Porque es necesario un Sistemas Operativos

- **Como una máquina extendida**
  - La función del sistema operativo es presentar al usuario el equivalente de una **máquina extendida** o **máquina virtual** que es más fácil de programar que el hardware subyacente

- **Como gestor de recursos**
  - Asegurar un reparto ordenado y controlado de los procesadores, memorias y dispositivos de E/S, entre los diversos programas que compiten por obtenerlos

Conceptos de Sistemas Operativos

- **Proceso**: Programa en ejecución. Posee un espacio de direcciones propias del proceso. El espacio de direcciones contiene el programa ejecutable, sus datos y su(s) stack(s). 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 uno, para otorgárselo a otro.

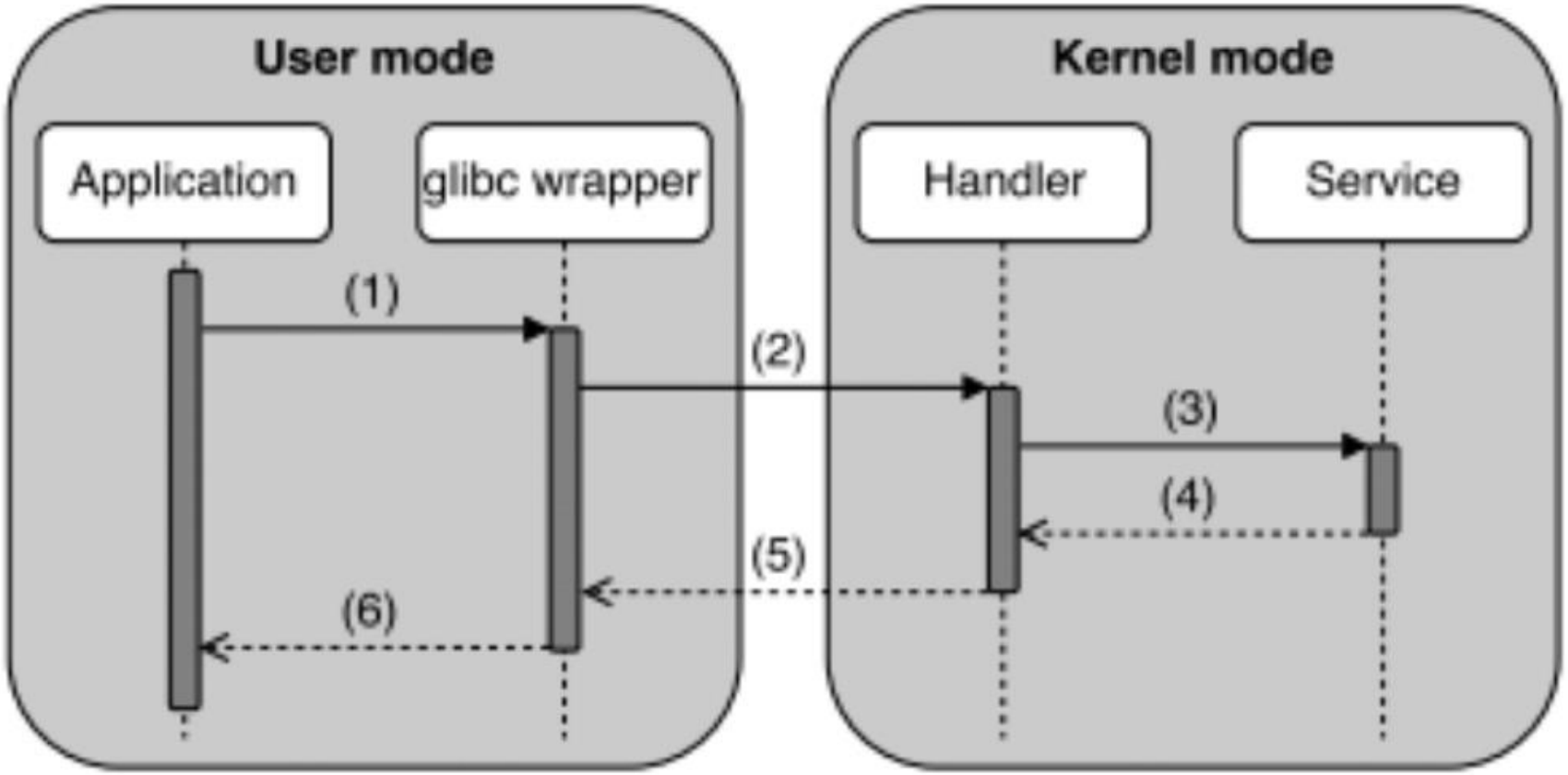
Modos de operación de una CPU

- En general, las CPUs tienen dos modos principales de ejecución: Kernel y Usuario. Está definido por un conjunto de bits en el PSW.
  - **Kernel Mode**: Pueden ejecutar cualquier instrucción, incluyendo el acceso al hardware. Un crash en kernel mode puede ser catastrófico
  - **User Mode**: Pueden ejecutar solo un subconjunto del set de instrucciones. No pueden cambiar los bits de modo del PSW. Debe delegar en funciones de sistema para el acceso al hardware.

