Amirkabir University of Technology
(Tehran Polytechnic)

Operating Systems

Synchronization Tools-Part1

Seyyed Ahmad Javadi

sajavadi@aut.ac.ir

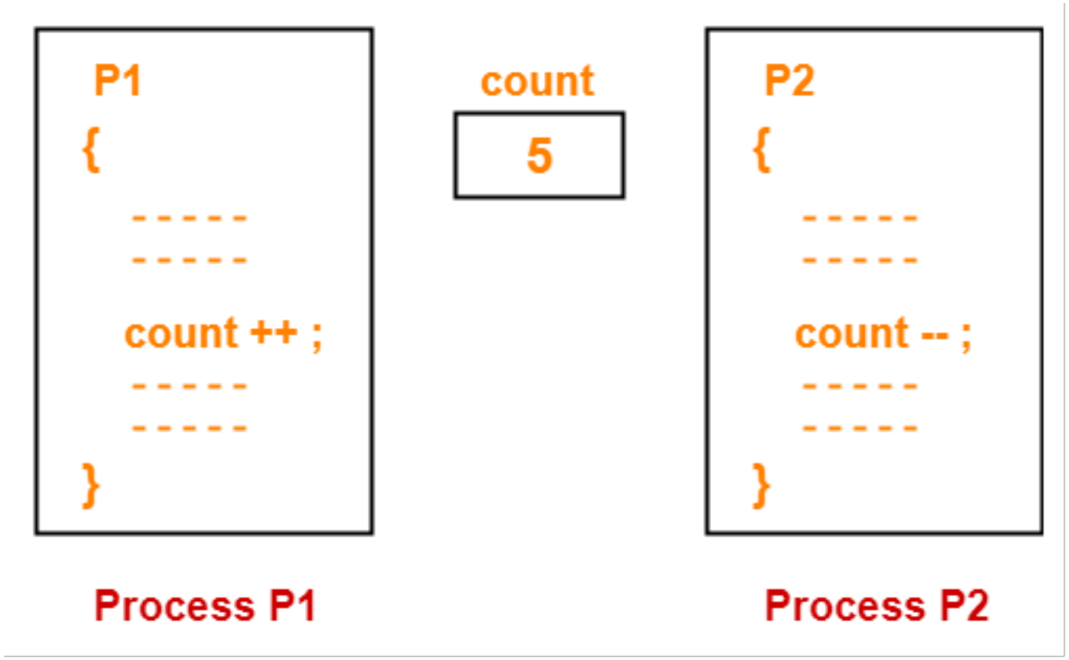
Spring 2023

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**.

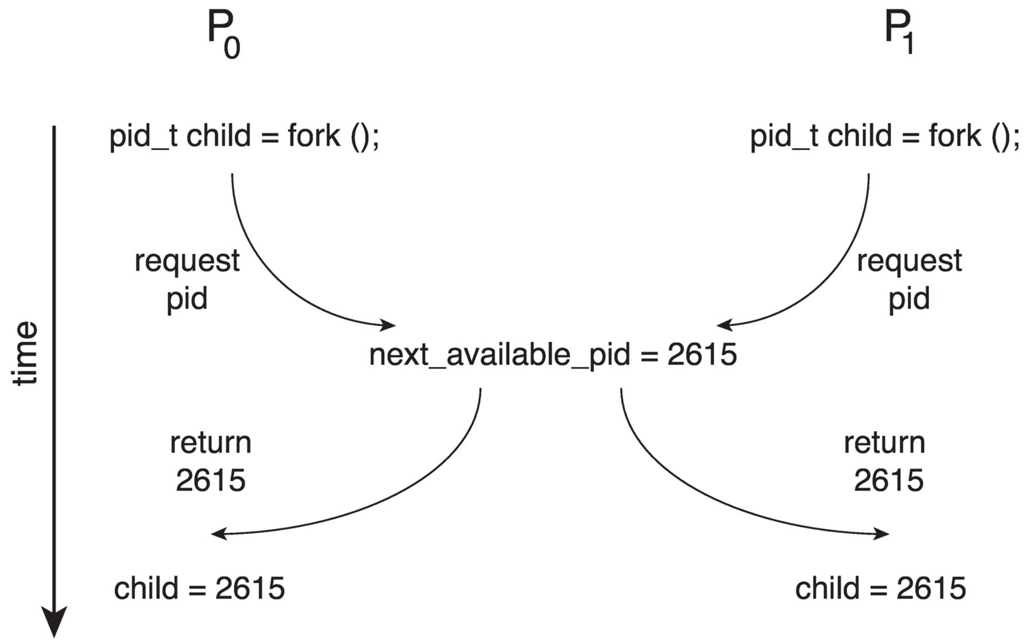
Background (cont.)

