Notes for Lecture 12

[redacted]

Lecture 12 Process Synchronization

12.1 Mutex Lock

Recall the mutual exclusion problem introduced in the last lecture, one of the most basic and fundamental process synchronization problems. There are many approaches to solve the mutual exclusion problem, with different degrees of sophistication and performance overheads. We will study mutual exclusion using (i) mutual exclusion locks (or simply mutex locks), (ii) condition variables, and (iii) semaphores. We will also see how these can be used for other kinds of synchronization later. A mutex lock, after being initialized, is accessed using one of only two functions: (i) lock and (ii) unlock. It can be used to solve the mutual exclusion problem as shown by the following figure:

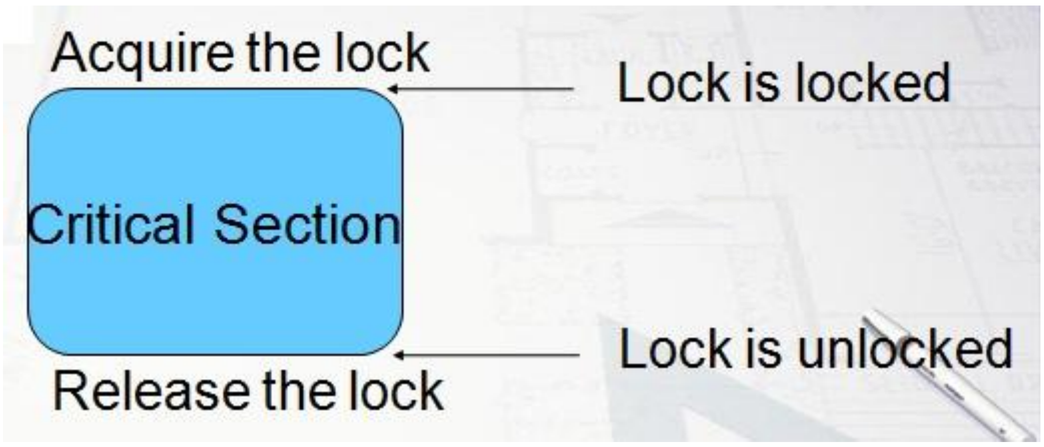


Figure 12.1 - Using a lock to solve the mutual exclusion problem. Once the thread acquires the lock, it will have mutually exclusive access to its critical section. Till it unlocks this lock, any other thread will have to wait to acquire the lock. Lock() is the entry section, while unlock() is the exit section here.

12.2 Implementing mutex locks by disabling interrupts

On uniprocessor systems, simplest solution to achieve mutex lock is to disable interrupts during a process's critical section. This will prevent any interruption while process is in critical section. However, this solution has many drawbacks, limiting its generality:

(i) certain interrupts may be unmaskable, implying if one of these occurs, the critical section's execution would be interrupted,

(ii) if the critical section code causes a trap, there would be a switch to a trap handler, again breaking the flow of execution of the critical section as one special case of this. If a process halts during its critical section, control will never be returned to another process, effectively halting the entire system.

(iii) (not discussed in class, [redacted]) if a critical section is long, then the system clock will drift every time a critical section is executed since the timer interrupt is no longer serviced, so tracking time would be impossible during the critical section.

