Concurrency: Mutual Exclusion and Synchronization - Part 1

1 Introduction

So far we have discussed process and thread, and according to multiprogramming and multithreading, we know either process or thread may run simultaneously with other processes or threads, which thus raises an issue of *concurrency*.

To deal with multiple processes or threads, the operating system needs to switch control between them from time to time based on the state transition model and accordingly save and restore their contexts. In a uniprocessor multiprogramming system, processes are interleaved in time to yield the appearance of simultaneous execution, as Figure 1 (a) illustrated, while in a multiple-processor system, it is possible not only to interleave processes but also to overlap them (Figure 1 (b)).

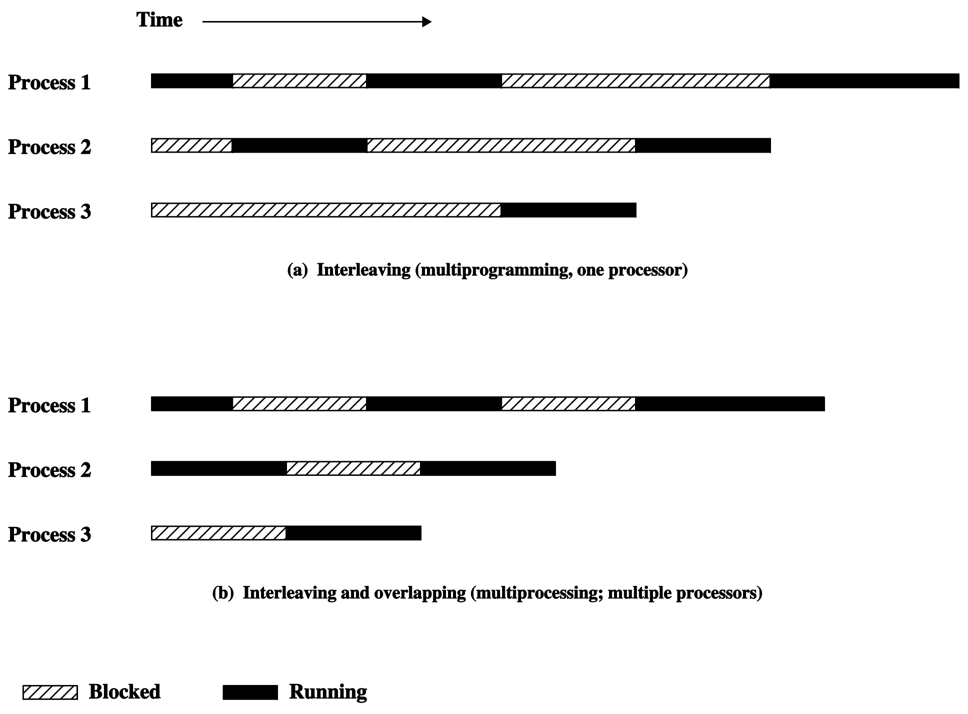

Time
Process 1
Process 2
Process 3
(a) Interleaving (multiprogramming, one processor)
Process 1
Process 2
Process 3
(b) Interleaving and overlapping (multiprocessing; multiple processors)
Blocked
Running

Figure 1: Multiprogramming and multiprocessing

Although on the surface the two cases are different, they present the same problems: *the relative speed of execution of processes cannot be predicted*.

1.1 A simple example

Consider the following procedure in a process:

```
void echo() {
  char_in = getchar();
  char_out = char_in;
  putchar(char_out);
}
```

where *char_in* and *char_out* are global variables. The procedure first obtains a char from the standard input and store it in *char_in*, then transfers the char to *char_out*, and finally display it on the user's screen.

