COMP3230 Principle of Operating Systems

Midterm Examination

(Solution)

Problem 1. Multiple Choice Questions (45 marks)

Each question may have one or more than one incorrect statement to be chosen unless "Choose One" is specified. [**Marking scheme**: **3 marks** are awarded for each question with all incorrect statement(s) chosen and give corrections (No need for those correct statements, or if you choose **(5)** None of the above). We give **at most 1 mark as partial credit** for your answer with ALL correct choice(s) chosen but incomplete corrections (i.e., not all incorrect statements are corrected). **You get a zero mark if you only put your choice(s) without correcting the wrong statement(s).**

1. ( ) Which of the following statements about the use of **Linux command** in workbench is/are **INCORRECT**? (**1, 2, 3, 4**)
   (1) "lscpu" shows the number of physical cores in workbench. (X, it shows the number of virtual cores due to the hyper-threading)
   (2) With hyper-threading, we can run two threads on the same physical core at the same time and double the processing speed. (X, it won't double the speed as it only saves the context switching time, at most 30% improvement can be achieved)
   (3) "ps aux" shows all the processes created by a single user. (X, it should be "ps -u")
   (4) "top -u" shows all user-space processes created in workbench. (X, "-u" is set to show only the current user's processes, not for all processes running in the system)
   (5) None of the above (i.e., (1)-(4) are all **CORRECT**)

2. ( ) Which of the following statements about ***microkernel*** is **INCORRECT**? (Choose One) (**1**)
   (1) The microkernel OS consists of only the minimal set of vital functions of the operating system, thus the execution time of your program is much faster (X, you just move the code to user space, may have more overhead. This is a past midterm exam question. Read lecture note)
   (2) The primary overhead of the microkernel architecture is inter-process communication (IPC). (O)
   (3) The microkernel architecture is easily extendable, i.e., if any new services are to be added they are added to user address space. (O)
   (4) The microkernel architecture is more secure, i.e., if one service crashes it does not directly affect others. (O)

(5) None of the above (i.e., (1)-(4) are all **CORRECT**)

3. (   ) Which of the following statements about ***virtual machine*** (VM) and ***container*** is/are **INCORRECT**? (**4**)

   (1) Multiple containers can run on the same server by sharing a single OS kernel. (O.)

   (2) Each VM requires its own operating system. (O)

   (3) workbench is just a "container", not a real machine. (O, this is why we can change the number of CPU cores from 56 to 80 easily before the A1 submission deadline)

   (4) vmlinux is the virtual machine running on workbench. (X, vmlinux is the Linux kernel, a statically linked executable file (OS image), not a virtual machine)

   (5) None of the above (i.e., (1)-(4) are all **CORRECT**)

4. (   ) Which of the following statements about ***static*** and ***dynamic*** library is/are **INCORRECT**? (**1, 3**)

   (1) One key advantage of static libraries is that they are directly executable. (X, it does not have a main() function by itself and thus cannot be directly executed)

   (2) Every time you change or up-grade the static libraries, you have to recompile your application source code. (O)

   (3) Shared library is loaded by the operating system into the calling process' stack segment. (X, it is between head and stack. The exact location is determined by OS. Please see the figure in lecture note)

   (4) Static libraries use a lot more memory as they copy code into executable files. (O)

   (5) On Linux, the file names of static libraries usually end with a suffix of ".a", while those of shared libraries end with a suffix of ".so". (O)

5. (   ) Which of the following statements about **htop** is **INCORRECT**? (Choose One) **(4)**

   (1) In htop, RES column is the amount of physical memory this process is using. (O)

   (2) Right after a child process is created by fork(), the child should have the same VIRT as its parent. (O)

   (3) After the child process executes for a while, its VIRT and RES could be different from its parent's. (O, child/parent can malloc() their own memory)

   (4) If a process performs "malloc(4 * 1024 * 1024)", it's VIRT and RES will both increase by 4MB. (X, only the VIRT grows. If you don't use (e.g., read/write) the newly allocated memory, the RES, i.e., physical memory used by the process, never grows.)

   (5) None of the above (i.e., (1)-(4) are all **CORRECT**).

