Sistemas Operativos 72.11

Inter process communication (IPC)

ITBA
Instituto Tecnológico de Buenos Aires

IPC

- Proceso independiente vs cooperativo
- Motivación
  - Compartir información
  - Acelerar computación
  - Modularidad
  - Conveniencia
- 2 modelos fundamentales
  - Memoria compartida
  - Pasaje de mensajes

IPC

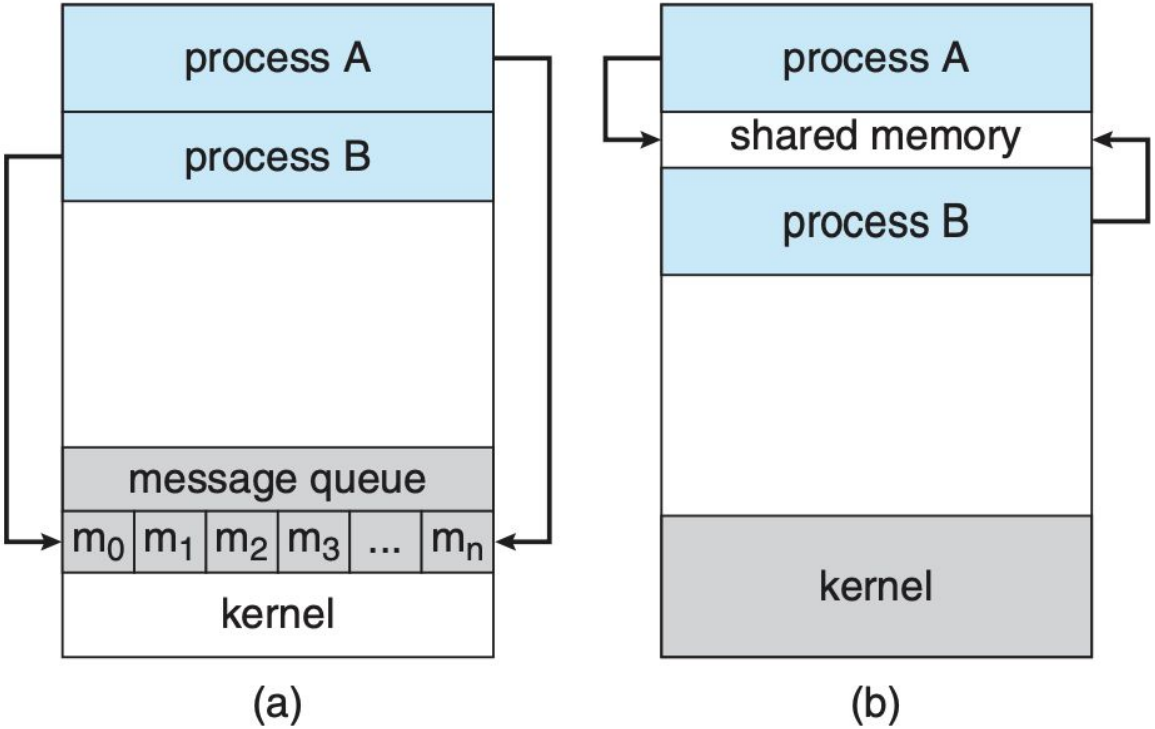


Figure 3.12 Communications models. (a) Message passing. (b) Shared memory.

IPC

Memoria compartida

- Procesos acuerdan compartir memoria física, ¿con quién hay que hablar?
- Generalmente no requiere de syscalls para acceder a la información
- La organización de la información y sincronización es determinada por los procesos
- Problemas de cache coherency al aumentar los núcleos

IPC
Pasaje de mensajes

- Más simple en sistemas distribuidos
- Útil para pequeñas cantidades de información
- Intervención del kernel
- Tamaño de mensaje fijo vs variable
- Operaciones:
  - send(message), receive(message)
- Es necesario un canal de comunicación (memoria, bus, red)

IPC

Pasaje de mensajes: sincronización

send y receive pueden ser bloqueantes o no bloqueantes

- send bloqueante: hasta que llega a la casilla / proceso
- send no bloqueante: resume inmediatamente
- receive bloqueante: hasta que hay mensaje disponible
- receive no bloqueante: recibe mensaje o null

IPC

Pasaje de mensajes: buffering

Los mensajes residen en en un buffer

- Capacidad 0: no buffering -> send debe bloquear
- Capacidad acotada: send debe bloquear si se agota el espacio
- Capacidad no acotada: send no bloquea

IPC

Files (memoria compartida)

- Ejemplos

```
vim main.c [edit]
gcc -Wall main.c


ls > files.txt
grep “*.c” files.txt > sources.txt
sort sources.txt
```

IPC

Pipes (pasaje de mensajes)

- Permite que 2 procesos se comuniquen bajo el esquema productor-consumidor
- Uno de los primeros mecanismos de IPC en UNIX

- Buffering: acotado
- Sincronización: send bloquea al llenarse el pipe, receive bloquea hasta que haya algo
- Identidad: …

