Chapter 6: Synchronization Tools

Objectives

- To introduce the critical-section problem, whose solutions can be used to ensure the consistency of shared data.

- To illustrate a race condition

- To present both software and hardware solutions of the critical-section problem.

    - 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.

- To explore several tools that are used to solve process synchronization problems.

Contents

Background

- CPU scheduler switches rapidly between processes to provide concurrent execution. This means that one process may only partially complete execution before another process is scheduled.
- In parallel execution, instruction streams representing different processes execute simultaneously on separate processing cores.

Concurrent or Parallel execution can contribute to issues involving the integrity of data shared by several processes.

In this chapter, we discuss various mechanisms to ensure the orderly execution of cooperating processes that share a logical address space, so that data consistency is maintained.

Cooperating Processes

A **cooperating process** is one that can affect or be affected by other processes executing in the system.

Cooperating processes can:

1. either **directly share a logical address space** (that is, both code and data)
   - achieved through the use of threads
2. Or be allowed to share data **only through files or messages**.

The producer-consumer problem is a representative paradigm of many operating system functions.

A bounded buffer could be used to enable processes to share memory.

Producer–consumer problem

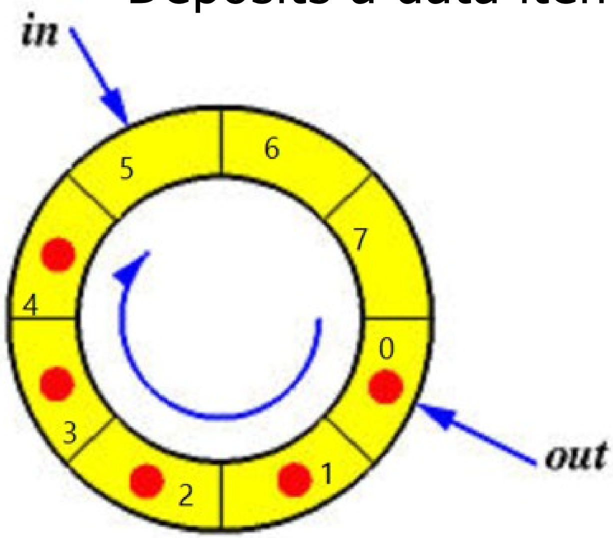
The producer-consumer problem is a model of a system consisting of cooperating sequential processes or threads, all running asynchronously and possibly sharing data.

For example,

- A compiler may produce assembly code that is consumed by an assembler. The assembler, in turn, may produce object modules that are consumed by the loader.
- In client-server paradigm - a web server produces (that is, provides) HTML files and images, which are consumed (that is, read) by the client web browser requesting the resource.

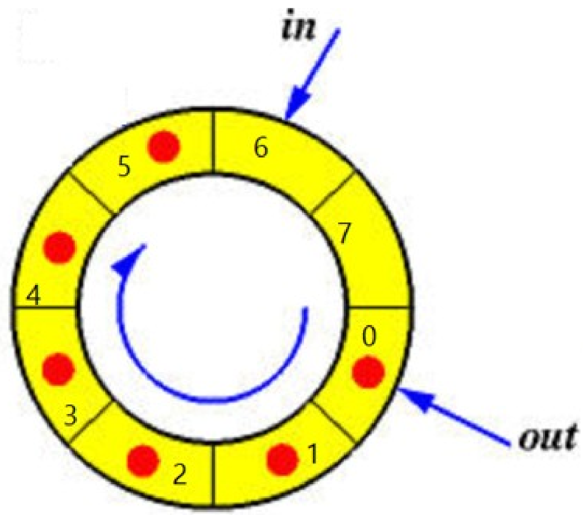
Producer

- ***in*** indicates the next available position for depositing data.
- Waits if ***in+1 = out*** (buffer Full)
- Deposits a data items into ***in*** and advances the pointer ***in*** *to* ***in+1.***

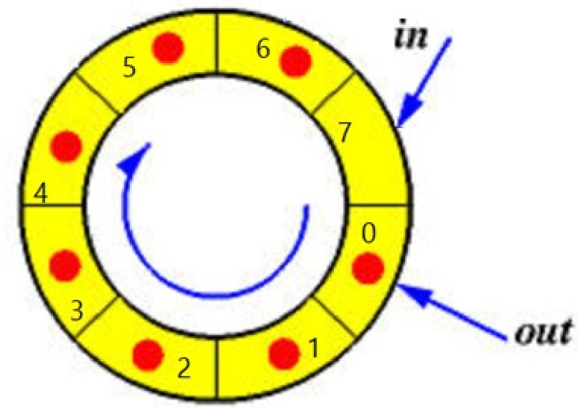


Current state:
in = 5 and **out** = 0

Producer:
in+1 = 6 **out**.
No wait needed.
produces an item
deposits in 5.
increments **in** to 5+1 =6



State:
in = 6 and **out** is still 0
(Assume consumer has not consumed any)
Producer:
in+1 = 7 **out**.
No wait needed.
produces an item
deposits in 6.
increments **in** to 6+1 =7

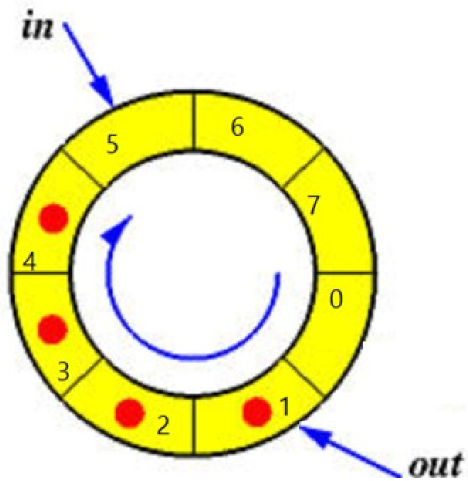


State:
in = 7; if **out** = 0,

Producer:
in+1 mod 8 =7+1 mod 8 = 0 = **out**.
Buffer full.
Producer waits.

