SISTEMAS OPERATIVOS

4

PROCESOS CONCURRENTES

Universidad del Cauca

ISO 9001:2015 SC- CER 450832

IQNet: CO- SC-CER450832

Una Acreditación con Rostro Humano

Hacia una Universidad comprometida con la paz territorial

CONCEPTO

- La **CONCURRENCIA** se refiere al acceso simultáneo (para lectura y/o modificación) de un recurso por parte de dos o más procesos (o hilos).

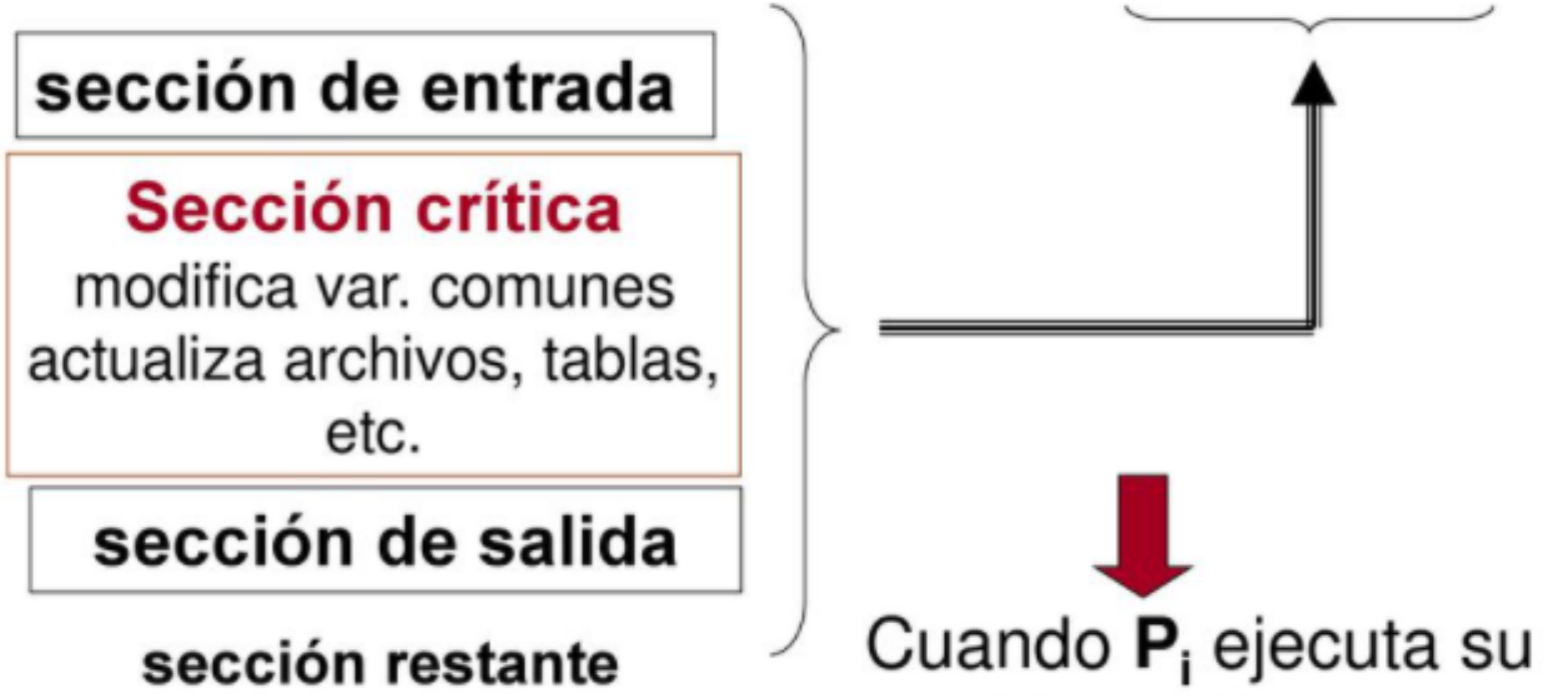
- A pesar que el sistema operativo planifica los procesos para que sólo un proceso tenga la CPU en un momento determinado, al cambiar de proceso se pueden presentar estados inconsistentes en el acceso a los recursos
  - Ejemplo: Incremento de una variable global por parte de dos hilos.

EL PROBLEMA DE LA SECCIÓN CRÍTICA

- Porción de código en la cual se accede a un recurso compartido.
- El SO puede quitar la CPU a un proceso cuando se encuentra en su sección crítica.
- Si el proceso que recibe la CPU entra a su sección crítica en la cual accede y modifica el mismo recurso compartido, se presenta una situación denominada **condición de carrera.**
- El SO debe garantizar que sólo un proceso se encuentre ejecutando su sección crítica para cada recurso compartido, es decir, debe garantizar que se proporciona **exclusión mutua** en el acceso al recurso.

