Fundamentos de los Sistemas Operativos (FSO)

Departamento de Informática de Sistemas y Computadoras (DISCA)
Universitat Politècnica de València

Consolidation

Exercises 2

Problem

Each thread T_i, i = 1, 2, …, 9 is coded as follows:

```
sem_init(&sem,0,1);

while (1) {
  sem_wait(&sem);
  { Critical Section }
  sem_post(&sem);
}
```

The code for T_{10} is identical except that it uses post(mutex) in place of wait(mutex). What is the largest number of threads that can be inside the critical section at any moment?

```
while (1) {
  sem_post(&sem);
  { Critical Section }
  sem_post(&sem);
}
```

1. 1
2. 2
3. 3
4. None of these

Problem

Scenario:
Initially, process P1 arrives.
It performs the wait operation on mutex and sets its value to 0.
It then enters the critical section.

Consider the processes P2, P3, … , P9 arrives when process P1 is executing the critical section.
All the other processes get blocked and are put to sleep in the waiting list.

Process P10 arrives.
It performs the signal operation on mutex.
It selects one process from the waiting list say process P2 and wakes it up.
Process P2 can now enter the critical section (P1 is already present).
Now, process P10 enters the critical section (P1 and P2 are already present).
After executing critical section, during exit, it again performs the signal operation on mutex.
It selects one process from the waiting list say process P3 and wakes it up.
Process P3 can now enter the critical section (P1 and P2 are already present).
Process P10 may keep on executing repeatedly.
It wakes up 2 processes each time during its course of execution.
In this manner, all the processes blocked in the waiting list may get entry inside the critical section.

Thus, Option (D) is correct.

Problem

Let m[0]…m[4] be mutexes (binary semaphores) and P[0]…P[4] be threads. Suppose each thread P[i] executes the following:

```
sem_t m[5];

//P[i] executes

sem_wait(m[i]);
sem_wait(m[(i+1) mod 4]);
{ Critical Section }
sem_post(m[i]);
sem_post(m[(i+1) mod 4]);
```

This could cause:

1. Thrashing (over-scheduling)
2. Deadlock
3. Starvation but not deadlock
4. None of the above

Problem

```
sem_t m[5];

//P[i] executes

sem_wait(m[i]);
sem_wait(m[(i+1) mod 4]);
{ Critical Section }
sem_post(m[i]);
sem_post(m[(i+1) mod 4]);
```

1. Thrashing
2. Deadlock
3. Starvation but not deadlock
4. None of the above

Threads will use:
Thread P0 : wait(m[0]); wait(m[1]);
Thread P1 : wait(m[1]); wait(m[2]);
Thread P2 : wait(m[2]); wait(m[3]);
Thread P3 : wait(m[3]); wait(m[0]);
Thread P4 : wait(m[4]); wait(m[1]);

Sequence:
1. Thread P0 arrives. It executes wait(m[0]) and gets preempted.
2. Thread P1 gets scheduled. It executes wait(m[1]) and gets preempted.
3. Thread P2 gets scheduled. It executes wait(m[2]) and gets preempted.
4. Thread P3 gets scheduled. It executes wait(m[3]) and gets preempted.
5. Thread P4 gets scheduled. It executes wait(m[4]) and gets preempted.

The system is in a deadlock state since no process can proceed its execution.
Thus, Option (2) is correct.

Problem

- In the above question, which of the following pairs of threads may be present inside the critical section at the same time?

1. (P0, P2)
2. (P1, P3)
3. (P2, P4)
4. (P3, P4)
5. All of these

Problem

The following program consists of 3 concurrent threads and 3 binary semaphores. The semaphores are initialized as S0 = 1, S1 = 0 and S2 = 0.

How many times will thread P0 print 'hola'?

1. At least twice
2. Exactly twice
3. Exactly three time
4. Exactly once

```
Th0:
for(;;) {
  sem_wait(&S0);
  printf("hola\n");
  sem_post(&S1);
  sem_post(&S2);
}

Th1:
  sem_wait(&S1);
  ...
  sem_post(&S0);


Th2:
  sem_wait(&S2);
  ...
  sem_post(&S0);
```

Problem

The following program consists of 3 concurrent threads and 3 **binary semaphores**. The semaphores are initialized as

S0 = 1, S1 = 0 and S2 = 0.

What is the **maximum**/**minimum** number thread P0 will print 'hola'?

1. At least twice
2. Exactly twice
3. Exactly three time
4. Exactly once

```
Th0:
while (1) {
  sem_wait(&S0);
  printf("hola\n");
  sem_post(&S1);
  sem_post(&S2);
}

Th1:
  sem_wait(&S1);
  ...
  sem_post(&S0);


Th2:
  sem_wait(&S2);
  ...
  sem_post(&S0);
```

Problem

The following program consists of 3 concurrent threads and 3 **binary semaphores**. The semaphores are initialized as

S0 = 1, S1 = 0 and S2 = 0.

What is the **maximum**/**minimum** number thread P0 will print 'hola'?

