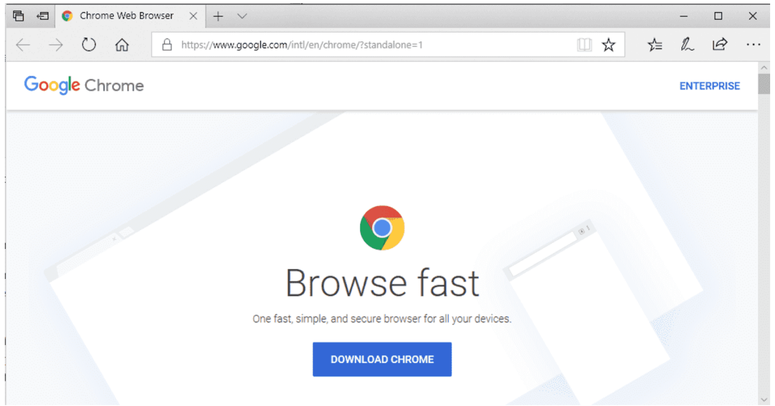
Inter-Process Communication

IPC Using Shared Memory

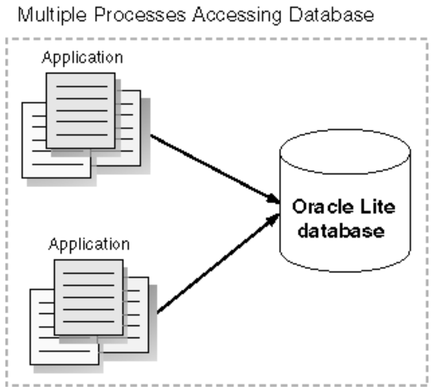
Inter-Process Communication

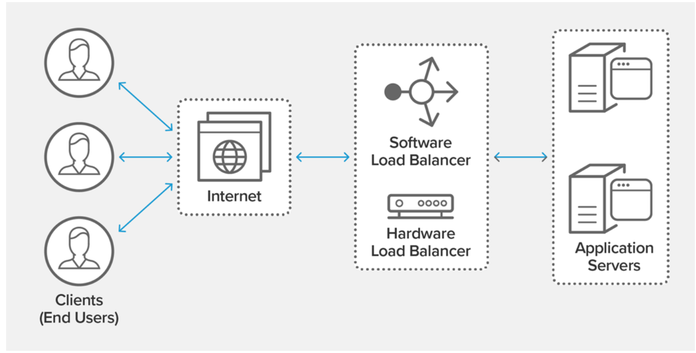
- Processes running inside a computer can be **independent** or **cooperating** with each other
- A cooperating process can affect or get affected by another process
- Reason for cooperating processes:
  - Information sharing
  - Computation speedup
  - Modularity