EL PROBLEMA DE LA SECCIÓN CRÍTICA

Situación: se tienen ***n*** procesos $P_1, P_2, \ldots P_n$

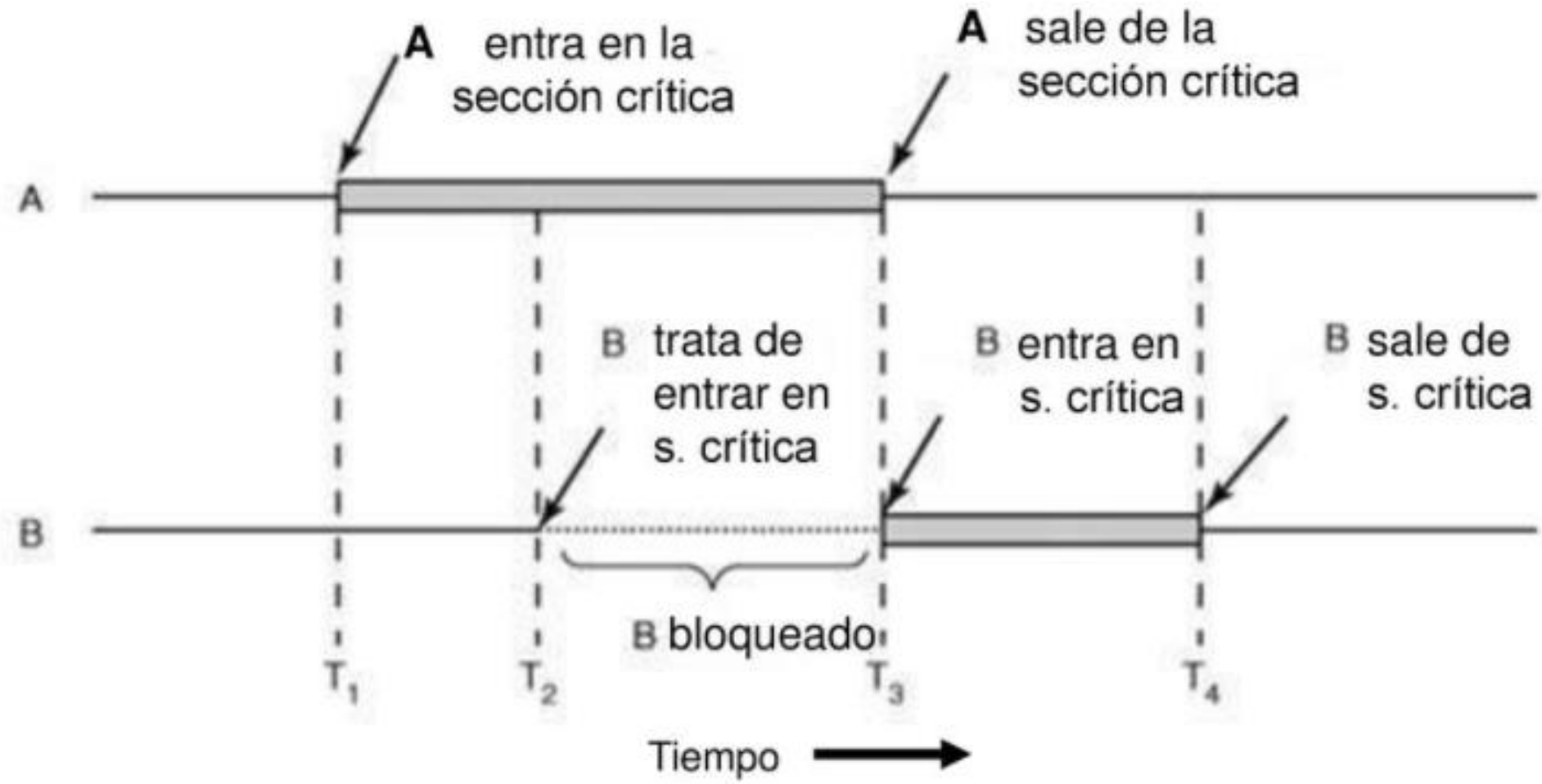


Cuando P_i ejecuta su **sección crítica** ningún otro proceso puede ejecutar la suya.

EXCLUSIÓN MUTUA

- Sólo un proceso puede estar ejecutando su sección crítica.
- No se pueden realizar suposiciones con respecto a la velocidad del procesador o al número de CPU que tiene el sistema.
- Ningún proceso que no esté ejecutando su sección crítica puede bloquear a otro proceso.
- Ningún proceso debe esperar indefinidamente para entrar a su sección crítica.

EL PROBLEMA DE LA SECCIÓN CRÍTICA



Exclusión mutua mediante el uso de regiones críticas

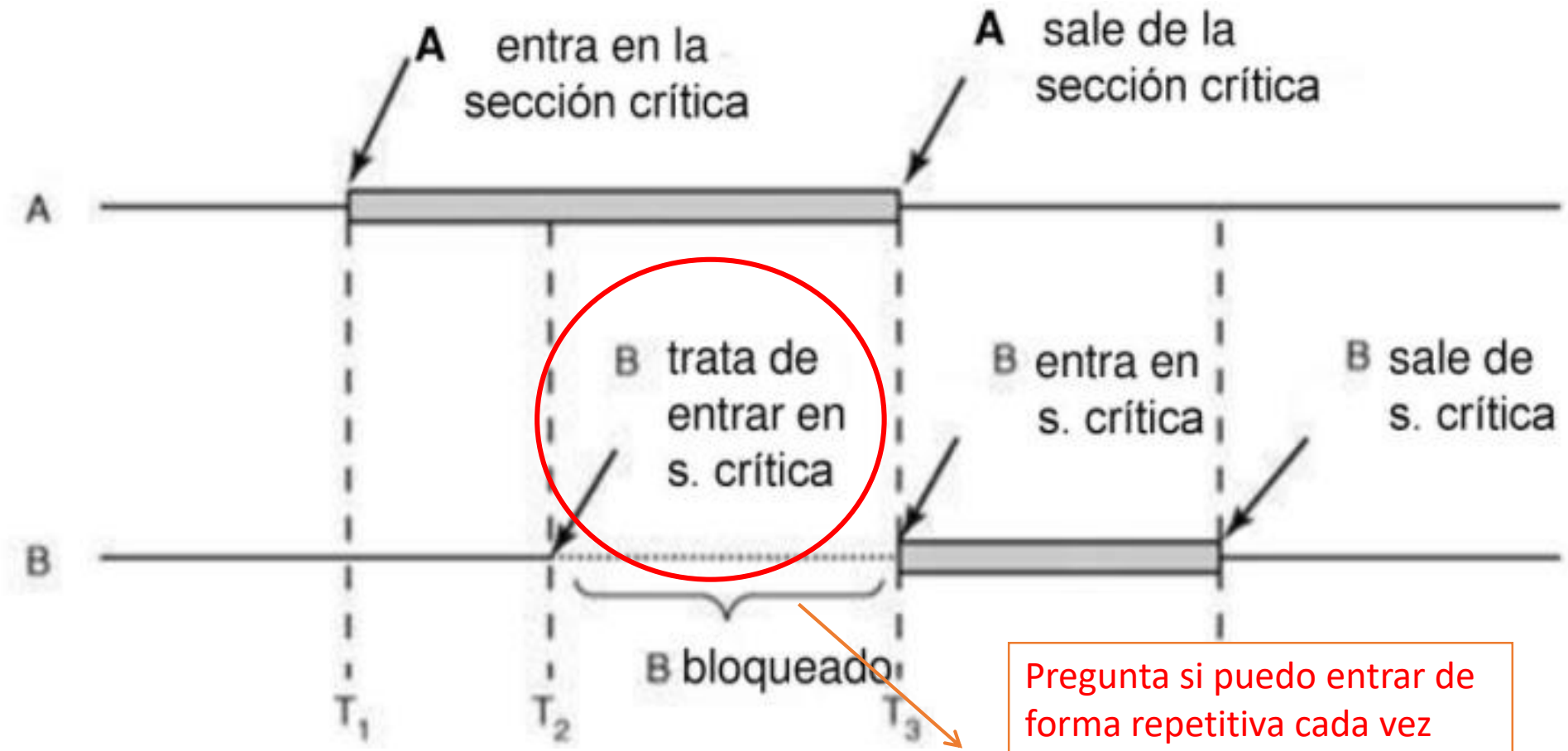
CONCEPTO DE SINCRONIZACIÓN

- La sincronización de procesos se refiere a la estrategia que se usa para que dos o más procesos puedan **ejecutar sus secciones críticas sin que se produzcan inconsistencias** en el estado de los recursos compartidos.

- La sincronización se puede implementar mediante:

  - **Esperas activas**: Los procesos sincronizan su ejecución usando estructuras repetitivas.
  - **Llamadas al sistema**: Los procesos realizan una llamada al sistema cuando desean entrar a sus secciones críticas, y una llamada al sistema cuando terminan de ejecutar sus secciones críticas.