1. At least twice
2. Exactly twice
3. Exactly three time
4. Exactly once

```
Max: Th0;Th1;Th0;Th2;Th0        => 3
Min: Th0;Th1;Th2;Th0;--         => 2
```

```
Th0:
while (1) {
  sem_wait(&S0);
  printf("hola\n");
  sem_post(&S1);
  sem_post(&S2);
}

Th1:
  sem_wait(&S1);
  ...
  sem_post(&S0);


Th2:
  sem_wait(&S2);
  ...
  sem_post(&S0);
```

Problem

Suppose we want to synchronize two concurrent threads P and Q using binary semaphores S1 and S2. The code is:

This ensures:

1. Mutual Exclusion
2. Deadlock
3. Starvation but not deadlock
4. None of these

```
P:
while(1)
{
   P(S1);
   P(S2);
   Critical Section
   V(S1);
   V(S2);
}
```

```
Q:
while(1)
{
   P(S1);
   P(S2);
   Critical Section
   V(S1);
   V(S2);
}
```

Problem

Considers the following processes: arrival time, prio, (cpu, io, …

P1: 0, 2, (1, 5, 3)

P2: 2, 3, (3, 3, 1)

P3: 3, 1, (2, 2, 1)

If the CPU scheduling policy is Priority Scheduling, calculate the average waiting time and average turn around time.

Turn Around time = Exit time – Arrival time
Waiting time = Turn Around time – Burst time

Average Turn Around time = (10 + 13 + 6) / 3 = 29 / 3 = 9.67 units
Average waiting time = (6 + 9 + 3) / 3 = 18 / 3 = 6 units

Problem

Four jobs to be executed on a single processor system arrive at time 0 in the order A, B, C, D.

Their burst CPU time requirements are 4, 1, 8, 1 time units respectively.

Computes the completion time of A, B, C, D under round robin scheduling with quantum = 1:

Question

An optimal scheduling algorithm in terms of minimizing the average waiting time of a given set of processes is:

1. FCFS scheduling algorithm
2. Round robin scheduling algorithm
3. Shortest job - first scheduling algorithm
4. None of the above

Question

In Priority Scheduling a priority number (integer) is associated with each process. The CPU is allocated to the process with the highest priority (smallest integer = highest priority). The problem of Starvation of low priority processes may never execute, is solved by:

1. Terminating the process
2. Aging
3. Mutual Exclusion
4. Semaphore

Aging is used to gradually increase the priority of a task, based on its waiting time in the ready queue.

Question

With the Round Robin CPU scheduling in a time-shared system:

1. Using very large quantum degenerates in to first come first served algorithm
2. Using extremely small quantum improve performance
3. Using extremely small quantum degenerate in to last in first out algorithm
4. Using medium sized quantum leads to shortest job time first algorithm

Question

Which of the following is a criterion to evaluate a scheduling algorithm?

1. CPU Utilization: Keep CPU utilization as high as possible
2. Throughput: number of processes completed per unit time
3. Waiting Time: Amount of time spent ready to run but not running
4. All of the above

Question

In interactive environments such as time-sharing systems, the primary requirement is to provide reasonably good response time and in general, to share system resources equitably. In such situations, the scheduling algorithm that is most popularly used is:

1. Shortest Remaining Time First (SRTF) Scheduling
2. Priority Based Preemptive Scheduling
3. Round Robin Scheduling
4. None of the above

Problem

Given the following C code, that produces the executable file named "prog" and considering that **COND** can be defined as "= =" or ">", answer the following items:

a) Suppose that "prog" is executed and its parent PID is 4000, and its own PID is 4001. Along its execution the system assigns to the processes that are created the sequence of PIDs: 4002, 4003, 4004, etc. Fill the following tables with the values that are shown in the screen with the strings PID_IF, PPID_IF, PID_FOR and PPID_FOR, considering that COND is defined as:

a1) #define COND ==

a2) #define COND >

b) Considering "#define COND ==" explain if there can be zombie or orphan processes, and if so how many of each type can appear.

```
#include <all required headers>
int main() {
  pid_t pid;
  int i;

  for (i=0; i<3; i++){
    pid = fork();
    if (pid COND 0){
        printf("PID_IF = %d, PPID_IF = %d \n",
                    getpid(), getppid());
        sleep(5);
        break;
    }
    printf("PID_FOR = %d, PPID_FOR = %d \n",
                    getpid(), getppid());
   }
   sleep(5);
   return(0);
}
```