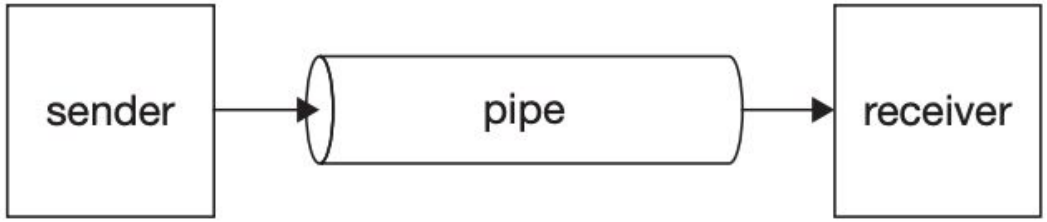
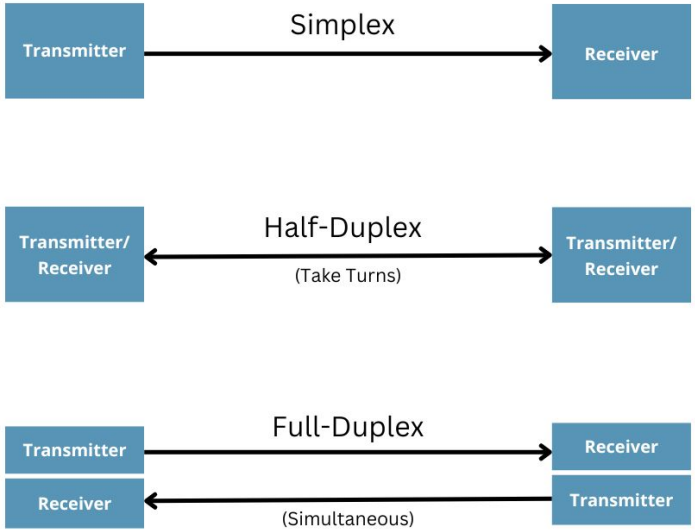


FIGURE 1.11 Communication via a pipe

IPC

Pipes: consideraciones de implementación

- Comunicación unidireccional o bidireccional
- Si es bidireccional, ¿es half duplex o full duplex?
- ¿Debe existir relación padre-hijo?
- ¿Los procesos deben estar en la misma computadora?



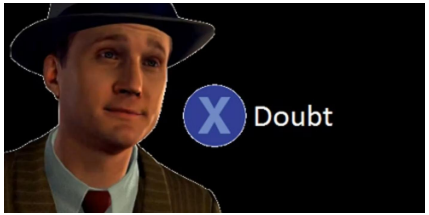
https://www.totalphase.com/blog/2022/10/difference-between-half-duplex-vs-full-duplex/

IPC

Pipes: anónimos / ordinarios

- Permite comunicar 2 procesos emparentados
- UNIX: unidireccionales
- UNIX: se crean con pipe(2) y se heredan al hacer fork(2) y execve(2)
- Identidad: … -> File descriptors
- UNIX: Consultar pipe(7)

An ordinary pipe cannot be accessed from outside the process that created it. Typically, a parent process creates a pipe and uses it to communicate with a child process that it creates via `fork()`. Recall from Section 3.3.1 that a child



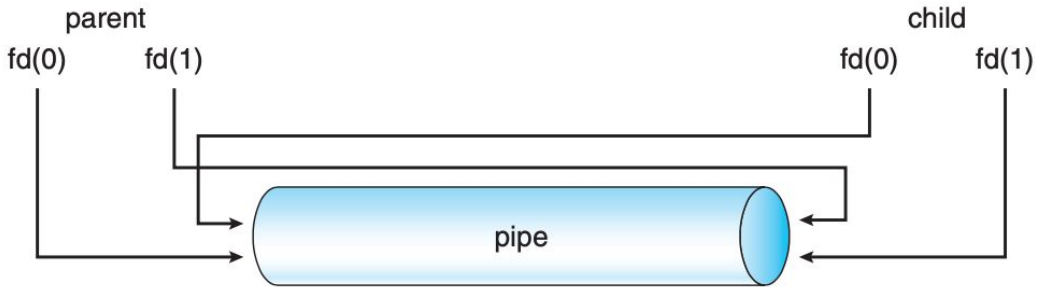


Figure 3.24 File descriptors for an ordinary pipe.

IPC

Pipes: named pipes

- Permite comunicar 2 procesos arbitrarios
- UNIX: FIFOs
- UNIX: unidireccionales
- UNIX: se crean con mkfifo(3) y se abren con open(2)
- Identidad: … -> Path en el file system
- UNIX: Consultar fifo(7)

IPC

Files vs pipes

```
ls > files.txt
grep "*.c" files.txt > sources.txt
sort sources.txt
rm files.txt sources.txt
```


```
ls | grep "*.c" | sort
```

Ventajas de los pipes

- Se eliminan automáticamente (anónimos)
- Cantidad arbitraria de información
- Ejecución paralela
- Sincronización implícita

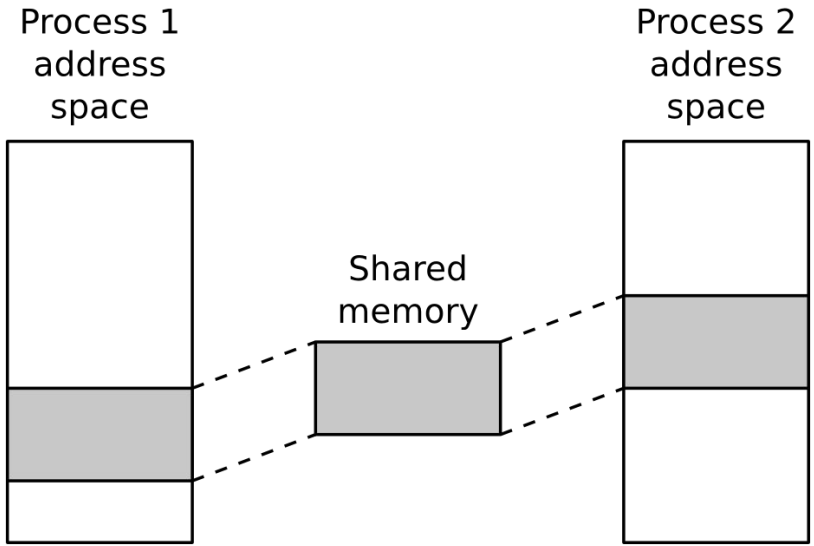
IPC

Shared memory (memoria compartida)

