7.2/10

COSC 450 Operating System Test #2

3/7/2024

Name: Kyle Tramfoglia .

1. (2 pt.) Consider the following set of processes (each process is 100 % CPU-bounded).

Process	CPU-Time	Arrival Time	Priority
P_1	8 10	0	1
P_2	3 5 7	2	3
P_3	7	4	3
P_4	5	6	2

What is the average process waiting times and average turnaround time for the preemptive priority scheduling algorithms and Shortest Remain time first algorithm? There are rules for some cases

- Between processes with same priorities, use round robin with time unit 4.
- If preempted process has highest priority and does not use its time unit, it will keep CPU time.

P_2 has highest priority so continue using round-robin 4 seconds

use round-robin because same priority

a) Preemptive Priority Scheduling

X

P1	P2	P2	P3	P2	P3	P4	P1
0	2	4	8	12	13	14	19 – 27

-.2

Average process wait: ((19-2) + (12-8) + (13-12) + (14-6)/4 = 7.5

Average Turnaround: ((27-0) + (13-2) + (14-4) + (19-6))/4 = 15.25

b) Shortest remain time first

X

P1	P2	P2	P4	P2	P3	P1
0	2	4	6	11	14	21 – 29

Average process wait: ((21-2) + (11-6) + (0) + (14-4))/4 = 8.5

Average Turnaround: ((29-0) + (14-2) + (21-14) + (11-6))/4 = 13.25

2. (0.5 pt.) What are 5 criteria for comparing CPU- Scheduling Algorithms?

-.2

X

1) Average process wait ✓
2) Average turnaround time ✓
3) minimal average process wait
4) minimal average turnaround time
5) Preemption issues/race conditions

i.e. Petersons Priority inversion problem

3. (2 pt.) Let's consider two processes, P_0 and P_1, concurrently accessing a shared resource within the critical section. The solution presented here addresses the race condition using busy waiting. A variable, Permit, is employed and can take on values of 0 or 1. When Permit is set to 0, only process P_0 is granted access to the critical section. Upon completion of its tasks within the critical section, P_0 sets Permit to 1, allowing process P_1 to enter the critical section. Conversely, when Permit is set to 1, only process P_1 is authorized to access the critical section. After completing its operations within the critical section, P_1 sets Permit back to 0, enabling process P_0 to enter the critical section. **It is assumed that a process, once inside the critical section, does not terminate prematurely.**

```
repeat
        while Permit ≠ i do
            ; (no-operation)
          [ Critical Section ]
        Permit = j;
          [ Remainder Section ]
until false
```

-1

Show this solution cannot solve race condition.

If a process could terminate prematurely, this would result in a process waiting forever to enter critical region. Without considering this, this still does not solve race condition as it does not satisfy mutual exclusion. For example, if P_0 enters critical region with Permit = 0, then P_0 finishes its task and sets Permit = 1, allowing P_1 to enter critical region, but P_0 timesout before leaving critical region, then P_1 will enter the critical region and P_0 and P_1 will be in critical region at same time, which violates mutual exclusion.

4. (1 pt.) Peterson's solution effectively addresses the race condition but is marred by the drawback of necessitating busy waiting. This approach not only consumes CPU time needlessly but can also give rise to an unexpected issue known as the priority inversion problem. What exactly is the priority inversion problem when it comes to busy waiting?

1

Priority inversion has to do with a low priority process getting stuck in the critical region because it is not given CPU time, as a higher priority task gets CPU time but stays in Busy wait.

For example: P_L and P_H are processes. $P_H > P_L$. P_L is in critical selection and P_H is in Block State. P_H moves to ready state and P_L gets timed out. P_H is selected by CPU scheduler to run, but P_H is stuck waiting in Busy wait. P_L does not get selected by CPU scheduler because it is low priority, so P_H stays in Busy wait. Aging may be used to help this issue, although it mainly solves starvation.

5. (0.5 pt.) A solution for the race condition should have four necessary conditions. Discuss four necessary conditions.

   1) mutual exclusion - No two processes can enter critical region at same time
   2) No process outside critical region may block another process
   3) No process is to wait forever to enter critical region
   4) May make no assumptions about speeds or # of CPUs

6. (2 pt.) Mr. Computer attempts to address the race condition using semaphores in the context of the Producer-Consumer problem. He devises the following solutions.

   Does his solution solve the race condition? Discuss Mr. Computer's solution, if his method works for avoiding race condition. In the event that his method proves unsuccessful, outline a scenario that leads to a situation violating the race condition..

```
#define N 100
semaphore mutex = 1;
semaphore empty = N;
semaphore full = 0;

void producer ()
{
    int item;
    while (ture)
    {
        item = produce_item();
        down (&empty);
        down (&mutex);
        insert_item(item);
        up(&mutex);
        up(&full);
    }
}
```

```
void consumer()
{
    int item;

    while (true)
    {
        down(&mutex);
        down(&full);
        item = remove_item();
        up(&mutex);
        up(&empty);
        consume_item(item);
    }
}
```

This Does NOT solve the race condition because of

Scenario:

- Full = 0, empty = N, mutex = 1
- Consumer is scheduled
- Consumer downs mutex to 0
- Consumer can not down full because full = 0
- Consumer waits
- Producer is scheduled
- Producer tries to insert an item but mutex is at 0 so it can not enter critical region

Now Both processes are stuck outside critical region

Deadlock!

7. (0.5 pt.) There are two types of parallelism in multicore programming with threads: Data parallelism and Task parallelism. Briefly explain both Data and Task Parallelism.

- Data Parallelism multi core programming uses a bus to interconnect data storages (cache). Data in one core's cache can be accessed and processed in parallel with another core as it can access its cache and perform data transfers in parallel.
- Task Parallelism User thread runs concurrently in parallel with kernel thread. Two user threads can use CPU resources at same time by utilizing different cores to run in parallel.

-.12 ??

??

-.1

8. (0.5 pt.) There are two fundamental models of interprocess communication: shared memory and message queue. To use shared memory, programmer has responsible for synchronization and mutual exclusion of the shared memory. What is main advantage to use the shared memory? Shared memory allows for easier communication between related processes and can save memory. If multiple processes require the same resources or are altering/storing the same information during intercommunication, shared memory favors performance and memory space. It is also used in part to solve race conditions by helping maintain the necessary conditions.

??

9. (1 pt.) To improve disadvantage of kernel's level thread and user's level thread, some operating system provides a combined user and kernel level thread facility. There are three types of models: Many-to-One, One-to-One and Many-to-Many. Briefly describe each type of model.

- Many-to-One: Does not utilize parallelism. Many user threads generate one kernal thread such that if one user thread blocks, the kernal thread is blocked for all other user threads
- One-to-One: utilizes parallelism but must be careful not to create too many user threads as it results in poor performance
- Many-to-Many: Best of one-to-one and many-to-many. Most accurate and consistent. utilizes parallelism, but does not have to be careful of creating too many user threads. Also, if a kernal thread is blocked, other user threads will not also block as they have their own parallelized kernal threads.