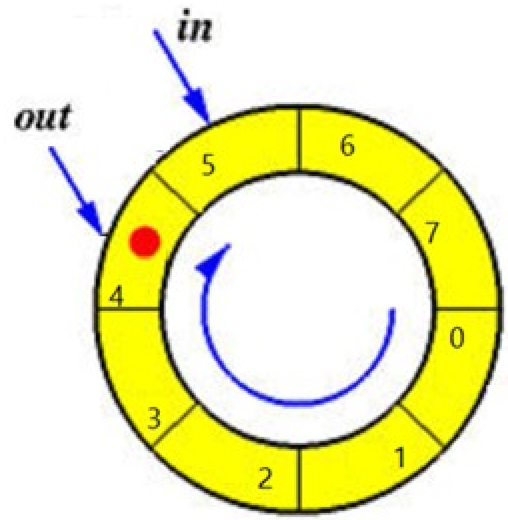
Consumer

- ***out*** indicates the position that contains the next data to be retrieved.
- Waits if ***in = out*** (buffer Empty)
- Retrieves the data item in ***out*** and advances the pointer ***out***+1.

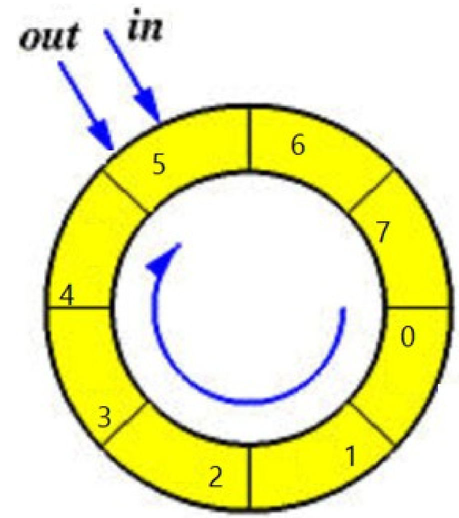


if current state is: **in** = 5 **out** = 1

in **out**. Buffer not empty.
No need to wait.
Remove item from buffer 1, consume it.
increment **out** to 1+1 =2



Continue consuming and emptying buffers while incrementing **out.**
current state: **in** = 5 **out** = 4
in **out**.
Remove item from buffer 4, increment **out** to 4+1 =5



State: **out**= 5. **in** is 5
in = **out**
Buffer empty.
Consumer waits.

First Producer-consumer problem

```
#define BUFFERSIZE 10
typedef struct {...
}item;

Item buffer[BUFFER_SIZE];
int in = 0; // next free position
int out = 0; // first full position
```

Producer Process:

```
item next_produced;
while (true) {
   /* produce an item */
   while (((in + 1)% BUFFER_SIZE) == out); // do nothing
   buffer[in] = next_produced;
   in = (in + 1) % BUFFER_SIZE;
}
```

Consumer Process

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

First bounded-buffer solution is correct, but can only use **BUFFERSIZE-1** elements

New Solution for producer-consumer problem

Suppose that we wanted to provide a solution to the producer-consumer problem that fills the buffer.

Modify the algorithm to by introducing a new integer variable *count.*

Count keeps track of the number of full buffers.

- incremented by the producer every time it adds a new item
- decremented by the consumer after it consumes an item.

- Producer and consumer routines shown next are correct separately.
- May not function correctly when executed concurrently, if not synchronized.

Producer-Consumer Problem

Integer **count** keeps track of the number of filled buffers. initially, count = 0

If count = BUFFERSZE, then
buffer is full, *n* filled buffers
If buffer is full, no place to put item
Producer waits.

If count = 0, then
buffer is empty, no filled buffer
If buffer empty, nothing to consume,
Consumer waits.

Producer

```
while (true)
{
    /* produce an item */
    //wait if buffer is full
    while(count == BUFFERSIZE);

    buffer[in] = nextproduced;
    in = (in + 1) % BUFFERSIZE;
    count++;
}
```

Consumer

```
while (true)
{
    //wait if buffer is empty
    while (count == 0); /*wait*/
    nextconsumed = buffer[out];
    out = (out + 1) % BUFFERSIZE;
    count--;
    /*consume item in nextconsumed*/
}
```

Race Condition

`count++` could be implemented as

```
register1 = count
register1 = register1 + 1
count = register1
```

`count--` could be implemented as

```
register2 = count
register2 = register2 - 1
count = register2
```

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

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

Concurrent access to shared data may result in data inconsistency

Bounded Buffer problem with use of a counter that is updated concurrently by the producer and consumer may lead to race condition.

