07. Synchronization I

CS 4352 Operating Systems

Adopted from slides of Drs. Silberschatz, Huang, & Bryant

Background

- Processes and threads can execute concurrently
  - May be interrupted at any time, partially completing execution
- Concurrent access to shared data may result in data inconsistency
- Maintaining data consistency requires mechanisms to ensure the orderly execution of cooperating processes and threads

Race Condition

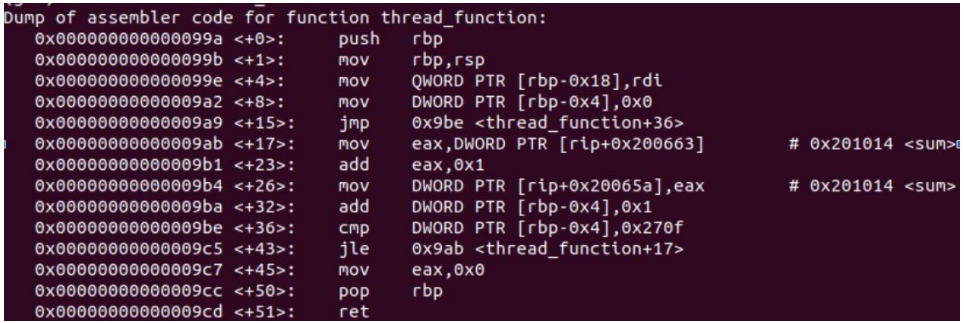
- Race condition: a situation where the result produced by multiple threads (or processes) operating on shared resources depends (in an unexpected way) on the relative order in which the processes gain access to the CPU(s)
- What causes race conditions?
  - In a nutshell: non-atomic operations

Example

```c
#include <stdio.h>
#include <pthread.h>

int sum = 0;

// each thread will increment sum 10000 times
void *thread_function(void *arg)
{
  for (int i = 0; i < 10000; i++) {
    sum++;
  }

  return NULL;
}

int main(int argc, char *argv[])
{
    pthread_t p1, p2;

    // create two threads
    pthread_create(&p1, NULL, thread_function, NULL);
    pthread_create(&p2, NULL, thread_function, NULL);

    // wait for both to finish
    pthread_join(p1, NULL);
    pthread_join(p2, NULL);

    // print the final value of sum
    printf("Final value of sum: %d\n", sum);
    return 0;
}
```

What Causes This Behavior?

- This race condition is caused by the fact that sum++ is implemented as three individual instructions:
  - mov eax, DWORD PTR [rip+0x200663]
  - add eax, 0x1
  - mov DWORD PTR [rip+0x20065a], eax



- Key point: the OS can switch to a different process after any of these instructions!

Bonus Time!

If you earn this point, it will be added to your assignment final grade.

```
int x = 0;
```

Thread 1

```
void foo()
{
    x++;
}
```

Thread 2

```
void bar()
{
    x--;
}
```

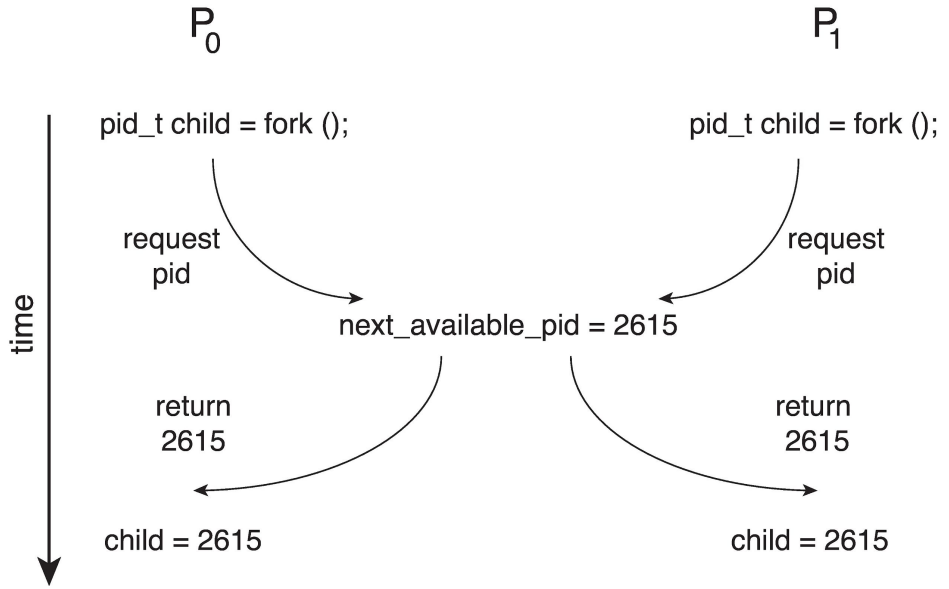
After thread 1 and thread 2, what can the value of x be?

a. -3
b. -2
c. -1
d. 0
e. 1
f. 2
g. 3

Another Example

- Processes P0 and P1 are creating child processes using the fork() system call
- Race condition on kernel variable next_available_pid which represents the next available PID

- Unless there is a mechanism to prevent P0 and P1 from accessing the variable next_available_pid, the same PID could be assigned to two different processes!



