**4.**

a) No, it should not give the entire processor to each application until it no longer needs it. The reason is that when there is a CPU constrained process and many IO constrained processes, it is possible after those IO constrained processes are done, the CPU is still be taken. That leads to lower CPU and device utilization. When there are multiple tasks ready to go at the same time, to maintain high CPU utilization, I think we can use round-robin algorithm. That is, we can define a tiny time quantum, usually between 10-100ms. When the execution time of task A reaches the time quantum, even task A is still not finished, it will be preempted by the next task B which is in the queue, and task A will be added to the queue again.

b) The system allocates physical memory according to an algorithm called not recently used. The system will label the memory locations that in used to recently used and label the other to none recently used. When the system needs to assign more memory to application, it will remove that one is not recently used out of the memory and assign it to the new task. If there is not enough memory available to keep all running processes in memory at the same time, then some processes who are not currently using the CPU may have their memory swapped out to a fast local disk called the backing store.

c) The operating system allocates the disk space base on the scheme called linked list allocation. In this scheme each file is a linked list of disk blocks which need to be contiguous. For each block in the disk, it contains the start pointer and end pointer for each file. The first user should ask to acquire all the free space. The likely outcome would be that there might not be enough space if there are more than one user. However, in case there is only one user, we should let the first user to acquire all the free space.

**9.**

I would design a system that updates complex data structures on disk in a consistent manner by breaking updates into small units, despite machine crashes. If the machine crashes during an update, it can be restarted and continue with the crashed task. Each task in the process creates an artifact that represents the state of the data structure before the task begins, so the original data can be recreated if it becomes corrupted or unavailable due to a task exiting during processing.

**1.**

When an interrupt occurs, there is a transition from user mode to kernel mode, and the interrupt instruction forces the program to go to the specified address and perform some tasks. The operation system switch to the kernel mode and interruption is completed, when it wants to perform further, it should be involved in user mode this is done through the line number stored in the stack. Finally, when an interrupt occurs, it switches from user mode to kernel mode, and the hardware switches the stack pointer to the kernel stack.

**7.**

a)

The IRET instruction is used at the end of an interrupt service procedure to return execution to the interrupted program. Interrupt handler uses IRET instruction to restore the kernel interrupt stack to return the control back to the user process after the interrupt handler complete its routine.

b)

First the CPU performs a permission check, then restores the code segment register and the register containing the address of the next instruction to their respective values prior to the interrupt event. The restore contains a register which containing the current state of the processor. The operating system then restores the stack and stack pointer registers to their pre-interrupt values. Finally, the interrupted program is resumed by the CPU.