Race Condition & Critical section

- A race condition occurs when
  - Multiple processes access and manipulate same data concurrently
  - Outcome of execution depends on the particular order in which the access takes place.
- Critical section/region
  - the segment of code where process modifying shared/common variables (tables, files)
- Critical section problem, mutual exclusion problem
  - No two processes can execute in critical sections at the same time

Processes be Synchronized

To guard against the race condition above, we need to ensure that only one process at a time can be manipulating the variable count .

To make such a guarantee, we require that the **processes be synchronized** in some way.

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**

Critical Section

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

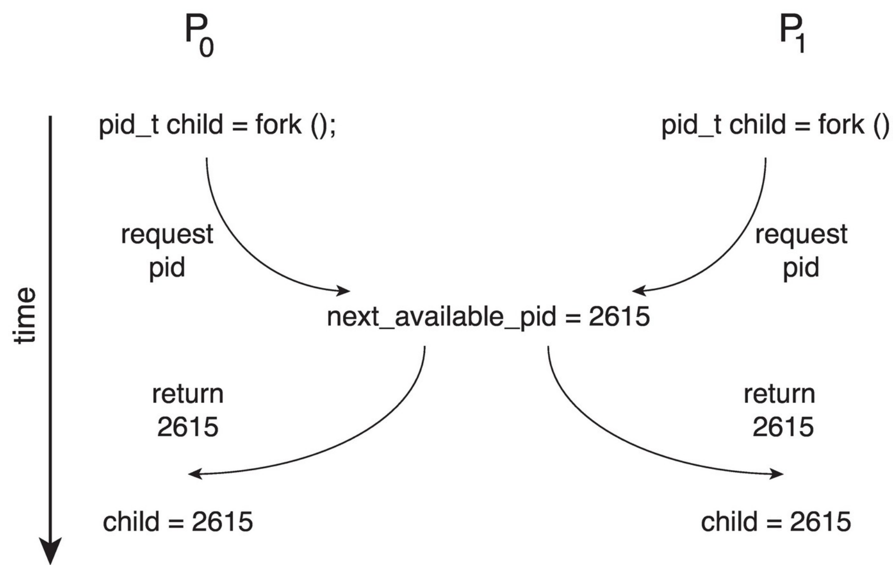
2. **Progress** - If no process is executing in its critical section and some processes wish to enter their critical sections, then only those processes that are not executing in their remainder sections can participate in the deciding which will enter its critical section next, and this selection cannot be postponed indefinitely

3. **Bounded Waiting** - There exists a bound, or limit, 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

Race condition in kernel-mode processes

Processes P_0 and P_1 are creating child processs 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!

Interrupt-based Solution

The critical-section problem could be solved simply in a single-core environment if we could prevent interrupts from occurring while a shared variable was being modified.

- Can some processes starve – never enter their critical section?
- What if the critical section is code that runs for an hour?

This solution is not as feasible in a multiprocessor environment.

- Disabling interrupts on a multiprocessor can be time consuming, since the message is passed to all the processors

Critical-Section handling in OS

Two general approaches are used to handle critical sections in operating systems: preemptive kernels and nonpreemptive kernels.

1. **Preemptive kernels**

    A preemptive kernel allows a process to be preempted while it is running in kernel mode.

    Must be carefully designed to ensure that shared kernel data are free from race conditions.

    A preemptive kernel may be more responsive, since there is less risk that a kernel-mode process will run for an arbitrarily long period before relinquishing the processor to waiting processes.

    A preemptive kernel is more suitable for real-time programming

2. **Non-preemptive kernels**

    A nonpreemptive kernel does not allow a process running in kernel mode to be preempted; a kernel-mode process will run until it exits kernel mode, blocks, or voluntarily yields control of the CPU.

    A nonpreemptive kernel is essentially free of race conditions in kernel mode as only one process is active in the kernel at a time.

Peterson's Solution

A classic software-based solution to the critical-section problem.

- Not guaranteed to work on modern architectures, but good algorithmic description of solving the critical-section problem
- Restricted to two processes that alternate execution between their critical sections and remainder sections

Assume that the `load` and `store` machine-language instructions are atomic; that is, cannot be interrupted

The two processes share two data items:

int turn; // turn indicates whose turn it is to enter the critical section

boolean flag[2] // 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

// flag[j] = *true* implies that process P_j is ready

Algorithm for Process P_i

1. To enter the critical section, process Pi first sets flag[i] to be true and then sets turn to the value j, thereby asserting that if the other process wishesto enter the critical section, it can do so.
2. If both processes try to enter at the same time, turn will be set to both i and j at roughly the same time, but one will be overwritten immediately. The eventual value of turn determines which of the two processes is allowed to enter its critical section first.

```
while (true){
       flag[i] = true;
       turn = j;
       while (flag[j] && turn == j);
       // enters CS if flag[j] == false or if turn ==I

       /* critical section */

       flag[i] = false;

       /* remainder section */
}
```

