CS415 Module 5 Part A - Interprocess Communication Introduction

Athens State University

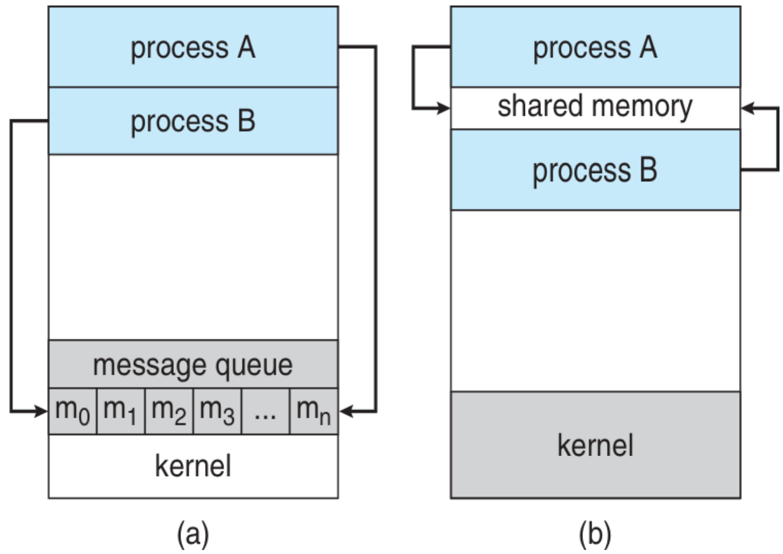
Outline

Contents

Interprocess Communication

- Processes within a system may be *independent* or *cooperating*
- Cooperating processes affect or be affected by other processes, including sharing of data
- Reasons for process to cooperate
  - Information sharing
  - Computation speedup
  - Modularity
  - Convenience
- Cooperating processes require tools for *interprocess communication* (IPC)

IPC Models



A few notes:

1. Figure (a) illustrates the message passing model
2. Figure (b) illustrates the shared memory model
3. To start, we will limit ourselves to communication between processes in the same machine
    - Some IPC mechanisms, such as Sockets are network oriented

IPC - Message Passing

- Mechanism that allows processes to communicate and to synchronize their actions
- Two basic operations:
    - `send(message)`
    - `receive(message)`
- Message size can be fixed or variable
- IPC mechanism is responsible for queuing and delivery of messages

IPC - Shared Memory

- An area of memory that is shared among the processes that wish to communicate
- The communication is under control of the users rather than the operating system
- Need to provide mechanisms that allow user processes to synchronize when they access shared memory

1 IPC Design and Implementation Issues

Message Passing Workflow

- If two processes wish to communicate, they need to:
  - Establish a communication link between them
  - Exchange messages via send/receive
- Implementation issues:
  - How are links established?
  - Can a link be associated with more than two processes?
  - How many links can there be between every pair of communicating processes?
  - What is the capacity of a link?
  - Is the size of a message that the link can accommodate fixed or variable?
  - Is a link unidirectional or bi-directional?

How are messages communicated?

- How are links established?
- Can a link be associated with more than two processes?
- How many links can there be between every pair of communicating processes?
- What is the capacity of a link?
- Is the size of a message that the link can accommodate fixed or variable?
- Is a link unidirectional or bi-directional?

How are messages communicated?

- Implementation of communication link
  - Physical:
    - Shared memory
    - Hardware bus
    - Network
  - Logical:
    - Direct or indirect
    - Synchronous or asynchronous
    - Automatic or explicit buffering

1.1 More on direct vs. indirect communication

Direct Communication

- Processes must name each other explicitly
  - `send(P, message)` - send a message to process `P`
  - `receive(Q,message)` - receive a message from process `Q`
- Properties of the communication link
  - Links are established automatically
  - Each link is associated with exactly one pair of communicating processes
  - Each pair uses exactly one link
  - Link may unidirectional, but is usually bi-directional

Indirect Communication

- Messages are sent to and received from mailboxes
  - Also referred to as ports
  - Processes can communicate only if they share a mailbox
- Properties of the communication link
  - A link is established only if processes share a common mailbox
  - Links may be associated with many processes
  - Each pair of processes may share several communication links
  - Links may be unidirectional or bi-directional

