OS homework3

November 2019

CONTENT

1 Chapter 6

1.1 Question 6.2

What is the meaning of the term busy waiting? What other kinds of waiting are there in an operating system? Can busy waiting be avoided altogether? Explain your answer.

Answer: Busy waiting means that a process is waiting for a condition and continuously check if the current condition is satisfied in a tight loop. And in this period, this process will not give up the processor resource which is occupied by itself. Only the condition is satisfied, this process will go on.

The other kind of waiting is that a process can relinquish the processor, block on a condition and wait to be awakened at some appropriate time in the future that the condition is satisfied.

Busy waiting can be avoided. Let me elaborate my reason. As we can transform the busy waiting to the waiting mentioned above, we can set block and make the process relinquish the processor until the condition is satisfied. But there are also some reasons for preserving busy waiting existed. Firstly, busy waiting can avoid the overhead of putting a process to sleep and having to wake it up when the appropriate program state is reached, as it never puts a process to sleep. Secondly, when we need do delayed operation on some device which do not have corresponding interrupt, busy waiting is also a choice.

1.2 Question 6.3

Explain why spinlocks are not appropriate for single-processor systems yet are often used in multiprocessor systems.

Answer: Spinlocks are not appropriate for single-processor systems. The reason is that the condition that would break a process out of the spinlock can be obtained only by executing a different process. In other words, the if a process set a spinlock, it can not obtain it by itself, only other process has ability to obtain it and the most important thing is that the original process will not give up the processor resource. In the other hand, a single-processor system, if the process is not relinquishing the processor, other processes do not get the opportunity to set the program condition required for the first process to make progress.

But in a multiprocessor system, this delemma is not existing. Althogh the original process will not give up the processor resource, other processes can execute on other processors, so they can modify the program state in order to release the first process from the spinlock, which is necessary to promise the spinlocks work properly.

1.3 Question 6.8

Race conditions are possible in many computer systems. Consider an online auction system where the current highest bid for each item must be maintained. A person who wishes to bid on an item calls the bid(amount) function, which compares the amount being bid to the current highest bid. If the amount exceeds the current highest bid, the highest bid is set to the new amount. This is illustrated below:

```
void bid(double amount) {
    if (amount > highestBid)
    highestBid = amount;
}
```

Answer: In this situation, the race condition actually happens in the operation **highestBid = amount**, because we can't promise that when we judged that the amount is bigger than highestBid, the highestBid can be updated at the same time. If there are several threads or processes hold a amount which are all higher than the previous highestBid, then they all want to update the value of highestBid, if the last one which modified the value do not hold the highest amount, we will get the wrong answer.

In order to prevent this bad situation, we can control this functiobn only one thread or process can access the function bid(), and this can promise that if the amount is higher than highestBid, than the highestBid must be updated. In practice we can use some method such as Mutual Exclusion.

1.4 Question 6.11

One approach for using compare and swap() for implementing a spin lock is as follows:

```
void lock spinlock(int *lock) {
    while (compare and swap(lock, 0, 1) != 0)
    ; /* spin */
}
```

A suggested alternative approach is to use the “compare and compare and-swap” idiom, which checks the status of the lock before invoking the compare and swap() operation. (The rationale behind this approach is to invoke compare and swap()only if the lock is currently available.) This strategy is shown below:

```
void lock spinlock(int *lock) {
{
    while (true) {
        if (*lock == 0) {
        /* lock appears to be available */
        if (!compare and swap(lock, 0, 1))
            break;
        }
    }
}
```

Does this “compare and compare-and-swap” idiom work appropriately for implementing spinlocks? If so, explain. If not, illustrate how the integrity of the lock is compromised.

Answer:

No. In the former function, if the lock's value is 1, then it will return 0, while if the value of lock is 0, then it will continuously return 1. So the function will be blocked by this sentence. The most important condition is that compare and swap is a atomic operation, so we don't need to worry about the value of lock changed in the function running process, while the latter do not is atomic, if the lock is changed afer the sentence **if (*lock**

== 0), then the wrong situation will appear.

1.5 Question 6.13

The first known correct software solution to the critical-section problem for two processes was developed by Dekker. The two processes, P0 and P1, share the following variables:

```
boolean flag[2]; /* initially false */
int turn;
```

The structure of process Pi (i == 0 or 1) is shown below. The other process is Pj (j == 1 or 0). Prove that the algorithm satisfies all three requirements for the critical-section problem.

```
while (true) {
    flag[i] = true;
    while (flag[j]) {
        if (turn == j) {
            flag[i] = false;
            while (turn == j)
            ; /* do nothing */
            flag[i] = true;
        }
    }
    /* critical section */
    turn = j;
    flag[i] = false;
    /* remainder section */
}
```

Answer: This algorithm satisfies all three requirements for the critical-section problem, and I will elaborate them one by one.

