Process Synchronization

Process Synchronization

Module 6: Process Synchronization

Objectives

Background

Background -2-

Example -1-

PROCESS SYNCHRONIZATION

Producer / Consumer

Race Condition

Example -2-

Race Conditions

Critical Sections

Critical Sections -2-

Solution to Critical-Section Problem

Critical section problem

Peterson's Solution

2 process -Algorithm 1

2 process -Algorithm 2

2-PROCESS -Algorithm 2

Petersons Solution: Pseudocode

Petersons Solution: Locking: Disadvantages

Synchronization Hardware

Semaphore

Semaphore as General Synchronization Tool

Semaphores Usage

Semaphores Usage

Semaphore Implementation

Disadvantage: Basic Semaphore

Semaphore - Busy Waiting

Semaphore Implementation with no Busy waiting (Cont.)

Semaphore

Module 6: Process Synchronization

- Background
- The Critical-Section Problem
- Peterson's Solution
- Synchronization Hardware
- Semaphores
- Classic Problems of Synchronization
- Synchronization Examples

Objectives

- To introduce the critical-section problem, whose solutions can be used to ensure the consistency of shared data
- To present both software and hardware solutions of the critical-section problem

Background

- Concurrent access to shared data may result in data inconsistency →Multiple threads in a single process

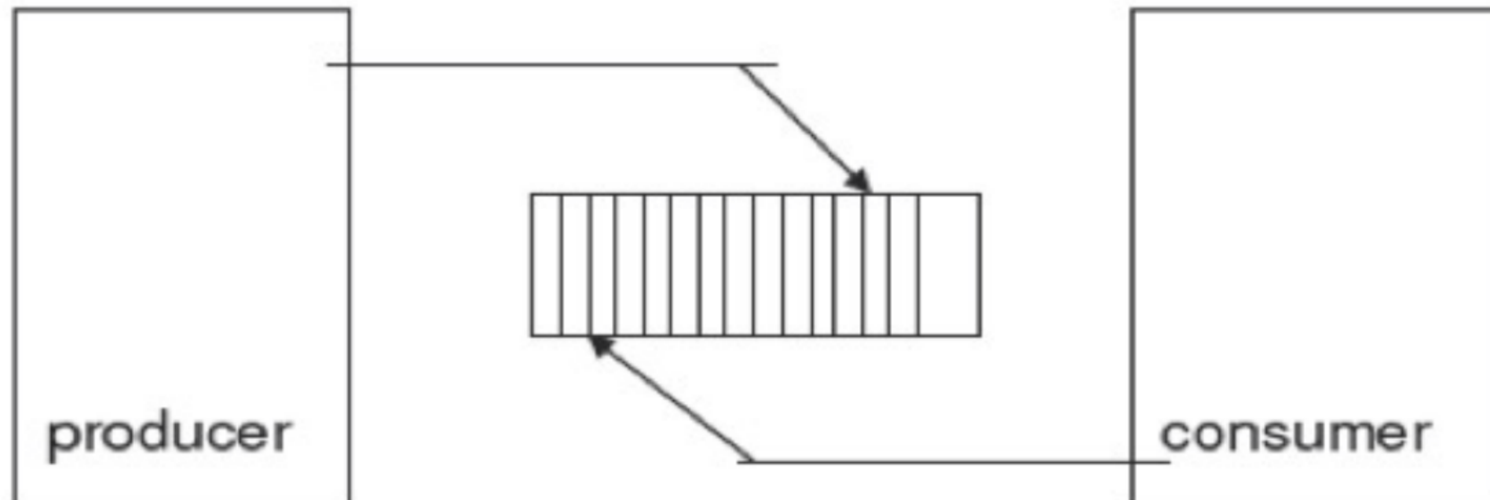
- Maintaining data consistency requires mechanisms to ensure the orderly execution of cooperating processes

Background -2-

- **Disjoint** threads use disjoint sets of variables, and do not use shared resources.
  - The progress of one thread is independent of all other threads, to which it is disjoint.
- **Non-disjoint** threads influence each other by using shared data and/or resources.
- Two possibilities:
  - **Competing** threads: compete for access to the data resources
  - **Cooperating** threads: e.g. producer / consumer model
- Without synchronization, the effect of non-disjoint parallel threads influencing each other is not predictable and not reproducible.

Example -1-

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 count that keeps track of the number of full buffers. Initially, count 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.



