Faculty of Computer Science & Engineering

Operating Systems

Nguyen Minh Tri
nmtribk@hcmut.edu.vn
302-B9

Ho Chi Minh City
University of Technology

Lab 6 - Synchronization

Objective

- Understand race condition in concurrence programming.
- Know how to use synchronization techniques to solve race condition.

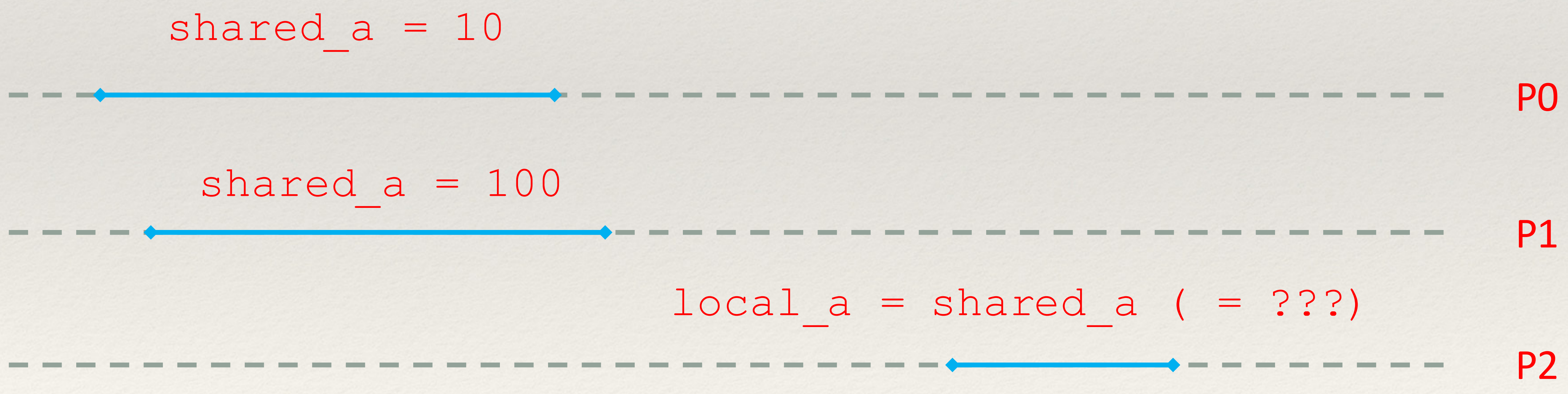
Race condition definition?

- Exercise: Compile and run the following program. Remember to compile with -lpthread option.

```
#include <stdio.h>
#include <pthread.h>
int shared_data = 0;
void * mythread(void * arg) {
     int i;
     for (i = 0; i < 1000000; i++) {
          shared_data++;
     }
}
int main() {
     pthread_t t1, t2;
     pthread_create(&t1, NULL, mythread, NULL); pthread_create(&t2, NULL, mythread, NULL);
     pthread_join(t1, NULL); pthread_join(t2, NULL);
     printf("The value of shared_data: %d\n", shared_data);
}
```

Race condition definition?

- A race condition (or race hazard) occurs when two or more threads can access shared data and they try to change it at the same time.
- In these situations, the output is dependent on the sequence or timing of uncontrollable events.



Race condition definition?

- Race condition occurs in shared memory system (i.e. a computer with multiple processing cores).
- Race condition also occurs in distributed system (shared variables) and uniprocessor system (multitasking)

Other terms?

- What is critical section?
- What is mutual exclusion?
- What is lock?
- What is semaphore?

Other terms?

- A critical section (region) is a piece of code where the shared resource is accessed. The critical section must be protected from being executed by multiple processes (or threads).
- In order to protected critical sections, threads (or processes) must access their own critical sections mutually exclusive: one thread never enter its mutual exclusion at the same time that another thread enters its own critical section.
- A lock (mutex) is a synchronization mechanism designed to enforce mutual exclusion concurrence control policy.
- Semaphore is a variable (or abstract data type) used for controlling access from multiple processes to a shared resource.

Locking

- Imagine we have a big room accessible through multiple doors.
- We need only one key to open those door since those doors use identical locks.
- The room is shared by many people, each of them go to the room through their own door.
- When a person enter the room, he must remember to lock his door and keep the key with him. When this person leave the door, he also remember to lock this door but can give the key to any person who want to enter the room.

Locking

- Room -> share resource
- Door -> critical section
- People -> Threads (processes)
- Key -> Lock (mutex) variable

Locking

- Before entering critical section, thread A must acquire a lock.
- If there is no thread entering their critical section, lock is acquired and A is permitted to enter its critical section
- If there is a thread B that is currently in its critical section then A is blocked until B leave the critical section.
- After leaving critical section, thread A must release the lock to make it available to another threads.

Locking

- Linux supports locking mechanism through POSIX pthread library:
- Mutex variable: pthread_mutex_t
- Initialize a mutex variable before use
- int pthread_mutex_init(*mutex, *attr)
- Acquire lock:
- int pthread_mutex_lock(*mutex)
- Release lock:
- int pthread_mutex_unlock(*mutex)

Locking

- Exercise: Correct the previous example using Pthread lock functions. Remember to compile with -lpthread option.

Locking

- Solution

```c
#include <stdio.h>
#include <pthread.h>
int shared_data = 0;
pthread_mutex_t lock;
void * mythread(void * arg) {
    int i;
    for (i = 0; i < 1000000; i++) {
        pthread_mutex_lock(&lock);
        shared_data++;
        pthread_mutex_unlock(&lock);
    }
}
int main() {
    pthread_t t1, t2;
    pthread_mutex_init(&lock, NULL);
    pthread_create(&t1, NULL, mythread, NULL); pthread_create(&t2, NULL, mythread, NULL);
    pthread_join(t1, NULL); pthread_join(t2, NULL);
    printf("The value of shared_data: %d\n", shared_data);
}
```

Locking

- Exercise: Use time command to measure the running time of the two implementation.

Locking

- Exercise: Compile and run the following program.

```
#include <stdio.h>
#include <pthread.h>
Pthread_mutex_t lock[2];
void * tfunc(void * arg) {
    int id = (int)arg;
    pthread_mutex_lock(&lock[id]);
    printf("Thread %d got lock %d\n", id, id); sleep(1);
    pthread_mutex_lock(&lock[1 - id]);
    printf("Hello from thread %d\n", id);
    pthread_mutex_unlock(&lock[id]);
    pthread_mutex_unlock(&lock[1 – id]);
}
int main() {
    pthread_t t1, t2;
    pthread_mutex_init(&lock[0], NULL); pthread_mutex_init(&lock[1], NULL);
    pthread_create(&t1, NULL, tfunc, (void *)0); pthread_create(&t2, NULL, tfunc, (void *)1);
    pthread_join(t1, NULL); pthread_join(t2, NULL);
}
```

Locking

- Disadvantage of using lock:
  - Low performance: Contention, overhead.
  - Deadlock
  - Priority inversion
  - Convoying
  - Hard to debug

Only use lock if we have no choice.

Semaphore

- Imagine we have multiple single rooms in a hotel, each has its own door, lock and key.
- Those rooms are available for many people to use but only one person is allowed to use one room at a given moment.
- A room clerk is hired and he has responsibility for room assignment.
- A person must request a room from the clerk. If no room is free then he must wait until someone checkout his room.
- To access the room, the person must acquire the key from the clerk and after checkout, he must return it to the clerk.

Semaphore

- Room -> share resources
- People -> Processes (threads)
- Room Clerk -> Semaphore variable

Semaphore

- Exercise: Do semaphore exercise in the instruction.

End

Thanks!