CS 344 : OPERATING SYSTEMS LAB 2

GROUP NUMBER : M14
GROUP MEMBERS:

Varun Bhardwaj (200123068)

Akshat Jain (200123005)

Jwalit Bharatkumar Devalia (200123026)

PART-A

1. getNumProc() and getMaxPid()

We implemented the **getNumProc**() and **getMaxPid**() system calls. We also ran the test case to check the correctness. There are three active processes at the time of execution of test case program ***partA_test1.c***, which is. ***init, sh, partA_test1***. The maximum PID is 3 initially. The MaxPID continuously increases because after termination of programs that were already executed, it remains in the process table. While looping over the process table for getting MaxPID, in which already executed programs are also present, the MaxPID increases every time we run the user test program.

Code:

```
int
getNumProc(void){
  struct proc *p;

  acquire(&ptable.lock);

  int activeProcesses = 0;

  for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
    if(p->state != UNUSED){
      activeProcesses++;
    }
  }
  release(&ptable.lock);
  return activeProcesses;
}
```

```c
int
getMaxPid(void){
  struct proc *p;

  acquire(&ptable.lock);
  int maxPID = -1e9;
  for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
    if(maxPID < (p -> pid)){
      maxPID = (p -> pid);
    }
  }
  release(&ptable.lock);
  return maxPID;
}
```

Output of getMaxPid() :

```
$ partA_test1
Total Number of Active Processes, as computed by getNumProc() system call: 3
Maximum PID among those Active Processes, as computed by getMaxPid() system call: 3
$ partA_test1
Total Number of Active Processes, as computed by getNumProc() system call: 3
Maximum PID among those Active Processes, as computed by getMaxPid() system call: 4
$ partA_test1
Total Number of Active Processes, as computed by getNumProc() system call: 3
Maximum PID among those Active Processes, as computed by getMaxPid() system call: 5
$ partA_test1
Total Number of Active Processes, as computed by getNumProc() system call: 3
Maximum PID among those Active Processes, as computed by getMaxPid() system call: 6
```

2. getProcInfo(pid, &processInfo)

Here we have implemented the system call **getProcInfo(pid, &processInfo).** It takes 2 arguments, **processInfo** a structure, containing information related to process and **pid** an integer, which is process ID. We also include the header files in **proc.c** and the corresponding structure in **user.h**.

We have added a new field in the **struct proc** called **numebr_of_context_switches** and it calculates the number of context switches of the process. It's initialised to zero when the process is brought to the **EMBRYO state** for the first time by **allocproc()** and its value increases when the process makes a context switch.

Code :

(in **proc.c)**

```
int
getProcInfo(int pid,struct processInfo* procInfo){
  struct proc *p;

  acquire(&ptable.lock);
  int isProcessPresent = -1;
  for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
    if(p -> pid == pid){
      isProcessPresent = 0;
      procInfo -> psize = p -> sz;
      procInfo -> numberContextSwitches = p -> number_of_context_switches;
      procInfo -> ppid = p -> parent -> pid;
    }
  }
  release(&ptable.lock);
  return isProcessPresent;
}
```

(in **proc.h)**

```
// 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 number_of_context_switches;         // Number of Times this proceess context
  int burst_time;              // CPU burst time of the process
};
```

Output of getProcInfo() :

```
$ partA_test2
PID     ParentPID     SIZE     Number of Context Switches
1       -1923918107       12288     13
2       1       16384     17
3       2       12288     7
```

3. set_burst_time(n) & get_burst_time()

set_burst_time(n) is used to set the burst time of the process to a user input value. We have included a new field, called **burst_time(int)** in the **proc(struct)**, which contains all the information about the process, to implement this functionality. **burst_time(int)** stores the burst time of a process. At the end of set_burst_time(n), yield() is called, so that after running this system call the scheduler is instantiated again and thus scheduling happens according to new burst times.

```
int
set_burst_time(int n){

  acquire(&ptable.lock);
  int BurstTimeSet = -1;
  mycpu()->proc->burst_time = n;
  BurstTimeSet = 0;
  release(&ptable.lock);
  yield();
  return BurstTimeSet;
}
```

get_burst_time() is used to get the burst time of the process at any instant of time. It is used to check if the value of burst time is correct or not. It's also used to check the correctness of the SJF and Hybrid scheduling algorithms.

