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 state (memory, filesystem) among process
  - Deterministic: Input alone determines results
  - Block and restart without adverse effects
- Dependent/Cooperative processes: Non-independent (not necessarily "cooperative")
  - shared some states 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

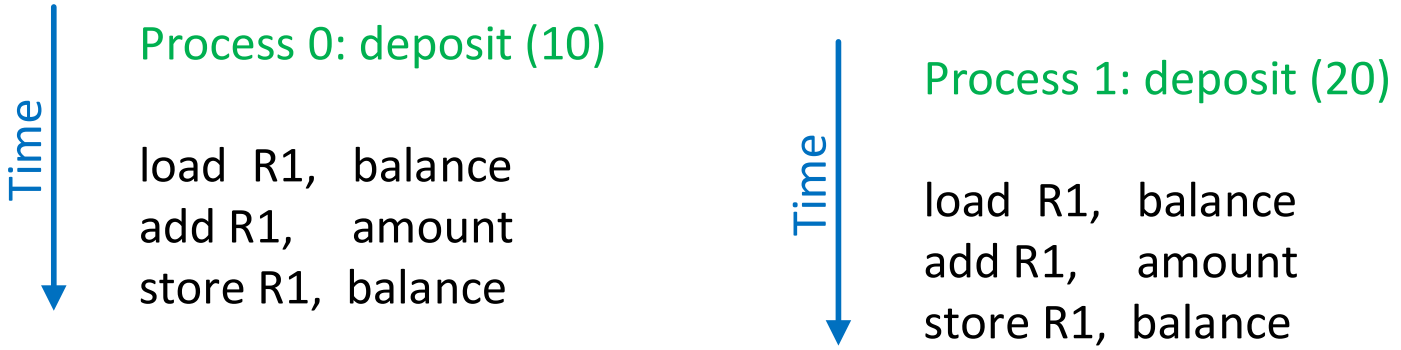
```
void deposit (int amount) {
balance += amount;}
```

  - "Making deposit" can be complied into

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

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

Balance=100



  - What is the final balance? 100, 110, 120, or 130?
  - Race condition: when two or more processes are accessing the shared resource, result depends on 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 that accesses a shared resource (file, input or output port, global data, etc.) that must NOT be concurrently accessed by more than one thread of execution.

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

  - ✓ For a given shared resource, the critical sections may be:
    1. the same code segment in the one process, repeated many times
    2. the different code segments in the same process
    3. the same or different code segments in different processes

Cooperating Requires Synchronization

❑ Properties/Requirements of a critical section

- Mutual exclusion
  - ✓ Only one process in critical section at a time
- Guarantee progress
  - ✓ If no process is in its critical section, and if one or more processes are waiting to execute the critical section, then any one of these threads must be allowed to get into the critical section.
- Bounded waiting
  - ✓ Must eventually allow waiting process 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;
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 variables

```
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;
```

  - ✓ Not work since 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 variables

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

✓ Not work since guarantee progress does not satisfy, e.g., Process B may be blocked (in accessing the CS) by Process A.

Cooperating Requires Synchronization

❑ How to establish/realize a CS?

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

```
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;
```

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 →

- ✓ It works!
- ✓ Complicated and error-prone when dealing with N processes.
- ✓ A simple and low-level solution is needed.