

Preview

- Mutual Exclusion in a Critical Section
 - With Sleep and Wake up
 - Producer Consumer Problem
 - Race Condition Producer Consumer problem
 - Semaphore
 - Concept of Semaphore
 - Semaphore Operation
 - Semaphore Implementation
 - Producer Consumer problem with semaphores
 - Careless Usage of semaphore causes deadlock
 - Dining Philosophers Problem
 - Reader's and Writer's Problem
 - Mutexes
 - Monitor
 - Implementation of Monitor
 - Producer Consumer with Monitor
 - Message Passing
 - Producer Consumer with Message Passing

Mutual Exclusion with Sleep and Wakeup

- Sleep and Wakeup-
 - A process check a resource (critical section), if not available go to sleep.
 - When the resource become available, the process will be waked up by system or the process release the resource.

The Producer-Consumer Problem

Description

- ❑ Two processes share a common, fixed-sized buffer.
- ❑ Producer puts information into the buffer, and consumer takes it out.

Troubles arises

- ❑ When the producer wants to put a new item in the buffer, but it is already full.
- ❑ When the consumer tries to take a item from the buffer, but buffer is already empty.

The Producer-Consumer Problem

- ❑ When the producer wants to put a new item in the buffer, but it is already full.
 - Solution – producer is go to sleep, awakened by customer when customer has removed one or more items.
- ❑ When the consumer tries to take a item from the buffer, but buffer is already empty.
 - Solution – customer is go to sleep, awakened by the producer when producer puts one or more information into the buffer.

The Producer-Consumer Problem

```

#define N 100 //buffer size
int count = 0; // # of item
void producer()
{
    int item
    while (true)
    {
        item = produce_item();
        if (count == N)
            sleep();
        insert_item(item)
        count = count + 1;
        if (count == 1)
            wakeup(consumer);
    }
}

void consumer()
{
    int item;
    while(true)
    {
        if (count == 0)
            sleep();
        item = remove_item();
        count = count - 1;
        if (count == N - 1)
            wakeup(producer);
        consume_item(item);
    }
}

```

Semaphores – by E. W. Dijkstra

- A semaphore is an integer variable which could have value
 - 0: no wakeups are saved
 - + i: i wakeups are pending
- A semaphore is accessed only through two standard atomic operations *down* (or P) and *up* (or V).

Concept of Semaphores

- ❑ Modification to the integer value of the semaphore in the **down** and **up** operations are executed **indivisibly**.
- ❑ Which means that when a process is modifying the semaphore value, no other process can simultaneously modify that same semaphore value.

Semaphore Operation

```
void down (S)
```

```
{
  if S == 0
  {
    1. Add this process to the sleeping list
    2. block;
  }
  S = S - 1;
}
```

```
void up (S)
```

```
{
  S = S + 1;
  If S = 1
  {
    1. choose one process P from the
       sleeping list or let them move to
       ready state
    2. wakeup(P) to finish down operation
  }
}
```

Semaphore Implementation

The normal way for implementing a semaphore

- ❑ Implement semaphore operations **up** and **down** as system call.
- ❑ operating system briefly disabling all interrupts while it is testing the semaphore, updating it and putting the process to sleep.

Solving the Producer-Consumer Problem using Semaphores

```
#define N 100
typedef int semaphore;
semaphore mutex = 1; //mutual exclusion
semaphore empty = N; // empty space
semaphore full = 0; // number of item
void producer ()
{
    int item;

    while (ture)
    {
        item = produce_item();
        down (&empty);
        down (&mutex);
        insert_item(item);
        up(&mutex);
        up(&full);
    }
}
```

```
void consumer()
{
    int item;

    while (true)
    {
        down (&full)
        down (&mutex)
        item = remove_item();
        up (&mutex);
        up (&empty);
        consume_item(item);
    }
}
```

Careless usage of Semaphore causes deadlock

```
#define N 100
typedef int semaphore;
semaphore mutex = 1; //mutual exclusion
semaphore empty = N; // empty space
semaphore full = 0; // number of item
void producer ()
{
    int item;

    while (ture)
    {
        item = produce_item();
        down (&mutex);
        down (&empty);
        insert_item(item);
        up(&mutex);
        up(&full);
    }
}
```

```
void consumer()
{
    int item;

    while (true)
    {
        down(&full)
        down(&mutex)
        item = remove_item();
        up(&mutex);
        up(&empty);
        consume_item(item);
    }
}
```