Code : **Output :**

```
int
get_burst_time(){

  acquire(&ptable.lock);
  int burstTime = -1;
  burstTime = mycpu()->proc->burst_time;
  release(&ptable.lock);
  return burstTime;
}
```

```
$ partA_test3
Burst Time: 11
Burst Time: 14
Burst Time: 17
Burst Time: 3
Burst Time: 6
Burst Time: 9
Burst Time: 12
Burst Time: 15
Burst Time: 1
```

PART-B :

(i) SJF SCHEDULING ALGORITHM :-

By default in xv6, **Round Robin** Scheduler is used, which preempts the process after every clock cycle. yield() function in **trap.c** file necessitate the process to free up the CPU. Trap file calls the yield function at the end of each interrupt. Yield in turn calls **sched**, which calls **swtch(),** which saves the current context in **proc->context** and it also switches the scheduler context to previously saved state 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();
```

EXPLANATION OF SCHEDULING POLICY

We have implemented **Shortest Job First(SJF)** as the scheduling policy. In the SJF scheduler, the scheduler chooses the process which has the minimum burst time present in the Ready Queue and schedules it for execution. We have added an extra field in struct proc accounting for the burst time of the processes. The scheduler will use these burst times to schedule the processes for execution.

Code of SJF Scheduling Algorithm :

```
// // SJF scheduler
void
scheduler(void)
{
  struct proc *proc_to_be_executed ;
  struct cpu *c = mycpu();
  c -> proc = 0;

  struct proc* iterator;
  for(;;){
    // Enable interrupts on this processor.
    sti();

    proc_to_be_executed = 0;
    int min_time_so_far = 1e9; // It's the minimum time obtained so far.
    acquire(&ptable.lock);

    // Loop over process table looking for process to run.
    for(iterator = ptable.proc; iterator < &ptable.proc[NPROC]; iterator++){
      if((iterator -> state == RUNNABLE) && (iterator -> burst_time) < min_time_so_far){
        min_time_so_far = iterator -> burst_time;
        proc_to_be_executed = iterator;
      }
    }
```

```
    if(proc_to_be_executed != 0){
      c->proc = proc_to_be_executed;
      switchuvm(proc_to_be_executed);
      proc_to_be_executed->state = RUNNING;

      swtch(&(c->scheduler), proc_to_be_executed->context);
      (proc_to_be_executed -> number_of_context_switches)++;
      switchkvm();

      c->proc = 0;
    }
    release(&ptable.lock);

  }
}
```

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. **Locks are acquired** by the scheduler for reading the process table and finding a suitable process for execution

3. A **variable proc_to_be_executed** is defined and initialised to **zero**, which stores the process having minimum burst time while iterating through the process table.

4. Also, we initialise a variable **min_time_so_far = 1e9**. It is used to find the process having minimum burst time, so we initialise it maximum burst time possible which is **1e9**.
5. Then we iterate over the process table to check whether a process is in the ready queue or not. We also find the process having burst time less than the min burst time of the process found till now. If we get such a process then we update the value of **proc_to_be_executed** and **min_time_so_far** .

6. After the termination of the loop, if the value of **proc_to_be_executed** is still not zero, then we increment the number of **context switches** for the processes.

7. The value of **per-CPU variable proc** is set back to p i.e **c->proc = proc_to_be_executed** is executed and this variable indicates the current executing process on cpu is set to current scheduled process **proc_to_be_executed**.

8. **Switchuvm(p)** is used to set up the kernel stack of the process and switch to its page table.

9. We mark the process's state 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. Control gets transferred back to the scheduler and it switches back to the page table and kernel stack.

12. The value of **per-CPU variable proc** is set back to **0**.

13. Finally, the lock is also released.

TIME COMPLEXITY OF SCHEDULING ALGORITHM

SJF Scheduling Algorithm works in **O(N)** time where N is the number of processes in the process table. It essentially works in constant time as the maximum number of processes which can be present in the process table is **64(i.e. NPROC = 64) in xv6**.It loops over all the processes present in the process table to find the process having minimum burst time.

HANDLING CORNER CASES

CASE-I:DEFAULT BURST TIME

When the process is brought to EMBRYO state in the allocproc() function, the burst time of each process is set to 0. To ensure smooth working of the system, 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.

CASE-II:EQUAL BURST TIMES

FCFS Scheduling policy is used in case of two processes having the same burst time. It means that the process which came first, gets scheduled first.

EXPLANATION OF OUTPUT OF TEST CASES

CASE-I:RANDOM ORDER BURST TIMES

To all the child processes created using fork(), a random burst time is allocated. Dummy delay is also introduced so that all child processes could be created and scheduled at the same time. Following are the results of running the SJF and Round Robin scheduler.

Output of SJF algo:

