Process Co-ordination and Synchronization

We have discussed definition of a process along types of process and different states a process goes through during its life cycle. In this lecture we will further describe how processes coordinate their activities while sharing information and data with other processes. A cooperating process can affect or be affected by other processes executing in the system. Cooperating processes can either directly share a logical address space (that is, both code and data) or be allowed to share data only through files or messages. Concurrent access to shared data may result in data inconsistency, we will discuss various mechanisms, to ensure the orderly execution of cooperating processes that share a logical address space, so that data consistency is maintained.

Processes execute concurrently, and CPU is multiplexed (switched rapidly between processes) to support concurrent execution. This means that one process may only partially complete execution before another process is scheduled. In fact, a process may be interrupted at any point in its instruction stream, and the processing unit may be assigned to execute instructions of another process. By using a simple example, we explain how concurrent execution can contribute to issues involving the integrity of data shared by several processes.

In a single-processor multiprogramming system, the user can switch from one application to another, and each application uses the same keyboard for input and the same screen for output. Because each application needs to use the procedure echo, it makes sense for it to be a shared procedure.

```
void echo()
{
chin = getchar();
chout = chin;
putchar(chout);
}
```

This procedure shows the essential elements of a program that will provide a character echo procedure; input is obtained from a keyboard one keystroke at a time. Each input character is stored in variable _chin_. It is then transferred to variable _chout_ and sent to the display. Any program can call this procedure repeatedly to accept user input and display it on the user's screen. If 2 processes P1 and P2 call echo() procedure and execute it separately, it will display character as input by the process. Since echo() procedure is shared, concurrent execution of processes accessing echo() procedure can lead to problems. Context switch occurs while two processes are executing statements of echo() procedure and concurrent execution of statements can produce wrong results as shown in sequence listed below:

T0: Process P1 invokes the echo() procedure and is interrupted immediately after Getchar() returns its value and stores it in _chin_. At this point, the most recently entered character, 'A' is stored in variable _chin_.

T1: Process P2 is activated and invokes the echo() procedure, which runs to conclusion, inputting and then displaying a single character, 'B' on the screen.

T2: Process P1 is resumed. By this time, the value 'A' has been overwritten in _chin_ and lost. Instead, _chin_ contains 'B', which is transferred to chout() and displayed.

Thus, the first character is lost and the second character is displayed twice. This incorrect state occurs because both processes manipulated the variable _chin_ concurrently. A situation like this, where several processes access and manipulate the same data concurrently and the outcome of the execution depends on the particular order in which the access takes place, is

called a *race condition*. We further describe race condition using simple book-keeping application where two processes share two global variables.
Suppose two items of data A and B are to be maintained in the relationship A = B. That is, any program that updates one value must also update the other to maintain the relationship. Now consider the following two processes:

```
P1:
        A = A + 1;
        B = B + 1;
P2:
        B = 2 *B;
        A = 2 *A;
```

If the state is initially consistent, each process taken separately will leave the shared data in a consistent state. Now consider the following concurrent execution sequence, of process P1 and P2:

```
T0:     A = A + 1;;
T1:     B = 2 *B;
T2:     B = B + 1;
T3      A = 2 *A;
```

At the end of this execution sequence, the condition A = B, no longer holds. For example, if we start with A = B = 1, at the end of this execution sequence we have
A = 4 and B = 3.

To guard against the race condition, we need to ensure that only one process at a time can be manipulating shared variables. Situations such as the one just described occur frequently in operating systems as different parts of the system manipulate shared resources. Such situations are studied as a critical problem, which needs to be addressed to get required results of concurrent processes sharing resources.

