CISC 3320
Race Condition and Critical Section

Hui Chen

Department of Computer & Information Science

CUNY Brooklyn College

Acknowledgement

- These slides are a revision of the slides provided by the authors of the textbook via the publisher of the textbook

Outline

- Race condition
- The Critical-Section Problem
- Peterson’s Solution

Background

- Processes 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
- Otherwise, a race condition occurs
  - Examining two examples

Race Condition: Example 1

- Illustration of the problem using the consumer-producer problem
  - Suppose that we wanted to provide a solution to the consumer-producer problem that fills all the buffers.
  - We can do so by having an integer counter that keeps track of the number of full buffers.
  - Initially, counter is set to 0.
  - It is incremented by the producer after it produces a new buffer and is decremented by the consumer after it consumes a buffer.

Producer

```
while (true) {
      /* produce an item in next produced */


      while (counter == BUFFER_SIZE)
            ; /* do nothing */
      buffer[in] = next_produced;
      in = (in + 1) % BUFFER_SIZE;
      counter++;
}
```

Consumer

```
while (true) {
      while (counter == 0)
            ; /* do nothing */
      next_consumed = buffer[out];
      out = (out + 1) % BUFFER_SIZE;
         counter--;
      /* consume the item in next consumed */
}
```

Increment and Decrement Counter

- counter++ could be implemented as

```
register1 = counter
register1 = register1 + 1
counter = register1
```

- counter-- could be implemented as

```
register2 = counter
register2 = register2 - 1
counter = register2
```

