Q1. [1.5 marks x 5 = 7.5 marks].

Write short textual answers. Drawings are not allowed.

a) Discuss why dual mode of operation is utilized in a multitasking operating system.
OS kernel needs control over all user processes to preempt them, thus allowing them equal (and fair) access to all hardware resources. This is only possible if there are two modes of operations: a **kernel mode** to access all resources with the help of CPU's supervisory mode, and a **user mode** to limited user programs their own allocated resources.

b) Consider monolithic and microkernel operating system structures. **Which one executes a system call that uses file system and device drivers faster than the other and why?**
Microkernel runs file system and device drivers as user mode services. Microkernel talks to them using message passing. Monolithic kernel as single address space and thus can directly execute file system and devices driver code. Therefore, a system call that utilizes these services generates many messages which takes time.

c) Suppose a process P0 is executing. **What will happen if the next instruction is: i) a read () system call, ii) an interrupt?** Write concise steps for both. Avoid missing key steps or writing them out of order.
i) read() system call is executed (few assembly instructions) → switch to kernel mode to execute kernel code → P0 relinquishes the CPU (voluntarily) → waits in the device queue against which read is requested (~10s of millisecond for HDD) → when the device complete the requested number of bytes P0 is put back in the ready queue → mode switch to user mode → the instruction after the read() call will be executed on the next scheduling of P0.
ii) The current instructions is allowed to complete → process context is saved → mode switch to kernel mode → interrupt vector is used to lookup the address of interrupt service routing (ISR) → kernel executes the lookup ISR (sometime further interrupts are disabled) → mode switch back to user code → the interrupted process is resumed thus it executed next instruction.

Give only labelled diagrams (Use pencil only). *Note: textual answers will not be graded.*

d) For two executing processes, **illustrate how operating system will stop execution of process P0 and allow P1 to execute**. Show all states these processes would take at different times in your diagram. Use technical terms in labels and specify all steps to get full scores.

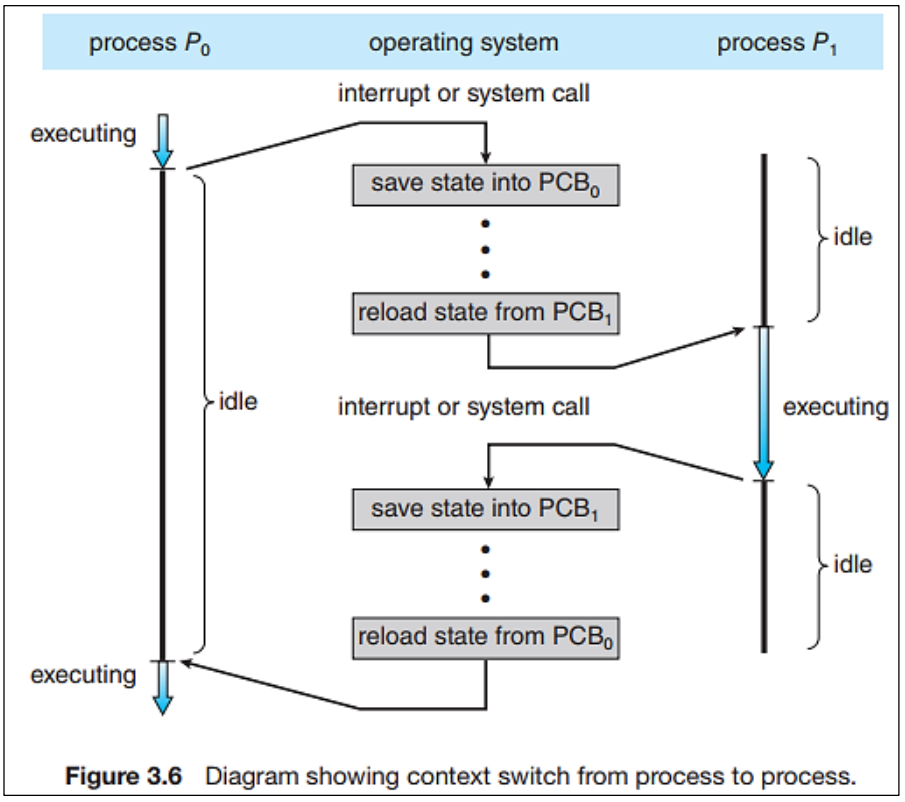


Figure 3.6 Diagram showing context switch from process to process.

e) Illustrate different (four or more) ways **a process can wait in different queues** after leaving the CPU till admitted back in the ready queue. Assume a multitasking operating system.