Critical-Section Problem

A process is in critical section, when the process may be changing common variables, updating a table, writing a file, and so on. Consider a system consisting of n processes {P0, P1, ..., Pn−1}. Each process has a segment of code, called a critical section. The important feature of the system is that, when one process is executing in its critical section, no other process is allowed to execute in its critical section. That is, no two processes are executing in their critical sections at the same time. The solution to critical-section problem is to design a protocol that the processes can use to cooperate. Each process must request permission to enter its critical section. The section of code implementing this request is the entry section. The critical section may be followed by an exit section. The remaining code is the remainder section. The general structure of a typical process Pi is listed below.

```
while (true); {
        beginning section;
        entry section;
                critical section;
        exit section;
        remainder section;
}
```

The entry section and exit section are underlined to highlight these important segments of code. A solution to the critical-section problem must satisfy following 3 requirements:

1. *Mutual exclusion*: If process Pi is executing in its critical section, then no other processes can be executing in their critical sections.
2. *Progress*: If no process is executing in its critical section and some processes wish to enter their critical sections, then only those processes that are not executing in their remainder sections can participate in deciding which will enter its critical section next, and this selection cannot be postponed indefinitely.
3. *Bounded waiting*: There exists a bound, or limit, on the number of times that other processes are allowed to enter their critical sections after a process has made a request to enter its critical section and before that request is granted.

In operating system design, different mechanisms were explored to solve critical section problem. Commonly used approaches are:

- Software
- Hardware
- Operating System
- High Level Language Constructs

Software Based Solution of the Critical Section Problem

Code of the processes is embedded with the instructions to discipline their activities to ensure that shared resources are accessed and satisfy the requirements for the solution of the critical-section problem. Different attempts were made to provide software based solution, here we describe Peterson's solution [Peterson 1981]. This solution is restricted to two processes that alternate execution between their critical sections and remainder sections. The processes are numbered P0 and P1. Peterson's solution requires two processes to share two data items:

```
int turn;
boolean flag[2];
```

For convenience, when presenting Pi, we use Pj to denote the other process; which in this case means, j equals 1 − i. The variable turn indicates whose turn it is to enter its critical section. That is, if turn == i, then process Pi is allowed to execute in its critical section. The flag array is used to indicate if a process is ready to enter its critical section. For example, if flag[i] is true, this value indicates that P_0 is ready to enter its critical section. Now we can describe the algorithm listed below:

```
Process P(i)  {
        while (true) {
                beginning section
                flag[i] = true;
                turn = j;
                while (flag[j] && turn == j)
                        do-nothing;
                <Critical Section>;
                flag[i] = false;
                remainder section
        }
```

To enter the critical section, process Pi first sets flag[i] to be true and then sets turn to the value j, thereby asserting that if the other process wishes to enter the critical section, it can do so. If both processes try to enter at the same time, turn will be set to both i and j at roughly the same time. Only one of these assignments will last; the other will occur but will be overwritten immediately. The eventual value of turn determines which of the two processes is allowed to enter its critical section first.
We can prove that this solution is correct and can do that by showing that:

1. Mutual exclusion is preserved.
2. The progress requirement is satisfied.
3. The bounded-waiting requirement is met

To prove property 1, we note that each Pi enters its critical section only if either flag[j] == false or turn == i. Also note that, if both processes can be executing in their critical sections at the same time, then flag[0] == flag[1] == true. These two observations imply that P0 and P1 could not have successfully executed their while statements at about the same time, since the value of turn can be either 0 or 1 but cannot be both. Hence, one of the processes say, Pj must have successfully executed the while statement, whereas Pi had to execute at least one additional statement ("turn == j"). However, at that time, flag[j] == true and turn == j, and this condition will persist as long as Pj is in its critical section; as a result, mutual exclusion is preserved.
To prove properties 2 and 3, we note that a process Pi can be prevented from entering the critical section only if it is stuck in the while loop with the condition flag[j] == true and turn == j; this loop is the only one possible. If Pj is not ready to enter the critical section, then flag[j] == false, and Pi can enter its critical section. If Pj has set flag[j] to true and is also executing in its while statement, then either turn == i or turn == j. If turn == i, then Pi will enter the critical section. If turn == j, then Pj will enter the critical section. However, once Pj exits its critical section, it will reset flag[j] to false, allowing Pi to enter its critical section. If Pj resets flag[j] to true, it must also set turn to i. Thus, since Pi does not change the value of the variable turn while executing the while statement, Pi will enter the critical section (progress) after at most one entry by Pj (bounded waiting).