```
All children executed successfully
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

Child Proceses 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
```

Output of Round-Robin algo:

```
All children executed successfully
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

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

CASE-II:BURST TIMES IN INCREASING ORDER

To all the child processes created using fork(), burst time is allocated in increasing order. Dummy delay is also introduced so that all child processes could be created and scheduled at the same time. Following are the results of running the SJF and Round Robin scheduler.

Output of SJF algo:

```
All children executed successfully
Child 0    pid 4    burst time = 3
Child 1    pid 5    burst time = 8
Child 2    pid 6    burst time = 13
Child 3    pid 7    burst time = 18
Child 4    pid 8    burst time = 23
Child 5    pid 9    burst time = 28
Child 6    pid 10    burst time = 33
Child 7    pid 11    burst time = 38
Child 8    pid 12    burst time = 43
Child 9    pid 13    burst time = 48

Child Proceses Exit order
pid 4     burst time = 3
pid 5     burst time = 8
pid 6     burst time = 13
pid 7     burst time = 18
pid 8     burst time = 23
pid 9     burst time = 28
pid 10     burst time = 33
pid 11     burst time = 38
pid 12     burst time = 43
pid 13     burst time = 48
```

Output of Round-Robin algo:

```
All children executed successfully
Child 0    pid 4    burst time = 3
Child 1    pid 5    burst time = 8
Child 2    pid 6    burst time = 13
Child 3    pid 7    burst time = 18
Child 4    pid 8    burst time = 23
Child 5    pid 9    burst time = 28
Child 6    pid 10    burst time = 33
Child 7    pid 11    burst time = 38
Child 8    pid 12    burst time = 43
Child 9    pid 13    burst time = 48

Child Proceses Exit order
pid 10     burst time = 33
pid 11     burst time = 38
pid 8     burst time = 23
pid 12     burst time = 43
pid 4     burst time = 3
pid 6     burst time = 13
pid 7     burst time = 18
pid 9     burst time = 28
pid 5     burst time = 8
pid 13     burst time = 48
```

CASE-III:BURST TIMES IN DECREASING ORDER

To all the child processes created using fork(), burst time is allocated in decreasing order. Dummy delay is also introduced so that all child processes could be created and scheduled at the same time. Following are the results of running the SJF and Round Robin scheduler.

Output of SJF algo:

```
All children executed successfully
Child 0    pid 4    burst time = 15
Child 1    pid 5    burst time = 14
Child 2    pid 6    burst time = 13
Child 3    pid 7    burst time = 12
Child 4    pid 8    burst time = 11
Child 5    pid 9    burst time = 10
Child 6    pid 10    burst time = 9
Child 7    pid 11    burst time = 8
Child 8    pid 12    burst time = 7
Child 9    pid 13    burst time = 6

Child Proceses Exit order
pid 13     burst time = 6
pid 12     burst time = 7
pid 11     burst time = 8
pid 10     burst time = 9
pid 9     burst time = 10
pid 8     burst time = 11
pid 7     burst time = 12
pid 6     burst time = 13
pid 5     burst time = 14
pid 4     burst time = 15
```