- Permite que procesos arbitrarios compartan memoria
- UNIX: shm_open(3) ftruncate(2) mmap(2)
- Intervención del kernel para crearla?
- Intervención del kernel para usarla (R/W)?
- UNIX: Consultar shm_overview(7)

```
// ...
char buf[1024];
char *buf = (char *) malloc(1024);
char *buf = createSHM(..., 1024, ...);

sprintf(p, "Hola mundo\n");
// ...
```

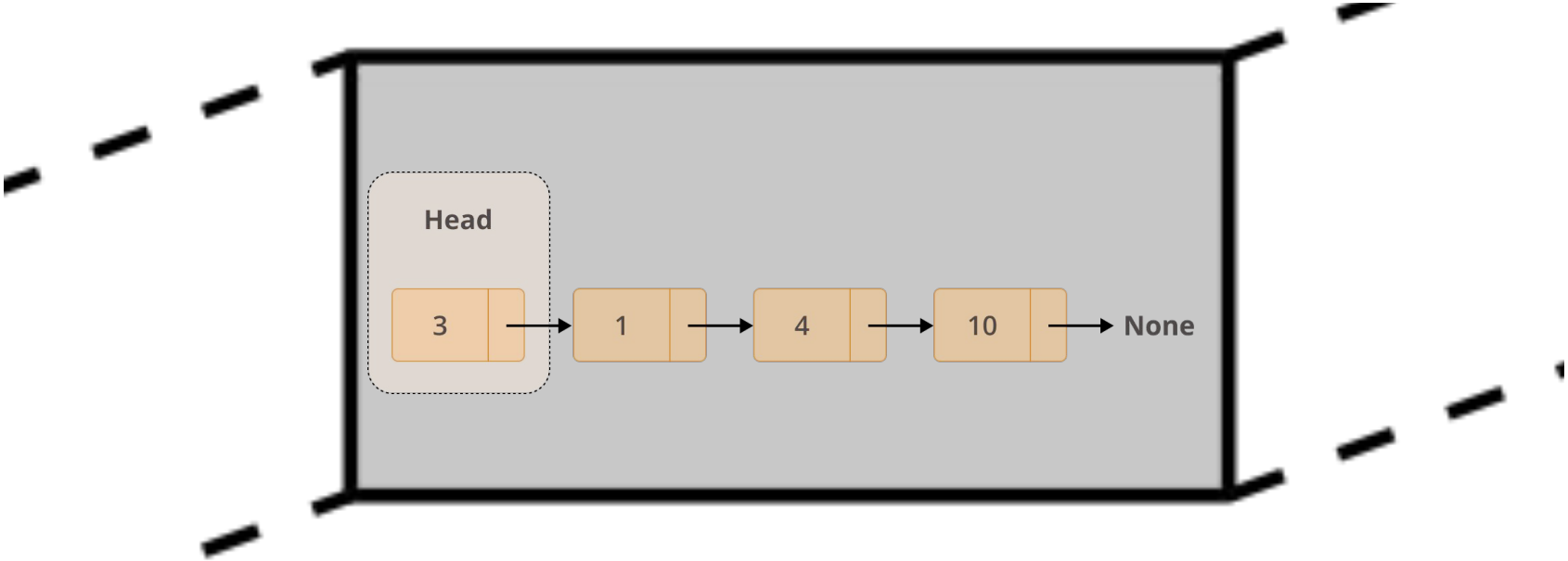


https://www.oreilly.com/library/view/mastering-embedded-linux/9781787283282/7732a483-9260-4511-a063-896f66d37bb4.xhtml