Indirect Communication

- Operations
  - Create a new mailbox
  - send and receive messages through mailbox
  - destroy mailbox
- Primitives defined as:
  - `send(A,message)` - send message to mailbox A
  - `receive(A,message)` - receive message from mailbox A

Indirect Communications: Issues

- Mailbox sharing
  - P_1, P_2, and P_3 share a mailbox A
  - P_1 sends; P_2 and P_3 receive
  - Who gets the message?
- Solutions
  - Limit links to at most two processes
  - Allow only one process at a time to execute a receive
  - System arbitrarily selects receiver, notify sender of winner

1.2 Synchronization and buffering

Synchronization

- Message passing may be either blocking or non-blocking
- **Blocking** is synchronous sending, may choose to have either the sender or receiver to block
- **Non-Blocking** is asynchronous sending of messages
  - **Non-blocking send**: The sender sends the message and continues
  - **Non-blocking receive**: the receiver receives eihter a valid message or a NULL message
- Different combinations are possible, most often non-blocking send with blocking receive

Implicit Message Synchronization and Producer/Consumer

The implicit synchronization provided by message sending makes producer-consumer trvial:

```
// Producer
struct message next_produced;
while (true) {
  // produce an item and put reference to item in
  // payload of next_produced
  send(next_produced);
}

// Consumer
struct message next_consumed;
while (true) {
  receive(next_consumed);
  // Extract reference to item from message payload
  // and do what you need to do
}
```

Buffering of messages

- Queue of messages attached to the link
- Implemented in one of three ways:
  1. *Zero capacity* - No messages are queued, sender must wait for receiver
  2. *Bounded capacity* - Finite capacity queue, sender must wait if link full
  3. *Unbounded capacity* - Infinite length, sender never waits

2 Examples of IPC Systems

POSIX Shared Memory

```
shm_fd = shm_open(name, O_CREAT | O_RDWR, 0666);
ftruncate(shm_fd, 4096);
sprintf(shm_fd, "writing to shared memory");
```

- Operating system allocates block(s) of memory shared among processes

- Process first creates shared memory segment
- Set the size of the object to match
- Now write to segment as memory is in own memory space

Using shared memory to producer-consumer: producer

```c
#include <stdio.h.
#include <stdlib.h>
#include <string.h>
#include <fcntrl.h>
#include <sys/shm.h>
#include <sys/stat.h>
int main(int argc, char *argv[]) {
    const int SIZE = 4096;
    const char *name = "SH_SEG";
    const char *message0 = "Hello";
    const char *message1 = "World";
    int shmFD;
    void *ptr;
    shmFD = shm_open(name, O_CREAT | O_RDWR, 0666);
    ftruncate(shmFD, SIZE);
    ptr = mmap(0, SIZE, PROT_WRITE, MAP_SHARED, shmFD, 0);
    sprintf(ptr, "%s", message0);
    ptr += strlen(message0);
    sprintf(ptr, "%s", message1);
    ptr += strlen(message0);
    return 0;
}
```

Using shared memory to producer-consumer: consumer

```c
#include <stdio.h.
#include <stdlib.h>
#include <string.h>
#include <fcntrl.h>
#include <sys/shm.h>
#include <sys/stat.h>
int main(int argc, char *argv[]) {
    const int SIZE = 4096;
    const char *name = "SH_SEG";
    int shmFD;
    void *ptr;
    shmFD = shm_open(name, O_CREAT | O_RDWR, 0666);
    ptr = mmap(0, SIZE, PROT_WRITE, MAP_SHARED, shmFD, 0);
    printf("%s", (char *)ptr);
    shm_unlink(name);
    return 0;
}
```

Examples of IPC Systems: The Mach kernel

- IPC in the Mach kernel (used in macOS and iOS) is message based
  - Even system calls are messages
  - Each task gets two mailboxes at creation: Kernel and Notify

  - Three calls: `msg_send()`, `msg_receive()`, and `msg_rpc()`
  - Mailboxes are created using `port_allocate`

Examples of IPC Systems: Windows LPC

- Message-passing centric via advanced local procedure call
- Only works between processes in the same system
- Uses ports(like mailboxes) to establish and maintain communications
  - Client opens a handle to a connection port object
  - Client sends a connection request
  - Server creates two private communication ports and returns handle to one of them back to client
  - Client and server uses port handles to communicate and listen for replies

3 Key Points

Key Points

-