Birla Institute of Technology & Science Pilani, Hyderabad Campus
First Semester 2019-2020
CS F372: Operating Systems
Mid Semester Examination (Regular) Solutions

1.a) In microkernel based OS architecture, all nonessential components are removed from the kernel and are implemented as system and user-level programs. Here, user services and kernel services are in separate address spaces. The result is a smaller kernel. Typically, microkernels provide minimal process and memory management, in addition to a communication facility. The main function of the microkernel is to provide communication between the user/client program and the various services that are also running in user space. Communication is provided through message passing. One benefit of the microkernel approach is that it makes extending the operating system easier. All new services are added to user space and do not require modification of the kernel. The microkernel also provides more security and reliability, since most services are running as user, rather than kernel processes. If a service fails, the rest of the operating system remains untouched. The performance of microkernels can suffer due to increased system-function overhead. Microkernels are comparatively slow in performance because of the message passing overhead from user space to kernel space.

In layered architecture, the operating system is broken into a number of layers (levels). The bottom layer (layer 0) is the hardware and the highest (layer N) is the user interface. An operating-system layer is made up of data and the operations that can manipulate those data. A typical operating-system layer, say layer M, consists of data structures and a set of routines that can be invoked by higher-level layers. Layer M, in turn, can invoke operations on lower-level layers. The main advantage of the layered approach is simplicity of construction and debugging. The layers are selected so that each uses functions (operations) and services of only lower-level layers. This approach simplifies debugging and system verification. The first layer can be debugged without any concern for the rest of the system, because it uses only the basic hardware (which is assumed correct) to implement its functions. Once the first layer is debugged, its correct functioning can be assumed while the second layer is debugged, and so on. If an error is found during the debugging of a particular layer, the error must be on that layer, because the layers below it are already debugged. Thus, the design and implementation of the system are simplified. Each layer is implemented only with operations provided by lower-level layers. A layer does not need to know how these operations are implemented; it needs to know only what these operations do. Hence, each layer hides the existence of certain data structures, operations, and hardware from higher-level layers. The major difficulty with the layered approach involves appropriately defining the various layers. Because a layer can use only lower-level layers, careful planning is necessary. Another problem with layered implementations is that they tend to be less efficient than other types because of the overhead involved in the execution of a single system call which may need to be transferred through the various layers.

1.b) The following are the different parts of a PCB:

- **Process state** - The state may be new, ready, running, waiting, halted, and so on.
- **Program counter** - The counter indicates the address of the next instruction to be executed for this process.
- **CPU registers** - The registers vary in number and type, depending on the computer architecture. They include accumulators, index registers, stack pointers, and general-purpose registers and any condition-code information. Along with the program counter, this state information must be saved when an interrupt occurs, to allow the process to continue correctly afterward.
- **CPU-scheduling information** - This information includes process priority, pointers to scheduling queues, and any other scheduling parameters.
- **Memory-management information** - This information may include information related to memory allocation of the process such items as the value of the base and limit registers and the page tables, or the segment tables, depending on the memory system used by the operating system.
- **Accounting information** - This information includes the amount of CPU and clock (real) time used, time limits, account numbers, job or process numbers, and so on.
- **I/O status information** - This information includes the list of I/O devices allocated to the process, a list of open files, and so on.

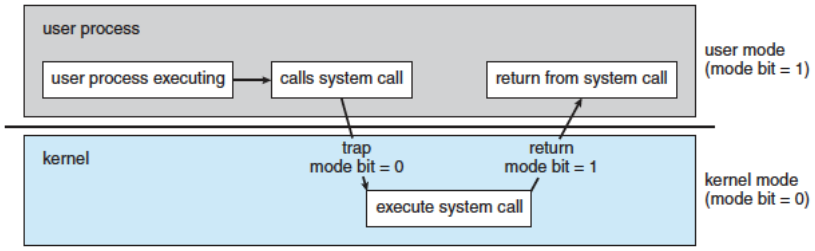
1.c) The procedure of starting a computer by loading the kernel is known as booting the system. On most computer systems, a small piece of code known as the bootstrap program or bootstrap loader locates the kernel, loads it into main memory, and starts its execution. Some computer systems, such as PCs, use a two-step process in which a simple bootstrap loader fetches

