

Sistemas Operativos

72.11

Inter process communication (IPC)



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

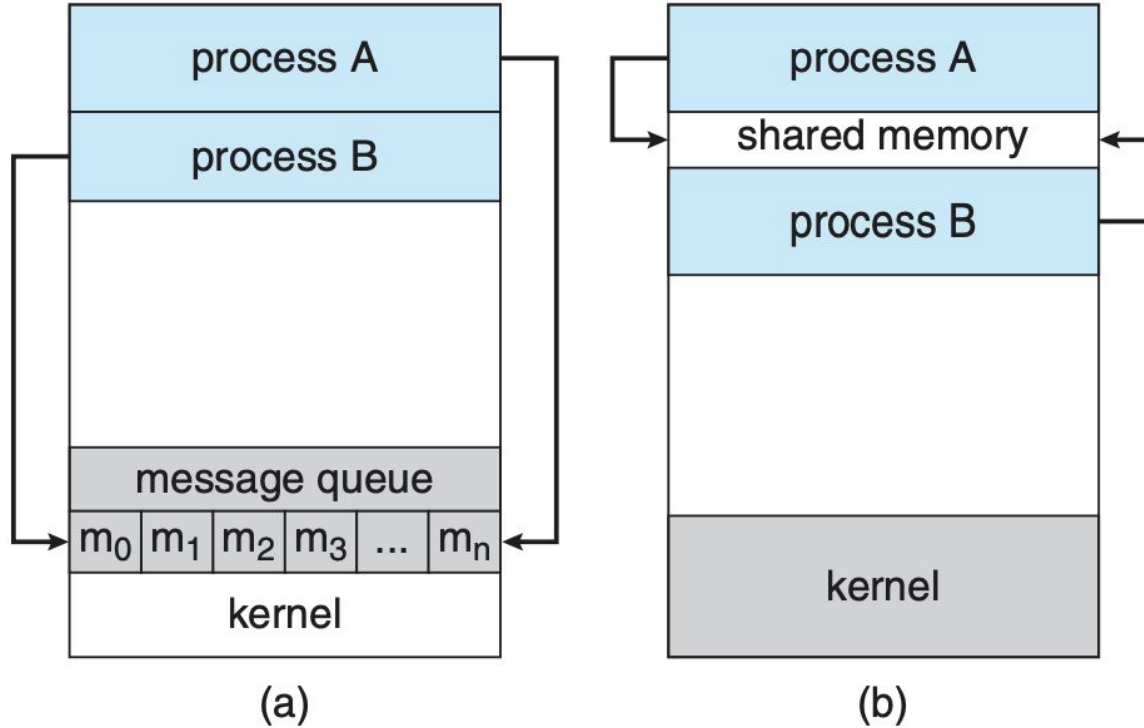


Figure 3.12 Communications models. (a) Message passing. (b) Shared memory.

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 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: ...

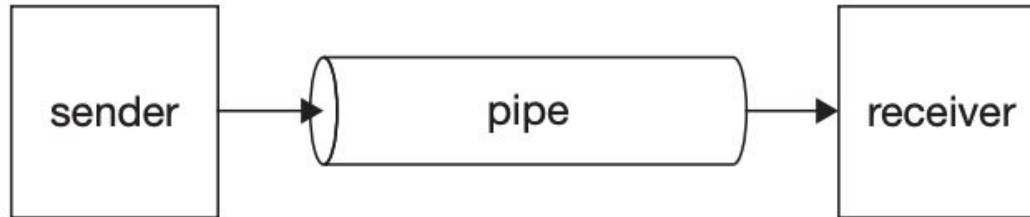
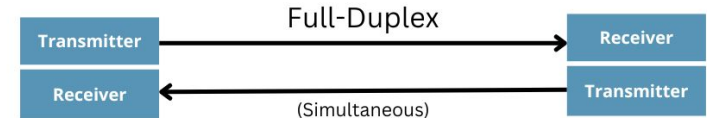
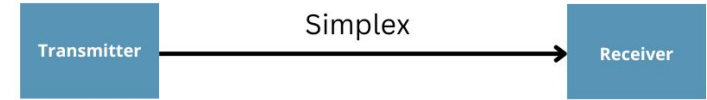


FIGURE 1.11 Communication via a pipe

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?



