IPC

CPE 545

Purposes for IPC

- Data Transfer
- Sharing Data
- Event notification
- Resource Sharing and Synchronization
- Process Control

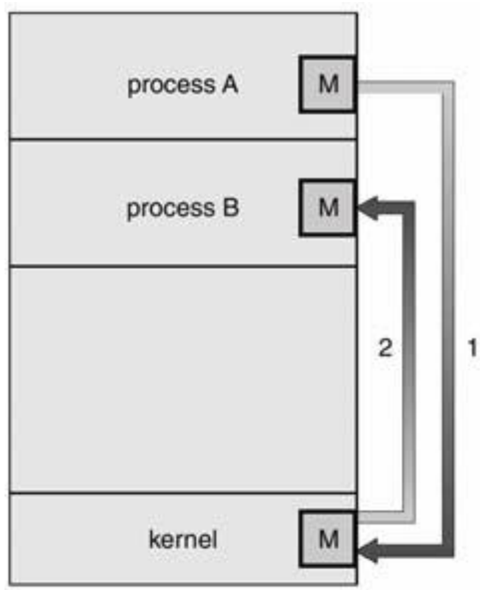
IPC Mechanisms

- *Mechanisms used for communication and synchronization*
  - Message Passing
    - message passing interfaces, mailboxes and message queues
    - sockets, STREAMS, pipes
  - *Shared Memory*: Non-message passing systems
  - *Synchronization* – primitives such as semaphores to higher level mechanisms such as monitors
  - *Event Notification* - UNIX *signals*

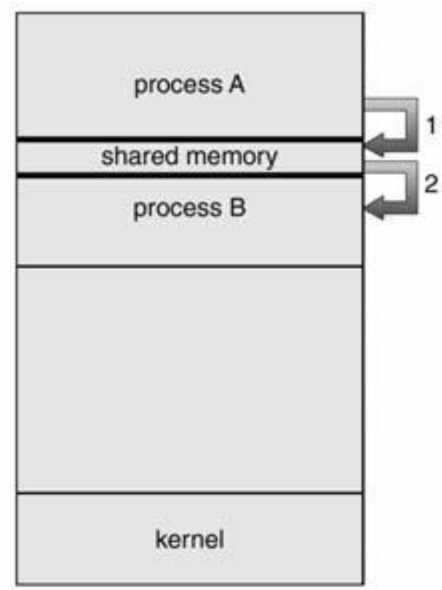
Message Passing

- In a *Message system* there are no shared variables. IPC facility provides two operations for fixed or variable sized message:
  - *send(message)*
  - *receive(message)*
- If processes *P* and *Q* wish to communicate, they need to:
  - establish a *communication link*
  - exchange messages via *send and receive*
- Implementation of communication link
  - physical (e.g., shared memory, hardware bus)
  - logical (e.g., syntax and semantics, abstractions)

Shared Memory



Message passing



Shared Memory

Mailboxes

- **Indirect communications** :
  - Messages sent to and received from *mailboxes* (or *ports*)
    - mailboxes can be viewed as objects into which messages placed by processes and from which messages can be removed by other processes
  - Each mailbox has a unique ID
  - Two processes can communicate only if they have a shared mailbox

    send (A, message) : send a *message* to mailbox *A*
    receive (A, message) : receive a *message* from mailbox *A*
  - A communications link is only established between a pair of processes if they have a shared mailbox
  - A pair of processes can communicate via several different mailboxes if desired
  - A link can be either unidirectional or bidirectional
  - A link may be associated with more than two processes
    - allows *one-to-many*, *many-to-one*, *many-to-many* communications

