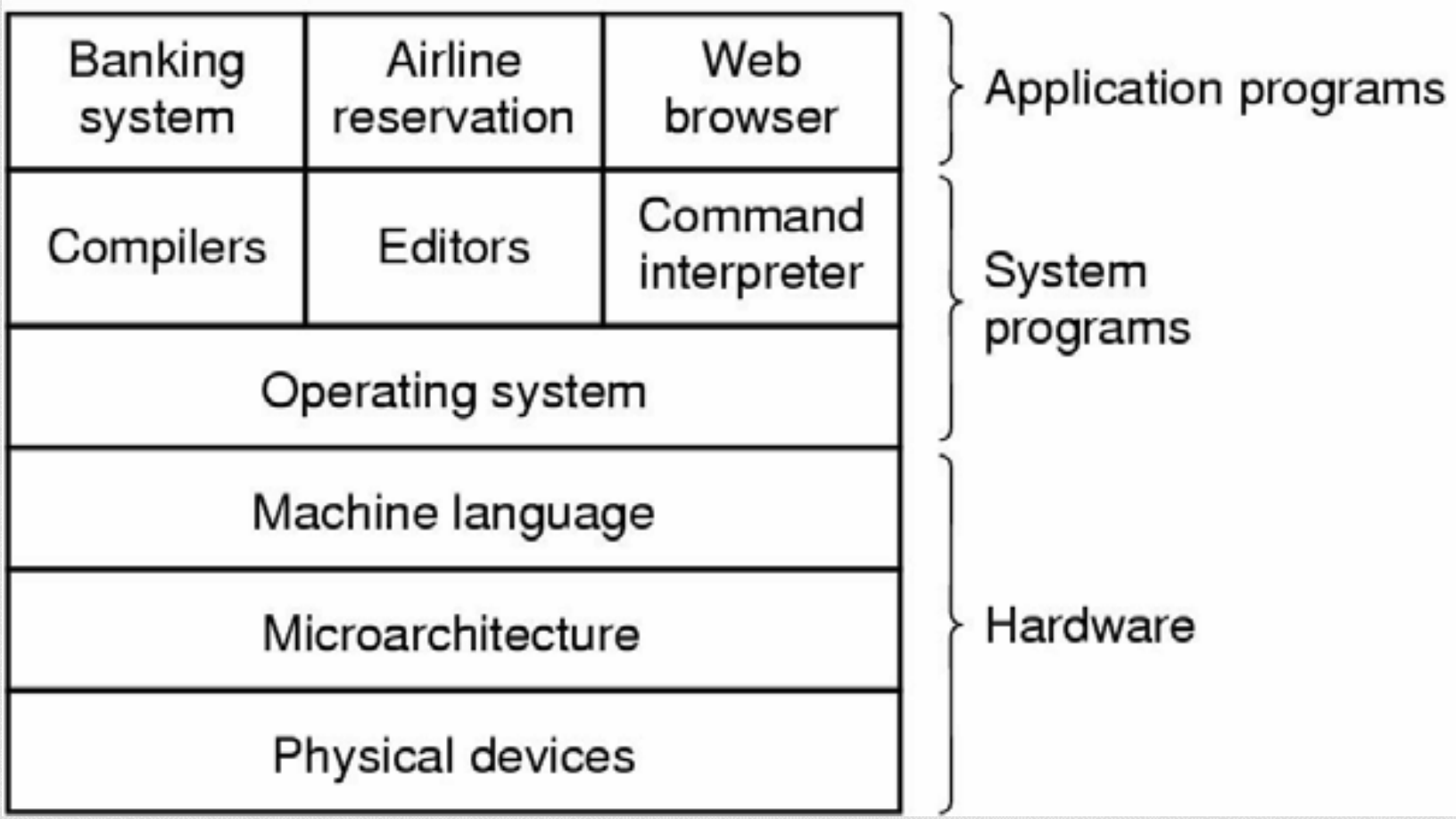
System Programming
(Programación de sistemas y redes)

Instituto Balseiro
Universidad Nacional de Cuyo

Conceptos Introductorios

- Computadora como sistema muy complejo.
  - Extremadamente difícil escribir programas que controlen todos los componentes y los utilicen en forma correcta.

- Sistema operativo: capa de software para gestionar los dispositivos. Proporcionar a los programas de usuario una interfaz con el hardware mas sencilla.


Banking system
Airline reservation
Web browser
Compilers
Editors
Command interpreter
Operating system
Machine language
Microarchitecture
Physical devices
Application programs
System programs
Hardware

Sistema Operativo como virtual machine

- Visión top-down
  - Presenta una abstracción de la complejidad para controlar dispositivos. (ej.: control de disco vs. visión de filesystem y funciones para manipularlo).
  - Presta una variedad de servicios que los programas pueden usar a través de instrucciones especiales conocidas como llamadas al sistema (system calls).

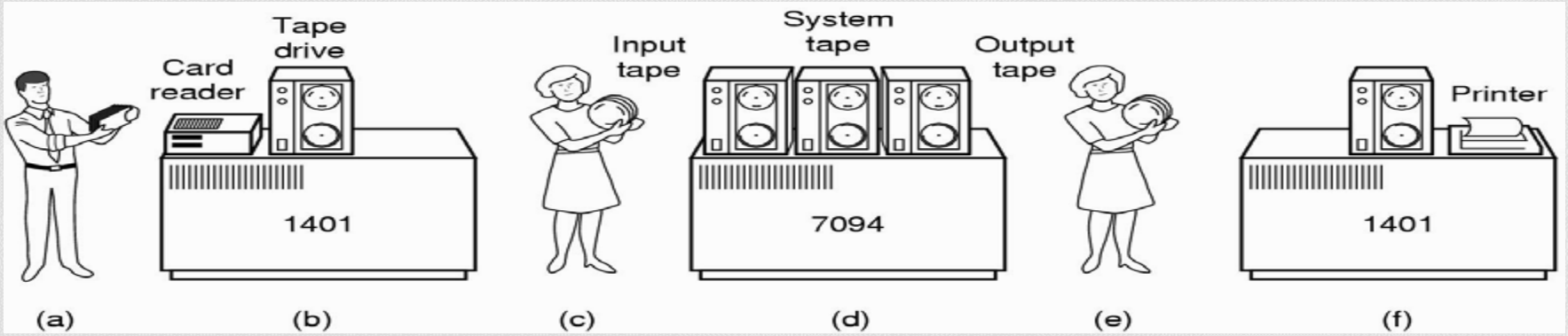
Sistema Operativo como gestor de recursos

- Visión botton-up
  - Administra todos los elementos de un sistema complejo.
  - Asegura el reparto ordenado y controlado de los procesadores, memoria y dispositivos de I/O entre los distintos programas que compiten por obtenerlo.

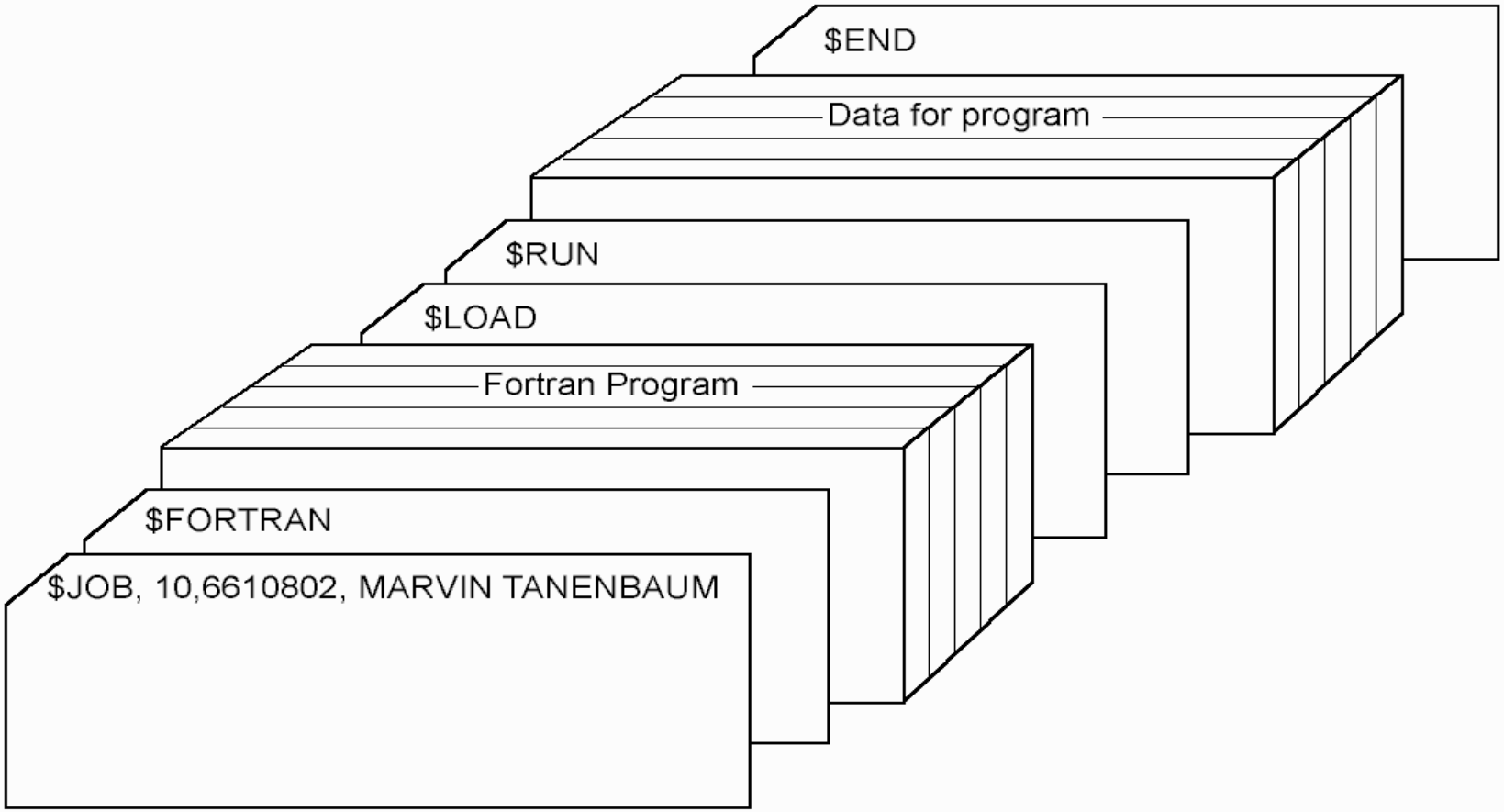
    - Multiplexado en tiempo (CPU, printer, net).
    - Multiplexado en espacio (memoria, disco).

