2.

a. Batch systems – a batch system supports running just one jub at a time it has a tray full of jubs to do and they are done separately no job can run along side another a job has to finish in order for the successor job to start , those jobs are run at the mentioned way till the tray is empty and the jobs are done.

b. Multiprogramming system – in a multiprogramming system the memory is seperated into partitions those partitions are used to store several jobs. While one job is waiting for I/o other jobs can continue their execution, and then the cpu can be utilized while the program is waiting for I/o instead of just waiting like it used to do in the batch system.
Although it is important to note that the cpu will not switch context to another job unless the current job has finished or is waiting for I/o, hence the difference from timeshare

c. timesharing system – This is a flavor of a multiprogramming system. every job gets a certaub amount of time from the CPU to use
if the job is waiting for i\o or if its blocked it wont get time.
This way multiple jobs get CPU time.
This prevents one job from hogging the CPU on one hand and On the other , idle and blocked jobs (I/O) allow the CPU to continue handling large jobs that need the CPU at that time.

d. A distributed operating system appears to the user as one computing element.
However, a distributed system consists of several processors that process in parallel different parts of a job or different jobs. It is transparent to The user that there are multiple processing units handling his jobs.

3.

a. The TRAP instruction switches from user mode to kernel mode and starts execution at fix address within the kernel ,once the system call handler has completed its task the control may be returned to the user mode by the TRAP instruction

b. An interrupt is an asynchronous signal from hardware indicating the need for attention or a synchronous event in software indicating the need for a change in execution. A hardware i interrupt causes the processor to save its state of execution via a context switch, and begin execution of an interrupt handler. Software interrupts are usually implemented as instructions
in the instruction set, which cause a context switch to an interrupt handler similar to a hardware interrupt. TRAPS are used by a program to invoke system calls and switch to kernel mode and back.

4. A program is an algorithm written into a programming language , the code itself.
A process is the code running in the memory.

5. Calling blocking I/O in such case can bring system to deadlock or starvation situation for example
if the threads are implemented in the kernel and make blocky I/O call, the calling thread will block and the kernel will
do a context switch and schedule another thread for running. However, if hardware interrupts are temporarily blocked, when the I/O device finishes and sends an interrupt, it won't be handled. The Blocking I/O operation will forever continue to block since it won't get the data it required. And since the blocking I/O operation is in a critical section, no other thread will be able to execute the critical section because it already locked.
Which might result in a deadlock if all threads fell asleep on the mutex of that critical section

6. yes it can , if we use a priority scheduler for instance it would be stupid to do fifo for a semaphore, the first thread that fell asleep on it might have the lowest priority for example and that will hinder the efficiency and usefulness of the priority scheduling , the threads that fell asleep in such a case will adhere to the “rules of the semaphore” instead of the scheduler , a thing we do not want.

7. in Peterson's solution, the critical section is final and the process that goes in will go out and let another process inside consequentially.
   Also the processes are generous, so when a process goes in, it checks if there is another process that want to go in. each process will not wait an infinite time to enter the critical section, because the process that now in critical section execures it's instructions while the other process does a busy wait on a tight loop, and in the moment that the process in critical section calls leaveCriticalSection , the waiting process will immediately enter the critical section, (without them meeting vis a vis) because one of the while condition (interested[otherProcess]) will become false, and the waiting process will enter the critical section.
   The critical section occupier will not cause starvation due to the assumption that it will do it's instructions in finite time.

   jacob kilimnik , 300446408.