Process Synchronization

CS3600 Spring 2022

Synchronization

- Concurrency: Multiple process(or threads) executing at the same time.
- Concurrent access to shared data may result in data inconsistency.
- Race Condition – data inconsistency when two processes/threads trying to access the same data space at same time

Race Condition

- In P1
- Shared Buffer count++ could be implemented in assembly as

```
register1 = count
register1 = register1 + 1
count = register1
```

- In P2
- Shared Buffer count-- could be implemented as

```
register2 = count
register2 = register2 - 1
count = register2
```

Sequential Program

- Consider a sequential execution with “count = 5” initially:

Time	Instruction	Register/variables
T0 P1 execute	register1 = count	register1 = 5
T1 P1 execute	register1 = register1 + 1	register1 = 6
T2 P1 execute	count = register1	count = 6
T3 P2 execute	register2 = count	register2 = 6
T4 P2 execute	register2 = register2 - 1	register2 = 5
T5 P2 execute	count = register2	count = 5

Running concurrently

- Consider a sequential execution with "count = 5" initially:

Time	Instruction	Register/variables
T0 P1 execute	register1 = count	register1 = 5
T1 P1 execute	register1 = register1 + 1	register1 = 6
T2 P2 execute	register2 = count	register2 = 5
T3 P2 execute	register2 = register2 - 1	register2 = 4
T4 P1 execute	count = register1	count = 6
T5 P2 execute	count = register2	count = 4

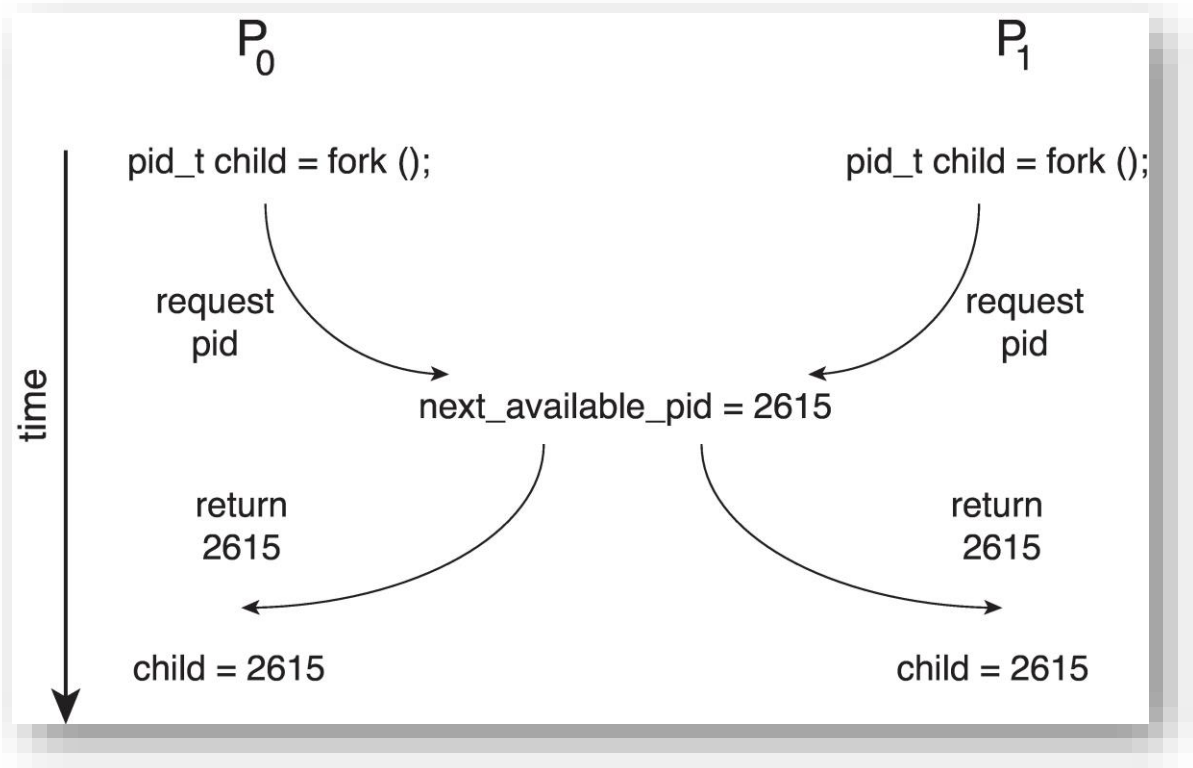
Race Condition

- Consider a sequential execution with “count = 5” initially:

Time	Instruction	Register/variables
T0 P1 execute	register1 = count	register1 = 5
T1 P1 execute	register1 = register1 + 1	register1 = 6
T2 P2 execute	register2 = count	register2 = 5
T3 P2 execute	register2 = register2 - 1	register2 = 4
T4 P1 execute	count = register1	count = 6
T5 P2 execute	count = register2	count = 4

Race Condition-Example 2

- Processes P_0 and P_1 are creating child processes using the `fork()` system call
- Race condition on kernel variable `next_available_pid` which represents the next available process identifier (pid)



CRITICAL SECTION

- Critical section, in which the process may be
  - changing common variables,
  - updating a table,
  - writing a file

```
do {
        entry section
            critical section
        exit section
            remainder section
} while (true);
```

Software solution to Critical Sections

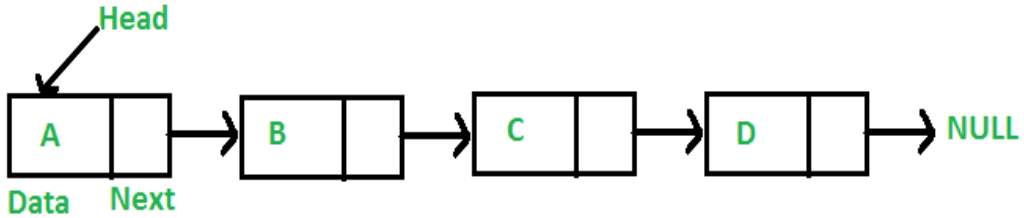
- Guarantee ***mutual exclusion***: Only one process may be executing within the CS.
- Prevent ***lockout***: A process not attempting to enter the CS must not prevent other processes from entering the CS.
- Prevent ***starvation***: A process (or a group of processes) must not be able to repeatedly enter the CS while other processes are waiting to enter.
- Prevent ***deadlock***: Multiple processes trying to enter the CS at the same time must not block each other indefinitely.

Mutex Lock

- OS designers build software tools to solve critical section problem
- Simplest is mutex lock
  - Boolean variable indicating if lock is available or not
- Protect a critical section by
  - First `acquire()` a lock
  - Then `release()` the lock
- Calls to **`acquire()`** and **`release()`** must be **atomic**
  - Usually implemented via hardware atomic instructions such as compare-and-swap.
- But this solution requires **busy waiting**
  - This lock therefore called a **spinlock**

