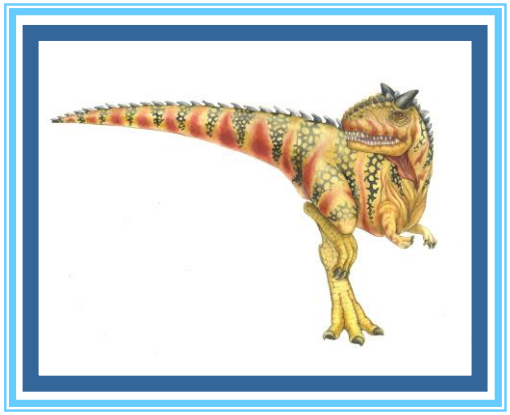
Chapter 6: Synchronization Tools

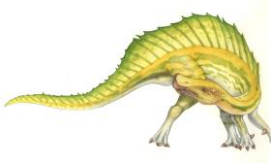


Outline

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

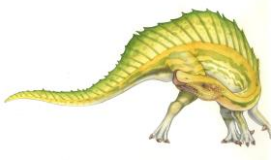
Objectives

- 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



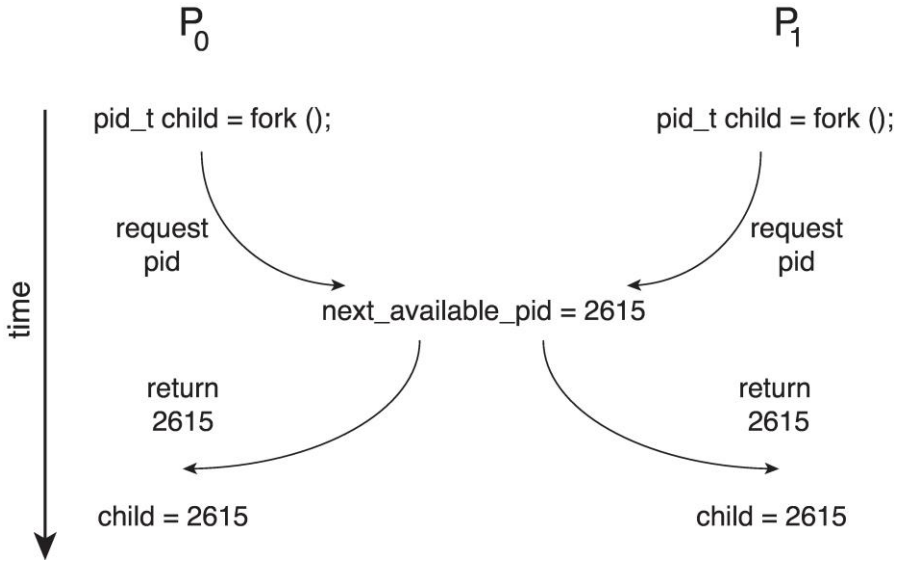
Background

- 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 led to race condition.

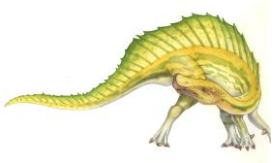


Race Condition

- 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, … 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

Interrupt-based Solution

- Entry section: disable interrupts
- Exit section: enable interrupts
- Will this solve the problem?
  - Should be good in a single-core environment
    - Guaranteed no preemption during CS
  - But what if the critical section is code that runs for an hour?
    - System's clock is updated via interrupts
  - What if it's in a multiprocessor environment?
    - Interrupts must be disabled on all processors (time consuming)

