Control Systems Optimization

Igor Wojnicki

AGH – Univeristy of Science and Technology

2010

Outline

IPC

- Two processes do not interfere each other.
- Synchronization.
  - eg. A waits for data from B.
- One process passing information to another.
  - this issue is relaxed for threads

Shared Data Structures

- Any variable in case of threads.
- A memory region shared among processes.

A Problem: Race conditions

- Two or more processes access shared memory at the same time
- **The winner gets it all. . .**
- The result depends on which runs precisely when.
- Example: two processes try to access a printer (or actuator).

Mutual Exclusion

- *Mutual Exclusion* – if one process is using a shared resource, no other process is allowed to use it.
- *Critical Region* (Critical Section): the part of the program where the shared memory is accessed.

Competition Among Processes for Resources

- Mutual Exclusion
  - Critical regions
    - Only one program at a time is allowed in its critical region
    - Example: only one process at a time is allowed to send command to the printer
- Problems:
  - Starvation
  - Deadlock

Four Conditions to Provide Mutual Exclusion

1. No two processes simultaneously in critical region
2. No assumptions made about speeds or numbers of CPUs
3. No process running outside its critical region may block another process
4. No process must wait forever to enter its critical region

Critical Region

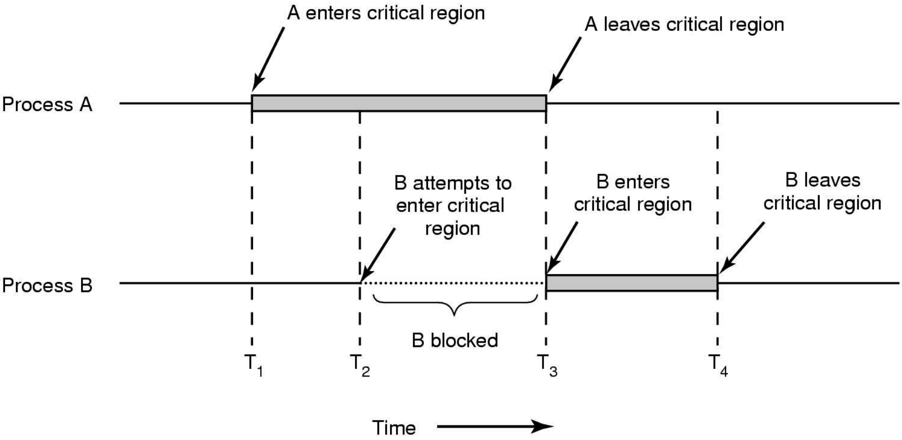


Figure: Mutual exclusion using critical regions

Requirements for Mutual Exclusion I

- Only one process at a time is allowed in the critical section for a resource
- A process that halts in its non-critical section must do so without interfering with other processes
- No deadlock or starvation
- A process must not be delayed access to a critical section when there is no other process using it
- No assumptions are made about relative process speeds or number of processes
- A process remains inside its critical section for a finite time only

Mutual Exclusion Solutions

Provided by the underlying OS and/or hardware.

- Mutex
- Semaphore
- Message

Mutex

- Binary Semaphore.
- It is a semaphore which may be in one of two states:
  - 0 – unlocked
  - 1 – locked
- Efficient for managing mutual exclusion to a shared resource.

Semaphore

- An integer counter.
- Operations:
  - down (wait): `if (value>0) {value-; continue;} else sleep;`
  - up (signal): `value++`, one of the sleepers should be awaken.
- The operations are atomic!

Producer-Consumer Problem

- Bounded Buffer Problem
- Producer puts information into the buffer.
- Consumer reads information from the buffer.

```
#define N 100                           /* number of slots in the buffer */
typedef int semaphore;                  /* semaphores are a special kind of int */
semaphore mutex = 1;                    /* controls access to critical region */
semaphore empty = N;                    /* counts empty buffer slots */
semaphore full = 0;                     /* counts full buffer slots */

void producer(void)
{
     int item;

     while (TRUE) {                     /* TRUE is the constant 1 */
          item = produce_item( );       /* generate something to put in buffer */
          down(&empty);                 /* decrement empty count */
          down(&mutex);                 /* enter critical region */
          insert_item(item);            /* put new item in buffer */
          up(&mutex);                   /* leave critical region */
          up(&full);                    /* increment count of full slots */
     }
}


void consumer(void)
{
     int item;

     while (TRUE) {                     /* infinite loop */
          down(&full);                  /* decrement full count */
          down(&mutex);                 /* enter critical region */
          item = remove_item( );        /* take item from buffer */
          up(&mutex);                   /* leave critical region */
          up(&empty);                   /* increment count of empty slots */
          consume_item(item);           /* do something with the item */
     }
}
```

Message Passing

- Enforce mutual exclusion
- Exchange information

```
send (destination, message)
receive (source, message)
```

Synchronization with Messages

- Sender and receiver may or may not be blocking (waiting for message)
- Blocking send, blocking receive
    - Both sender and receiver are blocked until message is delivered
    - Called a rendezvous
- Nonblocking send, blocking receive
    - Sender continues processing such as sending messages as quickly as possible
    - Receiver is blocked until the requested message arrives
- Nonblocking send, nonblocking receive
    - Neither party is required to wait

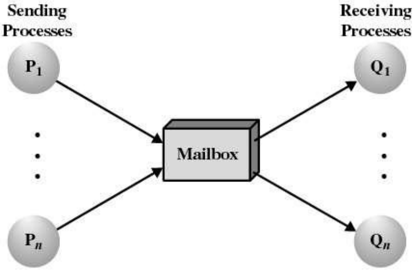
Message Addressing

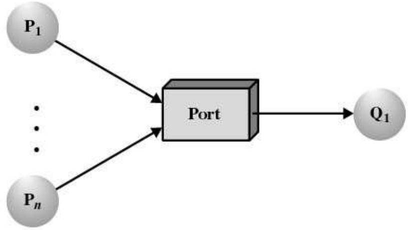
- Direct addressing
  - send primitive includes a specific identifier of the destination process
  - receive primitive could know ahead of time which process a message is expected
  - receive primitive could use source parameter to return a value when the receive operation has been performed

Message Addressing cont.

- Indirect addressing
  - messages are sent to a shared data structure consisting of queues
  - queues are called mailboxes
  - one process sends a message to the mailbox and the other process picks up the message from the mailbox

Message Addressing example





Producer-Consumer with Messages

```
#define N 100                          /* number of slots in the buffer */

void producer(void)
{
    int item;
    message m;                         /* message buffer */

    while (TRUE) {
        item = produce_item( );        /* generate something to put in buffer */
        receive(consumer, &m);         /* wait for an empty to arrive */
        build_message(&m, item);       /* construct a message to send */
        send(consumer, &m);            /* send item to consumer */
    }
}

void consumer(void)
{
    int item, i;
    message m;

    for (i = 0; i < N; i++) send(producer, &m);  /* send N empties */
    while (TRUE) {
        receive(producer, &m);         /* get message containing item */
        item = extract_item(&m);       /* extract item from message */
        send(producer, &m);            /* send back empty reply */
        consume_item(item);            /* do something with the item */
    }
}
```

Inter-Process Communication OS level

- Commad-line interface
  - `ipcs` – list defined IPC resources
  - `ipcrm` – remove
  - `ipcmk` – create

Other Common Problems

- Dining Philosophers
- Readers and Writers
- Sleeping Barber