Problem

Given the following C code, that produces the executable file named “prog” and considering that **COND** can be defined as “= =” or “>”, answer the following items:

```
#include <all required headers>
int main() {
 pid_t pid;
 int i;

 for (i=0; i<3; i++){
  pid = fork();
  if (pid COND 0){
    printf("PID_IF = %d, PPID_IF = %d \n",
           getpid(), getppid());
    sleep(5);
    break;
  }
  printf("PID_FOR = %d, PPID_FOR = %d \n",
           getpid(), getppid());
 }
 sleep(5);
 return(0);
}
```

a) Suppose that “prog” is executed and its parent PID is 4000, and its own PID is 4001. Along its execution the system assigns to the processes that are created the sequence of PIDs: 4002, 4003, 4004, etc. Fill the following tables with the values that are shown in the screen with the strings PID_IF, PPID_IF, PID_FOR and PPID_FOR, considering that COND is defined as:

a1) #define COND ==

PPID_IF	PPID_IF	PPID_FOR	PPID_FOR

a2) #define COND >

PPID_IF	PPID_IF	PPID_FOR	PPID_FOR

Problem

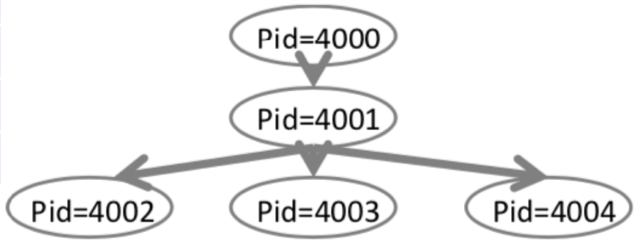
Given the following C code, that produces the executable file named "prog" and considering that **COND** can be defined as "= =" or ">", answer the following items:

```
#include <all required headers>
int main() {
  pid_t pid;
  int i;

  for (i=0; i<3; i++){
    pid = fork();
    if (pid COND 0){
        printf("PID_IF = %d, PPID_IF = %d \n",
                    getpid(), getppid());
        sleep(5);
        break;
    }
    printf("PID_FOR = %d, PPID_FOR = %d \n",
                    getpid(), getppid());
   }
   sleep(5);
   return(0);
}
```

a) Suppose that "prog" is executed and its parent PID is 4000, and its own PID is 4001. Along its execution the system assigns to the processes that are created the sequence of PIDs: 4002, 4003, 4004, etc. Fill the following tables with the values that are shown in the screen with the strings PID_IF, PPID_IF, PID_FOR and PPID_FOR, considering that COND is defined as:

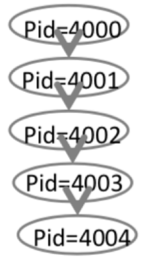
a1) #define COND ==

PPID_IF	PPID_IF	PPID_FOR	PPID_FOR
4002	4001	4001	4000
4003	4001	4001	4000
4004	4001	4001	4000



a2) #define COND >

PPID_IF	PPID_IF	PPID_FOR	PPID_FOR
4001	4000	4002	4001
4002	4001	4003	4002
4003	4002	4004	4003



Problem

Given the following C code, that produces the executable file named “prog” and considering that **COND** can be defined as “= =” or “>”, answer the following items:

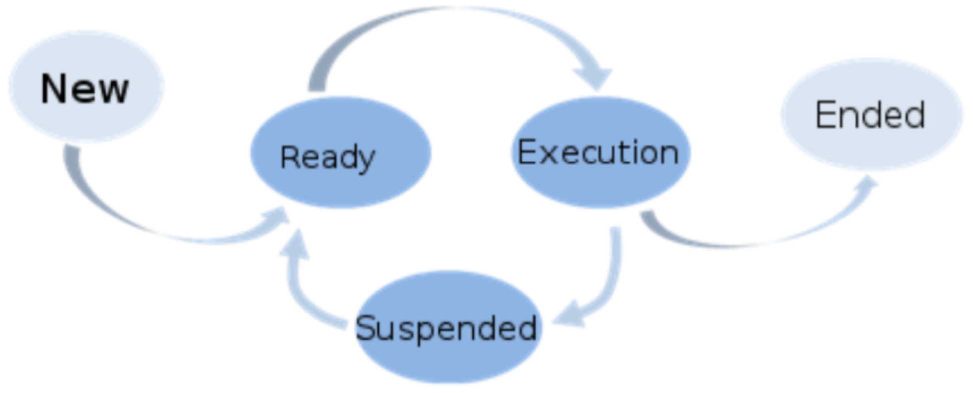
```
#include <all required headers>
int main() {
  pid_t pid;
  int i;

  for (i=0; i<3; i++){
    pid = fork();
    if (pid COND 0){
        printf("PID_IF = %d, PPID_IF = %d \n",
                  getpid(), getppid());
        sleep(5);
        break;
    }
    printf("PID_FOR = %d, PPID_FOR = %d \n",
                  getpid(), getppid());
   }
   sleep(5);
   return(0);
}
```

b) Considering “#define COND ==” explain if there can be zombie or orphan processes, and if so how many of each type can appear.

Zombie and orphan processes with “#define COND ==”
With this condition a process fan is generated, so the sentences inside “if (pid==0)” are only executed by the children that go into suspension after 5 seconds and then they go out of the loop to enter again into suspension for another 5 seconds, 10 seconds in all.
The initial process with PID 4001 executes all the for loop iterations, and before ending goes into suspension for 5 seconds. Furthermore the parent doesn’t call to wait so it will end before its children that become 3 orphan processes.
There will be no zombie processes because the children execution time is greater that the parent.

Question

A given operating system has the following diagram of possible states and transitions for processes:



Explain if this operating system could be working with a preemptive scheduler.