Tema 1. Comunicación e sincronización entre procesos e fíos

Comunicación entre procesos (IPC)

Realmente, hai **comunicación implícita** entre procesos xa que todos *comparten o kernel* (rexión de memoria).
A comunicación entre procesos **emprégase para**:

- *Pasar información* entre procesos.
- *Evitar interferencias* (carreiras críticas entre procesos).
- Garantir unha *orde correcta* (sincronización mediante dependencias).
  Os dous últimos puntos pódense tratar coas mesmas ferramentas tanto para procesos como para fíos; pero o primeiro punto é moito máis sinxelo para *fíos*, xa que estes *comparten espazo de direccións* (os procesos non comparten memoria excepto no caso dun espazo virtual ou mediante o envío de mensaxes).

> 🖊 **Exemplo: Pipe**
> Ao executar "comando1 | comando2", o pipe colle a saída do primeiro proceso e a garda nun ficheiro temporal; despois comeza a execución do segundo proceso, que tomará o contido do ficheiro temporal (saida de comando1) como entrada. A este ficheiro só pode acceder un proceso á vez, polo que para os sistemas de **multiprogramación**, existe a opción de marcar **puntos de referencia** para que se intercalen e o lean pseudoparalelamente.

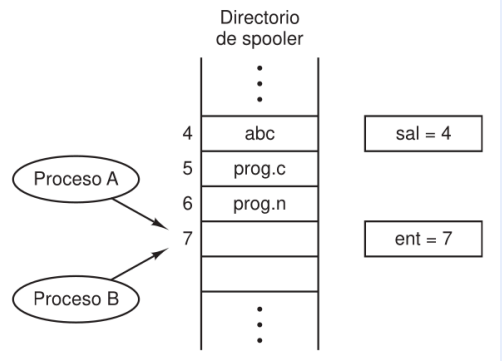
Principal inconvinte: os sistemas son *multiprogramados*, polo que hai varios fíos e procesos simultáneos, polo que poden interferir.

Condicións de carreira

Os procesos que traballan en conxunto poden **compartir un espazo de almacenamento** no que poden ler e escribir datos, e que pode estar na *memoria principal* ou pode ser un *arquivo compartido*. Como consecuencia desto, pódense producir **condicións de carrera**, que son situacións nas que dous ou máis procesos *acceden simultáneamente a datos compartidos* sen ningunha orde específica, o que pode xerar resultados inesperados. Esto fai que *depurar* programas con condicións de carrera sexa *complexo*.

Exemplo: spooler de impresión

Cando un proceso quere imprimir un arquivo, introduce o seu nome nun **directorio de spooler**. Entón, o **demonio de impresión** (outro proceso) comproba periódicamente se hai arquivos que imprimir, e se os hai, os imprime e elimina os seus nomes do directorio.
Imaxina que temos un directorio spooler con *moitas ranuras* que poden conter o nome dun arquivo, e que hai dúas *variables compartidas*: `sal`, que apunta ao *seguinte arquivo* a imprimir, y `ent`, que apunta á *seguinte ranura libre*.
Situación: Ranuras 0-3 baleiras e ranuras 4-6 cheas. Pseudosimultáneamente, os procesos A e B queren poñer un arqivo na cola de impresión.



No **peor dos casos** pasaría o seguinte:

1. *Proceso A* lee `ent` = 7 → garda 7 na súa variable local `seg_ranura_libre`
2. Ocorre unha interrupción de reloxo e o planificador lle da a *CPU ao proceso B*
3. *Proceso B* lee `ent` = 7 → garda 7 na súa variable local `seg_ranura_libre`
   ~ Ambos procesos pensan que a seguinte ranura libre é a 7. ~
4. O *proceso B* continúa ca súa execución → garda o nome do seu arquivo na ranura 7 e actualiza `ent++`
5. O *proceso A* execútase de novo dende onde quedou → ve que a súa variable local dille que a seguinte ranura libre é a 7, polo que escribe o nome do seu arquivo na ranura 7, *sobreescribindo* o que escribira antes o proceso B; e actualiza `ent++`.
   ~ **Resultado**: o nome do arquivo que escribiu o proceso B perdeuse e, polo tanto, nunca se imprimirá. ~

Para tratar problemas de carreiras críticas hai que ter sempre presente a **Lei de Murphy**, que di que *se algo pode saír mal, entón sairá mal*.