a more complex boot program from disk, which in turn loads the kernel. When a CPU receives a reset event like power up or reboot, the instruction register is loaded with a predefined memory location, and execution starts there. At that location is the initial bootstrap program. This program is in the form of read-only memory (ROM or EEPROM). The bootstrap program can perform a variety of tasks. Usually, one task is to run diagnostics to determine the state of the machine. This diagnostic is known as Power-On Self-Test (POST) and is the diagnostic testing sequence that a computer's basic input/output system (or "starting program") runs to determine if the computer keyboard, random access memory, disk drives, and other hardware are working correctly. In case of any error, the same is notified via a sequence of beeps. If the diagnostics pass, the program can continue with the booting steps. The bootstrap reads a single block at a fixed location (say block zero) from disk into memory and executes the code from that boot block. Boot block is a dedicated block usually at the beginning (first block on first track) of a storage medium that holds special data used to start a system. Some systems use a boot block of several physical sectors, while some use only one boot sector. If a disk contains a boot block it is called a boot disk. If a partition contains a boot block it is called a boot partition. The program stored in the boot block may be sophisticated enough to load the entire operating system into memory and begin its execution. More typically, it is simple code (as it fits in a single disk block) and knows only the address on disk and length of the remainder of the bootstrap program. Now that the full bootstrap program has been loaded, it can traverse the file system to find the operating system kernel, load it into memory, and start its execution. It is only at this point that the system is said to be running.

1.d) In asynchronous thread cancellation, one thread immediately cancels the target thread. The difficulty with asynchronous cancellation occurs in situations where resources have been allocated to a canceled thread or where a thread is canceled while in the midst of updating data it is sharing with other threads. Often, the operating system will reclaim system resources from a canceled thread but will not reclaim all resources. Therefore, canceling a thread asynchronously may not free a necessary system-wide resource.

In deferred cancellation, the target thread periodically checks whether it should terminate, allowing it an opportunity to terminate itself in an orderly fashion. Here, one thread indicates that a target thread is to be canceled, but cancellation occurs only after the target thread has checked a flag/condition to determine whether or not it should be canceled. The thread can perform this check at a point at which it can be canceled safely. Cancellation occurs only when a thread reaches a cancellation point.
In Pthreads, thread cancellation is initiated using the `pthread_cancel()` function. One technique for establishing a cancellation point is to invoke the `pthread_testcancel()` function. If a cancellation request is found to be pending, a function known as a cleanup handler is invoked. This function allows any resources a thread may have acquired to be released before the thread is terminated.

2.a) OS has two separate modes of operation: user mode and kernel mode. A bit, called the mode bit, is added to the hardware of the computer to indicate the current mode: kernel (0) or user (1). With the mode bit, we can distinguish between a task that is executed on behalf of the operating system and one that is executed on behalf of the user. When the computer system is executing on behalf of a user application, the system is in user mode. However, when a user application requests a service from the operating system (via a system call), the system must transition from user to kernel mode to fulfill the request. At system boot time, the hardware starts in kernel mode. The operating system is then loaded and starts user applications in user mode. When a trap or interrupt occurs, the hardware switches from user mode to kernel mode (that is, changes the state of the mode bit to 0). The system always switches to user mode (by setting the mode bit to 1) before passing control to a user program. The dual mode of OS operation provides us with the means for protecting the operating system from errant users. We accomplish this protection by designating some of the machine instructions that may cause harm as privileged instructions. The hardware allows privileged instructions to be executed only in kernel mode. If an attempt is made to execute a privileged instruction in user mode, the hardware does not execute the instruction but rather treats it as illegal and traps it to the operating system.



2.b) A process that has terminated, but whose parent has not yet called `wait()`, i.e., the process table entry for the process still exists is known as a zombie process. All processes transition to this state when they terminate, but generally they exist as zombies only briefly. Once the parent calls `wait()`, the process identifier of the zombie process and its entry in the process table are released.

```
int main()
{
    pid_t pid = fork();
    if (pid > 0) {
        sleep(10);
    }
    else{
        printf("\n%d",getpid());
    }
    return 0;
}
```