We further explain the Peterson's solution by using simple book-keeping application discussed earlier. Shared variables are listed below:

```
int turn, A, B;  (A=B=2)
boolean flag[2]; Initially flag[i] = flag[j] = false;
```

Pi and Pj code along with critical section code, entry to critical section and exit sections.

```
Pi ()
        while (true); {
                ;    /* beginning section */
        flag[i] = true;
        turn = j;
        while (flag[j] && turn ==j);
                A = A + 1;  /* Critical Section  */
                B = B + 1;
        flag[i] = false;
        remainder section
        } do
```

```
Pj ()
        while (true); {
                ; /* beginning section */
        flag[j] = true;
        turn = i;
        while (flag[i] && turn == i);
        B = 2 *B;    /* Critical Section  */
        A = 2 * A;
        flag[j] = false;
        ;  /* remainder section  */
        } do
```

Now we execute statements (for that we execute process Pi and Pj) in the same sequence.

T0: A = A+1; → process Pi is executed. It will set flag[i] to true, and turn to 'j'. Process Pi will enter critical section and increment value of 'A' (3). Time slice expires, and context switch is made.

T1: B = 2*B; → process Pj is executed. It will set flag[j] to true, and turn to 'i'. Now flag[j] == true and turn ==i, so Pj is executing while statement and will not enter in its Critical Section and will waste its time slice.

T2: B= B+1; → process Pi is to be executed. It will execute statement from the point when T0 time slice expired. Process Pi increment value of 'B' (3). It will continue to execute other statements and exit critical section by setting flag[i] to false.

T3: Process Pj is allocated another time slice, and will continue from the point where it was context switched at expiry of T1. Now flag[i] is false and turn = i, so Pj will enter critical section, and execute B = 2 *B; and setting value of B to '6'. After that next statement A = 2 * A; is executed, setting value of A to '6'. After that Pj will exit critical section by setting flag[i] to false.

While tracing the code, we noticed that Peterson's solution ensures that an any given time only one process is in critical section, to satisfy Mutual Exclusion requirement. To prove properties 2 and 3, we note that a process Pj is prevented from entering the critical section, since it was stuck in the while loop with the condition flag[i] == true and turn == i. When Pi exits its critical section, it will reset flag[i] to false, allowing Pj to enter its critical section. If Pj resets flag[j] to true, it must also set turn to i. Thus, since Pj does not change the value of the variable turn while executing the while statement, Pj will enter the critical section (progress) after at most one entry by Pi (bounded waiting). If you look carefully Peterson's solution in this example, it actually executes processes in alternate way, by synchronizing processes to satisfy critical section solution requirements. Software-based solutions are not guaranteed to work on modern computer architectures. Software-based solutions employ busy waiting, where a process continues to use CPU time while it is waiting for access to a critical section.