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



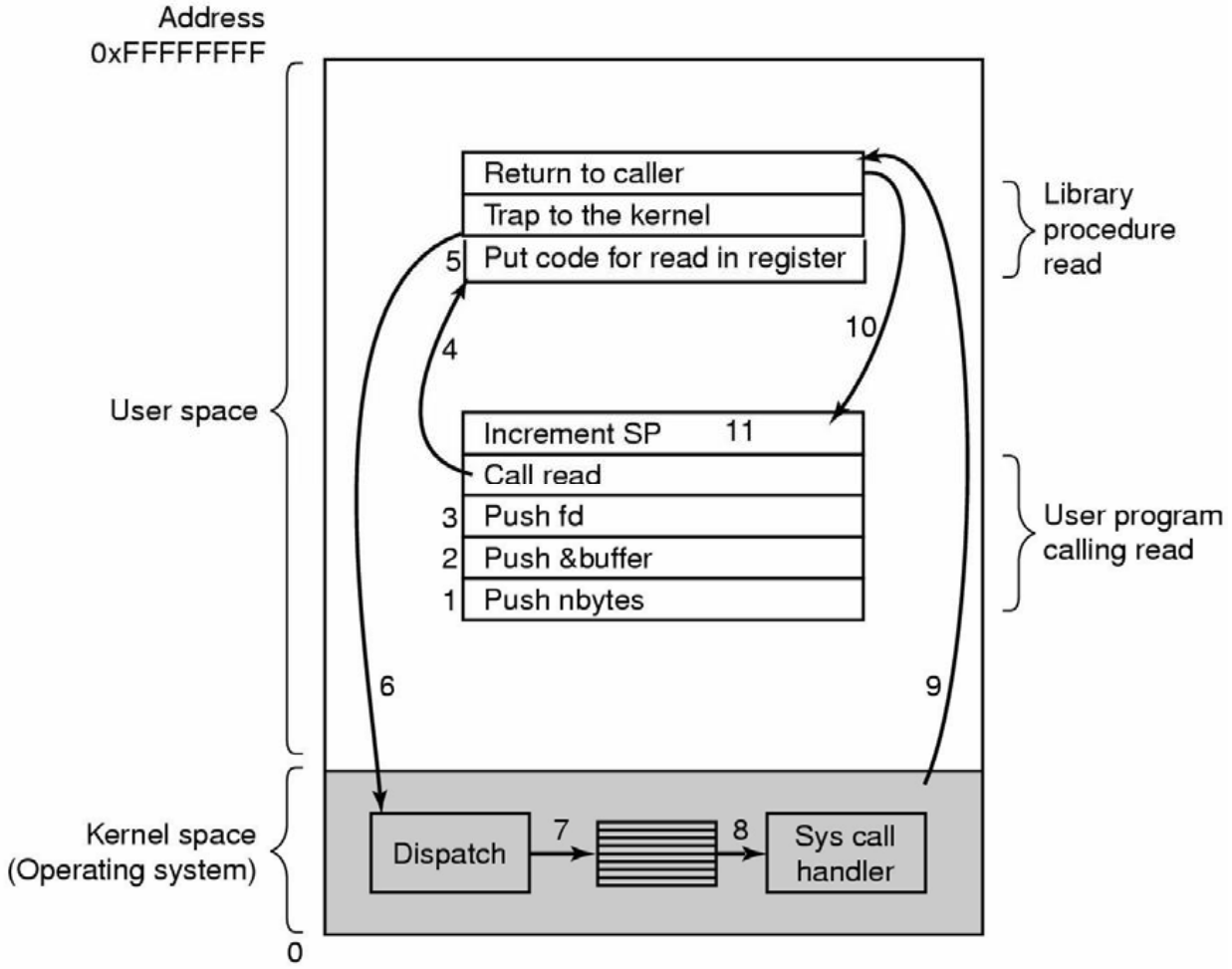
System Calls

- La interfaz entre el sistema 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




```
(0) bytesRead = read(fd, &buffer, nbytes);
```

