Processes – 2
The MUTEX Problem

Critical Section
Basic Process Coordination
Coordination Problems

The critical section (CS) problem

Recall that a process is defined as a program that has started and not yet terminated.

A program is a sequence of machine instructions that are executed sequentially.

While we may assume that an individual machine instruction is atomic, a process may be interrupted after any one instruction

Definition:
Any code segment that accesses a shared data area is referred to as Critical Section (CS)!

Shared data areas include:
- Memory Locations
- Files
- I/O devices
- Other exclusive resources

So what is so critical about this section?? Why do we care??

..CS example

Consider the following example from the literature:

```
cobegin
p1:   ...
x = x+1;
...
||
p2:   ...
x = x+1;
...
coend
```

The corresponding machine instructions are:
1. load the value of x into internal register R (R=x)
2. increment R (R = R+1)
3. store the new value into memory location x (x = R)

Processes p1 and p2 will execute the following sequence:

pi: Ri = x; Ri = Ri+1; x = Ri;

Note: pi can be interrupted after any one of the instructions.

This may result in different execution sequences

...CS example

Consider the following 3 execution sequences and determine the value of x:

Sequence #1	p1: R1 = x; R1 = R1 + 1; x = R1;	
	p2:	R2 = x; R2 = R2 + 1; x = R2;
Sequence #2	p1:	R1 = x; R1 = R1 + 1; x = R1;
	P2: R2 = x; R2 = R2 + 1; x = R2;	
Sequence #3	p1: R1 = x;	R1 = R1 + 1; x = R1;
	p2: R2 = x; R2 = R2 + 1; x = R2;	

How can we prevent from multiple processes being active in the CS??

Towards a solution…

- Note that processes can be interrupted in the most inconvenient situations:
  - While evaluating the condition(s) in any programming construct:
    - while(cond)
    - for(…)
    - until(..)
    - if (….) then
  - While inserting or removing an item from a data structure:
    - Tree
    - Linked list
    - Heap
- The solution to the CS problem is to allow at most one process to be active in the CS → Mutual Exclusion
- On way to accomplish this is to disable interrupts before entering the CS.
  - What if there are multiple CPUs?
  - What if processes have multiple (distinct) critical sections?
  - What if the CS is large?

Mutual Exclusion

Any solution to guarantee mutual exclusion must limit the no. of processes in the CS to 1.

Mutual Blocking must be prevented!

We can/must distinguish the four types of blocking:

1. A process that is currently not executing in its CS must not prevent other processes to enter the CS (progress)
2. A process must not repeatedly enter its CS, thereby preventing other processes to enter the CS (starvation)
3. Two processes that are about to enter their CS must not block each other indefinitely (deadlock)
4. Two processes about to enter their CS must not repeatedly yield to each other indefinitely (livelock)

Software solutions

The following are a number of "different" attempts to provide mutual exclusion for 2 processes! We need to evaluate each of them to understand what works and what doesn't!

Inspect each solution for all 4 types of blocking!

```
\\Mutual Exclusion is easy!!!

int in_CS = 0;   \\ global
cobegin
  p1:   while(1){
     while(in_CS);
     in_CS = 1;
     DO-CS;
     in_CS = 0;
            prog1-outside CS }


  p2:  while(1){

while(in_CS);
in_CS = 1;
DO-CS;
in_CS = 0;
prog1-outside CS }
coend
```

?

…from the literature …..

```
// Try #1
int turn = 1;
cobegin
p1:   while(1) {
  while(turn == 2); //busy wait
  DO-CS;
  turn = 2;
  prog1 outside CS;
}
//
p2:   while(1) {
  while(turn == 1); //busy wait
  DO-CS;
  turn = 1;
  prog2 outside CS;
}
```

Issues: Mutex? Blocking? Startvation?

DISCUSS!!

```
// Try #2
int c1 = 0, c2 = 0;
cobegin
p1:   while(1) {
  c1 = 1;
  while(c2); //busy wait
  DO-CS;
  c1 = 0;
  prog1 outside CS;
}
//
p2:    … // analogous tp p1 //
```

Issues: Mutex? Blocking? Startvation?

