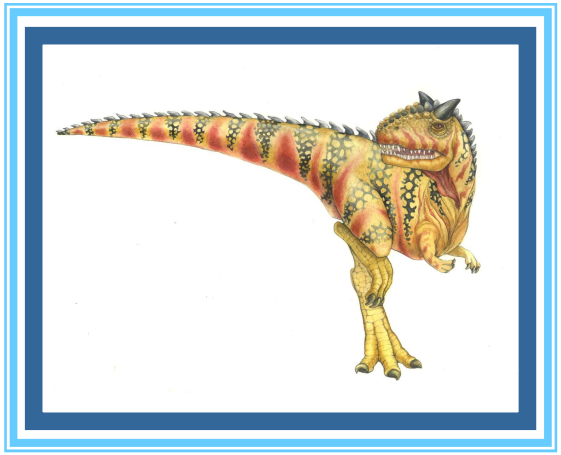
Chapter 5: Process Synchronization



Chapter 5: Process Synchronization

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

Objectives

- To present the concept of process synchronization.
- 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
- To examine several classical process-synchronization problems

Background

- Processes can execute concurrently
  - May be interrupted at any time, partially completing execution
- Concurrent access to shared data may result in data inconsistency
- Maintaining data consistency requires mechanisms to ensure the orderly execution of cooperating processes
- Illustration of the problem:
  Suppose that we wanted to provide a solution to the consumer-producer problem that fills ***all*** the buffers. We can do so by having an integer **`counter`** that keeps track of the number of full buffers. Initially, **`counter`** is set to 0. It is incremented by the producer after it produces a new buffer and is decremented by the consumer after it consumes a buffer.

Producer

```
while (true) {
        /* produce an item in next produced */

        while (counter == BUFFER_SIZE) ;
                /* do nothing */
        buffer[in] = next_produced;
        in = (in + 1) % BUFFER_SIZE;
        counter++;
}
```

Consumer

```
while (true) {
        while (counter == 0)
                ; /* do nothing */
        next_consumed = buffer[out];
        out = (out + 1) % BUFFER_SIZE;
         counter--;
        /* consume the item in next consumed */
}
```

Race Condition

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

```
register1 = counter
register1 = register1 + 1
counter = register1
```

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

```
register2 = counter
register2 = register2 - 1
counter = register2
```

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

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

Race Condition

- We would arrive at this incorrect state because we allowed both processes **to manipulate the variable counter concurrently**.
- We have several processes access and manipulate the same data concurrently and
- **the outcome** of the execution **depends on the particular order** in which the access takes place,
- This is called **a race condition**.
- We need to ensure that **only one process at a time can manipulate** the variable counter.
- The processes **need to 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);
```

Algorithm for Process P_i

```
do {

    while (turn == j);

            critical section

    turn = j;

            remainder section

} while (true);
```

Solution to Critical-Section Problem

A solution to the critical-section problem **must satisfy the following three 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** **(i.e. the selection process must take a finite time)**
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

Critical-Section Handling in OS

- the code implementing an operating system (kernel code) is subject to several possible race conditions
- For example
  - a kernel data structure that maintains a list of all open files in the system (.If two processes were to open files simultaneously, the separate updates to this list could result in a race condition)
  - structures for maintaining memory allocation,
  - for maintaining process lists, and
  - for interrupt handling

Critical-Section Handling in OS

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

- **Preemptive** – allows preemption of process when running in kernel mode
- **Non-preemptive** – runs until exits kernel mode, blocks, or voluntarily yields CPU
  - Essentially free of race conditions in kernel mode (as only one process is active in the kernel at a time)
- Preemptive kernels, so they must be carefully designed to ensure that shared kernel data are free from race conditions.
- Which could be difficult
- A pre-emptive 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 process or to waiting processes

Peterson's Solution

- Good algorithmic description of solving the problem
- 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

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

                critical section

        flag[i] = false;

                remainder section

 } while (true);
```

Peterson's Solution (Cont.)