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

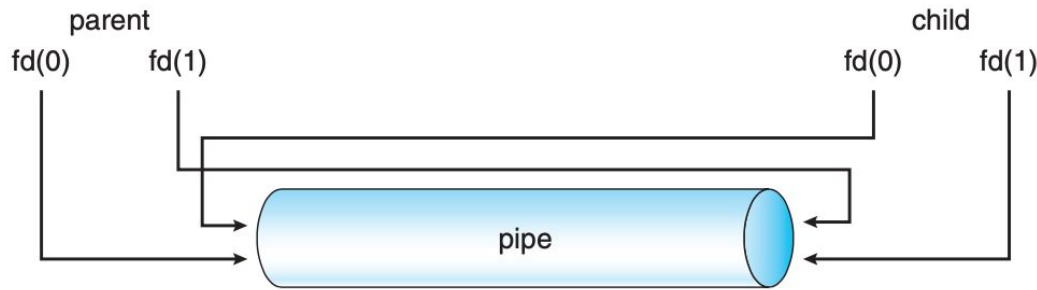
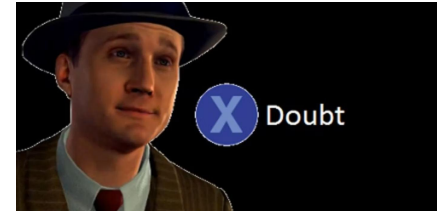


Figure 3.24 File descriptors for an ordinary pipe.

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

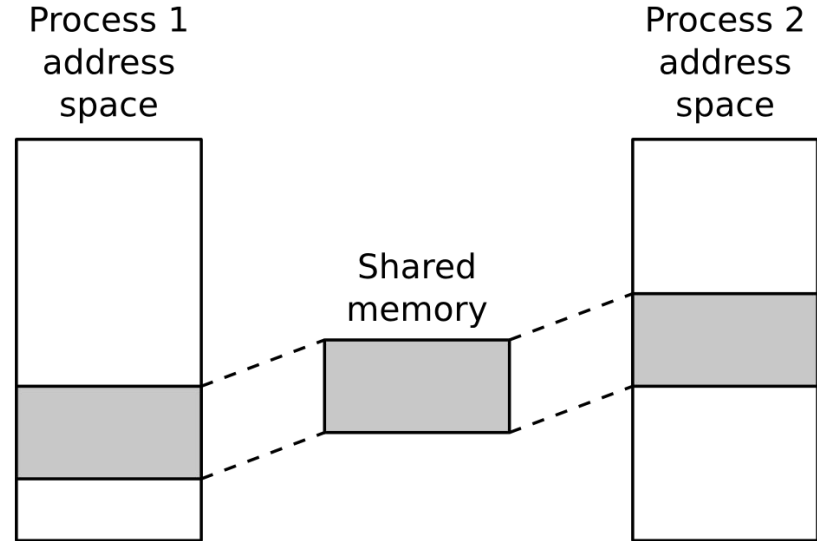
- Se eliminan automáticamente (anónimos)
- Cantidad arbitraria de información
- Ejecución paralela
- Sincronización implícita

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)

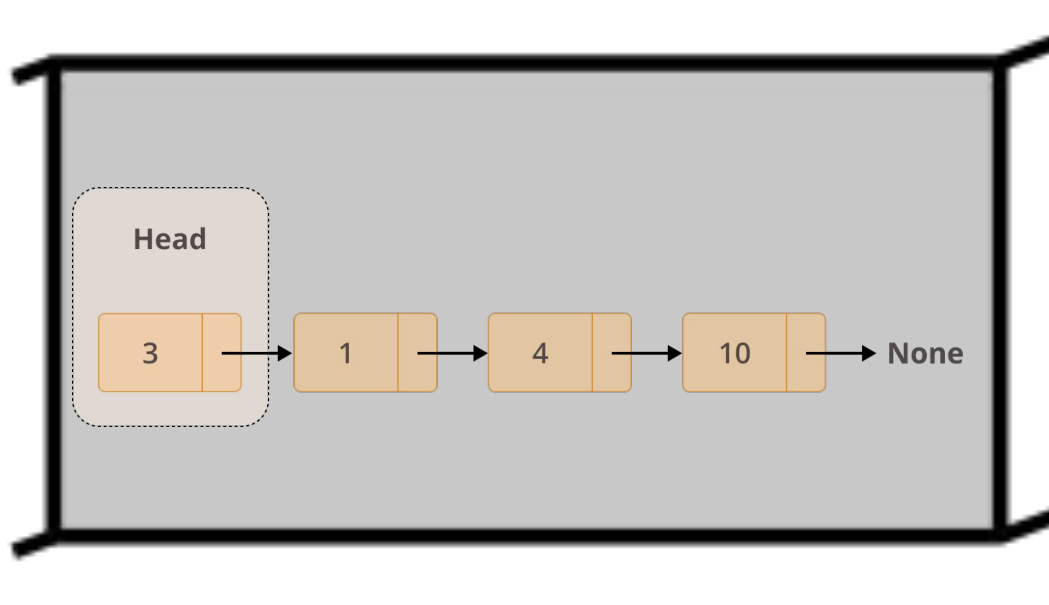
```
// ...  
char buf[1024];  
char *buf = (char *) malloc(1024);  
char *buf = createSHM(..., 1024, ...);  
  
sprintf(p, "Hola mundo\n");  
// ...
```