CONCEPTO DE SINCRONIZACIÓN



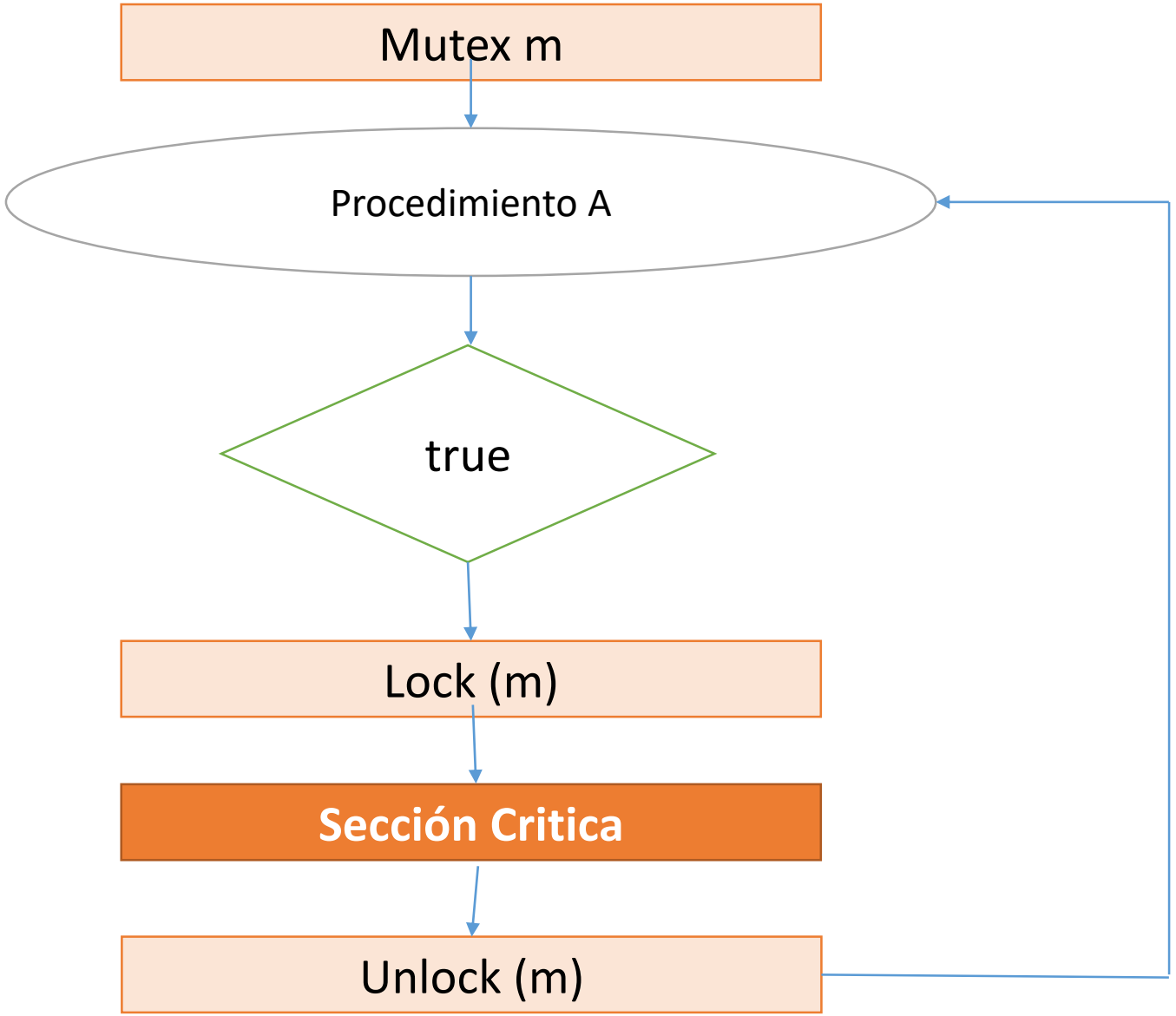
ESPERA ACTIVA

Pregunta si puedo entrar de forma repetitiva cada vez que se le da la CPU hasta que se le permita entrar cuando se termina la sección critica de otro proceso


CONCURRENCIA
PRESENTA
SECCION CRITICA
SE SOLUCIONA CON
EXCLUSION MUTUA
POR MEDIO DE
SINCRONIZACION

MUTEX

- Primitiva de sincronización, que usada de forma consistente, permite garantizar la exclusión mutua (de ahí su nombre).
- Se basa en el llamado a la biblioteca
- Se pueden realizar dos operaciones sobre un mutex:
  - **Bloquear:** Si el mutex se encuentra desbloqueado, se continúa la ejecución. En caso contrario, el proceso (hilo) se bloquea hasta que el mutex sea desbloqueado por otro proceso.
  - **Desbloquear:** Desbloquea uno de los procesos que se encuetran bloqueados por el mutex.

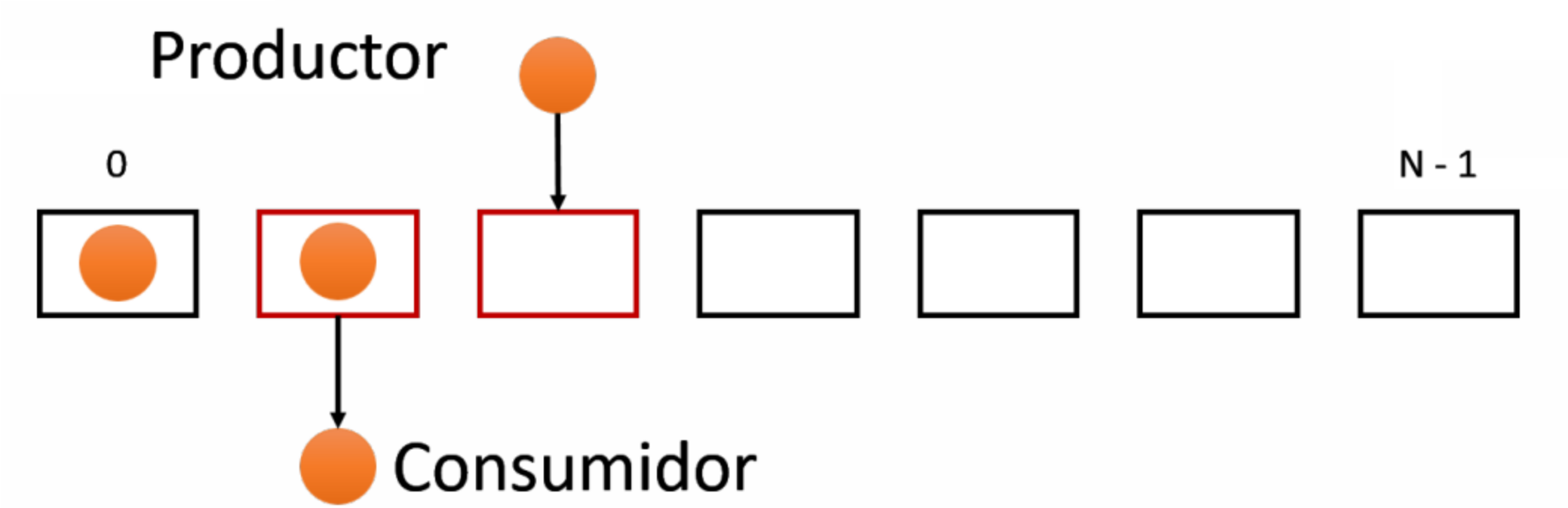

Mutex m
Procedimiento A
true
Lock (m)
Sección Critica
Unlock (m)

VARIABLES DE CONDICIÓN

- Permiten realizar esperas condicionadas, es decir, bloquear un hilo ó proceso hasta que se cumpla determinada condición.
- Se usan en conjunto con un mutex, para evitar la concurrencia al momento de evaluar la condición.
- Antes de evaluar la condición se debe adquirir un mutex.
  - Si la condición no se cumple, se libera atómicamente el mutex adquirido y el proceso se bloquea.
  - Si la condición se cumple, se adquiere el mutex y el proceso continúa con su ejecución.