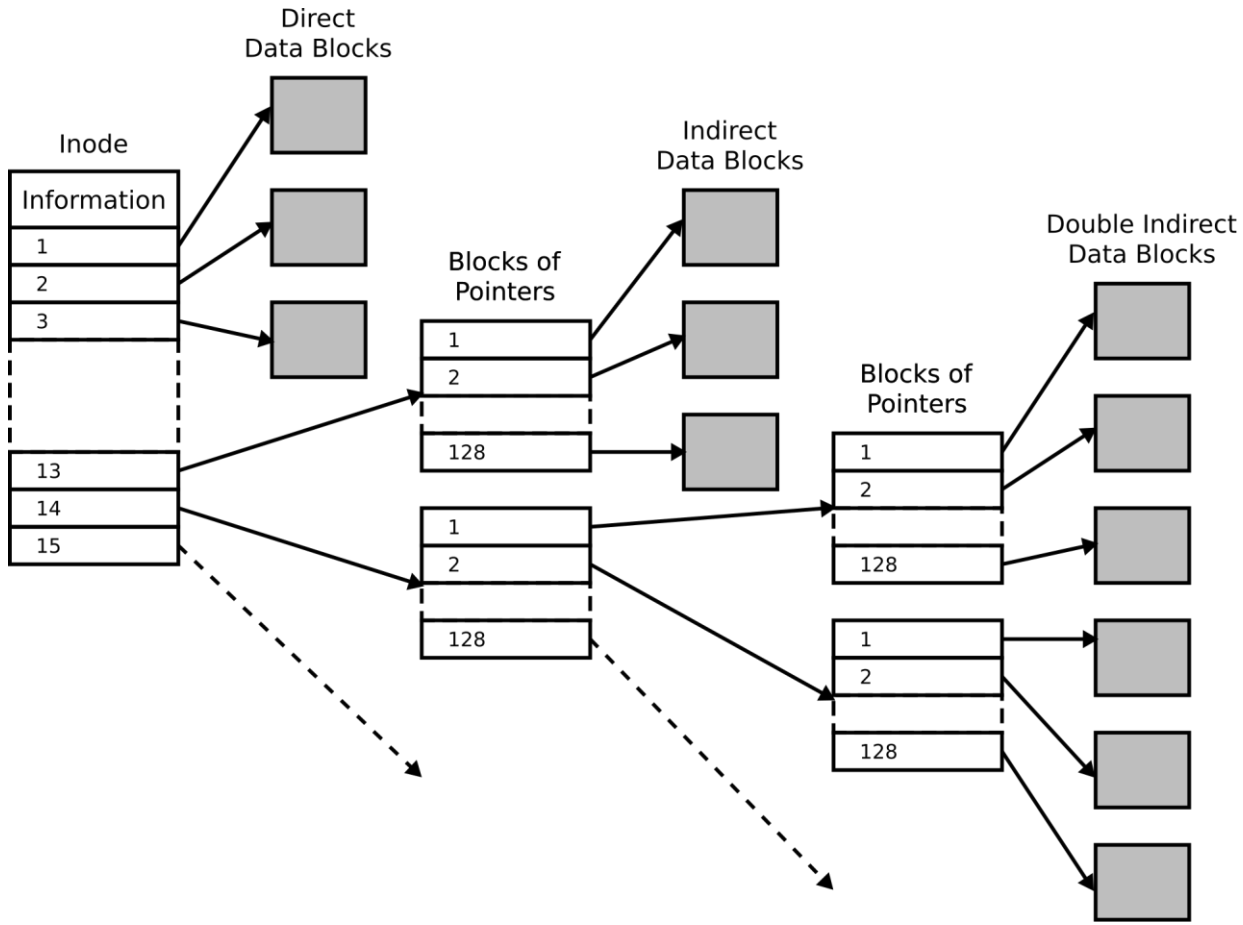
File System

- 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
- Comparación de filesystems

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/hard links

Ejemplo de ínodo



man pages

```
man man
man [section] page
```

Section	Description
1	General commands
2	System calls
3	Library functions, covering in particular the C standard library
4	Special files (usually devices, those found in /dev) and drivers
5	File formats and conventions
6	Games and screensavers
7	Miscellanea
8	System administration commands and daemons

File System API, Unix vfs

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

select

- Permite monitorear varios file descriptors esperando por que alguno de ellos esté “ready”.

