6.Synchronization

6.1 Race conditions are possible in many computer systems. Consider a banking system that maintains an account balance with two functions: deposit(amount) and withdraw(amount). These two functions are passed the amount that is to be deposited or withdrawn from the bank account balance. Assume that a husband and wife share a bank account. Concurrently, the husband calls the withdraw() function and the wife calls deposit(). Describe how a race condition is possible and what might be done to prevent the race condition from occurring.

Answer:

Assume the balance in the account is 250.00 and the husband calls withdraw(50) and the wife calls deposit(100). Obviously the correct value should be 300.00. Since these two transactions will be serialized, the local value of balance for the husband becomes 200.00, but before he can commit the transaction, the deposit(100) operation takes place and updates the shared value of balance to 300.00. We then switch back to the 1husband and the value of the shared balance is set to 200.00 - obviously an incorrect value.

6.2 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;
 do {
      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
} while(true);
```

Figure 6.25 The structure of process Pi in Dekker's algorithm

The structure of process Pi (i == 0 or 1) is shown in Figure 6.25; the other process is Pj (j == 1 or 0). Prove that the algorithm satisfies all three requirements for the critical-section problem.

Answer:

This algorithm satisfies the three conditions of mutual exclusion.

(1) Mutual exclusion is ensured through the use of the flag and turn variables. If both processes set their flag to true, only one will succeed, namely, the process whose turn it is. The waiting process can only enter its critical section when the other process updates the value of turn.

(2) Progress is provided, again through the flag and turn variables. This algorithm does not provide strict alternation. Rather, if a process wishes to access their critical section, it can set their flag variable to true and enter their critical section. It sets turn to the value of the other process only upon exiting its critical section. If this process wishes to enter its critical section again—before the other process—it repeats the process of entering its critical section and setting turn to the other process upon exiting.

(3) Bounded waiting is preserved through the use of the turn variable. Assume two processes wish to enter their respective critical sections. They both set their value of flag to true; however, only the thread whose turn it is can proceed; the other thread waits. If bounded waiting were not preserved, it would therefore be possible that the waiting process would have to wait indefinitely while the first process repeatedly entered—and exited—its critical section. However, Dekker's algorithm has a process set the value of turn to the other process, thereby ensuring that the other process will enter its critical section next.

6.3 The first known correct software solution to the critical-section problem for n processes with a lower bound on waiting of n − 1 turns was presented by Eisenberg andMcGuire. The processes share the following variables:

```
enum pstate {idle, want_in, in_cs};
pstate flag[n];
int turn;
```

All the elements of flag are initially idle. The initial value of turn is immaterial (between 0 and n-1). The structure of process Pi is shown in Figure 6.22. Prove that the algorithm satisfies all three requirements for the critical-section problem.

```
do{
    while(true){
        flag[i] = want_in;
        j = true;
        while(j != i){
            if(flag[j] != idle){
                j = turn;
            else
                j = (j+1) % n;
        }
        Flag[i] = in_cs;
        j = 0;

