ECE437/CS481

M04A: PROCESS COORDINATION
RACE CONDITION & CRITICAL SECTION

CHAPTER 5.1-5.4

Xiang Sun

The University of New Mexico

Dependent & Independent Processes

- Independent processes: not affected by rest of universe
  - No shared resource (memory, filesystem) among processes
  - Deterministic: results depends on the inputs
  - Block and restart without adverse effects
- Dependent/Cooperative processes: Non-independent
  - shared some resources with other processes
  - Non-deterministic: may have different results each time
  - May be irreproducible: difficult to debug and test

Cooperating Requires Synchronization

- Two processes share account balance

```
void deposit (int amount) {
balance += amount;}
```

  - "Making deposit" can be complied into

```
load balance, R1
add amount, R1
store R1, balance
```

  - What happens if two processes make deposit at same time?

balance=100

Process 0: deposit (10)

Time

```
load balance, R1
add amount, R1
store R1,  balance
```

Process 1: deposit (20)

Time

```
load balance, R1
add amount, R1
store R1,  balance
```

  - What could be the final balance? 100, 110, 120, or 130?
  - Race condition: when two or more processes/threads are accessing the shared resource, the result depends on the ordering of interleaving.

Cooperating Requires Synchronization

❑ Deal with race condition

- ➢ To avoid race condition → setup a critical section (CS)
  - ✓ A critical section is a code segment intended to access a shared resource (file, input or output port, global variable, etc.) that must NOT be concurrently accessed by more than one process/thread of execution.

```
void deposit (int amount) {
disable accessing balance by other processes;
balance += amount;
enable accessing balance by other processes;
}
```

Cooperating Requires Synchronization

❑ Properties/Requirements of a critical section

- Mutual exclusion
  - ✓ Only one process/thread in critical section at a time.
- Guaranteed progress
  - ✓ If no process/thread is in its critical section, and if one or more processes/threads are waiting to execute the critical section, then any one of these processes/threads must be allowed to get into the critical section.
- Bounded waiting
  - ✓ Must eventually allow waiting processes/threads to proceed.
- Solution should be independent on
  - ✓ # of CPUs, # of processes/threads, and speed of CPUs.

Cooperating Requires Synchronization

❑ How to establish/realize a CS?

- ➢ Disable interrupts, which essentially disable the context switch.

```
disable interrupt;
Critical Section (CS);
enable interrupt;
```

  - ✓ Not work if there are multiple CPUs. Disabling interrupts affects just one CPU.
  - ✓ Unsafe to give the system privilege to user.

Cooperating Requires Synchronization

❑ How to establish/realize a CS?

➢ Using lock variable

```
shared bool lock = FALSE;
Process A                          Process B
while (lock == TRUE)               while (lock == TRUE)
    /* null */ ;                       /* null */ ;
lock = TRUE;                       lock = TRUE;
    CS;                                CS;
lock = FALSE;                      lock = FALSE;
```

✓ Not work since setting the lock does not satisfy mutual exclusion, thus the CS is not mutual exclusion.

Cooperating Requires Synchronization

❑ How to establish/realize a CS?

➢ Using separated lock variables

```
shared bool lock0 = FALSE, lock1 = FALSE;

Process A                       Process B
lock0 = TRUE;                   lock1 = TRUE;
while (lock1 == TRUE)           while (lock0 == TRUE)
    /* null */ ;                    /* null */ ;
CS;                             CS;
lock0= FALSE;                   lock1 = FALSE;
```

- ✓ Can achieve mutual exclusion in single core system.
- ✓ Bounded waiting does not satisfy, i.e., the two processes are waiting for each other forever.

Cooperating Requires Synchronization

❑ How to establish/realize a CS?

➢ Using turn variable

```
shared int turn = 0,
Process A                    Process B
while (turn != 0)            while (turn != 1)
/* null */ ;                 /* null */ ;
CS;                          CS;
turn = 1;                    turn = 0;
```

- ✓ Can achieve mutual exclusion in single core system.
- ✓ Guaranteed progress does not satisfy.

Cooperating Requires Synchronization

❑ How to establish/realize a CS?

➢ Using turn + separated locks (Peterson's solution)

I request to access the CS →
Assume that it is your turn →
If you request to access the CS and it is your turn, then I will wait. →

```
shared int turn=0; shared bool lock[2] = {FALSE, FALSE};
Process A                          Process B
lock[0] = TRUE;                    lock[1] = TRUE; /*req*/
turn = 1;                          turn = 0;
while (lock[1] && turn==1)         while (lock[0] && turn==0)
    /* null */ ;                       /* null */ ;
CS;                                CS;
lock[0] = FALSE;                   lock[1] = FALSE;
```

- ✓ Can achieve mutual exclusion in single core system. But it is still unable to guarantee progress.
- ✓ Complicated and error-prone when dealing with N processes.
- ✓ A simple and low-level solution is needed.