Inter-process within an application



Inter-process within a computer

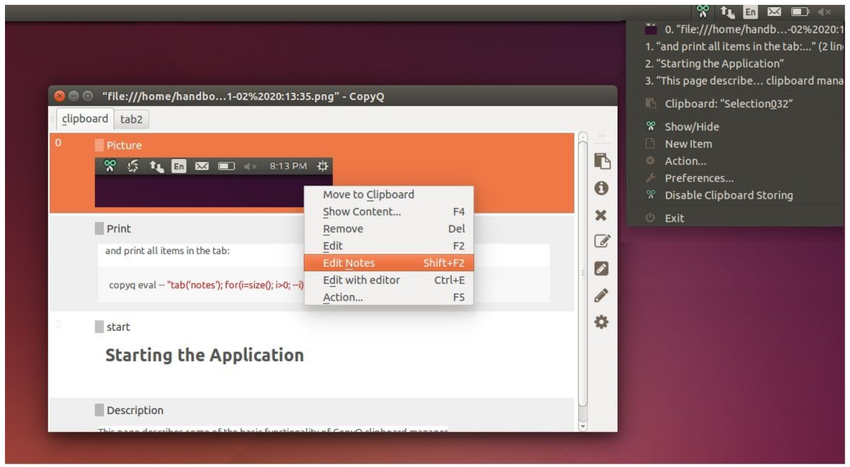


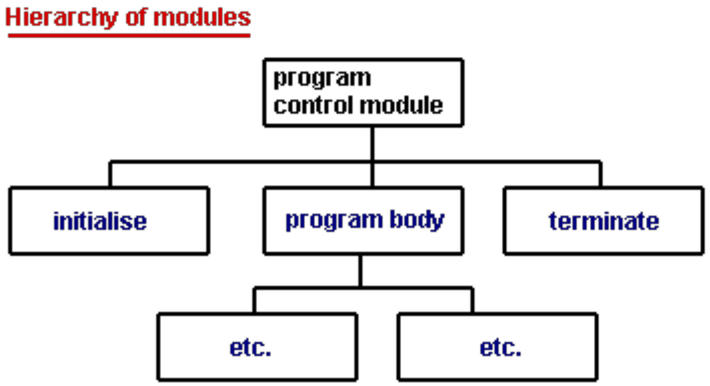


Inter-process across many machines

Information Sharing & Modularity

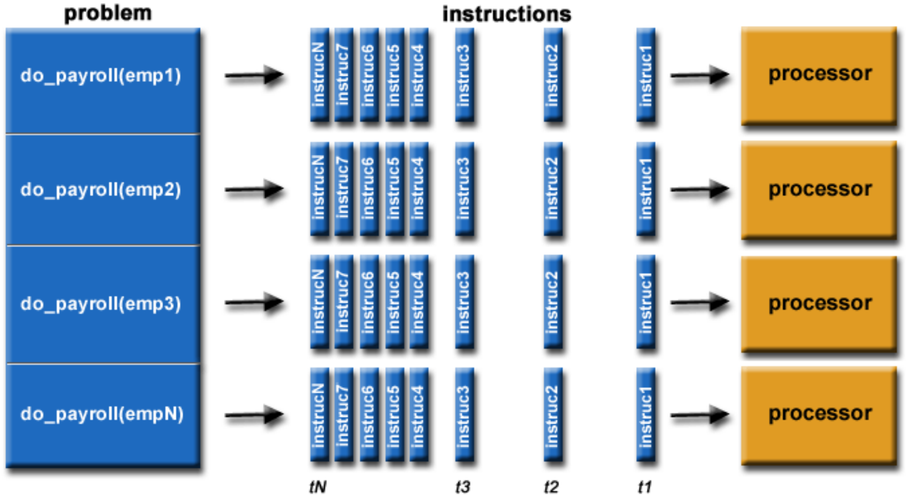
- Several applications are interested in the **same piece of information**
- Example: clip board - cut and paste to the clip board!
- We must control concurrent access to the shared space for correct informations sharing
- **Modularity** is another reason for cooperation - all of the processes are working towards a common objective



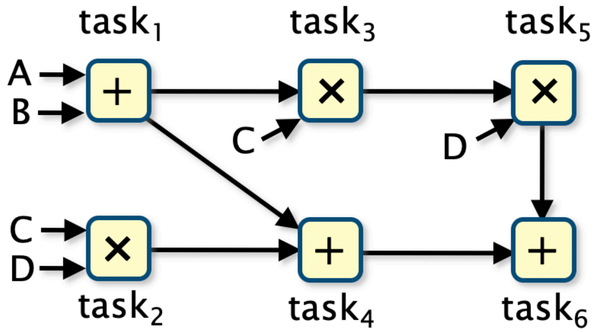


Computation Speedup

- To make the application faster, we need to break it into smaller chunks
- Each chunk goes into a process
- Processes need to cooperate to do the whole application
- If the computer has hardware parallelism (multiple cores or processors) we can get higher performance



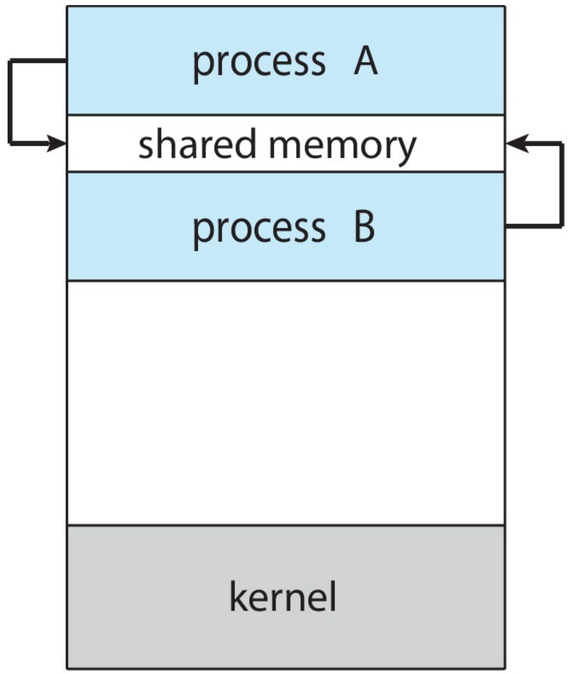
Parallel computing by splitting the data