Careless usage of Semaphore causes deadlock

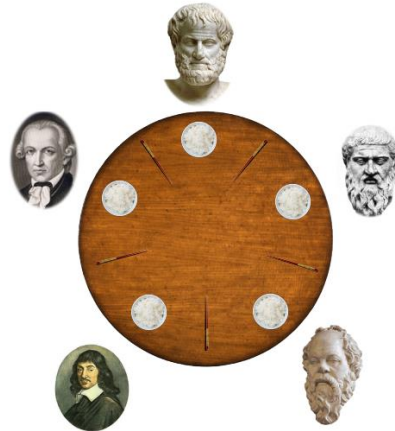
```
#define N 100
typedef int semaphore;
semaphore mutex = 1; //mutual exclusion
semaphore empty = N; // empty space
semaphore full = 0; // number of item
void producer ()
{
    int item;

    while (ture)
    {
        item = produce_item();
        down (&empty);
        down (&mutex);
        insert_item(item);
        up(&mutex);
        up(&full);
    }
}
```

```
void consumer()
{
    int item;

    while (true)
    {
        down(&mutex);
        down(&full);
        item = remove_item();
        up(&mutex);
        up(&empty);
        consume_item(item);
    }
}
```

Dining Philosophers Problem



COSC450 Operating System, Fall 2020
Dr. Sang-Eon Park

13

Dining Philosophers Problem

- ❑ Five silent philosophers sit at a round table with bowls of spaghetti. Chopsticks are placed between each pair of adjacent philosophers.
- ❑ Each philosopher must alternately think and eat. However, a philosopher can only eat spaghetti when they have both left and right chopsticks.
- ❑ Each chopstick can be held by only one philosopher and so a philosopher can use the chopstick only if it is not being used by another philosopher.
- ❑ After an individual philosopher finishes eating, they need to put down both chopsticks so that the chopsticks become available to others. A philosopher can take the chopstick on their right or the one on their left as they become available, but cannot start eating before getting both chopsticks.

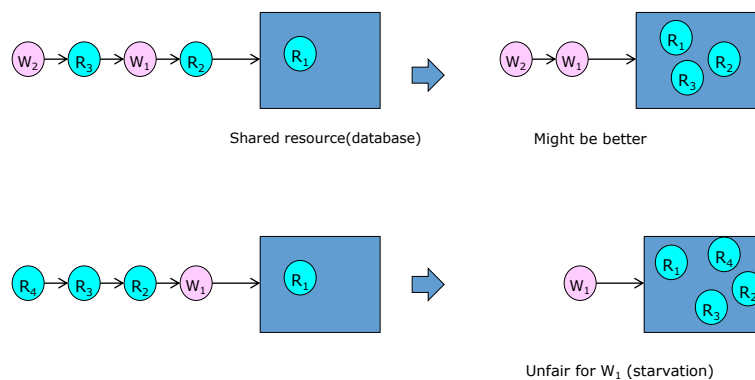
COSC450 Operating System, Fall 2020
Dr. Sang-Eon Park

14

Readers-Writers Problem

- Process reader R and writers W are sharing resources at one time. Only one process (reader or writer) can access the shared resources at any time.
 1. It is possible that a reader R_1 might have the lock to a shared resource, and then another reader R_2 requests access. It would be foolish for R_2 to wait until R_1 was done before starting its own read operation; instead R_1 and R_2 can read same resource at the same time since both are reading.
 2. It is possible that a reader R_1 might have the lock, a writer W be waiting for the lock, and then a reader R_2 requests access. It would be unfair for R_2 to jump in immediately, ahead of W ; if that happened often enough, W would starve.

Readers-Writers Problem



Mutexes

- ❑ When the semaphore's ability to count is not needed, the simplified version of the semaphore, called mutex is used.
- ❑ It is good for managing a mutual exclusion to some shared resources or pieces of code
- ❑ It is useful in thread packages that are implemented in user's space.
- ❑ A mutex is a variable that can be in one of two state: unlocked (0), locked(1).
- ❑ A mutex concept is same as binary semaphore which has value 0 or 1.