Process P_j

```
int turn;           // turn == i means Pi's turn to enter its critical section
boolean flag[2];    // flag[j] true, Pj is ready to enter its critical section

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

                /* critical section */

                flag[j] = 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`**

    - If both processes can be executing in their critical sections at the same time, then flag[0] == flag[1] == true. Not possible.
    - Value of turn can be either i or j, but cannot be both
2. Progress requirement is satisfied
3. Bounded-waiting requirement is met

    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)

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.

1. For single-threaded this is ok as the result will always be the same.
2. 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!

Effects of instruction reordering in Peterson's Solution

Process i

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

Process j

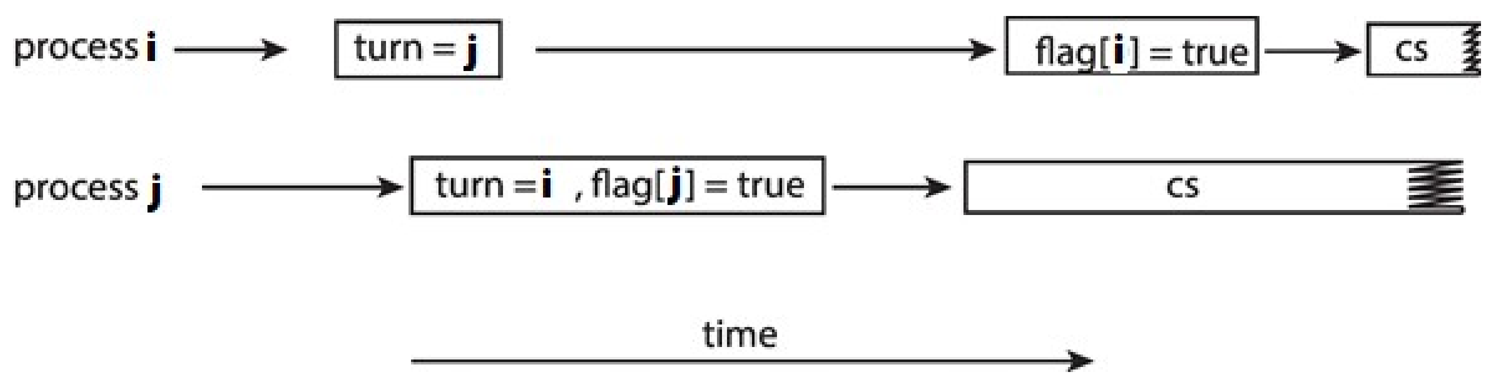
```
while (true){
        flag[j] = true;
        turn = i;
        while (flag[i] && turn == i);
        /* critical section */
        ...
}
```

```
Correct order:
flag[i] = true;
turn = j;
```

```
If reordered:
turn = j;
flag[i] = true;
```

```
Correct order:
flag[j] = true;
turn = i;
```

```
If reordered:
turn = i;
flag[j] = true;
```



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**.

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. Memory barriers

2. Hardware instructions

3. Atomic variables

Memory Barriers

System may reorder instructions. This leads to unreliable data states.

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.

In parallel computing, a **barrier** is a type of synchronization method.

A barrier for a group of threads or processes in the source code means any thread/process must stop at this point and cannot proceed until all other threads/processes reach this barrier.

Memory Barrier Example

Returning to the example of slides 6.26 - 6.27

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
```

- For Thread 1 we are guaranteed that that the value of `flag` is loaded before the value of `x`.

Thread 2 now performs

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

- For Thread 2 we ensure that the assignment to `x` occurs before the assignment `flag`.

Hardware Instructions

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

Test-and-Set instruction

Compare-and-Swap instruction

test_and_set Instruction

Definition:

```
boolean test_and_set (boolean *target)
{
        boolean rv = *target;
        *target = true;  // Sets the value of passed parameter to true
        return rv:       // Returns the original value of passed parameter
}
```

- Executed atomically
  - Lock the memory bus, loadBus lock
- Thus, if two test and set() instructions are executed simultaneously (each on a different CPU), they will be executed sequentially in some arbitrary order.
- If the machine supports the test and set() instruction, then we can implement mutual exclusion by declaring a boolean variable lock, initialized to false.

Mutual-exclusion implementation with test_and_set()

- test_snd_set() is atomic
- if lock = true, busy wait

Shared boolean variable lock:
Lock is false: no process is in critical section
Lock is true: one process is in critical section

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

1. Initially, lock = false.
2. First process passes 'false' to test_and_set()
   i. Sets lock to true, so no other process can execute
   ii. Returns the passed 'false', so this process can enter the critical section.

```
do {
      while (test_and_set(&lock))
         ; // do nothing if lock is true

             /* critical section */

       lock = false;
             /* remainder section */
   } while (true);
```

Process exits 'while loop' if returned value (of test_and_set) is false. Process then:

i. completes critical section
ii. sets lock to false, so the next process can enter critical section

compare_and_swap Instruction

1. Executes atomically
2. Returns the original value of passed parameter **`value`**
3. 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.

Mutual exclusion is provided as follows:

1. A global variable **lock** is initialized to **0**.
2. The first process that invokes `compare_and_swap()` will **set lock to 1**
3. It will then **enter its critical section**, because the original value of lock **0** was equal to the expected value of **0**.
4. Subsequent calls to `compare_and_swap()` will not succeed, because lock is now **1** which is not equal to the expected value of **0**.
5. When a process exits its critical section, it sets lock back to **0**, which allows another process to enter its critical section.

