Name: Harsh Mohan Sason

4.)

a.) The operating system should only allocate time on the processor only when there is a need of it. The operating system should be able to provide multiple cores for the processes so that they can be run in parallel.
It should not give an entire processor to each application until it is not needed. Instead, it should give an illusion to each application that it has the entire processor to itself even though at a physical level there may be only a single processor shared among all the applications running on the computer.
If there were multiple tasks ready to go at the same time, it should schedule the task with the least amount work first. Doing the least amount of work is better because it reduces the time taken for each process so that the bigger tasks have much more time allocated to them .It would allow processes to be finished faster and better. Nevertheless, It should also be able to run the tasks simultaneously because otherwise the processes will just stack up which may cause the system to even crash.

b.) The Operating system should provide an illusion of infinite memory even though there is only a limited amount of it. Hardware only has a finite amount memory but the operating system provides all the applications an illusion of a nearly incite amount of virtual memory
If the entire application does not fit in the memory at the same time, then the OS should swap. Swapping can be done to swap the data or the application which is consuming space but is not currently in use with the application which needs the memory to be allocated to it.

c.) The operating system should allocate disk space the same way it allocates memory; by creating an illusion of a nearly infinite amount of disk space. However, in reality, it only provides a small amount of it and increases that amount over time as the application grows.
When the first user asks to acquire all the free space, there is plenty of

space for the request to be given but the OS in reality doesn't really give it all to the application. The first user should not be given all the free space but should be given an illusion of it. If we give all the free space to the first user, it would be difficult to allocate memory to the other applications which need it.

9.) I would design a system to update complex data structures on disk in a consistent fashion despite machine crashes by splitting up the updates into much smaller units. Therefore, even if the system crashes during the update process, it can restart and pick up with the task that crashed it the last time. This will help utilize the resources much more efficiently and will cause recreating of tasks in the middle of the processing.

———————————————————————————————

1.) In x86, when a user process is interrupted or causes a processor exception, the x86 hardware switches the stack pointer to a kernel stack, before saving the current process state because of two reasons:

a.) Reliability: The processes at user level stack pointer may or may not be at a valid emory address, but in-spite of this, the kernel handler must always continue to work properly and keep the security and integrity of data of the current user.

b.) Security: On a multiprocessor, other threads running in the same process can modify the user memory during the system call. If the kernel handler stores the local variables on the user level stack, the program of the user might just modify the kernel's current return address, causing the kernel to jump to some arbitrary code.

7.)

a.) The operating system will use the ire command whenever it wants to give the control to a user defined program. It is usually used in a case of immediate attention and control. It provides the details to the processor to make the same program a priority. It could be used in four cases:

I.) New process: To start a new process, the kernel copies the program into the memory, sets the program counter to the first intersection of the process, sets the stack pointer to the base of the user stack and switches to user mode.

II.) Resume after an interrupt, processor exception or system call. When the kernel finishes handling the request, it resumes execution of the interrupted processes by restoring its program counter, restoring the registers, and changing the mode back to the user level.

III.) Switch to a different process. In some cases, such as on a timer interrupt, the kernel switches to a different process than the one that had been running before the interrupt. Since the kernel will eventually resume the old process, the kernel needs to save the process state, counter and registers. Thus it can then resume a different process by loading that state, counter and registers.

IV.) User-level upcall: Many operating systems provide user programs with the ability to receive asynchronous notifications of events. The mechanism is similar to kernel interrupt handling, except at a user level.

b.) When an Application executes this instruction, the following happens:

1.) CPU checks for any privileges and then restores the code registers **(EIP registers)** to their values before the interrupt.
2.) It restores the register that contains the current state of the processor **(EFLAGS register).**
3.) Similarly Restores the stack pointer and stack register **(ESP**

and EBP) to their previous value before the interrupt

4.) Thus, after correct restoration, it switches the mode of operation from kernel mode to user mode.