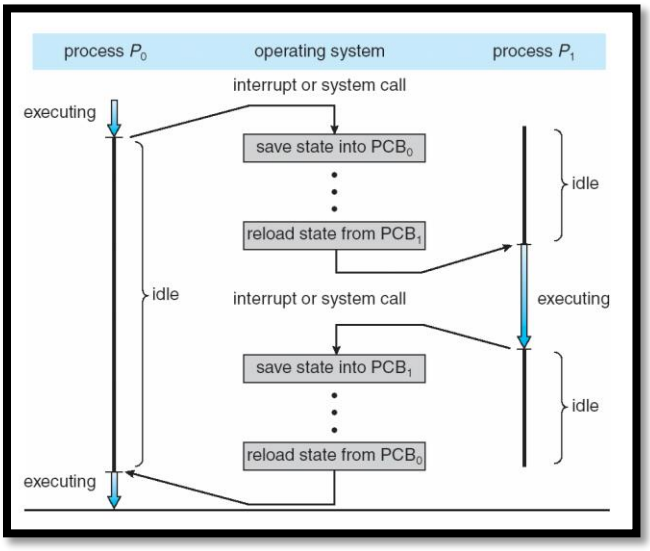
A zombie process can be searched using the following command - `ps -el | grep "Z"` or `ps -el | grep "defunct"`.

A process whose parent has terminated is known as an orphan process. Linux and UNIX address this scenario by assigning the `init` process as the new parent to orphan processes. The `init` process periodically invokes `wait()`, thereby allowing the exit status of any orphaned process to be collected and releasing the orphan's process identifier and process-table entry.

```
int main()
{
    pid_t pid = fork();
    if (pid > 0){
    }
    else{
       sleep(20);
    }
     return 0;
}
```

2.c) Parallelism refers to that aspect of a system where it can perform more than one task simultaneously. In contrast, concurrency supports more than one task by allowing all the tasks to make progress. Thus, it is possible to have concurrency without parallelism. On a single processor system, it is possible to achieve concurrency but not parallelism, only an illusion of parallelism can be created.

2.d) Switching the CPU to another process is known as context switching. This requires performing a state save of the current process and a state restore of a different process. The context is represented in the PCB of the process. It includes the value of the CPU registers, the process state and memory-management information. When a context switch occurs, the kernel saves the context of the old process in its PCB and loads the saved context of the new process scheduled to run.



2.e) Under direct communication, each process that wants to communicate must explicitly name the recipient or sender of the communication. In this scheme, the `send()` and `receive()` primitives are defined as:

- `send(P, message)` — Send a `message` to process `P`.
- `receive(Q, message)` — Receive a `message` from process `Q`.

With indirect communication, the messages are sent to and received from mailboxes, or ports. A mailbox is an object into which messages can be placed by processes and from which messages can be removed. Each mailbox has a unique identification. A process can communicate with another process via a number of different mailboxes, but two processes can communicate only if they have a shared mailbox. The `send()` and `receive()` primitives are defined as follows:

- `send(A, message)` — Send a `message` to mailbox `A`.
- `receive(A, message)` — Receive a `message` from mailbox `A`.

3.a)