1. Does this solution satisfy mutual exclusion? Yes
2. Bounded waiting ? No.

   (Figure 6.9 algorithm in the book, satisfies all the critical-section requirements)

Solution using compare_and_swap

```
while (true){
                while (compare_and_swap(&lock, 0,
1) != 0)
                ; /* do nothing */
                /* critical section */
                lock = 0;
           /* remainder section */
      }
```

```
int compare _and_swap(int *value, int expected, int
new_value) {
         int temp = *value;

         if (*value == expected)
            *value = new_value;
      return temp;
     }
```

Note: On Intel x86 architecture, the assembly language statement `cmpxchg` is used to implement `compare_and_swap()`

Atomic Variables

Typically, instructions such as compare-and-swap are used as building blocks for other synchronization tools.

Many modern computer systems provide special hardware instructions that allow us either to test and modify the content of a word or to swap the contents of two words **atomically**—that is, as one uninterruptible unit.

One tool is an **atomic variable** that provides ***atomic* (uninterruptible) updates on basic data types such as integers and booleans.**

Most systems that support atomic variables provide special atomic data types as well as functions for accessing and manipulating atomic variables. These functions are often implemented using **compare_and_swap()** operations.

Atomic Operation

In the old days, it would be a "lock" pin on the processor(s) - an actual external pin on the chip - that would be wired to anything else that could access the memory bus, and when that pin was active, all other devices has to wait for it to become inactive before accessing the memory bus.

These days, most systems use an "exclusive cache content" method: The processor signals all it's peers that "I want this address to be exclusive in my cache", at which point all other processors will "flush and invalidate" that particular address in their caches. Then the **atomic operation** is performed in the cache, and the results available to be read by other processors only when the atomic operation is completed.

Atomic Variables - Example

Let **sequence** be an atomic variable

Let **increment()** be operation on the atomic variable **sequence**

The following ensures **sequence** is incremented without interruption:

increment(&sequence);

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));
}
```

Mutex Locks

The hardware-based solutions to the critical-section problem presented are complicated as well as generally inaccessible to application programmers.

- Instead, operating-system designers build higher-level software tools to solve the critical-section problem.
- The simplest of these tools is the **mutex lock.**
- The term ***mutex*** is short for ***mut***ual ***ex***clusion.
- We use the **mutex lock** to **protect critical sections** and thus **prevent race conditions**
- But this solution requires **busy waiting**

Busy Waiting, Spinlocks

While a process is in its critical section, any other process that tries to enter its critical section must loop continuously in the call to acquire().

Busy waiting is this continual looping wasting CPU cycles that some other process might be able to use productively.

It is a problem in a real multiprogramming system, where a single CPU core is shared among many processes.

Spinlock:
This type of mutex lock is also called a **spinlock** because the process
“spins” while waiting for the lock to become available.

Spinlocks are often identified as the locking mechanism of choice on multiprocessor systems when the lock is to be held for a short duration.

Mutex Lock Definitions

- Protect a critical section by first acquire() a lock before entering a critical section then release() the lock when it exits the critical section
  - A mutex lock has a boolean variable available whose value indicates if the lock is available or not.
  - If the lock is available, a call to acquire() succeeds, and the lock is then considered unavailable.
  - A process that attempts to **acquire an unavailable lock** is **blocked** until the lock is released.
- Calls to acquire() and release() must be **atomic**
  - Usually implemented via **hardware atomic instructions** such as **test-and-set** and **compare-and-swap.**

Solution to Critical-section Problem Using Mutex Locks

The definition of acquire():

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

The definition of release():

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

```
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.

A **semaphore** S is an integer variable that, apart from initialization, is **accessed only through** two standard **atomic operations**: wait() and signal().

Introduced by the Dutch computer scientist Dijkstra. Originally termed **`P()`** from the Dutch **`proberen`**, **`"to test"`** **`and`** V from ***verhogen,*** "to increment"

Definition of the `wait()` operation.

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

Definition of the `signal()` operation.

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

Semaphore Implementation

- All modifications to the integer value of the semaphore in the wait() and signal() operations must be executed atomically.
- That is, when one process modifies the semaphore value, no other process can simultaneously modify that same semaphore value.
- In addition, in the case of wait(S), the testing of the integer value of S ($S \leq 0$), as well as its possible modification (S--), must be executed without interruption.
- The implementation becomes the critical section problem where the wait code is placed before the critical section and signal code is placed after the critical section
- Could now have busy waiting in critical section implementation.
- Note that applications may spend lots of time in critical sections and therefore this is not a good solution

Counting Semaphores

Counting semaphore – integer value can range over an unrestricted domain.

Can be used to control access to a given resource consisting of a finite number of instances:

- The semaphore is initialized to the number of resources available.
- Each process that wishes to use a resource performs a wait() operation on the semaphore (thereby decrementing the count).
- When a process releases a resource, it performs a signal() operation (incrementing the count).
- When the count for the semaphore goes to 0, all resources are being used.
- After that, processes that wish to use a resource will block until the count becomes greater than 0

Binary Semaphore

Binary semaphore – integer value can range only between 0 and 1

- Similarly to **mutex locks**

  On systems that do not provide mutex locks, binary semaphores can be used instead for providing mutual exclusion.

  Solution to the Critical Section Problem:

  Create a semaphore "**`mutex`**" initialized to 1

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

Semaphore Usage Example

With semaphores we can solve various synchronization problems.

Problem:

Consider two concurrently running processes: P_1 with a statements S_1, and P_2 that with a statements S_2 and the requirement that S_1 to happen before S_2.