🖊 Outro exemplo de condición de carreira

(Este é das diapos de ffrivera, non aparece no Tanembaum)
Código con condicións de carrera:

```
int suma = 0;
void suma(int ni, int nf){
    int j;
    for (j = n1; j <= nf; j++)
        suma += j;
}
```

Código correxido para evitar condicións de carreira:

```
int suma = 0;
void suma(int ni, int nf){
    int j, suma_parcial = 0;
    for (j = ni; j <= nf; j++)
        suma_parcial += j;
    suma += suma_parcial;
}
```

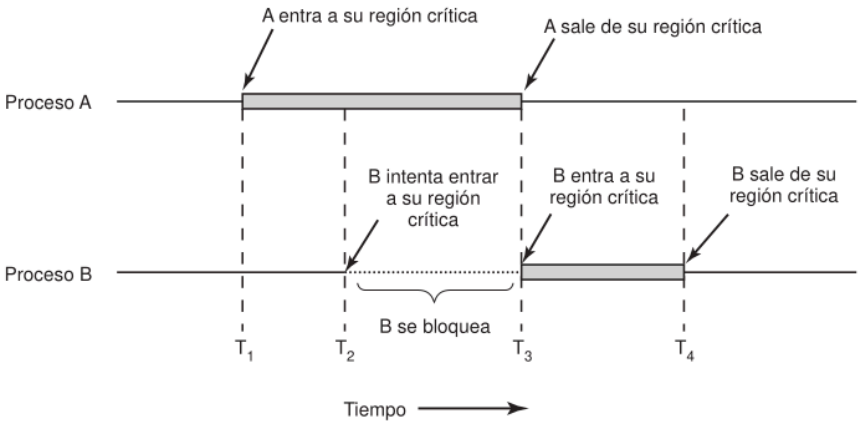
No segundo código, *redúcese a sección crítica* xa que cada fío ou proceso ten a súa variable local `suma_parcial`, polo que a zona con rexión crítica (cando se modifica `suma`) só se executa unha vez por fío/proceso (en vez de executarse dentro do for), reducindo o risco de carreiras críticas.

Rexións críticas

A **rexión crítica** é a parte do programa na que se *accede á memoria compartida*.
Para **evitar as condicións de carreira**, necesitamos **exclusión mutua**, que asegure que, se un proceso está empregando unha variable ou arquivo compartido, *os demáis procesos non poidan acceder* a esa variable ou arquivo.
Condicións para unha boa solución:

1. Non pode haber *dous procesos* na *mesma rexión crítica* simultáneamente
2. Non podemos facer *suposicións* sobre as *velocidades ou o número de CPUs*
3. Ningún *proceso* que se execute *fóra da sua rexion critica* pode *bloquear outros procesos*
4. Ningún *proceso* ten que *esperar de maneira indefinida* para entrar á súa rexión crítica

Comportamento abstracto da exclusión mutua:



Exclusión mutua con espera ocupada

A **espera ocupada** ou espera activa consiste na acción de *comprobar de forma continua unha variable* esperando certo valor e, polo xeral, debe *evitarse* debido a que desperdicia tempo de CPU.
Para lograr a exclusión mutua con espera ocupada hai varios métodos:

Deshabilitando interrupcións

Consiste en que cada proceso **deshabilite as interrupcións xusto antes de entrar á súa rexión crítica** e as rehabilite despois, evitando así que ocorran interrupcións de reloxo. Esto causa que *non se poida expropiar a un proceso*, xa que *non hai cambios de contexto*. Aínda que o *kernel o fai con frecuencia*, é *perigoso en procesos de usuario*, e resulta *inútil en sistemas multicore*.