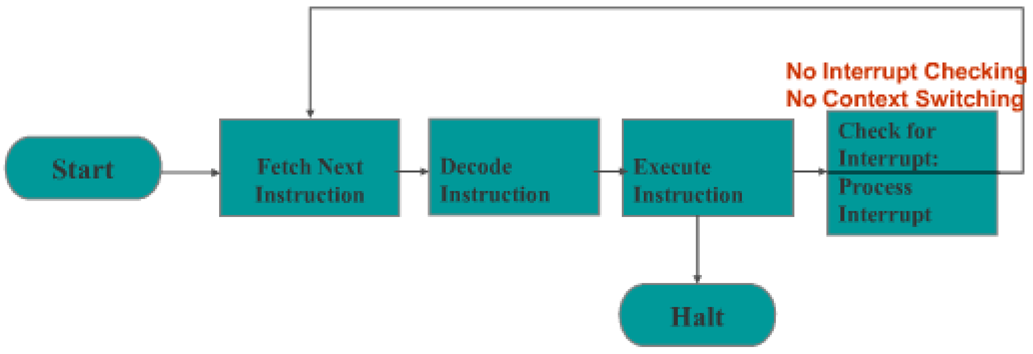
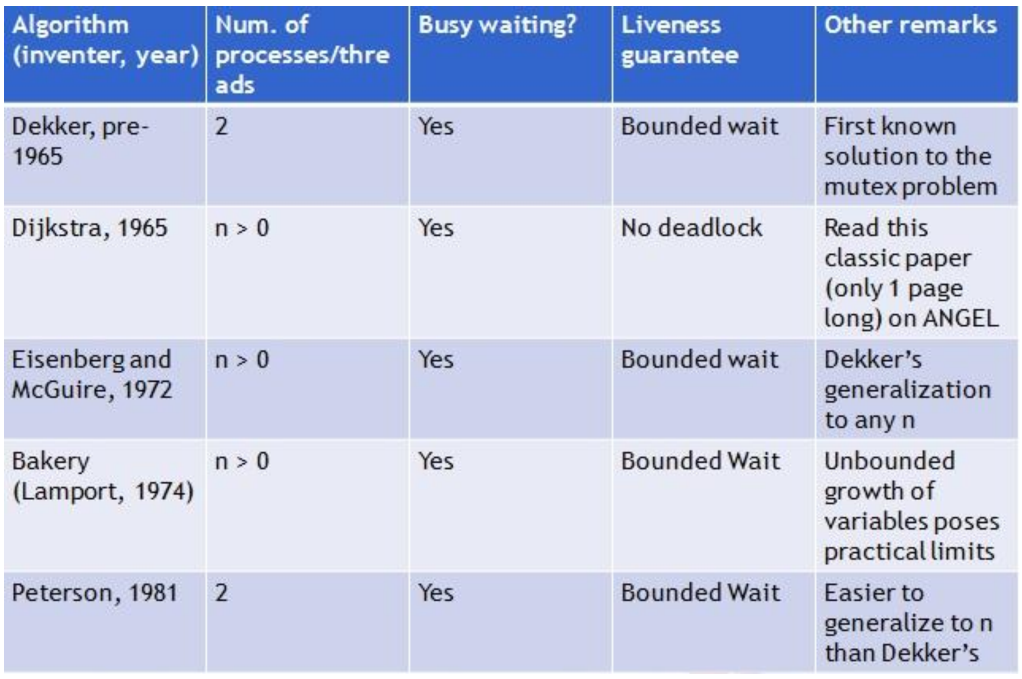


Figure 12.2 – A flowchart showing how checking of interrupts is done by the CPU as part of each instruction's execution. Upon the occurrence of an interrupt, it is processed. When interrupts are masked, this checking is disabled

12.3 Solutions using only atomic load and store operations

There's various algorithms exists to solve the mutex lock. The below table shows various algorithms that provides solution for mutual exclusion.



Algorithm (inventer, year)	Num. of processes/threads	Busy waiting?	Liveness guarantee	Other remarks
Dekker, pre-1965	2	Yes	Bounded wait	First known solution to the mutex problem
Dijkstra, 1965	n > 0	Yes	No deadlock	Read this classic paper (only 1 page long) on ANGEL
Eisenberg and McGuire, 1972	n > 0	Yes	Bounded wait	Dekker's generalization to any n
Bakery (Lamport, 1974)	n > 0	Yes	Bounded Wait	Unbounded growth of variables poses practical limits
Peterson, 1981	2	Yes	Bounded Wait	Easier to generalize to n than Dekker's

Figure 12.3 – Type of Mutual Exclusion and brief explanation

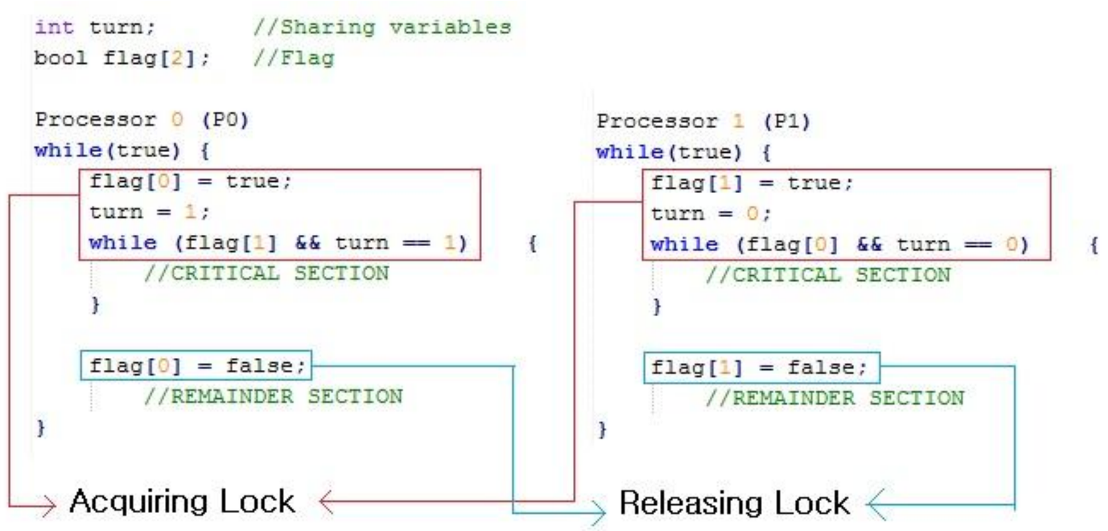
In this class, we will study one such algorithm, namely the Peterson's algorithm from this table.