IPC

Shared memory: punteros

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

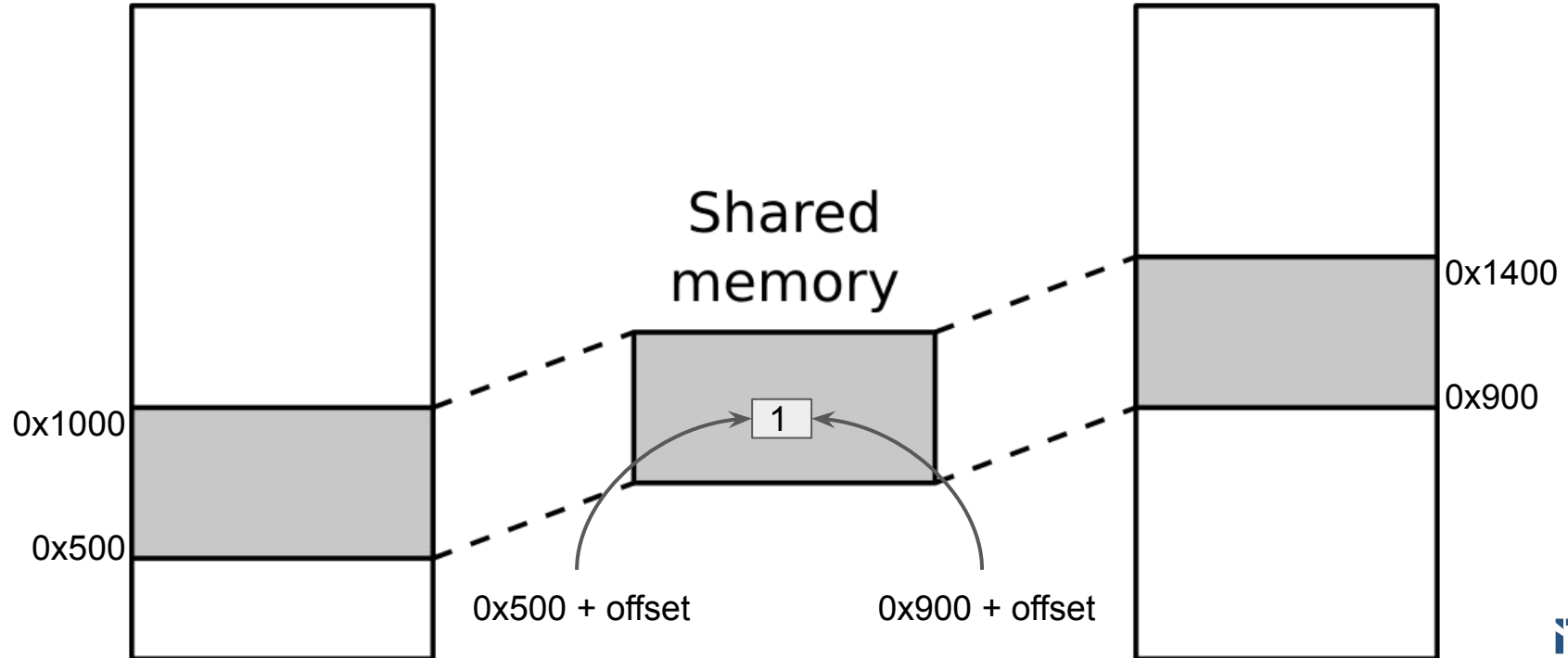


IPC

Shared memory: punteros

Process 1
address
space

Process 2
address
space



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)`

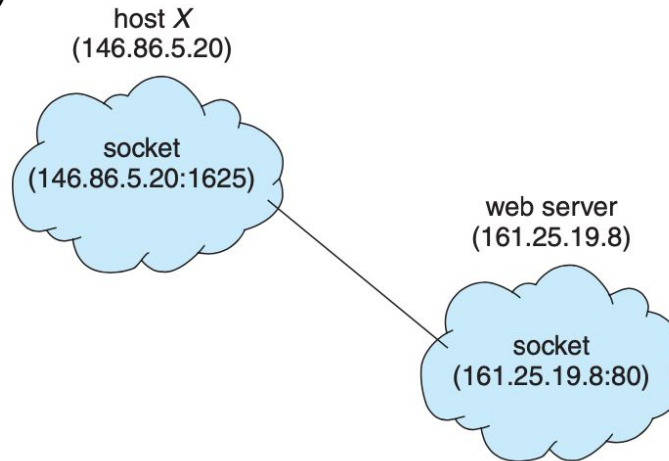


Figure 3.20 Communication using sockets.

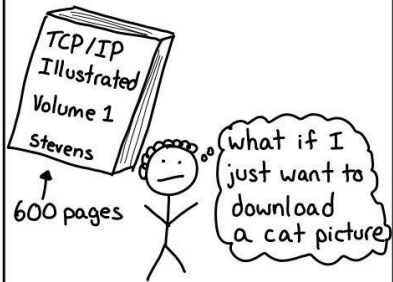
IPC

sockets

SULIA EVANS
@b0rk

drawings.jvns.ca

networking protocols
are complicated



Unix systems have
an API called the
"socket API" that
makes it easier to make
network connections
(Windows too! ☺)



Unix

you don't need to
know how TCP works,
I'll take care of it!

here's what getting
a cat picture with the
socket API looks like:

① Create a socket

```
fd = socket(AF_INET, SOCK_STREAM ...
```

② Connect to an IP/port
`connect(fd, 12.13.14.15:80)`

③ Make a request

```
write(fd, "GET /cat.png HTTP/1.1 ...
```

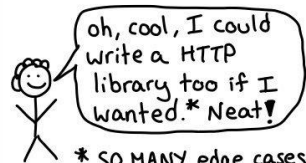
④ Read the response

```
cat_picture = read(fd ...
```

Every HTTP library uses
sockets under the hood

\$ curl awesome.com
Python: requests.get("yay.us")

sockets



* SO MANY edge cases
though! ☺

AF_INET?
What's that?

AF_INET means basically
"internet socket": it lets you
connect to other computers
on the internet using their
IP address.

The main alternative is
AF_UNIX ("unix domain socket")
for connecting to programs
on the same computer

3 kinds of internet