PROBLEMAS DE SINCRONIZACIÓN: PRODUCTOR - CONSUMIDOR

- **Se tiene un buffer de tamaño limitado N.**
- **El productor inserta nuevos elementos en el buffer.**
- **El consumidor quita elementos que han sido insertados por el productor.**



PROBLEMAS DE SINCRONIZACIÓN: PRODUCTOR - CONSUMIDOR

- **El consumidor sólo puede quitar elementos si el productor ha insertado alguno. Cuando el buffer se encuentre vacío, se debe bloquear.**

- **El productor no puede insertar más de N ítems. Si el buffer está lleno, se debe bloquear.**

- **Se debe garantizar exclusión mutua en el acceso al buffer. Es decir, el consumidor no puede quitar un elemento si el productor está insertando. Por su parte, el productor no puede insertar un elemento si el consumidor se encuentra quitando.**

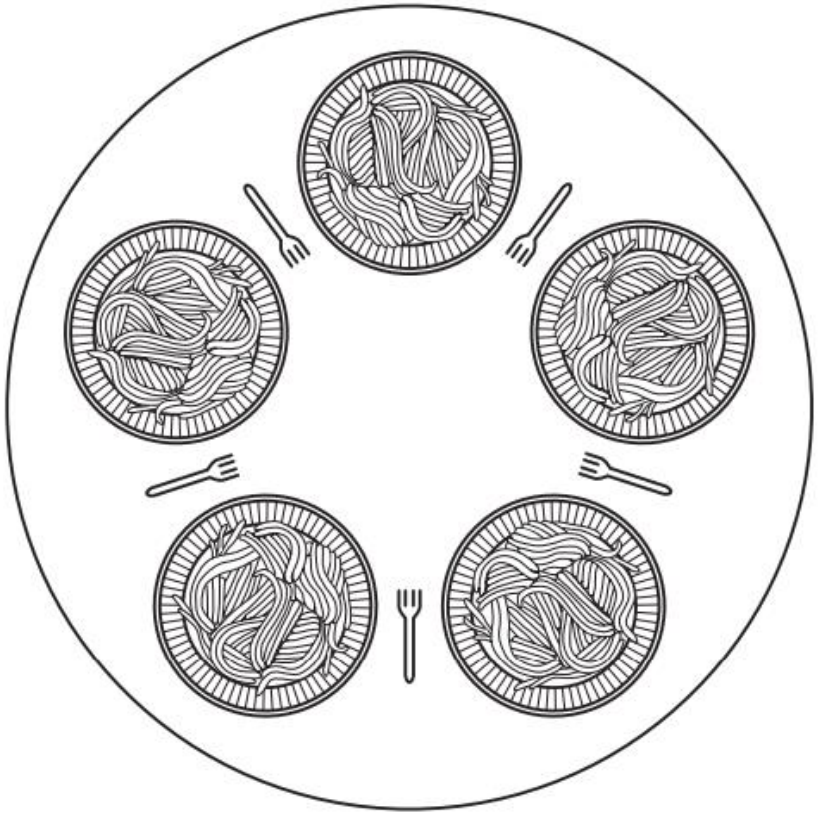
PROBLEMAS DE SINCRONIZACIÓN: PRODUCTOR - CONSUMIDOR

```
#define N 100                                   /* número de ranuras en el búfer */
int cuenta = 0;                                 /* número de elementos en el búfer */

void productor(void)
{
        int elemento;

        while (TRUE) {                          /* se repite en forma indefinida */
                elemento = producir_elemento();         /* genera el siguiente elemento */
                if (cuenta == N) sleep();               /* si el búfer está lleno, pasa a inactivo */
                insertar_elemento(elemento);            /* coloca elemento en búfer */
                cuenta = cuenta + 1;                    /* incrementa cuenta de elementos en búfer */
                if (cuenta == 1) wakeup(consumidor);    /* ¿estaba vacío el búfer? */
        }
}

void consumidor(void)
{
        int elemento;

        while (TRUE) {                          /* se repite en forma indefinida */
                if (cuenta == 0) sleep();               /* si búfer está vacío, pasa a inactivo */
                elemento = quitar_elemento();           /* saca el elemento del búfer */
                cuenta = cuenta – 1;                    /* disminuye cuenta de elementos en búfer */
                if (cuenta==N–1) wakeup(productor);     /* ¿estaba lleno el búfer? */
                consumir_elemento(elemento);            /* imprime el elemento */
        }
}
```

Figura 2-27. El problema del productor-consumidor con una condición de carrera fatal.

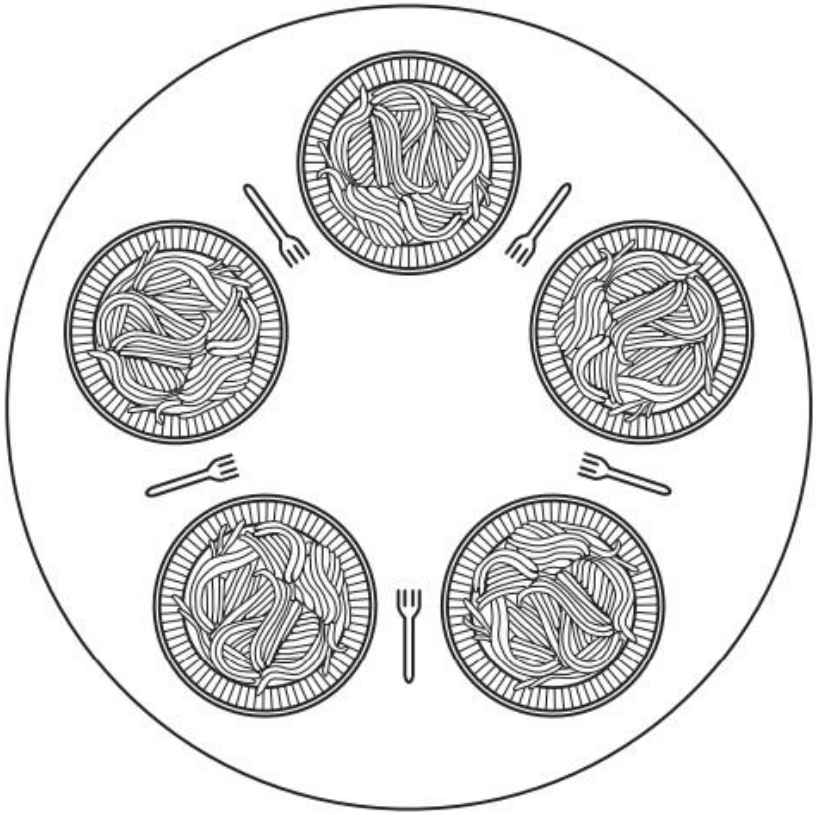
PROBLEMAS DE SINCRONIZACIÓN: FILÓSOFOS COMELONES

El problema de los filósofos comelones es útil para modelar procesos que compiten por el acceso exclusivo a un número limitado de recursos, como los dispositivos de E/S.



