Operating System Homework 6

Jinyan Xu, 3160101126, Information Security

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 other processes must loop continuously to wait to enter the critical section in its CPU time slice, when a process is in its critical section.

Rather than engaging in busy waiting, the process can suspend itself. The suspend operation places a process into a waiting queue associated with the semaphore, and the state of the process is switched to the waiting state. Then control is transferred to the CPU scheduler, which selects another process to execute.

Yes, busy waiting can be avoided by processes giving up their CPU time slice, but block and wake processes will also carry overhead.

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

Answer:

The following may happen: In CPU, T_0 gets the lock, runs in the critical section -> T_1 spins -> T_0 runs -> T_1 spins -> T_0 runs -> T_1 spins …

Unless T_0 exit its critical section, T_1 will always spins. Time for T_1 has been totally wasted, so in single-processor systems, if the process is not quitting critical section, other processes have no choice but to spin, wasting their CPU time.

In multiprocessor systems, one process can spin on one processing core while another thread performs its critical section on another core.

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;
}
```

Describe how a race condition is possible in this situation and what might be done to prevent the race condition from occurring.

Answer:

The line: highestBid = amount; has a race condition, this operation isn't atomically, which can be interrupted. If many processes try to change this variables at the same time, its value is uncertain.

We can change the codes like this:

```
void bid(double amount) {
  double tmp = highestBid;
  if (amount > highestBid)
    compare_and_swap(highestBid, tmp, amount);
    //highestBid = amount;
}
```

We can also use a mutex lock to solve this:

```
Semaphore mutex = 1;
void bid(double amount) {
  wait(mutex);
  if (amount > highestBid)
    highestBid = amount;
  signal(mutex);
}
```

The mainly idea is to make sure the value isn't changed in the assignment process, and the assignment operation must be atomically.

6.11 One approach for using compare_and_swap() for implementing a spinlock 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) {
//-----------0----------
    if (*lock == 0) {
//-----------1----------
      /* 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:

This idiom works. When lock is 1, the process will spin in the while loop. And when lock is 0, it jumps out of the loop successfully.

Let's consider the special cases with lock is 0.

If the interrupt happens at --0--, and lock is changed to 1, just like lock is 1, the process will spin in the loop.

If the interrupt happens at --1--, and lock is changed to 1, ***compare_and_swap*** will help us to handle this, it will return 1, so statement "***break;***" will not execute.

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 in **Figure 6.19**. 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 */
}
```

Figure 6.19 The structure of process P_i in Dekker's algorithm.

Answer:

- Mutual Exclusion:

Only flag[j] == false and turn == i, P_i can enter critical section. If both flag[i] == flag[j] == true, then only one of them can enter the critical section, because turn can't be 1 and 0 at the same time. And when P_i is in the critical section, flag[i] == true and turn == i, which means P_j can't enter critical section.

- Progress:

Pi can be prevented from entering the critical section if flag[j] == true and turn == j. If P_j is not ready to enter the critical section, then flag[j] == false, and P_i can enter critical section. If P_i and P_j has set flag to true, it depends on turn. If turn == i, then P_i will enter the critical section. If turn == j, then P_j will enter the critical section. However, once P_j exits its critical section, it will reset flag[j] = false, turn = i, allowing Pi to enter the critical section. If P_j resets flag[j] = true, because turn == i, P_i will enter the critical section.

- Bound Waiting:

According above discussion, we can know that P_i will enter the critical section after at most one entry by P_j.