CSCI 4730/6730 OS
(Chap #6 Synchronization Tools)

In Kee Kim

Department of Computer Science

University of Georgia

Where are we

- We studied
  - OS Structure – Kernel
  - Process
  - Threads and Concurrency
  - CPU Scheduling

- Which one was the most interesting?
- Which one was the least interesting?

In Chapter 5

- Uni-processor (single CPU, single Core) scheduling
  - FCFS, SJF, SRTF, RR, Priority, MLQ, MLFQ
- Multi-processor scheduling
  - SMP, AMP, CMT, NUMA
  - HMP – ARM Big.LITTLE
- Real-time Scheduling
  - Rate-Monolithic, EDF
- Linux Scheduling
  - Completely Fair Scheduler

Chapter 6: Synchronization Tools

- Background and Motivation
- The Critical-Section Problem
- Peterson’s Solution
- Hardware Support for Synchronization
- Mutex Locks
- Semaphores
- Liveness

Chapter 6, 7, and 8 will cover synchronization problems.

Synchronization Motivation

```
int counter = 0;
void *mythread(void *arg)
{
    int i;
    for (i=0; i<10000; i++){
        counter++;
    }
    return NULL;
}

int main(int argc, char *argv[])
{
    pthread_t p1;
    printf("main: begin (counter = %d)\n", counter);
    pthread_create(&p1, NULL, mythread, "A");

    // join waits for the threads to finish
    pthread_join(p1, NULL);
    printf("main: done (counter = %d)\n", counter);
    return 0;
}
```

Implementation of `counter++`

- `counter++` could be implemented as

```
register = counter          (mov &counter, %reg1)
register = register + 1 (add 0x1, %reg1)
counter  = register         (mov %reg1, &counter)
```

```
load    &counter, %reg
add     0x1, %reg
store   %reg, &counter
```

Let’s consider two threads executions

- What happens in your processor?

- Let’s change the source code a little bit

Let’s consider two threads executions

- What happens in your processor?

Initially, `count = 0`

S0: T1 executes	`register1 = counter`	{register1 = 0}
S1: T1 executes	`register1 = register1 + 1`	{register1 = 1}
S2: T2 executes	`register2 = counter`	{register2 = 0}
S3: T2 executes	`register2 = register2 + 1`	{register2 = 1}
S4: T1 executes	`counter = register1`	{counter = 1 }
S5: T2 executes	`counter = register2`	{counter = 1}

Let's consider two threads executions

S0: T1 executes	`register1 = counter`	{register1 = 0}
S1: T1 executes	`register1 = register1 + 1`	{register1 = 1}
S2: T2 executes	`register2 = counter`	{register2 = 0}
S3: T2 executes	`register2 = register2 + 1`	{register2 = 1}
S4: T1 executes	`counter = register1`	{counter = 1 }
S5: T2 executes	`counter = register2`	{counter = 1}

Time

Thread 1		Thread 2	
load	&counter, %reg1		
add	0x1, %reg1		
______________		load	&counter, %reg2
______________		add	0x1, %reg2
store	%reg1, &counter	______________	
		store	%reg2, &counter