12.3.1 Peterson's Algorithm

- Formulated by Gary L. Peterson in 1981
- Peterson's solution is a concurrent programming algorithm for mutex lock that allows two processes to share a single-use resource without conflict, using only shared memory for communication.
- Peterson's original formulation worked with only two processes(threads), the algorithm can be generalized for more than two

Here's how Peterson's Solution works

- Assume that the Load and Store instructions are atomic (cannot be interrupted)
- The two processes share two variables (int turn, bool flag[2])
- Variable turn indicates whose turn it is to enter the critical section
- The flag array is used to indicate if process is ready to enter the critical section.
- flag[i] = true implies that Processor i is ready to enter the Critical Section



```
int turn;        //Sharing variables
bool flag[2];    //Flag

Processor 0 (P0)
while(true) {
    flag[0] = true;
    turn = 1;
    while (flag[1] && turn == 1)    {
        //CRITICAL SECTION
    }

    flag[0] = false;
        //REMAINDER SECTION
}
```

```
Processor 1 (P1)
while(true) {
    flag[1] = true;
    turn = 0;
    while (flag[0] && turn == 0)    {
        //CRITICAL SECTION
    }

    flag[1] = false;
        //REMAINDER SECTION
}
```

Figure 12.4 – Pseudo code of implementing Peterson's solution

12.3.2 Proof that Peterson's algorithm provides mutual exclusion

Statement : Prove that P0 and P1 can never be in their critical sections together in same time.

Proof: Assume that P0 has entered its critical section

When Processor 1 is in its critical section, either

- flag[1] == false

  P1 has left its critical section

- turn == 0

  P1 is in its entry section but has let P0 "go ahead at its own coast" and unable to proceed beyond its entry section till P0 comes out of its critical section

12.3.3 Proof that Peterson's algorithm provides bounded wait

Lets consider that P0 is stuck in its entry section.

That is, (flag[1] == true) and (turn == 1)

Suppose P1 comes out of its critical section and sets flag[1] to false (flag[1] = false). Then there are two cases.

Case 1: P0 gets to check flag[1] before P1 has a chance to set it back to true. If flag[1] is false, P0 enters to the critical section.

Case 2: P1 "races ahead" and does flag[1] = true. Then

→ P0 will continue to be stuck in its while loop if it is context switched in

→ Eventually, P1 will execute turn = 0

→ Now, P1 will get stuck in its while loop

Since flag[0] == true and turn == 0

→ Eventually, P0 gets to run and finds that while() condition is false

Since flag[1] == true, but turn == 0

12.4 Solutions using sophisticated atomic instructions

12.4.1 Test and Set Lock instruction

Test and set instruction is an instruction used to write to a memory location and return its old value as a single atomic(non-interruptable) operation.

If multiple processes may access the same memory, and if a process is currently performing a test-and-set instruction, no other process may begin another test-and-set until the first process is done.

Figure 12.6 shows that threads are sharing one Boolean value, lock.

```
bool TestAndSet (bool *target)
{
    bool rv = *target;
    *target = true;
    return rv:
}
```

```
bool lock = false;  //shared variable
void thread()
{
    while (true) {
        while ( TestAndSet (&lock ))    {
            // Do nothing
        }
        //CRITICAL SECTION

        lock = false;
        //REMAINDER SECTION
    }
}
```

Figure 12.5 – Pseudo code of Mutex lock implementing Test-and-set instruction. Important: Although the code shown on the left appears to have many instructions, a Test-and-Set instruction would achieve all of this as a single atomic operation

12.8.2 Swap instruction

Swap instruction is an atomic CPU instruction used in multithreading to achieve synchronization. It swaps the contents of two specified memory locations.

Figure 12.7 shows that threads are sharing Boolean value lock, and each thread has local Boolean value, key.

```
void Swap(bool *a, bool *b)
{
    bool temp = *a;
    *a = *b;
    *b = temp;
}
```

```
bool lock = false;  //shared variable

void thread() {
    bool key = false;   //local variable for each process
    while(true) {
        key = true;
        while(key == true)  {
            Swap(&lock, &key);
            //CRITICAL SECTION
        }
        lock = false;
        //REMAINDER SECTION
    }
}
```

Figure 12.6 – Pseudo code of Mutex lock implementing Swap instruction

12.5 Shortcomings of locks

One key shortcoming of locks is that they cause busy waiting. This can lead to the "Priority inversion" wherein a higher priority task is indirectly preempted by a lower priority task that happens to have acquired a lock which the higher priority task needs to acquire to enter its own critical section.

