Lecture 9: Synchronization

- Concurrency examples and the need for synchronization
- definition of mutual exclusion (MX)
- programming solutions for MX
  - MX algorithm for 2 processes (Peterson's algorithm)
  - MX algorithm for multiple processes (bakery algorithm)

"Too much milk" example

Time	You	Your Roommate
3:00	Arrive home	
3:05	Look in fridge, no milk	
3:10	Leave for grocery	
3:15		Arrive home
3:20	Arrive at grocery	Look in fridge, no milk
3:25	Buy milk, leave	Leave for grocery
3:30		
3:35	Arrive home	Arrive at grocery
3:36	Put milk in fridge	
3:40		Buy milk, leave
3:45		
3:50		Arrive home
3:51		Put milk in fridge
3:51	Oh, no! **Too much milk!!**	

- The problem here is that the lines:

  Look in fridge, no milk through "*Put milk in fridge*

  are not an **atomic** operation - an atomic operation cannot be overlapped by another atomic operation

"running contest" example

Thread A	Thread B
i = 0	i = 0
while (i < 10)	while (i > –10)
i = i + 1	i = i – 1
print "A wins"	print "B wins"

- Assumptions:
  - Memory load and store are atomic
  - Increment and decrement at not atomic
- Questions:
  - Who wins?
  - Is it guaranteed that someone wins?
  - What if both threads have their own CPU, running concurrently at exactly the same speed? Is it guaranteed that it goes on forever?
  - What if they are sharing a CPU?

Synchronization problem of concurrency

- *Concurrency* - simultaneous execution of multiple threads of control
- concurrency appears in:
  - *mutiprogramming* - management of multiple processes on a uniprocessor architecture
  - *mutliprocessing* - management of multiple processes on a multiprocessor architecture
  - *distributed computing* - management of multiple processes on multiple independent computers
- *Synchronization* — using *atomic* (indivisible) operations to ensure cooperation between threads
- race condition - the outcome of the multiple process execution depends on the order of the execution of the instructions of the processes

Mutual exclusion (MX)

- To avoid race conditions mutual exclusion is used:
- *Mutual exclusion* (MX) — ensures that only one thread does a particular activity at a time — all other threads are *excluded* from doing that activity
- *Critical section* (CS)— code that only one thread can execute at a time (e.g., code that modifies shared data)
- threads alternate between executing CS and non-CS. A thread may execute non-CS code indefinitely but it spends only finite amount of time within CS
- Solution must be *fair,* that is:
  - Avoid *starvation* — if a thread starts trying to gain access to the critical section, then it should eventually succeed
- Methods to enforce mutual exclusion
  - Up to user — threads have to explicitly coordinate with each other
  - Up to OS — OS provides support for mutual exclusion
  - Up to hardware — hardware provides architectural support for mutual exclusion

MX algorithm 1

```
t1 ( )   {
   while (true)   {
      while (turn != 1)
         ;   /* do nothing */
      CS
      turn = 2;
      non-CS
   }
}

t2 ( )   {
   while (true)   {
      while (turn != 2)
         ;  /* do nothing */
      CS
      turn = 1;
      non-CS
   }
}
```

- Threads take turns entering CS.
  - t1 checks if it's her turn. It it is not (`turn!=1`) then it waits doing nothing
  - if it is `t1`'s turn it proceeds with it's CS and setting turn to 2 giving `t2` an opportunity to enter her CS
- advantages:
  - enforces MX
- problems - unfair:
  - processes must alternate in CS requests, what if they are executing with different speeds? What if t2 does not want to get into CS for a while?

MX algorithm 2a

```
t1 ( ) {
   while (true) {
      while (t2_in_CS == true)
          ;   /* do nothing */
      t1_in_CS = true;
      CS
      t1_in_CS = false;
      non-CS
   }
}

t2 ( ) {
   while (true) {
      while (t1_in_CS == true)
          ;   /* do nothing */
      t2_in_CS = true;
      CS
      t2_in_CS = false;
      non-CS
   }
}
```

- Before entering CS thread checks if the other is in CS
  - **t1_in_CS,t2_in_CS** indicate if corresponding thread is in CS
- advantage - fair
- problem - no MX!
- After **t1** decides that **t2** is not in CS it is already in CS. Yet it takes time for **t1** to set its own flag **t1_in_CS** to **true**

MX algorithm 2b

```
t1 ( ) {
   while (true) {
      t1_in_CS = true;
      while (t2_in_CS == true)
          ;    /* do nothing */
      CS
      t1_in_CS = false;
      non-CS
   }
}
t2 ( ) {
   while (true) {
      t2_in_CS = true;
      while (t1_in_CS == true)
          ;      /* do nothing */
      CS
      t2_in_CS = false;
      non-CS
   }
}
```

- Let's move the setting of the **t1_in_CS** flag outside of the neighbor's flag checking
- advantage - MX
- problem - not fair!
  - Both threads cannot get to CS if they set their flags at the same time!
  - This condition is called *deadlock* and we say that threads may *starve*

MX algorithm 3 (Peterson's alg.)

```
t1 ( ) {
   while (true) {
      t1_in_CS = true;
      turn = 2;
      while (t2_in_CS == true &&
             turn == 2)
          ;  /* do nothing */
      CS
      t1_in_CS = false;
      non-CS
   }
}
t2 ( ) {
   while (true) {
      t2_in_CS = true;
      turn = 1;
      while (t1_in_CS == true &&
             turn == 1)
          ;  /* do nothing */
      CS
      t2_in_CS = false;
      non-CS
   }
}
```

- Let's join algs 1 and 2
- condition of **while** of **t1** does NOT hold (**t1** is allowed to proceed to CS) if:
  - **t2** is out of CS (**t2_in_CS != true**)
  - **t1** is blocked on its **while (turn == 1)**: **t2** set **turn** to 1 after **t2** executed: **turn = 2**
- Peterson's alg. enforces MX
- Peterson's alg. avoids deadlock: **t1** cannot be blocked when:
- **t2** is in ***non-CS***: **t2_in_CS == false**
- **t2** is in **while**: **turn == 1**

Multiple Process MX (bakery alg.)

```
int i; /* unique process id */
while (true)  {
   choosing[i]=true;
   number[i]=max(number[0],…,
             number[n-1])+1;
   choosing[i]=false;
   for (j=0; j<n; j++) {
      while (choosing[j])
          ;   /* do nothing */
      while (number[j]>0 &&
               (number[j]<number[i] ||
                number[j]==number[i] &&
                j<i))
          ;   /do nothing */
   }
   CS
   number[i] = 0;
   non-CS
}
```

- Bakery alg. is a generalization of Peterson's alg. for **n** processes.
- Each process has unique identifier
- the process takes a number when CS is needed; in case of a tie process ids are used
- the process with the lowest number is allowed to proceed
- why does it check that **choosing** is not set when trying to access CS?