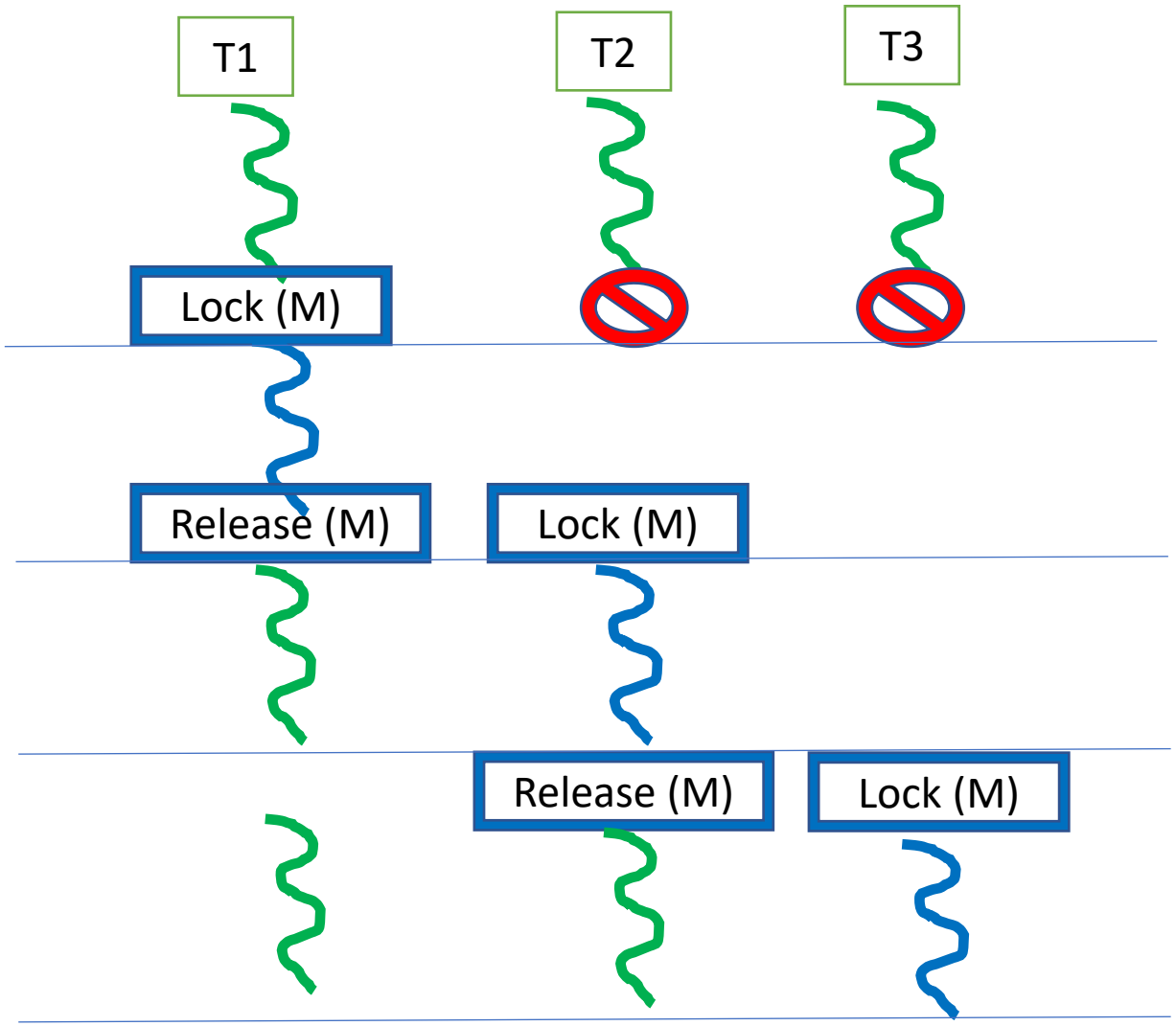
Critical section problem

```
List mylist[10];
Mutex M;

Void *insert(Node n){
Mylist.insert(n);
}

create thread(p1,NULL,insert,4);
create thread(p1,NULL,insert,6);
```



Mutex

```
Lock (Mutex) {
   Critical Section
}
Release(Mutex)
```



Mutex solution

```
List mylist[10];
Mutex M;

Void insert(Node n){
lock(M);
Mylist.insert(n);
unlock(M);
}

create thread(p1,NULL,insert,4);
create thread(p2,NULL,insert,8);
```

Using Mutex locks

```
while (true) {
        acquire lock

            critical section

        release lock

remainder section
}
```

Pthread Mutex

//Declaring a mutex using POSIX library

```
pthread_mutex_t  lock   =   PTHREAD_MUTEX_INITIALIZER;

Pthread_mutex_lock(&lock);

 //Critical section

Pthread_mutex_unlock(&lock);
```

Do we use same variable for all critical sections?

Demo

Classwork 01

- See the program mutex1.c in Canvas and run to see the working.

- Remove the comments from the program and observe how the identified the critical section works now and how it avoids the race condition using mutex.

Classwork 02 –lab 4 Team work in Breakout rooms

- Write a program with one thread function , define an array with 5 elements as global variable, define a static variable for increment value [use the file mutex2.c program for this lab] .

- Call the thread 2 times
  - First for incrementing the array elements by 2.
  - Second for incrementing the array elements by 3.
- Give a "sleep(2)" in the for loop of the thread function to model some process happening in between.
- Check if you are getting an expected output of incremented array elements by printing the array in main after closing the threads.

- Identify the critical section.
- Avoid the critical section using mutex.
- Show critical section start and end using comments //start cs and //end cs. Submit the program with mutex.