Ruyi Zhang
CIS 415 A1 Due Oct.20
Ch1: 4, 5, 7; Ch2: 1, 7, 10, 12, 15; Supplemental Questions

CH1, 4:
A) This question falls into the performance aspect of the operating system. The processor should not give the entire processor to each application until it no longer needs it for although it would not change the throughput of the system, it would delay the processes other than the one is running. That way it lowers the response time of certain processes. If there were multiple tasks ready to go at the same time, it should at least schedule the same amount of time to each task that way the task with least amount of work would have the fastest response time. It's constant that way.
B) The operating system should only allocate physical memory to applications as much as they need, but give them the impression of able to acquire all the memory resources of the entire system. If the set of applications does not fit in memory at the same time, the operating system should move those unused memory blocks to a larger, maybe slower, media to be temporarily stored, and move them back once needed again.
C) The operating system should only allocate its disk space to users as much as they need. The first user to ask should not acquire all of the free space as it would result in no disk space for other users, and the multi user system, such as a server system, would not be very efficient when one user takes all its free space.

CH1, 5:
A) The OS needs to prevent one process to access the data from another process without the right permission.
B) The OS needs to separate one user from another user and prevent one user from accessing the other users' files without the right permission.
C) The OS needs to monitor network connection and prevent income and outgoing connections from changing data or executing programs without the right permission.

CH1, 7:
I can give a huge fake space to each application, and then map them to certain area of the physical memory. And when certain block is not frequently used, I can move then to a larger but slower media to store until the next time it is needed. That way applications don't need to know what's under the hood so I can change strategies when there is a better solution or when it needs more space than the current strategy can handle.

CH2, 1
It switches the SP to a kernel stack because it saves the program states onto the kernel stack, and that way the program states can't be easily changed, thus can't be easy hacked or corrupted by accidents.

CH2, 7:
A) Everywhere when kernel finishes handling certain event and wants to give back control to the user process, such as when finishing an I/O Interrupt or finishing a system call.
B) When return instruction executes, kernel first restores the saved registers from kernel stack and then pop back SP and PC with the return instruction to get back to the user mode.

CH2, 10: The kernel.

CH2, 12:
When doing a procedure call, the program will have to push the arguments and caller saved registers, return address and local variables to the stack, and then change PC to the new procedure address. Callee procedure will only have to save the callee saved registers, and after it's done, it'll save the result into a specified register and restore the callee saved registers, and return the SP and PC + 1 to the caller procedure.
When doing a system call, program will also have to push the arguments to the stack, but in contrast, the program will have to trap the OS. Then OS will save process states by pushing the process control block to the stack. After that, OS will have to check memory access permission and copy the arguments from user stack and then check the execute permission of the process. If all checks are passed, it will call the right kernel function to execute the kernel program. After it's done it'll have to copy back the result to user memory space, and then restore the process control block, SP and PC + 1, and finally restore the execution of the process.

CH2, 15:
When a interrupt happens, OS will save process states by pushing the process control block to the stack, handle the interrupt, and restore the process control block, SP and PC + 1, and restore the execution of the process.

Supplemental Questions:

Q1: Separation of policy and mechanism is important because it gives flexibility if policy need to be changed later, and the mechanism can also be modularized to work with other policies.

Q2:When switching to another process, OS will save process states by pushing the process control block to the kernel stack, pop control block of another process, SP and PC + 1, and restore the execution of the process.

Q3:

A) Because fork(); exec(); does not close open file descriptors, inter-process communication between two different binaries may be easily achieved by this strategy, so I think that would be the most appropriate circumstance.

B) Since vfork() achieves sharing address space between parent and child, I think the most appropriate circumstance is when two processes need to work on the same thing very badly... That's... It... I... Think...

Q4:

A) With every process is able to perform I/O, OS can achieve multi-programming passively according to I/O interrupts, and at the same time programs can give control back to the OS by a specific system call. There is no reliable way to prevent a process monopolizing the CPU.

B) With time interrupts, OS can evenly distribute CPU resource to processes and itself, so it can check I/O devices every once a while to achieve the communication between processes and I/O devices.

C)I would choose time interrupts over I/O interrupts for a better user experience. At least using time interrupt I can make sure that each process is reliably responsive, and I don't have to spend a whole lot of time and resource to make sure every program running on my plate-form will have a good program-hygiene. However if all the processes are doing are I/Os, time interrupts would perform worse than I/O interrupts only.