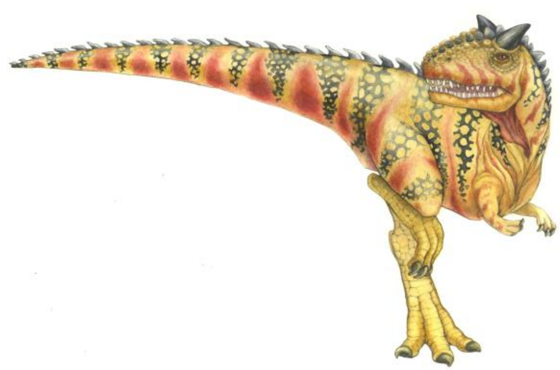
Chapter 5: Process Synchronization



Chapter 5: Process Synchronization

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

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

Background

- Processes can execute concurrently
  - May be interrupted at any time, partially executed
- Concurrent access to shared data may result in data inconsistency
- Maintaining data consistency requires mechanisms to ensure orderly execution of cooperating processes
- Illustration of the problem:
  - Suppose that we wanted to provide a solution to the consumer-producer problem that fills **<u>all</u>** 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 and 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 "counter = 5":

  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}

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 is:

```
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 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 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

Peterson's Solution

- Good algorithmic description of solving the problem
- Two process solution (numbered P_i *and* P_j)
- Assume that the **load** and **store** 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);
```

Entry section

Exit section

Provable that

1. Mutual exclusion is preserved
2. Progress requirement is satisfied
3. Bounded-waiting requirement is met

Synchronization Hardware

- Many systems provide hardware support for 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** (non-interruptible) hardware instructions:
  - Either test memory word and set value
  - Or swap contents of two memory words

test_and_set Instruction

- Definition:

```
boolean test_and_set (boolean *target)

{

        boolean rv = *target;

        *target = TRUE;

        return rv:

}
```

Solution using test_and_set()

- Shared boolean variable **lock**, initialized to FALSE
- Solution:

Not satisfy bounded-waiting requirement

```
do {
    while (test_and_set(&lock))
        ; /* do nothing */

    /* critical section */

    lock = false;

    /* remainder section */

} while (true);
```

Entry section

Exit section

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;

}
```

Solution using compare_and_swap

- Shared Boolean variable `lock` initialized to 0
- Solution:

Not satisfy bounded-waiting requirement

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

Bounded-waiting 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);
```

Entry section

Exit section

Mutex Locks

- Previous solutions are complicated and generally inaccessible to application programmers
- OS designers build software tools to solve critical section problem
- Simplest tool is **`mutex`** lock
  - Protect critical regions with it by first **`acquire()`** a lock then **`release()`** it
  - Boolean variable indicating if lock is available
  - Calls to **`acquire()`** and **`release()`** must be atomic
    - Usually implemented via hardware atomic instructions
- But this solution requires **busy waiting**
  - This lock therefore called a **spinlock**

acquire() and release()

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

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

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

Semaphore

- Synchronization tool that does not require **busy waiting**
- 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:

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

```
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
  - Then a **mutex lock**
- Can implement a counting semaphore ***S*** as a binary semaphore
- Can solve various synchronization problems
  - Ex: consider two concurrently processes P_1 with a statement S_1 and P_2 with a statement S_2 and we require S_2 be executed only after S_1 has completed

```
P1:
   S1;
   signal(synch);

P2:
   wait(synch);
   S2;
```

Semaphore Implementation

- Must guarantee that no two processes can execute **`wait()`** and **`signal()`** on the same semaphore at the same time
- Thus, 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
  - **wakeup** – remove one of processes in the waiting queue and place it in the ready queue

Semaphore Implementation with no Busy waiting

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

wait(semaphore S) {
   S.value--;
   if (S.value < 0) {
      add this process to S.list;
      block();
   }
}

signal(semaphore S) {
   S.value++;
   if (S.value <= 0) {
      remove a process P from S.list;
      wakeup(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