- **Un número N > 2 de filósofos se sientan alrededor de una mesa redonda.**
- **Los filósofos pasan su tiempo pensando y comiendo.**
- **Para comer, cada filósofo necesita tomar los tenedores que se encuentran a la izquierda y la derecha de su plato.**

PROBLEMAS DE SINCRONIZACIÓN: FILÓSOFOS COMELONES



- **Se debe garantizar que cada filósofo adquiera los dos tenedores. ¡Si toma sólo uno, y se bloquea tomando el otro, puede causar que otro filósofo (y todos los demás) se bloqueen!**
- **Se puede implementar una estrategia de dar paso:**
  - **Si un filósofo quiere comer, y sus vecinos no están usando los tenedores, los puede tomar. En caso contrario, debe bloquearse.**
  - **Al terminar de comer, el filósofo debe verificar si sus vecinos desean comer, y si están bloqueados esperando por los tenedores que acaba de soltar.**

PROBLEMAS DE SINCRONIZACIÓN: LECTORES Y ESCRITORES

- **Los procesos lectores pueden acceder al recurso compartido (e.g. una base de datos) de forma simultánea, pero sólo si no hay procesos escritores modificando el recurso.**

- **Los procesos escritores deben tener acceso exclusivo al recurso compartido para modificarlo.**

- **Los procesos escritores sólo pueden acceder al recurso si no existen procesos lectores.**

UNA ALTERNATIVA DE SOLUCIÓN: SEMÁFOROS

Primitivas de sincronización que se componen de:

- **Una variable entera cuyo valor es controlado por el sistema operativo**
- **Dos llamadas al sistema que modifican el valor del semáforo:**
  - **Down: Si el valor del semáforo es cero, bloquea al proceso que realizó la llamada. En caso contrario, decrementa el valor del semáforo y no bloquea al proceso.**
  - **Up: Incrementa incondicionalmente el valor de la variable. Si existe algún proceso bloqueado por el semáforo, lo pasa a la cola de listos, donde eventualmente recibirá la CPU.**

SEMÁFOROS

```
#define N 100                                   /* número de ranuras en el búfer */
typedef int semaforo;                           /* los semáforos son un tipo especial de int */
semaforo mutex = 1;                             /* controla el acceso a la región crítica */
semaforo vacias = N;                            /* cuenta las ranuras vacías del búfer */
semaforo llenas = 0;                            /* cuenta las ranuras llenas del búfer */

void productor(void)
{
        int elemento;

        while(TRUE){                            /* TRUE es la constante 1 */
                elemento = producir_elemento();  /* genera algo para colocar en el búfer */
                down(&vacias);                  /* disminuye la cuenta de ranuras vacías */
                down(&mutex);                   /* entra a la región crítica */
                insertar_elemento(elemento);    /* coloca el nuevo elemento en el búfer */
                up(&mutex);                     /* sale de la región crítica */
                up(&llenas);                    /* incrementa la cuenta de ranuras llenas */
        }
}
void consumidor(void)
{
        int elemento;

        while(TRUE){                            /* ciclo infinito */
                down(&llenas);                  /* disminuye la cuenta de ranuras llenas */
                down(&mutex);                   /* entra a la región crítica */
                elemento = quitar_elemento();   /* saca el elemento del búfer */
                up(&mutex);                     /* sale de la región crítica */
                up(&vacias);                    /* incrementa la cuenta de ranuras vacías */
                consumir_elemento(elemento);    /* hace algo con el elemento */
        }
}
```

Libro de Tanenbaum
(3ª Edición. Página 130)

El semáforo **mutex**: para asegurar que **el productor y el consumidor** no tengan acceso al búfer al mismo tiempo.

Los semáforos que se inicializan a 1 y son utilizados por dos o más procesos para asegurar que sólo uno de ellos pueda entrar a su región crítica en un momento dado
se llaman **semáforos binarios**.

Si cada proceso realiza una operación **down justo antes** de entrar a su región crítica y una operación **up justo después** de salir de ella,
se garantiza la exclusión mutua.

Tengamos en cuenta...

1. El hardware mete el contador del programa a la pila, etc.
2. El hardware carga el nuevo contador de programa del vector de interrupciones.
3. Procedimiento en lenguaje ensamblador guarda los registros.
4. Procedimiento en lenguaje ensamblador establece la nueva pila.
5. El servicio de interrupciones de C se ejecuta (por lo general lee y guarda la entrada en el búfer).
6. El planificador decide qué proceso se va a ejecutar a continuación.
7. Procedimiento en C regresa al código de ensamblador.
8. Procedimiento en lenguaje ensamblador inicia el nuevo proceso actual.

Figura 2-5. Esqueleto de lo que hace el nivel más bajo del sistema operativo cuando ocurre una interrupción.

SEMÁFOROS

Tengamos en cuenta…

1. El hardware mete el contador del programa a la pila, etc.
2. El hardware carga el nuevo contador de programa del vector de interrupciones.
3. Pr
4. Pr
5. El
6. El
7. Pr
8. Pr

- **En un sistema que utiliza semáforos, la forma natural de ocultar las interrupciones es asociar un semáforo (que al principio es 0) con cada dispositivo de E/S.**
- **Justo después de iniciar un dispositivo de E/S, el proceso administrativo realiza una operación down en el semáforo asociado, con lo cual se bloquea de inmediato.**
- **Cuando entra la interrupción, el manejador de interrupciones realiza una operación up en el semáforo asociado, lo cual hace que el proceso relevante esté listo para ejecutarse de nuevo.**

Tengamos en cuenta…

En este modelo, el paso 5 de la figura 2-5 consiste en realizar una operación up en el semáforo del dispositivo, de manera que en el paso 6 el planificador pueda ejecutar el administrador de dispositivos.

1.
2.
3.
4.
5. El servicio de interrupciones de C se ejecuta (por lo general lee y guarda la entrada en el búfer).
6. El planificador decide qué proceso se va a ejecutar a continuación.
7. Procedimiento en C regresa al código de ensamblador.
8. Procedimiento en lenguaje ensamblador inicia el nuevo proceso actual.

Figura 2-5. Esqueleto de lo que hace el nivel más bajo del sistema operativo cuando ocurre una interrupción.

¿Y los filósofos comelones?

Libro de Tanenbaum
(3ª Edición. Página 166)

```
#define N                5          /* número de filósofos */
#define IZQUIERDO        (i+N-1)%N  /* número del vecino izquierdo de i */
#define DERECHO          (i+1)%N    /* número del vecino derecho de i */
#define PENSANDO         0          /* el filósofo está pensando */
#define HAMBRIENTO       1          /* el filósofo trata de obtener los tenedores */
#define COMIENDO         2          /* el filósofo está comiendo */
typedef int semaforo;               /* los semáforos son un tipo especial de int */
int estado[N];                      /* arreglo que lleva registro del estado de todos*/
semaforo mutex = 1;                 /* exclusión mutua para las regiones críticas */
semaforo s[N];                      /* un semáforo por filósofo */

void filosofo(int i)                /* i: número de filósofo, de 0 a N−1 */
{
    while(TRUE){                    /* se repite en forma indefinida */
        pensar();                   /* el filósofo está pensando */
        tomar_tenedores(i);         /* adquiere dos tenedores o se bloquea */
        comer();                    /* come espagueti */
        poner_tenedores(i);         /* pone de vuelta ambos tenedores en la mesa */
    }
}
```

¿Y los filósofos comelones?