IPC

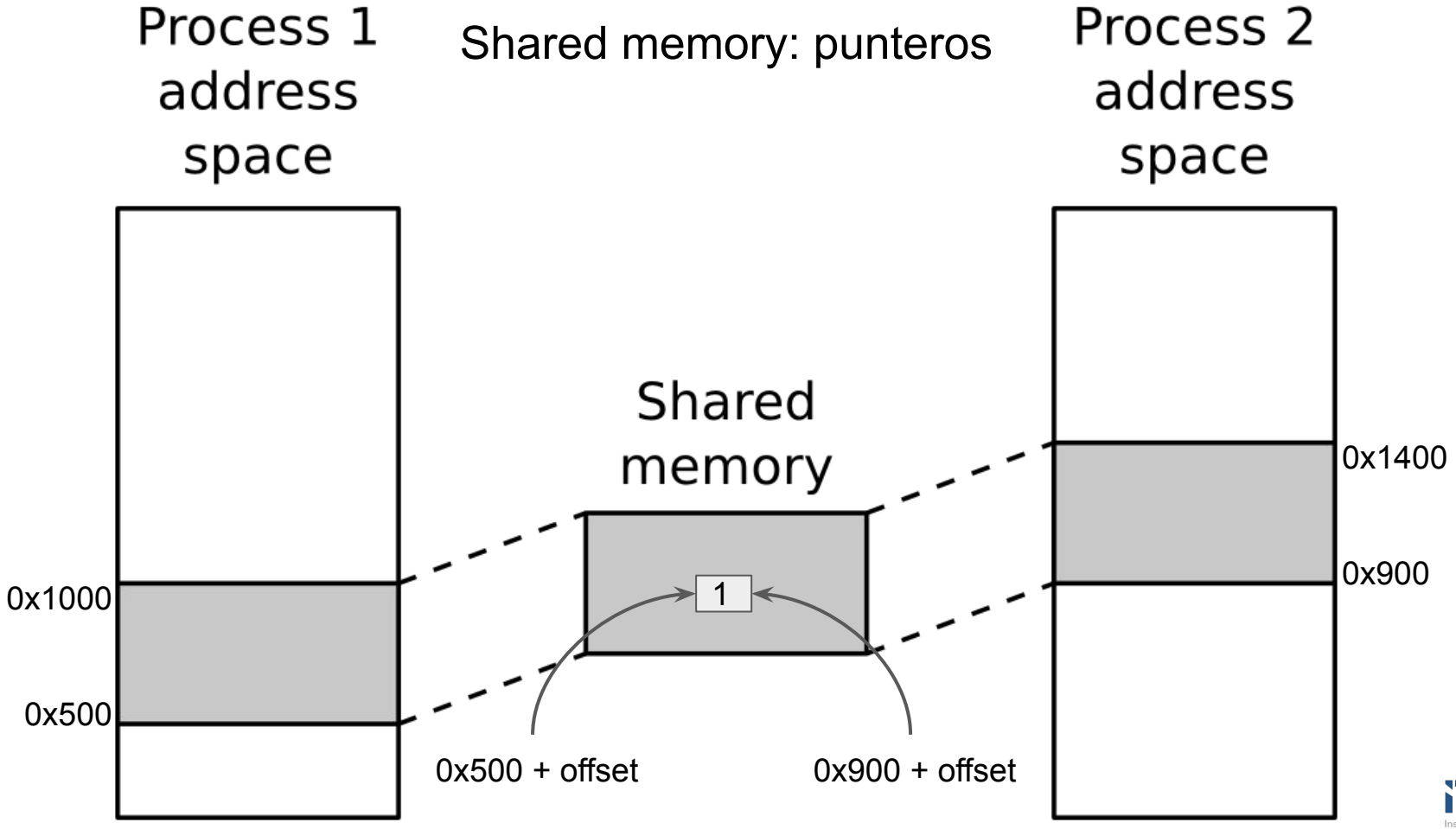
Shared memory: punteros

- Queremos estructurar la información dentro de la shared memory, por ejemplo, una lista:



IPC

Shared memory: punteros



IPC

Sockets (pasaje de mensajes)

- Permite comunicar procesos a través de una red
- Bidireccional - full duplex
- UNIX: socket(2), listen(2), accept(2), bind(2), send(2), recv(2)
- Un socket se identifica con una IP y un puerto
- Generalmente aplicación cliente - servidor
- UNIX: Consultar socket(7)

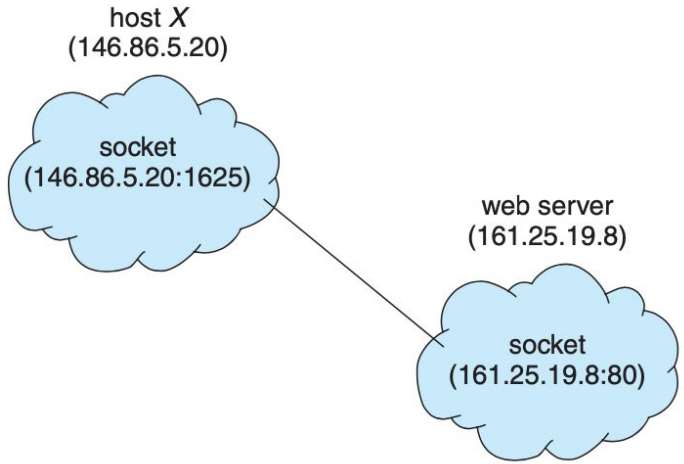
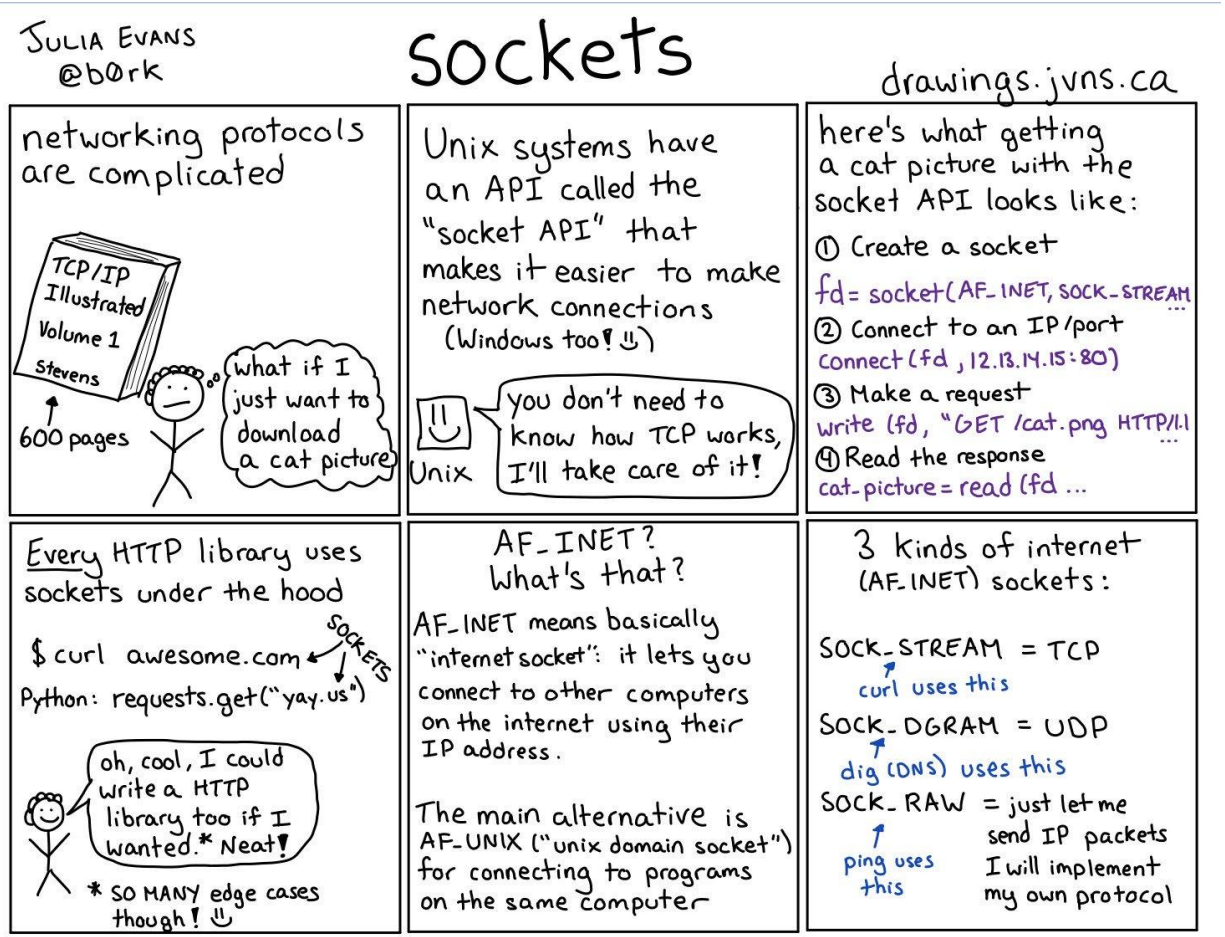