- In chapter 4, when we considered the Bounded Buffer problem with use of a counter that is updated concurrently by the producer and consumer, which **lead to race condition.**

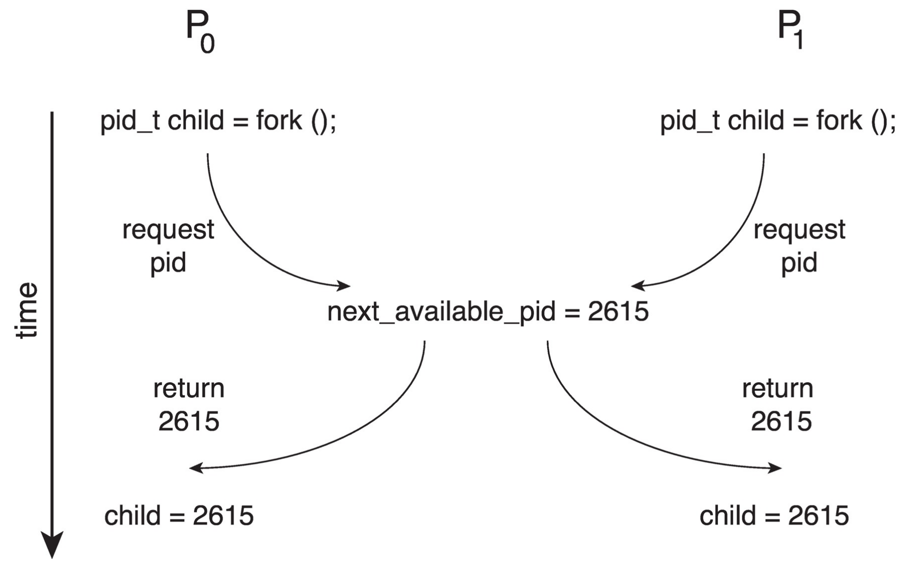


Race Condition

- 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)



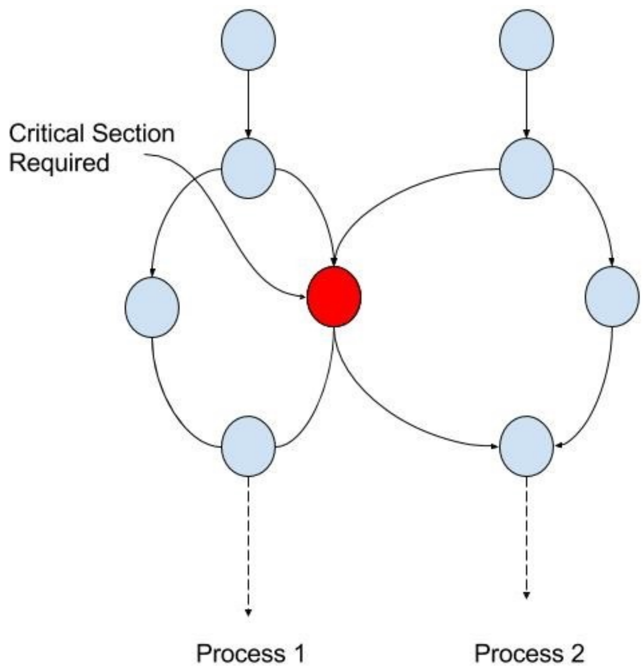
Race Condition (Cont.)



- Unless there is a mechanism to prevent P_0 and P_1 from accessing the variable `next_available_pid` **the same pid** could be assigned **to two different processes**!

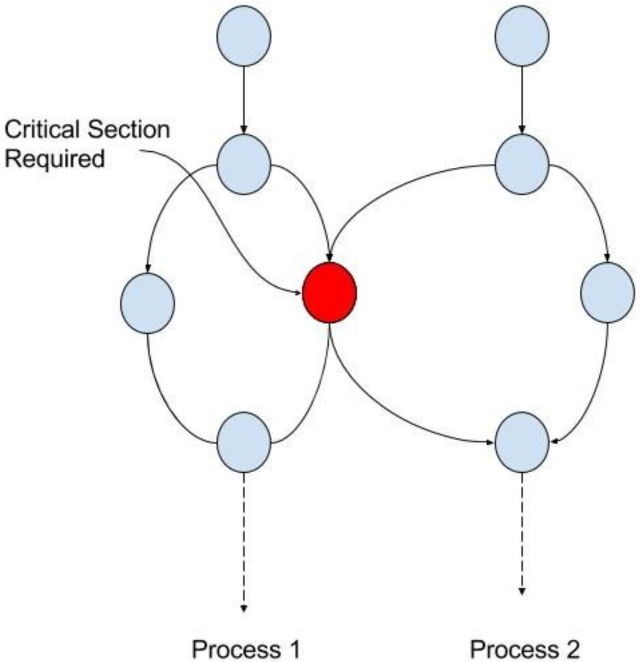
Critical Section Problem

- Consider system of n processes $\{p_0, p_1, \ldots p_{n-1}\}$
- Each process has **critical section** segment of code
  - Process may be changing common variables, updating table, writing file, etc.