Historia de los Sistemas Operativos

- Primer generación (1945-1955): Tubos de vacío y tableros de conexiones. Un único grupo de personas diseñaba, construía programaba, operaba y mantenía cada máquina.
- Segunda generación (1955-1965): Transistores y sistemas por lotes.



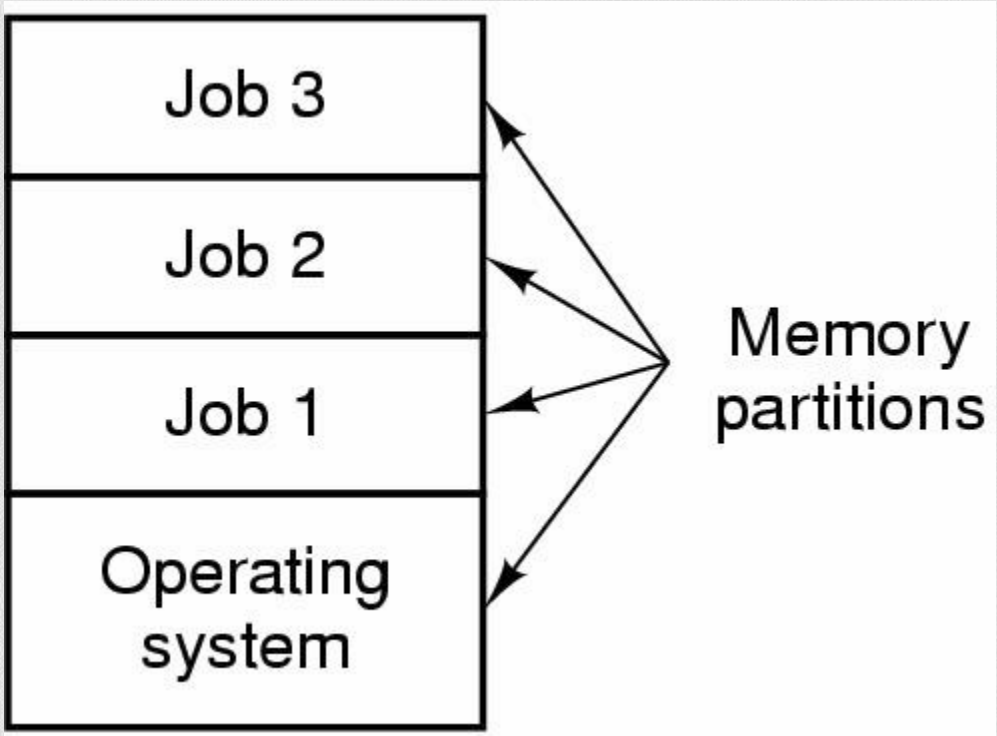
Estructura de un programa típico



Historia de los Sistemas Operativos

- Tercera generación (1965-1980): Circuitos integrados y multitareas. Unificación de línea baja y alta a través de una familia de computadoras. IBM 360.
  - Uso de circuitos integrados en lugar de transistores individuales.
  - Compatibilidad de software en toda la familia.
  - Mayor fortaleza como punto mas débil. (S.O. enormemente complejo -> lleno de errores).
  - Introducción de la Multitarea.

Particionado de memoria entre tareas



Historia de los Sistemas Operativos

- Problema de la no interactividad.
- Aparición de timesharing (multiuser). Necesidad de hardware de protección entre tareas.
  - CTSS (Compatible Time Sharing System) desde el MIT.
  - MULTICS(MULTIplexed Information and Computing Service) desde MIT, Bell, GE.
- Aparición de los miniordenadores. DEC PDP-1 (4K palabras de 18 bits a solo USD 120.000). Hasta el PDP-11.

Aparición de UNIX

- Ken Thompson, de Bell (había trabajado en el proyecto MULTICS), sobre un PDP-7 en desuso, reescribió una versión de MULTICS para un solo usuario, recortando partes del sistema original. Esto sentó la base para el sistema UNIX.
- El código fuente se podía conseguir fácilmente => varias compañías desarrollaron sus propias versiones (incompatibles) => caos. Las dos versiones principales fueron System V (de AT&T) y UNIX BSD (de Berkeley).

Aparición de UNIX

- Con el fin de compatibilizar programas para que se puedan ejecutar en variantes de UNIX; IEEE creó un estándar de UNIX, llamado POSIX.
- POSIX define una interfaz mínima de llamadas al sistema que deben entender los sistemas UNIX compatibles.
- Tanenbaum (1987) escribe un pequeño UNIX con fines educativos: MINIX, compatible con POSIX.
- Con el deseo de contar con una versión de producción libre de MINIX (no solo educativo), Linus Torvalds escribe LINUX.

Historia de los Sistemas Operativos

- Cuarta generación (1980-actualidad): Computadoras personales. Masificación.
  - 1974, Intel presentó el 8080. Contrato a Gary Kildall para que escribiera un S.O. para este procesador. Éste creó la compañía Digital Research, para desarrollar y vender el S.O. CP/M.
  - 1980, IBM diseñó la PC. Contactó cpm Bill Gates para utilizar su interprete BASIC. DR no quizo venderle CP/M. IBM pidió a Gates que desarrolle un S.O.

Historia de los Sistemas Operativos

- Gates fundó Microsoft, compró DOS a una compañía local. Contratando además a su desarrollador para introducir modificaciones solicitadas por IBM. Rebautizando al sistema MS-DOS. Decisión estratégica de vender el sistema a los fabricantes de computadoras y no a los usuarios directamente.
- Steve Jobs, tomando ideas de Xerox PARC, creó Lisa primero y Apple Macintosh luego.
- Microsoft creó Windows, primero como un shell arriba de MS-DOS, después como un S.O. completo.

Historia de los Sistemas Operativos

- Aparición de LINUX (en alguna de sus múltiples distribuciones) en el mundo de las computadoras personales, como contendiente importante frente al sistema de Apple y de Microsoft.

Conceptos de Sistemas Operativos

- Proceso: Programa en ejecución. Posee un espacio de direcciones. El espacio de direcciones contiene el programa ejecutable, sus datos y su stack. Cada proceso posee además un conjunto de registros.
- Thread: Hilo de ejecución dentro de un proceso.
- Scheduler: Componente interno del S.O. encargado de la distribución de tiempo del procesador (o procesadores) entre los diferentes procesos/threads que compiten por ese recurso. En su operación, el scheduler puede quitarle la CPU a un uno, para otorgárselo a otro.

Conceptos de Sistemas Operativos

- Context switch: Operación que se produce cuando se cambia la asignación de CPU de un proceso a otro. Involucra el salvado del contexto del proceso a quien se retira el recurso y la carga del contexto al proceso a quien se le reasigna. Costos.
- Signals: Análogo en software a las interrupciones de hardware.
- Filesystem. Archivo. Directorio. Servicios para su manejo.