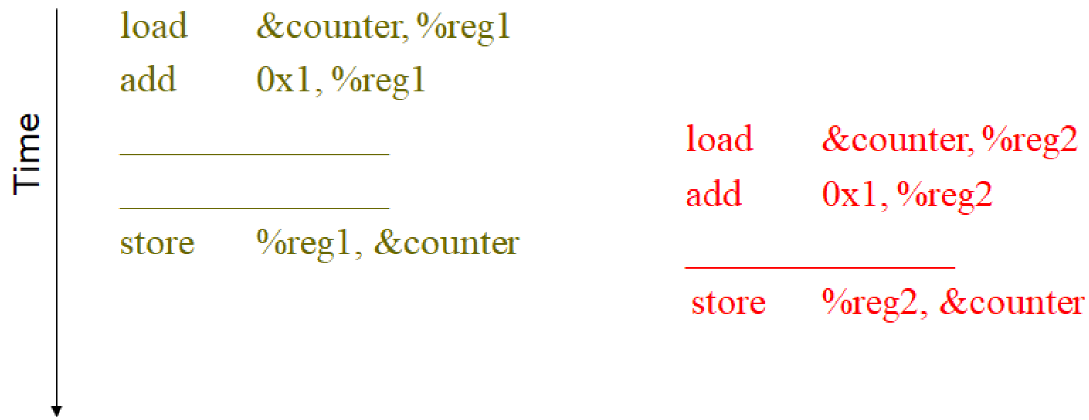
Let’s consider two threads executions

- This problem can happen in both single-core or multi-core CPUs.

This is Race Condition

- Output of a concurrent program depends on the order of operations between threads

```
S0: T1 executes register1 = counter            {register1 = 0}
S1: T1 executes register1 = register1 + 1      {register1 = 1}
S2: T2 executes register2 = counter            {register2 = 0}
S3: T2 executes register2 = register2 + 1      {register2 = 1}
S4: T1 executes counter = register1            {counter = 1 }
S5: T2 executes counter = register2            {counter = 1}
```



Recap: Non Preemptive vs. Preemptive Scheduling

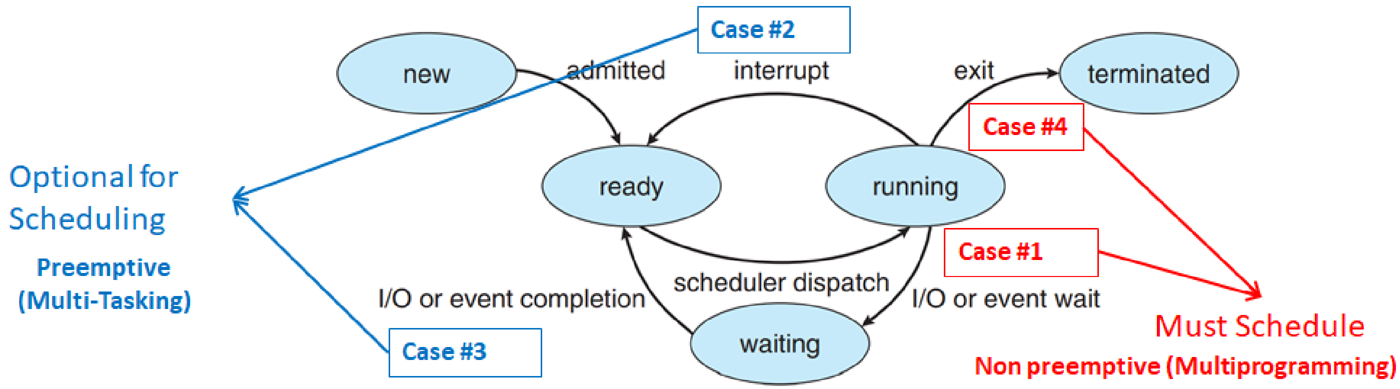


Figure 3.2 Diagram of process state.

- When scheduling takes place only under case 1 and 4, the scheduling scheme is **non preemptive**.
  - Once the CPU has been allocated to a process, the process keeps the CPU until it releases it either by terminating or by switching to the waiting state.
- Otherwise, it is **preemptive**.
  - Preemptive scheduling can result in ***race conditions*** when data are shared among several processes.
  - Virtually all modern operating systems use preemptive scheduling.

Synchronization Motivation

- When threads concurrently read/write shared memory, program behavior is undefined
  - Thread haters -- **“Threads are asynchronous parallelism!”**
  - Two threads write to the same variable; which one should win?
- Thread schedule is non-deterministic
  - Behavior changes when re-run program
- Compiler/hardware instruction reordering
- Multi-word (register) operations are not ***atomic***

What is Atomic Operation?

```
register = counter          (mov &counter, %reg1)
register = register + 1 (add 0x1, %reg1)
counter  = register         (mov %reg1, &counter)
```

```
load          &counter, %reg
add           0x1, %reg
store          %reg, &counter
```

- "load-add-store" must be performed in a single step
- "Atomic" in this context means "all or nothing"
  - Either successful completion of operation with no interruptions or going back to the initial state

Critical Section Problem

- **Critical section:**
  - Piece of code that only one thread can execute at once

Where is critical section???

```
int counter = 0;

int main() {
    CreateThread(fn, 4);
    CreateThread(fn, 4);
    ThreadJoin();
    ThreadJoin();
}
void fn() {
    for(int i = 0; i < 1000000; i++)
    {
        counter++;
    }
}
```

Too Much Milk!