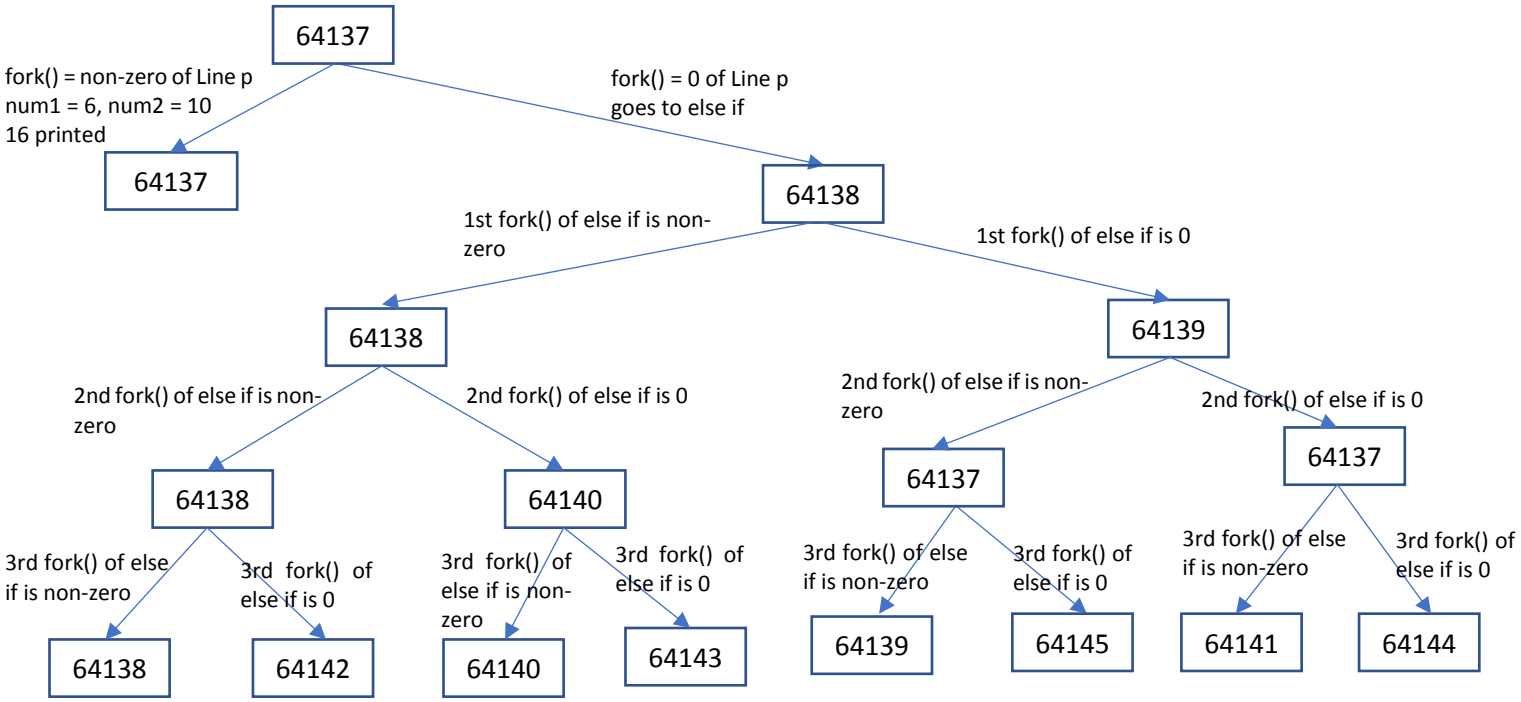
```
#include<stdio.h>
#include<unistd.h>
#include<sys/wait.h>
#include<string.h>
int main()
{
    int fds1[2], fds2[2];
    pid_t pid1, pid2;
    pipe(fds1);
    pid1=fork();
    if(pid1 > 0) //Parent Process
    {
        printf("\nThis is the parent with parent PID %d\n", getppid());
        char buf1[100] = "Houseguest";
        close(fds1[0]);
        write(fds1[1], buf1, strlen(buf1)+1);
        wait(NULL);  printf("Parent Exiting...\n");
    }
    else if(pid1 == 0)
    {
       char buf2[100];
       close(fds1[1]);
       read(fds1[0], buf2, 100);
       pipe(fds2);
       pid2 = fork();
       if(pid2 == 0)
       {
          printf("\nThis is the grand child with parent PID %d\n", getppid());
          char buf3[100] = "SecurityCall";
          close(fds2[0]);
          write(fds2[1], buf3, strlen(buf3)+1); printf("Grand Child Exiting...\n");
       }
       else{
          printf("\nThis is the child with parent PID %d\n", getppid());
          char buf4[100];
          close(fds2[1]);
          read(fds2[0], buf4, 100);
          wait(NULL);
          int l=strlen(buf2)>strlen(buf4)?strlen(buf2)-strlen(buf4):strlen(buf4)-
strlen(buf2);
          printf("The difference in string lengths is %d\n", l);
          printf("Child Exiting...\n");
       }
    }
    return 0;

}
```

3.b)

(i) A sample output is

```
16 64137
25 64138
25 64139
25 64142
25 64141
25 64140
25 64143
10 64144
25 64145
```

(ii) Each process has its own copy of `x`, `num1` and `num2`.