6. ( ) Which of the following statements about **System Calls** is/are **INCORRECT**? (**1, 2, 3, 4**)

(1) System call is a user defined function but running in kernel mode. (X, it is the OS provided interface)

(2) All system call functions are defined in GNU C Library (glibc). (X, the standard C library only provides wrapper functions for (most) system calls. There is no one-to-one correspondence. Additional info: The syscall table for x86_64 architecture can be found in arch/x86/entry/ syscalls/syscall_64.tbl)

(3) The integer value 80 in "INT $0x80" is a system call number. (x, INT is an assembly instruction which raises the (software) "interrupt")

(4) System calls overhead is much less than user-mode procedure calls. (x, actually more overhead, e.g., mode transition from user space to kernel space, registers contents have to be changed, etc.)

(5) None of the above (i.e., (1)-(4) are all **CORRECT**).

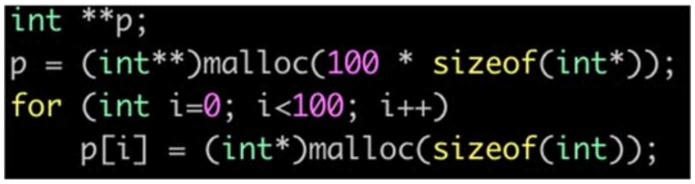
7. ( ) Which of the following statements about ***process address space*** is/are **INCORRECT**? (**2,3,5**)

(1) Global variables with non-zero initial value are stored in *data segment*. (O)

(2) Local variables with non-zero initial value are stored in *data segment* (X, stack)

(3) Uninitialized local static variables are stored in *stack segment*. (x, BSS)

(4) The pointer variable returned from malloc() is stored in *heap* or *stack segment*. (O, depending on the way it is declared/used. Here is an example to show pointer variables can be stored in heap segment created by the first malloc():



```
int **p;
p = (int**)malloc(100 * sizeof(int*));
for (int i=0; i<100; i++)
    p[i] = (int*)malloc(sizeof(int));
```

The "second-level pointer" is very useful when you need to allocate a large memory for a 2d matrix or determine the size of matrix when execution.

(5) Shared memory created by shmget(), is allocated at *heap segment*. (X, somewhere between heap and stack)

8. ( ) Which of the following statements about ***stack and heap*** in **Linux** is/are **INCORRECT**? (**2,4**)

(1) Stack access is usually faster than heap. (O, variables are allocated and freed automatically in stack)

(2) Variables stored in stack are usually accessed via *stack pointer* (%esp). (X, accessed via frame pointer %ebp)

(3) Arrays that may change size dynamically are better to be allocated on the heap. (O)

(4) The size of heap can be reported by the Linux *size* command. (X, size commands only gets the size information of text, data, BSS from a.out file, which is not in execution. The stack and heap sizes remain unknown. Check the screenshot we added in the lecture note)

(5) None of the above (i.e., (1)-(4) are all **CORRECT**)

9. ( ) Which of the following statements about **fork()** is/are **INCORRECT**? (**3,4**)

(1) The child process is created by duplicating the parent's task structure (struct task_struct) and page table. (O, most resource entries of the parent's task structure such as memory descriptor, file descriptor table, signal descriptors, and scheduling attributes are inherited by the child)

(2) During fork(), OS will duplicate the page table and mark all the page table entries *read-only*. (O, this has been emphasized several times in the lecture)

(3) Right after fork(), the same variable (e.g., int a) defined in parent and child will have the same virtual address but different physical addresses. (X, yes, same physical address right after fork and before any process updates/write the data)

(4) The child process needs to wait until parent process changes those read-only pages to writable before it can update the data. (x, child can update the data after it is created, no need to wait for the parent)

(5) None of the above (i.e., (1)-(4) are all **CORRECT**)

10. ( ) Which of the following statements about **wait()** and **waitpid()** is **INCORRECT** ? (Choose One) (**3**)

(1) The wait() function returns child pid on success. (O. If successful, wait returns the process ID of the child process. If unsuccessful, a -1 is returned.)

(2) The call wait(&status) is equivalent to waitpid(-1, &status, 0). (O)

(3) The call wait(NULL) waits for all child processes to terminate. (X. wait for any child to terminate).