Figure 3.20 Communication using sockets.

IPC



IPC

unix domain sockets

JULIA EVANS
@b0rk

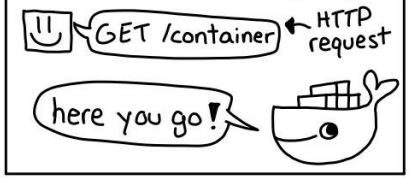
unix domain sockets are files.

```
$ file mysock.sock
socket
```

the file's permissions determine who can send data to the socket

they let 2 programs on the same computer communicate.

Docker uses Unix domain sockets, for example!



There are 2 kinds of unix domain sockets:

stream — like TCP! Lets you send a continuous stream of bytes

datagram — like UDP! Send discrete chunks of data

advantage 1

Lets you use file permissions to restrict access to HTTP/ database services!

```
chmod 600 secret.sock
```

This is why Docker uses a unix domain socket

advantage 2

UDP sockets aren't always reliable (even on the same computer).
unix domain datagram sockets are reliable! And won't reorder!

advantage 3

You can send a file descriptor over a unix domain socket.
Useful when handling untrusted input files!

IPC

FIFOs vs Unix Domain Sockets

- Un socket es bidireccional
- Pueden ser usados por muchos procesos a la vez. Aceptar muchas conecciones en el mismo socket y atender a muchos clientes simultáneamente
- UNIX: consultar unix(7)

- Son necesarios 2 FIFOs para comunicación bidireccional
- Por cada cliente necesitamos 2 FIFOs
- No hay forma de leer o escribir de forma selectiva

IPC

signals

Julia Evans
@b0rk

drawings.jvns.ca

If you've ever used kill you've used signals

the Linux kernel sends your process signals in lots of situations

- your child terminated
- that pipe is closed
- illegal instruction
- the timer you set expired
- Segmentation fault

you can send signals yourself with the kill system call or command

SIGINT Ctrl-C
SIGTERM kill
SIGKILL kill -9
} various levels of "die"

SIGHUP kill -HUP
often interpreted as "reload config", eg by nginx

Every signal has a default action, one of:

- ignore
- Kill process
- kill process AND make core dump file
- stop process
- resume process

Your program can set custom handlers for almost any signal

exceptions:
SIGSTOP & SIGKILL can't be ignored
got SIGKILLED

signals can be hard to handle correctly since they can happen at ANY time

Race condition

- 2 instancias de home banking intentando debitar

```
//...
int saldo_en_cuenta;

void debitar(int *saldo, int debito){
        if (*saldo >= debito)
                *saldo -= debito;
}
//...
```

Race condition

Situación ideal

int saldo_en_cuenta = 100;	
Home banking 1	Home banking 2
debitar(&saldo_en_cuenta, 100);	debitar(&saldo_en_cuenta, 100);
if (*saldo >= debito) // *true* *saldo -= debito;	
	if (*saldo >= debito) // *false* *saldo -= debito;
int saldo_en_cuenta = 0;	

Context switch

Race condition

Situación patológica

int saldo_en_cuenta = 100;	
Home banking 1	Home banking 2
debitar(&saldo_en_cuenta, 100);	debitar(&saldo_en_cuenta, 100);
if (*saldo >= debito) // *true*	
	if (*saldo >= debito) // *true*
	*saldo -= debito;
*saldo -= debito;	
int saldo_en_cuenta = -100;	

Context switch

Race condition

Definición

Una situación en las que dos o más procesos están leyendo o escribiendo datos compartidos y el resultado final depende de quién corre en cada momento

called **race conditions**. Debugging programs containing race conditions is no fun at all. The results of most test runs are fine, but once in a blue moon something weird and unexplained happens. Unfortunately, with increasing parallelism due to increasing numbers of cores, race condition are becoming more common.