(AF_INET) sockets:

SOCK_STREAM = TCP

↑
curl uses this

SOCK_DGRAM = UDP

↑
dig (DNS) uses this

SOCK_RAW = just let me

↑
ping uses this

send IP packets
I will implement
my own protocol

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

Four small colored squares (yellow, red, purple, blue) arranged horizontally, representing 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 conexiones 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.juns.ca

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



the Linux kernel sends
your process signals in
lots of situations



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

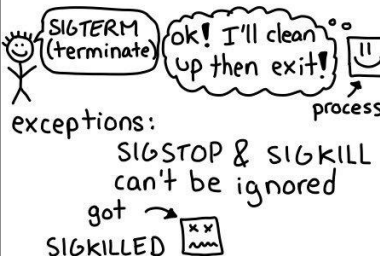
SIGINT ctrl-C } various
SIGTERM kill } levels of
SIGKILL kill -9 } "die"

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

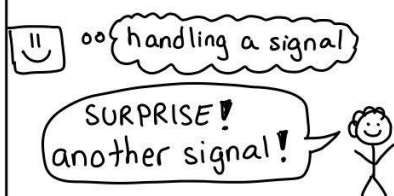
Every signal has a default
action, one of:

- ☺ ignore
- ☒ kill process
- ☒ ☒ kill process AND
make core dump file
- ⏸ stop process
- ⏪ resume process

Your program can set
custom handlers for
almost any signal



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

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

Context switch

Race condition

Situación patológica

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

Context switch

Race condition

Definición

Una situación en la 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 conditions are becoming more common.