Parallel computing by splitting task graph

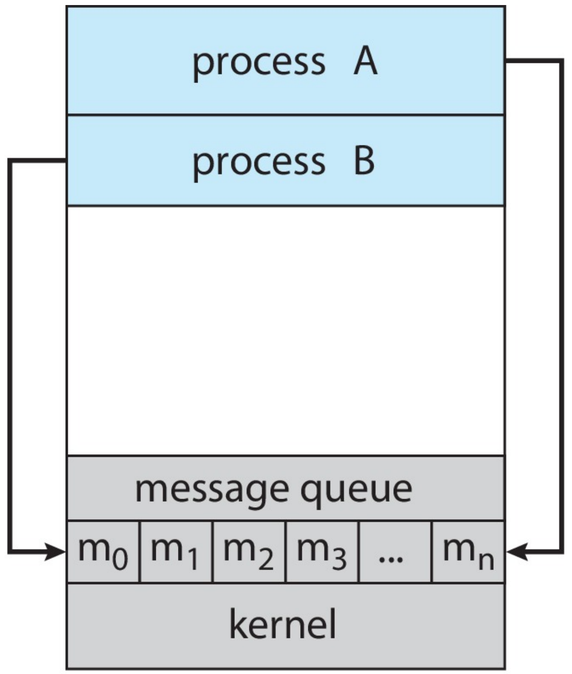
Two Models of IPC

Shared memory needs a physical or virtual memory that is accessible to both processes



(a)

Shared Memory



(b)

Message Passing

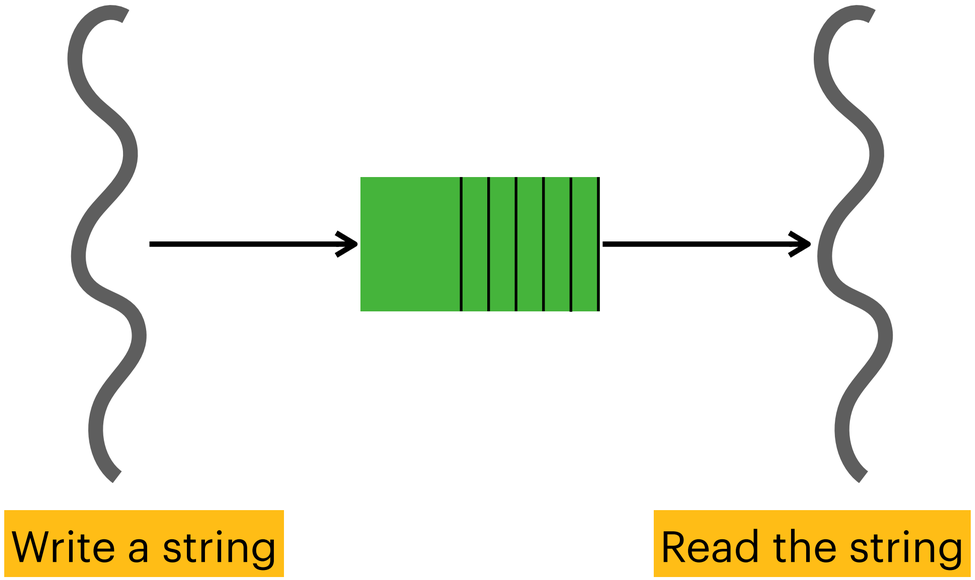
Message passing is suitable for small message transfers

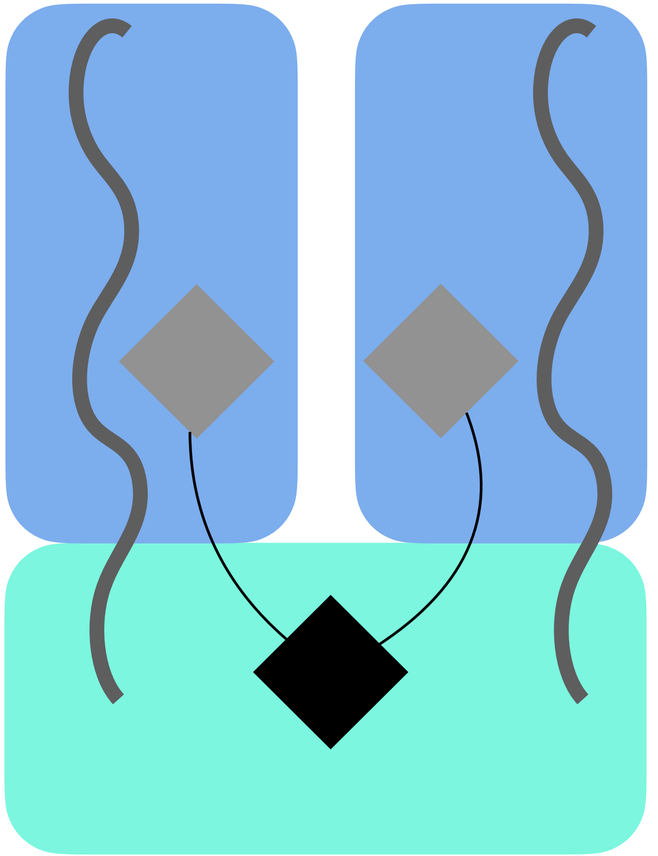
Shared memory can be faster - because there is no need for kernel access at every memory access