Libro de Tanenbaum
(3ª Edición. Página 166)

```
void tomar_tenedores(int i)          /* i: número de filósofo, de 0 a N−1 */
{
    down(&mutex);                    /* entra a la región crítica */
    estado[i] = HAMBRIENTO;          /* registra el hecho de que el filósofo i está hambriento */
    probar(i);                       /* trata de adquirir 2 tenedores */
    up(&mutex);                      /* sale de la región crítica */
    down(&s[i]);                     /* se bloquea si no se adquirieron los tenedores */
}

void poner_tenedores(i)              /* i: número de filósofo, de 0 a N–1 */
{
    down(&mutex);                    /* entra a la región crítica */
    estado[i] = PENSANDO;            /* el filósofo terminó de comer */
    probar(IZQUIERDO);               /* verifica si el vecino izquierdo puede comer ahora */
    probar(DERECHO);                 /* verifica si el vecino derecho puede comer ahora */
    up(&mutex);                      /* sale de la región crítica */
}

void probar(i)              /* i: número de filósofo, de 0 a N−1 */
{
    if (estado[i] == HAMBRIENTO && estado[IZQUIERDO] != COMIENDO && estado[DERECHO] != COMIENDO) {
        estado[i] = COMIENDO;
        up(&s[i]);
    }
}
```

¿Y los filósofos comelones?

```
#define N                5          /* número de filósofos */
#define IZQUIERDO        (i+N-1)%N  /* número del vecino izquierdo de i */
#define DERECHO          (i+1)%N    /* número del vecino derecho de i */
#define PENSANDO         0          /* el filósofo está pensando */
#define HAMBRIENTO       1          /* el filósofo trata de obtener los tenedores */
#define COMIENDO         2          /* el filósofo está comiendo */
typedef int semaforo;               /* los semáforos son un tipo especial de int */
int estado[N];
semaforo mutex = 1;
semaforo s[N];

void filosofo(int i)
{
      while(TRUE){
            pensar();
            tomar_tenedores(i);
            comer();
            poner_tenedores(i);
      }
}
```

- **Utiliza un arreglo llamado estado para llevar el registro de si un filósofo está comiendo, pensando o hambriento (tratando de adquirir tenedores).**
- **Un filósofo sólo se puede mover al estado de comer si ningún vecino está comiendo. Los i vecinos del filósofo se definen mediante las macros IZQUIERDO y DERECHO. En otras palabras, si i es 2, IZQUIERDO es 1 y DERECHO es 3.**
- **El programa utiliza un arreglo de semáforos, uno por cada filósofo, de manera que los filósofos hambrientos puedan bloquearse si los tenedores que necesitan están ocupados.**

¿Y los lectores y escritores?

Libro de Tanenbaum
(3ª Edición. Página 168)

```
typedef int semaforo;                   /* use su imaginación */
semaforo mutex=1;                       /* controla el acceso a 'cl' */
semaforo bd=1;                          /* controla el acceso a la base de datos */
int cl=0;                               /* # de procesos que leen o desean hacerlo */

void lector(void)
{
        while(TRUE){                    /* se repite de manera indefinida */
            down(&mutex);               /* obtiene acceso exclusivo a 'cl' */
            cl = cl + 1;                /* ahora hay un lector más */
            if (cl == 1) down(&bd);     /* si este es el primer lector ... */
            up(&mutex);                 /* libera el acceso exclusivo a 'cl' */
            leer_base_de_datos();       /* accede a los datos */
            down(&mutex);               /* obtiene acceso exclusivo a 'cl' */
            cl = cl – 1;                /* ahora hay un lector menos */
            if (cl == 0) up(&bd);       /* si este es el último lector ... */
            up(&mutex);                 /* libera el acceso exclusivo a 'cl' */
            usar_lectura_datos();       /* región no crítica */
        }
}

void escritor(void)
{
            while(TRUE){                /* se repite de manera indefinida */
            pensar_datos();             /* región no crítica */
            down(&bd);                  /* obtiene acceso exclusivo */
            escribir_base_de_datos();   /* actualiza los datos */
            up(&bd);                    /* libera el acceso exclusivo */
        }
}
```

Sin embargo, así como está en determinadas circunstancias el "escritor" podría no entrar en mucho tiempo.

Se sugiere un cambio:

- **Cuando llega un lector y hay un escritor en espera, el lector se suspende detrás del escritor, en vez de ser admitido de inmediato.**
- **De esta forma, un escritor tiene que esperar a que terminen los lectores que estaban activos cuando llegó, pero no tiene que esperar a los lectores que llegaron después de él.**
- **Desventaja: logra una menor concurrencia y por ende, menor rendimiento.**

COMUNICACIÓN ENTRE PROCESOS

- **La comunicación de proceso se refiere al intercambio de información entre procesos.**

- **En el proceso de exclusión mutua y sincronización, también se intercambia información entre procesos, por lo que muchos académicos también los clasifican como comunicación de proceso de bajo nivel.**

- **Comunicación Local: Transferencia de datos entre procesos que se ejecutan en el mismo entorno (sistema operativo).**

- **Comunicación Remota: Transferencia entre procesos que se ejecutan en diferentes entornos, potencialmente sobre diferente hardware (o diferentes sistemas operativos).**

Señales

- Permiten el paso de "alertas" entre procesos.
- Sólo permiten enviar un conjunto predefinido de alertas.
- El comportamiento por defecto de un proceso, al recibir la señal, es terminar.
- Por esta razón el comando para enviar señales en POSIX se denomina kill.
- El proceso que desea recibir la señal debe prepararse, de forma que al recibirla no termine su ejecución.
- Funciones POSIX: kill, sigaction

- **Comando kill**
- **kill –l**

```
mariaisabel@debian:~$ kill -l
 1) SIGHUP       2) SIGINT       3) SIGQUIT      4) SIGILL       5) SIGTRAP
 6) SIGABRT      7) SIGBUS       8) SIGFPE       9) SIGKILL     10) SIGUSR1
11) SIGSEGV     12) SIGUSR2     13) SIGPIPE     14) SIGALRM     15) SIGTERM
16) SIGSTKFLT   17) SIGCHLD     18) SIGCONT     19) SIGSTOP     20) SIGTSTP
21) SIGTTIN     22) SIGTTOU     23) SIGURG      24) SIGXCPU     25) SIGXFSZ
26) SIGVTALRM   27) SIGPROF     28) SIGWINCH    29) SIGIO       30) SIGPWR
31) SIGSYS      34) SIGRTMIN    35) SIGRTMIN+1  36) SIGRTMIN+2  37) SIGRTMIN+3
38) SIGRTMIN+4  39) SIGRTMIN+5  40) SIGRTMIN+6  41) SIGRTMIN+7  42) SIGRTMIN+8
43) SIGRTMIN+9  44) SIGRTMIN+10 45) SIGRTMIN+11 46) SIGRTMIN+12 47) SIGRTMIN+13
48) SIGRTMIN+14 49) SIGRTMIN+15 50) SIGRTMAX-14 51) SIGRTMAX-13 52) SIGRTMAX-12
53) SIGRTMAX-11 54) SIGRTMAX-10 55) SIGRTMAX-9  56) SIGRTMAX-8  57) SIGRTMAX-7
58) SIGRTMAX-6  59) SIGRTMAX-5  60) SIGRTMAX-4  61) SIGRTMAX-3  62) SIGRTMAX-2
63) SIGRTMAX-1  64) SIGRTMAX
mariaisabel@debian:~$
```