Critical Section Problem

- Consider system of n processes {p0, p1, …}
- Each process has **critical section** segment of code
  - Process may be changing common variables, updating table, writing file, etc.
  - When one process in critical section, no other may be in its critical section
- **Critical section problem** is to design protocol to solve this

General Structure of Process

- Each process must ask permission to enter critical section in **entry section**, may follow critical section with **exit section**, then **remainder section**

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

Three Requirements

- Mutual exclusion (mutex)
  - If a process is executing in its critical section, then no other processes can be executing in their critical sections
- Progress
  - If some process is not in the critical section, it cannot prevent some others from entering the critical section
  - A process in the critical section will eventually leave it
- Bounded waiting (no starvation)
  - If some process is waiting on the critical section, it will eventually enter the critical section

Interrupt-based Solution

- Entry section: disable interrupts
- Exit section: enable interrupts
- Is this solution good enough in all cases?
  - What if the critical section is code that runs for an hour?
    - No interrupt is served in that hour!
  - Can some processes starve?
    - Suppose no starvation due to scheduling
  - What if there are two CPUs?
    - You have to disable both

Peterson’s Solution

- Two process solution
- Assume that the **load** and **store** machine-language instructions are atomic; that is, cannot be interrupted
- The two processes share two variables:
  - `int turn`
    - The variable turn indicates whose turn it is to enter the critical section
  - `boolean flag[2]`
    - The flag array is used to indicate if a process is ready to enter the critical section.
    - flag[i] = true implies that process Pi is ready!

Algorithm for Process Pi

```
while (true){

        flag[i] = true;
        turn = j;
        while (flag[j] && turn = = j)
                ;

        /* critical section */

        flag[i] = false;

        /* remainder section */

}
```

```
ture = j;
while (turn == j)
  ;

Critical section

turn = i;
```

Correctness of Peterson's Solution

- Provable that the three critical section requirements are met:
  - Mutual exclusion is preserved
    - Pi enters critical section only if either flag[j] = false or turn = i
    - What if we remove flags, will this requirement still be satisfied?
  - Progress requirement is satisfied
    - What if we remove flags, will this requirement still be satisfied?
  - Bounded-waiting requirement is met
    - Each tries to let the other guy run first

Peterson’s Solution and Modern Architecture

- Although useful for demonstrating an algorithm, Peterson’s Solution is not guaranteed to work on modern architectures.
- To improve performance, processors and/or compilers may reorder operations that have no dependencies

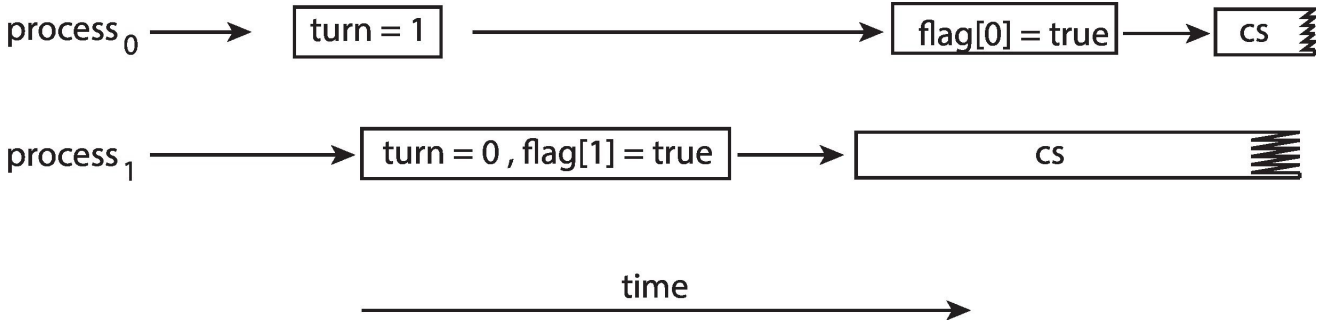
Modern Architecture Example

- Two threads share the data:
  boolean flag = false;
  int x = 0;
- Thread 1 performs
  while (!flag)
    ;
  print x
- Thread 2 performs
  x = 100;
  flag = true
- What is the expected output? 100