        while( (j < n) && (j == i || flag[j] != in_cs))
            j++;
        if( (j >= n) && (turn == i || flag[turn] == idel))
            break;
}
    /* critical section */
j = (turn + 1) % n;
while(flag[j] == idle)
    j = (j+1) %n;
turn = j;
flag[i] = idle;
    /* remainder section */
} while(ture);
```

Figure 6.22 The structure of process Pi in Eisenberg and McGuire's algorithm

Answer:
This algorithm satisfies the three conditions:
(1) Mutual exclusion is ensured through the use of the flag and turn variables. If both processes set their flag to true, only one will succeed, namely, the process whose turn it is. The waiting process can only enter its critical section when the other process updates the value of turn.
(2) Progress is provided, again through the flag and turn variables. This algorithm does not provide strict alternation. Rather, if a process wishes to access their critical section, it can set their flag variable to true and enter their critical section. It sets turn to the value of the other process only upon exiting its critical section. If this process wishes to enter its critical section again – before the other process – it repeats the process of entering its critical section and setting turn to the other process upon exiting.
(3)Bounded waiting is preserved through the use of the turn variable. Assume two processes wish to enter their respective critical sections. They both set their value of flag to true; however, only the thread whose turn it is can proceed; the other thread waits. If bounded waiting were not preserve, it would therefore be possible that the waiting process would have to wait indefinitely while the first process repeatedly entered – and exited – its critical section. However, Dekker's algorithm has a process set the value of turn to the other process, thereby ensuring that the other process will enter its critical section next.

6.4 Explain why implementing synchronization primitives by disabling interrupts is not appropriate in a single-processor system if the synchronization primitives are to be used in user level programs.
Answer:
If a user-level program is given the ability to disable interrupts, then it can disable the timer interrupt and prevent context switching from taking place, thereby allowing it to use the processor without letting other processes execute.

6.5 Explain why interrupts are not appropriate for implementing synchronization primitives in multiprocessor systems.
Answer:

Interrupts are not sufficient in multiprocessor systems since disabling interrupts only prevents other processes from executing on the processor in which interrupts were disabled; there are no limitations on what processes could be executing on other processors and therefore the process disabling interrupts cannot guarantee mutually exclusive access to program state.

6.6 The Linux kernel has a policy that a process cannot hold a spinlock while attempting to acquire a semaphore. Explain why this policy is in place.
Answer:
You cannot hold a spinlock while you acquire a semaphore, because you might have to sleep while waiting for the semaphore, and you cannot sleep while holding a spinlock.

6.7 Describe two kernel structures in which race conditions are possible and show how a race condition can occur.
Answer:
Some kernel data structures include a process id (pid) management system, kernel process table, and scheduling queues.With a pid management system, it is possible two processes may be created at the same time and there is a race condition assigning each process a unique pid. The same type of race condition can occur in the kernel process table: two processes are created at the same time and there is a race assigning them a location in the kernel process table. With scheduling queues, it is possible one process has been waiting for IO which is now available. Another process is being context-switched out. These two processes are being moved to the Runnable queue at the same time. Hence there is a race condition in the Runnable queue.

6.8 Describe how the Swap() instruction can be used to provide mutual exclusion that satisfies the bounded-waiting requirements.
Answer:

```
do {
        waiting[i] = TRUE;
        key = TRUE;
        while (waiting[i] && key)
                key = Swap(&lock, &key);
        waiting[i] = FALSE;
        / critical section /
        j = (i+1) % n;
        while ((j != i) && !waiting[j])
                j = (j+1) % n;
        if (j == i)
                lock = FALSE;
        else
                waiting[j] = FALSE;
        / remainder section /
} while (TRUE);
```

6.9 Consider how to implement a mutex lock using an atomic hardware instruction. Assume that the following structure defining the mutex lock is available:

```
typedef struct {
int available;
} lock;
```

(available == 0) indicates that the lock is available, and a value of 1 indicates that the lock is unavailable. Using this struct, illustrate how the following functions can be implemented using the test and set() and compare and swap() instructions:

- void acquire(lock *mutex)
- void release(lock *mutex)

Be sure to include any initialization that may be necessary.

Answer:

Test and Set:

```
typedef struct
        {
                int available;
        }lock;
void init(lock *mutex)
{
// available==0 -> lock is available, available==1 -> lock unavailable
mutex->available=0;
}
int test_And_Set(int *target)
{
                                        int rv = *target;
                                        *target = true;
                                        return rv
}
void acquire(lock *mutex)
{
        while(test_and_set(&mutex->available,1)==1);
}
void release(lock *mutex)
{
                mutex->available=0;
}
```

Compare and Swap:

```
int compare_and_Swap(int *ptr,int expected,int new)
{
                                    int actual = *ptr;
                                    if(actual == expected)
                                    *ptr = new;
                                    return actual;
}

void acquire(lock *mutex)
{
                while(compare_and_swap(&mutex->available,0,1)==1);
}
```