IPC in Shared Memory Systems

- IPC problem would have a **writer** (also known as **producer**) that is creating the data
- A **reader** (also known as the **consumer**) that reading the data
- A producer/consumer would do the following to access shared memory
  - Create shared memory
  - Map shared memory into their address space
  - Write or read to the shared memory

Producer-Consumer in Shared Memory





Producer-Consumer Code

```
producer() {

  fd = open_shared_memory_segment(name, “w”)
  resize_segment(fd, size);
  ptr = map_segment(fd, permissions)

  write_memory(ptr)
  close_shared_memory(fd)
}
```

```
consumer() {

  fd = open_shared_memory_segment(name, “r”)
  ptr = map_segment(fd, permissions)

  read_memory(ptr)
  close_shared_memory(fd)
}
```

IPC in Message Passing Systems

- No shared memory with this type of IPC
- Producer and consumer interact by sending messages
- send(message) — used to send a fixed or variable message
- receive(message) — responsible for retrieving messages injected by prior send() on the communication link
- send() and receive() form a communication link

Communication Link

- Communication links can categorized based on the following concepts:
  - Direct or indirect communication
  - Synchronous or asynchronous communication
  - Automatic or explicit buffering

Naming in Communication Links

Direct Communication

- send(P, message) — P is the address (name) of the receiving process
- receive(Q, message) — Q is the address (name) of the sending process

Disadvantage: Delay of sending the message could be higher, throughput (number of messages sent) could be lower

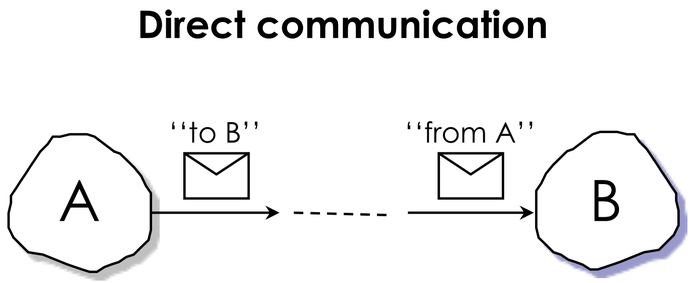
Indirect Communication

- send(A, message) — A is the mailbox to send to
- receive(A, message) — A is mailbox to receive from

Advantage of using mailbox: Sender and Receiver identities can change the communicating program need not be updated

