Jon Tice

2/8/2018

CSCI 411

Homework 2

1. Draw the 5-state thread model. Label the states and describe what causes a thread to move to/from each state.
2. When a process creates a new process using the *fork()* operation, which of the following states is shared between the parent process and the child process
   1. Stack
   2. Heap
   3. Shared Memory segments
3. Consider the following piece of code:

void main() {

fork();

fork();

exit();

}

How many processes are created upon execution of this process?

You have 4 total processes upon the execution of this process.

1. Child processes
   1. Explain a zombie process, how it comes to be and how it is terminated.

A zombie process is created when the child terminates before the parent performs wait() and the Kernel turns the process into a Zombie. After the parent dies the zombie will be inherited by INIT which will clear it from the process table.

* 1. Explain an orphan process, how it comes to be and how it is terminated.

An orphan process is a child process whose parent process ended before it did, now INIT becomes the parent and will periodically clean up orphans.

1. Which of the following are shared across threads in a multithreaded process?
   1. Register values
   2. Heap memory
   3. Global variables
   4. Stack memory
2. What are the 3 requirements for a solution to the critical-section problem?

1-Mutual Exclusion

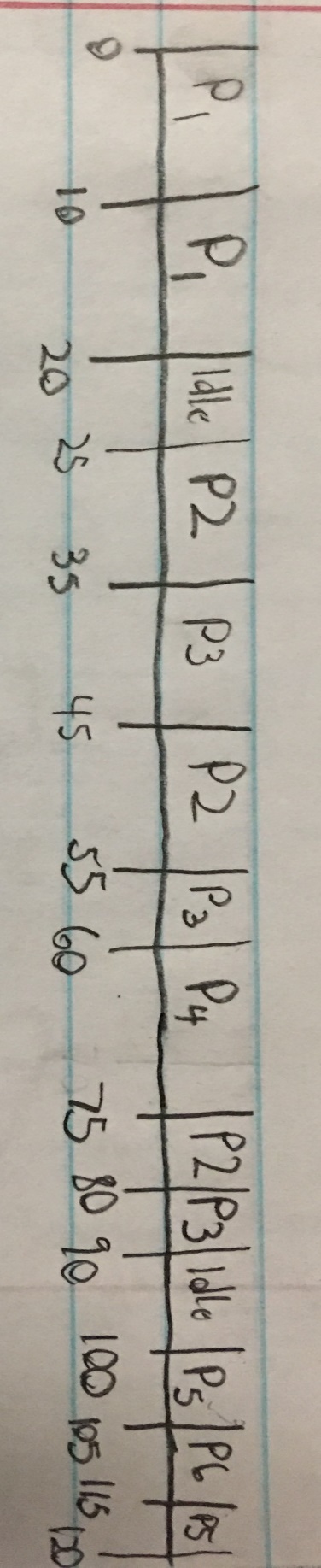
2-Progress

3-Bounded Waiting

1. Describe the Peterson solution to the critical section problem.

Peterson’s Solution is a two-process solution in which the LOAD and STORE instructions are atomic, and cannot be interrupted. The solution requires the two processes share two variables, turn, which indicates whose turn it is to enter the critical section and flag, an array used to indicate if the process is ready to enter the critical section.

1. What is a spinlock? Why is it not appropriate for single-processor systems yet are often used in multiprocessor systems?

A spinlock is a lock that causes the thread trying to acquire it to wait in a loop constantly checking if the lock is available. Spinlocks are not appropriate for single processor systems because the condition to break a process out of it can only be achieved by executing different process, this is why it can be used in multiprocessor systems other processes run on other processors and change the program state to release the first process from the spinlock.

1. The following processes are being scheduled using a preemptive, round- robin scheduling algorithm. Each process is assigned a numerical priority, with a higher number indicating a higher relative priority. In addition to the processes listed below, the system also has an idle task (which consumes no CPU resources and is identified as Pi ). This task has priority 0 and is scheduled whenever the system has no other available processes to run. The length of a time quantum is 10 units. If a process is preempted by a higher-priority process, the preempted process is placed at the end of the queue.
   1. Show the scheduling order of the processes using a Gantt chart.
   2. What is the turnaround time for each process

P1:20-0 = 20

P2:80-25 = 55

P3:90-30 = 60

P4:75-60 = 15

P5:120-100 = 20

P6:115-105 = 10

* 1. What is the waiting time for each process?

P1:0

P2:30

P3:35

P4:0

P5:10

P6:0

|  |  |  |  |
| --- | --- | --- | --- |
| Thread | Priority | Burst | Arrival |
| P1 | 40 | 20 | 0 |
| P2 | 30 | 25 | 25 |
| P3 | 30 | 25 | 30 |
| P4 | 35 | 15 | 60 |
| P5 | 5 | 10 | 100 |
| P6 | 10 | 10 | 105 |

1. Using Semaphores In class we discussed a solution to the bounded-buffer problem using three semaphores (mutex, emptyBuffers, and fullBuffers):

|  |  |
| --- | --- |
| Producer () {  emptyBuffers. wait ();  mutex. wait ();  put 1 coke in machine;  mutex. post ();  fullBuffers. post ();  } | Consumer() {  fullBuffers. wait ();  mutex. wait ();  take 1 coke from machine;  mutex. post ();  emptyBuffers. post ();  } |

Given each of the following variations, say whether it is correct or incorrect. If you say correct, explain any of the advantages and disadvantages of the new code. If you say incorrect, explain what could go wrong (i.e., trace through an example where it does not behave properly).

* 1. Variation 1

|  |  |
| --- | --- |
| Producer () {  mutex. wait ();  emptyBuffers.wait();  put 1 coke in machine;  fullBuffers.post();  mutex. post ();   } | Consumer() {  mutex. wait ();  fullBuffers. wait ();  take 1 coke from machine;  mutex. post ();  emptyBuffers. post ();  } |

Incorrect, the mutex lock is acquired by both the producer and consumer without checking if either can perform their own actions, if the consumer releases the mutex lock and is going to consume again the producer could acquire the mutex lock and be stuck in a loop waiting for the consumer to consume more items but the consumer cannot do that because the producer already has the mutex lock. Also if the empty buffers is not decremented before mutex is unlocked in the consumer the producer is free to acquire a mutex lock before the action of increasing the empty buffers is completed in the consumer and the producer is waiting on the empty buffers to be increased. As compared to the end of the producer code where the full buffers is incremented then the mutex is unlocked, it could cause the two to become out of synch.

* 1. Variation 2

|  |  |
| --- | --- |
| Producer () {  emptyBuffers.wait();  mutex. wait ();  put 1 coke in machine;  fullBuffers.post();  mutex. post ();   } | Consumer() {  fullBuffers. wait ();  mutex. wait ();  take 1 coke from machine;  emptyBuffers. post ();  mutex. post ();  } |

Correct, however a disadvantage to this the increasing of the emptyBuffer and fullBuffer should not be in the critical part that is being protected by the mutex lock.