> É unha técnica útil dentro do mesmo SO, pero non é apropiada como mecanismo de exclusión mutua xeral para os procesos de usuario.

Variables de candado

Consiste en ter unha **variable compartida** que ao principio vale 0 e que ten que ser **avaliada por cada proceso que queira entrar á rexión crítica**: se é 0, pona a 1 e entra, e se é 1, espera a que sexa 0.
Polo tanto:

- Candado = 0 → *ningún proceso* na súa rexión crítica
- Candado = 1 → *algún proceso* na súa rexión crítica

> Non funciona. **Candado de xiro*: candado que emprega a espera ocupada.

Alternancia estricta

Consiste en ter unha **variable enteira `turno` que leva a conta de a qué proceso lle toca acceder á rexión crítica**. Cada proceso *avalía se `turno` vale un número concreto* (0, 1, ...) e se é así, significa que pode entrar á rexión crítica. Despois, cando vai saír da rexión crítica, *actualiza turno* para que outro proceso poida acceder a ela.

Exemplo

Exemplo con 2 procesos:

```
while (TRUE) {
    while (turno !=0) ;
    region critica();
    turno=1;
    region_no_critica();
}
```

```
while (TRUE) {
    while (turno !=1) ;
    region critica();
    turno=0;
    region_no_critica();
}
```

> Non é unha boa solución para procesos (é mellor para fíos) xa que un proceso pode quedar bloqueado por outro máis lento (os procesos sempre traballan ao ritmo do mais lento), reducindo a productividade do procesador. Además, **incumpre a condición 3**.

Solución de Peterson

Consiste en que, cada proceso, antes de entrar á rexión crítica, chame a unha función (`entrar_region`) que fai que **o proceso espere ata que sexa seguro entrar** (while do codigo de abaixo); e cando quere saír da rexión crítica chama a `salir_region`, que actualiza o array `interesado[]`, permitindo así que *outro proceso poida accder*.

Solución de Peterson para 2 procesos:

```
#define FALSE 0
#define TRUE 1
#define N 2 // Num. de procesos

int turno;
int interesado[N]; // Array de procesos esperando. Ao principio = {0}

void entrar_region(int proceso) {
    int otro; // O numero do outro proceso
    otro = 1 - proceso;
    interesado[proceso] = TRUE; // Indicams que está ineresado
    turno = proceso; // Damoslle o turno
    while (turno == proceso && interesado[otro] == TRUE); // Espera para
entrar
}

void salir_region(int proceso) {
    interesado[proceso] = FALSE;
}
```

Exemplo de execución

1. Proceso 0 chama a `entrar_region` → `interesado[otro] = FALSE` → returnea inmediatamente (non se cumple o `&&` do while)
2. Proceso 1 chama a `entrar_region` → `interesado[otro] = TRUE` → quédase esperando no while ata que o outro proceso execute `salir_region` e poña o seu `interesado = FALSE`

> **¿Espera activa?** Sí debido a que o while compróbase continuamente polo proceso que estea esperando

Instrucción TSL (Hardware)

É unha **instrucción atómica** que require **axuda do hardware** xa que **bloquea o bus de memoria** mentres se executa. A instrucción é da *forma*: `TSL REXISTRO, CANDADO`

Que fai? *Le* o contido de `CANDADO` e o *garda* no rexistro `REXISTRO`, para despois *almacenar un valor* (distinto de 0) en `CANDADO`. Esto faino **atómicamente** gracias a que a *CPU bloquea o bus de memoria* mentres tanto.

> *Bloquear o bus de memoria != Deshabilitar as interrupcións*
> Cando se deshabilitan as interrupcións non se evita que outro procesador acceda á palabra, xa que deshabilitar as interrupcións no procesador 1 non ten ningún efecto no procesador 2. Para que o procesador 2 non interfira a ñunica forma é bloquear o bus (require de hardware).