- 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

- 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.**

- However, the solution **provides a good algorithmic description** of solving the critical-section problem

Synchronization Hardware

- Many systems provide hardware support for implementing the critical section code.
- All solutions below based on idea of **locking**
  - Protecting critical regions via locks
- Uniprocessors – could disable interrupts
  - Currently running code would execute without preemption
  - Generally too inefficient on multiprocessor systems
    - Operating systems using this not broadly scalable
- Modern machines provide special atomic hardware instructions
    - **Atomic** = non-interruptible
  - Either test memory word and set value
  - Or swap contents of two memory words

Solution to Critical-section Problem Using Locks

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

test_and_set Instruction

Definition:

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

1. Executed atomically
2. Returns the original value of passed parameter
3. 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);
```

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;
}
```

1. Executed 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”. That is, the swap takes place only under this condition.

Solution using compare_and_swap

- Shared integer “lock” initialized to 0;
- Solution:

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

- Although these algorithms satisfy the mutual-exclusion requirement, they **do not satisfy the bounded-waiting requirement**.
- Next we present another algorithm using the test and set() instruction **that satisfies all the critical-section requirements**

- The common data structures are

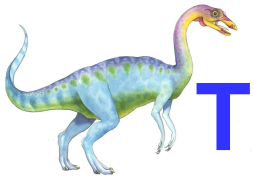
  boolean waiting[n];

  boolean lock;

- These data structures are initialized to false

Bounded-waiting Mutual Exclusion with test_and_set

```
do {
   waiting[i] = true;
   key = true;
   while (waiting[i] && key)
      key = test_and_set(&lock);
   waiting[i] = false;
   /* critical section */
   j = (i + 1) % n;
   while ((j != i) && !waiting[j])
      j = (j + 1) % n;
   if (j == i)
      lock = false;
   else
      waiting[j] = false;
   /* remainder section */
} while (true);
```



The Mutual Exclusion Requirement

- To prove that the mutual exclusion requirement is met, we note that process Pi can enter its critical section only if either waiting[i] == false or key == false.
- The value of key can become false only if the test_and_set() is executed.
- The first process to execute the test_and_set() will find key==false; all others must wait.
- The variable waiting[i] can become **false only if another process leaves** its critical section;
- only one waiting[i] is set to false, maintaining the mutual-exclusion requirement.

The bounded-waiting Requirement

- When a process leaves its critical section, it scans the array waiting in the cyclic ordering (i + 1,i + 2, ...,n−1, 0, ..., i−1).
- It designates the first process in this ordering that is in the entry section (waiting[j] == true) as the next one to enter the critical section.
- Any process waiting to enter its critical section will thus do so within n−1 turns.

Mutex Locks

- Previous solutions are complicated and generally inaccessible to application programmers
- Instead, OS designers build software tools to solve critical section problem
- Simplest is mutex lock
- the term mutex is short for mutual exclusion
- Protect a critical section by first **`acquire()`** a lock then **`release()`** the lock
  - Boolean variable indicating if lock is available or not
- Calls to **`acquire()`** and **`release()`** must be atomic
  - Usually implemented via hardware atomic instructions
- If the lock is available, a call to acquire()succeeds, and the lock is then considered unavailable

acquire() and release()

```
acquire() {
        while (!available)
            ; /* busy wait */
        available = false;;
    }
    release() {
        available = true;
    }
    do {
    acquire lock
        critical section
    release lock
      remainder section
 } while (true);
```

- But this solution requires **busy waiting**
  - **Because 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()**
  - This lock therefore called a **spinlock**

  - **Busy waiting wastes CPU cycles that some other process might be able to use productively.**

  - **Spinlocks do have an advantage, however, in that no context switch is required when a process must wait on a lock, and a context switch may take considerable time.**
  - **Thus, when locks are expected to be held for short times, spin locks are useful.**
  - **They are often employed on multiprocessor systems where one thread can "spin" on one processor while another thread performs its critical section on another processor**

Semaphore

- Synchronization tool that provides more sophisticated ways (than Mutex locks) for process 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()`**
- Definition of the **`wait()` operation**

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

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

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

Semaphore Usage

- **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**
  - **In fact, on systems that do not provide mutex locks, binary semaphores can be used instead for providing mutual exclusion.**
- Counting semaphores can be used to control access to a given resource consisting of a finite number of instances (e.g. 3 printers).
  - 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.

