* **Describe process, process state, and process control block.**

According to Silberschatz, Galvin, and Gagne (2014), a process is a program that is being executed. While a program itself is static, “process” is the word used to describe a program whose instructions, along with possible interaction from the user, have been fetched from memory with the intention of execution. Processes can be categorized into different stages at any particular moment in time. This classification of the stage is known as the process state. A new process created has the state of that process as “new.” Example, a user clicks on the icon for the application Microsoft Word, the state of the first process initiated is “new”. When the instructions for the program are being executed, the process state is “running.” In this example, this might occur when the program is opened and the application asks the user what they would like to do, create a new document, open an existing document, etc. While the application waits for user input, the state of the process is “waiting.” The process state is “ready” when the process (creating a new Word document) is waiting to be assigned to a processor. After execution of the program, when the user has created their Word document, saved it, and closed the Word application, the state of the process is “terminated.” In this state, execution has finished and all the resources that were being used during execution are freed up (Silberschatz, Galvin, & Gagne, G., 2014).

A process control block (PCB), or task control block, is a data structure that stores the temporary data relevant to the state of a process. Each process has its own process control block, and all PCBs are maintained in a process table by the operating system. In multiprocessing, the process control block is extremely important, since the execution of processes often switch back and forth so that the CPU is never idle, leading to increased performance. If we go back to the Word application example, when Word initially asks the user what they would like to do, the process needs to wait for user input. The process would, at this point, get added to the I/O queue, freeing up the CPU for other processes that are running simultaneously. The CPU needs a way to “remember” where the process was in its state of execution before it got added to the I/O queue, so that it can resume where it left off once user input is received, the OS relies on the PCB for this function. The PCB stores information including the process ID, process state, program counter, register information, scheduling information, memory related information, accounting information, and I/O information (resource utilization and files opened), all to restore the process back to its correct state when it resumes execution on the CPU (Shukla, 2017).

* **Compare single and multi-threaded motivations and models.**

According to Silberschatz, Galvin, and Gagne (2014), a thread is “a flow of control within a process.” Multi-threaded models are more efficient and yield higher performance than single-threaded processes. In a single-threaded model, a single process requires all of the resources necessary for execution (source code, data, files, memory, registers, a program counter, and a “stack which contains the execution history” (Tutorials Point, 2018). The advantage of a multi-threaded approach is that parallelism is employed using multiple threads that run simultaneously and share resources. Whereas a single-thread is heavy on the system (uses a lot of resources), a multi-threaded approach overall uses fewer resources, since the threads share code, data, and files. In addition, a single-threaded model requires interactions with the OS in order to employ switches, whereas OS interactions are not required in a multi-threaded model. Lastly, a single-threaded model can cause stalls when a process gets blocked, however, in a multi-threaded model, if a thread gets blocked, other threads for the same task can continue to run (Tutorials Point, 2018).

A multi-threaded approach may seem more advantageous than a single-threaded approach, there are challenges to multi-threaded models. There are two different levels of threads. User-level threads are controlled by the user and are not seen by the kernel. Additionally, they are faster to create. Kernel-level threads are created and managed by the operating system and are slower to create (Tutorials Point, 2018). It is important to know about the user and kernel levels to grasp the interactions between the levels during multi-threading.

There are three different models for multi-threading: Many to Many, Many to One, One to One. With the Many to Many model, multiple user-level threads multiplex to many kernel-level threads. If a thread gets blocked, another can be scheduled so that the process does not come to a halt. In this model, threads run in parallel, providing the best accuracy on concurrency. The next model is Many to One. With this model, many user threads multiplex with a single kernel thread. Therefore, if a thread gets blocked, the whole system must wait until it becomes unblocked. Finally, the multi-threaded model One to One is the model in which there is one user thread for every kernel thread. Multiple threads can run simultaneously on a multiprocessor, and if a thread gets blocked, the other threads can still run. However, in order to create a user thread, there must be an available kernel thread (Tutorials Point, 2018).

* **Describe the critical-section problem and explain a software solution that resolves this problem.**

The critical-section problem refers to the fact that processes running in parallel cannot be allowed to access shared resources simultaneously. Most processes contain a segment of code that instructs the process to change a variable, update a table, or change a file (this set of instructions is considered the critical-section of code). For example, if two processes are running, and both need to access the same file, the software needs to be written in a way that these two processes are not executing their critical-sections at the same time. Otherwise, one process may be editing a file that the other process is trying to simultaneously access, causing unpredictable behavior in a program. The critical-section problem can be solved using software that exhibits a mutual exclusion, not allowing multiple processes to execute their critical-sections simultaneously, progress (a thread that wants to execute its critical section can do so when no other thread is executing its own), and bounded waiting. There is a limit to how many times threads can enter their critical-sections before a request to enter by another thread is granted (Silberschatz, Galvin, & Gagne, 2014). The following is an example of a software solution to the critical-section problem provided by our textbook. This example portrays what is known as Peterson’s Solution:

do {

flag[i] = true;

turn = j;

while (flag[j] && turn == j);

critical section

flag[i] = false;

remainder section

} while (true);

This solution alternates execution between processes’ critical sections and remainder sections.

The variable turn represents which process has the right to enter its critical section. The readiness of a process to enter its critical-section is indicated by the flag array (Silberschatz, Galvin, & Gagne, 2014).

**References**

Silberschatz, A., Galvin, P. B., & Gagne, G. (2014). [Operating system concepts essentials](https://ashford.instructure.com/courses/66667/modules/items/3374055)(2nd ed.). Retrieved from https://redshelf.com/

Shukla, A. (2017). *Process Control Block in Operating System*. Retrieved September 20, 2018, from https://www.includehelp.com/operating-systems/process-control-block-in-operating-system.aspx

Tutorials Point. (2018). *Operating System - Multi-Threading*. Retrieved September 20, 2018, from https://www.tutorialspoint.com/operating\_ system/os\_multi\_threading.htm