PROCESS SYNCHRONIZATION

- **The Producer Consumer Problem :**
- A **producer process "produces" information "consumed" by a consumer process.**
- Here are the variables needed to define the problem:

```
#define BUFFER_SIZE 10
typedef struct {
DATA data;
} item;
item buffer[BUFFER_SIZE];
int in = 0; // Location of next input to buffer
int out = 0; // Location of next removal from buffer
int count = 0; // Number of buffers currently full
```

Producer / Consumer

```
Producer :
item nextProduced;
while (true) {
          /*  produce an item and put in nextProduced  */
        while (count == BUFFER_SIZE)
                        ; // do nothing
                buffer [in] = nextProduced;
                in = (in + 1) % BUFFER_SIZE;
                count++;
}
  Consumer:
  item nextConsumed;
  while (true)  {
          while (count == 0)
                ; // do nothing
                nextConsumed =  buffer[out];
                 out = (out + 1) % BUFFER_SIZE;
               count--;
                    /*  consume the item in nextConsumed
```

http://cs.uttyler.edu/Faculty/Rainwater/COSC3355/Animations/processsync.htm

Race Conditions

- A **race condition** is where multiple processes/ threads concurrently read and write to a shared memory location and the result depends on the order of the execution.

- The part of the program, in which race conditions can occur, is called **critical section**.

Critical Sections

- A **critical section** is a piece of code that accesses a shared resource (data structure or device) that must not be concurrently accessed by more than one thread of execution.
- The goal is to provide a mechanism by which only one instance of a critical section is executing for a particular shared resource.
- Unfortunately, it is often very difficult to detect critical section bugs

Critical Sections -2-

- A Critical Section Environment contains:
  - **Entry Section** Code requesting entry into the critical section.
  - **Critical Section** Code in which only one process can execute at any one time.
  - **Exit Section** The end of the critical section, releasing or allowing others in.
  - **Remainder Section** Rest of the code AFTER the critical section.

```
do {
            Entry Section;
                critical section;
            Exit Section;
                remainder sections;
} while (true);
```

Solution to Critical-Section Problem

A solution to the critical-section problem must satisfy the following requirements:

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 processes that will enter the critical section next cannot be postponed indefinitely
3. Bounded Waiting - A bound, or limit 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

Critical section problem

- an example , a kernel data structure that maintains a List of all open files in the system, If it will be accessed simultaneously by two processes this will result in a race condition.
- Two general approaches to handle CS in OS:
  - Preemptive kernels e.g. Linux(carefully designed).
  - Nonpreemptive kernels e.g. WinXP (free from race conditions ?)

Why, then, would anyone favor a preemptive kernel over a nonpreemptive one?

- A preemptive kernel is more suitable for real-time programming.
- A preemptive kernel may be more responsive.

Peterson's Solution

- Two processes solution: **Shared Variable**
- It provides a good algorithmic description of solving the critical-section problem
- Algorithm is only for 2 processes at a time
- Processes are
  - P_0
  - P_1
- Or can also be represented as P_i and P_j ,
- i.e. j=1-i

Gary L. Peterson in 1981.

G. L. Peterson: "Myths About the Mutual Exclusion Problem", Information Processing Letters 12(3) 1981, 115–116

Because of the way modern computer architectures perform basic machine-language instructions, such as load and store, there are no guarantees that Peterson's solution will work correctly on such architectures.

Petersons Solution: Pseudocode

For Process P_i

```
do {
        flag [i] = true;        // declare your interest to enter
        turn = j;    // assume it is the other's turn-give PJ a chance
        while (flag [ j ] and turn == j) ;
                critical section
        flag [i] = false;
                remainder section
} while (1);
```

http://williamstallings.com/OS/Animation/Animations.html

Petersons Solution: Locking: Disadvantages

```
do {
    acquire lock
        critical section
    release lock
        remainder section
} while (TRUE);
```

The major disadvantage is that it only worked for two processes.