- Ejemplo: select.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

mmap

- System call para crear un nuevo mapping en el virtual address space del proceso que llama al servicio. (**man 2 mmap**)

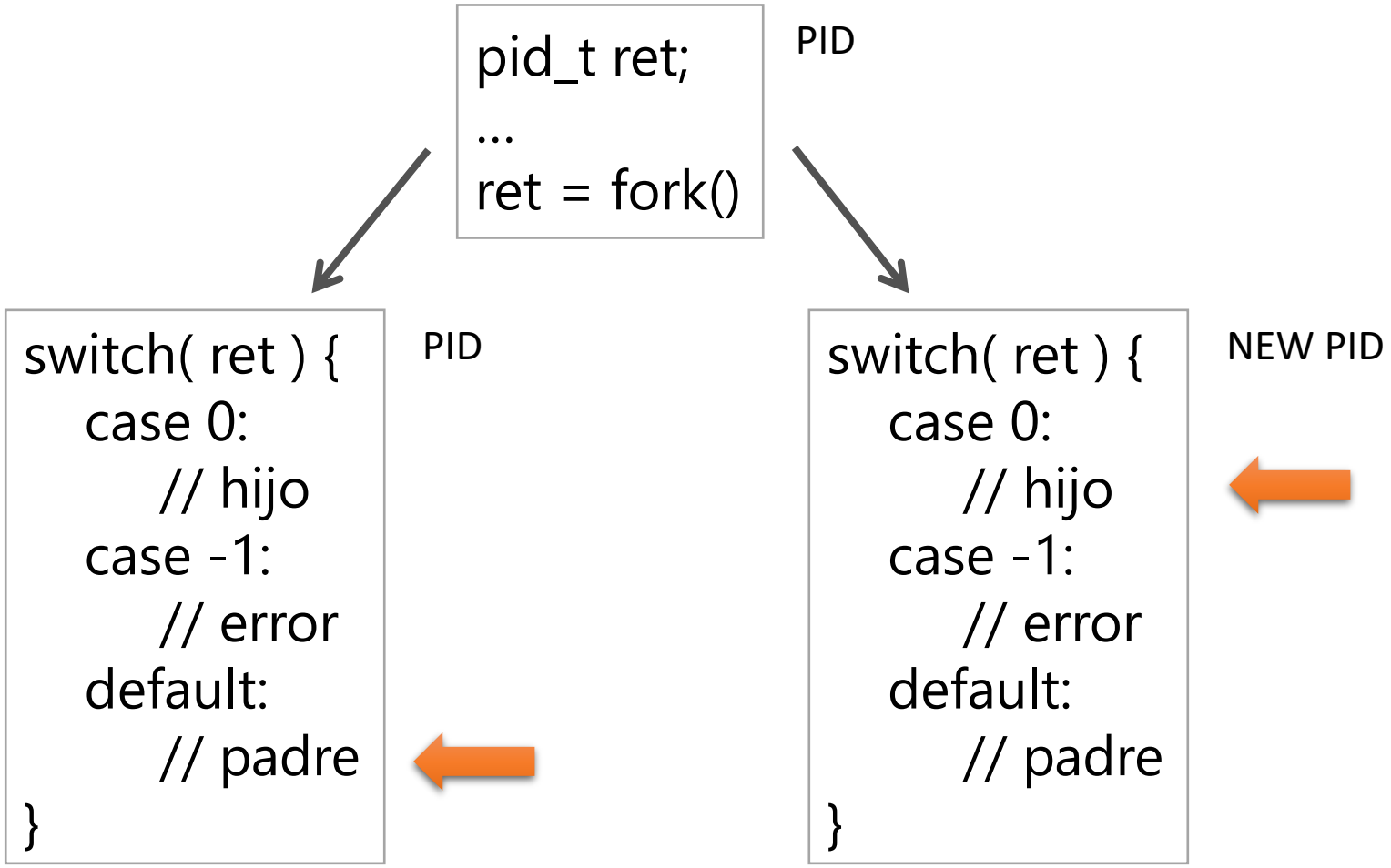
```
void *mmap(void *addr, size_t length, int prot, int flags,
           int fd, off_t offset);

int munmap(void *addr, size_t length);
```