Modos de operación de una CPU

- En general, las CPUs tienen dos modos principales de ejecución: Kernel y Usuario. Está definido por un conjunto de bits en el PSW.
  - Modo Kernel: Pueden ejecutar cualquier instrucción, incluyendo el acceso al hardware.
  - Modo Usuario: Pueden ejecutar solo un subconjunto del set de instrucciones. No pueden cambiar los bits de modo del PSW.

System Calls

- Para obtener un servicio del sistema operativo, el programa de usuario debe hacer una llamada al sistema, la cual realiza un trap dentro del kernel e invoca al sistema operativo.
- La instrucción TRAP cambia de modo usuario a kernel y cede el control al sistema operativo.
- Una vez completado el trabajo solicitado, se devuelve el control al programa de usuario justo en la instrucción siguiente al llamado al sistema.
- Existen otros TRAPs: interrupciones, excepciones, reset, etc.

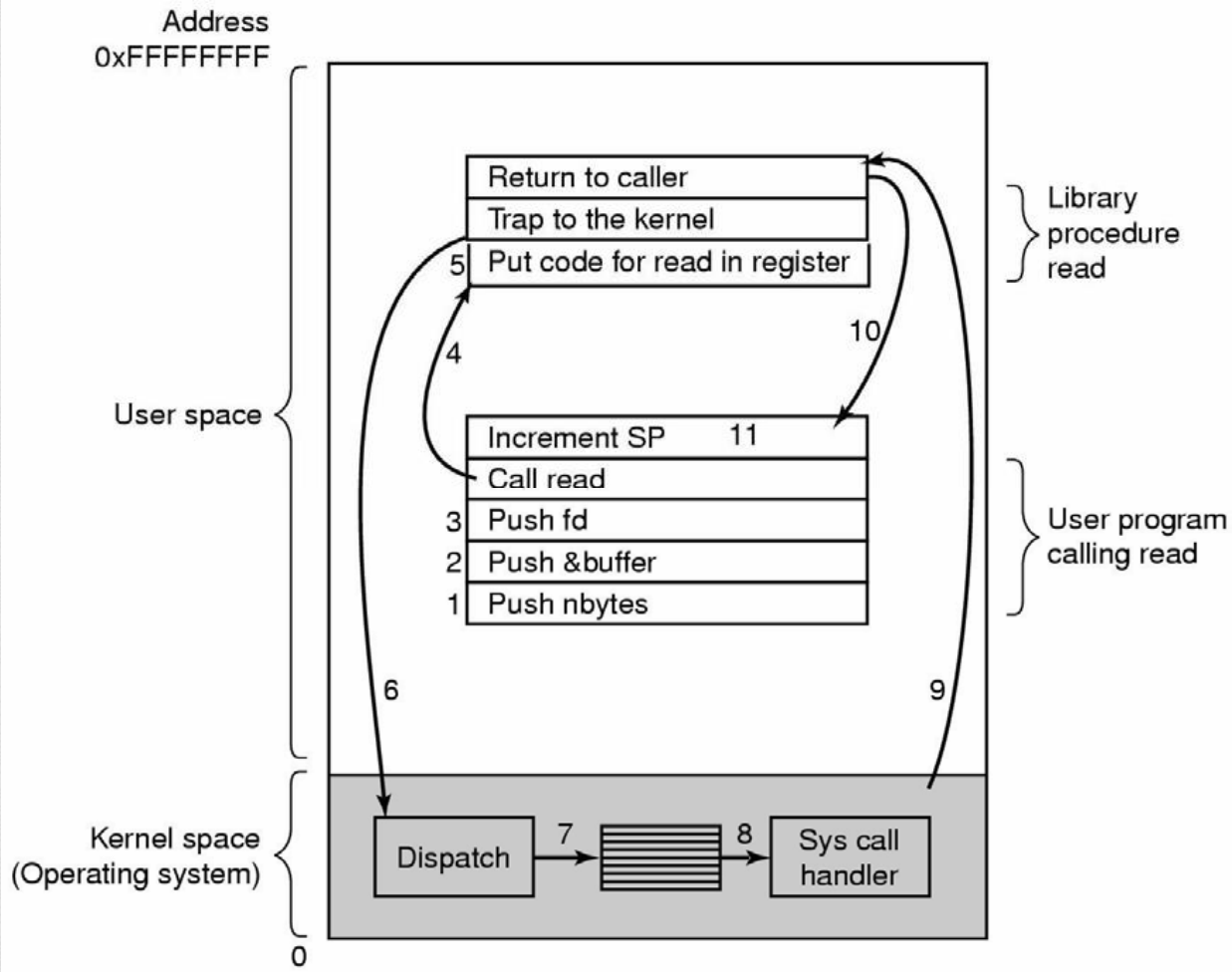
System Calls

- La interfaz entre el sitema operativo y los programas de usuario está definida por el conjunto de llamadas al sistema ofrecidas por el S.O. Varían de un S.O. a otro, aunque los conceptos subyacentes son similares.

- Ejemplo de un system call:

```
bytesRead = read(fd, &buffer, nBytes);
```

System Calls



Bibliografía

- Modern Operating Systems. Andrew Tanenbaum. ISBN-13: 978-0136006633.

File System

- Estructura para guardar y recuperar información, en dispositivos de almacenamiento.
- Concepto de archivos y directorios.
- Estructura de árbol.
- Muchos tipos de filesystems:
  - ext2/3/4
  - NTFS
  - FAT12/16/32/ExFAT
  - ISO9660/RR extensions/Joliet
  - nfs / CIFS

File System - Unix

- Ínodo y bloques de datos. Árbol único.
- El ínodo tiene toda la metadata del archivo.
- El nombre de un archivo no está en el ínodo. Está en los directorios, que no son más que archivos que tienen como dato un mapa de nombre a ínodo.
- Los archivos son “bag-o-bytes”. No tienen estructura.
- Existen archivos especiales:
  - Named fifos, sockets
  - Archivos de dispositivos. Bloques y caracteres.
  - Links simbólicos

File System API, unix vfs

- File descriptor o file handle: entero positivo.
- API básico:
  - create/ open / close
  - read / write
  - lseek, stat, unlink
- API avanzado:
  - dup, link, symlink, mknod, umask, fsync, flock
  - chmod, chown
  - mkdir, readdir (2), getdents, readdir (3)
  - fcntl, ioctl, pread, pwrite
  - mount, umount

man pages

- open / creat
- close
- read
- write
- lseek / lseek64
- pread/pwrite
- dup
- fcntl
- aio_*

Ejemplos

- open_test.c
- read_test.c
- seek_read.c
- fcntls.c
- locker0.c locker1.c
- aio_read.c

File System API, win32

- NTFS Streams, los files no son sólo bag-o-bytes.

- CreateFile / CloseHandle
- ReadFile / ReadFileEx
- WriteFile / WriteFileEx
- SetFileAttributes
- CreateDirectory
- GetFileInformationByHandleEx
- SetFileInformationByHandle

Memory Mapped Files

- El contenido de un archivo accedido a través de una dirección de memoria virtual en el proceso.
- mmap
- munmap
- msync
- mlock / munlock

- Mapping anónimo: sin file (fd=-1) -> allocación de memoria virtual
- Ejemplos: mmaper.c, mmap_find.c

