid = 4 CPU (burst time 4), pid = 6 CPU (burst time 20), pid = 8 CPU (burst time 20), pid = 10 CPU (burst time 6), pid = 5 I/O (burst time 20), pid = 7 I/O (burst time 20), and pid = 9 I/O (burst time 5). The terminal also shows the initial state of the system, including the size of the sb, the number of blocks, nodes, and log entries, and the fact that all processes have been executed.

```
Machine View
sb: size 1000 nblocks 941 ninodes 200 nlog 30 logstart 2 inodestart 32 bmap star
t 58
init: starting sh
$ PartBSJF 7
Priority of parent process = 0
\
All processes have been executed
pid = 4 CPU
pid = 5 I/O
pid = 6 CPU
pid = 7 I/O
pid = 8 CPU
pid = 9 I/O
pid = 10 CPU

Processes have been executed in this order:
pid = 4 Type = CPU Burst Time = 4
pid = 6 Type = CPU Burst Time = 20
pid = 8 Type = CPU Burst Time = 20
pid = 10 Type = CPU Burst Time = 6
pid = 5 Type = I/O Burst Time = 20
pid = 7 Type = I/O Burst Time = 20
pid = 9 Type = I/O Burst Time = 5
$ _
```

So we can see from the screenshots above that our algorithm works for multiple CPUs too.