Ejemplos: mmaper, mmap_find y comparación con la seek_test

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

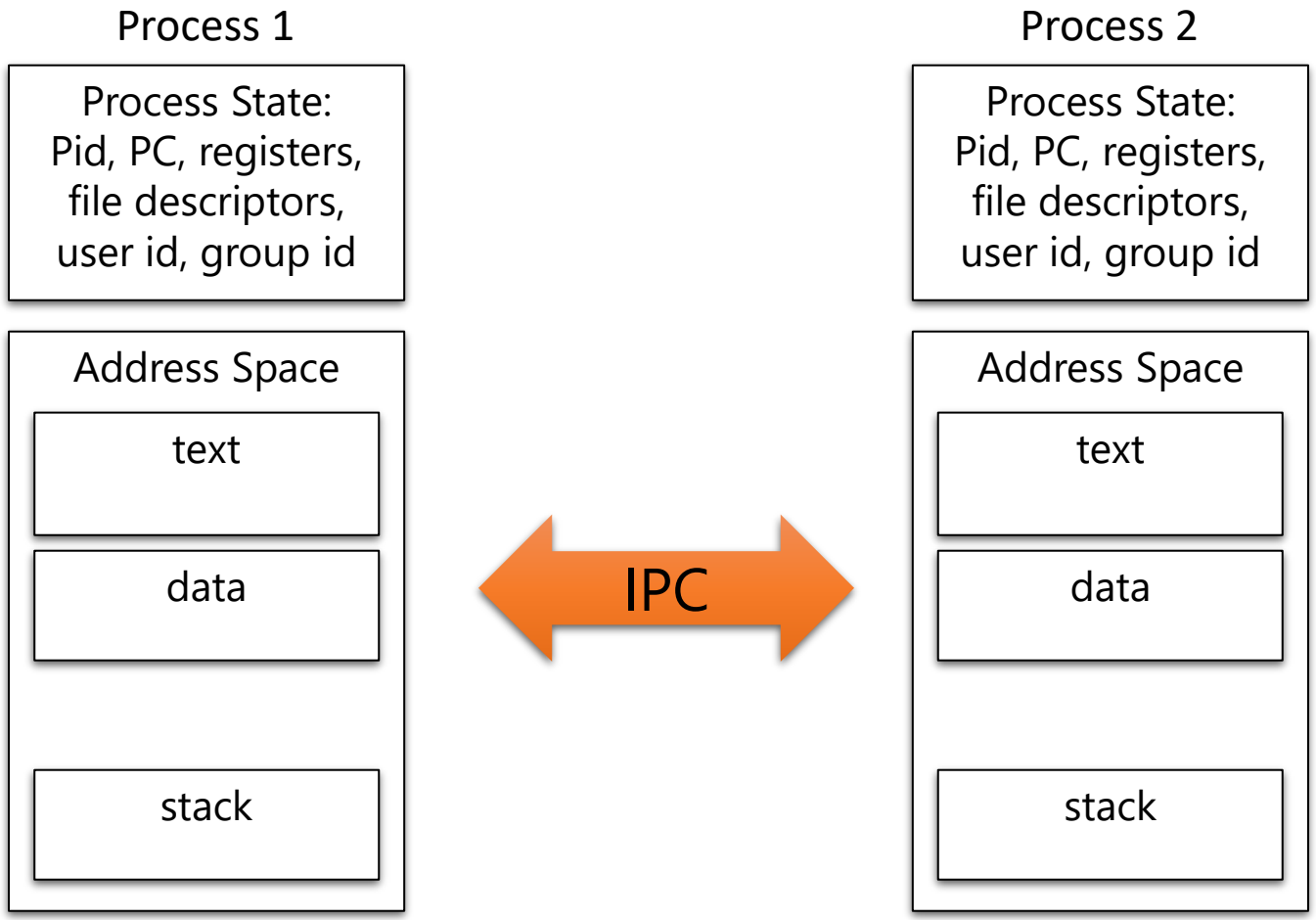
fork()

- 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 el 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

Distintos procesos



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

- 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.

