Theory assignment 2

By:

Jordan Hubscher

ID: 27019696,

Diego Kulibaba

ID: 27010540

1.) Processes are completely independent executing programs and (usually) do not share resources/values with each other. Threads, however are instances of processes that share its registry values, its memory, and code base. Threads can perform multiple segments of code independently for the same program, without using extra memory.

2.) When a thread is created it uses shared virtual address space. Each thread also has their own resources such as scheduling priority, exception handlers, local storage, an identifier and thread context which consists of the kernel stack, user stack and machine registers. A process on the other hand has at least one thread of execution, executable code, and handlers to system objects, environmental variables, an address space and a priority class.

3.) User-level threads are subsets of the instructions. They exist within a process and are scheduled. The operating system is not aware of user-level threads. Kernel-level threads execute the trusted and privileged instructions of the operating system therefore the operating system is aware of kernel level threads. Kernel level threads can be accessed through either a system call or message passing.

4.) It is possible, but it would be terrible because then you would essentially have a batch system, you wouldn’t be able to take advantage of multiprogramming since there be no room to even PLACE the processes you desire to execute within memory

5.)

(a)

semaphore s1 = 0, s2 = 0; s3 = 0; s4 = 0;

process P1 {

<phase I>

P(s1)

P(s4)

V(s2)

<phase II>

}

process P2 {

<phase I>

V(s1)

P(s2)

V(s3)

<phase II>

}

process P3 {

<phase I>

V(s4)

P(s3)

<phase II>

}

(b)

semaphore s1 = 0, s2 = 0; s3 = 0; s4 = 0;

process P1 {

<phase I>

P(s1)

P(s4)

<phase II>

V(s2)

}

process P2 {

<phase I>

V(s1)

P(s2)

<phase II>

V(s3)

}

process P3 {

<phase I>

V(s4)

P(s3)

<phase II>

}

6.) P and V must be implemented as a critical section because both of them manipulate the same variable therefore it is at risk of losing mutual exclusion with its sibling threads. A basic example is given below.

Semaphore s = 1;

process P1 {

P(s)

<<C.S>>

V(s)

}

process P2 {

P(s)

<<C.S>>

V(s)

}

In this example, if the semaphores are not treated as critical sections then, say, process 1 starts first. When it executes P(s), right before it decrements the value of s from 1 to 0, it gets interrupted. Process 2 begins, calls P(s), successfully executes and enters the critical section. Before V(s) is called, process 2 is interrupted and process 1 takes place back in P(s) where it decrements its value and then enters the critical section. This cases causes both processes to be in the critical section thus failing at maintaining mutual exclusion.

7.) Security would have to be the number one problem with multiprogramming. Since multiprogramming shares resources such as memory between processes. Therefore one program can access the memory of another program and this can lead to potential corruption and malicious access to data and other resources.