1. Mutual exclusion: There are two cases to consider: a. A process is inside the critical section: Without loss of generality, assume process j is inside the critical section. Before entering the critical section the process sets its own flag to 1. If process i tries to enter the critical section. it will see that flag[j] is up and gets caught up in the while loop. It will continue in the while loop until the other process sets its own flag to 0, which happens only at the end of the critical section. b. Two processes are trying to enter simultaneously: In this situation, if both processes reach their respective while loop at the top, then the variable turn will ensure that only one of them passes through. The variable turn is alternating between the allowing either process, and is only modified at the exit of the critical section.

2. Progress: There are also two cases to consider: a. One process is trying to enter with no competition: In such a case, the flag of the other process is down, and the process goes past the while look into the critical section directly. b. Two processes are trying to enter simultaneously. In this case if the first process is trapped into the while loop, then the variable turn will make one of the two variables lower its flag and goes into a loop waiting for the variable turn to change (the inner while loop). The other process whose turn is set by the variable turn will be able to get through.

3. Bounded Waiting: Assume there is a process blocked inside the inner while loop, while another process is in critical section. In such a case, if the process inside the critical section tries to re-enter, it will be blocked because on exit of the critical section. It has already set the variable turn to point to the other process. Therefore, the process that just got out of the critical section will be forced to wait for its own turn. As a result, bounded waiting is taken care of.

2 Chapter 7

2.1 Question 7.1

Explain why Windows and Linux implement multiple locking mechanisms. Describe the circumstances under which they use spinlocks, mutex locks, semaphores, and condition variables. In each case, explain why the mechanism is needed.

Answer: First of all, we need to know that these all operating systems provided different locking mechanisms is in order to cover the application developers’ needs.

1. Spinlocks are useful for multiprocessor systems where a thread can run in a busy-loop (for a short period of time) rather than incurring the overhead of being put in a sleep queue. The costs of frepquently switch the thread's state is unacceptable sometimes.

2. Mutexes are good idea for locking resources.

3. Semaphores and condition variables are more appropriate tools for synchronization when a resource must be held for a long period of time, since spinning is inefficient for a long duration. And a process occupying the calculate resource for a long time is also unacceptable as the resource of a computer is limited.

2.2 Question 7.3

Describe what changes would be necessary to the producer and consumer processes blocked so that a mutex lock could be used instead of a binary semaphore.

```
/*The structure of the producer process.*/
while (true) {
    ...
    /* produce an item in next produced */
    ...
    wait(empty);
    wait(mutex);
    ...
    /* add next produced to the buffer */
    ...
    signal(mutex);
    signal(full);
}


/*The structure of the consumer process.*/
while (true) {
    wait(full);
    wait(mutex);
    ...
    /* remove an item from buffer to next consumed */
    ...
    signal(mutex);
    signal(empty);
    ...
    /* consume the item in next consumed */
    ...
}
```

Answer: First of all, we need to know that they are work for different purpose. Mutual Exclusion semaphores are used to protect shared resources (data structure, file, etc..). While a binary semaphore is NOT protecting a resource from access. The act of Giving and Taking a semaphore are fundamentally decoupled. It is not work for protect resources. But it's not the key point in this question, in this situation, we should add a integer status representing full or empty.

```
/*The structure of the producer process.*/
while (true) {
    int status; // 0 empty, 1 full
    ...
    /* produce an item in next produced */
    ...
    status = 0;
    wait(mutex);
    ...
    /* add next produced to the buffer */
    ...
    signal(mutex);
    status = 1;
}
```

```
/*The structure of the consumer process.*/
while (true) {
    int status; // 0 empty, 1 full
    status = 1;
    wait(mutex);
    ...
    /* remove an item from buffer to next consumed */
    ...
    signal(mutex);
    status = 0;
    ...
    /* consume the item in next consumed */
    ...
}
```

3 Chapter 8

3.1 Question 8.12

Consider the traffic deadlock depicted in Figure 8.12.

(a) Show that the four necessary conditions for deadlock hold in this example.

(b) State a simple rule for avoiding deadlocks in this system.