pipes

```
int p[2];
...
ret = pipe(p);
```

pipes

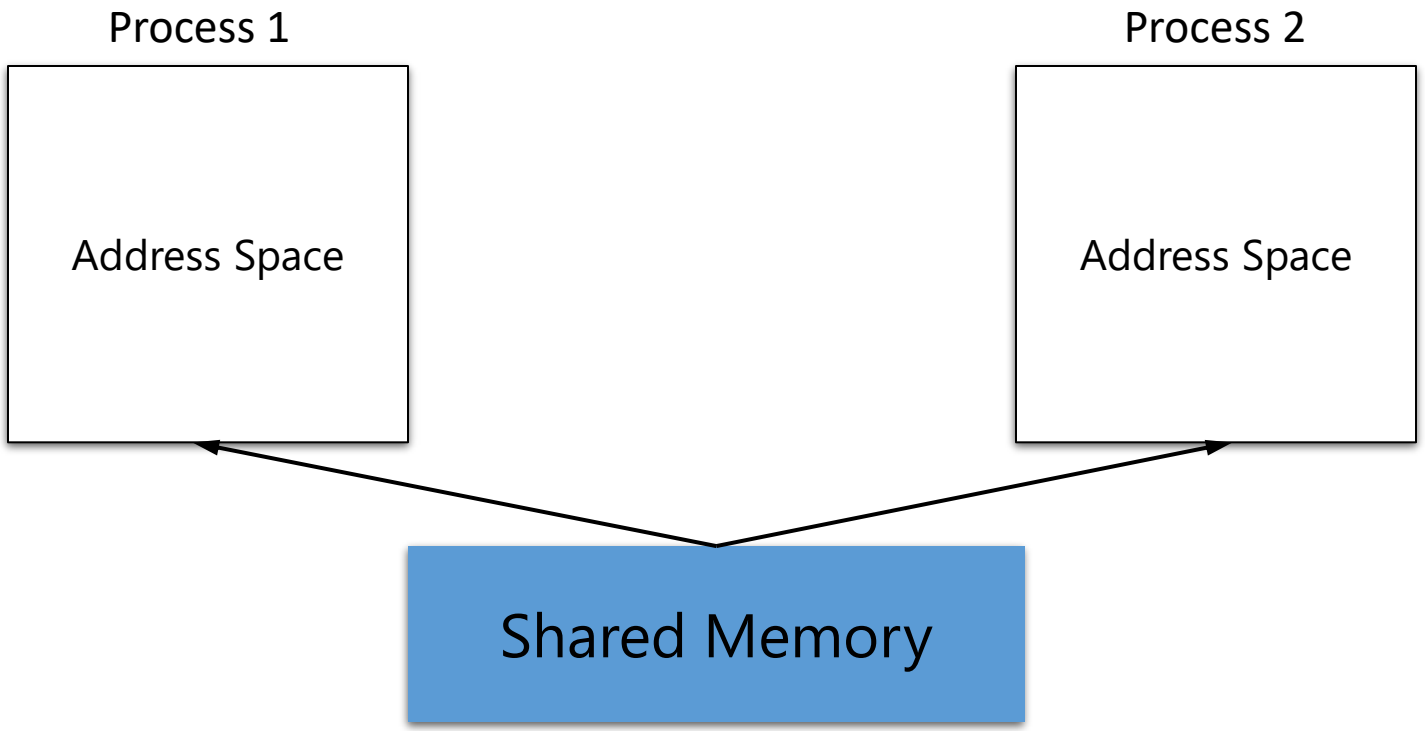
- Poco sentido usar pipe dentro de un único proceso (aunque puede convertirse en una buena forma (selectable) de comunicar comandos entre distintos threads.)

- Muy usado para comunicar un proceso padre con un proceso hijo.

- Establece una comunicación unidireccional. Para tener bidireccionalidad, hay que crear 2 pipes.

Shared Memory

- Mecanismo altamente eficiente de comunicación entre procesos.



Problema de acceso a recursos compartidos

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

- Necesidad de utilizar algún mecanismo de sincronización de acceso

POSIX shared Memory

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

int shm_unlink(const char *name);