(4) The call waitpid(-1, &status, WNOHANG) returns immediately if no child has exited. (O)

(5) None of the above (i.e., (1)-(4) are all **CORRECT**)

11. ( ) Which of the following statements about **zombie process** is/are **INCORRECT**? (**1, 2, 3, 4**)

(1) If a parent process crashes during the middle of execution, all its child processes that are still running become the zombie processes. (X, zombie processes are all dead, i.e., terminated)

(2) A zombie process is a running process, but whose parent process has finished or terminated. (X, it has completed execution (i.e., dead already))

(3) Zombie processes are harmful as they may still occupy memory (i.e., RSS is usually non-zero). (X, only keep an entry in the process table, RSS =0)

(4) Zombie process can be killed by "kill -9 [zombiePID]". (X, you cannot kill a zombie process as it is already dead.)

(5) None of the above (i.e., (1)-(4) are all **CORRECT**)

12. ( ) Which of the following statements about ***process scheduling*** is/are **INCORRECT** ? (**3**)

(1) Shortest Job First (SJF) may lead to process starvation. (O, with bad luck short processes are continually added, long processes in the queue have to keep waiting.)

(2) Round Robin (RR) is better than FCFS in terms of response time. (O)

(3) If the quantum time of Round Robin (RR) is very large, then it is equivalent to SJF. (X, FIFO)

(4) Multilevel Feedback Queue (MLFQ) is the most general CPU-scheduling algorithm. (O, read the summary table given in the lecture note)

(5) None of the above (i.e., (1)-(4) are all **CORRECT**)

13. ( ) Which of the following statements about **Highest Response Ratio Next (HRRN)** scheduling is **INCORRECT**? (Choose One) (**1, 3, 4**)

(1) HRRN is a preemptive version of Shortest Job First (SJF) algorithm. (X, HRRN is non-preemptive)

(2) HRRN adopts dynamic priorities, but it still favors the shorter jobs. (O)

(3) HRRN may lead to process starvation if there are many short jobs. (X, HRRN is designed to solve the starvation problem)

(4) HRRN has some overhead in tracking the remaining service time for each process in the ready queue. (X, just keep track of waiting time)

(5) None of the above (i.e., (1)-(4) are all **CORRECT**)

14. ( ) Which of the following statements about ***signal*** is/are **INCORRECT**? (**1,2,4**)

(1) kill() system call will immediately terminate the identified process (X, it just sends signal)

(2) signal() system call is used to send a signal to a process (X, it is used to register the signal handler)

(3) Signal handlers can be called anytime when the program is running. (O, signals can be received any time during the program execution, thus signal handlers can be called when the corresponding signals occur. Note: signals can be blocked by signal masks (This is out of scope, don't make it complicated.)

(4) SIGCHLD is sent from a parent process to a child process. (X, this signal is sent to a parent process whenever one of its child processes terminates or stops)

(5) None of the above (i.e., (1)-(4) are all **CORRECT**)

15. ( ) Which of the following statements about **multiprocessor scheduling** is/are **INCORRECT**? (**1,3**)

(1) Current Linux uses a global queue to schedule processes. (X, global + per-core queue)

(2) A long-running program could be scheduled to run on different CPU cores during its execution. (O)

(3) "taskset 0x7 [pid]" forces the process with PID=[pid] to run on core #7. (X, it means the process/thread can execute on core #0, #1, #2 as 0x7=0111)

(4) Gang scheduling is a good strategy for scheduling parallel programs (e.g., parallel 4-way merge sort) on multicore systems. (O)

(5) None of the above (i.e., (1)-(4) are all **CORRECT**)

Problem 2: Process Creation using Fork() (15 marks)

The code snippets below omit irrelevant details and error handling due to space constraint. We assume every fork() succeeds.

```
// we don't expect you to run the code on workbench
main() {
int count = 0;
pid_t pid1, pid2;

pid1 = fork();
count++;
if (pid1 == 0) {
    pid2 = fork();
    count++;
    if (pid2==0) {
       count=count+10;
       exit(0);
    }
} else {
   count++;
}
printf("Hello world!\n");
printf("Count is  %d\n", count);
}
```

University Number:

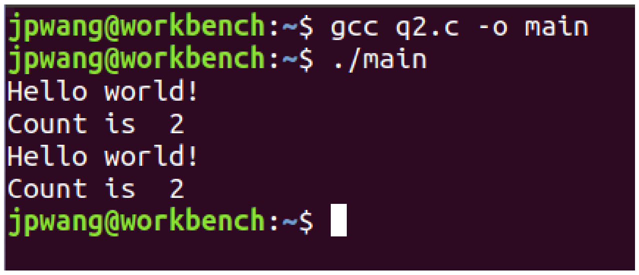
Answer the following two questions:

(a) **(7%)** How many times will the message “Hello world!\n” be displayed? Explain why?