Suppose in the process there are two threads: *T1* and *T2*, both of which will invoke *echo()* to accept user input and echo it on screen. If the execution traces of the two threads are as follows:

```
T1:                                     T2:
    char_in = getchar();                    .
    char_out = char_in;                     .
    putchar(char_out);                      .
    .                                       char_in = getchar();
    .                                       char_out = char_in;
    .                                       putchar(char_out);
```

That is *T2* did not invoke *echo()* until *T1* finishes its invocation. Obviously there is no problem here, and every thread gets what it wants. However, how about the following sequence:

```
T1:                                     T2:
    char_in = getchar();                    .
    .                                       char_in = getchar();
    char_out = char_in;                     .
    putchar(char_out);                      .
    .                                       char_out = char_in;
    .                                       putchar(char_out);
```

In this case, *T1* accepts a char first and stores it in *char_in*, then the control switches to *T2* which does the same thing. Here comes a problem. That is the char stored in *char_in* by *T1* is overwritten and thus lost due to the execution of *T2*. Finally the char accepted by *T2* will be outputted twice by the two threads. This is not what the user expects. The point of this example is that the unpredicted relative speed of execution of threads or processes may cause unpredicted results.

The above example is assumed to be on a uniprocessor system. For a multi-processor system, if we take the following sequence as an example, obviously the same problem also exists.

```
T1:                                          T2:
    char_in = getchar();                         .
    .                                            char_in = getchar();
    char_out = char_in;                          char_out = char_in;
    putchar(char_out);                           .
    .                                            putchar(char_out);
```

1.2 Another simple example

The operating system may allocate resources to processes upon their requests, however the concurrency of the processes may lead to a dilemma.

Suppose there are two processes in the system: *P1* and *P2*. Either of them needs both *Resource A* and *Resource B*. Typically they request for the resources one at a time. After the operating system allocates the requested resource, the owner may hold the resource for a while before it is released. Thus at some moment, we may have a snapshot of the allocation of resources as illustrated in Figure 2, where an arrow from a process to a resource indicates a relationship of *Requests*, and one from a resource to a process a relationship of *held by*. According to Figure 2, *Resource A* and *Resource B* are held by respectively *P2* and *P1*, who also requests for the resource already held by the other. This forms a *circular wait*. To proceed, both *P1* and *P2* have to wait until the other releases their resource, but without external interference there is no possibility for them to give up what they have got. So from this moment, no progress can be made and the processes enter a dilemma, called *deadlock*.

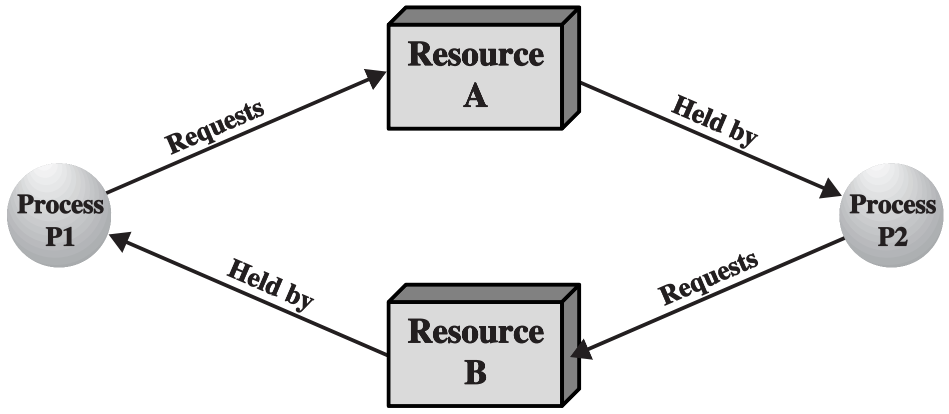


Figure 2: Circular waiting

1.3 Problems caused by concurrency

The above example presents the following difficulties:

- The concurrent access to global resources by multiple processes may lead to unpredictable results.
- The exclusive access to resources may lead to a deadlock.
- Unpredictable results make it difficult to locate a programming error in a concurrent application since the results are not deterministic and reproducible.

1.4 Process interaction

Before we explore how to solve the above problems, we first need to know something behind the scene, i.e. the relationship between concurrent threads and/or processes.

Competition among processes for resources

Concurrent processes come into conflict with each other when they are competing for the use of the same resource. They are not necessarily aware of each other, but the execution of one process may affect the behavior of competing processes.

There are three control problems that must be considered:

- **Mutual exclusion**

  Suppose two processes both wish access to a single resource, say a printer. Obviously such a resource cannot be accessed by more than one process at the same time. We cannot image a sheet with the upper half for one process and the lower half for the other process.

  We refer to such a resource like the printer as a *critical resource* and the portion of the program that uses it a *critical section* of the program. For example, in the above *echo()* procedure, global variable *char_in* and *char_out* are critical resources, and the body of *echo()* is a critical section.