Solution:

Create to let P_1 and P_2 share a common semaphore "**synch**", initialized to 0.
Insert the wait() in P_2 and signal() in P_1.

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

Because synch is initialized to 0, *P*2 will execute *S*2 only after *P*1 has invoked signal(synch), which is after statement *S*1 has been executed.

Semaphore Implementation with No Busy Waiting

Avoids busy waiting by temporarily putting the waiting process to sleep and then awakening it once the lock becomes available.

- With each semaphore there is an associated waiting queue

Each entry in a waiting queue has two data items:

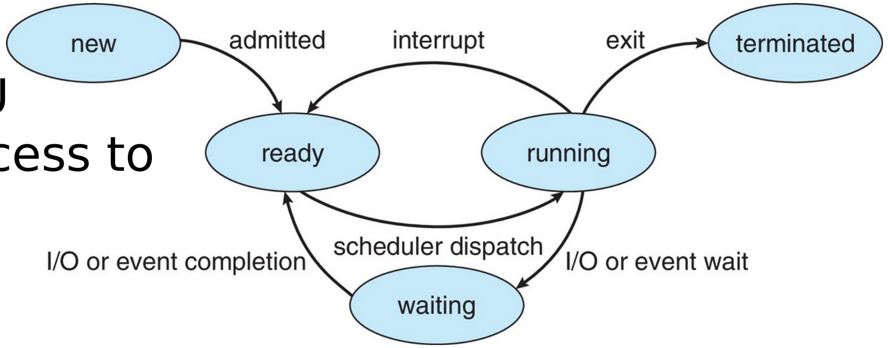
value (of type integer)

pointer to next record in the list

Semaphores: Process Blocking and Awakening

Suspending a process:

- When a process executes the wait() operation and finds that the semaphore value is not positive, it must wait.
- However, rather than engaging in busy waiting, the process can suspend itself.
- The suspend operation places a process into a waiting queue associated with the semaphore, and the state of the process is switched to the waiting state.
- Then control is transferred to the CPU scheduler, which selects another process to execute.



Restarting the process:

- A process that is suspended, waiting on a semaphore S, should be restarted when some other process executes a signal() operation.
- The process is restarted by a wakeup() operation, which changes the process from the waiting state to the ready state.
- The process is then placed in the ready

Implementation with no Busy waiting (Cont.)

block – place the process invoking the operation on the appropriate waiting queue

```
wait(semaphore *S) {
    S->value--;
    if (S->value < 0) {
        add this process to S->list;
        sleep();
    }
}
```

Each semaphore has an integer value and a list of process list. Entry in a waiting list contains semaphore value and a pointer to next record in the list.

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

wakeup – remove one of processes in the waiting queue and place it in the ready queue

```
signal(semaphore *S) {
    S->value++;
    if (S->value <= 0) {
        remove a process P from S->list;
        wakeup(P);
    }
}
```

The sleep() and wakeup() operations are provided by the operating system as basic system calls.

Semaphore values

Semaphore values are never negative under the classical definition of
Semaphores with busy waiting.

In Semaphore implementation with no busy waiting, semaphore values may be negative. If a semaphore value is negative, its magnitude is the number of processes waiting on that semaphore.

Semaphore order

- Although semaphores provide a convenient and effective mechanism for process synchronization, using them incorrectly can result in timing errors that are difficult to detect, since these errors happen only if particular execution sequences take place, and these sequences do not always occur.

- To illustrate how, we review the semaphore solution to the critical-section problem. All processes share a semaphore variable mutex, which is initialized to 1.
  - Each process must execute wait(mutex) before entering the critical section and signal(mutex) afterward.
  - If this sequence is not observed, two processes may be in their critical sections simultaneously.

Problems with Semaphores

1. Incorrect order in which the wait() and signal() operations on the semaphore mutex

   process interchanges the operations

   `signal (mutex)` … **`critical section`** … **`wait (mutex)`**

   several processes maybe executing in their critical sections simultaneously, violating the mutual-exclusion requirement

2. **`wait (mutex)`** … **`wait (mutex)`**

   a deadlock will occur.

3. process omits the wait(mutex), or the signal(mutex), or both

   Omitting of **`wait (mutex)`** and/or **`signal (mutex)`**

   either mutual exclusion is violated or a deadlock will occur.

These – and others – are examples of what can occur when semaphores and other synchronization tools are used incorrectly.

Monitors

A ***monitor type*** is an ADT that includes a set of programmer-defined operations that are provided with mutual exclusion within the monitor.