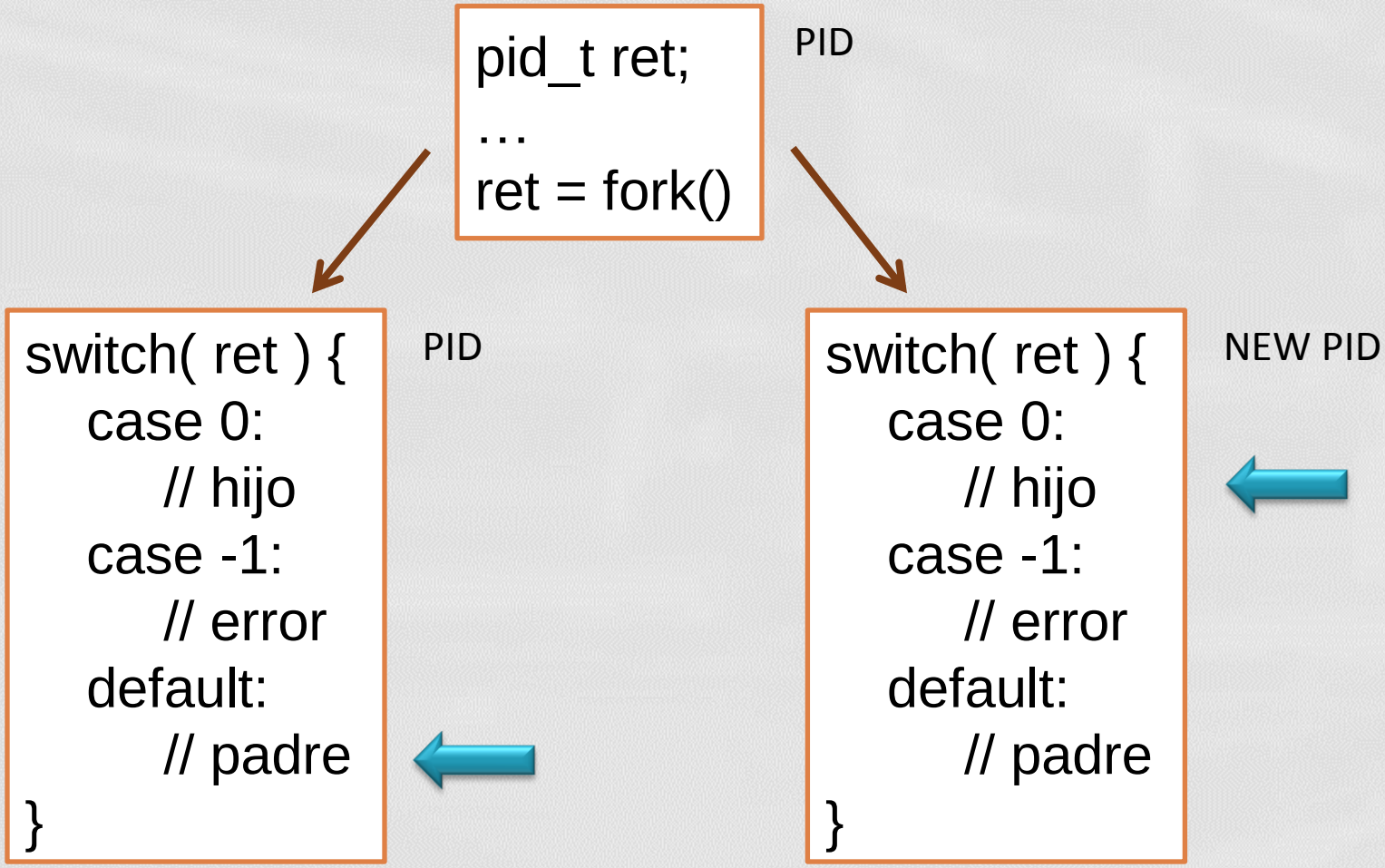
select

- Permite monitorear varios file descriptors esperando por que algun de ellos esté “ready”.
- Ejemplo: select.c

Creación de procesos

- En Unix: system call fork().
- Genera un nuevo proceso que es exactamente una copia del proceso que lo invoca.
- El nuevo proceso (hijo) tiene un nuevo PID.
- Al proceso ya existente (padre) fork() le retorna el PID del nuevo proceso.
- Al nuevo proceso hijo, fork() le retorna 0.
- En caso de error, retorna -1 y setea errno. No hay hijo.

fork()


pid_t ret;
…
ret = fork()
PID
switch(ret) {
case 0:
// hijo
case -1:
// error
default:
// padre
}
PID
switch(ret) {
case 0:
// hijo
case -1:
// error
default:
// padre
}
NEW PID

fork()

- Cuando un hijo termina, informa a su padre el estado de salida del proceso. El valor de retorno de main, “maquillado”.
- Si el padre no se da por enterado de la muerte de un hijo, el hijo queda en estado “zombie”.
- El padre espera por eventos de sus hijos con el syscall wait().

execve

- Reemplaza el proceso que se está ejecutando por un nuevo programa, especificado por nombre de archivo.
- En caso de error, retorna -1.
- En caso de éxito, no retorna.
- El programa tiene que ser un binario de formato reconocido o un script.
- Los script empiezan con:

  #! interprete [argumentos opcionales]
- Familia de funciones convenientes en libc:

  execl, exclp, execle, execv, execvp

signals

- Mecanismo muy básico de IPC.
- Se envía una señal de un proceso a otro. La señal está definida por un entero positivo chico.
- Números de señales predefinidos.
- Muchas señales son enviadas automáticamente por el sistema operativo ante ciertos eventos.
- Un proceso puede enviar señales a otros programáticamente, usando el syscall kill().
- Los procesos pueden elegir que hacer cuando les llega una señal determinada (no todas) con el syscall sigaction(). Handling asincrónico.

signals – man 7 signal

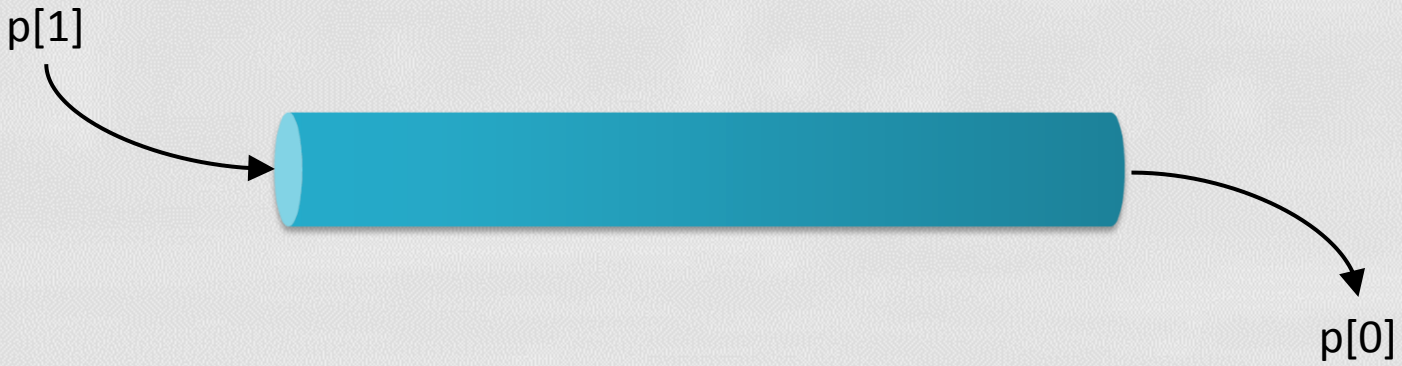
- No hay que usar el syscall signal() para cambiar el handling de las señales. Tiene comportamiento distinto según el flavor de unix. Usar sigaction().
- Para enviar señales, además de kill() existen:
  raise, killpg, pthread_kill, tgkill, sigqueue
- pause() y sigsuspend() sirven para esperar por señales.
- Las señales se pueden bloquear (enmascarar) con sigprocmask() y pthread_sigmask().
- No todas las funciones pueden ser llamadas desde un handler de señales.
- Los syscalls pueden ser interrumpidos por señales.

pipes

- pipe: canal de comunicación unidireccional.
- pipe() retorna 2 file descriptors que son la puntas del caño.
- Por un fd se escribe, por el otro se lee.
- El buffer está implementado en el kernel.
- Tratamiento standard sobre los 2 file descriptors: read, write, close, fcntl, select.
- SIGPIPE signals.

pipe()

```
int p[2];
…
ret = pipe(p);
```



pipes

- Uso a través de syscalls:

```
write(p[1], …);
read(p[0], …);
```

- Uso a través de funciones de libc:

```
FILE *fout = fdopen(p[1], "w");
fwrite/fprintf/...(fout, ...);
```

pipes

- Poco sentido usar pipe dentro de un único proceso (aunque puede convertirse en una buena forma (selectable) de comunicar comandos entre distintos threads.)
- Muy usado para comunicar un proceso padre con un proceso hijo. (pipe4.c)
- Establece una comunicación unidireccional. Para tener bidireccionalidad, hay que crear 2 pipes. (pipe5.c)