  It is important to allow only one process at a time in its critical section, so that expectable results are obtained always, which is referred to as *mutual exclusion*.

- **Deadlock**

  Deadlock that we have mentioned above is actually an effect of mutual exclusion. As we can see, deadlock always involves more than one process and more than one resource.

- **Starvation**

  Starvation is another control problem due to the enforcement of mutual exclusion. Consider we have three processes, *P1*, *P2*, and *P3*, competing for a resource *R*. Suppose each of them require periodic access to *R*, which is not sharable, and *P1* is first granted access to *R*. Then when *P1* exits its critical section, either *P2* or *P3* may be allowed access to *R*. Assume that *R* is allocated to *P3* and *P1* requires access to *R* again. If the operating system alternately allocates *R* to *P1* and *P3*, then *P2* has to wait indefinitely and thus experience starvation.

Cooperation among processes by sharing

Besides competition, concurrent processes may also cooperate with each other. For example, multiple processes may have access to shared variables or files or databases. They may alter the content of these resources respectively, but they must cooperate to ensure the integrity of the shared data.

The above three control problems are also of concern in this case. The difference here is that data items may be accessed in two different modes, reading and writing. Later we will cover the classic Reader/Writer problem.

Cooperation among processes by communication

In the first two cases, processes interact with each other indirectly and are not explicitly aware of the others' existence. It is also possible for processes to cooperate directly by communicating with each other, say sending messages.

In this case, nothing is shared between processes, so mutual exclusion is not a control requirement. However deadlock and starvation remain. For example, there are two processes each providing a RPC service, and either of them also requires the other's service. If at some moment, both send a RPC request to the other and wait, then neither will receive the response since neither can move on to serve the other before its own request is served. Thus a circular waiting exists as well and so does a deadlock.

How to solve the above control problems involved in concurrency? As for deadlock and starvation, we will discuss them in the following chapter. We now move on to discuss mutual exclusion.

2 Mutual exclusion

Based on the above discussion and examples about mutual exclusion, it is clear that any facility or capability that is to provide support for mutual exclusion should meet the following requirements:

1. Only one process at a time is allowed into its critical section, among the processes that have critical sections for the same resource or shared object.

   To impose some form of control upon the execution of critical sections, we usually use the following programming structure:

   ```
   ...
   enter_critical();
   /* critical section */
   exit_critical();
   ...
   ```

   where *enter_critical()* and *exit_critical()* are provided by the system in some way and they together guarantee the exclusive access to shared resources.

2. When no process is in a critical section, any process that requests entry to its critical section must be permitted to enter without delay.

3. A process remains inside its critical section for a finite time only.

4. No assumptions should be made about relative process speeds or number of processors. That is the facility provided should be powerful enough to solve the control problems unconditionally, without any limit on the nature of concurrent processes.

There are many ways to satisfy these requirements. We may leave the responsibility with the processes that wish to execute concurrently. That's the user programs have to deal with these without support from programming language or the operating system. We refer to these as software approaches. Obviously application programmers have to take the responsibility and facility at the application level may lead to hight processing overhead. A second approach involves the use of special-purpose machine instructions. This reduces overhead but is not a general-purpose solution as shown later on. A third approach is to provide some kind of support within the operating system or programming languages.

2.1 Software approaches

Whether in a uniprocessor system or a multiprocessor one with shared main memory, it usually assumed that only one access to a memory location can be made at a time, so that the simplest operation, an assignment to a location, is always performed without interference.

2.1.1 Dekker's algorithm

First, let's discuss an algorithm designed by a Dutch mathematician Dekker step by step.

First attempt

Intuitively, based on the mutual exclusive access to a location of the main memory, we may use a location, which is represented as a variable in a programming language, to contain a flag to control the mutual exclusive execution of critical sections. Consider we have two processes, *P0* and *P1*, and a integer variable *turn*. If we use the different values of *turn* to indicate the permission for a specific process to enter its critical section, then the code given in Figure 3 (a) will be obtained.