Critical Section Problem

- **When one process in critical section, no other may be in its critical section.**
- Critical section problem is to design protocol to solve this.



Critical Section

- Each process
  - must ask permission to enter critical section in **entry section**,
  - may follow critical section with **exit section**,
  - then **remainder section**.
- General structure of process $\boldsymbol{P_i}$

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

Requirements for solution to critical-section problem

1. **Mutual Exclusion**

2. **Progress**

3. **Bounded Waiting**

1- Mutual Exclusion

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

- No two processes simultaneously in critical region.

2- 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 process that will enter the critical section next cannot be postponed indefinitely.

- No process running outside its critical region may block another process.

3- 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.
  - Assume that each process executes at a nonzero speed
  - No assumption concerning **relative speed** of the n processes

- **No process must wait forever to enter its critical region.**

Interrupt-based Solution

- Entry section: disable interrupts
- Exit section: enable interrupts
- Will this solve the problem?
  - What if the critical section is code that runs for an hour?
    - Can some processes starve -- never enter their critical section.
  - What if there are two CPUs?

Course Logistics

- Present Phase1 during this week
- You do Phase2 in groups of two
  - Talk to me if you did not submit Phase1 (three students)
- No delayed submission for HW3 after Friday
- Midterm is next week
  - Use standard description of fork in the source book/slides
  - TA class to solve example exam questions

Software Solution 1

- **Two process solution.**
- **Assume that the load and store machine-language instructions are atomic; that is, cannot be interrupted.**
- The two processes share one variable:
  - int ***turn***;
  - ***turn*** indicates whose turn it is to enter the critical section.

Algorithm for Process P_i

```
while (true){
        while (turn = = j);

        /* critical section */

        turn = j;

        /* remainder section */

}
```

Algorithm for P_0 and P_1

Initially turn = 0

```
while (TRUE) {
    while (turn != 0)        /* loop */ ;
    critical_region( );
    turn = 1;
    noncritical_region( );
}
```

(a)

```
while (TRUE) {
    while (turn != 1)        /* loop */ ;
    critical_region( );
    turn = 0;
    noncritical_region( );
}
```

(b)

(a) Process 0.

(b) Process 1.

Correctness of the Software Solution

- **Mutual exclusion is preserved**
  - P_i enters critical section only if:

    turn = i
  - **turn cannot be both 0 and 1 at the same time**
- It ***wastes*** CPU time
  - So we should avoid busy waiting as much as we can.
- **Can be used** only when the waiting period is expected to be **short**.

Correctness of the Software Solution (cont.)

- However there is a problem in the above approach!
  - What about the **Progress requirement**?
  - What about the **Bounded-waiting requirement**?

Correctness of the Software Solution (cont.)

- ***P_0 leaves its critical region*** and sets turn to 1, enters its non-critical region.
- ***P_1 enters its critical region***, sets turn to 0 and leaves its critical region.
- ***P_1 enters its non-critical region***, quickly finishes its job and goes back to the while loop.
- Since turn is 0, process 1 ***has to wait*** for process 0 to finish its non-critical region so that it can enter its critical region.
- This violates the **second condition (progress)** of providing mutual exclusion.

Initially turn = 0

P_0

```
while (TRUE) {
    while (turn != 0)
    critical_region( );
    turn = 1;
    noncritical_region( );
}
```

P_1

```
while (TRUE) {
    while (turn != 1)
    critical_region( );
    turn = 0;
    noncritical_region( );
}
```

How About this solution?

```
//Algorithm for Pi
while (true){

    turn = i;
    while (turn == j);

    /* critical section */

    turn = j;

    /* remainder section */

}
```

How About this solution?

```
//Algorithm for P0
while (true){

  turn = 0;
  while (turn == 1);

 /* critical section */

  turn = 1;

 /* remainder section */

}
```

```
//Algorithm for Pi
while (true){

  turn = 1;
  while (turn == 0);

 /* critical section */

  turn = 0;

 /* remainder section */

}
```

No mutual exclusion

Peterson’s Solution

- The previous solution solves the problem of one process blocking another process while its outside its critical section.

- Peterson’s Solution is a neat solution with busy waiting, that defines the procedures for entering and leaving the critical region.

Peterson’s Solution (cont.)

- 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;
  - boolean flag[2]

Peterson's Solution (cont.)

- 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!