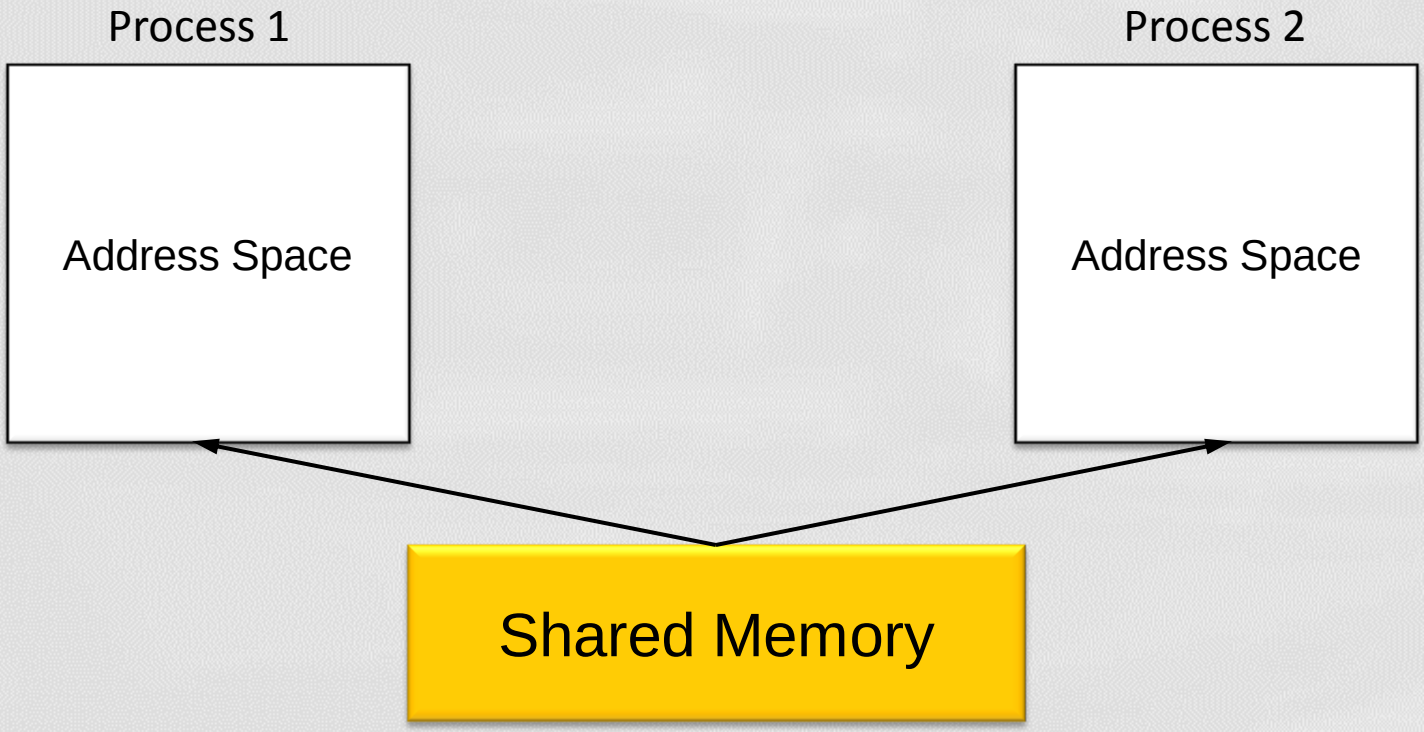
Named pipes: FIFOs

- Función **`mkfifo`**, crea un archivo especial FIFO.
- El Sistema operativo asocia un pipe a cada fifo.
- No hay storage en disco asociado al fifo.
- (Ejemplos **`fifow.c`** y **`fifor.c`**)

- **`open`** tanto de read como de write quedan bloqueados hasta que el otro extremo esté también conectado.

Shared Memory

- Mecanismo altamente eficiente de comunicación entre procesos



POSIX shared memory

```
int shm_open(const char *name,
             int oflag,
             mode_t mode);

int shm_unlink(const char *name);

int mmap(void *addr, size_t length,
         int prot, int flags,
         int fd, off_t offset);
int munmap(void *addr, size_t length);
```

Problema de acceso a recursos compartidos

Proceso 1	Proceso 2
`a = (*pdata);`	`b = (*pdata);`
`a++;`	`b--;`
`*pdata = a;`	`*pdata = b;`

- Necesidad de utilizar algun mecanismo de sincronización de acceso.

POSIX semaphores

```
#include <semaphore.h>

int sem_init(sem_t *sem, int pshared,
             unsigned int value);

int sem_wait(sem_t *sem);
int sem_trywait(sem_t *sem);
int sem_timedwait(sem_t *sem,
                  const struct timespec *absTime);

int sem_post(sem_t *sem);

int sem_destroy(sem_t *sem);
```

POSIX messages queue

- Mecanismo eficiente de IPC.
- Se crean y abren con **`mq_open`**. Esta función retorna un message queue descriptor (**`msd_t`**).
- Cada cola es identificada por un nombre ("**`/msq`**")
- Los mensajes son transferidos utilizando **`mq_send`** y **`mq_receive`**.
- Se cierran con **`mq_close`**.
- Se destruyen con **`mq_unlink`**.
- **`mq_getattr`** y **`mq_setattr`** para acceso a atributos.
- Cada mensaje tiene una prioridad asociada.

POSIX messages queue

- Interface en /proc
  - **`/proc/sys/fs/mqueue/msg_max`**
    Cantidad máxima de mensajes por cola
  - **`/proc/sys/fs/mqueue/msgsize_max`**
    Máximo tamaño de mensaje
  - **`/proc/sys/fs/mqueue/queues_max`**
    Número de colas que pueden ser creadas (system wide)

POSIX messages queue

- Message queue file system
  - Sobre Linux, las colas de mensajes son creadas en un virtual file system.

```
# mkdir /dev/mqueue
# mount -t mqueue none /dev/mqueue

$ cat /dev/mqueue/mymq
QSIZE:129 NOTIFY:2 SIGNO:0
NOTIFY_PID:8260
```

POSIX messages queue

```
#include <mqueue.h>

mqd_t mq_open(const char *name, int oflags);
mqd_t mq_open(const char *name, int oflags,
              mode_t mode, struct mq_attr *attr);

O_RDONLY
O_WRONLY
O_RDWR

O_NONBLOCK -> operaciones bloqueantes salen con EAGAIN
O_CREAT    -> se deben aplicar los otros 2 parámetros
O_EXCL
```

POSIX messages queue

```
#include <mqueue.h>

int mq_send(mqd_t q, const char *ptr,
            size_t len, unsigned msg_prio);

int mq_timedsend(mqd_t q, const char *ptr,
                 size_t len, unsigned msg_prio,
                 const struct timespec *absT);

ssize_t mq_receive(mqd_t q, const char *ptr,
            size_t len, unsigned *pMsg_prio);

ssize_t mq_timedreceive(mqd_t q, const char *ptr,
            size_t len, unsigned *pMsg_prio,
            const struct timespec *absT);
```

POSIX messages queue

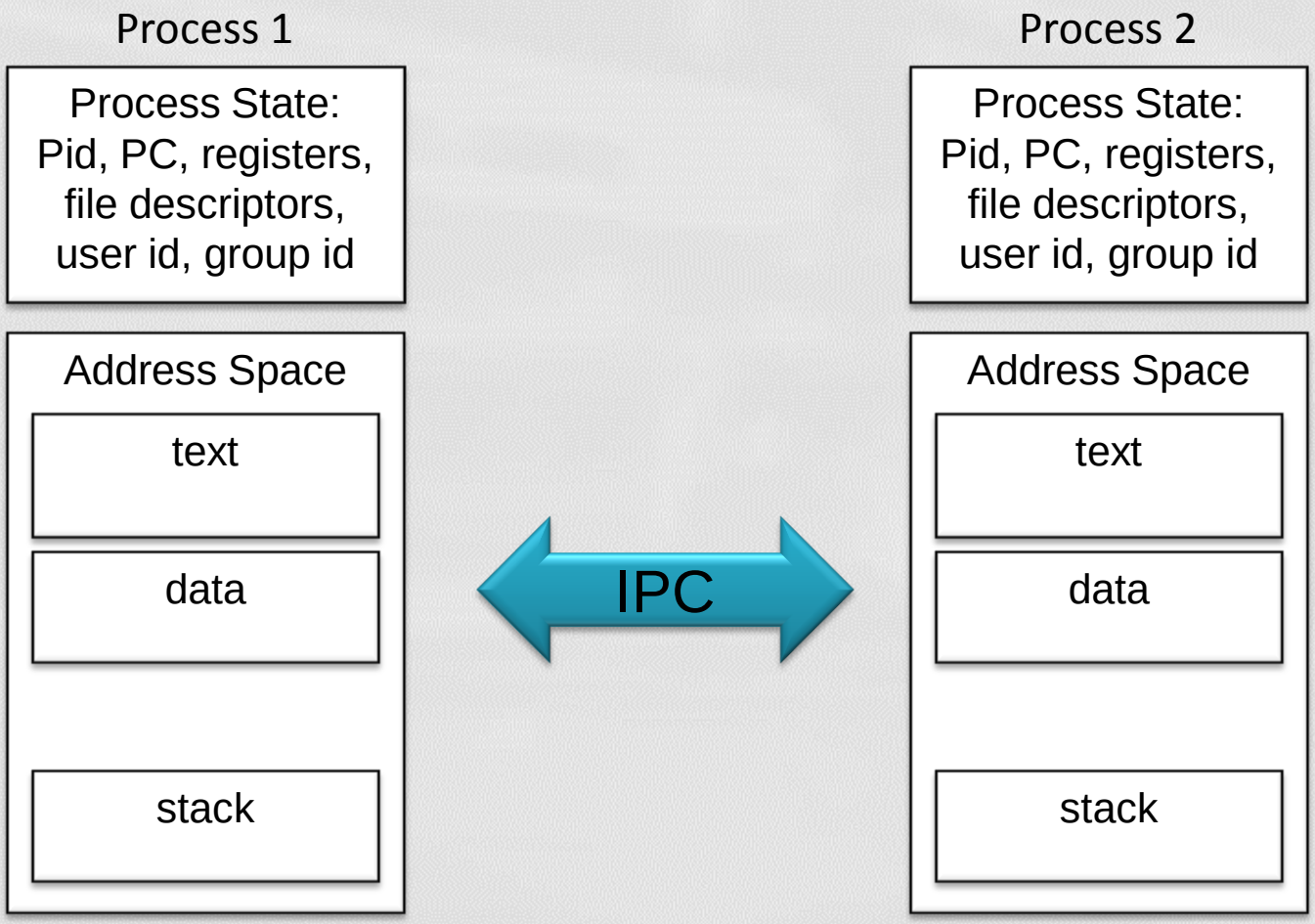
```
#include <mqueue.h>

int mq_getattr(mqd_t mqdes, struct mq_attr *attr);

int mq_setattr(mqd_t mqdes, struct mq_attr *newattr,
                    struct mq_attr *oldattr);


struct mq_attr {
    long mq_flags;   /* Flags: 0 or O_NONBLOCK */
    long mq_maxmsg;  /* Max. # of messages on queue */
    long mq_msgsize; /* Max. message size (bytes) */
    long mq_curmsgs; /* #of messages currently in q */
};
```