Either process before entering its critical section must check the value of *turn*. If it is not allowed, then it has to wait. The mechanism of the *while* loop to check and wait is called **busy waiting**, which consumes much processor time unfortunately. Once a process gains access to its critical section, it may manipulate the shared resources; and when it exits, it must update the value of *turn* to allow the other process to enter.

This method has two drawbacks:

1. processes must strictly alternate in their use of their critical section.
2. if one process fails, the other process is permanently blocked, whether the failure happens in the critical section or outside of it.

Second attempt

The problem with the first attempt is that only one variable is used to indicate the states of both processes and the maintenance of its value relies on both processes. We should have state information about each process and maintained by themselves. Thus, an boolean array *flag* of 2 units may be used as below:

```
/* PROCESS 0 /*

•
•
while (turn != 0)
   /* do nothing */ ;
/* critical section*/;
turn = 1;
•
```

```
/* PROCESS 1 */

•
•
while (turn != 1)
   /* do nothing */;
/* critical section*/;
turn = 0;
•
```

(a) First attempt

```
/* PROCESS 0 */

•
•
while (flag[1])
   /* do nothing */;
flag[0] = true;
/*critical section*/;
flag[0] = false;
•
```

```
/* PROCESS 1 */

•
•
while (flag[0])
   /* do nothing */;
flag[1] = true;
/* critical section*/;
flag[1] = false;
•
```

(b) Second attempt

```
/* PROCESS 0 */

•
•
flag[0] = true;
while (flag[1])
   /* do nothing */;
/* critical section*/;
flag[0] = false;
•
```

```
/* PROCESS 1 */

•
•
flag[1] = true;
while (flag[0])
   /* do nothing */;
/* critical section*/;
flag[1] = false;
•
```

(c) Third attempt

```
/* PROCESS 0 */

•
•
flag[0] = true;
while (flag[1])
{
  flag[0] = false;
  /*delay */;
  flag[0] = true;
}
/*critical section*/;
flag[0] = false;
•
```

```
/* PROCESS 1 */

•
•
flag[1] = true;
while (flag[0])
{
   flag[1] = false;
   /*delay */;
   flag[1] = true;
}
/* critical section*/;
flag[1] = false;
•
```

(d) Fourth attempt

Figure 3: Mutual exclusion attempts

```
boolean flag[2] = {false, false};
```

Figure 3 (b) gives the algorithm.

Now even if one process fails outside the critical section, the other process may still enter its critical section; however if the failure happens inside the critical section, the active process will still be permanently blocked.

What makes the situation worse is that this algorithm does not even guarantee mutual exclusion. If *P0* just passed the *while* loop but have not set its flag to be *true*, then *P1* may be scheduled and thus passes the *while* loop check as well. Thus both processes enter their respective critical sections. The problem here is that the proposed solution is not independent of relative execution speed of processes.

Third attempt

Since the second attempt fails just because the flag of a process is flipped after its entering the critical section, how about flipping the flag before that? Thus we obtained the third attempt, illustrated in Figure 3 (c).

Now let us check if the mutual exclusion is guaranteed in this attempt. Suppose *P0* is going to enter its critical section. First it sets *flag[0]* to be *true*. If at this moment, *P1* is outside its critical section, then it has to wait until *P0* has entered and left its critical section; if *P1* is already in the critical section, then *P0* will be blocked and has to wait until *P1* has left its critical section. So the requirement of mutual exclusion is met.

Unfortunately, another problem is created. If both processes set their flags to *true* before either has passed the *while* loop, then either will think the other has entered its critical section, which is actually circular waiting, thus a deadlock occurs.

Fourth attempt

If we examine the third attempt, we may find out the reason why a deadlock may be caused is that either process, once setting its flags preventing the other from entering the critical section, will never back off from this position. So we may introduce some kind of *courtesy* here: each process sets its flag to indicate its desire to enter its critical section but is prepared to reset the flag to defer to the other process, as shown in Figure 3 (d).

This is not perfect yet. It is possible that both processes run at almost the same pace: both set the flags to be *true* first, then check them simultaneously, thus do the courtesy things, and later set the flags back to *true*. This indicates that the processes may both wait indefinitely and can never enter critical sections. This is similar to deadlock, but not exactly, since deadlock means a situation of no progress without external interference while the current subtle situation may disappear once the relative speeds of the processes change a little bit. We call the condition a *livelock*.