An **abstract data type**—or **ADT**—

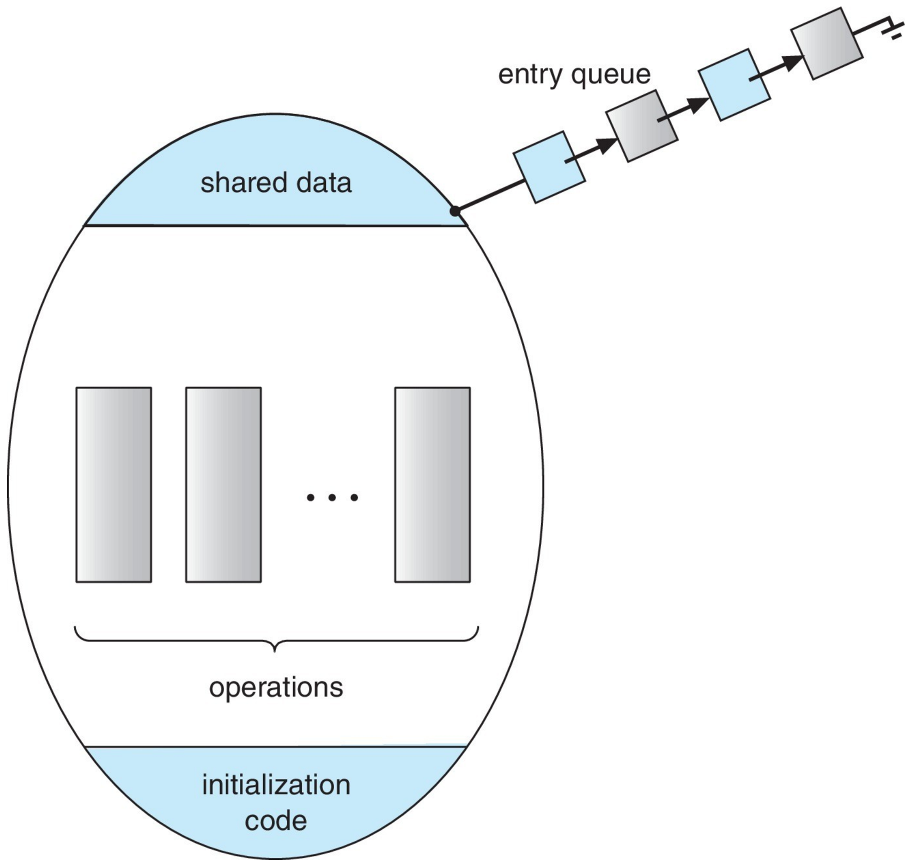
- encapsulates data with a set of functions to operate on that data
- internal variables only accessible by code within the procedure

Only one process may be active within the monitor at a time

Pseudocode syntax of a monitor:

```
monitor monitor-name
{
        // shared variable declarations
        function P1 (…) { …. }
        function P2 (…) { …. }
        function Pn (…) {……}
 initialization code (…) { … }
}
```

Schematic view of a Monitor



Monitor Type

A ***monitor type*** is an ADT that includes a set of programmer-defined operations that are provided with mutual exclusion within the monitor.

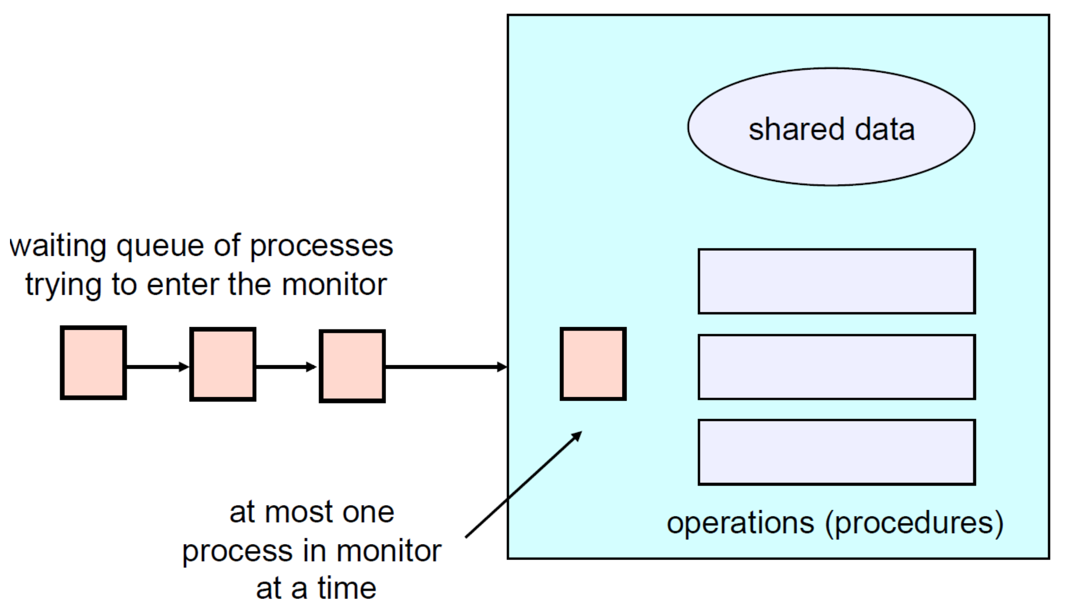
- A programming language construct that supports controlled access to shared data. It encapsulates:
  - Shared data
  - Procedures that operate on the shared data
  - Synchronization between concurrent processes that invoke those procedures (only one thread can execute any of the procedures at a time)

- A Monitor has a mutex lock and a queue
  - Processes has to acquire the lock before invoking a procedure in the monitor
  - If the lock has been acquired by a process, other requesting processes are put in the queue

Monitors (cont'd)

1. Shared data can only be accessed through procedures
2. Only one thread can enter the monitor to invoke procedures at a time
3. These rules guarantee that all threads access the shared data in a mutual exclusive way

Monitor is easy to use, but less flexible than Semaphore



Condition Variables

For modeling some synchronization schemes, additional synchronization mechanisms are provided by the condition construct.