Race condition

Solución: exclusión mutua

int saldo_en_cuenta = 100;	
Home banking 1	Home banking 2
debitar(&saldo_en_cuenta, 100);	debitar(&saldo_en_cuenta, 100);
// Pido acceso exclusivo - OK **if** (*saldo >= debito) *// true*	
	// Pido acceso exclusivo - BLOCK …
*saldo -= debito; *// Libero acceso exclusivo*	*// Blocked …* *// Blocked …*
	if (*saldo >= debito) *// true* *saldo -= debito; *// Libero acceso exclusivo*
int saldo_en_cuenta = 0;	

Context switch

Región crítica

Definición

Parte del programa que accede a la memoria compartida

```
//...
int saldo_en_cuenta;

void debitar(int *saldo, int debito){
  if (*saldo >= debito)
    *saldo -= debito;
}
//...
```

Región crítica

Región crítica

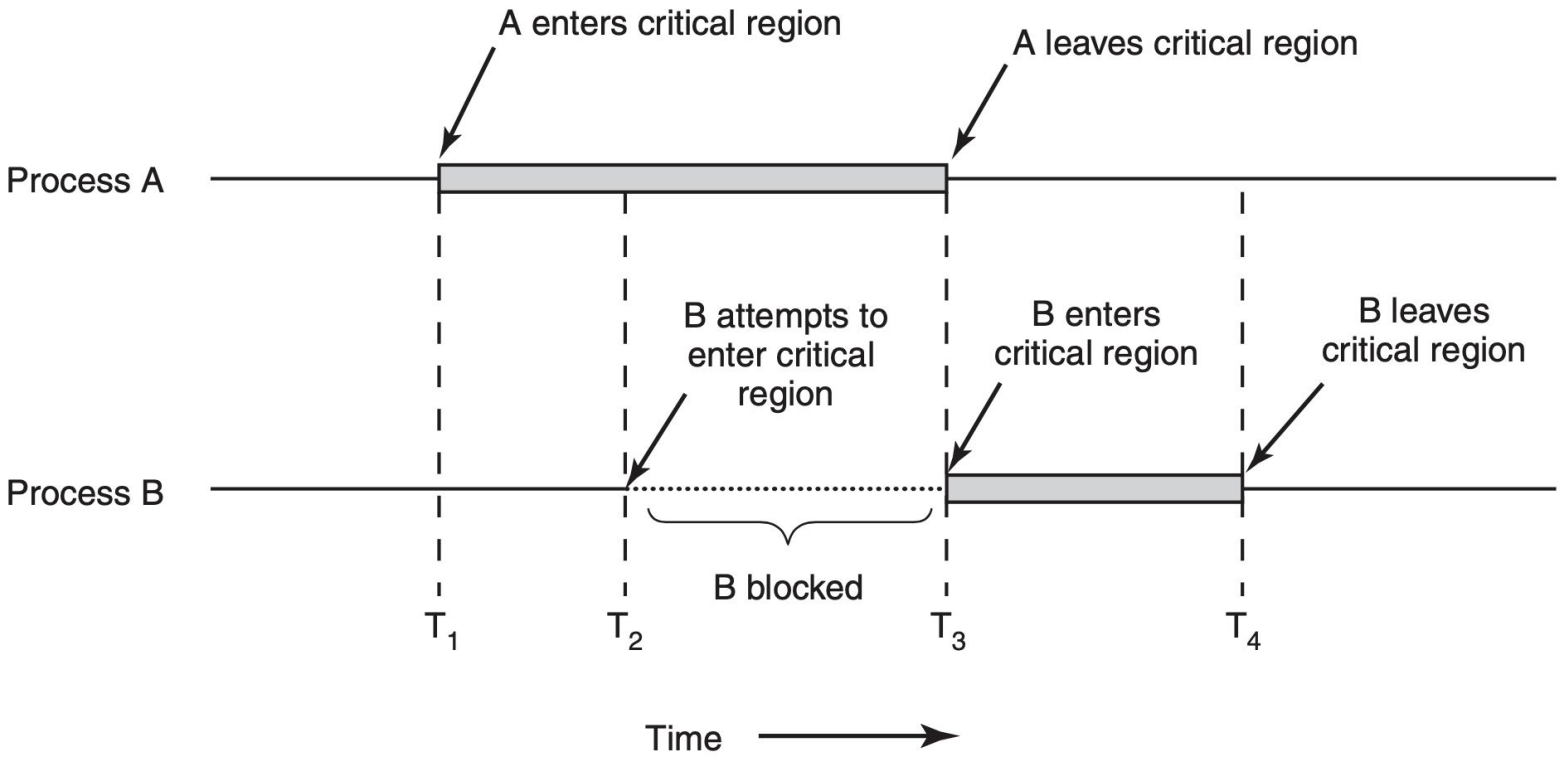


Figure 2-22. Mutual exclusion using critical regions.

Race condition

Otro ejemplo

```
#define N 100                                   /* number of slots in the buffer */
int count = 0;                                  /* number of items in the buffer */

void producer(void)
{
     int item;

     while (TRUE) {                             /* repeat forever */
          item = produce_item( );               /* generate next item */
          if (count == N) sleep( );             /* if buffer is full, go to sleep */
          insert_item(item);                    /* put item in buffer */
          count = count + 1;                    /* increment count of items in buffer */
          if (count == 1) wakeup(consumer);     /* was buffer empty? */
     }
}


void consumer(void)
{
     int item;

     while (TRUE) {                             /* repeat forever */
          if (count == 0) sleep( );             /* if buffer is empty, got to sleep */
          item = remove_item( );                /* take item out of buffer */
          count = count – 1;                    /* decrement count of items in buffer */
          if (count == N – 1) wakeup(producer); /* was buffer full? */
          consume_item(item);                   /* print item */
     }
}
```

Figure 2-27. The producer-consumer problem with a fatal race condition.

- sleep bloquea al proceso hasta que alguien ejecuta wakeup
- wakeup toma como parámetro el id del proceso a despertar

¿syscalls?