Assume that there's P0 and P1, sharing same resources. P1 has higher priority than P0.

Now think that P1 has to wait until P0 unlock the lock. When certain time come, scheduler will schedule P1 for next. The problem arises if P0 did not finish the work, still using resource. Then P1 just stuck in the while loop until P0 release the resource and unlock the lock.

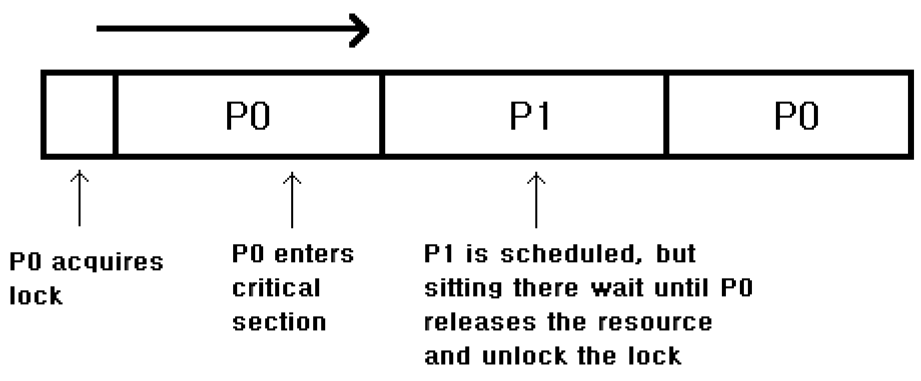
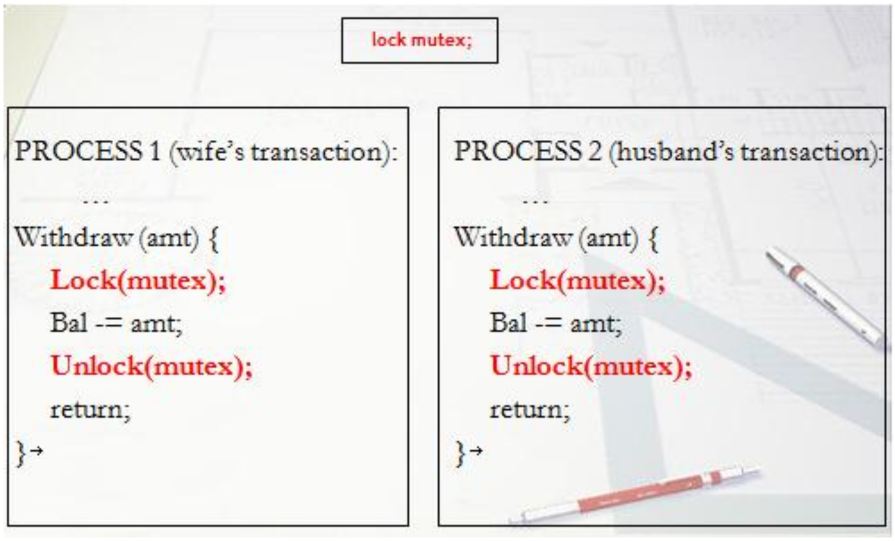

P0
P1
P0
P0 acquires lock
P0 enters critical section
P1 is scheduled, but sitting there wait until P0 releases the resource and unlock the lock

Figure 12.7 – Example diagram of Priority Inversion

12.6 Examples of Solving Synchronization Problems using Locks

12.6.1 Husband and Wife’s bank transaction.



```
lock mutex;

PROCESS 1 (wife’s transaction):
    ...
Withdraw (amt) {
    Lock(mutex);
    Bal -= amt;
    Unlock(mutex);
    return;
}

PROCESS 2 (husband’s transaction):
    ...
Withdraw (amt) {
    Lock(mutex);
    Bal -= amt;
    Unlock(mutex);
    return;
}
```

Figure 12.8 – Pseudo code of wife and husband’s transaction by using mutex lock

12.6.2 Producer and Consumer

→ To think: How can we safely increment and decrement ‘count’?

```
#define BUFFER_SIZE n

//Shared variables
lock mutex;
int buffer[BUFFER_SIZE];
int count;

void Producer()
{
    int in, out;
    int nextProduced = 0;

    while(true) {
        while(count == BUFFER_SIZE) {
            //Do nothing
        }

        buffer[in] = nextProduced;
        in = (in + 1) % BUFFER_SIZE;
        lock(mutex);
        count++;
        unlock(mutex);
    }
}

void Consumer()
{
    int in, out;
    int nextConsumed = 0;

    while(true) {
        while(count == 0)    {
            //Do nothing
        }

        nextConsumed = buffer[out];
        out = (out + 1) % BUFFER_SIZE;
        lock(mutex);
        count--;
        unlock(mutex);
    }
}
```

Figure 12.9 – Pseudo code of ‘producer and consumer’ using mutex lock