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

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 (**`mqd_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

```
#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

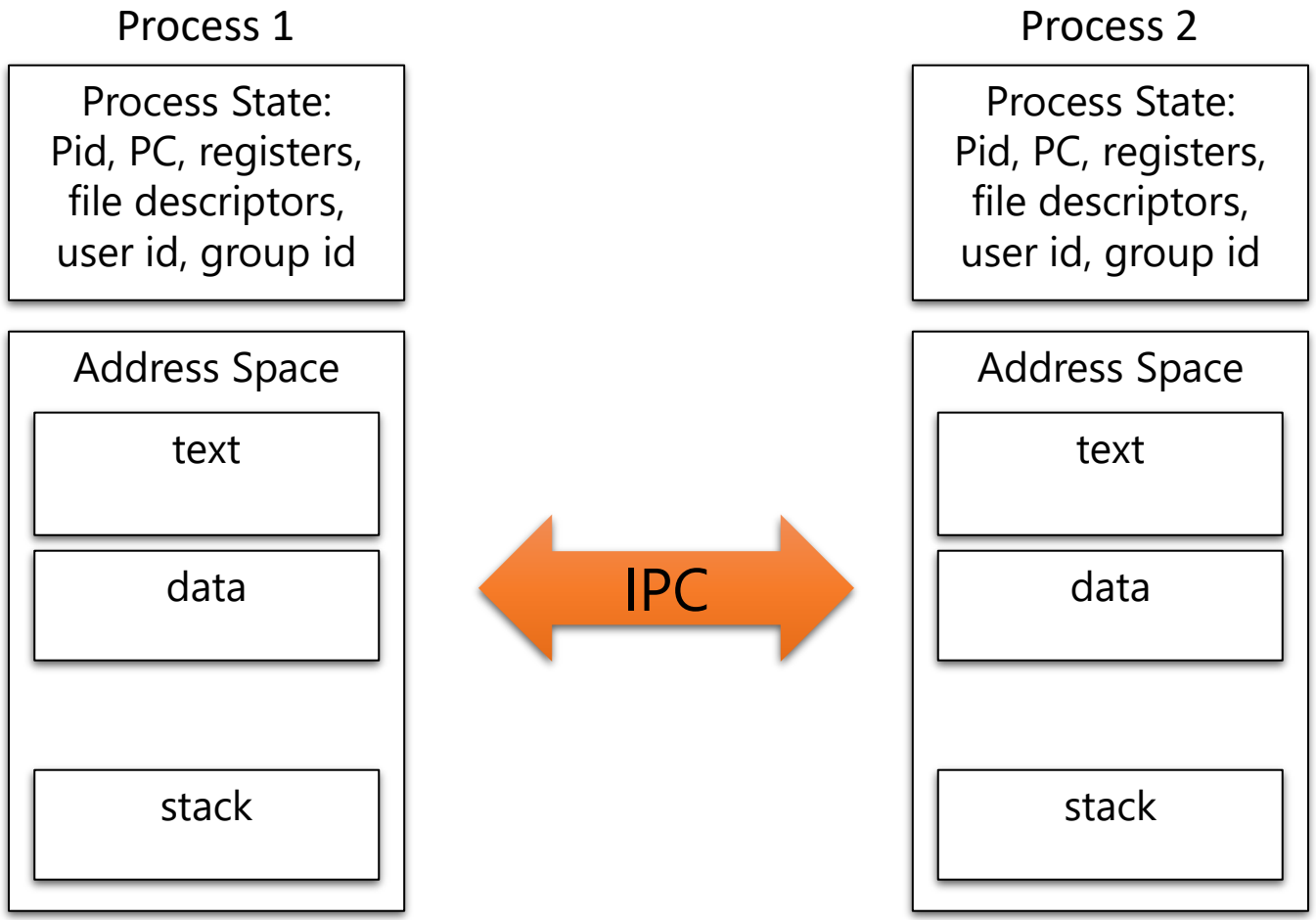
```
#include <mqueue.h>

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

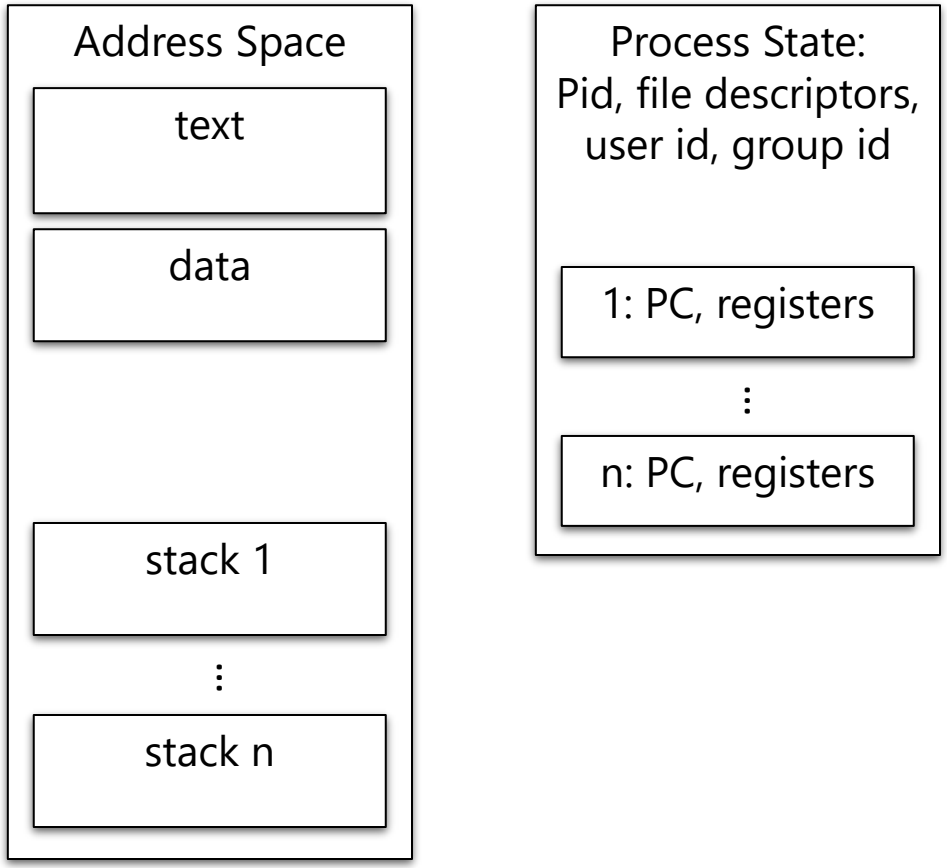
```
#include <mqueue.h>

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);
```