- Signal lost
- Wakeup waiting bit

Mecanismos de sincronización

Semáforos

- El Wakeup waiting bit se plantea (Dijkstra - 1965) como un entero representando la cantidad de wakeups disponibles [0-n], el cual llamaremos semáforo
- Se proponen 2 operaciones
  - down (sleep): **atómicamente** decrementa el semáforo o bloquea al caller si ya es 0
  - up (wakeup): **atómicamente** incrementa el semáforo y desbloquea a algún bloqueado si existe

Mecanismos de sincronización

Semáforos

- 2 instancias de home banking intentando debitar… pero esta vez con semáforos

```
//...
int saldo_en_cuenta;
semaphore mutex = 1;

void debitar(int *saldo, int debito){
        down(mutex);
        if (*saldo >= debito)
                *saldo -= debito;
        up(mutex);
}
//...
```

```
#define N 100                                   /* number of slots in the buffer */
typedef int semaphore;                          /* semaphores are a special kind of int */
semaphore mutex = 1;                            /* controls access to critical region */
semaphore empty = N;                            /* counts empty buffer slots */
semaphore full = 0;                             /* counts full buffer slots */

void producer(void)
{
    int item;

    while (TRUE) {                              /* TRUE is the constant 1 */
        item = produce_item( );                 /* generate something to put in buffer */
        down(&empty);                           /* decrement empty count */
        down(&mutex);                           /* enter critical region */
        insert_item(item);                      /* put new item in buffer */
        up(&mutex);                             /* leave critical region */
        up(&full);                              /* increment count of full slots */
    }
}


void consumer(void)
{
    int item;

    while (TRUE) {                              /* infinite loop */
        down(&full);                            /* decrement full count */
        down(&mutex);                           /* enter critical region */
        item = remove_item( );                  /* take item from buffer */
        up(&mutex);                             /* leave critical region */
        up(&empty);                             /* increment count of empty slots */
        consume_item(item);                     /* do something with the item */
    }
}
```

Figure 2-28. The producer-consumer problem using semaphores.

Mecanismos de sincronización

Semáforos

- Semántica del semáforo
- Inicialización del semáforo
- Contar vs acceso exclusivo

Región crítica

```
int main(){
  // code
  // code
  // code
  // code
  // code
  down(mutex);
  // code that access shared data
  // code that access shared data
  up(mutex);
  // code
  // code
  // code
  // code
}
```

```
int main(){
  down(mutex);
  // code
  // code
  // code
  // code
  // code
  // code that access shared data
  // code that access shared data
  // code
  // code
  // code
  // code
  up(mutex);
}
```

Problemas clásicos de IPC

Filósofos comensales

```
#define N 5                                     /* number of philosophers */

void philosopher(int i)                         /* i: philosopher number, from 0 to 4 */
{
    while (TRUE) {
        think( );                               /* philosopher is thinking */
        take_fork(i);                           /* take left fork */
        take_fork((i+1) % N);                   /* take right fork; % is modulo operator */
        eat( );                                 /* yum-yum, spaghetti */
        put_fork(i);                            /* put left fork back on the table */
        put_fork((i+1) % N);                    /* put right fork back on the table */
    }
}
```

Figure 2-46. A nonsolution to the dining philosophers problem.

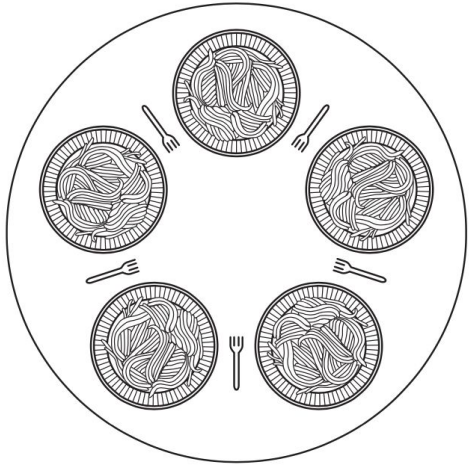


Figure 2-45. Lunch time in the Philosophy Department.

Deadlock

Definición

Un conjunto de procesos está bloqueado (en estado de deadlock) si cada proceso del conjunto está esperando un evento que solo puede causar otro proceso del conjunto.

Deadlock

Ejemplos

```
//...
semaphore mutex = 1;

void foo(){
        down(mutex);
        down(mutex);
}
//...
```

```
//...
semaphore mutexA = 1;
semaphore mutexB = 1;

void P1(){
        down(mutexA);
        down(mutexB);
}
//...
```

```
//...
semaphore mutexA = 1;
semaphore mutexB = 1;

void P2(){
        down(mutexB);
        down(mutexA);
}
//...
```

Problemas clásicos de IPC

Filósofos comensales

```
#define N              5                /* number of philosophers */
#define LEFT           (i+N–1)%N        /* number of i's left neighbor */
#define RIGHT          (i+1)%N          /* number of i's right neighbor */
#define THINKING       0                /* philosopher is thinking */
#define HUNGRY         1                /* philosopher is trying to get forks */
#define EATING         2                /* philosopher is eating */

typedef int semaphore;                  /* semaphores are a special kind of int */
int state[N];                           /* array to keep track of everyone's state */
semaphore mutex = 1;                    /* mutual exclusion for critical regions */
semaphore s[N];                         /* one semaphore per philosopher */


void philosopher(int i)                 /* i: philosopher number, from 0 to N–1 */
{
      while (TRUE) {                    /* repeat forever */
            think( );                   /* philosopher is thinking */
            take_forks(i);              /* acquire two forks or block */
            eat( );                     /* yum-yum, spaghetti */
            put_forks(i);               /* put both forks back on table */
      }
}
```