Para **empregar TSL** necesitamos unha *variable compartida* `CANDADO` que coordine o acceso ás rexións críticas: cando `CANDADO == 0` → calquera proceso pode poñelo a 1 (chamada a TSL) e devolvelo a 0 cando remate (`MOVE`).

Para evitar que dous procesos entren ao mesmo tempo nunha rexión crítica:

```
entrar_region:
    TSL REGISTRO,CANDADO ;Copia o candado ao rexistro e pon candado a 1
    CMP REGISTRO,#0 ;Comproba se candado era 0
    JNE entrar_region ;Se non era 0, volve a intentalo
    RET ;Se era 0, volve ao procedemento chamador

salir_region:
    MOVE CANDADO,#0 ;Pon o candado a 0
    RET ;Volve ao chamador
```

Instrucción alternativa: XCHG

A instrucción XCHG **intercambia o contido de dúas ubicacións de forma atómica**. Está implementada nos procesadores Intel x86, que a empregan para a *sincronización a baixo nivel*. Polo tanto, o codigo 1 e o codigo 2 son equivalentes (se o 2 se executara de forma atómica):

```
;Codigo 1
XCHG REG1, REG2

;Codigo 2
MOVE TMP, REG1
MOVE REG1, REG2
MOVE REG2, TMP
```

Sleep and Wakeup

As *solucións anteriores* comproban se un proceso que pide entrar a unha rexión crítica pode facelo, se non pode, quédase esperando. Esto pode ocasionar un **problema de inversión de prioridades**, que ocorre cando un proceso de *baixa prioridade* mantén un recurso (como un candado), e un proceso de *alta prioridade* que o necesita non pode continuar porque o proceso de baixa prioridade non pode executarse para liberarlo.

Para *evitar as esperas activas* que poden dar lugar a problemas de inversión de prioridades, pódense empregar as **chamadas ao sistema** `sleep`, que fai que o sistema se bloquee ou desactive ata que outro programa a active, e `wakeup`, que desperta ao proceso pasado como parámetro. *Para asociar ambas chamadas, emprégase unha dirección de memoria* (zona de memoria compartida).

O problema do productor-consumidor

Mais info no Informe 1 da practica 2 de SOII
O **problema produtor-consumidor (ou buffer limitado)** consiste en **dous procesos** (produtor e consumidor) que comparten un **buffer de tamaño fixo**. Basicamente, o *produtor* dedícase a *colocar elementos* no buffer mentres o *consumidor os elimina*. Cando un dos dous *non pode realizar a súa función* (colocar ou eliminar), *vai durmir* (cun `sleep`) e agarda a que *o outro proceso o esperte* (cun `wakeup`). Para *levar a conta dos elementos* do buffer, ambos procesos comparten a variable `conta`, que é actualizada polos procesos según van introducindo ou eliminando elementos do buffer.

O **problema** ocorre cando a variable `conta` *non se actualiza correctamente*, xa que, ao non ter acceso restrinxido, é susceptible a *carreiras críticas*, o que acaba provocando tanto o produtor como o consumidor queden durmidos para sempre.

Problema do productor-consumidor cunha condición de carreira crítica:

```
#define N 100 // Tamaño do buffer
int cuenta = 0; // Número de elementos que contén o buffer

void productor(void) {
    int elemento;
    while (TRUE) {
        elemento = producir_elemento();
        if (cuenta == N) sleep(); // O buffer está cheo
        insertar_elemento(elemento);
        cuenta = cuenta + 1;
        if (cuenta == 1) wakeup(consumidor);
    }
}

void consumidor(void) {
    int elemento;
    while (TRUE) {
        if (cuenta == 0) sleep(); // O buffer está baleiro
        elemento = quitar_elemento();
        cuenta = cuenta - 1;
        if (cuenta == N - 1) wakeup(productor);
        consumir_elemento(elemento);
    }
}
```

*Para solucionar o problema de perda de sinais, pódese engadir un *bit de espera de espertar*, que se fixe cando un proceso que esté esperto, reciba un bit de espera a espertar, aínda uqe é unha solución pouco escalable.

