OPERATING SYSTEMS

Mutual Exclusion and Synchronization

Chandravva Hebbi
Department of Computer Science

OPERATING SYSTEMS

Software Approaches

Chandravva Hebbi

Department of Computer Science

Slides Credits for all PPTs of this course

- The slides/diagrams in this course are an **adaptation**, **combination**, and **enhancement** of material from the following resources and persons:

1. Slides of Operating System Concepts, Abraham Silberschatz, Peter Baer Galvin, Greg Gagne - 9th edition 2013 and some slides from 10th edition 2018
2. Some conceptual text and diagram from Operating Systems - Internals and Design Principles, William Stallings, 9th edition 2018
3. Some presentation transcripts from A. Frank – P. Weisberg
4. Some conceptual text from Operating Systems: Three Easy Pieces, Remzi Arpaci-Dusseau, Andrea Arpaci Dusseau

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
- Illustration of the 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 */
}
```

Race Condition

- **counter++** could be implemented as

  register1 = counter
  register1 = register1 + 1
  counter = register1

- **counter--** could be implemented as

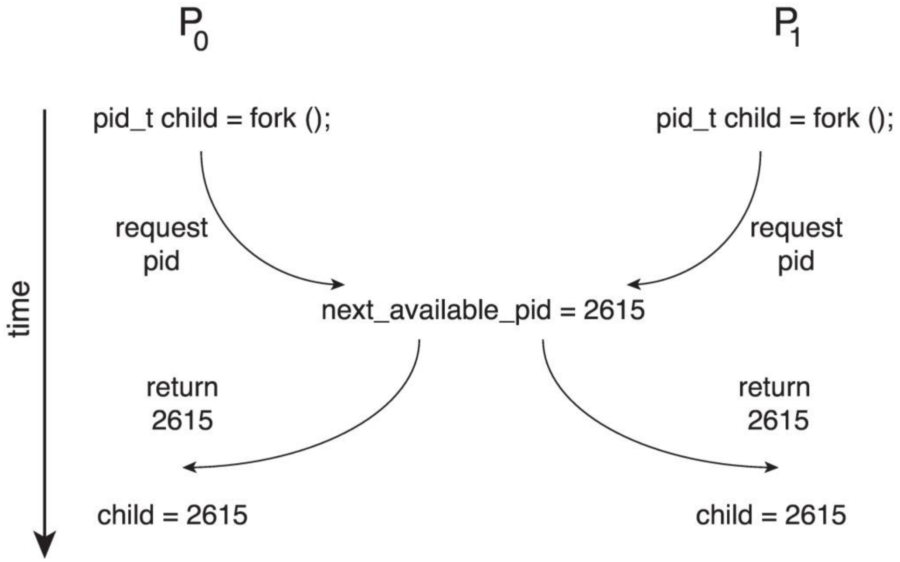
  register2 = counter
  register2 = register2 - 1
  counter = register2

- 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 (Another Example)

- 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)



- 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
  - When one process in critical section, no other may be in its critical section
- ***Critical section problem*** is to design protocol to solve this
- Each process must ask permission to enter critical section in **entry section**, may follow critical section with **exit section**, then **remainder section**

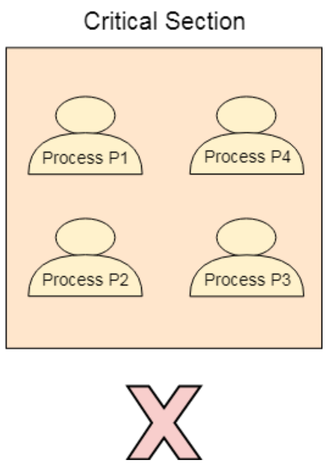
```
do {
    entry section
        critical section
    exit section
        remainder section
} while (true);
```

- General structure of process P_i

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

Solution to Critical-Section Problem

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



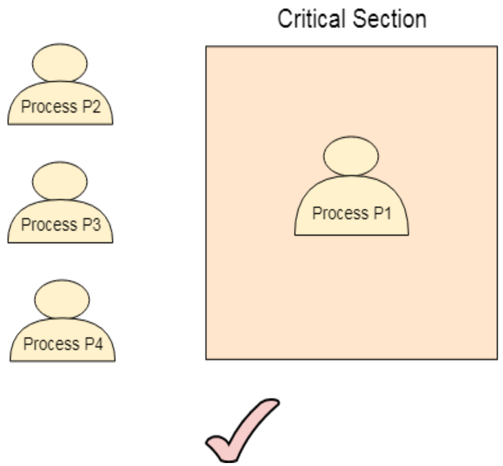


Image Src: https://www.javatpoint.com/os-critical-section-problem

Solution to Critical-Section Problem

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 processes that will enter the critical section next cannot be postponed indefinitely
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

Two approaches depending on if kernel is preemptive or non-preemptive

- **Preemptive** – allows preemption of process when running in kernel mode.
- Preemptive kernels are difficult to design on SMP architectures
- **Non-preemptive** – runs until exits kernel mode, blocks, or voluntarily yields CPU
  - Essentially free of race conditions in kernel mode
  - It is free from race conditions on kernel data structures
- Preemptive kernel may be more responsive. Suitable for real-time systems.

Peterson's Solution

- Software-based solution to the Critical Section problem.
- Good algorithmic description of solving the problem
- Assume that the **load** and **store** machine-language instructions are atomic; that is, cannot be interrupted
- **Peterson's Solution** restricted to two process.
- Pi and Pj are two process, j=1-i
- Peterson solution requires two processes share two data items:
  - 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!

Algorithm for Process Pi

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

The structure of process P_i in Peterson's solution

Software Solution

- To prove solution is correct, we need show
- Mutual exclusion is preserved

  P_i enters critical section only if:

  turn = i

and **turn** cannot be both 0 and 1 at the same time

- What about the Progress requirement?
- What about the Bounded-waiting requirement?

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

Correctness of Peterson's Solution

To prove properties 2 and 3

- we note that a process *Pi* can be prevented from entering the critical section only if it is stuck in the while loop with the condition
  - flag[j] == true and turn == j; this loop is the only one possible.
- If *Pj* is not ready to enter the critical section, then flag[j] == false, and *Pi* can enter its critical section.
- If *Pj* has set flag[j] to true and is also executing in its while statement, then either turn == i or turn == j. If turn == i, then *Pi* will enter the critical section.
- If turn == j, then *Pj* will enter the critical section.
- once *Pj* exits its critical section, it will reset flag[j] to false, allowing *Pi* to enter its critical section.
- If *Pj* resets flag[j] to true, it must also set turn to i.
- Thus, since *Pi* does not change the value of the variable turn while executing the while statement.
- *Pi* will enter the critical section (progress) after at most one entry by *Pj* (bounded waiting).

THANK YOU

Chandravva Hebbi
Department of Computer Science Engineering
chandravvahebbi@pes.edu