Mutexes

`mutexes mutex = 0`

repeat

`mutex_lock (mutex);`

Critical Section

`mutex_unlock (mutex);`

Remainder Section

until false

Monitor

- Monitor – High level synchronizing primitive
 - A collection of procedures, variables, and data structures that are all grouped together in a special kind of module.
 - Only one process can be active in a monitor at any instant.
 - Compiler knows that monitors are special and can handle calls to monitor procedure differently from other procedure call (create special code for monitor).
 - When a process call a procedure inside a monitor,
 1. check whether any process is active within monitor or not.
 2. If so, the calling process will be suspended until the other process has left the monitor.

Implementation of Monitor

- Since **monitor** is a **construct** for a programming language, Monitor implementation is based on the compiler
- Compiler knows monitor is special kind of module, compiler use **mutex** or **binary semaphore** for mutual exclusion.
- Monitor provide an easy way to achieve mutual exclusion. But we need to consider, how a process can be blocked and how blocked process can be waked up?
 - Introduction of **Condition variables**

Implementation of Monitor

- ❑ Conditional variables are used in the monitor.
- ❑ There are two operation on the conditional variables (**wait**, **signal**).
- ❑ When a monitor procedure discovers that it cannot continue, it does wait on some condition variable (ex full). This action causes the calling process to **block**. – allows other process get into the monitor.
- ❑ Other process (ex. consumer) can *wake* up its sleeping partner by doing a *signal* on the condition variable that its partner is waiting on.
- ❑ If there are more than one processes are waiting on a condition variable, system scheduler choose one of them

Implementation of Monitor

Once a process do a signal, what is next step for the process do a signal, to avoid having two active processes in the monitor at the same time?

Solution 1) by Hoare

- Letting the newly awakened process run, suspending the one do the signal

Solution 2) by Brinch Hansen

- A signal statement may appear only as the final statement in a monitor procedure.

Producer-Consumer with Monitor

```

monitor ProducerConsumer
condition full, empty;
integer count;
procedure insert (item: integer);
begin
  if count = N then
    wait (full);
  insert_item(item);
  count :=count + 1;
  if count = 1 then
    signal (empty)
end;

function remove: integer;
begin
  if count = 0 then
    wait (empty);
  remove = remove_item;
  count := count - 1;
  if count = N - 1 then
    signal (full);
end;
count := 0;
end monitor

```

```

procedure producer
begin
  while true do
    begin
      item = produce_item;
      ProducerConsumer.insert(item)
    end
  end;

procedure consumer;
begin
  while true do
    begin
      item = ProducerConsumer.remove;
      consume_item(item)
    end
  end;

```

Message Passing

- ❑ Message Passing is a method of interprocess communication by using two primitive system calls
 - send(destination, &message);
 - receive(source, &message);
- ❑ Usually Message Passing is used between processes located in different system since it is slower than using semaphore or monitor in the same system.
- ❑ If there is no message is available, the receiver will be blocked by system until one arrived.
- ❑ If there is no message to send, the sender will be blocked by system until one become available.

Message Passing

(Design Issues for Message Passing System)

- Message can be lost
 - Solution:
 - When a message is received, receiver send acknowledgement message.
 - If sender has not received the ack. message within a certain time interval, retransmits the message.
 - This solution cause new design issue.
- Receiver receive a message from sender, receiver send Ack. message. If the Ack. message lost, sender send same message again. Then receiver receive same message twice.
 - Solution:
 - Each message is assigned with sequence number.
 - Receiver site system can recognize duplicated message and discard one of them.

Producer-Consumer Problem

(with Message Passing)

```

#define N 100                                /* number of slots in the buffer */
void producer ()
{
    int item;
    message m;
    while (ture)
    {
        item = produce_item();              /* generate item to put in buffer */
        receive(consumer, &m);             /* wait for an empty slot (ACK)*/
        build_message(&m, item);           /* construct a message to send */
        send(consumer, &m);                /* send item to consumer */
    }
}

void consumer()
{
    int item, i;
    message m;
    for (i=0; i < N; i++)                   /* send N empty messages*/
        send(producer, &m);
    while (true)
    {
        receive(producer, &m);             /* receive a message from producer */
        item = extract_item(&m);           /* extract a message */
        send(producer, &m);                /* send an empty message to producer (ACK)*/
        consume_item(item);
    }
}

```