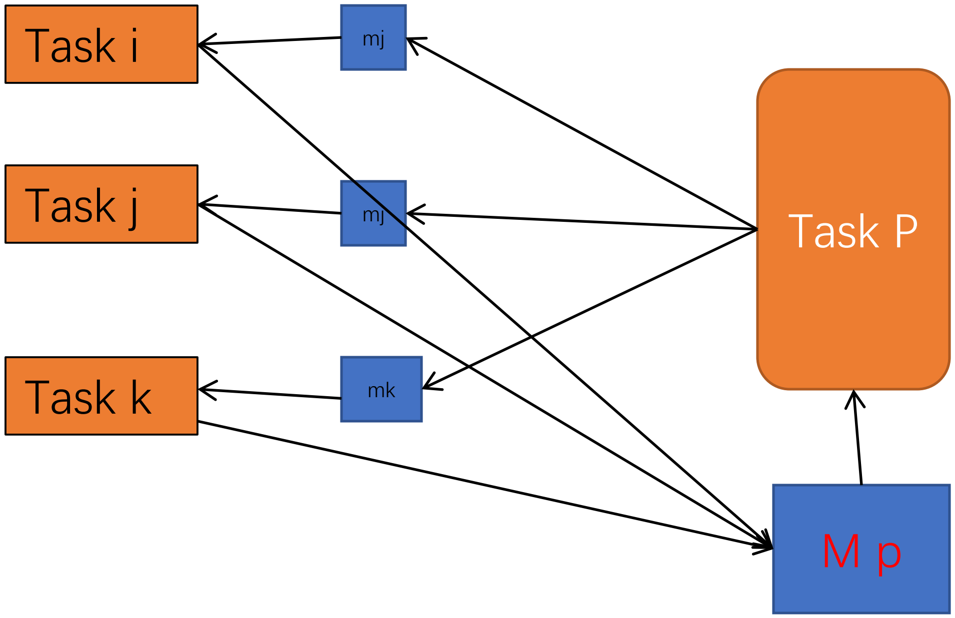
Mailboxes- system ownership

- one-to-many : any of several processes may receive from the mailbox
  - which of the receivers gets the message?
    - only allow one process at a time to wait on a receive
- many-to-one : many processes sending to one receiving process
  - file server, network server, mail server etc.
  - receiver can identify the sender from the message header contents
- many-to-many :
  - e.g. multiple senders requesting service and a pool of receiving servers offering service - a *server farm*

Mailboxes- Process mailbox ownership

- only the process may receive messages from the mailbox
- other processes may send to the mailbox
- mailbox can be created with the process and destroyed when the process dies
    - process sending to a dead process's mailbox will need to be signaled
- *or* through separate **create_mailbox** and **destroy_mailbox** calls
    - possibly declare variables of type 'mailbox'

Mailboxes



Mailboxes

- A mailbox can correspond to a semaphore:
  non-blocking send + *blocking* receive
  - equivalent to :
    signal by sender (semGive) + *wait* by receiver (semTake)
- *Mutual Exclusion* :
  - initialize :

```
create_mailbox (mutex)
send (mutex, null-message)
```

  - for each process :

```
while (TRUE)
{
        receive (mutex, null-message);
        critical section
        send (mutex, null-message);
}
```

  - mutual exclusion just depends on whether mailbox is empty or not
    - message is just a token, possession of which gives right to enter C.S.

Homework
Synchronization with Mutex

- Task A has higher priority than Task B
- In a special corner case condition in task A, if procedure getAlarmStat() is called, a call to UpdateAlarms() must be called by task B first, or the status of alarms are useless.
- Use Mutex or Mailbox to make sure the above order of execution is enforced

Producer / Consumer

- *Producer / Consumer* problem using messages :
  - *Binary* semaphores : one message token
  - *General* (*counting*) semaphores : more than one message token
  - message blocks used to buffer data items
  - scheme uses two mailboxes
    - *mayproduce* and *mayconsume*

  - *producer :*
    - get a message block from *mayproduce*
    - put data item in block
    - send message to *mayconsume*
  - *consumer :*
    - get a message from *mayconsume*
    - consume data in block
    - return empty message block to *mayproduce* mailbox

Producer / Consumer

- parent process creates message slots
  - buffering capacity depends on number of slots created
  - slot = *empty message*
    capacity = *buffering capacity*
    create_mailbox (mayproduce);
    create_mailbox (mayconsume);
    for (i=0; i<capacity; i++) send (mayproduce, slot);
    start producer and consumer processes
- producer :
  - while (TRUE) {
        receive (mayproduce, slot);
        slot = *new data item*
        send (mayconsume, slot);
    }
- consumer :
  - while (TRUE) {
        receive (mayconsume, slot);
        consume data item in slot
        send (mayproduce, slot);
    }

Event/Signals

- *Signals*
  - The mechanism whereby processes are made aware of events occurring
  - Asynchronous - can be received by a process at any time in its execution
  - Examples of Linux signal types:
    - SIGINT : interrupt from keyboard
    - SIGFPE : floating point exception
    - SIGKILL : terminate receiving process
    - SIGCHLD : child process stopped or terminated
    - SIGSEGV : segment access violation
  - Default action is usually for kernel to terminate the receiving process

Event/Signals

- Process can request some other action
  - ignore the signal - process will not know it happened
    - SIGKILL and SIGSTOP cannot be ignored
  - restore signal's default action
  - execute a pre-arranged signal-handling function
    - process can register a function to be called
    - like an interrupt service routine
    - when the handler returns, control is passed back to the main process code and normal execution continues

  - to set up a signal handler:
    ```
    void (*signal(int signum, void (*handler)(int)))(int);
    ```

  - signal is a call which takes two parameters
    - signum : the signal number
    - handler : a pointer to a function which takes a single integer parameter and returns nothing (void)
  - return value is itself a pointer to a function which:
    - takes a single integer parameter and returns nothing

Signals: Changing the behavior of ^C

```
main () {
int c;
    old_handler = signal (SIGINT, ctrl_c );
    while ((c = getchar()) != '\n');  //stuck here till hit <RET>
    printf("ctrl_c count = %d\n", ctrl_c_count);
    (void) signal (SIGINT, old_handler);

    for (;;);
}

void ctrl_c(int signum) {
    (void) signal (SIGINT, ctrl_c);     // signals are
automatically reset
    ++ctrl_c_count;
}
```

- gets characters until a newline typed, then goes into an infinite loop
- uses signals to count ctrl-c's typed at keyboard until newline typed

Homework

How would you design an event handler framework like Linux Signals?