Semáforos

Tal e como os propuxo Dijkstra, un **semáforo** é unha *variable enteira* que conta o *número de sinais de espertar* e as almacena para un uso futuro. Ríxese por dúas **operacións atómicas**:

- `up`: *incrementa o valor* do semáforo. Se hai algún *proceso inactivo* nese semáforo, que non completara o seu `down`, o sistema selecciona algún proceso inactivo e *lle permite completalo*.
- `down`: se o valor do semáforo é *maior a 0*, o *decrementa*, e se é *0*, *bloquea* o proceso.

Resolución do problema do productor-consumidor con semáforos

Mais info no Informe 2 da practica 2 de SOII
Os semáforos serven para resolver a perda de sinais de espertar. Normalmente, impleméntanse `up` e `down` como **chamadas ao sistema** que *deshabilitan brevemente tódalas interrupcións* mentres avalían o semáforo, o actualizan e poñen o proceso a durmir se fai falta.

Esta solución emprega **tres semáforos**: un para contabilizar o número de *ranuras cheas*, outro para as *baleiras*, e un chamado *mutex* para que o productor e o consumidor non accedan ao buffer simultaneamente. Así, aseguramos a **exclusión mutua** facendo que cada proceso execute un *down antes de entrar* á rexión crítica, e un *up despois de salir* da mesma.
*A semáforos que se inicializan a 1 e se comportan como o mutex se lles chama *semáforos binarios*

Solución:

```
#define N 100
typedef int semaforo;
semaforo mutex = 1;
semaforo vacias = N;
semaforo llenas = 0;

void productor(void) {
    int elemento;
    while (TRUE) {
        elemento = producir_elemento();
        down(&vacias);
        down(&mutex);
        insertar_elemento(elemento);
        up(&mutex);
        up(&llenas);
    }
}

void consumidor(void) {
    int elemento;
    while (TRUE) {
        down(&llenas);
        down(&mutex);
        elemento = quitar_elemento();
        up(&mutex);
        up(&vacias);
        consumir_elemento(elemento);
    }
}
```

Mutexes

Básicamente son como unha *versión simplificada dos semáforos* (sen a habilidade de contar), é decir, unha variable que toma valores binarios (0 ou 1). Aos mutexes dáselles ben únicamente **administrar a exclusión mutua**, e son eficientes e sinxelos de implementar. Están rexidos por **dúas operacións atómicas**:

- `lock`: se o mutex está aberto (*mutex = 0*) → a rexión crítica está dispoñible → a chamada ten éxito (en caso contrario, o fío bloquéase ata que a rexión crítica estea dispoñible)

- `unlock`: pon o *mutex a 1* → desbloquea a rexión crítica

As operacións `lock` e `unlock` pódense implementar facilmente *con instruccións TSL ou XCHG*:

```
mutex_lock:
    TSL REXISTRO, MUTEX    ; copia mutex ao rexistro e pon mutex a 1
    CMP REXISTRO, $0       ; mutex == 0?
    JZE ok                 ; if (mutex == 0) return
    CALL thread_yield      ; if (mutex != 0) planifica outro fio
ok: RET                    ; return (entra na rexion critica)

mutex_unlock:
    MOVE MUTEX, $0         ; pon o mutex a 0
    RET                    ; return (volve ao procedemento chamador)
```

Diferencia entre procesos e fíos

Unha **diferencia entre procesos e fíos** é que no caso dos procesos, tarde ou cedo o proceso que mantén o mutex pasa a executarse (gracias o clock) e o libera; mentres que no caso dos fíos **non hai un reloxo que pare os fíos que levan moito tempo executándose**. Por eso no procedemento `mutex_lock`, cando este non pode adquirir un mutex chama a `thread_yield` para ceder a CPU a outro fío, polo que *non hai espera ocupada* per se (xa que cando o fío se volve a executar, volve a avaliar o mutex). De esta forma, dado que `thread_yield` só é unha chamada ao planificador de fios, as funcións de `mutex_lock` e `mutex_unlock` *non requiren chamadas ao kernel*, o que fai que sexan procedementos *moi rápidos*.