A correct solution

To avoid the above *livelock*, some facility should be used to assign different priorities to the processes so that they know who goes first. The variable *turn* in the first attempt may be used for this purpose, but in this case, it is used only for indicating an order when both processes desire to enter their critical sections. So the problem in the first attempt is not duplicated here. The solution is given as the left half of Figure 4.

2.1.2 Peterson's algorithm

Dekker's algorithm solves the mutual exclusion problem fully but is a little bit complex. Peterson later provided a simpler solution as the right half of Figure 4. The textbook presents a short analysis of this algorithm, please read it yourself.

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

void P0() {
  while (true) {
    flag [0] = true;
    while (flag [1])
      if (turn == 1) {
        flag [0] = false;
        while (turn == 1)
          /* do nothing */;
        flag [0] = true;
      }
    /* critical section */;
    turn = 1;
    flag [0] = false;
    /* remainder */;
  }
}

void P1() {
  while (true) {
    flag [1] = true;
    while (flag [0])
      if (turn == 0) {
        flag [1] = false;
        while (turn == 0)
          /* do nothing */;
        flag [1] = true;
      }
    /* critical section */;
    turn = 0;
    flag [1] = false;
    /* remainder */;
  }
}

void main () {
  flag [0] = false;
  flag [1] = false;
  turn = 1;
  parbegin (P0, P1);
}
```

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

void P0() {
  while (true) {
    flag [0] = true;
    turn = 1;
    while (flag [1] && turn == 1)
      /* do nothing */;
    /* critical section */;
    flag [0] = false;
    /* remainder */;
  }
}

void P1() {
  while (true) {
    flag [1] = true;
    turn = 0;
    while (flag [0] && turn == 0)
      /* do nothing */;
    /* critical section */;
    flag [1] = false;
    /* remainder */
  }
}

void main() {
  flag [0] = false;
  flag [1] = false;
  parbegin (P0, P1);
}
```

Figure 4: Dekker's algorithm and Peterson's algorithm

2.2 Hardware supports

2.2.1 Interrupt disabling

In a uniprocessor system, the fundamental reason of concurrency is that the processor may dispatch different processes to run from time to time. And the switching of control happens due to some kind of interrupt, such as timeout interrupt in a time-sharing environment, I/O operation completion interrupt, etc.. Based on this observation, if we could in some way avoid concurrency while the execution of critical sections is in progress, the mutual exclusion problem will then be solved naturally. Fortunately, the capabilities of disabling and enabling interrupts are usually provided in all kinds of computer systems in the form of primitives. Hence we obtain a hardware-based solution as below:

```
while (true) {
  /* disable interrupts */
  /* critical section */
  /* enable interrupts */
  /* remainder */
}
```

However this approach also eliminates the concurrency between irrelevant processes and thus the efficiency could be noticeably degraded. A second problem is that this approach does not work in a multiprocessor architecture.