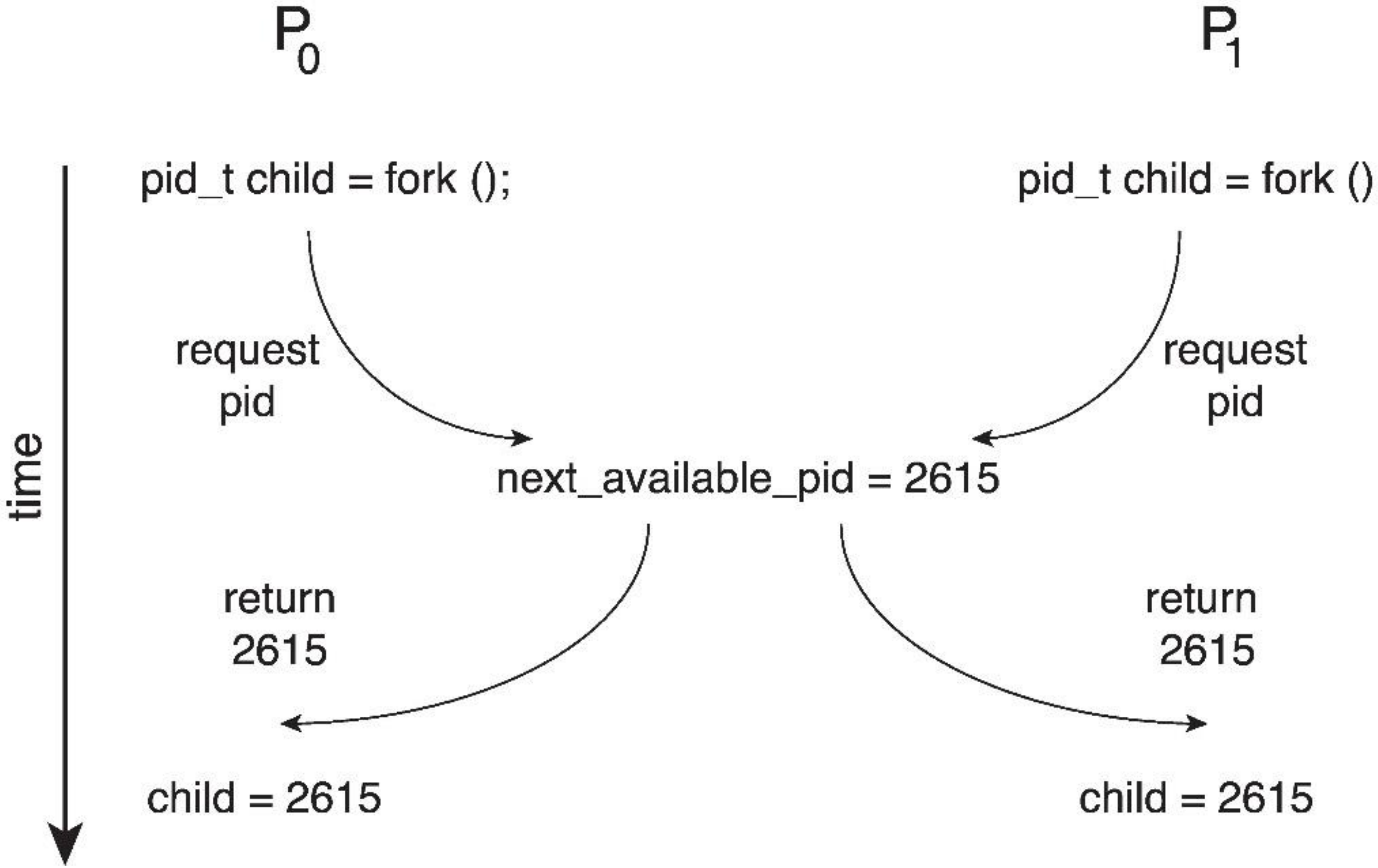
Race Condition

- Consider this execution interleaving with "count = 5" initially:

S0: producer execute register1 = counter {register1 = 5}
S1: producer execute register1 = register1 + 1 {register1 = 6}
S2: consumer execute register2 = counter {register2 = 5}
S3: consumer execute register2 = register2 – 1 {register2 = 4}
S4: producer execute counter = register1 {counter = 6 }
S5: consumer execute counter = register2 {counter = 4}

Race Condition: Example 2

- 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 process identifier (pid)
- The same pid could be assigned to two different processes!


P0
P1
time
pid_t child = fork ();
pid_t child = fork ();
request pid
request pid
next_available_pid = 2615
return 2615
return 2615
child = 2615
child = 2615

Questions?

- Race condition may occur when processes/threads execute concurrently
- There is a need for process synchronization

Critical Section Problem

- Consider system of ***n*** processes {$p_0, p_1, \ldots p_{n-1}$}
- 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 **otherwise** a race condition may occur
- ***Critical section problem*** is to design **protocol** to solve this
- **The protocol**
  - Each process must ask permission to enter critical section in **entry section**, may follow critical section with **exit section**, then **remainder section**

Critical Section

- General structure of process P_i

```
do {

    entry section

        critical section

    exit section

        remainder section

} while (true);
```

Requirements to Critical-Section Problem

- 3 requirements must be met
  - Mutual Exclusion
  - Progress
  - Bounded Waiting

- Assume that each process executes at a nonzero speed
- No assumption concerning relative speed of the *n* processes

Mutual Exclusion

- If process P_i is executing in its critical section, then no other processes can be executing in their critical sections

Progress

- If no process is executing in its critical section and there exist some processes that wish to enter their critical section, then the selection of the processes that will enter the critical section next cannot be postponed indefinitely

Bounded Waiting

- A bound must exist on the number of times that other processes are allowed to enter their critical sections after a process has made a request to enter its critical section and before that request is granted

Questions?

- Requirements for critical section problem
  - Mutual exclusion
  - Progress
  - Bounded waiting

Peterson's Solution

- Not guaranteed to work on modern architectures! (But good algorithmic description of solving the problem)
- Two-process solution
  - Assume that the load and store machine-language instructions are atomic; that is, cannot be interrupted
- The two processes P_0 and P_1 share two variables:
  - `int turn; boolean flag[2];`
  - The variable `turn` indicates whose turn it is to enter the critical section
  - The `flag` array is used to indicate if a process is ready to enter the critical section.
    - `flag[i] = true` implies that process P_i is ready!
- For convenience, we represent P_0 and P_1 as P_i and P_j noting $j = 1 - i$ and $i = 1 - j$ where $i, j \in \{0, 1\}$

Algorithm for Process P_i

- Notice notations of "i" and Process P_i

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

        /* critical section */

        flag[i] = false;

