Lecture 4 - Synchronisation part 1

Gidon Rosalki

2025-04-20

Notice: If you find any mistakes, please open an issue at `https://github.com/robomarvin1501/notes_operating_systems`

1 Motivation

Let us consider two processes that want to print. Two processes write jobs to the spooler. This is a shared queue from which the printer extracts print jobs. We could do this as follows: NULL when there is no job to print, OUT indicates the next job to be printed, and IN is the end of the queue, and a new job is written there. Then to add a job we wimply do Spooler[IN] = job, and then perform IN++. Since IN and Spooler are shared amongst all processes, this can lead to race conditions, where two processes try and print at the same time:

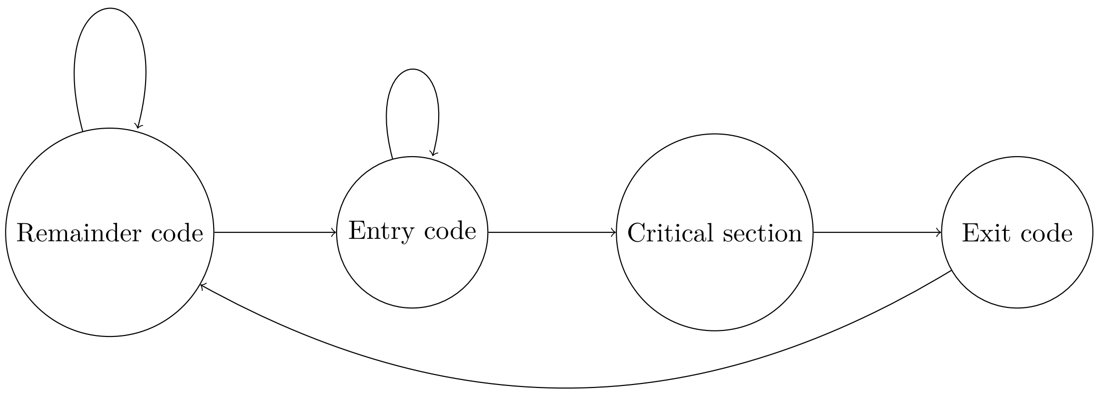
1. (1) Spooler[IN] = job 1
2. (2) Spooler[IN] = job 2
3. (1) IN++
4. (2) IN++

So in the end job 1 is lost, since it was overwritten by job 2, and there will be a null value between job 2 and IN, resulting in any new jobs added to the queue never being printed, because the queue was terminated earlier by the NULL.
We have further motivation for when there is shared data between many threads in a single process. If every thread can alter this data, then we can lose values in a similar manner to what was described above.

2 Solutions to the critical section

In short, the problem here is uncoordinated access to a shared resource. We will call where this happens the **critical section**, which is a piece of code that accesses the shared resource. This can be one or more instructions, and in order to be correct there must not be more than 1 critical section being executed by more than 1 process. A solution to this is using **mutual exclusion** algorithms. These avoid the simultaneous execution of a critical section. However, sometimes mutual exclusion is not enough, for example, when the order of execution matters. This will be discussed more later. Edsgar Dijkstra created a state machine to model this problem, and formalised the success criteria:

1. Mutual exclusion: No two processes are in the critical section simultaneously
2. Progress: If a process is trying to enter the critical section, then some process eventually enters the critical section.
3. Starvation freedom: A process trying to enter the critical section will eventually succeed
4. Generality: There are no assumptions about speeds or number of participants (number of threads, processors, processes, etc)
5. No blocking in the remainder: No process running outside its critical section may block other processes



Our first attempt to solve this is to simply disable interrupts. When we leave the entry code state, we disable all interrupts, and when we leave the exit code we re-enable all interrupts. This solves 1, 2, 3, and 5, but does not solve 4 if we have a multi processor system (where real parallelism exists). Additionally, user processes should not be allowed to disable interrupts, and disabling interrupts should only be done for very short periods of time.