Answer: a. According the course slides we can know that the four necessary conditions for a deadlock are:

1. mutual exclusion
2. hold and wait
3. no preemption
4. circular wait

And the mutual exclusion condition holds since only one car can occupy a space in the road. Hold-and-wait occurs where a car holds onto its place in the road while it waits to advance in the road. A car cannot be removed, just like preempted, from its position in the road. Lastly, there is indeed a circular wait as each car is waiting for a subsequent car to advance. If there are not a circular wait, the road position which representing the resource will be available later, so the deadlock will not appear, but we need to attention that, circular wait is not equal to the deadlock. If the same resource can be available as other car(process) completes its action, deadlock will not appear as well.

b. According to the shown system, a simple rule that would avoid this traffic deadlock is that a car may not advance into an intersection if it is obviously that the intersection can't be cleared immediately. This rule use waiting to avoid process join in the request of the resource.

3.2 Question 8.18

Which of the six resource-allocation graphs shown in Figure 8.12 illustrate deadlock? For those situations that are deadlocked, provide the cycle of threads and resources. Where there is not a deadlock situation, illustrate the order in which the threads may complete execution.

Answer:

The (b) and (d) illustrate deadlock.

(b): R1->T1->R3->T3->R1

(d): T2->R2->T4->R1->T2 and T1->R2->T3->R1->T1

Attention: I do not list all the possible order.

The possible order of (a) is: T2->T1->T3

The possible order of (c) is: T2->T3->T1

The possible order of (e) is: T2->T1->T3->T4

The possible order of (f) is: T2->T4->T1->T3

3.3 Question 8.22

Consider a system consisting of four resources of the same type that are shared by three threads, each of which needs at most two resources. Show that the system is deadlock free.

Answer: Suppose the system is deadlocked. According to the deadlock's basic condition we can know that each process is holding one resource and is waiting for one more. Since there are three processes and four resources, one process must be able to obtain two resources. We can infer that this process requires no more resources. Therefore it will return its resources when done. To be more specificly, we divide 4 to 3 part, there must be one part hold two resources or more, as each of which needs at most two resources, these one process can run with these resources, if it is completed, two more resources can be free, then 4 resources can is obviously enough for other two process.

3.4 Question 8.23

Consider a system consisting of m resources of the same type being shared by n threads. A thread can request or release only one resource at a time. Show that the system is deadlock free if the following two conditions hold:

(a) The maximum need of each thread is between one resource and m resources.

(b) The sum of all maximum needs is less than m + n

Answer: The proof:

condition a can be translated to Max$_i \geq 1$ for all i

condition b can be translated to $\sum_{i=1}^{n} Max_i < m + n$

If there is a deadlock, Need $_i = Max_i -$ Allocation $_i$ and $\sum_{i=1}^{n}$ Allocation $_i = m$.

According to the equation, we can get that $\sum_{i=1}^{n} Need_i < n$, but if the deadlock existed, the essential condition is that all process request one more resource. Obviously, $\sum_{i=1}^{n} Need_i < n$ can not support this condition, the system is deadlock free.

3.5 Question 8.28

Consider the following snapshot of a system: Allocation Max Available

	Allocation	Max	Available
	ABCD	ABCD	ABCD
T0	3141	6473	2224
T1	2102	4232	
T2	2413	2533	
T3	4110	6332	
T4	2221	5675	

Answer the following questions using the banker’s algorithm:

a. Illustrate that the system is in a safe state by demonstrating an order in which the threads may complete.

b. If a request from thread T4 arrives for (2, 2, 2, 4), can the request be granted immediately?

c. If a request from thread T2 arrives for (0, 1, 1, 0), can the request be granted immediately?

d. If a request from thread T3 arrives for (2, 2, 1, 2), can the request be granted immediately?

Answer:

a. We can run T2 first. As the allocated resources add the available resources is A4 B6 C3 D7, which is enough for cover the requirements A2 B5 C3 D3. Then the available resources become A4 B6 C3 D7, we can run T0. Then available resources become A7 B7 C7 D8, we can cover all process now. So the order is T2->T3->T0->T1->T4.

b. No, it will cause the available resources become 0,0,0,0, and T4 is still not hold the enough resources to run, while other process do not have enough resources as well, so all the process can not run.

c. Yes, if we allocate these resources to T2, we can still run all process in the order T2->T3->T0->T1->T4.

d. Yes, it can run in the order T3->T0->T1->T2->T4.