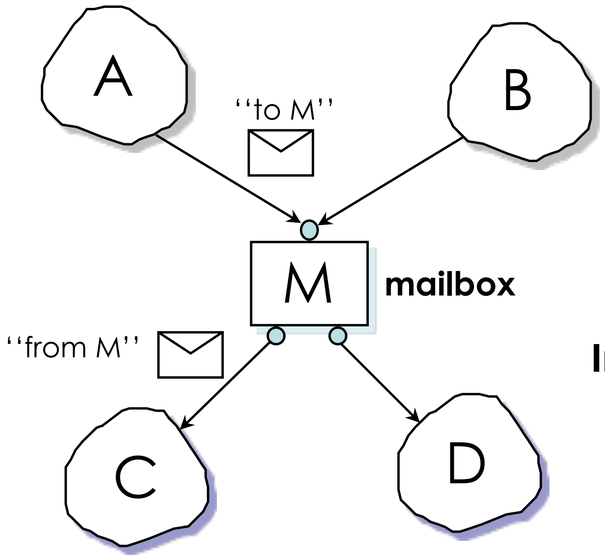
Direct vs Indirect Communication

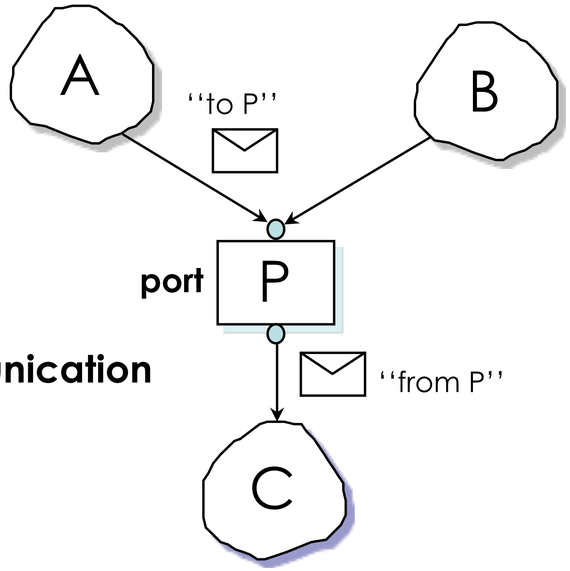
In the general case, A could be having many communication links



A message is delivered to only one consumer.

We can load balance the message processing by distributing among C and D.





Little About Mailboxes

- Mailboxes have to be created and deleted
- Mailbox can be created by a process - in that case the process becomes the owner of the mailbox
- Mailbox becomes unavailable when the owner deletes it
- Mailbox can be created by the Operating System in that case it would be available independent of the communicating processes

Synchronization With Message Passing

- Message passing can be either blocking or non-blocking
- **Blocking** is considered **synchronous**
  - Blocking send - sender is blocked until the message is received
  - Blocking receive - receiver is blocked until the message is available
- **Non blocking** is considered **asynchronous**
  - Non-blocking send - the sender sends the message and continue
  - Non-blocking receive - the receiver receives:
    - A valid message, or
    - Null message
- If both sender and receiver are synchronous, we have a **rendezvous**

Buffering in Communication Links

- Three types of messaging based on capacity of buffer in the link
  - Zero capacity — no messages are queued in the link. Sender must wait for the receiver (get rendezvous)
  - Bounded capacity — finite length of n messages. Sender must wait if buffer in link is full
  - Unbounded capacity - infinite length - sender never waits

Examples of IPC - POSIX Shared Memory

- First step is to open a shared memory segment, much like opening a file for I/O

  shm_fd = shm_open(name, O_CREAT|O_RDWR, 0666);

- The above can be used to open existing segments or create new segments
- Set the size of the segment opened by the above command

  ftruncate(shm_fd, 4096);

- Use mmap() to memory map a file pointer to the shared memory object
- Reading or writing to the shared memory object is possible to the mmap() returned pointer

Producer-Consumer Code (in POSIX)

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <fcntl.h>
#include <sys/shm.h>
#include <sys/stat.h>

#include <sys/mman.h>

int main()
{
/* the size (in bytes) of shared memory object */
const int SIZE = 4096;
/* name of the shared memory object */
const char *name = "OS";
/* strings written to shared memory */
const char *message_0 = "Hello";
const char *message_1 = "World!";

/* shared memory file descriptor */
int fd;
/* pointer to shared memory obect */
char *ptr;

    /* create the shared memory object */
    fd = shm_open(name,O_CREAT | O_RDWR,0666);

    /* configure the size of the shared memory object */
    ftruncate(fd, SIZE);

    /* memory map the shared memory object */
    ptr = (char *)
      mmap(0, SIZE, PROT_READ | PROT_WRITE, MAP_SHARED, fd, 0);

    /* write to the shared memory object */
    sprintf(ptr,"%s",message_0);
    ptr += strlen(message_0);
    sprintf(ptr,"%s",message_1);
    ptr += strlen(message_1);

    return 0;
}
```

```c
#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>
#include <sys/shm.h>
#include <sys/stat.h>

#include <sys/mman.h>

int main()
{
/* the size (in bytes) of shared memory object */
const int SIZE = 4096;
/* name of the shared memory object */
const char *name = "OS";
/* shared memory file descriptor */
int fd;
/* pointer to shared memory obect */
char *ptr;

    /* open the shared memory object */
    fd = shm_open(name, O_RDONLY, 0666);

    /* memory map the shared memory object */
    ptr = (char *)
      mmap(0, SIZE, PROT_READ | PROT_WRITE, MAP_SHARED, fd, 0);

    /* read from the shared memory object */
    printf("%s",(char *)ptr);

    /* remove the shared memory object */
    shm_unlink(name);

    return 0;
}
```