Hardware Based Solution of the Critical Section Problem

Mutual exclusion can easily be ensured by preventing a process from being interrupted. By using disabling and enabling interrupts primitives defined by the OS kernel. In the critical section problem process we can use disable interrupt at the entry section, and enable interrupt at exit section as listed below.

```
while (true); {
        disable interrupts      /* entry section  */
                critical section
        disable interrupts      /* exit section  */
        remainder section
        }
```
Though the solution seems simple, the system may miss some important interrupts. Processor designers have provided special hardware instructions that allow us either to test and modify the content of a word or to swap the contents of two words atomically-that is, as one uninterruptible unit. During execution of this type of instruction, access to the memory location is blocked for any other instruction referencing that location. In a relatively simple manner, these special instructions can be used to solve the critical-section problem. In the following section we describe at an abstract level concepts behind these types of instructions by describing the test and set() and compare and swap instructions.
The test&set() instruction can be defined as follows:

```
test&set (lock) {
   result = lock;       /* return result value of 'lock' and */
   lock  = 1;          /* set value of 'lock' to 1                  */
   return result;
}
```
The important characteristic of this instruction is that it read a value and write a new value atomically, and hardware is responsible for implementing this correctly. Thus, if two test&set() instructions are initiated by two processes, they will be executed sequentially in some arbitrary order. If the machine supports the test&set() instruction, then it can be used to ensure mutual exclusion by declaring a variable lock, initialized to '0'.
The structure of process Pi is listed below:
```
Process P(i)
while (condition) {
        while (test&set(&lock) == 1)
                ; /* do nothing */
          < C.S.>;     /* critical section  */
        lock = 0;
        /* remainder section */
}
```
Compare and Swap instruction (compare&swap) also called a compare and exchange instruction, can be defined as follows:
```
compare&swap (*memory, reg1, reg2) {
        if (reg1 == *memory) {   /* If address memory  == reg1,
                *memory  = reg2;      /* then  put reg2 in memory
         return *memory;
         }
```
The instruction checks a memory location (*memory) against reg1 value. If the memory location's current value is reg1, it is replaced with value of reg2; otherwise, it is left unchanged. The old memory value is always returned; thus, the memory location has been updated if the returned value is the same as the reg1 value. This atomic instruction therefore has two parts: A compare is made between a memory value and a reg1 value; if the values are the same, a swap occurs. The entire compare and swap function is carried out

atomically, that means, it is not subject to interruption. This instruction can be used to ensure mutual exclusion by declaring a shared variable lock, which is initialized to ‘0’.

```
Process P(i)
while (true) {
        while (compare&swap(lock, 0, 1) == 1)
                /* do nothing */;
        <C.S. code>;            /* critical section */;
        lock = 0;
        /* remainder */;
        }
}
```

Locks (spin locks)

To solve the critical-section problem using special hardware instructions is complicated and generally these instructions are inaccessible to application programmers. Operating-systems designers build software tools to solve the critical-section problem. The simplest of these tools is the lock and works as follow:

Lock: prevents someone from doing something.
Lock before entering critical section and before accessing shared data.
Unlock when leaving, after accessing shared data.
Wait if locked.

We can use the lock to protect critical sections and prevent race conditions. In this case, process must acquire the lock before entering a critical section; it releases the lock when it exits the critical section. The acquire() function acquires the lock, and the release() function releases the lock as listed below:

```
acquire() {
  while (!available)
     ; /* busy wait */
  available = false;;
}
```

```
release() {
     available = true;
  }
```

A lock has a boolean variable whose value indicates if the lock is available or not. If the lock is available, a call to acquire() succeeds, and the lock is then considered unavailable. Calls to either acquire() or release() must be performed atomically. A process that attempts to acquire an unavailable lock, waits till the lock is released. The critical section problem can be solved by using locks as listed below:

```
P (i)
while(condition) {
        acquire() lock;
                <Critical Section>
        release(lock);
  remaining Section
}
```

The locks are often implemented using special hardware instructions (test&set or compare&swap) or disabling/enabling interrupts. The main disadvantage of using locks is that it requires busy waiting, and wastes CPU cycles that some other process might be able to use productively. While a process is in its critical section, any other process that tries to enter its critical section must loop continuously in the call to acquire() the lock. This type of lock is also called a spinlock because the process “spins” while waiting for the lock to become available.

References

[Peterson 1981] G. L. Peterson, “Myths About the Mutual Exclusion Problem”, *Information Processing Letters*, Volume 12, Number 3 (1981).

[Silberschatz 2013] A. Silberschatz, P. B. Galvin, and G. Gagne, *Operating System Concepts, Ninth Edition*, Wiley 2013.

[Stallings 2012] W. Stallings, Operating systems: internals and design principles, 7th Edition, Prentice Hall 2012.