Problemas clásicos de IPC

Filósofos comensales

```
void take_forks(int i)                  /* i: philosopher number, from 0 to N–1 */
{
        down(&mutex);                   /* enter critical region */
        state[i] = HUNGRY;              /* record fact that philosopher i is hungry */
        test(i);                        /* try to acquire 2 forks */
        up(&mutex);                     /* exit critical region */
        down(&s[i]);                    /* block if forks were not acquired */
}

void put_forks(i)                       /* i: philosopher number, from 0 to N–1 */
{
        down(&mutex);                   /* enter critical region */
        state[i] = THINKING;            /* philosopher has finished eating */
        test(LEFT);                     /* see if left neighbor can now eat */
        test(RIGHT);                    /* see if right neighbor can now eat */
        up(&mutex);                     /* exit critical region */
}

void test(i) /* i: philosopher number, from 0 to N–1 */
{
        if (state[i] == HUNGRY && state[LEFT] != EATING && state[RIGHT] != EATING) {
                state[i] = EATING;
                up(&s[i]);
        }
}
```

Figure 2-47. A solution to the dining philosophers problem.

Problemas clásicos de IPC

Lectores y escritores

```
typedef int semaphore;                  /* use your imagination */
semaphore mutex = 1;                    /* controls access to rc */
semaphore db = 1;                       /* controls access to the database */
int rc = 0;                             /* # of processes reading or wanting to */

void reader(void)
{
    while (TRUE) {                      /* repeat forever */
        down(&mutex);                   /* get exclusive access to rc */
        rc = rc + 1;                    /* one reader more now */
        if (rc == 1) down(&db);         /* if this is the first reader ... */
        up(&mutex);                     /* release exclusive access to rc */
        read_data_base( );              /* access the data */
        down(&mutex);                   /* get exclusive access to rc */
        rc = rc – 1;                    /* one reader fewer now */
        if (rc == 0) up(&db);           /* if this is the last reader ... */
        up(&mutex);                     /* release exclusive access to rc */
        use_data_read( );               /* noncritical region */
    }
}


void writer(void)
{
    while (TRUE) {                      /* repeat forever */
        think_up_data( );               /* noncritical region */
        down(&db);                      /* get exclusive access */
        write_data_base( );             /* update the data */
        up(&db);                        /* release exclusive access */
    }
}
```

Figure 2-48. A solution to the readers and writers problem.

Ejercicio 1

Asumiendo que dispone de semáforos en su sistema:

```
init(sem_t s, int value);
down(sem_t s);
up(sem_t s);
```

Implemente su propia librería de semáforos haciendo uso de la ya disponible:

```
my_init(int *s, int initial_value);
my_down(int *s);
my_up(int *s);
```

- Notar que por simplicidad esta nueva librería puede tomar un int* en lugar de sem_t.
- se puede apuntar a una variable global por simplicidad.
- Puede recurrir a espera activa (busy waiting) para simular el bloqueo usual de un proceso al ejecutar down.

Ejercicio 2

La solución presentada al problema de los filósofos está libre de deadlocks, pero no está libre de (starvation) inanición.

“no estar libre de inanición” significa que existe al menos una traza infinita de ejecución en la que un proceso no progresa. Casualmente los procesos modelan filósofos comiendo, pero eso es solo una coincidencia.

Analice la solución en busca de esta traza. Es recomendable definir a priori qué filósofo vamos a intentar “matar de hambre”, por ejemplo, el filósofo 0. Luego, tomar decisiones de scheduling (qué proceso ejecutará en cada momento) apuntando a la inanición del filósofo elegido. Si bien la traza debe ser infinita, la cantidad de estados no isomorfos que contiene esta traza es finita, lo que implica que la traza será un ciclo infinito a través de estos estados.

Proponga una solución al problema

Ejercicio 3

Un pequeño local comercial del centro tiene un único baño para sus empleados, los cuales son de ambos sexos. El dueño desea que su uso sea exclusivo para un único sexo y para ello instaló una señal con 3 posiciones en la puerta:

LIBRE, MUJERES, HOMBRES

Ahora necesita capacitar al personal para que use el baño de forma adecuada. Para esto solicita instrucciones precisas (código) para ambos sexos, que garanticen que en ningún momento habrá personas de distintos sexos dentro del baño.

Ejercicio 4

La siguiente función incrementa el número almacenado en el archivo *file* 100 veces

```
inc100() {
  for i in {1..100}; do
    n=$(tail -n 1 file);
    n=$(($n + 1));
    sed -i "s/.*/$n/g" file;
  done &
}
```

El siguiente script bash guarda el número 0 en el archivo *file* y luego ejecuta *inc100*

```
echo 0 > file
inc100
```

Ejecute este script (puede copiar y pegar en la terminal) y consulte el valor final del archivo, por ejemplo con cat.

Ahora ejecute el siguiente script que llama a la función inc 4 veces y consulte el valor final del archivo.

```
echo 0 > file
for i in {1..4}; do
  inc100
done
```

¿Qué observa?
Si repite el último ejercicio, ¿el resultado es siempre el mismo?
¿Cómo se conoce a esta situación?

Ejercicio 5

Dos procesos A y B ejecutan 2 instrucciones cada uno

A: a1;a2
B: b1;b2

Sincronice ambos procesos utilizando semáforos de manera tal que a1 ocurra antes que b2 y que b1 ocurra antes que a2.

Ejercicio 6

El problema de los lectores y escritores como está presentado favorece a los lectores, ya que pueden seguir accediendo a la base de datos mientras haya uno presente. Los escritores no tienen otra opción más que esperar a que salgan todos los lectores para poder entrar. Modifique la solución de manera que un escritor solo tenga que esperar a los lectores presentes al momento de su llegada, es decir, los lectores que lleguen luego del escritor deberán esperar a que el escritor finalice.