Output of Round-Robin algo:

```
All children executed successfully
Child 0    pid 4    burst time = 15
Child 1    pid 5    burst time = 14
Child 2    pid 6    burst time = 13
Child 3    pid 7    burst time = 12
Child 4    pid 8    burst time = 11
Child 5    pid 9    burst time = 10
Child 6    pid 10    burst time = 9
Child 7    pid 11    burst time = 8
Child 8    pid 12    burst time = 7
Child 9    pid 13    burst time = 6

Child Proceses Exit order
pid 4     burst time = 15
pid 13     burst time = 6
pid 12     burst time = 7
pid 6     burst time = 13
pid 7     burst time = 12
pid 8     burst time = 11
pid 9     burst time = 10
pid 10     burst time = 9
pid 11     burst time = 8
pid 5     burst time = 14
```

CASE-IV:I/O bound process

To all the child processes created using fork(), burst time is allocated in increasing order. Dummy delay is also introduced so that all child processes could be created and scheduled at the same time. Following are the results of running the SJF and Round Robin scheduler.

Output of SJF algo:

```
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
```

Output of Round-Robin algo:

```
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
```

(ii) HYBRID 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. In this case, we kept this part uncommented, contrary to what we did in SJF Scheduling Algorithm.

```
// 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();
```

Hybrid Scheduler Algorithm :

```
// Hybrid Scheduler
void
scheduler(void)
{
  struct proc *p ;
  struct cpu *c = mycpu();
  c -> proc = 0;
  int time_quantum = 1e9;

  int proc_to_be_executed_idx = 0;

  for(;;){
    // Enable interrupts on this processor.
    sti();

    struct proc *procArray[NPROC];
    int num_of_proc_in_procArray = 0;

    acquire(&ptable.lock);

    // Loop over process table looking for process to run.
    for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
      if(p->state != RUNNABLE){
        continue;
      }
      procArray[num_of_proc_in_procArray++] = p;
    }
```

```
    if(num_of_proc_in_procArray > 0){
      insertionSort(procArray, num_of_proc_in_procArray);
      // mergeSort(procArray, 0, num_of_proc_in_procArray - 1);

      // cprintf("Sorting of processes is done.\n");

      // for(int i = 0; i < num_of_proc_in_procArray; i++){
      //   cprintf("i : %d     procArray[i]'s PID : %d     \n", i, procArray[i] -> pid);
      // }

      if(time_quantum == 1e9 && procArray[0] -> burst_time != 0){
        time_quantum = procArray[0] -> burst_time;
      }

      (procArray[proc_to_be_executed_idx    %    num_of_proc_in_procArray] -> number_of_context_switches)++;

      c->proc = procArray[proc_to_be_executed_idx    %    num_of_proc_in_procArray];
      switchuvm(procArray[proc_to_be_executed_idx    %    num_of_proc_in_procArray]);
      procArray[proc_to_be_executed_idx    %    num_of_proc_in_procArray]->state = RUNNING;

      swtch(&(c->scheduler), procArray[proc_to_be_executed_idx    %    num_of_proc_in_procArray]->context);
      if(procArray[proc_to_be_executed_idx    %    num_of_proc_in_procArray] -> remaining_completion_time < time_quantum){
        procArray[proc_to_be_executed_idx    %    num_of_proc_in_procArray] -> remaining_completion_time = 0;
      }else{
        procArray[proc_to_be_executed_idx    %    num_of_proc_in_procArray] -> remaining_completion_time -= time_quantum;
      }
      proc_to_be_executed_idx++;
      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 hybrid of round robin and shortest job first (SJF) algorithm where there is a Ready Queue(implemented as an array) of processes which are RUNNABLE. This ready queue has processes in Increasing Order of their Burst Times. Here, we set the Time Quantum as the Burst Time of First Process present in this sorted Ready Queue. In xv6, there is no concept of burst time & remaining completion time as such, so we have to add an extra field in struct proc to keep account of burst time & remaining completion 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 variables ***time_quantum = 1e9*** (a large random value) and ***proc_to_be_executed_idx = 0***, where *proc_to_be_executed_idx* is the Index of the process to be executed from the Ready Queue array.