POSIX messages queue

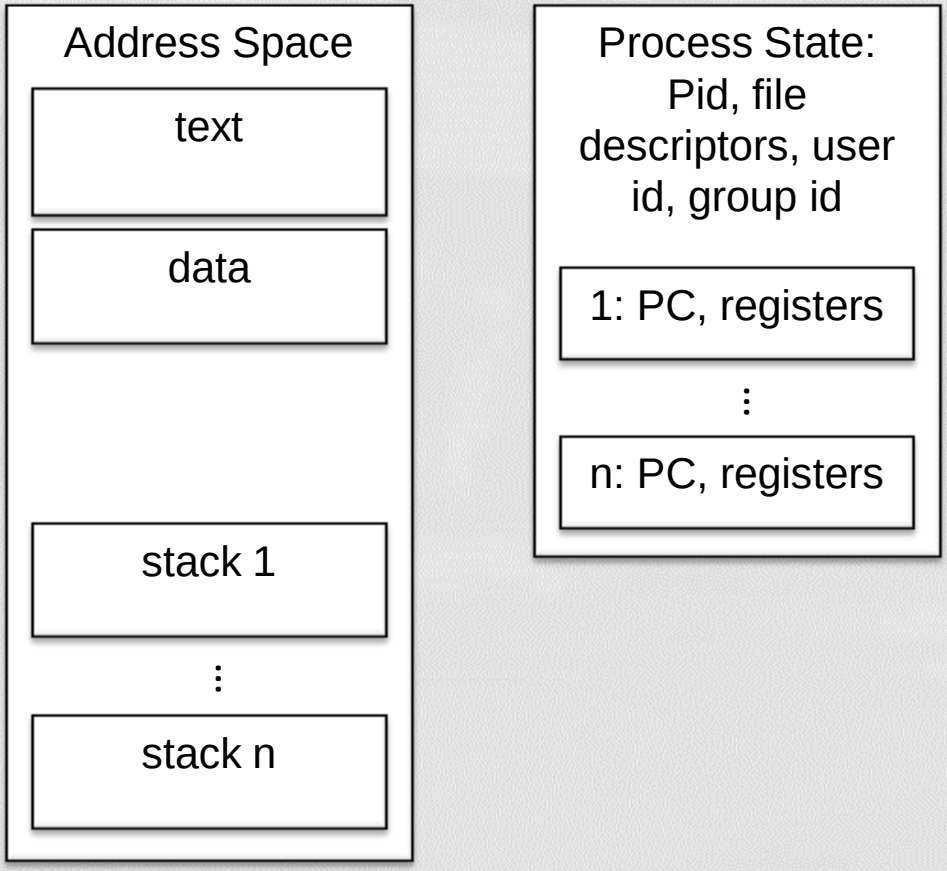
```
#include <mqueue.h>

int mq_notify(mqd_t mqdes,
              const struct sigevent *sevp);
```

Se registra o deregistra para la recepción asincrónica de notificaciones cuando un mensaje llega a una cola vacía.

Distintos procesos



Multithreading



Threads

- Cada thread mantiene sus propios:
  - Stack Pointer
  - Registers, PC
  - Scheduling properties
  - Signals behavior
  - Thread specific data
- Comparte:
  - Address Space
  - File descriptors
  - Process properties (pid, uid, gid, etc.)

Posix Threads

- Creación de un thread

```
#include <pthread.h>

int pthread_create(pthread_t *thread,
                   const pthread_attr_t *attr,
                   void *(*start_routine) (void *),
                   void *arg);
```

- Linkear con -lpthread

Posix Threads

- Terminación de un thread
  - Llamado explícito a:

```
void pthread_exit(void *retval);
```

  - Retorno de `start_routine`.
  - Cancelada con `pthread_cancel`.
  - Terminación del proceso.

- `pthread_self` retorna el pthread ID del calling thread.
- `pthread_equal` compara pthread IDs de manera portable.

Posix Threads

- Cuando un thread termina, el comportamiento es similar a los procesos zombies.
- `pthread_join` es similar a `wait` de un proceso hijo muerto. Espera por un thread terminado.

- Detached threads: liberación automática de recursos.
- No esperan por otro thread que las joinee.

- `pthread_detach`

- Una vez detachado un thread, no se puede joineear.

Posix Threads

- Thread attributes: `pthread_attr_t`
- Inicializados con `pthread_attr_init`
- Destruido con `pthread_attr_destroy`
- Atributos posibles:

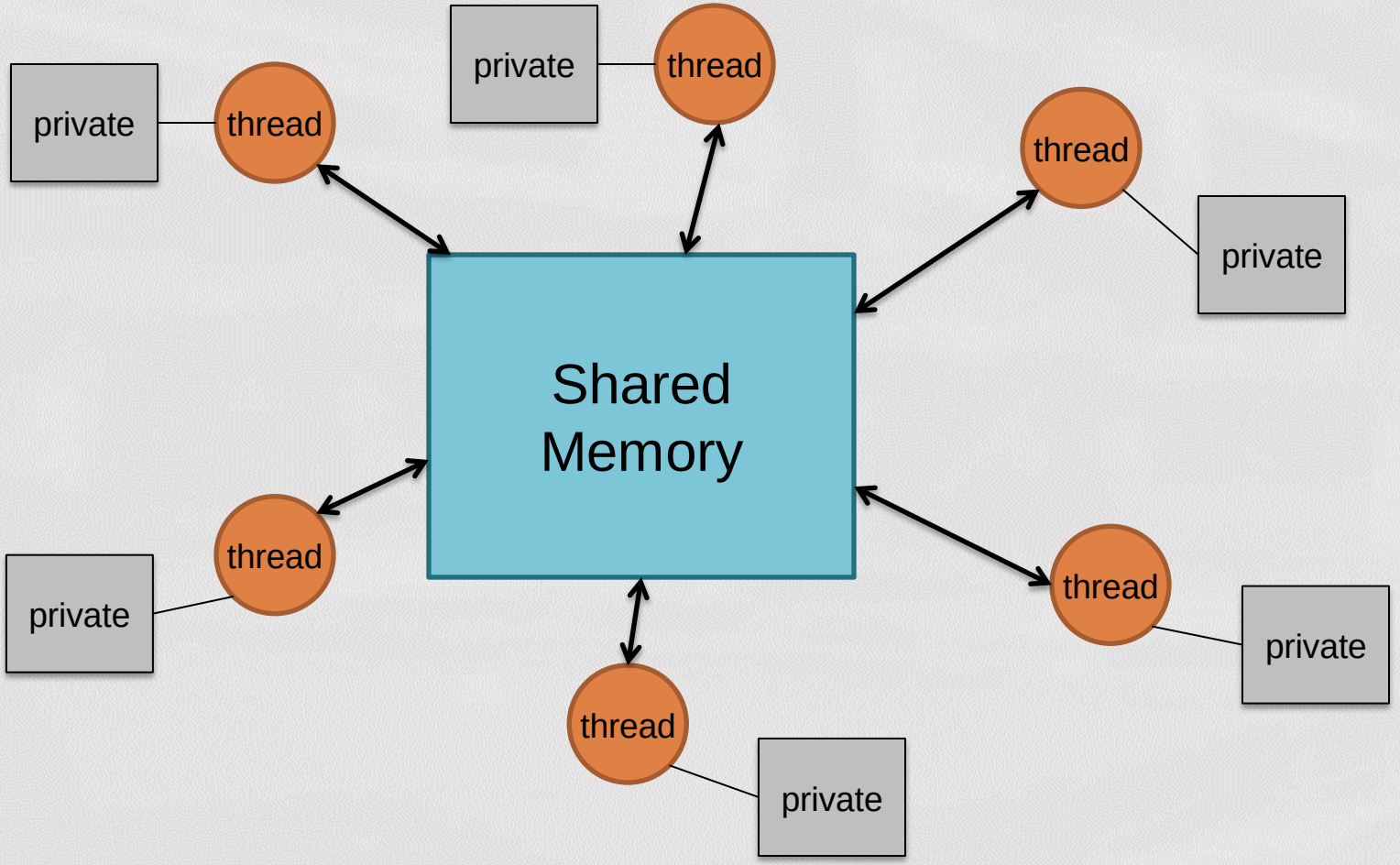
```
pthread_attr_setdetachstate
pthread_attr_setguardsize
pthread_attr_setinheritsched
pthread_attr_setschedparam
pthread_attr_setschedpolicy
pthread_attr_setscope
pthread_attr_setstack
pthread_attr_setstacksize
```

Threads

- Permiten paralelizar la programación. Modelos:
  - Manager/workers
  - Pipeline

- Ventajas en manejo de:
  - Eventos asincrónicos
  - Bloqueos por I/O.
  - Prioritización
  - Paralelismo en ejecución u operaciones sobre datos.