- Can solve various synchronization problems
- Consider P_1 and P_2 that require S_1 to happen before S_2

  Create a semaphore "**synch**" initialized to 0

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

- Can implement a counting semaphore ***S*** as a binary semaphore

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 entry in a waiting queue has two data items:
  - value (of type integer)
  - pointer to next record in the list
- Two operations:
  - **block** – place the process invoking the operation on the appropriate waiting queue.Then control is transferred to the CPU scheduler, which selects another process to execute.
  - **wakeup** – remove one of processes in the waiting queue and place it in the ready queue
-

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

Implementation with no Busy waiting (Cont.)

```
wait(semaphore *S) {
    S->value--;  // equivalent to(*S).value--
    if (S->value < 0) {
        add this process to S->list;
        block(); // suspends the process that invokes it
    }
}

signal(semaphore *S) {
    S->value++;
    if (S->value <= 0) {
        remove a process P from S->list;
        wakeup(P);
            // resumes the execution of a blocked process P.
    }
}
```

Deadlock and Starvation

- **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 then P1 executes wait(Q).**
- **WhenP0 executes wait(Q), it must wait until P1 executes signal(Q).**
- **Similarly, when P1 executes wait(S), it must wait untilP0 executes signal(S).**
- **Since these signal() operations cannot be executed, P0 and P1 are deadlocked.**

- **Deadlock**
  - **We say that a set of processes is in a deadlocked state when every process in the set is waiting for an event that can be caused only by another process in the set.**
- **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**

Classical Problems of Synchronization

- Classical problems used to test newly-proposed synchronization schemes
  - Bounded-Buffer Problem
  - Readers and Writers Problem
  - Dining-Philosophers Problem
- In our solutions to the problems, **we use semaphores for synchronization,** since that is the traditional way to present such solutions.

Bounded-Buffer Problem

- ***In our problem, the producer and consumer processes share the following data structures***
- ***n*** buffers, each can hold one item
- Semaphore **`mutex`** initialized to the value 1
  - Provides mutual exclusion for accesses to the buffer pool
- Semaphore **`full`** initialized to the value 0
  - Counts the number of full buffers
- Semaphore **`empty`** initialized to the value n
  - Counts the number of empty buffers

Bounded Buffer Problem (Cont.)

- the producer process

```
do {
    /* produce an
    item in next_produce */
        ...
     wait(empty);
     wait(mutex);
        ...
    /* add next produced
    to the buffer */
        ...
     signal(mutex);
     signal(full);
  } while (true);
```

- the consumer process

```
do {
     wait(full);
     wait(mutex);
        ...
/* remove an item from
    buffer to next_consumed */
        ...
     signal(mutex);
     signal(empty);
        ...
/* consume the item in next
    consumed */
        ...
 } while (true);
```

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
- Obviously, if two readers access the shared data simultaneously, no adverse effects will result.
- However, if a writer and some other process (either a reader or a writer) access the database simultaneously, chaos may ensue.
- Problem – allow multiple readers to read at the same time
  - Only one single writer can access the shared data at the same time
- **Several variations** of how readers and writers are considered – all involve some form of priorities

- The first readers–writers problem, requires that no reader be kept waiting unless **a writer has already obtained permission** to use the shared object.
- In other words, **no reader should wait** for other readers to finish simply **because a writer is waiting**
- **Note here writers may starve**
- **Solution to the first readers–writers problem**
- **The reader processes share** the following data structures:
- Shared Data
  - Data set
  - Semaphore **`rw_mutex`** initialized to 1
    - common to both reader and writer processes;
    - a mutual exclusion semaphore for the writers;
    - also used by the first or last reader that enters or exits the critical section

- Integer **`read_count`** initialized to 0
  - keeps track of how many processes are currently reading the object
- Semaphore **`mutex`** initialized to 1
  - used to ensure mutual exclusion when the variable read count is updated.

Readers-Writers Problem (Cont.)