Diferencia entre procesos e fíos

Outra diferencia entre procesos e fíos e que **os fíos comparten un espazo de direccións común mentres que os procesos non**. No caso dos *procesos*, teñen algunhas *estruturas de datos compartidas* que se poden *almacenar no kernel* e acceder a elas mediante *syscalls*; ademáis de poder compartir un cacho do seu espazo de direccións con outros procesos (no peor dos casos, tamén se pode empregar un arquivo compartido). Se os procesos compartisen os seus espazos de direccións, a diferencia entre procesos e fíos sería case nula, pero aínda así, dado que o kernel está moi involucrado na administración dos procesos, estes nunca terán a eficiencia dos fíos a nivel de usuario.

Chamadas de Pthreads relacionadas con mutexes:

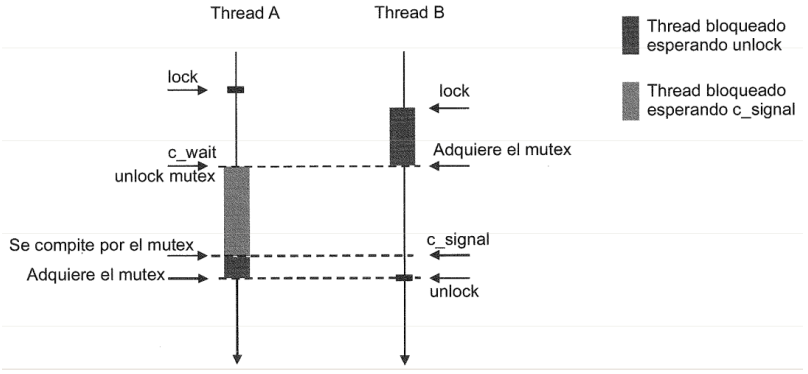
Llamada de hilo	Descripción
Pthread_mutex_init	Crea un mutex
Pthread_mutex_destroy	Destruye un mutex existente
Pthread_mutex_lock	Adquiere un mutex o se bloquea
Pthread_mutex_trylock	Adquiere un mutex o falla
Pthread_mutex_unlock	Libera un mutex

Variables de condición

As variables de condición son un *mecanismo de sincronización* que serve para **bloquear threads ata que se cumpla unha condición** determinada, polo que están *asociadas a mutexes*. Permiten que a espera e bloqueo se realicen de forma *atómica*.
Chamadas a Pthreads relacionadas con variables de condición:

Llamada de hilo	Descripción
Pthread_cond_init	Crea una variable de condición
Pthread_cond_destroy	Destruye una variable de condición
Pthread_cond_wait	Bloquea en espera de una señal
Pthread_cond_signal	Envia señal a otro hilo y lo despierta
Pthread_cond_broadcast	Envia señal a varios hilos y los despierta

Para a **sincronización de fíos**, empréganse tanto *mutexes* como *variables de condición* en conxunto, da seguinte forma: un *fío pecha un mutex* e despois *espera a unha variable de condición* cando non poida obter o que necesita.



