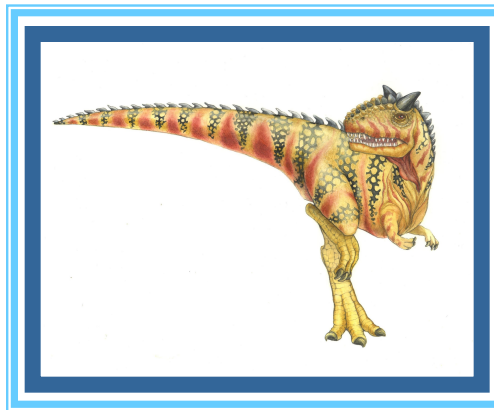


Chapter 6: Synchronization Tools





Outline

- Background
- The Critical-Section Problem
- Peterson's Solution
- Hardware Support for Synchronization
- Mutex Locks
- Semaphores
- Liveness





Objectives

- Describe the critical-section problem and illustrate a race condition
- Illustrate hardware solutions to the critical-section problem using memory barriers, compare-and-swap operations, and atomic variables
- Demonstrate how mutex locks, semaphores, monitors, and condition variables can be used to solve the critical section problem
- Evaluate tools that solve the critical-section problem in low-, Moderate-, and high-contention scenarios





Background

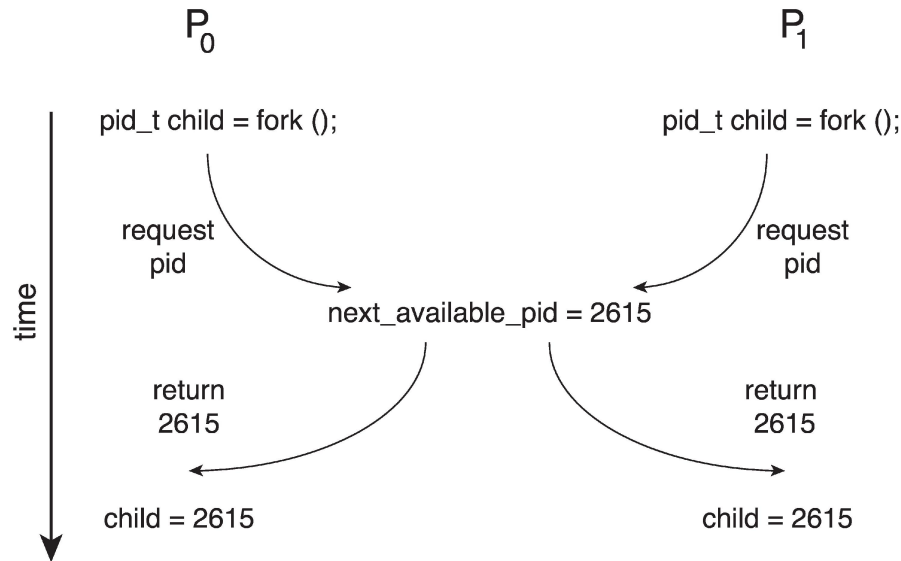
- Processes can execute concurrently
 - May be interrupted at any time, partially completing execution
- Concurrent access to shared data may result in data inconsistency
- Maintaining data consistency requires mechanisms to ensure the orderly execution of cooperating processes
- We illustrated in chapter 4 the problem 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)



- 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 $\{p_0, p_1, \dots, p_{n-1}\}$
- 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**





Critical Section

- General structure of process P_i

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





Critical-Section Problem (Cont.)

Requirements for 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
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
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





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;**
- The variable **turn** indicates whose turn it is to enter the critical section
- initially, the value of **turn** is set to *i*





Algorithm for Process P_i

```
while (true) {  
  
    while (turn == j);  
  
    /* critical section */  
  
    turn = j;  
  
    /* remainder section */  
  
}
```





Correctness of the Software Solution

- 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?





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;**
 - **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 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 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.
- 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

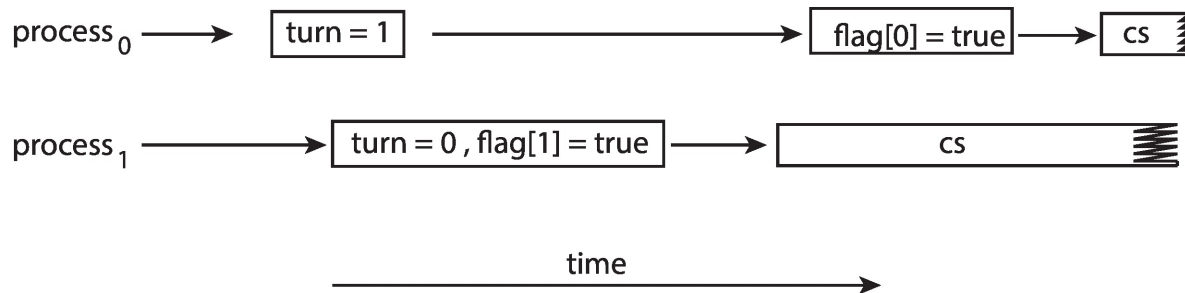
- If this occurs, the output may be 0!