Thread 1	Thread 2
Remainder	Remainder
while(turn == 1);	while(turn == 0);
Critical	Critical
turn = 1;	turn = 0;

Table 1: First attempt

Notice that the while loop has no body, so the thread does nothing while it waits for turn. This algorithm works for 2 threads/processes, it uses a single global variable named turn, which cannot be used in the remainder / critical, and has slightly different code for each thread. We can prove that mutual exclusion occurs: Assume that the 2 processes are in the critical condition, and then w.l.o.g thread 0 left its entry section (and entered the critical section). First at that point turn $\neq 1$, and so turn $= 0$. The only place that it can be changed to 1 is when thread 0 exits, and thus turn $= 0$ throughout the entire execution of the critical section by thread 0. Thus thread 1 is in its entry code when thread 0 is the critical section, and so we have a contradiction.
However there is still a problem here. Progress / no blocking in the remainder does not hold. Thread 1 can simply stay in its remainder, and thus never set turn back to 1, and thus thread 0 is unable to enter its critical section, and therefore starvation freedom also does not hold.

Thread 1	Thread 2
Remainder	Remainder

flag[0] = true;
while(flag[1]);
	flag[1] = true;
	while(flag[0]);

Critical	Critical
flag[0] = false;	flag[1] = false;

Table 2:

We can try and fix this problem by having 2 flags, one for each thread, where when they are true, the respective thread wants to enter the critical section. If the other thread wants to enter, then this threads waits until it exits, and changes its interest flag back to false. We may once again prove Mutual Exclusion:
This code holds No Blocking in the Remainder, a thread can only block another thread only if its flag is true, implying that it is either in its entry, critical, or exit section, but still Progress does not hold, which leads to starvation, and Freedom also does not hold. This can result in a **deadlock**. This is when there are two sections of code, both blocking the other from beginning, and both waiting on the other to unblock them to begin.
We can also prove mutual exclusion, similar to above: Assume that the two processes are in the critical section, and then w.l.o.g. thread 0 leaves its entry section, and enters the critical section. At that point `flag[0] = true`. Since the only place that `flag[0]` can be changed to false is when thread 0 exits the critical section, and thus this flag is true throughout the entire execution of the critical section by thread 0. From this, thread 1 is in its entry code when thread 0 is in the

critical section, which is a contradiction.

Perhaps we should try having a turn, and the flag bits, and hold while the other thread wants to being, and the turn?

Thread 1	Thread 2
Remainder	Remainder

turn = 1;
flag[0] = true;
while(flag[1] && turn == 1);
	turn = 0;
	flag[1] = true;
	while(flag[0] && turn == 0);

Critical	Critical
flag[0] = false;	flag[1] = false;

Table 3:

This does not work, since mutual exclusion does not hold.
We may fix this through Peterson's Algorithm (1981), which is created by a small change

Thread i
Remainder

flag[i] = true;
turn = 1 - i;
while(flag[1 - i] && turn == 1 - i);

Critical
flag[i] = false;

Table 4: Peterson

Mutual Exclusion: Assume that the two processes are in the critical section, and then w.l.o.g. thread 0 left its entry section, and entered the critical section first, at that point flag [1] = false or turn = 0.

1. If flag[1] = false, then thread 1 has not executed line 1 of its entry code, and when it eventually reaches line 2, then it will set turn to 0, and wait. The only place that turn is set to 1 is in thread 0's entry code, after it left the critical section, which is a contradiction

2. If turn = 0, then thread 1 executed line 2 between thread 0's line 2 and three, and thus a contradiction follows

It also holds no blocking in the remainder, and starvation freedom: Assume thread 1 tries to get into the critical section, but blocked (on line 3) for infinite time, then thread 0 enters the critical section an infinite number of times, and the second time thread 0 will execute the entry flag [0] = *true* turn = 1, and it will block. The next time that thread i gets the CPU, the `while` condition does not hold, and thread i enters the critical section, which is a contradiction.

However, this only solves for 2 threads, and cannot be easily extended to more. Furthermore, there is busy waiting (the while that does nothing). Additionally, compiler optimisations might change line order, and thus destroy the algorithm.

3 Recap

Race condition: The scheduling of processes and threads changes the final result. A different order results in different results, a typical scenario is that most of the time everything is OK, but sometimes not. This is very hard to debug.
Atomic instruction: An instruction that completes in a single step relative to other threads. For example, `x = x + 1` is **not** atomic. Read (lw) and write (sw) are atomic, however this does not always hold (if you have a wide word that spans memory addresses).
Busy waiting/spinning/busy looping: A technique in whicha process repeatedly checks to see if a condition is true. These should usually be avoided as they waste CPU cycles and result in a large overhead.