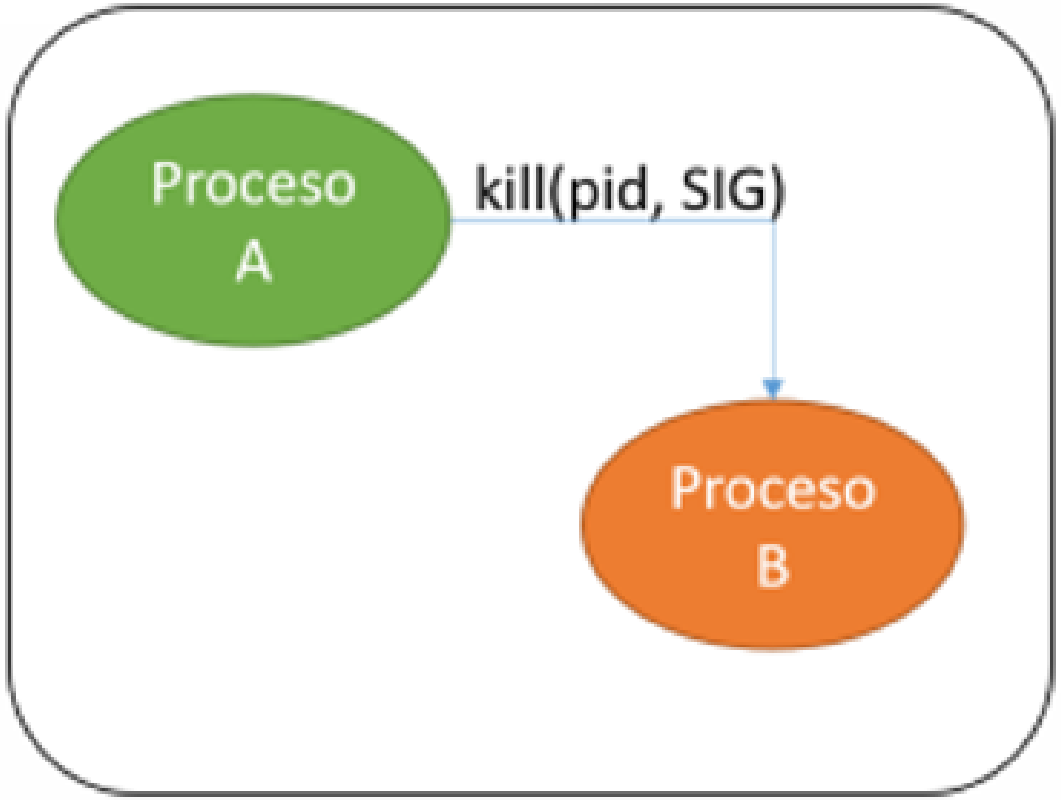
- **Comando kill**

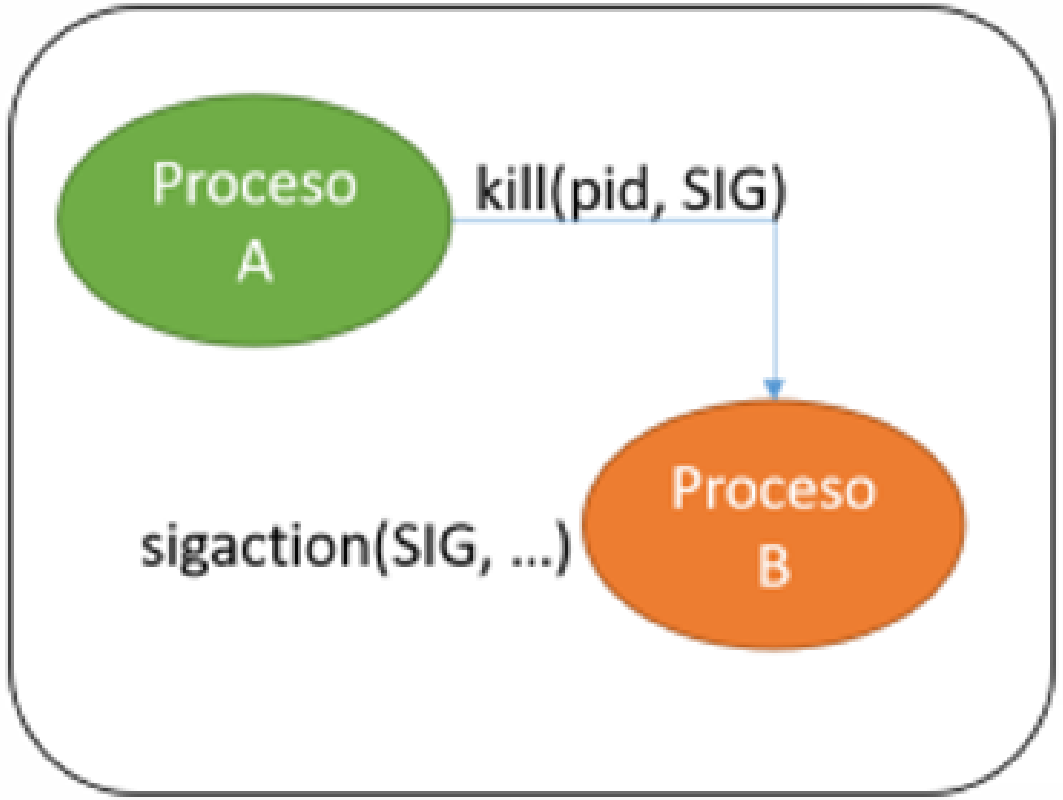
Sintaxis de `kill`

```
kill [señal] PID
```

- *señal* puede indicarse mediante el número o el código:
  - `kill -9` y `kill -KILL` son equivalentes
- las señales más comunes son:
  - `SIGHUP` (1): cuelgue del terminal o muerte del proceso controlador
  - `SIGTERM` (15): mata el proceso permitiéndole terminar correctamente
  - `SIGKILL` (9): mata el proceso sin permitirle terminar
  - `SIGSTOP` (19): para el proceso
  - `SIGCONT` (18): continúa si parado
  - `SIGINT` (2): interrupción de teclado (Ctrl-C)
  - `SIGTSTP` (20): stop de teclado (Ctrl-Z)
  - `SIGQUIT` (3): salida de teclado (Ctrl-\)

MECANISMOS DE COMUNICACIÓN ENTRE PROCESOS





MECANISMOS DE COMUNICACIÓN ENTRE PROCESOS

Archivos

- **Mecanismo básico para comunicar dos o más procesos**
- **Secuencia:**
  - **Un proceso abre y escribe los datos en un archivo Uno o más procesos abren el archivo y leen los datos de éste.**
  - **Se puede implementar mediante tuberías o archivos mapeados en memoria.**
- **En el caso de usar tuberías, la sincronización se realiza de forma implícita.**
- **En el caso de usar archivos mapeados a memoria, se debe establecer un mecanismo explícito de sincronización, para evitar que los procesos traten de leer datos antes que el proceso que está escribiendo haya terminado.**

Tuberías

- Son archivos especiales, administrados por el sistema operativo.
- Se reciben dos descriptores de archivo, uno para lectura y otro para escritura.
- Como los descriptores de archivo se preservan al realizar un fork,
- Se pueden usar como mecanismo de comunicación padre – hijo

- Secuencia:
  - El proceso "padre" crea la tubería
  - Luego, invoca fork para crear una copia
  - La copia hereda la tubería creada, por lo cual puede leer o escribir datos en la tubería
  - La tubería se usa en un solo sentido, si se requiere comunicación bidireccional se debe crear otra tubería.
  - Funciones: pipe, read, write, close, mkfifo