64144 executes the `else` statement and hence prints `num2 = 10`.
All the remaining leaf nodes (64138, 64142, 64140, 64143, 64139, 64145 and 64141) do the following.
For 1st call to `meth()` inside else if:
`num2 = 10`
Hence, `y = 10`
`x = 5 + 10 = 15`
`num1 = 15`

For 2nd call to `meth()` inside else if:
`num2 = 10`
Hence, `y = 10`
Since `x` is static, the previous value of `x` obtained from the 1st call to `meth()` will persist across function calls.
So, `x = 15 + 10 = 25`
`num1 = 25`

4.a)

```
#include <pthread.h>
#include <stdio.h>
#include <unistd.h>

void *runner(void *param)
{
   execlp("/bin/rcp", "rcp", "a.txt", "b.txt", NULL);
}

int main()
{
   pthread_t tid;
   pthread_attr_t attr;
   pthread_attr_init(&attr);
   pthread_create(&tid, &attr, runner, "a.txt");
   pthread_join(tid, NULL);
}
```

4.b) Say the 1st process to enter CS is P_i. P_i executes `test_and_set()`. As a result both `lock` and `enter` become `true`. So, `temp` = `false`. So, in the first iteration of the while loop, `false` is returned and P_i breaks out of the while loop and enters CS. This is how the 1st process enters CS.

Assume that P_i is currently executing CS. `lock` = `true`, `enter` = `true`. Now, P_j wants to enter CS. P_j executes `test_and_set()` in the first iteration of the while loop. As a result, `lock` = `true` and `enter` = `false`. Thus, `temp` = `true` and `true` is returned. Hence, P_j executes the second iteration of the while loop. Now, `lock` is set to `true` and `enter` is set to `true`. So, `temp` = `false`. Since `false` is returned, P_j breaks out of the while and enters CS. Thus, both P_i and P_j simultaneously execute CS thereby violating mutual exclusion.

Suppose P_i wants to enter CS and currently no other process is executing in CS. So, `lock` = `false` and `enter` = `false` since the last process to exit CS will have executed `reset()`. P_i executes `test_and_set()`, sets `lock` and `enter` to `true` and `temp` to `false` and thus breaks out of the while loop. Therefore, P_i enters CS. Hence progress is satisfied.

4.c) The given execution sequence does not lead to a deadlock.

Timestamp	Operation of P0	Operation of P1	X	Y	Z	State of Processes
t0	wait(X)	-	0	1	1	P0 continues, P1 continues
t1	wait(Y)	wait(Z)	0	0	0	P0 continues, P1 continues
t2	-	signal(Z)	0	0	1	P0 continues, P1 continues
t3	-	wait(Y)	0	0	1	P0 continues, P1 engages in busy waiting
t4	signal(X)	-	1	0	1	P0 continues, P1 is in busy waiting
t5	wait(Z)	-	1	0	0	P0 continues, P1 is in busy waiting
t6	signal(Z)	-	1	0	1	P0 continues, P1 is in busy waiting
t7	signal(Y)	Here P1 decrements Y after it is incremented by P0	1	0	1	P0 continues, P1 unblocks and continues with its remaining operations
t8	-	wait(X)	0	0	1	P1 continues
t9	-	signal(Y)	0	1	1	P1 continues
t10	-	signal(X)	1	1	1	P1 continues

4.d) (i)

Timestamp	Operation of P0	Operation of P1	X	Y	Z	State of Processes
t0	wait(X)	-	0	1	1	P0 continues, P1 continues
t1	wait(Y)	-	0	0	1	P0 continues, P1 continues
t2	-	wait(Z)	0	0	0	P0 continues, P1 continues
t3	signal(Y)	-	0	1	0	P0 continues, P1 continues
t4	-	wait(X)	0	1	0	P0 continues, P1 engages in busy waiting
t5	wait(Z)	-	0	1	0	P0 engages in busy waiting, P1 is in busy waiting, Deadlock
t6	-	signal(Z)				Deadlock
t7	-	wait(Y)				Deadlock
t8	signal(X)	-				Deadlock
t9	-	signal(Y)				Deadlock
t10	signal(Z)	signal(X)				Deadlock

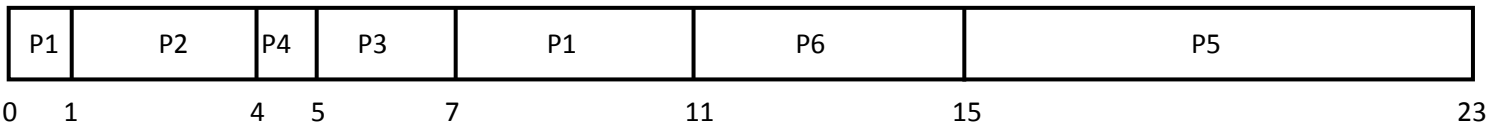
After timestamp **t5**, both processes engage in busy waiting and since neither can proceed with the remaining operations, **signal(X)** and **signal(Z)** cannot be performed thereby, leading to a deadlock.