```
   P0                         P1
wait(S);                   wait(Q);
wait(Q);                   wait(S);
   .                          .
signal(S);                 signal(Q);
signal(Q);                 signal(S);
```

- **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

Classical Problems of Synchronization

- Classical problems used to test newly-proposed synchronization schemes
  - Bounded-Buffer Problem
  - Readers and Writers Problem
  - Dining-Philosophers Problem

Bounded-Buffer Problem

- ***n*** buffers, each can hold one item
- Semaphore **`mutex`** initialized to the value 1
- Semaphore **`full`** initialized to the value 0
- Semaphore **`empty`** initialized to the value n

Bounded Buffer Problem

- The structure of the producer process:

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

Bounded Buffer Problem

- The structure of the consumer process:

```
do {
   wait(full);
   wait(mutex);
      ...
   /* remove 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 data set and 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
- Several variations of how readers and writers are treated – all involve priorities
  - Shared Data
    - Data set
    - Semaphore **`rw_mutex`** initialized to 1
    - Semaphore **`mutex`** initialized to 1
    - Integer **`read_count`** initialized to 0

Readers-Writers Problem

- The structure of a writer process:

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

Readers-Writers Problem

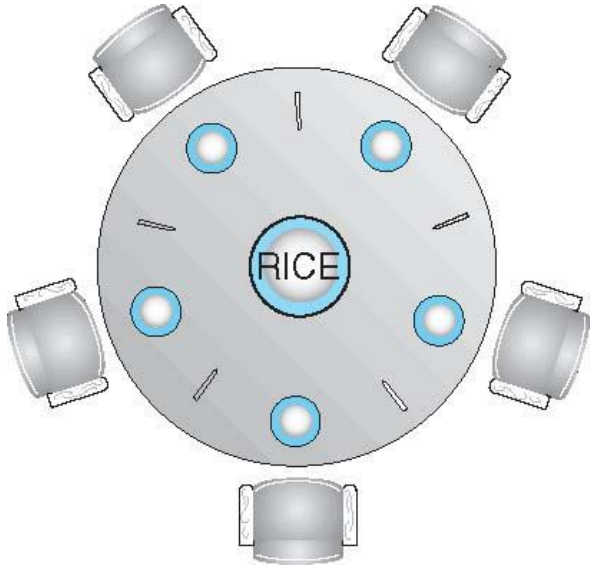
- The structure of a reader process:

```
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

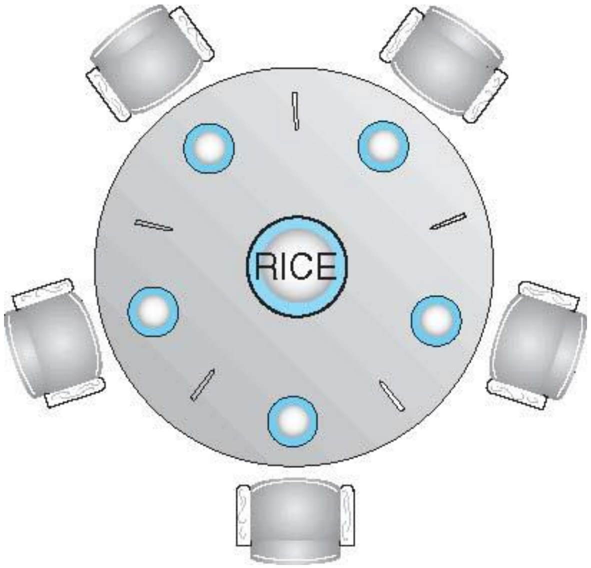
- ***First*** variation – no reader kept waiting unless writer has permission to use shared object
- ***Second*** variation – once writer is ready, it performs write ASAP
- Both may have starvation leading to even more variations
- Problem is solved on some systems by kernel providing reader-writer locks

Dining-Philosophers Problem



- Philosophers spend their lives 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

Dining-Philosophers Problem



- Shared data (in the case of 5 philosophers)
  - Bowl of rice (data set)
  - Semaphore chopstick [5] initialized to 1

Dining-Philosophers Problem

- 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?

Problems with Semaphores

- Incorrect use of semaphore operations:
  - signal (mutex) …. wait (mutex)
  - wait (mutex) … wait (mutex)
  - Omitting of wait (mutex) or signal (mutex) (or both)
- Deadlock and starvation

End of Chapter 5