- Linkar 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:

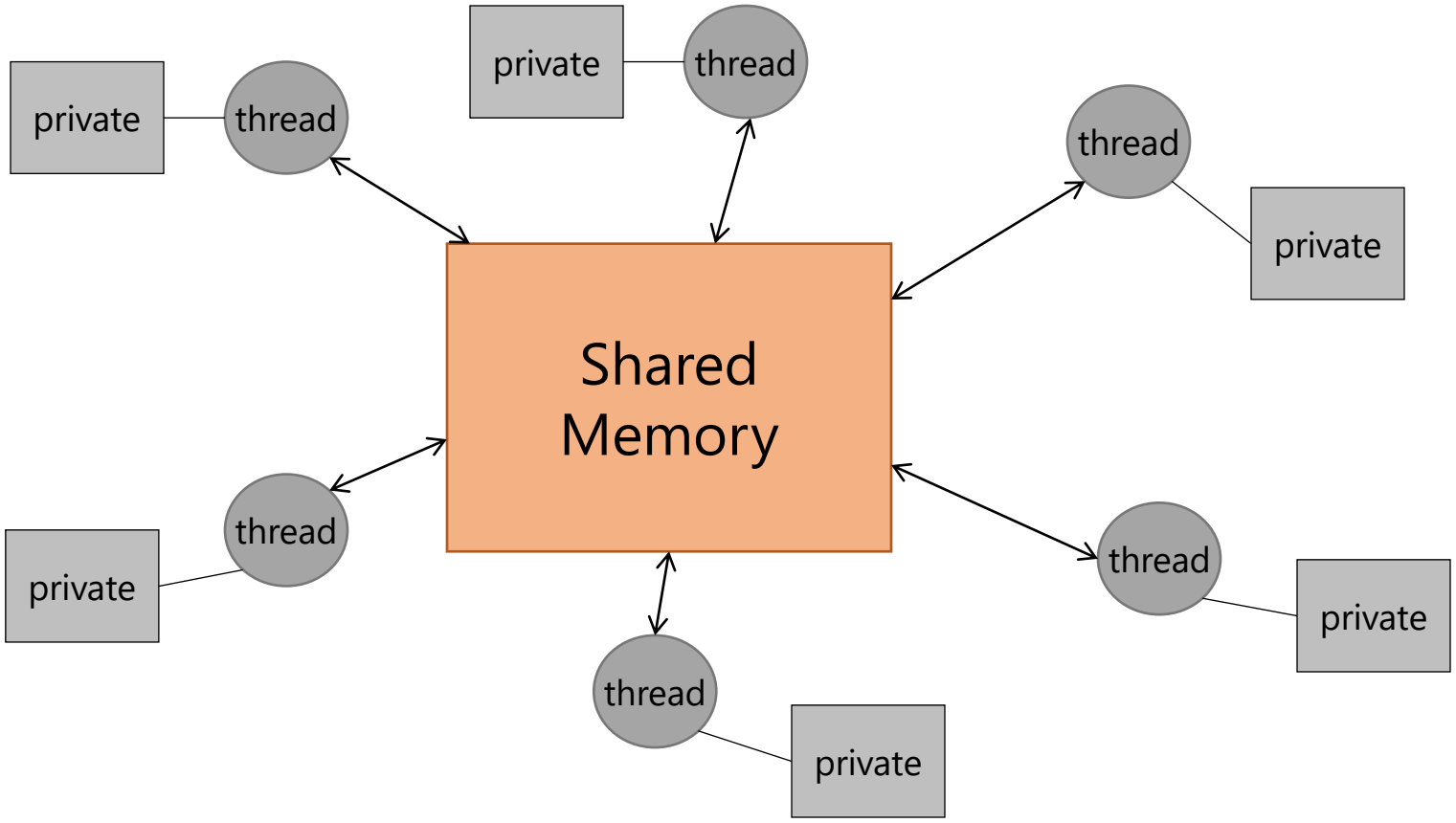
```
pthread_attr_setdetachstate
pthread_attr_setguardsize
pthread_attr_setinheritsched
pthread_attr_setschedparam
pthread_attr_setschedpolicy
pthread_attr_setscope
pthread_attr_setstack
pthread_attr_setstacksize
```

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
```