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 */

}
```

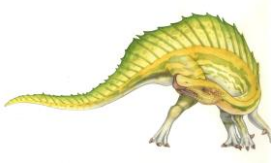
Correctness of the Software Solution

- 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?

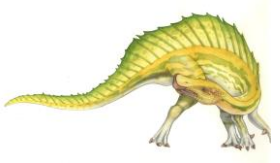


Peterson's Solution

- 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 requirements are met:
  1. Mutual exclusion is preserved

     If both P_i and P_j can enter CS:

     then **`flag[?]`** must both be **`true`**,

     but **`turn`** can't be both **`i`** and **`j`**
  2. Progress requirement is satisfied

     If a process doesn't intend to enter CS, its **`flag[?]`** will be **`false`** allowing another process to enter it
  3. Bounded-waiting requirement is met

     A process already having finished the CS but wishing to enter it again, must set **`turn`** to another process thus preventing starvation

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:

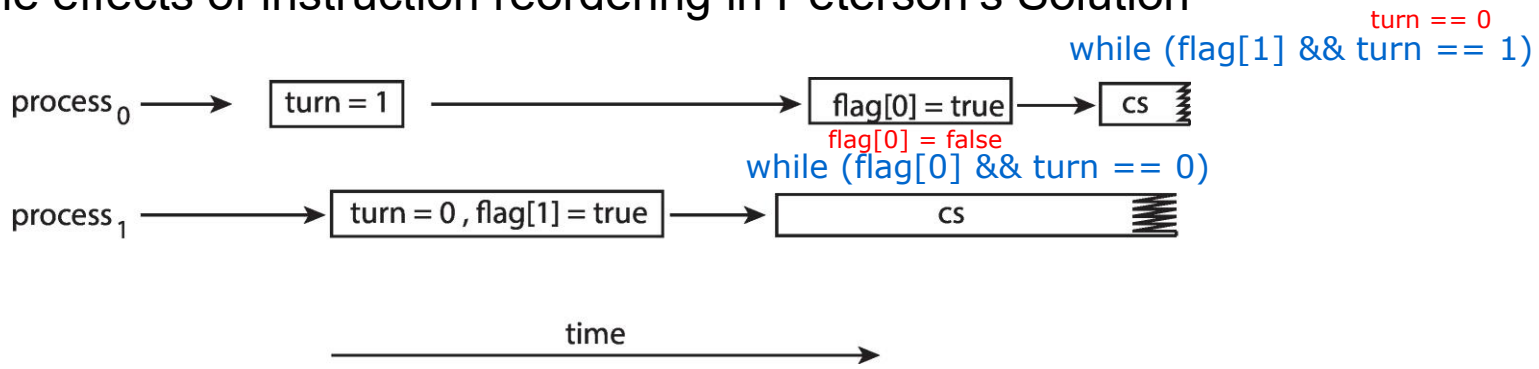
```
flag = true;
x = 100;
```

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



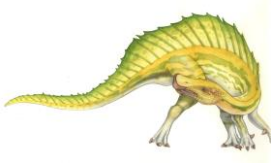
- This allows both processes to be in their critical section at the same time!
- To ensure that Peterson's solution will work correctly on modern computer architecture we must use **Memory Barrier.**

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. It is part of a hardware-based solution.

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.



Memory Barrier Example

- Returning to the example of slides 6.17 - 6.18
- We could add a memory barrier to the following instructions to ensure Thread 1 outputs 100:
- Thread 1 now performs

```
while (!flag)
   memory_barrier();
print x
```

- Thread 2 now performs

```
x = 100;
memory_barrier();
flag = true
```

- For Thread 1 we are guaranteed that that the value of `flag` is loaded before the value of `x`.
- For Thread 2 we ensure that the assignment to `x` occurs before the assignment `flag`.

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 three forms of hardware support:

1. Hardware instructions (test-and-set, compare-and-swap)
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

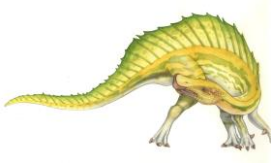
- Definition

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

*target is always set to true

return value indicates whether it's set to a true from a false (that's when a lock is grabbed anew)

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

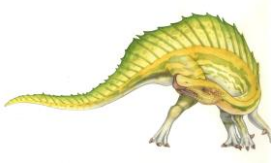


Solution Using test_and_set()

- 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

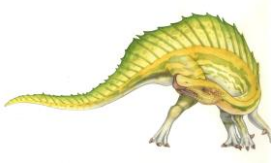
```
int compare_and_swap(int *value, int expected, int new_value)
 {
   int temp = *value;
   if (*value == expected)
       *value = new_value;
    return temp;
 }
```

Always returns the previous value

If passed `expected` is same as previous value, value gets updated (grabbed the lock, which could hold different values, not like in test-and-set)

If previous value is different from `expected`, it means another process is holding the lock, and no swap

- 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.

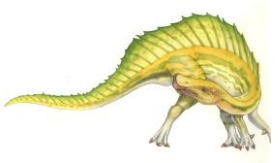


Solution using compare_and_swap

- 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?



Bounded-waiting with compare-and-swap

```
boolean waiting[n]; // all set to false, n: # of processes
int lock; // initialized to 0
…
while (true) { // this is for process Pi
   waiting[i] = true;
   key = 1;
   while (waiting[i] && key == 1) // to CS only when waiting[i]==false or key==0
      key = compare_and_swap(&lock,0,1); Key reset to 0 only when lock is available and grabbed
   waiting[i] = false; // I'm not waiting anymore
   /* critical section */
   j = (i + 1) % n; // I'm done, trying to pick another Pj to run
   while ((j != i) && !waiting[j]) Circular search guarantees at most n-1 turns
      j = (j + 1) % n;
   if (j == i) Not one process in need of CS, release lock
               so a future process in need gets served
      lock = 0;
   else
      waiting[j] = false; j is the first process in order waiting for the CS, let it run
   /* remainder section */
}
```

Set by another process just finished CS

Set because lock is available

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.
- For example:
  - Let **`sequence`** be an atomic variable
  - Let **`increment()`** be operation on the atomic variable **`sequence`**
  - The command:

```
increment(&sequence);
```

ensures **`sequence`** is incremented without interruption:

Atomic Variables

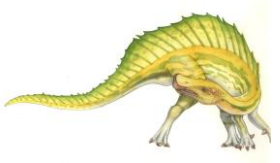
- The **`increment()`** function can be implemented as follows:

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

This guarantees *v must be changed from a previous value to +1

When will it be stuck in the do-while loop?
Suppose *v is 5 originally, gets decreased elsewhere (4) after temp=*v, then CAS won't update, and the inequality holds. Do-while forces temp to be updated to 4 and +1 will be tried again (*v ends up being 5 which is correct)

The key is the +1 operation will never be broken up (which may result in *v being a 6)



Mutex Locks

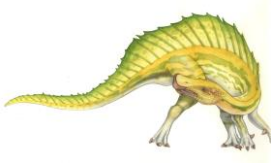
- Previous solutions are complicated and generally inaccessible to application programmers
- OS designers build software tools to solve critical section problem
- Simplest is mutex lock
  - Boolean variable indicating if lock is available or not
- Protect a critical section by
  - First **`acquire()`** 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.
- But this solution requires **busy waiting**
  - This lock therefore called a **spinlock**

Solution to CS Problem Using Mutex Locks

```
while (true) {
        acquire lock

            critical section

        release lock

        remainder section
}
```

```
acquire() {
  while (!available)
    ; /* busy waiting */
  available = false;
}
```

```
release() {
  available = true;
}
```



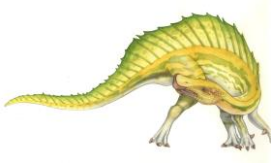
Semaphore

- 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()`**
    - Originally called **`P()`** and **`V()`** (from Dutch words, to test & to increment)
- 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**
- With semaphores we can solve various synchronization problems

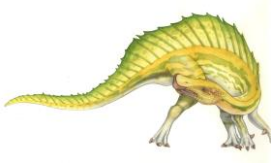
Semaphore Usage Example

- Solution to the CS Problem
  - Create a semaphore "**mutex**" initialized to 1

```
wait(mutex);
    CS
signal(mutex);
```

- Consider P_1 and P_2 that with two statements S_1 and S_2 and the requirement that S_1 to happen before S_2
  - Create a semaphore "**synch**" initialized to 0

```
P1:
    S1;
    signal(synch);
P2:
    wait(synch);
    S2;
```

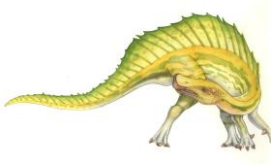


Semaphore Implementation

- Must guarantee that no two processes can execute the **`wait()`** and **`signal()`** on the same semaphore at the same time
- Thus, the implementation becomes the critical section problem where the **`wait`** and **`signal`** code are placed in the critical section
- Could now have **busy waiting** in critical section implementation
  - But implementation code is short
  - Little busy waiting if critical section rarely occupied
- Note that applications may spend lots of time in critical sections and therefore this is not a good solution

Semaphore Implementation with no Busy waiting

- With each semaphore there is an associated waiting queue
- Each semaphore has two data items:
  - Value (of type integer)
  - Pointer to next record in the list
- Two operations:
  - **block (or sleep)** – place the process invoking the operation on the appropriate waiting queue
  - **wakeup** – remove one of processes in the waiting queue and place it in the ready queue



Implementation with no Busy waiting (Cont.)

- Waiting queue

```
typedef struct {
   int value;
   struct process *list;
} semaphore;
```

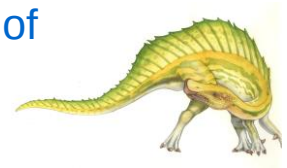
All processes waiting on the semaphore, added via wait(), removed via signal()

Implementation with no Busy waiting (Cont.)

```
wait(semaphore *S) {
    S->value--;
    if (S->value < 0) {
        add this process to S->list;
        sleep(); // suspends the process
    }
}
signal(semaphore *S) {
    S->value++;
    if (S->value <= 0) { // there are sleeping processes
        remove a process P from S->list;
        wakeup(P);
    }
}
```

Notice that in this implementation, semaphore values may be negative, whereas semaphore values are never negative under the classical definition of semaphores with busy waiting.

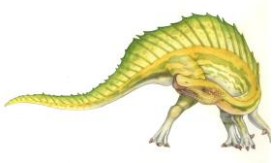
If a semaphore value is negative, its magnitude is the number of processes waiting on that semaphore



Problems with Semaphores

- Incorrect use of semaphore operations:
  - **`signal(mutex)`** …. **`wait(mutex)`**
    mutex could be 2 - two processes could be in CS at same time
  - **`wait(mutex)`** … **`wait(mutex)`**
    mutex could be -1, a process will block permanently on the second wait()
  - Omitting of **`wait(mutex)`** and/or **`signal(mutex)`**

    Some combination of the above
- These – and others – are examples of what can occur when semaphores and other synchronization tools are used incorrectly.
- There are more polished solutions such as Monitors which is not covered here



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 (cont')

- **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
`wait(S);`	`wait(Q);`
`wait(Q);`	`wait(S);`
`...`	`...`
`signal(S);`	`signal(Q);`
`signal(Q);`	`signal(S);`

- 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 (cont')

- Other forms of deadlock:
- **Priority Inversion** – Scheduling problem when lower-priority process holds a lock needed by higher-priority process
  - Solved via **priority-inheritance protocol**

Priority-inheritance protocol: All processes that are accessing resources needed by a higher-priority process inherit the higher priority until they are finished with the resources in question.

When they are finished, their priorities revert to their original values.

End of Chapter 6