4 Lamport's bakery algorithm

At every step, when a thread wants to enter the critical section, it takes an incrementing number. A scheduler decides which number gets to be in the critical section at a given time. However, what if two threads take a number at the same time? Then currently both will take the same number, so we have failed mutual exclusion.

```
// Remainder
number[i] = 1 + max{number[j]} // Where j is in 1..n
for (j = 1; j < n; j++) {
    while(i != j && number[j] > 0 && number[j] < number[i]);
}
// Critical
number[i] = 0;
```

We may fix the 2 threads taking the same number as follows: We look for a thread who has the number that is less than equal.

```
// Remainder
number[i] = 1 + max{number[j]} // Where j is in 1..n
for (j = 1; j < n; j++) {
    while(i != j && number[j] > 0 && number[j] <= number[i]);
}
// Critical
number[i] = 0;
```

However, this then results in a deadlock.

We may instead use lexicographical order

```
// Remainder
number[i] = 1 + max{number[j]} // Where j is in 1..n
for (j = 1; j < n; j++) {
    while(i != j && number[j] > 0 && (number[j], j) < (number[i], i));
}
// Critical
number[i] = 0;
```

However, we then have a problem with mutual exclusion. Should threads 4 and 5 try and take the same number, then we will give both the same number, let us suppose that there is then a context switch, resulting in 5 entering the critical section. Then when we switch back to 4, it will see that they have the same number, but it is before 5 lexicographically, and will thus also enter the critical section.

There is a correct implementation here:

```
// Remainder
choosing[i] = true;
number[i] = 1 + max{number[j]} // Where j is in 1..n
choosing[i] = false;
for (j = 1; j < n; j++) {
    while(choosing[j]);
    while(i != j && number[j] > 0 && (number[j], j) < (number[i], i));
}
// Critical
number[i] = 0;
```

This also ensures First In First Out. If a process i is waiting, and thread j has not yet started the entry, then j will not enter the critical section before i.

4.1 Read-Modify-Write instructions

So far we assumed that read (load) and write (store) are atomic instructions. However, there might be a context switch between a read and its corresponding write. Modern CPUs support in hardware some RMW instructions, and they are thus atomic:

- Test&Set(&Lock) `i = *lock; *lock = 1; return i;`
- Fetch&Add(&p,inc) `val = *p; *p = val + inc; return val;`
- Compare&Swap(&p,old,new) `if (*p != old) return false; *p = new; return true;`

With this we can solve mutual exclusion with Test&Set, since it is atomic:

```
// Remainder
while(test&set(lock));
```

```
// Critical
lock = 0;
```

However, Starvation Freedom does not hold. The same thread can enter over an over again, leaving the second outside the critical section.

4.2 Burns' algorithm

```
// Remainder
waiting[i] = true;
key[i] = 1;
while(waiting[i] && key[i]) {
   key[i] = test&set(lock);
}
waiting[i] = false;
// Critical
j = (i+1) % n;
while(j != i && !waiting[j]) {
   j = j + 1 % n ;
}
if(j != i) {
   waiting[j] = false;
} else {
   lock = 0;
}
```

Mutual exclusion occurs since to enter a critical section one needs either waiting = false, or key = 0. If another thread is in the critical section, then lock = 1, and so key = 1. Another thread must "let us in". This can only be done by the thread leaving the critical section. We use Test&Set, since without it two threads could read that lock = 0, set lock = 1, and then proceed to the critical section together.
This also uses round robin lock passing, so since other threads can enter at most $n - 1$ times, and after that the waiting thread is given priority, this holds starvation freedom.

We can also use **Synchronisation Primitives**. We define abstract data types that are used to provide synchronisation. They have a clear interface to the data type, which cannot be accessed in any other way. These are usually provided by the programming languages, with perhaps some assistance from the OS.
Our first example is the semaphore. It records with two fields, value and list. There are also two operations:

```
Down(s)
S.value = S.value - 1
if S.value < 0 {
   Add this thread to S.L
   sleep();
}
```

```
Up(s)
S.value = S.value + 1
if S.value <= 0 {
   Remove thread T from S.L
   wakeup(T);
}
```

The operations are executed atomically. This doesn't make sense since the implementation is orthogonal to the definition, however, the motivation here is to avoid busy waiting.