Threads – Shared Memory Model



Threads - Problemas

- Reentrancy: las funciones deben poder ser ejecutadas desde distintos threads concurrentemente.
  - No guardar estado estático entre llamados.
  - No retornar nada asociado a buffers propios.
    - `man -k _r`
- Thread safeness: protección de datos compartidos.
- Race conditions.
- Mutual exclusion. Serialización de acceso.

Mutex

- Recurso que permite sincronizar threads.
- Un solo thread puede tomar el mutex al mismo tiempo.
- Operaciones: lock y unlock.
- Similar a un semáforo con cuenta máxima de 1.
- API:

```
pthread_mutex_init
pthread_mutex_destroy

pthread_mutex_lock
pthread_mutex_trylock
pthread_mutex_timedlock

pthread_mutex_unlock
```

Condition Variables

- Recurso que permite sincronizar threads basándose en eventos. Un thread espera hasta que una condición sea cierta. De otra manera debería hacer polling.
- Las variables de condición están asociadas a un mutex.
- API:

```
pthread_cond_init
pthread_cond_destroy

pthread_cond_wait
pthread_cond_timedwait
pthread_cond_signal
pthread_cond_broadcast
```

Barriers

- Recurso que permite sincronizar threads basándose en que *n* threads arriben al punto de sincronización.
- API:

```
pthread_barrier_init
pthread_barrier_destroy

pthread_barrier_wait
```

Read/Write Locks

- Recurso que permite sincronizar threads permitiendo múltiples accesos de threads para lectura (compartido) y único para escritura (exclusivo).
- API:

```
pthread_rwlock_init
pthread_rwlock_destroy

pthread_rwlock_rdlock
pthread_rwlock_wrlock
pthread_rwlock_unlock

pthread_rwlock_tryrdlock      pthread_rwlock_trywrlock
pthread_rwlock_timedrdlock    pthread_rwlock_timedwrlock
```

Spinlocks

- Recurso que permite sincronizar threads.
- La espera por el recurso se hace “spinning” (busy waiting). Evita context switching.
- Tiene sentido en multiprocesadores.
- API:

```
pthread_spin_init
pthread_spin_destroy

pthread_spin_lock
pthread_spin_trylock
pthread_spin_unlock
```

OSI model

Open **S**ystems **I**nterconnection.

Data unit	Layer	Function
Data	7. Applicación	Network process <-> applicación
	6. Presentación	Representación dedatos, encripción/decripción. Conversión a machine independent data.
	5. Sesión	Comunicación entre hosts. Manejo de sesiones entre aplicaciones.
Segments	4. Transporte	Conexiones end-to-end. Control de flujo y veracidad.
Packet/datagram	3. Network	Determinación de camino y direccionamiento lógico.
Frame	2. Data link	Direccionamiento físico
Bit	1. Física	Medio, señalización y transmisión binaria.

Modelo en capas simplificado

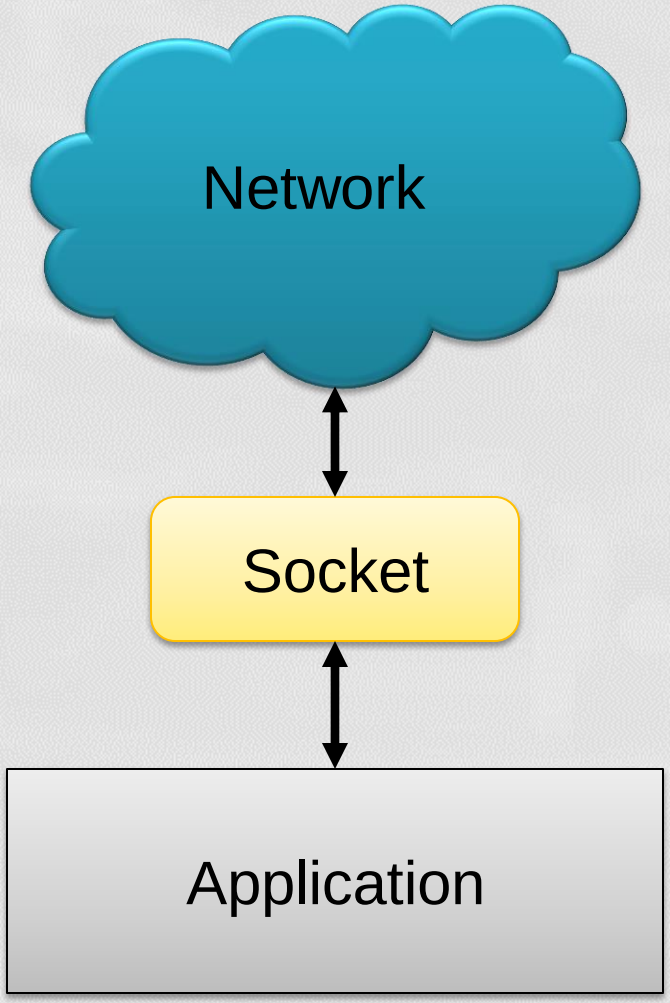
Layer	Function
Applicacion	telnet, ftp, http, etc.
Host-to-Host Transport	TCP, UDP
Internet	IP y routing
Network access	Ethernet, wi-fi, bluetooth, etc.

Sockets

- Provee una interface estándar para comunicar 2 procesos, tanto locales o ejecutándose en máquinas diferentes.
- Cuando un socket es creado, se debe especificar el dominio y el tipo. Dos procesos se comunican solo si sus sockets son del mismo dominio y tipo.
- Dominios principales:
  - UNIX domain: los procesos utilizan una entrada en el filesystem como dirección.
  - Internet domain: cada host tiene su IP address (32 bits). Es necesario además un port sobre ese host.

Sockets en Internet domain

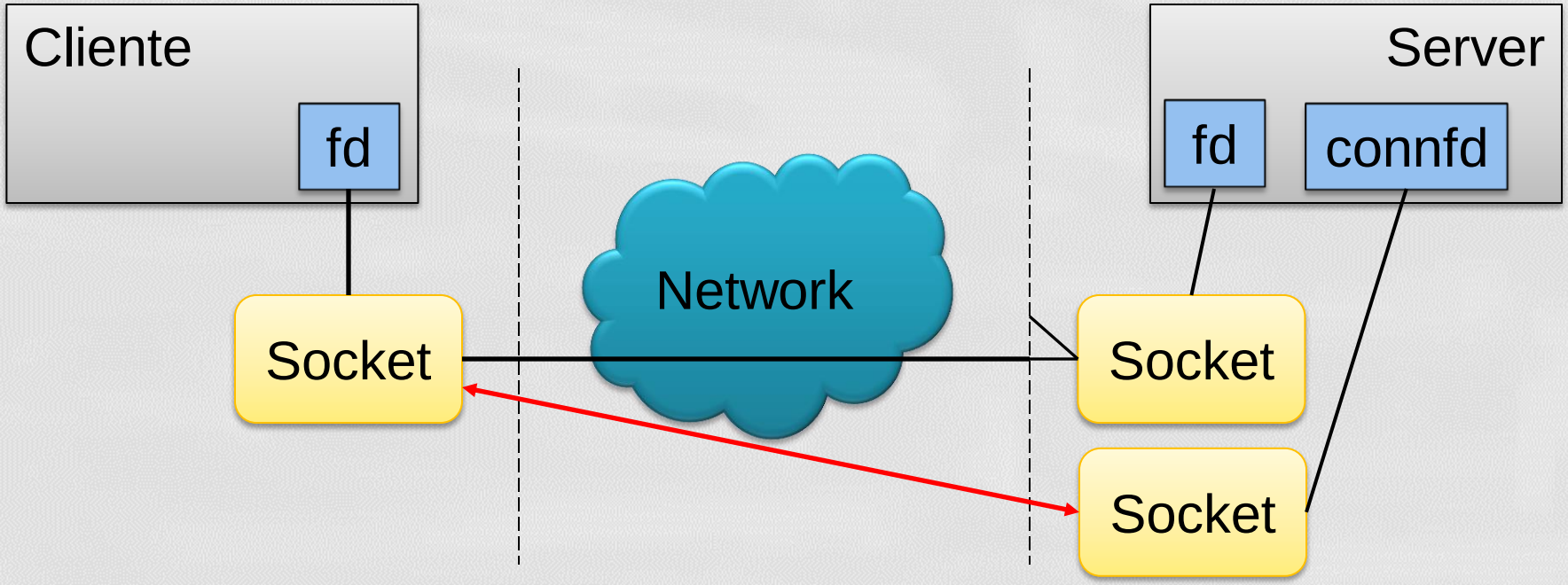
- Provee una interface estándar entre la aplicación y la red.
- 2 tipos:
  - Stream: utilizan TCP (transmission Control Protocol). provee servicio de circuitos virtuales
  - Datagram: utilizan UDP (Unix Datagram Protocol) envío de paquetes individuales.