Ver Problema do productor-consumidor resolto con mutexes e variables de condición. (Practica 3)

Monitores

Un monitor é un **mecanismo de sincronización de alto nivel** que asegura unha *menor probabilidade de erro* pero é *menos versátil*. Ten unha estrutura similar á de un *obxecto* na que se definen procedementos, variables e estruturas de datos.
Permiten a **exclusión mutua garantizada** gracias a que nun monitor só pode haber *un proceso activo* en cada instante: o compilador recoñece o monitor e programa a exclusión mutua:

- Para *pausar o proceso* emprega variables de condición → wait e signal
- Para *evitar ter máis dun proceso activo* ao mesmo tempo hai varias solucións (como facer que a última función do monitor sexa un signal)

Transmisión de mensaxes

É un **método de comunicación entre procesos** que emprega dúas chamadas ao sistema, `send` e `receive` polo que se poden colocar facilmente en procedementos de biblioteca como:

- `send(destino, &mensaxe);`
- `receive(orixe, &mensaxe);`
  Ao igual que se explicou en Redes, se o receptor non recibiu ningún *ack* dentro do timeout, volve mandar a mensaxe, que ten un *número de secuencia* asociado para que non se perda información.

As mensaxes se poden **direccionar** de varias fromas:

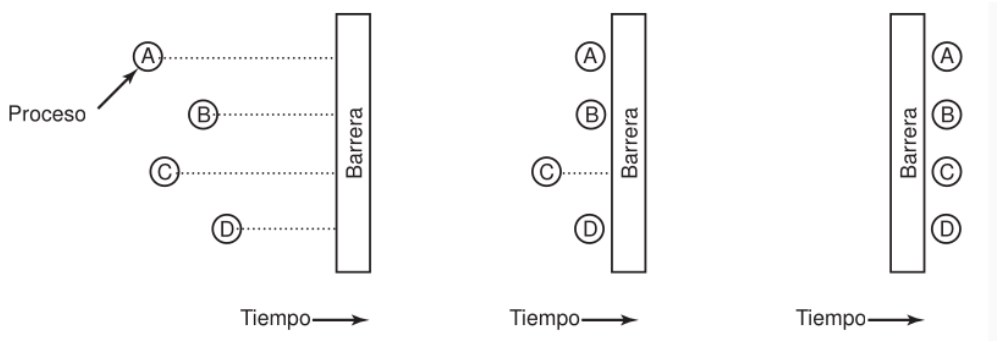
- **Buzón**: é un onde *se colocan nun buffer certo número de mensaxes*, polo que actúan como intermediarios mentres se aceptan ou non as mensaxes no destino.
- **Encontro**: consiste en *eliminar todo uso do buffer* e facer que o emisor se bloqueee se a operación `send` remata antes que a `receive`.

Problema do productor-consumidor con transmisión de mensaxes

O consumidor envía N mensaxes baleiras ao productor. Cada vez que o productor produce un item, recibe unha mensaxe baleira e envía unha chea de volta, polo que sempre hai N mensaxes circulando.
*Ver Práctica 4

Barreiras

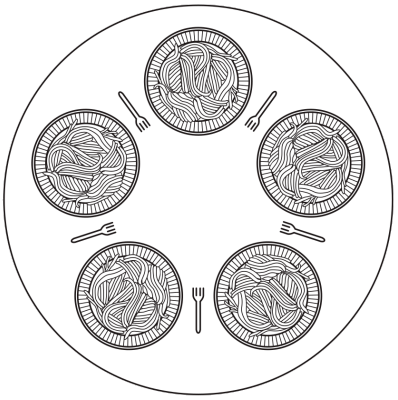
É un **mecanismo de sincronización** destinado aos **grupos de procesos** para lograr que *ningún proceso poida continuar á seguinte fase ata que todos estén listos*. Para esto, se coloca unha barreira ao final de cada fase, de forma que *cando un proceso chega á barreira, se bloquea* ata que tódolos procesos cheguen.