MECANISMOS DE COMUNICACIÓN ENTRE PROCESOS

Archivos mapeados a memoria

- **Permiten acceder al contenido de un archivo directamente en memoria.**
- **Luego de obtener un descriptor al archivo (mediante open),**
- **Se hace uso de la llamada mmap para mapear el archivo a una región de memoria administrada por el sistema operativo**
- **Los procesos pueden modificar la información del archivo directamente en la memoria, usando punteros (C).**
- **Funciones: fopen, fileno, open, read, write, close, fclose**

MECANISMOS DE COMUNICACIÓN ENTRE PROCESOS

Memoria Compartida

- **La memoria compartida permite que varios procesos diferentes tengan acceso a una región fuera de su espacio de memoria.**

- **Básicamente se implementa como un archivo virtual mapeado en memoria, este archivo es administrado por el sistema operativo.**

- **Secuencia: Un proceso crea la región de memoria compartida (fuera de su espacio de memoria), la mapea a su espacio de memoria y luego escribe datos en ella.**

MECANISMOS DE COMUNICACIÓN ENTRE PROCESOS

Memoria Compartida

- **Otros procesos abren la región de memoria, la mapean a su espacio de memoria y luego pueden leer los datos almacenados en ella.**
- **Funciones: shm_open, ftruncate, mmap, shm_ unlink**

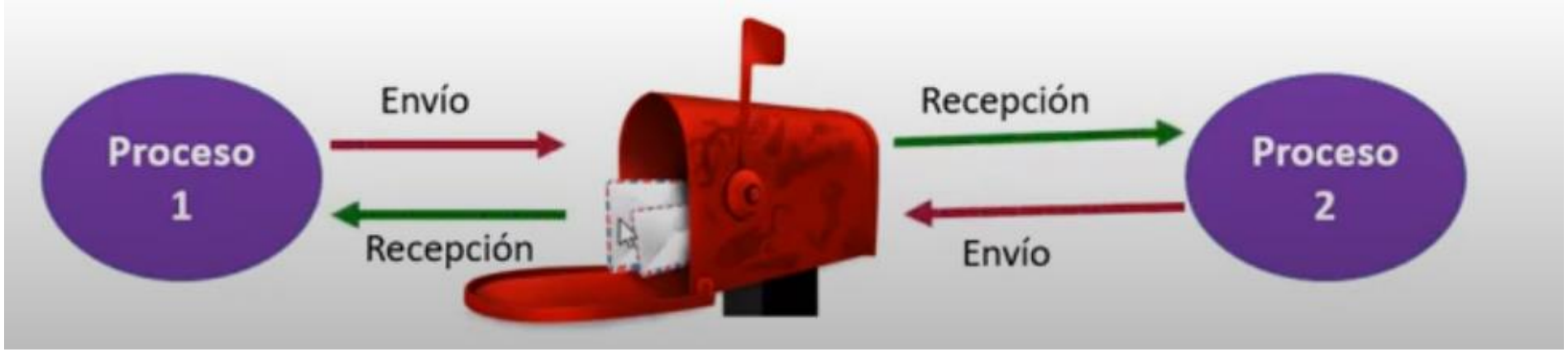
MECANISMOS DE COMUNICACIÓN ENTRE PROCESOS

Paso de mensajes

- **Mecanismo mediante el cual los procesos "envían" mensajes a una estructura central denominada "cola de mensajes", en la cual son procesados por un proceso específico.**

- **Cada mensaje tiene un tamaño predefinido, y pueden existir diferentes tipos de mensajes en una cola.**

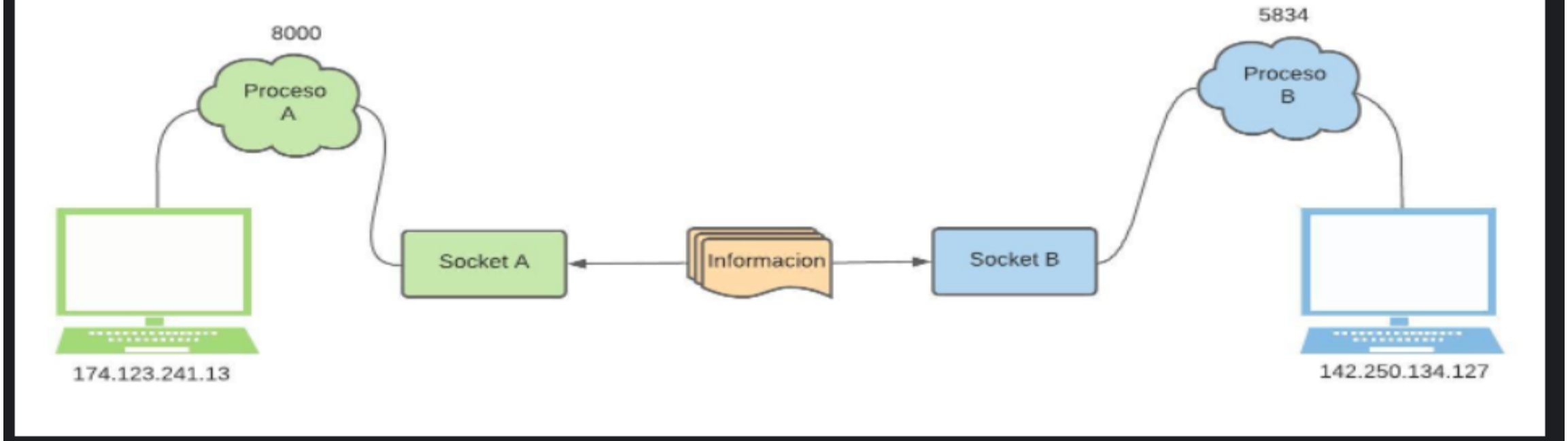
- **Funciones: mq_open, mq_close, mq_notify, mq_send, mq_receive, mq_unlink**



Sockets

- Mecanismo de comunicación local / remota
- Un socket es un conector que permite acceder a un extremo de una conexión (local o remota)
- Una vez que se ha realizado la conexión, se puede leer y escribir del socket como si se tratara de un archivo
- En el esquema local, un socket funciona como un archivo.
- Funciones: socket, bind, listen, accept, connect, send, recv, shutdown

Sockets



En este esquema hay 2 ordenadores con su dirección IP propia (`174.123.241.13` y `142.250.134.127`). En cada ordenador se ejecuta un proceso (A y B) que hacen uso de los puertos `8000` y `5834` respectivamente. Como hemos dicho antes, cada socket será identificado por su IP y su puerto.

Así pues el socket A será: `174.123.241.13:8000` y el socket B será: `142.250.134.127:5834`.

Sockets

- En el esquema remoto, la dirección a la que se conecta el socket es una dirección de internet.
- Una vez creado y conectado, se maneja exactamente igual a un socket local.
- Sobre los sockets se pueden implementar protocolos más sofisticados, como HTTP, FTP, etc.
- A su vez, sobre estos protocolos funcionan los programas más comunes hoy en día: páginas web, multimedia, mensajería, etc.
- Funciones: socket, bind, listen, accept, connect, send, recv, shutdown, htons, inet_addr.

¡Gracias por su atención!

Universidad del Cauca

www.unicauca.edu.co