(b) **(8%)** What will be the largest value of “count” displayed? Explain why?

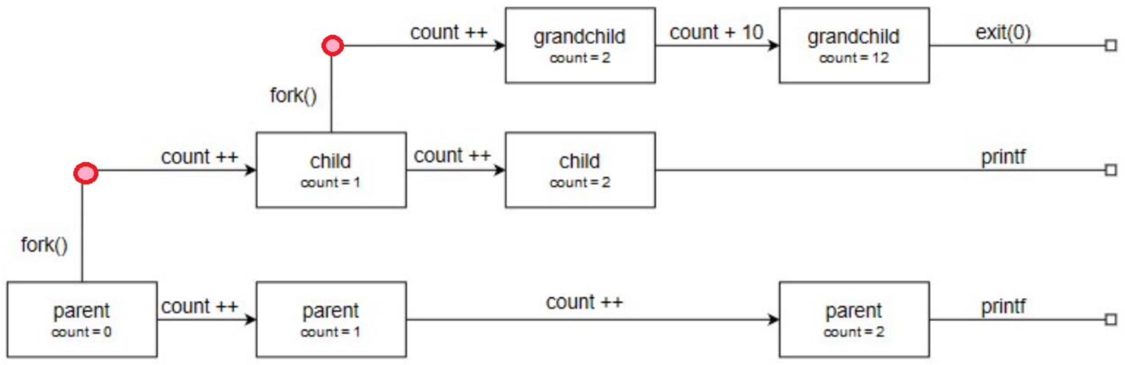
[**Note**: You need to show you understand how the above program work and explain how it leads to the result. If you only give the answer without explanation (or give wrong explanations), you will get a zero mark.]

Answer: Only 2. After first fork(): there are 2 processes and only the child calls second fork() to create a third process (let’s call it ‘grandchild’). “Hello world” will be printed by parent process and the child. The grandchild calls exit() after it increases the counter value. It won’t execute printf(“Hello world!\n”), nor printf(“Count is %d\n”, count). Marking scheme: without explicitly tell the grandchild is terminated after exit() and never reaches the last two lines of the code (two printf()), 2 marks are deducted.



```
jpwang@workbench:~$ gcc q2.c -o main
jpwang@workbench:~$ ./main
Hello world!
Count is  2
Hello world!
Count is  2
jpwang@workbench:~$ 
```

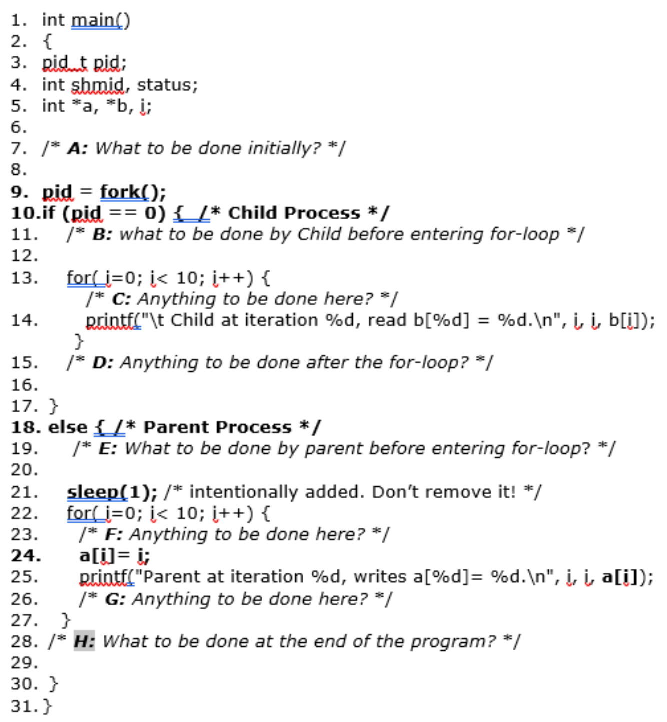
Answer: See the figure (from one student’s answer). Grandchild did execute the line count+=10 but it exits before printing anything. So you won’t see “Count is 12” printed on screen. Marking scheme: To get full marks, you need to explicitly mention the grandchild actually execute “count+=10” and has “count == 12”, but just not print it.