```
void release(lock *mutex)
{
        mutex->available=0;
}
```

6.10 The implementation of mutex locks provided in Section 6.5 suffers from busy waiting. Describe what changes would be necessary so that a process waiting to acquire a mutex lock would be blocked and placed into a waiting queue until the lock became available.

Answer:

To prevent the diminishing efficiency of operating systems in such cases the concept of introducing spinlocks may be beneficial enough.

- ⇨ A lock that makes threads of a process enter into wait condition, while searching for availability of desired lock is known as a spinlock
- ⇨ Spinlocks are taken for small durations and hence may not impact that much on the computer resources.
- ⇨ Also, introducing spinlocks may allow threads to divide processors actively according to their needs. Where the thread waiting for a spinlock may run on one processor and other threads can concurrently run over other processors without getting interrupted.

6.11 Assume that a system has multiple processing cores. For each of the following scenarios, describe which is a better locking mechanism—a spinlock or a mutex lock where waiting processes sleep while waiting for the lock to become available:

• The lock is to be held for a short duration.

• The lock is to be held for a long duration.

• A thread may be put to sleep while holding the lock.

Answer:

(1) If the lock is to be held for a short duration, it makes more sense to use a spinlock as it may in fact be faster than using a mutex lock which requires suspending - and awakening - the waiting process.

(2) If it is to be held for a long duration, a mutex lock is preferable as this allows the other processing core to schedule another process while the locked process waits.

(3) If the thread may be put to sleep while holding the lock, a mutex lock is definitely preferable as you wouldn't want the waiting process to be spinning while waiting for the other process to wake up.

6.12 Assume that a context switch takes T time. Suggest an upper bound (in terms of T) for holding a spinlock. If the spinlock is held for any longer, a mutex lock (where waiting threads are put to sleep) is a better alternative.

Answer:

The upper bound should be double of T (2*T). If the spinlock is held any longer than this, the thread(s) should be put to sleep. So instead of potentially doubling (or more) the time for a process to complete by switching constantly between two processes, one process can finish then awaken the sleeping thread.

6.13 A multithreaded web server wishes to keep track of the number of requests it services (known as hits.) Consider the following two strategies to prevent a race condition on the variable hits. The first strategy is to use a basic mutex lock when updating hits:

```
int hits;
mutex_lock hit_lock;
hit_lock.acquire();
hits++;
hit_lock.release();
```

A second strategy is to use an atomic integer:

```
Atomic_t hits;
Atomic_inc(&hits);
```

Explain which of these two strategies is more efficient.

Answer:

First method: System call is required for lock and it will put a process to sleep. So, it requires context switching. If the lock is unavailable. The process awakening will also require another subsequent context switch.

Second method: Atomic update of hits variables are provided by atomic integer. It also ensures norace condition on hits. No kernel intervention is need in this case. Therefore the second method ismore efficient than the first one.

6.14 Consider the code example for allocating and releasing processes shown in the Figure below.

```
#define MAX PROCESSES 255
int number of processes = 0;
/* the implementation of fork() calls this function */
int allocate process() {
        int new pid;
        if (number of processes == MAX PROCESSES)
                return -1;
        else {
/* allocate necessary process resources */
        ++number of processes;
        }
}
return new pid;
/* the implementation of exit() calls this function */
void release process() {
        /* release process resources */
        --number of processes;
}
```

a. Identify the race condition(s).

b. Assume you have a mutex lock named mutex with the operations acquire() and release(). Indicate where the locking needs to be placed to prevent the race condition(s).

c. Could we replace the integer variable

Answer:

a. There is a race condition on the variable number of processes.

b. A call to acquire() must be placed upon entering each function and a call to release() immediately before exiting each function.

c. No, it would not help. The reason is because the race occurs in the allocate process() function where number of processes is first tested in the if statement, yet is updated afterwards, based upon the value of the test. it is possible that number of processes = 254 at the time of the test, yet because of the race condition, is set to 255 by another thread before it is incremented yet again.

6.15 Servers can be designed to limit the number of open connections. For example, a server may wish to have only N socket connections at any point in time. As soon as N connections are made, the server will not accept another incoming connection until an existing connection is released. Explain how semaphores can be used by a server to limit the number of concurrent connections.

Answer:

A semaphore is initialized to the number of allowable open socket connections. When a connection is accepted, the acquire() method is called; when a connection is released, the release() method is called. If the system reaches the number of allowable socket connections, subsequent calls to acquire() will block until an existing connection is terminated and the release method is invoked.

6.16 Windows Vista provides a new lightweight synchronization tool called slim reader-writer locks. Whereas most implementations of reader-writer locks favor either readers or writers or perhaps order waiting threads using a FIFO policy, slim reader-writer locks favor neither readers or writers, nor are waiting threads ordered in a FIFO queue. Explain the benefits of providing such a synchronization tool.

Answer:

Simplicity. IF RW locks provide fairness or favor readers or writers, there is more overhead to the lock. By providing such a simple synchronization mechanism, access to the lock is fast. Usage of this lock may be most appropriate for situations where reader-locks are needed, but quickly acquiring and releasing the lock is similarly important.

6.17 Show how to implement the wait() and singal() semaphore operations in multiprocessor environments using the TestAndSet() instruction. The solution should exhibit minimal busy waiting.

Answer:

Here is the pseudocode for implementing the operations:

```
int guard = 0;
int semaphore value = 0;
wait()
{ while (TestAndSet(&guard) == 1);
  if (semaphore value == 0) { atomically add process to a queue of processes
                                    waiting for the semaphore and set guard to 0;
                                    }
  else { semaphore value--;
            guard = 0;
       }
}
signal()
{ while (TestAndSet(&guard) == 1);
if (semaphore value == 0 && there is a process on the wait queue)
                        wake up the first process in the queue of waiting processes
else
  semaphore value++;
guard = 0;
}
```

6.18 從缺

6.19 Demonstrate that monitors and semaphores are equivalent in so far as they can be used to implement the same types of synchronization problems.

Answer:

A semaphore can be implemented using the following monitor code:

```
monitor semaphore {
        int value = 0;
        condition c;
        semaphore increment() {
                value++;
                c.signal();
        }
        semaphore decrement() {
                while (value == 0)
                c.wait();
                value--;
        }
}
```

A monitor could be implemented using a semaphore in the following manner. Each condition variable is represented by a queue of threads waiting for the condition. Each thread has a semaphore associated with its queue entry. When a thread performs a wait operation, it creates a new semaphore (initialized to zero), appends the semaphore to the queue associated with the condition variable, and performs a blocking semaphore decrement operation on the newly created semaphore. When a thread performs a signal on a condition variable, the first process in the queue is awakened by performing an increment on the corresponding semaphore.

6.20 Design an algorithm for a bounded-buffer monitor in which the buffers(portions) are embedded within the monitor itself.

Answer:

```
monitor bounded     buffer {
        int items[MAXITEMS];
        int numItems = 0;
        condition full, empty;

        void produce(int v) {
                while (numItems == MAXITEMS) full.wait();
                items[numItems++] = v;
                empty.signal();}
```

```
int consume() {
    int retVal;while (numItems == 0) empty.wait();
    retVal = items[--numItems];
    full.signal();return retVal;}}
```

6.21 The strict mutual exclusion within a monitor makes the bounded-buffer monitor of Exercise 6.20 mainly suitable for small portions.
a) Explain why this is true.
b) Design a new scheme that is suitable for larger portions.
Answer:
The solution to the bounded buffer problem given above copies the produced value into the monitor's local buffer and copies it back from the monitor's local buffer to the consumer. These copy operations could be expensive if one were using large extents of memory for each buffer region. The increased cost of copy operation means that the monitor is held for a longer period of time while a process is in the produce or consume operation. This decreases the overall throughput of the system. This problem could be alleviated by storing pointers to buffer regions within the monitor instead of storing the buffer regions themselves. Consequently, one could modify the code given above to simply copy the pointer to the buffer region into and out of the monitor's state. This operation should be relatively inexpensive and therefore the period of time that the monitor is being held will be much shorter, thereby increasing the throughput of the monitor.

6.22 Discuss the tradeoff between fairness and throughput of operations in the readers-writers problem. Propose a method for solving the readers/writers problem without causing starvation.
Answer:
Throughput in the readers-writers problem is increased by favoring multiple readers as opposed to allowing a single writer to exclusively access the shared values. On the other hand, favoring readers could result in starvation for writers. The starvation in the readers/writers problem could be avoided by keeping timestamps associated with waiting processes. When a writer is finished with its task, it would wake up the process that has been waiting for the longest duration. When a reader arrives and notices that another reader is accessing the database, then it would enter the critical section only if there are no waiting writers. These restrictions would guarantee fairness.

6.23 How does the signal() operation associated with monitors differ from the corresponding operation defined for semaphores?
Answer:
The signal() operations associated with monitors is not
persistent in the following sense: if a signal is performed and if there are no waiting threads, then the signal is simply ignored and the system does
not remember the fact that the signal took place. If a subsequent wait operation is performed, then the corresponding thread simply blocks. In semaphores, on the other hand, every signal results in a corresponding increment of the semaphore value even if there are no waiting threads. A future wait operation would immediately succeed because of the earlier
increment.

6.24 Suppose the signal() statement can appear only as the last statement in a monitor function. Suggest how the implementation described in Section 6.8 can be simplified in this situation.
Answer:
If the signal operation were the last statement, then the lock could be transferred from the signalling process to the process that is the recipient of the signal. Otherwise, the signalling process would have to explicitly release the lock and the recipient of the signal would have to compete with all other processes to obtain the lock to make progress.

6.25 Consider a system consisting of processes P1, P2,..., Pn, each of which has a unique priority number. Write a monitor that allocates three identical line printers to these processes, using the priority numbers for deciding the order of allocation.
Answer:

Here is the pseudocode:

```
  type resource = monitor
var P: array[3] of boolean;    X: condition;
procedure acquire (id: integer, printer-id: integer);
    begin
       if P[0] and P[1] and P[2] then X.wait(id)
       if not P[0] then printer-id := 0;
             else if not P[1] then printer-id := 1;
                   else printer-id := 2;
       P[printer-id]:=true;
    end
procedure release (printer-id: integer)
   begin
       P[printer-id]:=false;
       X.signal;
    end
    begin
       P[0] := P[1] := P[2] := false;
end
```

6.26 A file is to be shared among different processes, each of which has a unique number. The file can be accessed simultaneously by several processes, subject to the following constraint: The sum of all unique numbers associated with all the processes currently accessing the file must be less than n. Write a monitor to coordinate access to the file.

Answer:

```
int sumid=0;       /* Shared var that contains the sum of the process ids currently accessing the file */
int waiting=0;     /* Number of process waiting on the semaphore OkToAccess */
semaphore mutex=1;     /* Our good old Semaphore variable ;) */
```

```
semaphore OKToAccess=0;        /* The synchronization semaphore */

get_access(int id)
{
        sem_wait(mutex);
        while(sumid+id > n) {
                waiting++;
                sem_signal(mutex);
                sem_wait(OKToAccess);
                sem_wait(mutex);
        }
        sumid += id;
        sem_signal(mutex);
}
release_access(int id)
{
        int i;
        sem_wait(mutex);
        sumid -= id;
        for (i=0; i < waiting;++i) {
                sem_signal(OKToAccess);
        }
        waiting = 0;
        sem_signal(mutex);
}
main()
{
        get_access(id); do_stuff(); release_access(id); }
```

6.27 When a signal is performed on a condition inside a monitor, the signaling process can either continue its execution or transfer control to the process that is signaled. How would the solution to the preceding exercise differ with the two different ways in which signaling can be performed?

Answer:

The solution to the previous exercise is correct under both situations. However, it could suffer from the problem that a process might be awakened only to find that it is still not possible for it to make forward progress either because there was not sufficient slack to begin with when a process was awakened or if an intervening process gets control, obtains the monitor and starts accessing the file. Also, note that the broadcast operation wakes up all of the waiting processes. If the signal also transfers control and the monitor from the current thread to the target, then one could check whether the target would indeed be able to make forward progress and perform the signal only if it were possible. Then the "while" loop for the waiting thread could be replaced by "if" condition since it is guaranteed that the condition will be satisfied when the process is woken up.

6.28 Suppose we replace the wait() and signal() operations of monitors with a single construct await(B),where B is a general Boolean expression that causes the process executing it to wait until B becomes true.
a. Write a monitor using this scheme to implement the readers– writers problem.
b. Explain why, in general, this construct cannot be implemented efficiently.
c. What restrictions need to be put on the await statement so that it can be implemented efficiently? (Hint: Restrict the generality of B; see Kessels [1977].)
42 Chapter 6 Process Synchronization
Answer:
The readers-writers problem could be modified with the following more generate await statements: A reader can perform "await(active writers == 0 && waiting writers == 0)" to check that there are no active writers and there are no waiting writers before it enters the critical section. The writer can perform a "await(active writers == 0 && active readers == 0)" check to ensure mutually exclusive access.
The system would have to check which one of the waiting threads have to be awakened by checking which one of their waiting conditions are satisfied after a signal. This requires considerable complexity as well as might require some interaction with the compiler to evaluate the conditions at different points in time.
One could restrict the boolean condition to be a disjunction of conjunctions with each component being a simple check (equality or inequality with respect to a static value) on a program variable. In that case, the boolean condition could be communicated to the runtime system, which could perform the check every time it needs to determine which thread to be awakened.