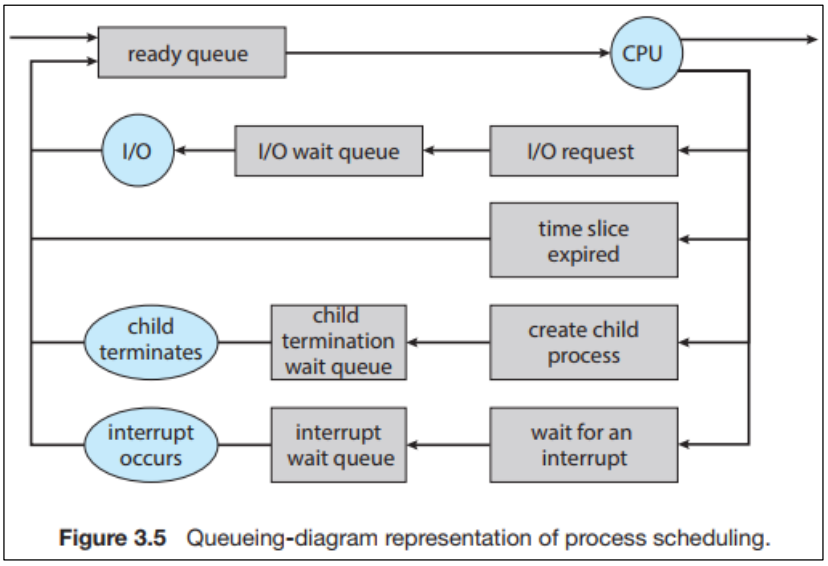


Figure 3.5 Queueing-diagram representation of process scheduling.

Q2. [1 marks + 2.5 marks + 4 marks = 7.5 marks]

Understanding and Design

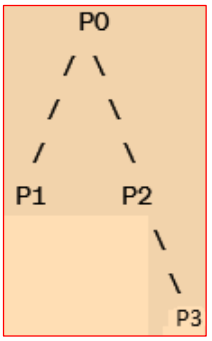
a) Suppose you need to implement an integer circular buffer of size 50 for a producer-consumer scenario. Write **one major reason to reject named pipes** and use shared memory instead. [1]

Named pipes use files to store shared contents and thus very slow (~10 milliseconds vs ~10 nanoseconds).

b) Suppose a process executes **fork () system call 2 times**. Draw a labelled diagram with meaningful hints to reflect your understanding of all the resultant processes along with their linkage. Explain which processes should issue wait () system for correct behavior. No grading without appropriate hints to show your understanding. [1.5 + 1]

Process hierarchy:

- 1st fork () creates P2 and controls return to 2nd fork () in P0, execution of which creates P1.
- The control returns in P2 to 2nd fork (), execution of which creates P3.



Wait():

- Process P0 should issue the wait () system call to wait for the termination of each of its child processes (P1 and P2).
- Process P2 should issue the wait () system call to wait for the termination of its child process P3.

Implementation (Use pencil only)

c) Write c language snippet that runs both i) **ls -al** command and ii) an executable file **/bin/foo** as two concurrent processes. Use a minimum number of processes. Error check for system call failures. The order of execution of processes is not important. [2+2=4]

```
6   int main() {
7       pid_t child_pid;
8       if ((child_pid = fork()) == 0) { // concurrent proces # 1
9           execl("/bin/ls", "ls", "-al", NULL); // no return if successful
10      }
11      else if (child_pid < 0) {
12          perror("fork"); // write to stderr
13          exit(1);
14      }
15      if ((child_pid = fork()) == 0) { // concurrent process # 2
16          execl("/bin/foo", "foo", NULL); // no return if successful
17      }
18      else if (child_pid < 0) {
19          perror("fork"); // write to stderr
20          exit(1);
21      }
22      wait (NULL); // wait for child # 1
23      wait (NULL); // wait for child # 2
24      exit(0); // only executed by child.
25  }
26
```