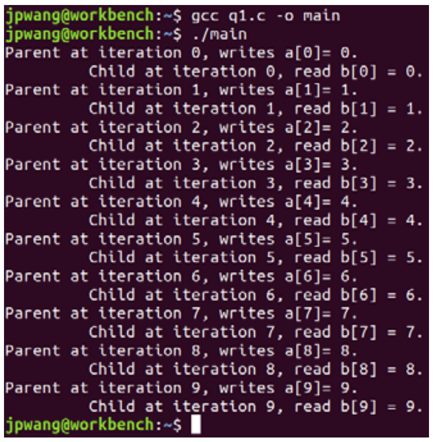
Problem 3: Signals and Shared Memory (20%)

In this program, you will create a shared memory segment to be shared by the parent and the child process and allow the child to read the updated values from the shared memory made by the parent.

Detailed Description: After the parent creates a child process, both the parent (writer) and child (reader) will attach the shared memory segment to their address space at ***a*** and ***b*** respectively. Each will then perform 10 iterations to access the shared memory as follows: in the *i*th iteration (0≤ i < 10), the parent process (writer) will first update the value of a[i] (i.e., a[i]=i), then let the child process (reader) read the updated value via reading b[i]. As discussed in Lecture 2, if there is no proper synchronization between the two processes, there could be a ***race condition*** since parent and child could run at different speed. To guarantee the child always reads the updated value written by the parent, you need to arrange **SIGSTOP** and **SIGCONT** signals in the given code template to make sure whenever the parent updates one entry in the shared memory the child should read the updated value afterwards before they advance to the next iteration (i → i++). To be more precise, the expected output and printing order are shown in the screenshot below:



```
int main()
{
pid_t pid;
int shmid, status;
int *a, *b, i;

/* A: What to be done initially? */

pid = fork();
if (pid == 0) { /* Child Process */
    /* B: what to be done by Child before entering for-loop */

    for( i=0; i< 10; i++) {
      /* C: Anything to be done here? */
      printf("\t Child at iteration %d, read b[%d] = %d.\n", i, i, b[i]);
    }
    /* D: Anything to be done after the for-loop? */

}
else { /* Parent Process */
    /* E: What to be done by parent before entering for-loop? */

    sleep(1); /* intentionally added. Don't remove it! */
    for( i=0; i< 10; i++) {
      /* F: Anything to be done here? */
      a[i]= i;
      printf("Parent at iteration %d, writes a[%d]= %d.\n", i, i, a[i]);
      /* G: Anything to be done here? */
    }
/* H: What to be done at the end of the program? */

}
}
```



```
jpwang@workbench:~$ gcc q1.c -o main
jpwang@workbench:~$ ./main
Parent at iteration 0, writes a[0]= 0.
         Child at iteration 0, read b[0] = 0.
Parent at iteration 1, writes a[1]= 1.
         Child at iteration 1, read b[1] = 1.
Parent at iteration 2, writes a[2]= 2.
         Child at iteration 2, read b[2] = 2.
Parent at iteration 3, writes a[3]= 3.
         Child at iteration 3, read b[3] = 3.
Parent at iteration 4, writes a[4]= 4.
         Child at iteration 4, read b[4] = 4.
Parent at iteration 5, writes a[5]= 5.
         Child at iteration 5, read b[5] = 5.
Parent at iteration 6, writes a[6]= 6.
         Child at iteration 6, read b[6] = 6.
Parent at iteration 7, writes a[7]= 7.
         Child at iteration 7, read b[7] = 7.
Parent at iteration 8, writes a[8]= 8.
         Child at iteration 8, read b[8] = 8.
Parent at iteration 9, writes a[9]= 9.
         Child at iteration 9, read b[9] = 9.
jpwang@workbench:~$
```

Some functions to be added to the given code template:

Shared memory creation: shmid = shmget(?, 10*sizeof(int),?). You need to figure out where to add this line and the right arguments to be added in the shmget() function call by yourself.

