M Andrew Baca

Operating Systems

Homework 3

11/30/18

1.1 (5.2 Book)

Windows, Linux, and Solaris use multiple different locking mechanisms based on different typed of development that takes place on the system, such as kernel development and basic application development. In mutex locks, the process acquires the lock before execution of the critical section and releases the lock after execution and are used in pthread implementation. Spinlocks are useful because there is no context switch required for a process to wait for a lock, so when locks are expected to be used for short intervals, spinlocks are useful. Semaphores are useful for locks that are going to be used for longer intervals since busy waiting in spinlocks is inneficient. Adaptive mutex locks protects access to every critical data item with the implementation of a spinlock, and these are used by solaris to protect data accessed by short code segments. Condition variables are similar to semaphores in the sense that they will be used for longer time spans and are in a certain state till a certain condition occurs.

1.2 (5.4 Book)

Spinlocks are more appropriate in multiprocessor systems because one thread can spin on one processor while another thread performs its critical section on another processor, so all processes can make progress, opposed to the starvation that can occur on a single processor system.

1.3 (5.5 Book)

It is possible that mutual exclusion can be broken if the semaphore operations are not executed automatically in the case that the semaphore equals 1 for two operations, since the wait() operation will decrement the semaphore values, and this would violate a mutual exclusion. Timing errors can often be difficult to detect in semaphores especially if operations are not executed automatically.

1.4 (5.29 Book)

The signal() operation in monitors resumes exactly one suspended process, and if no process is suspended, the signal will simply have no effect, where as the signal operation in the semaphore always affects the state of the semaphore regardless of suspended processes.

1.5

The dining philosopher problem is a problem used to represent a classic synchronization problem. Consider five philosophers who just think and eat. They are sitting at a table with a bowl of rice in the middle and five chopsticks, one between each philosopher, where the philosopher can only pick up one chopstick at a time, and only can get the two chopsticks closest to them. If the person is eating next to them, they have to wait for them to get back to thinking so they can pick up the chopstick and eat.

We will be using monitors (IMPORTANT NOTE: code retrieved from book, page 227-228) to solve the dining philosophers problem, where a philosopher can only pick up a chopstick if both are available. We will but the states of the philosophers in a data structure:

enum{THINKING, EATING, HUNGRY}state[5]

and philosopher[i] cannot eat unless both neighbors are not eating, and this can be represented by:

(state[(i+4) % 5] != EATING) && (state[i+1) %5) != EATING)

will result in the following code:

monitor DiningPhilosophers{

enum {THINKING, HUNGRY, EATING} state[5];

condition self[5];

void pickup(int i){

state[i] = HUNGRY;

test(i);

if(state[i] != EATING)

self[i].wait()

}

void putdown(int I) {

state[i] = THINKING;

test((i + 4) % 5);

test((i+ 1) % 5);

}

void test(int I) {

if((state[(i+4) % 5] != EATING) && (state[(i+1)%5] != EATING)){

state[i] = EATING;

self[i].signal();

}

}

initialization\_code() {

for (int I = 0; I < 5; i++)

state[i] = THINKING

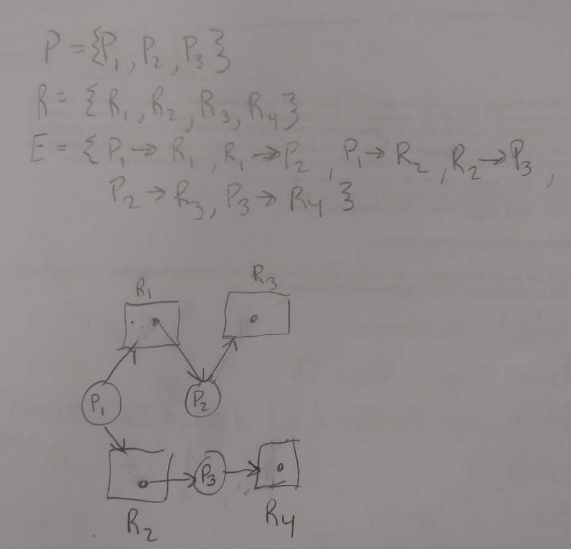
}

}

2.1 (7.10 Book)

You cannot have a deadlock in a single threaded process because a single threaded process has only one thread of communication so the necessary conditions would not be met to hold true for a deadlock to occur. For instance, the hold and wait condition would not be able to occur with just one single threaded process because additional resources cannot be held by other processes.

2.2 (7.17 Book)



2.3

Deadlock prevention uses protocols that cover each of the four conditions to see if one of the conditions will not hold true. Some protocols involve involves each process to request and allocate all resources before beginning execution, as well as implicitly releasing resources while others are needed and ordering all resources and lets processes request resources based on that information.

Deadlock avoidance algorithms prevent deadlocks by limiting how request can be made by requiring additional information about how resources are to be requested by declaring the max number of resources that each type may need. We also need to check whether a state is safe in order to insure system never reaches a deadlock. Resource allocation graphs and algorithms such as bankers and safety can be used to avoid deadlocks.

IF a system does not employ deadlock avoidance or prevention algorithms , then deadlock detection algorithms may be provided to determine whether a deadlock occurred and recover from it. We can use wait-for graphs for single instance resource types.

2.4

one real life example of a deadlock would be a traffic jam, just as mentioned in the book with the trains, such as a really busy 2 way highway with one lane each way.

Another example of a deadlock would be trying to reach somebody on the phone the exact same time they are trying to reach you.

The last example of a deadlock would be a bill being shut off for not receiving payments.

2.5

The assumption behind 3 is that we are currently not in a deadlock and we have enough space allocated for the resources that we need in order to complete process.