- Can it be 0?
- Since the variables flag and x are independent of each other, the instructions:
  flag = true;
  x = 100;
  for Thread 2 may be reordered

Peterson's Solution Revisited

- The effects of instruction reordering in Peterson's Solution



- This allows both processes to be in their critical section at the same time!
- To ensure that Peterson's solution will work correctly on modern computer architecture we must use **memory barriers**

Memory Barrier

- Memory models are the memory guarantees a computer architecture makes to application programs
- Memory models may be either:
  - Strongly ordered – where a memory modification of one processor is immediately visible to all other processors
  - Weakly ordered – where a memory modification of one processor may not be immediately visible to all other processors
    - E.g., x86
- A **memory barrier** is an instruction that forces any change in memory to be propagated (made visible) to all other processors

Memory Barrier Instructions

- When a memory barrier instruction is performed, the system ensures that all loads and stores are completed before any subsequent load or store operations are performed
- Examples:
  - x86
    - Mfence (both load and store)
    - Lfence (load)
    - Sfence (store)
  - ARM
    - DSB (data sync barrier)
    - DMB (data memory barrier)
    - ISB (instruction sync barrier)

Modern Architecture Example Revisited

- Two threads share the data:
  boolean flag = false;
  int x = 0;
- Thread 1 performs
  while (!flag)
    ;
  print x
- Thread 2 performs
  x = 100;
  MBI
  flag = true

Synchronization Hardware

- Many systems provide hardware support for implementing the critical section code
- Uniprocessors – could disable interrupts
  - Currently running code would execute without preemption
  - Generally too inefficient on multiprocessor systems
    - Operating systems using this not broadly scalable
- We will look at several forms of hardware support:
  - Hardware instructions
  - Atomic variables

Hardware Instructions

- Special hardware instructions that allow us to either test-and-modify the content of a word, or two swap the contents of two words atomically (uninterruptedly)
  - Test-and-Set instruction
  - Compare-and-Swap instruction
- Intel also introduced an interesting ISA extension called TSX (Transactional Synchronization Extensions)
  - Very interesting idea
  - Makes synchronization extremely efficient

The Test-and-Set Instruction

- Functionality definition

```
boolean test_and_set (boolean *target)
{
    boolean rv = *target;
    *target = true;
    return rv:
}
```

- Properties
  - Executed atomically
  - Returns the original value of passed parameter
  - Set the new value of passed parameter to true

Solution Using Test-and-Set

- Shared boolean variable **lock**, initialized to **false**
- Solution:

```
do {
    while (test_and_set(&lock))
        ; /* do nothing */

    /* critical section */

    lock = false;

    /* remainder section */
} while (true);
```

The Compare-and-Swap Instruction

- Functionality definition

```
int compare_and_swap(int *value, int expected, int new_value)
{
   int temp = *value;
   if (*value == expected)
       *value = new_value;
    return temp;
}
```

- Properties
  - Executed atomically
  - Returns the original value of passed parameter value
  - The swap takes place only under the condition *value == expected is true

Solution Using Compare-and-Swap

- Shared integer **lock** initialized to **0**
- Solution:

```
while (true){
    while (compare_and_swap(&lock, 0, 1) != 0)
        ; /* do nothing */

    /* critical section */

    lock = 0;

    /* remainder section */
}
```

TSX

- The intent of TSX is to both provide a simplified interface for synchronization and to enable optimistic concurrency
  - Atomic memory transactions are either committed or aborted
  - Processes abort only when a conflict exists, rather than when a potential conflict may occur, as with traditional locks
- Try hardware transactional memory intrinsics in gcc yourself
  - _xbegin()
  - _xend()

```
if (_xbegin () == _XBEGIN_STARTED) {
        ... critical section code ...
        _xend ();
}
```

Atomic Variables

- Typically, instructions such as compare-and-swap are used as building blocks for other synchronization tools
- One tool is an **atomic variable** that provides atomic (uninterruptible) updates on basic data types such as integers and booleans
  - E.g., increment(&x), where x is an atomic_int

```
void increment(atomic_int *v)
{
    int temp;
    do {
        temp = *v;
    }
    while (temp != compare_and_swap(v, temp, temp + 1));
}
```

Next Lecture

Continue synchronization