Shared memory attachment: Parent and child should attach the shared memory (shmid) to ***a*** and ***b*** respectively using (int *) shmat(shmid, 0, 0) and meet the two restrictions:

- Parent can only access the shared memory via pointer ***a***
- Child can only access the shared memory via pointer ***b***.

Shared memory de-attachment: Parent and child should separately de-attach the shared memory before they exit using shmdt().

Other system calls or library functions that are needed: kill(), getpid(), wait() (or waitpid()), exit(), shmctl(). **That is all, NOTHING ELSE.**

A Few More Notes:

- **No additional variables needed:** You are not allowed to add new variables in the given program code. "&status" could be used in wait() or waitpid() whenever needed.
- **No error checking needed:** We assume all the system calls and function calls always succeed and never fail.
- **No actual code execution:** We don't need you to write the complete code nor test the code in workbench. Some minor syntax errors are tolerable, but marks could be deducted if the error is severe.

Marking scheme:

- **(14%) Code part:** To get the full mark (14 marks), the code you added in **A-H** should be able to make the main program generate the exact outputs as shown in the screenshot. If some critical steps are missing or added at a wrong place, you are likely to get a zero mark as your program won't work correctly. In this case, no partial credit will be given.
- **(6%) Analysis Part:** You need to explain how your program would work correctly, particularly how the parent could ensure the child has firmly stopped and be instructed to advance to the next iteration at a proper time. We also expect you to explain how the shared memory segment was handled before the processes terminate. [**Note:** Please be specific about your answer. Don't waste your time in explaining the basic working mechanisms of signals or shared memory.]

Answer:

Code segment	C code to be added by you (leave it blank if nothing is needed)

A	// initialization
	shmid = shmget(IPC_PRIVATE, 10*sizeof(int), 0666|IPC_CREAT);

B	// child:
	b = (int *) shmat(shmid, 0, 0);

C	// child: for-loop **kill(getpid(), SIGSTOP);** **/* The students have been warned not to use getppid() */**

D	// child: after for-loop
	shmdt(**b**);

E	// parent **a** = (int *) shmat(shmid, 0, 0); /* 1% */
F	// parent: for-loop (before a[i]= i) **waitpid(pid, &status, WUNTRACED);** /* return if a child has stopped. It is fine to use wait()
G	// parent: for-loop (after a[i]= i) **kill(pid, SIGCONT);**

H	// end of the program
	wait(&status);
	shmdt(a);
	shmctl(shmid, **IPC_RMID**, 0);
	/* deleting the shared memory has to be done by only
	one process after making sure that none else
	will be using it */

Analysis Part (6%)	(You can have a look at Midterm_Q3.pdf posted in the Moodle. Some common errors made by students are discussed.

Problem 4: Process Scheduling (20 marks)

Consider the following processes, arrival times, and CPU processing time requirements:

Process ID (PID)	Arrival Time	CPU Time
1	0	4
2	1	4
3	4	3
4	8	2

Show the scheduling order for these processes under **(1) First-In-First-Out (FIFO)**, **(2) Round-Robin (RR)** with a **quantum = 1 time unit**, and **(3) Shortest-Remaining-Time First (SRTF)**. We assume the context switch overhead is 0. For each of the three scheduling algorithms, fill up the Gantt chart, calculate the *turnaround time* and *waiting time* for each process, and report the ***average turnaround time*** and ***average waiting time***. [**Note:** If additional tie-breaking rules are needed, you can use those we discussed in the Midterm Exam Guide. The first few entries have been filled for you. You can add more columns if needed.]

Answer:

FIFO: (5%)

Fill up the Gantt chart with the process ID that is running on the time slot.