2.2.2 Special machine instructions

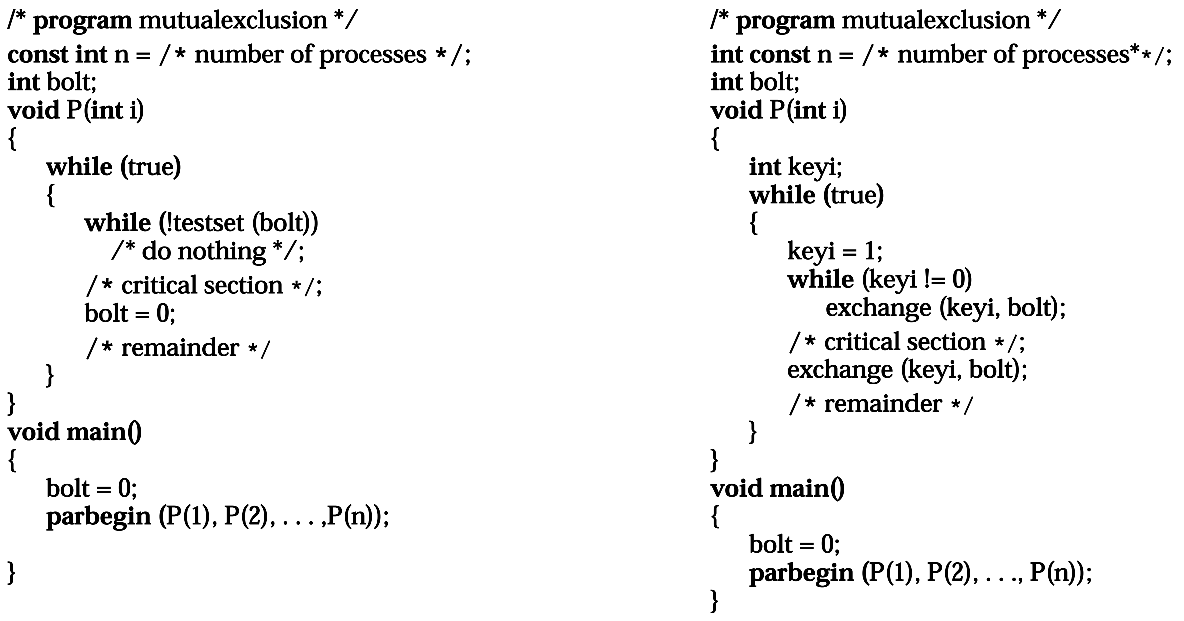
In a multiprocessor environment, the interrupt disabling approach does not work work any more; however it is assumed that the processors share access to a common main memory and at the hardware level, only one access to a memory location is permitted at a time. With this as a foundation, some computer processors designed several machine instructions that carry out two actions, such as reading and writing, of a single memory location. Since processes interleave at the instruction level, so such special instructions are atomic and are not subject to interference from other processes. Two of such kind of instructions are discussed in the following parts.

Test and Set instruction

The function of the test and set instruction may be presented as the following function:

```
boolean testset (int i) {
  if (i == 0) {
    i = 1;
    return true;
  } else {
    return false;
  }
}
```

where the variable *i* is used like a traffic light. If it is 0, meaning green, then the instruction sets it 1, i.e. red, and return *true*. Thus the current process is permitted to pass but the others are told to stop. On the other hand, if the light is already red, then the running process will receive *false* and realize not supposed to proceed. Accordingly, Figure 5 (a) shows a process that uses this instruction. Obviously at any



```
/* program mutualexclusion */
const int n = /* number of processes */;
int bolt;
void P(int i)
{
    while (true)
    {
        while (!testset (bolt))
          /* do nothing */;
        /* critical section */;
        bolt = 0;
        /* remainder */
    }
}
void main()
{
    bolt = 0;
    parbegin (P(1), P(2), . . . ,P(n));

}
```

(a) Test and set instruction

```
/* program mutualexclusion */
int const n = /* number of processes**/;
int bolt;
void P(int i)
{
    int keyi;
    while (true)
    {
        keyi = 1;
        while (keyi != 0)
            exchange (keyi, bolt);
        /* critical section */;
        exchange (keyi, bolt);
        /* remainder */
    }
}
void main()
{
    bolt = 0;
    parbegin (P(1), P(2), . . ., P(n));
}
```

(b) Exchange instruction

Figure 5: Hardware support for mutual exclusion

time, at most one process may enter its critical section and all others that desire to enter too go into a busy-waiting mode.

Exchange instruction

Similarly, the exchange instruction can be defined as follows:

```
void exchange (int register, int memory) {
  int temp;
  temp = memory;
  memory = register;
  register = temp;
}
```

This instruction exchanges the contents of a register with that of a memory location. During execution of the instruction, access to the memory location is blocked for any other instruction referencing that location.

Figure 5 (b) shows a solution based on this instruction. Note that the following expression always holds according to the initialization of variables and the nature of the exchange instruction:

$$bolt\ + \sum_{i} key_i = n$$

If $bolt = 0$, then no process is in its critical section; if $bolt = 1$, the exactly one process is in its critical section, and its key value is 0.

Properties of the machine-instruction approach

There are a number of advantages of this kind of approaches:

- It is simple and therefore easy to verify.
- It is applicable to any number of processes on either a single processor or multiple processors sharing main memory.

However there are also some serious disadvantages:

- Busy waiting is involved and thus processor time is wasted.
- Deadlock is possible.

Since the above approaches have such-and-such drawbacks, we need to look for other mechanisms, which will be discussed in the following class.