	Person A	Person B
12:30	Look in fridge. Out of milk.	
12:35	Leave for store.	
12:40	Arrive at store.	Look in fridge. Out of milk.
12:45	Buy milk.	Leave for store.
12:50	Arrive home, put milk away.	Arrive at store.
12:55		Buy milk.
1:00		Arrive home, put milk away. Oh no!

Too Much Milk!!!

Too Much Milk!

- Some Notes for **“Too Much Milk”** Problem
  - **Correctness**
    - Someone buys if needed
    - At most one person buys
  - ***Atomic operations*** are not guaranteed here.
  - Concurrent programs (or multi-threads) are non deterministic due to many possible interleaving
  - Person A and B are not “directly” talking each other (relationship may be bad, no cell-phone things), but ***using a note to avoid buying too much milk***
    - Leave a note before buying milk
    - Remove a note after buying milk
    - Don’t buy milk if there is a note

Too Much Milk, Try #1

- Try #1: leave a note

Thread A

```
if (!note)
    if (!milk) {
        leave note
        buy milk
        remove note
}
```

Thread B

```
if (!note)
    if (!milk) {
        leave note
        buy milk
        remove note
}
```

Is this working?

Too Much Milk, Try #2

Thread A

```
if (!milk) {
   leave note_A
   if(!note_B)
        buy milk
   remove note_A
}
```

Thread B

```
if (!milk) {
   leave note_B
   if(!note_A)
        buy milk
   remove note_B
}
```

Too Much Milk, Try #3

- What if thread leaves its note early?

Thread A

```
leave note_A
if (!note_B) {
    if (!milk)
        buy milk
}
remove note_A
```

Thread B

```
leave note_B
if (!note_A) {
    if (!milk)
        buy milk
}
remove note_B
```

Too Much Milk, Try #4

Thread A

```
leave note_A
while (note_B) // X
      do nothing;
if (!milk)
      buy milk;
remove note_A
```

Thread B

```
leave note_B
if (!note_A) {    // Y
     if (!milk)
          buy milk
}
remove note_B
```

Can guarantee at X and Y that either:

(i) Safe for me to buy

(ii) Other will buy, ok to quit

Too Much Milk, Try #4

- At point Y, either there is a note A or not.
  1. If there is no note A, it is safe for thread B to check and buy milk, if needed. (Thread A has not started yet).
  2. If there is a note A, then thread A is checking and buying milk as needed or is waiting for B to quit, so B quits by removing note B.
- At point X, either there is a note B or not.
  1. If there is not a note B, it is safe for A to buy since B has either not started or quit.
  2. If there is a note B, A waits until there is no longer a note B, and either finds milk that B bought or buys it if needed.
- Thus, thread B buys milk (which thread A finds) or not, but either way it removes note B. Since thread A loops, it waits for B to buy milk or not, and then if B did not buy, it buys the milk.

Thread A	Thread B
`leave note_A`	`leave note_B`
`while (note_B) // X`	`if (!note_A) {    // Y`
`    do nothing;`	`    if (!milk)`
`if (!milk)`	`        buy milk`
`    buy milk;`	`}`
`remove note_A`	`remove note_B`

Lessons

- Solution is complicated
  - "Obvious" code often has bugs
- Modern compilers/architectures reorder instructions
  - Making reasoning even more difficult
- Generalizing to many threads/processors
  - Even more complex
  - see Peterson's algorithm and HW supports for sync.