`condition x, y;`

Two operations are allowed on a condition variable:

- **`x.wait()`** – a process that invokes the operation is suspended until **`x.signal()`**
- **`x.signal()`** – resumes one of processes (if any) that invoked **`x.wait()`**
  - If no **`x.wait()`** on the variable, then it has no effect on the variable

Condition Variables Choices

If process P invokes **`x.signal()`**, and process Q is suspended in **`x.wait()`**, what should happen next?

  Both Q and P cannot execute in parallel. If Q is resumed, then P must wait

Options include

  Signal and wait – P waits until Q either leaves the monitor or it waits for another condition

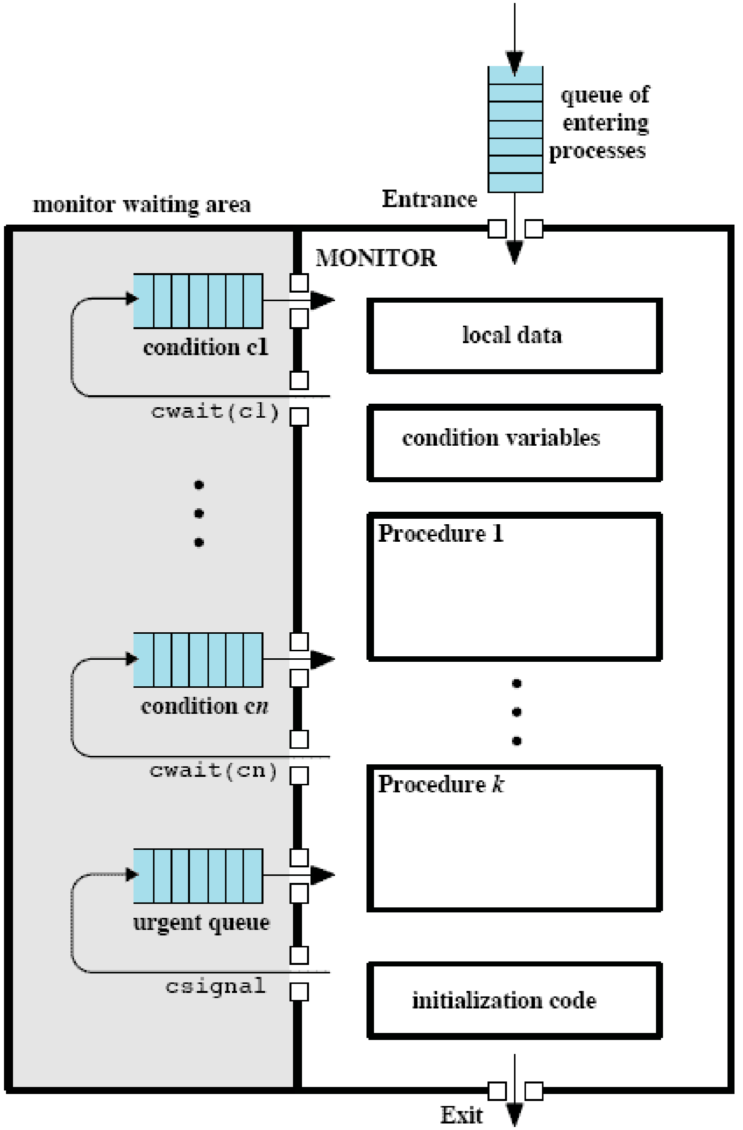
  Signal and continue – Q waits until P either leaves the monitor or it waits for another condition

  Both have pros and cons – language implementer can decide.

  A compromise implemented in languages including C#, Java

    P executing signal immediately leaves the monitor, Q is resumed

Illustration of a Monitor



A single entry point that is guarded so that only one process may be in the monitor at a time.

A process in monitor may block itself on condition x by issuing cwait(x) enters associated queue

A process in monitor detects a change in condition variable x; it issues csignal(x) that alerts the queue

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.

Deadlock

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_1 P_0

```
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**.

Lliveness problem - Priority Inversion

Priority Inversion – Scheduling problem when lower-priority process holds a lock needed by higher-priority process

Since kernel data are typically protected with a lock, the higher-priority process will have to wait for a lower-priority one to finish with the resource.

The situation becomes more complicated if the lower-priority process is preempted in favor of another process with a higher priority.

- Assume we have three processes—*L*, *M*, and *H*—whose priorities follow the order $L < M < H$.
- Assume that process *H* requires resource *R*, which is currently being accessed by process *L*.
- Ordinarily, process *H* would wait for *L* to finish using resource *R*.
- However, now suppose that process *M* becomes runnable, thereby preempting process *L*. Indirectly, a process with a lower priority—process *M*—has affected how long process *H* must wait for *L* to relinquish resource *R*.

Priority Inheritance Protocol

To prevent this from occurring, a **priority inheritance protocol** is used.

1. 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.
2. When they are finished, their priorities revert to their original values.

In the example:

- a priority-inheritance protocol would allow process *L* to temporarily inherit the priority of process *H*, thereby preventing process *M* from preempting its execution.
- When process *L* had finished using resource *R*, it would relinquish its inherited priority from *H* and assume its original priority.
- Because resource *R* would now be available, process *H* — not *M* — would run next.

End of Chapter 6