CmpE 322

January 29, 2022

Contents

1 Discussion III

1) (a) How is the state transition diagram for processes implemented? Which data structures are required to implement the state transition diagram?

Answer: As processes enter the system, they are put into a job queue, which consists of all processes in the system. The processes that are residing in main memory and are ready and waiting to execute are kept on a list called the **ready queue**. This queue is generally stored as a linked list.

The system also includes other queues. For example, the list of processes waiting for a particular I/O device is called a **device queue**.

A process waits in the ready queue until it is selected for execution. Then that process can issue an I/O request and be placed in an I/O queue, can create a new child process and wait for the child's termination in waiting state, or process can be removed forcibly from the CPU, as a result of an interrupt, and be put back in the ready queue. It's also possible that time allocated for the process expires and process is put back into the ready queue. A process continues this cycle until it terminates, at which time it is removed from all queuse and has its PCB and resources deallocated.

(b) What happens when a process requesting an I/O operation is scheduled to the CPU before its I/O request is completed?

Answer: Scheduler dispatches a process to the CPU by changing its state from **ready** to **running**. If the process requests an I/O operation, its state switches from **running** to **waiting** immediately and I/O operation is performed. Upon the I/O operation is completed, state of the process switches from **waiting** to **ready** state and it waits in the queue to be placed to **running** state again. Time allocated to a process can expire during the I/O operation, that's why we can't put the process to the **ready** state immediately. I/O operation consumes the time allocated for the specific process.

An OS will never try to schedule a process to the CPU if process is doing an I/O operation in waiting state.

(c) What is the functionality of PCB?

Answer: Information associated with each process is stored in the PCB (process control block). It includes the information related to a process such as the state of the process, process number, contents of the registers, memory allocated to the process, I/O devices allocated to process, list of open files, etc.

When a context switch occurs, (CPU switches from one process to another for execution), it has to know the state of the other process. For example, it has to know exactly from which instruction it should continue executing, contents of the registers during the process, which files were open for this process and other information.

(d) Consider a system in which 2 regular user processes want disk I/O (single disk), 1 root process wants network I/O. Out of these three processes, which one goes first?

Answer: Root process will perform network I/O, while one of the regular user processes perform disk I/O concurrently. Precedence between the regular user processes depend on the scheduling algorithm.

2) (a) Explain what the long term scheduler does.

Answer: Long-term scheduler selects which processes should be brought into the ready queue. It also controls the degree of multiprogramming, which corresponds to the number of processes in memory. Long term scheduler is responsible for making a careful selection between the processes. It should select a good process mix of I/O-bound and CPU-bound processes. If all processes are I/O bound, the ready queue will almost always be empty. If all processes are CPU bound, the I/O waiting queue will almost always be empty, devices will go unused and system will be unbalanced. The system with the best performance will thus have a combination of CPU-bound and I/O-bound processes.

(b) How about the medium term scheduler? Explain.

Answer: Medium-term scheduler can be used if degree of multiple programming needs to decrease, that is, if some of the processes have to be removed from the execution due to some reasons such as not having the enough memory.

Medium-term scheduler is responsible for **swapping**, it removes the process from memory, stores it on disk and brings it back into ready queue from disk to continue execution.

3) (a) What if the parent process does not call `wait()`? Which process completes first? Child or the parent?

Answer: `wait()` system call moves the parent process to the waiting state until the termination of the child. If parent process doesn't call `wait()` function, it would continue the execution with the remaining instructions.

When the parent process calls the `wait()` function, the OS will return the information in the Process Control Block of the child to the parent process, and only after that PCB of the child process will be released. If the parent doesn't call `wait()`, PCB of the child will remain there. When the child process

exits, OS still keeps its PCB for its parent. A process that has terminated, but whose parent has not yet called `wait()`, is known as a **zombie** process. If the parent terminates before calling the `wait()`, a zombie process becomes orphan. Linux and UNIX address this scenario by assigning the init process as the new parent to orphan processes. The init process periodically invokes `wait()`, thereby allowing the exit status of any orphaned process to be collected and releasing the orphan's process identifier and process-table entry.

(b) What if the child process does not call `exec()`?

Answer: There is nothing to prevent the child from not invoking `exec()` and instead continuing to execute as a copy of the parent process. In this case, the parent and child are concurrent processes running the same code instructions. Because the child is a copy of the parent, each process has its own copy of any data.

(c) Assume you start Eclipse GUI from the shell (by typing "eclipse &") and then exit the shell? What happens? Why? Then, how come the system services are started by the run scripts even though there is no associated shell?

Answer: When I used the command "eclipse &", bash will provide me with the PID of the child process created. Then it will open Eclipse. If I exit the shell, I'll also exit the Eclipse. Because the shell I'm using is actually the parent process of the subshell representing the Eclipse. Since the parent has terminated, child process (Eclipse) also gets terminated, which is called cascaded termination. To see other processes with their parent-child relationship, you can use `pstree` command.
When you boot your computer, kernel starts system services, sshdeamon, ftpdeamon, etc. All these services should be up and running all the time regardless of the existence of associated shell. If you start these services using a script, when the script terminates the service terminates. What we do instead is, all of these services are detached from bootscript and **init()** takes their responsibility. This is called **deamonizing**. We are breaking the children-parent relationship on purpose. We can finish the process with using its PID.

4) Consider a Google Chrome instance (with two installed plug-ins) running three tabs. How many processes are running? Explain what these processes do? Why are there so many separate processes, but not a single Chrome process?

Answer: There are in total six processes running. Browser process is created when Chrome is started. It is responsible for managing the user interface as well as disk and network I/O. Two plug-in processes are created. Plug-in processes contain the code for the plug-in as well as additional code that enables the plug-in to communicate with associated renderer processes and the browser process. There are three renderer processes since there are three tabs. They are responsible for rendering web pages. They contain the logic for handling HTML, Javascript, images, so on and so forth. There has to be so many separate processes, otherwise if there occurs a problem with any process, all other processes will crash as well. The advantage of the multiprocess approach is that websites run in isolation from one another.

5) (a) What are the advantages of fast and slow context switch?

Answer: Fast context switch = Switching from one process to another very frequently. It makes OS seem more responsive. Slow context switch = Longer time slices for each process.

(b) Do you recommend fast or slow context switch for a system for which majority of the processes are CPU bound?

Answer: When the majority of the processes are CPU bound, it means that most of the time will be consumed doing computations and keeping CPU busy.

(c) Are there any limits for deciding how often a context switch should occur at average (i.e., the length of the typical slice time)?

Answer: Typical slice time should be more than the time consumed while CPU stays idle. (typical slice time should be more than the overhead of the context switch)

Otherwise, we switch from one process to another without performing any instruction in the process.

6) Why shared memory is not sufficient for inter process communication?

Answer: Although the shared memory is faster compared to the message passing for inter process communication. It can't be used in a distributed environment, where the communicating process may reside on different computers, therefore different memories, connected by a network. In such cases, message passing is more useful.