Algorithm for Process P_i

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

        /* critical section */

        flag[i] = false;

        /* remainder section */

}
```

Correctness of Peterson's Solution

- Provable that the three CS requirement are met:
  1. Mutual exclusion is preserved

     P_i enters CS only if:

     either flag[j] = false or turn = I

  2. Progress requirement **is satisfied**

  3. Bounded-waiting requirement **is met**

Peterson's Solution

```
//P0
while (true){
   flag[0] = true;
   turn = 1;
    while (flag[1] && turn == 1);
    /* critical section */
    flag[0] = false;
    /* remainder section */
}
```

```
//P1
while (true){
   flag[1] = true;
   turn = 0;
    while (flag[0] && turn == 0);
    /* critical section */
    flag[1] = false;
    /* remainder section */
}
```

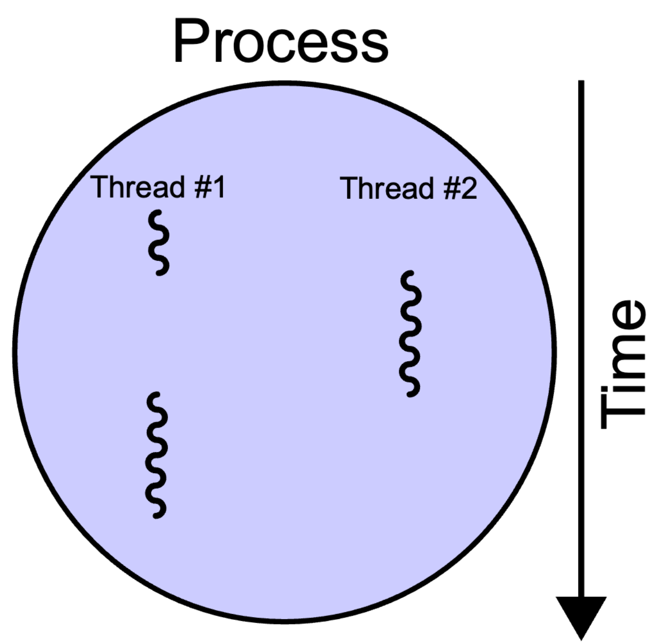
Peterson's Solution and Modern Architecture

- 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.**
- Understanding why it **will not work is useful for better understanding race conditions.**

Peterson’s Solution and Modern Architecture

- For **single-threaded this is ok** as the result will always be the same.

- For multithreaded the **reordering may produce inconsistent or unexpected results**!



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
```

Modern Architecture Example (cont.)

- However, since the variables `flag` and `x` **are independent** of each other, the instructions:

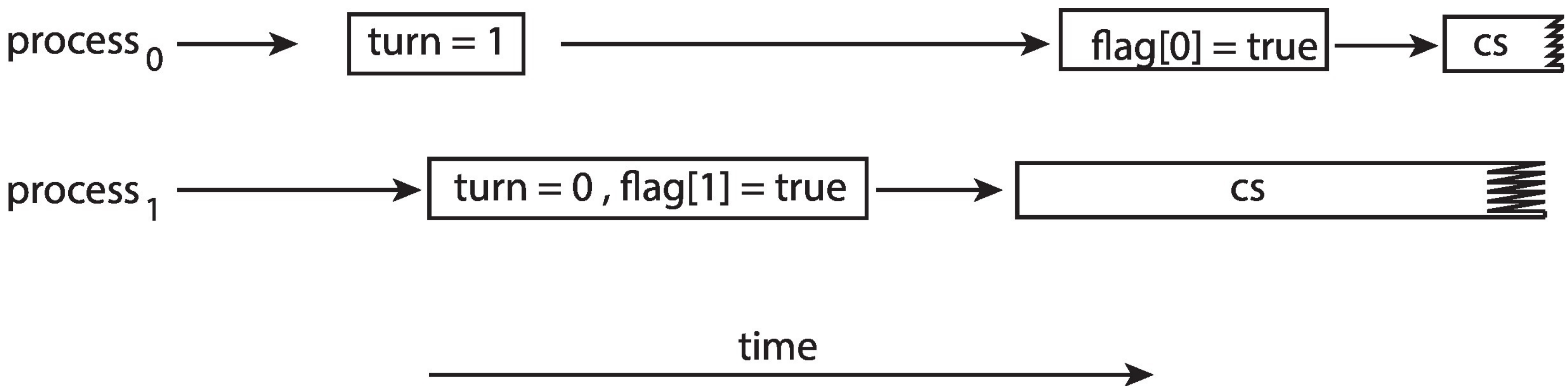
```
flag = true;
 x = 100;
```

  for Thread 2 may be reordered

- **If this occurs, the output may be 0!**

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 Barrier**.