Race condition

Solución: exclusión mutua

Context switch

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

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

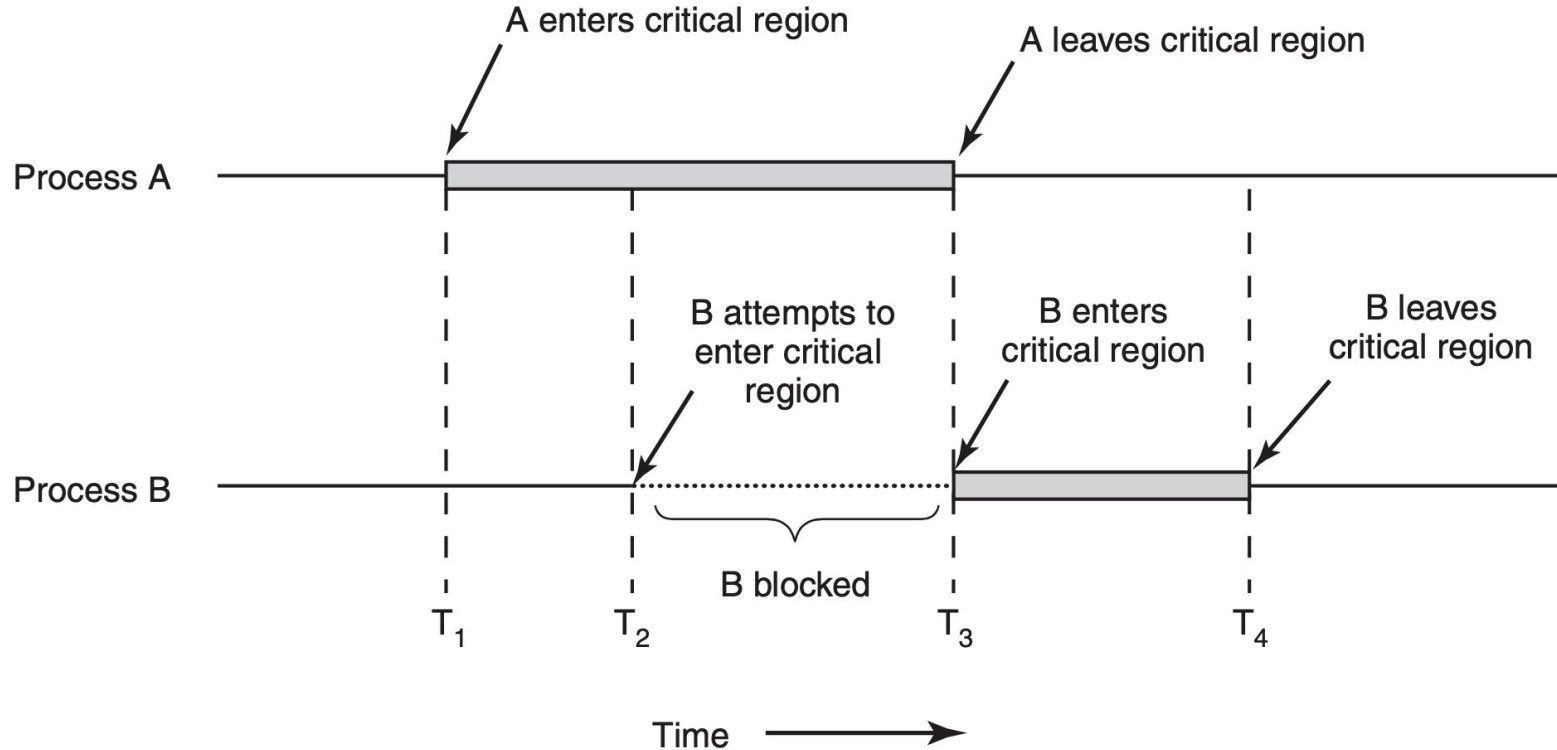


Figure 2-22. Mutual exclusion using critical regions.

```
#define N 100
int count = 0;
```

```
void producer(void)
{
```

```
    int item;
```

```
    while (TRUE) {
        item = produce_item();
        if (count == N) sleep();
        insert_item(item);
        count = count + 1;
        if (count == 1) wakeup(consumer);
    }
```

```
}
```

```
void consumer(void)
```

```
{
```

```
    int item;
```

```
    while (TRUE) {
        if (count == 0) sleep();
        item = remove_item();
        count = count - 1;
        if (count == N - 1) wakeup(producer);
        consume_item(item);
    }
```

```
}
```

```
/* number of slots in the buffer */
/* number of items in the buffer */
```

```
/* repeat forever */
/* generate next item */
/* if buffer is full, go to sleep */
/* put item in buffer */
/* increment count of items in buffer */
/* was buffer empty? */
```

```
/* repeat forever */
/* if buffer is empty, got to sleep */
/* take item out of buffer */
/* decrement count of items in buffer */
/* was buffer full? */
/* print item */
```