ii) If at **t3**, **P0** had performed **signal(X)** instead of **signal(Y)** and at **t8**, **P0** omits performing **signal(Y)**, then the execution sequence would have been

Timestamp	Operation of P0	Operation of P1	X	Y	Z	State of Processes
t0	wait(X)	-	0	1	1	P0 continues, P1 continues
t1	wait(Y)	-	0	0	1	P0 continues, P1 continues
t2	-	wait(Z)	0	0	0	P0 continues, P1 continues
t3	signal(X)	-	1	0	0	P0 continues, P1 continues
t4	-	wait(X)	0	0	0	P0 continues, P1 continues
t5	wait(Z)	-	0	0	0	P0 engages in busy waiting, P1 continues
t6	-	signal(Z)	0	0	0	P0 unblocks and continues, P1 continues
t7	-	wait(Y)	0	0	0	P0 continues, P1 engages in busy waiting
t8	-	-	0	0	0	P0 continues, P1 in busy waiting
t9	-	signal(Y)	0	0	0	P0 continues, P1 in busy waiting
t10	signal(Z)	signal(X)	0	0	1	P0 continues, P1 in busy waiting

Subsequently, **P0** finishes execution and terminates but **P1** starves.

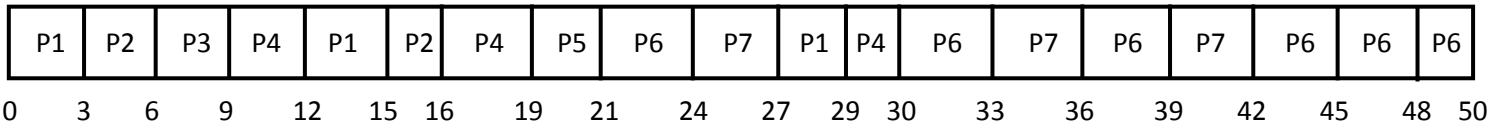
5.a)



Waiting of time for each process:
P1 → 0 + 6 = 6 ms
P2 → 0 ms
P3 → 2 ms
P4 → 0 ms
P5 → 9 ms
P6 → 4 ms

Avg. waiting time = (6 + 0 + 2 + 0 + 9 + 4)/6 = 21/6 = 3.5 milliseconds

5.b)



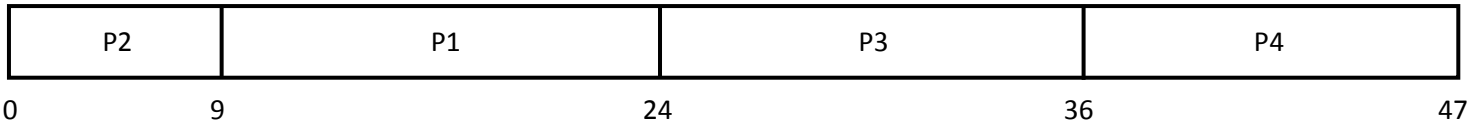
P1 → 0 + 9 + 12 = 21 ms
P2 → 3 + 9 = 12 ms
P3 → 4 ms
P4 → 6 + 4 + 10 = 20 ms
P5 → 6 ms
P6 → 6 + 6 + 3 + 3 = 18 ms
P7 → 6 + 6 + 3 = 15 ms

Avg. waiting time = (21 + 12 + 4 + 20 + 6 + 18 + 15)/7 = 96/7 = 13.7142 milliseconds

(ii)
TAT calculation:
P1 → 21 + 8 + 0.09 x 10 = 29.9 ms
P2 → 12 + 4 + 0.09 x 5 = 16.45 ms
P3 → 4 + 3 + 0.09 x 2 = 7.18 ms
P4 → 20 + 7 + 0.09 x 11 = 27.99 ms
P5 → 6 + 2 + 0.09 x 7 = 8.63 ms
P6 → 18 + 17 + 0.09 x 18 = 36.62 ms
P7 → 15 + 9 + 0.09 x 15 = 25.35 ms

Total TAT = 152.12 ms

5.c)



Waiting time of P4 → 36 - 2 = 34 milliseconds

****