- The structure of a writer process

```
do {
     wait(rw_mutex);
     ...
/* writing is performed */
     ...
     signal(rw_mutex);
} while (true);
```

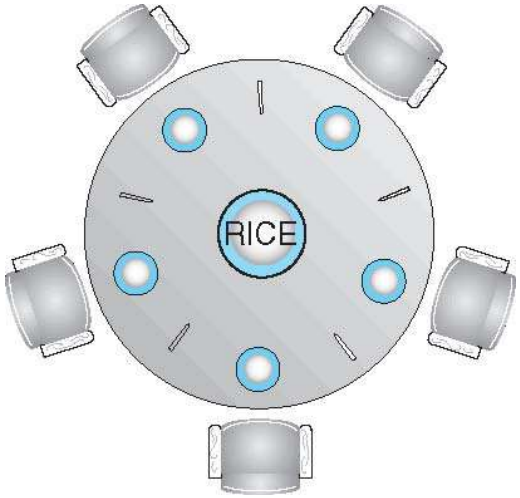
- The structure of a reader process

```
do {
     wait(mutex);
     read_count++;
     if (read_count == 1)
          wait(rw_mutex);
     signal(mutex);
     ...
/* reading is performed */
     ...
     wait(mutex);
     read count--;
     if (read_count == 0)
          signal(rw_mutex);
     signal(mutex);
} while (true);
```

Readers-Writers Problem Variations

- ***First*** variation – no reader kept waiting unless writer has permission to use shared object
- ***Second*** variation – once writer is ready, it performs the write ASAP
- Both may have starvation leading to even more variations
- Problem is solved on some systems by kernel providing reader-writer locks

Dining-Philosophers Problem



- Consider five philosophers who spend their lives thinking and eating. The philosophers share a circular table surrounded by five chairs, each belonging to one philosopher. In the center of the table is a bowl of rice, and the table is laid with five single chopsticks (Figure 5.13). When a philosopher thinks, she does not interact with her colleagues. From time to time, a philosopher gets hungry and tries to pick up the two chopsticks that are closest to her (the chopsticks that are between her and her left and right neighbors). A philosopher may pick up only one chop stick at a time. Obviously, she cannot pick up a chopstick that is already in the hand of a neighbor. When a hungry philosopher has both her chopsticks at the same time, she eats without releasing the chopsticks. When she is finished eating, she puts down both chopsticks and starts thinking a gain.

- Philosophers spend their lives alternating thinking and eating
- Don't interact with their neighbors, occasionally try to pick up 2 chopsticks (one at a time) to eat from bowl
  - Need both to eat, then release both when done
- It is a simple representation of the need to allocate several resources among several processes in a deadlock-free and starvation-free manner.
- A simple solution
  - represent each chopstick with a semaphore.
  - A philosopher tries to grab a chopstick by executing a wait() operation on that semaphore.
  - She releases her chopsticks by executing the signal() operation on the appropriate semaphores.
- In the case of 5 philosophers
  - Shared data
    - Bowl of rice (data set)
    - Semaphore chopstick [5] initialized to 1

Dining-Philosophers Problem Algorithm

- The structure of Philosopher *i*:

```
do {
    wait (chopstick[i] );
     wait (chopStick[ (i + 1) % 5] );

                // eat

     signal (chopstick[i] );
     signal (chopstick[ (i + 1) % 5] );

                // think

} while (TRUE);
```

- What is the problem with this algorithm?

Answer:

- It could create a deadlock.
  - Suppose that all five philosophers become hungry at the same time and each grabs her left chopstick.
  - All the elements of chopstick will now be equal to 0. When each philosopher tries to grab her right chopstick, she will be delayed forever.
- Several possible remedies to the deadlock problem (see next slide)

Dining-Philosophers Problem Algorithm (Cont.)

- Deadlock handling
  - Allow at most 4 philosophers to be sitting simultaneously at the table.
  - Allow a philosopher to pick up the forks only if both are available (picking must be done in a critical section.
  - Use an asymmetric solution -- an odd-numbered philosopher picks up first the left chopstick and then the right chopstick. Even-numbered philosopher picks up first the right chopstick and then the left chopstick.
- Monitors present a solution to the dining philosophers problem that ensures freedom of deadlocks

Problems with Semaphores

- Incorrect use of semaphore operations (caused by an honest programming error or an uncooperative programmer):
  - signal (mutex) …. wait (mutex)
    - Several processes may be executing in their critical sections simultaneously, violating the mutual-exclusion requirement
  - wait (mutex) … wait (mutex)
    - In this case, a deadlock will occur.
  - Omitting of wait (mutex) or signal (mutex) (or both)
    - In this case, either mutual exclusion is violated or a deadlock will occur.
- Deadlock and starvation are possible.

End of Chapter 5