P1	P1	P1	P1	P2	P2	P2	P2	P3	P3	P3	P4	P4		
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14

Time	Arrival	Running	Activity	Queue
0	P1	P1	P1 selected	
1	P2	P1		P2(4)
2		P1		P2(4)
3		P1		P2(4)
4	P3	P2	P2 selected	P3(3)
5		P2		P3(3)
6		P2		P3(3)
7		P2		P3(3)
8	P4	P3	P3 selected	P4(2)
9		P3		P4(2)
10		P3		P4(2)
11		P4	P4 selected	
12		P4		

University Number:

Process ID (PID)	Arrival Time	CPU Time	Completion Time	Turn Around Time	Waiting Time
1	0	4	4	4-0=4	4-4=0
2	1	4	8	8-1=7	7-4=3
3	4	3	11	11-4=7	7-3=4
4	8	2	13	13-8=5	5-2=3

Average Turn Around time = (4+7+7+5)/4 = 5.75
Average Waiting time = (0+3+4+3)/4 = 2.5

RR: (5%)
Fill up the Gantt chart with the process ID that is running on the time slot. If a new process arrives at the same time as a running process is preempted due to the time quantum expiration, the arriving process will always be placed ahead of the preempted process at the end of the ready queue.

P1	P2	P1	P2	P1	P3	P2	P1	P3	P2	P4	P3	P4		
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14

Time	**Arrival**	**Running**	**Activity**	**Queue**
0	P1	P1	P1 selected	
1	P2	P2	P1 expires, P2 selected	P1(3)
2		P1	P2 expires, P1 selected	P2(3)
3		P2	P1 expires, P2 selected	P1(2)
4	P3	P1	P2 expires, P1 selected, P2 and P3 are pushed into ready queue, P3 (arriving process) is ahead of P2 (preempted process)	P2(2)→P3(3)
5		P3	P1 expires, P3 selected, P1 pushed into queue	P1(1)→P2(2)
6		P2	P3 expires, P2 selected, P3 pushed into queue	P3(2)→P1(1)
7		P1	P2 expires, P1 selected, P2 pushed into queue	P2(1)→P3(2)
8	P4	P3	P1 finishes, P3 selected, P4 arrives and is pushed into queue	P4(2)→P2(1)
9		P2	P3 expires, P2 selected, P3 pushed into queue	P3(1)→P4(2)
10		P4	P2 finishes, P4 selected	P3(1)
11		P3	P4 expires, P3 selected	P4(1)
12		P4	P3 finishes, P4 selected	

Process ID (PID)	Arrival Time	CPU Time	Completion Time	Turn Around Time	Waiting Time
1	0	4	8	8-0=8	8-4=4
2	1	4	10	10-1=9	9-4=5

3	4	3	12	12-4=8	8-3=5
4	8	2	13	13-8=5	5-2=3

Average Turn Around time = (8+9+8+5)/4 = 7.5
Average Waiting time = (4+5+5+3)/4 = 4.25

SRTF: (5%)

Fill up the Gantt chart with the process ID that is running on the time slot. To break the tie (if needed), we assume an arriving process has a higher priority than the running process.

P1	P1	P1	P1	P3	P3	P3	P2	P4	P4	P2	P2	P2		
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14

Time	**Arrival**	**Running**	**Activity (RT: Remaining Time)**	**Waiting**
0	P1	P1	P1 selected	
1	P2	P1	RT[P2]=4 >RT[P1]=3, P1 selected	P2(4)
2		P1		P2(4)
3		P1		P2(4)
4	P3	P3	RT[P2]=4>RT[P3]=3, P3 selected	P2(4)
5		P3		P2(4)
6		P3		P2(4)
7		P2	P3 finishes, P2 selected	
8	P4	P4	RT[P2]=3>RT[P4]=2, P4 selected	P2(3)
9		P4		P2(3)
10		P2	P4 finishes, P2 selected	
11		P2		
12		P2		

Process ID (PID)	Arrival Time	CPU Time	Completion Time	Turn Around Time	Waiting Time
1	0	4	4	4-0=4	4-4=0
2	1	4	13	13-1=12	12-4=8
3	4	3	7	7-4=3	3-3=0
4	8	2	10	10-8=2	2-2=0

Average Turn Around time = (4+12+3+2)/4 = 5.25
Average Waiting time = (0+8+0+0)/4 = 2

---------------------- End of the exam paper ----------------------