4. Now we declare a variable ***procArray*** of size **NPROC** and initialize another variable ***num_of_proc_in_procArray= 0*** because we have to find the number of processes residing in the procArray.

5. Then we iterate over the process table to check whether the process is RUNNABLE or not. Only processes that are RUNNABLE are added into procArray.

6. After exiting the loop, if the value of ***num_of_proc_in_procArray*** is not zero i.e. there is some process that needs CPU for execution. Then we sort the procArray based on burst times of processes present in it, using Insertion Sort(we could have used merge sort for O(nlogn) sorting complexity, but since size of procArray is restricted to NPROC, which itself equal to 64, using Insertion Sort is a better approach).
Now, if time_quantum has not been set, i.e, its value is still 1e9, then we set *time_quantum to be equal to the Burst time of procArray[0]*.

7. The number of context switches for the process to be executed is incremented.

8. Now the value of *per-CPU variable* ***proc*** is set back to **procArray[proc_to_be_executed_idx % num_of_proc_in_procArray]** i.e.

c->proc = procArray[proc_to_be_executed_idx%num_of_proc_in_procArray]

is executed and the variable indicating the current executing process on cpu is set to current scheduled process **procArray[proc_to_be_executed_idx%num_of_proc_in_procArray].**

9. Then **switchuvm(**procArray[proc_to_be_executed_idx%num_of_proc_in_procArray]**)** is called which sets up the process's kernel stack and switches to its page table.

10. Then the process *state* is marked as **RUNNING**.

11.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.

12. Now the *remaining_completion_time* of recently executed process is decremented by *time_quantum* (if the decremented value is *non-negative*. If it is going to be negative, then we set it as *zero*).

13. Now control is transferred back to the scheduler and hence it switches back to kernel stacks and page table.

14. After this the ***per-CPU variable proc*** is set back to **0**.

15. After this the lock is released so that other CPUs can also access the process table.

TIME COMPLEXITY OF SCHEDULING ALGORITHM

Hybird Scheduling Algorithm works in O(N^2) 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 build the ready queue in increasing order of its processes's burst time. it loops over all the processes present in the process table to find RUNNABLE processes and then sort them using Insertion Sort Algorithm. So the time complexity of the algorithm becomes O(N^2).

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

➢ CASE-I : CPU Bound Processes with BURST TIMES IN DECREASING ORDER

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 *Hybrid Scheduler* schedules processes according to their burst times, and gives CPU first to processes with shortest burst_time while *Round-Robin scheduler* gives chance to every process for a particular time quantum.

Output of Hybrid Scheduler :

```
$ test1 4
Random burst times
Burst times of parent process = 2

All children executed
Child 0.    pid 4    burst time = 10000
Child 1.    pid 5    burst time = 1000
Child 2.    pid 6    burst time = 200
Child 3.    pid 7    burst time = 2

Exit order
pid 7   burst time = 2
pid 6   burst time = 200
pid 5   burst time = 1000
pid 4   burst time = 10000
```

Output of Round-Robin Scheduler :

```
$ test1 4
Random burst times
Burst times of parent process = 2

All children executed
Child 0.    pid 4    burst time = 10000
Child 1.    pid 5    burst time = 2000
Child 2.    pid 6    burst time = 200
Child 3.    pid 7    burst time = 2

Exit order
pid 7   burst time = 2
pid 6   burst time = 200
pid 4   burst time = 10000
pid 5   burst time = 2000
```

➢ CASE-II : 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 Hybrid Scheduler schedules processes according to their burst times while Round-Robin scheduler gives chance to every process for a particular time quantum.

Output of Hybrid Scheduler :

```
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
```

Output of Round-Robin Scheduler :

```
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
```

---------- **End** ----------