        /* remainder section */

}
```

Peterson’s Solution: 3 Requirements

- Provable that the 3 critical section requirement are met:

1. Mutual exclusion is preserved

2. Progress requirement is satisfied

3. Bounded-waiting requirement is met

Peterson's Solution: Mutual Exclusion

- Note that
  - P_i enters its critical section only if either flag[j] == false or turn == i.
  - if both processes can be executing in their critical sections at the same time, then flag[0] == flag[1] == true.
- These two observations imply that
  - P_0 and P_1 could not have successfully executed their while statements at about the same time, since the value of turn can be either 0 or 1 but cannot be both.
- Therefore,
  - one of the processes, e.g., P_j must have successfully executed the while statement, whereas P_i had to execute at least one additional statement ("turn == j").
  - However, at that time, flag[j] == true and turn == j, and this condition will persist as long as Pj is in its critical section; as a result, mutual exclusion is preserved.

Peterson’s Solution: Remarks

- Although useful for demonstrating an algorithm, Peterson’s Solution is not guaranteed to work on modern architectures.
- Understanding why it will not work is also useful for better understanding race conditions.
- *To improve performance, processors and/or compilers may reorder operations that have no dependencies.*
- For single-threaded this is OK as the result will always be the same.
- For multithreaded the reordering may produce inconsistent or unexpected results!

Peterson's Solution: 2-Thread 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
```

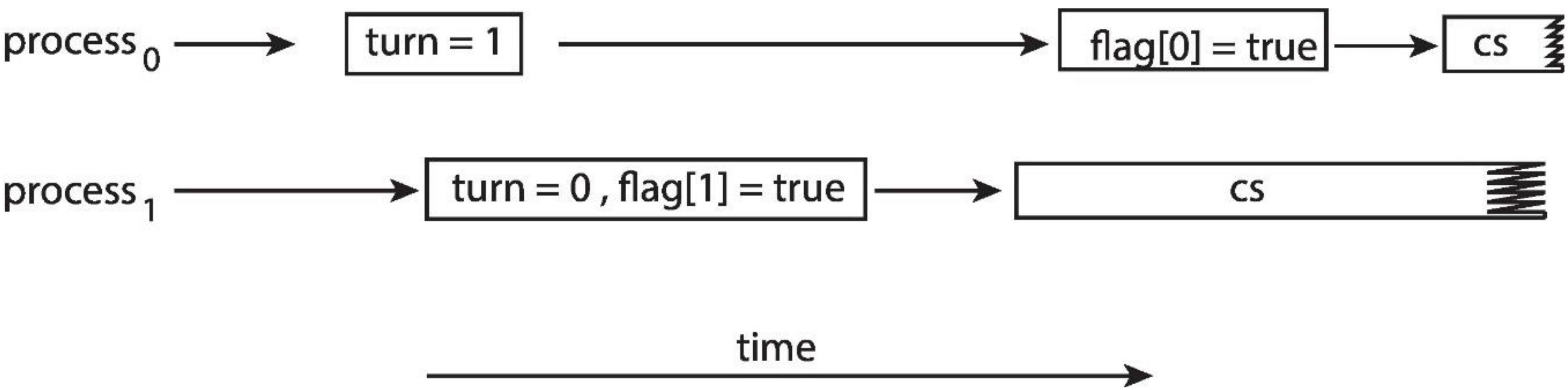
- What is the expected output?

Peterson's Solution

- 100 is the expected output.
- However, the operations for Thread 2 may be reordered:

```
flag = true;
x = 100;
```

- If this occurs, the output may be 0!
  - The effects of instruction reordering in Peterson's Solution
  - This allows both processes to be in their critical section at the same time!


process 0
turn = 1
flag[0] = true
cs
process 1
turn = 0 , flag[1] = true
cs
time

Questions?

- Peterson's solution
  - A software solution, a good description of an algorithm solving the problem
- Is it guaranteed to work on modern operating systems?

Critical-Section Handling in OS

- Two approaches to handle critical sections
- Non-preemptive kernels
  - Run until exits kernel mode, blocks, or voluntarily yields CPU, i.e., only one process is active in the kernel at a time
  - Essentially free of race conditions on kernel data structures as only one process is active in the kernel at a time
- Preemptive kernels
  - allow preemption of process when running in kernel mode, i.e., multiple processes are active in the kernel at a time
  - Must handle critical section, which results in more difficult/complex design of preemptive kernels than that of non-preemptive kernels
  - However, necessary for real-time and responsive kernels.

Questions?

- Critical section handling in OS kernels
  - Non-preemptive kernels?
  - Preemptive kernels?