Synchronization Hardware

- Many systems provide hardware support for 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

Semaphore

Dutch computer scientist Edsger Dijkstra

- **Semaphore: a synchronization tool,** A flag used to indicate that a routine cannot proceed if a shared resource is already in use by another routine.
- Semaphore S – integer variable
- Two standard operations modify S: wait() and signal()
  - Originally called P() and V()
- Less complicated
- Can only be accessed via two indivisible (atomic) operations

(from the Dutch **proberen**, "to test"); signal() was originally called V (from ***verhogen***, "to increment").

```
wait(mutex)
{
    value=value-1;
    if(value<0)
    {
        add process to the list;
        block process;
    }
}
```

```
signal(mutex)
{
    value=value+1;
    if(value<=0)
    {
        remove process from the list;
        wakeup process;
    }
}
```

Semaphore as General Synchronization Tool

- Two types:
  - Counting semaphore – integer value can range over an unrestricted domain
  - Binary semaphore – integer value can range only between 0 and 1; can be simpler to implement
    - Also known as mutex locks (they are locks that provide mutual exclusion)

There are many usage of Semaphore

Semaphores Usage

Usage1:

- Binary semaphores can solve the critical-section problem for n processes. The n processes share a semaphore, mutex, initialized to 1. Each process P_i is organized as:

```
Semaphore mutex;    //  initialized to 1
do {
    wait (mutex);
        // Critical Section
    signal (mutex);
     // remainder section
} while (TRUE);
```

- Only one process is allowed into Critical Section (mutual exclusion).

Semaphores Usage

Usage2:

- Counting semaphores control access to a given resource of many instances.
- Simply initialize the semaphore to K (# of available resources).
- This allows K processes to enter their critical-sections and use that resource at a time.
  - Each process wants to use a resource performs *wait( )* operation → decrementing the count of the semaphore.
  - Each process releases a resource, it performs *signal( )* operation → incrementing the count of the semaphore.
  - When the count =0, all resources are being used. Any process wants to use a resource will block until count >0.

Disadvantage: Basic Semaphore

As shown in the examples above, processes waiting on a semaphore must constantly check to see if the semaphore is not zero.
This continual looping is clearly a problem in a real multiprogramming system (where often a single CPU is shared among multiple processes).
This is called **busy waiting** and it wastes CPU cycles. When a semaphore does this, it is called a **spinlock**.

End Today: 8th Sept

Semaphore - Busy Waiting

Without

```
wait(mutex)
{
    while value <= 0
    ; // no-op
    value--;
}

signal(mutex)
{
    value++;
}
```

With

```
wait(mutex)
{
    value--;
    if(value<0)
    {
        add process to the list;
        block process;
    }
}

signal(mutex)
{
    value++;
    if(value<=0)
    {
        remove process from the list;
        wakeup process;
    }
}
```

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_0	P_1
wait (S);	wait (Q);
wait (Q);	wait (S);
.	.
.	.
.	.
signal (S);	signal (Q);
signal (Q);	signal (S);

  - Suppose that *P0* **executes** wait(S) and *P1* then executes wait(Q).
  - When *P0* executes wait(Q), it must wait until *P1* executes signal(Q) .
  - Similarly, when *P1* executes wait(S), it must wait until *P0* executes signal (S) .
  - Since these signal () operations cannot be executed, *P0* and *P1* are deadlocked.

Starvation

- Starvation – indefinite blocking. A process may never be removed from the semaphore queue in which it is suspended.

Classical Problems of Synchronization

- Bounded-Buffer Problem
  - Bounded buffers P/C can be seen in e.g. streaming filters or packet switching in networks.
- Readers and Writers Problem
  - Database readers and writers: online reservation systems; file systems.
- Dining-Philosophers Problem
  - Dining philosophers could be a sequence of active database transactions that have a circular wait-for-lock dependence.

Classical Problem 2: The Readers-Writers Problem

Multiple readers or a single writer can use DB.

writer | reader | reader | reader | reader | writer

writer | writer | reader | reader | reader | reader

Readers-Writers Problem

- A data set is shared among a number of concurrent processes
  - Readers – only read the data set; they do **not** perform any updates
  - Writers – can both read and write
- Problem – allow multiple readers to read at the same time. Only one single writer can access the shared data at the same time
- Shared Data
  - Data set
  - Semaphore mutex initialized to 1
  - Semaphore wrt initialized to 1
  - Integer readcount initialized to 0

Dining-Philosophers Problem

Shared data

- Bowl of rice (data set)
- Semaphore chopstick [5] initialized to 1

Dining-Philosophers Problem

- 5 philosophers with 5 chopsticks sit around a circular table. They each want to eat at random times and must pick up the chopsticks on their right and on their left.
- Clearly deadlock is rampant (and starvation possible.)
- Several solutions are possible:
- • Allow only 4 philosophers to be hungry at a time.
- • Allow pickup only if both chopsticks are available. (Done in critical section)
- • Odd # philosopher always picks up left chopstick1st, even # philosopher always picks up right chopstick 1st.