DISCUSS!!

```
// Try #3
int c1 = 0, c2 = 0;

cobegin
p1:   while(1) {
  c1 = 1;
  if (c2) c1 = 0; //busy wait
    else {
      DO-CS;
      c1 = 0;
      prog1 outside CS;
    }
}
//
```

```
p2:         while(1) {
  c2 = 1;
  if (c1) c2 = 0; //busy wait
    else {
      DO-CS;
      c2 = 0;
      prog1 outside CS;
    }
}

coend;
```

Issues: Mutex? Blocking? Startvation?

DISCUSS!!

Note that we cannot predict the exact execution timing between p1 and p2!!

…a working solution

```
// Peterson Solution
int c1 = 0, c2 = 0, will_wait;
cobegin
p1:   while(1) {
  c1 = 1;
  will_wait = 1;
  while(c2 && (will_wait ==1); //loop
  DO-CS; c1 = 0;
  prog1 outside CS;
}

//

p2:   while(1) {
  c2 = 1;
  will_wait = 2;
  while(c1 && (will_wait ==2); //loop
  DO-CS;  c2 = 0;
  prog1 outside CS;
}
coend;
```

- Process p_i sets flag c_i to indicate the intent to enter the CS.
- *Variable will_wait* breaks possible race conditions.
- p_i setting *will_wait* to its *pid* announces to the other process that it is willing to wait if both processes happen to attempt to enter the CS at the same time.
- The solution guarantees mutual exclusion and prevents all forms of blocking.
- Formal Proof??

..notes on SW-based mutex

Software solutions have several drawbacks:

1. Solutions a often difficult to understand and verify.
1. Extension to more than 2 processes is generally difficult.
1. SW-based mutex solutions apply to competition problems, not coordination/cooperation among processes.
1. Busy waiting results in the utilization of the CPU by a waiting process without resulting in any computational progress.

- Proving the correctness of a mutex solution is not easy.
- In general, a good first approach is to prove by contradiction.
  - Assume mutex is violated
  - Show that reasoning leads to a contradiction.
- Sometimes it is easier to prove the contrapositive:
  - A→B (implication)
  - ~B →~A (contrapositive)

Test & Set → Semaphores

The crux of the problem to provide mutual exclusion is the unpredictable nature of processes. → when does a process get interrupted and is preempted??

One solution provided by most CPUs is the test_and_set instruction. There are various versions of TS in the literature.

The general format is:

test_and_set(x)

Semantics:
x is a boolean variable, initialized to *0;*

test_and_set(x) returns the value of x and sets x to *1.*

```
boolean test_and_set(boolean x)
 {
   test_and_set := x;
   x := 1;
   return;
 }
```

test_and_set is executed in a single instruction cycle and cannot be interrupted.

…test_and_set example

note: there is an alternate form of TS in the book.

So, how does test_and_set help in providing mutual exclusion to a critical section?

Remember, TS is an indivisible or atomic operation provided by the CPU.

Hence, we can use TS to create a spin lock around a critical section!

```
Example test&set():

boolean lock := 0;
pi:
  while(1){
     while test_and_set(lock);
     process CS
     lock := 0;
     remaining code outside CS;
  }
```

More on spin locks later!!

Semaphores

A semaphore S is a data-structure that is maintained by the operating system.

Operation P and V on semaphores are indivisible or atomic.

How can the OS (i.e., software) provide atomic operations??

Guess!! Yes, YOU in the left corner…..

```
type semaphore…

typdef struct{
int count;
list_of_processes queue;

} semaphore;
```

```
P(s):     s.count --;
if s.count < 0 then block(s)

where block(s) places the process on
s.queue and invokes the process
scheduler.
```

```
V(s):     s.count ++;
if s.count <= 0 then wakeup(s)

where wakeup(s) removes a process from
s.queue and places it into the ready list.
```


Semaphore facts

- Semaphores where introduced by Dijkstra, 1968
- Operations P and V are acronyms for the Dutch words for:
  - P→ to test → *proberen*
  - V→ to signal → *verhogen*
- If several processes invoke P or V operations on the same semaphore, the operation will occur sequentially in arbitrary order.
- If more than one process is queued (inside a P operation) on the same semaphore, it is generally non-deterministic which process is selected upon the execution of V(s).
- However, this depends truly on the implementation of the semaphore and the scheduling discipline (or queuing discipline)
- Semaphores are maintained by the OS and are considered a system resource.