Race condition

Otro ejemplo

- 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

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

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
typedef int semaphore;
semaphore mutex = 1;
semaphore empty = N;
semaphore full = 0;

void producer(void)
{
    int item;

    while (TRUE) {
        item = produce_item();
        down(&empty);
        down(&mutex);
        insert_item(item);
        up(&mutex);
        up(&full);
    }
}

void consumer(void)
{
    int item;

    while (TRUE) {
        down(&full);
        down(&mutex);
        item = remove_item();
        up(&mutex);
        up(&empty);
        consume_item(item);
    }
}

```

```

/* number of slots in the buffer */
/* semaphores are a special kind of int */
/* controls access to critical region */
/* counts empty buffer slots */
/* counts full buffer slots */

/* TRUE is the constant 1 */
/* generate something to put in buffer */
/* decrement empty count */
/* enter critical region */
/* put new item in buffer */
/* leave critical region */
/* increment count of full slots */

```

Mecanismos de sincronización

Semáforos

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

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

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

void philosopher(int i)
{
    while (TRUE) {
        think();
        take_fork(i);
        take_fork((i+1) % N);
        eat();
        put_fork(i);
        put_fork((i+1) % N);
    }
}
```

```
/* number of philosophers */

/* i: philosopher number, from 0 to 4 */

/* philosopher is thinking */
/* take left fork */
/* take right fork; % is modulo operator */
/* yum-yum, spaghetti */
/* put left fork back on the table */
/* put right fork back on the table */
```

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

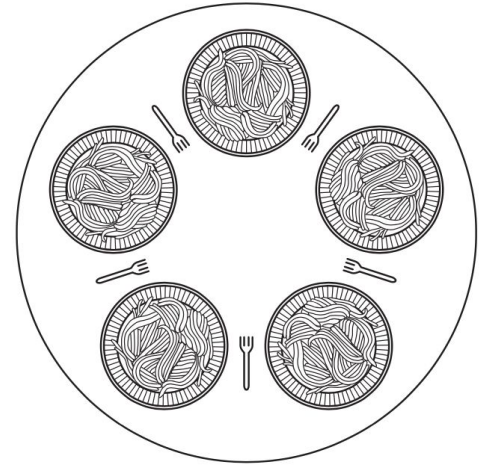


Figure 2-45. Lunch time in the Philosophy Department.

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;
int state[N];
semaphore mutex = 1;
semaphore s[N];

void philosopher(int i)
{
    while (TRUE) {
        think();
        take_forks(i);
        eat();
        put_forks(i);
    }
}
```

Problemas clásicos de IPC

Filósofos comensales

```
void take_forks(int i)
{
    down(&mutex);
    state[i] = HUNGRY;
    test(i);
    up(&mutex);
    down(&s[i]);
}

void put_forks(i)
{
    down(&mutex);
    state[i] = THINKING;
    test(LEFT);
    test(RIGHT);
    up(&mutex);
}

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;
semaphore mutex = 1;
semaphore db = 1;
int rc = 0;

void reader(void)
{
    while (TRUE) {
        down(&mutex);
        rc = rc + 1;
        if (rc == 1) down(&db);
        up(&mutex);
        read_data_base();
        down(&mutex);
        rc = rc - 1;
        if (rc == 0) up(&db);
        up(&mutex);
        use_data_read();
    }
}

void writer(void)
{
    while (TRUE) {
        think_up_data();
        down(&db);
        write_data_base();
        up(&db);
    }
}
```

/ use your imagination */*
/ controls access to rc */*
/ controls access to the database */*
/ # of processes reading or wanting to */*

/ repeat forever */*
/ get exclusive access to rc */*
/ one reader more now */*
/ if this is the first reader ... */*
/ release exclusive access to rc */*
/ access the data */*
/ get exclusive access to rc */*
/ one reader fewer now */*
/ if this is the last reader ... */*
/ release exclusive access to rc */*
/ noncritical region */*

/ repeat forever */*
/ noncritical region */*
/ get exclusive access */*
/ update the data */*
/ 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`.
- `s` 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.