Problemas de comunicación entre procesos

O problema dos filósofos comelones

Tal e como o propuxo Dijkstra en 1965, dicía así:

> Cinco filósofos se atopan sentados arredor dunha mesa circular. Estes filósofos só comen e pensan, e cada un ten un plato de espaguetis. Cada filósofo necesita dous tenedores para comer os espaguetis, pero entre cada par de pratos só hai un tenedor.
>
> 
>
> Desta forma, cando un filósofo ten fame, intenta coller os tenedores esquerdo e dereito, un de cada vez, en calquera orde. Se ten éxito, come por un momento, deixa os tenedores e segue pensando.

Unha posible solución sería a seguinte:

```
#define N 5 // Número de filósofos
void filosofo(int i) {
    while(TRUE){
        pensar();
        tomar_tenedor(i);
        tomar_tenedor((i+1) % N);
        comer(); // yum yum espaguetti :)
        poner_tenedor(i);
        poner_tenedor((i+1) % N);
    }
}
```

Pero, se todos collen un só tenedor, ningún podería comer xa que ningún podería adquirir nunca dous tenedores, polo que se produciría un *interbloqueo* (inanición).

Unha *solución correcta* sería:

```
#define N 5
#define IZQ (i-1)%N
#define DER (i+1)%N
#define PENS 0 // Pensando
#define FAME 1
#define COM 2 // Comendo
typedef int semaforo;
int estado[N]; // Array do estado dos filósofos
semaforo mutex = 1;
semaforo s[N]; // Array de semaforos dos filosofos

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

void tomar_tenedores(int i) {
    down(&mutex);
    estado[i] = FAME; // Indica que ten intencion de collelos tenedores
    probar(i); // Intenta collelos tenedores
    up(&mutex);
    down(&s[i]);
}

void poner_tenedores(i) {
    down(&mutex);
    estado[i] = PENS; // Indica que deixou de comer
    probar(IZQ); // Mira se o filosofo esquerdo pode comer
    probar(DER); // Mira se o filosofo dereito pode comer
    up(&mutex);
}

void probar(i) {
    if (estado[i] == FAME && estado[IZQ] != COM && estado[DER] != COM){
        estado[i] = COM;
        up(&s[i]);
    }
}
```

De esta forma, un filósofo só pode empezar a comer se ningún vecino está comendo.

O problema dos lectores e escritores

Trata sobre o acceso a unha **base de datos**. *Varios procesos poden ler* á vez da BD, pero *só un pode actualizala*, de froma que ata que non remate, ningún outro proceso pode acceder a ela.
Para programar os lectores e escritores da base de datos, unha **posible solución** é a seguinte: o *primer lector* en obter acceso á BD, faille un *down ao semáforo* `bd`; os *seguintes lectores incrementan e decrementan un contador* (`cl`) según van chegando e saindo; e o *último en saír*, faille un *up ao semáforo* de forma que, se hai un *escritor bloqueado, este poida acceder*.

```
typedef int semaforo;
semaforo mutex=1; // Mutex para cl
semaforo bd=1; // Semaforo de acceso a BD
int cl=0; // Contador de lectores
void lector(void) {
    while(TRUE){
        down(&mutex);
        cl = cl + 1;
        if (cl == 1) down(&bd); // Se e o primeiro lector fai un down
        up(&mutex);
        leer_base_de_datos();
        down(&mutex);
        cl = cl - 1;
        if (cl == 0) up(&bd); // Se e o ultimo lector fai un up
        up(&mutex);
        usar_lectura_datos();
    }
}
void escritor(void) {
    while(TRUE){
        pensar_datos();
        down(&bd);
        escribir_base_de_datos();
        up(&bd);
    }
}
```

Non obstante, esto ocasiona un **problema**, xa que mentres haxa un lector activo, os seguintes lectores serán admitidos, aínda que haxa un escritor esperando. Esto pode causar que, se hai un suministro contínuo de lectores, *o escritor nunca poderá acceder á base de datos*. Para **solucionar** este problema, podemos facer que *cando chega un lector, se hai un escritor en espera, o primeiro suspéndese* detrás do escritor. De esta forma, *un escritor non terá que esperar polos lectores que cheguen despois de el*.