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**.





Memory Barrier

- **Memory model** 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.
- 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.
- Therefore, even if instructions were reordered, the memory barrier ensures that the store operations are completed in memory and visible to other processors before future load or store operations are performed.





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
- We will look at two forms of hardware support:
 1. Hardware instructions
 2. Atomic variables





Hardware Instructions

- Special hardware instructions that allow us to either *test-and-modify* the content of a word, or to *swap* the contents of two words atomically (uninterruptedly.)
 - **Test-and-Set** instruction
 - **Compare-and-Swap** instruction





The test_and_set Instruction

- 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);
```

- Does it solve the critical-section problem?





The compare_and_swap Instruction

■ 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**
- Set the variable **value** the value of the passed parameter **new_value** but only if ***value == expected** is true. That is, the swap takes place only under this condition.





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 */  
}
```

- Does it solve the critical-section problem?





Mutex Locks

- Previous solutions are complicated and generally inaccessible to application programmers
- OS designers build software tools to solve critical section problem
- Simplest is mutex lock
 - Boolean variable indicating if lock is available or not
- Protect a critical section by
 - First **acquire()** a lock
 - Then **release()** the lock
- Calls to **acquire()** and **release()** must be **atomic**
 - Usually implemented via hardware atomic instructions such as compare-and-swap.





Solution to CS Problem Using Mutex Locks

```
while (true) {  
    acquire lock  
  
    critical section  
  
    release lock  
  
    remainder section  
}
```





Semaphore

- Synchronization tool that provides more sophisticated ways (than Mutex locks) for processes to synchronize their activities.
- Semaphore S – integer variable
- Can only be accessed via two indivisible (atomic) operations
 - **wait()** and **signal()**
 - 4 Originally called **P()** and **V()**
- Definition of the **wait()** operation

```
wait(S) {  
    while (S <= 0)  
        ; // busy wait  
    S--;  
}
```

- Definition of the **signal()** operation

```
signal(S) {  
    S++;  
}
```





Semaphore (Cont.)

- **Counting semaphore** – integer value can range over an unrestricted domain
- **Binary semaphore** – integer value can range only between 0 and 1
 - Same as a **mutex lock**
- Can implement a counting semaphore S as a binary semaphore
- With semaphores we can solve various synchronization problems





Liveness

- Processes may have to wait indefinitely while trying to acquire a synchronization tool such as a mutex lock or semaphore.
- Waiting indefinitely violates the progress and bounded-waiting criteria discussed at the beginning of this chapter.
- **Liveness** refers to a set of properties that a system must satisfy to ensure processes make progress.
- Indefinite waiting is an example of a liveness failure.





Liveness

- **Deadlock** – two or more processes are waiting indefinitely for an event that can be caused by only one of the waiting processes
- Let **S** and **Q** be two semaphores initialized to 1

| P_0 | | P_1 |
|------------------|--------------------------|--------------------------|
| | <code>wait(S) ;</code> | <code>wait(Q) ;</code> |
| | <code>wait(Q) ;</code> | <code>wait(S) ;</code> |
| <code>...</code> | <code>...</code> | |
| | <code>signal(S) ;</code> | <code>signal(Q) ;</code> |
| | <code>signal(Q) ;</code> | <code>signal(S) ;</code> |

- Consider if P_0 executes `wait(S)` and P_1 `wait(Q)`. When P_0 executes `wait(Q)`, it must wait until P_1 executes `signal(Q)`
- However, P_1 is waiting until P_0 execute `signal(S)`.
- Since these `signal()` operations will never be executed, P_0 and P_1 are **deadlocked**.





Liveness

- Other forms of deadlock:
- **Starvation** – indefinite blocking
 - A process may never be removed from the semaphore queue in which it is suspended
- **Priority Inversion** – Scheduling problem when lower-priority process holds a lock needed by higher-priority process
 - Solved via **priority-inheritance protocol**



End of Chapter 6