TCP/IP Connection

- Conexión como stream de bytes, confiable entre 2 computadoras.
  - Comúnmente usados en topologías cliente-servidor:
    - Un server escucha en un puerto determinado.
    - El cliente se conecta a ese puerto.
    - Una vez que se estableció la conexión, cualquiera de los lados puede escribir y/o leer.
- El API de sockets representa la conexión a través de un file descriptor.

TCP/IP Connection



TCP/IP Connection

```
int fd = socket(…);



connect(fd, …, …);

write(fd, data, datalen);

read(fd, buffer, buflen);

close(fd);
```

```
int fd = socket(…);

bind(fd, …, …);

listen(fd, …);

connfd = accept(fd, …);

read(connfd, buffer, buflen);

write(connfd, data, datalen);

close(connfd);
```

Creación de un socket

```
#include <sys/types.h>
#include <sys/socket.h>

...
...

int fd;
fd = socket(family, type, protocol);

if( fd == -1 ) {
    // Error: unable to create socket
    // actual error in "errno"...
}
...
```

AF_UNIX -> Unix socket
AF_INET -> IPv4
AF_INET6 -> IPv6

SOCK_STREAM -> TCP
SOCK_DGRAM -> UDP

0 (no usado en internet sockets)

Crea un socket no ligado ni conectado a la red. Puede utilizarse tanto como server o como cliente. man page.

Binding de un server socket

```
#include <sys/types.h>
#include <sys/socket.h>

...
...
if( bind(fd, (struct sockaddr *) &serv_addr, sizeof(serv_addr)) == -1) {
    // Error: unable to bind
    // actual error in "errno"...
}
...
```

- Necesario para servers. Generalmente no usado en clientes dado que tipicamente no importa que puerto local es utilizado.
- man page

listen sobre un server socket

```
#include <sys/types.h>
#include <sys/socket.h>

...
...
if( listen(fd, 3) == -1) {
    // Error: unable to listen
    // actual error in "errno"...
}
...
```

- Marca al socket como aceptando conexiones.
- man page

connect

```
#include <sys/types.h>
#include <sys/socket.h>

...
...
if (connect(fd,(struct sockaddr *)&serv_addr,sizeof(serv_addr)) < 0) {
    // Error: unable to listen
    // actual error in "errno"...
}
...
```

- Intentará hacer una conexión sobre un socket.
- man page

accept

```
#include <sys/types.h>
#include <sys/socket.h>

...
...
clilen = sizeof(cli_addr);
connfd = accept(sockfd, (struct sockaddr *) &cli_addr, &clilen);
...
```

- Extrae la primer conexión de la cola de conexiones pendientes o espera si **`O_NONBLOCK`** no se impuso en el descriptor (**`fcntl`**).
- man page

send

```
#include <sys/types.h>
#include <sys/socket.h>

...
...
ret = send(sockFd, &buff, length, flags);
...
```

- Inicia la transmisión de un mensaje desde el socket hacia su partner.
- Es equivalente al syscall write si flags == 0
- man page

sendto

```
#include <sys/types.h>
#include <sys/socket.h>

...
...
ret = sendto(sockFd, &buff, length, flags,
(const struct sockaddr *) &destAddr, sizeof(destAddr));
...
```

- man page

sendmsg

```
#include <sys/types.h>
#include <sys/socket.h>

...
...
ret = sendmsg(sockFd, &msgHdr, flags);
...
```

- man page

struct msghdr

```
struct iovec {                    /* Scatter/gather array items */
    void  *iov_base;              /* Starting address */
    size_t iov_len;               /* Number of bytes to transfer */
};

struct msghdr {
    void         *msg_name;       /* optional address */
    socklen_t     msg_namelen;    /* size of address */
    struct iovec *msg_iov;        /* scatter/gather array */
    size_t        msg_iovlen;     /* # elements in msg_iov */
    void         *msg_control;    /* ancillary data, see below */
    socklen_t     msg_controllen; /* ancillary data buffer len */
    int           msg_flags;      /* flags on received message */
};
```

recv

```
#include <sys/types.h>
#include <sys/socket.h>

...
...
ret = recv(sockFd, &buff, length, flags);
...
```

- Inicia la transmisión de un mensaje desde el socket hacia su partner.
- Es equivalente al syscall write si flags == 0
- man page

recvfrom

```
#include <sys/types.h>
#include <sys/socket.h>

...
...
ret = recvfrom(sockFd, &buff, length, flags
                (struct sockaddr *) &destAddr, &destAddrLen);
...
```

- man page

recvmsg

```
#include <sys/types.h>
#include <sys/socket.h>

...
...
ret = recvmsg(sockFd, &msgHdr, flags);
...
```

- man page

shutdown

```
#include <sys/types.h>
#include <sys/socket.h>

...
...
ret = shutdown(sockFd, SHUT_RDWR);
...
```

- man page

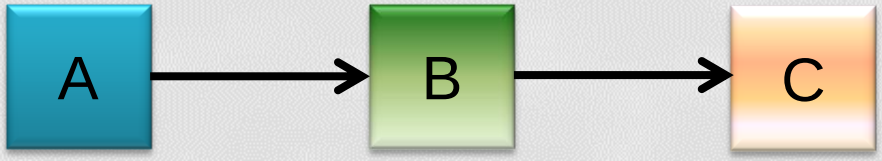
setsockopt/getsockopt

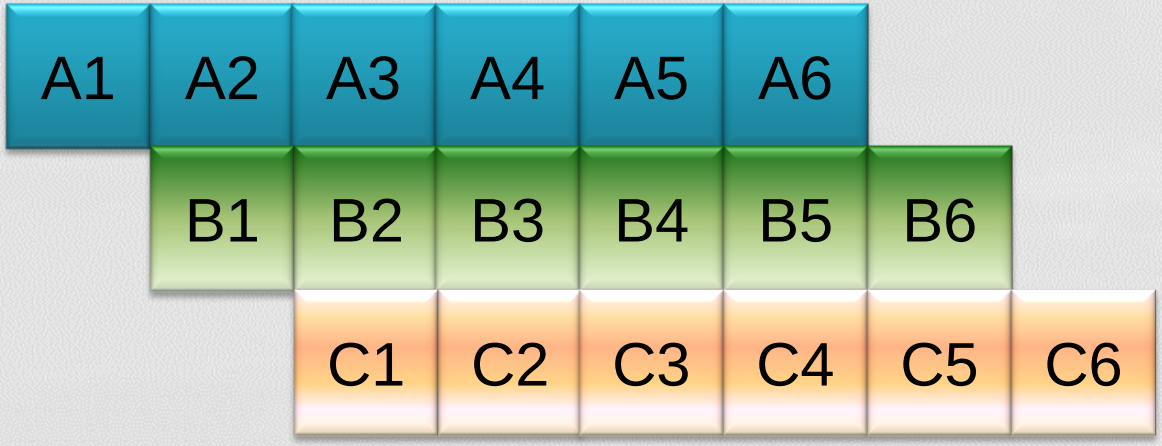
```
#include <sys/types.h>
#include <sys/socket.h>

...
...
ret = recvfrom(sockFd, &buff, length, flags
                (struct sockaddr *) &destAddr, &destAddrLen);
...
```

- man page

Pipeline

- Conjunto de elementos de procesamiento conectados en serie. La salida de un elemento es la entrada del siguiente. (Ejemplo de una línea de armado)





Pipeline

- Aspectos claves en el diseño de un pipeline:
  - Balanceo de etapas (a tiempos iguales hay throughput máximo y requerimiento mínimo buffering.
  - Buffering entre etapas. Especialmente si hay tiempos irregulares de procesamiento o creación/destrucción de elementos a lo largo del pipeline

Pipeline

- La implementación de pipelines en software tiene sentido en maquinas multiprocesador.
- Su implementación no es necesariamente lineal.