

Sistemas Operativos

7. Comunicación entre procesos y sincronización

Javier García Algarra
javier.algarra@u-tad.com
2020-2021

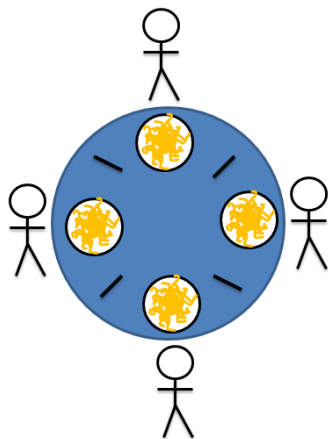
Introducción

En la actualidad, nadie diseña aplicaciones monolíticas, en las que un único proceso realice todas las funciones. Las buenas prácticas de desarrollo aconsejan un modelo modular, con varios procesos especializados cooperantes. Algunos de ellos pueden ser software de terceros y cada vez es más fuerte la tendencia a que los procesos puedan residir en distintas máquinas, con Sistemas Operativos diversos.

El Sistema Operativo tiene que ofrecer mecanismos estandarizados, eficientes y seguros para que los procesos puedan intercambiar información. La abreviatura IPC (Inter Process Communication) se utiliza de forma habitual para englobar todos estos mecanismos. Además, es imprescindible que el Sistema Operativo solucione otro problema que aparece en el trabajo colaborativo, la sincronización. ¿Cómo se evita que varios procesos puedan acceder de forma simultánea a un recurso único de forma desordenada?

Introducción

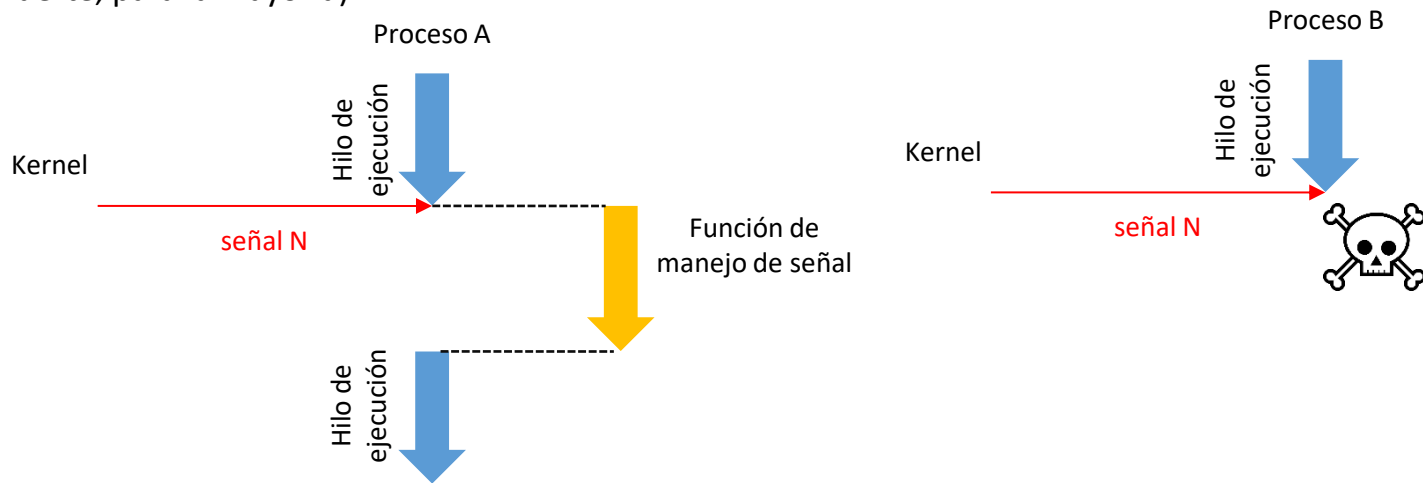
En muchos cursos de Sistemas Operativos se aprovecha la aparición de los mecanismos de sincronización para hacer una introducción a la programación concurrente. Nosotros nos limitaremos a describir qué ofrecen los sistemas manejar este tipo de situaciones pero no vamos a describir ni resolver el problema de la cena de los filósofos, el de la panadería, el del banquero y otros clásicos haciendo spoiler de otras asignaturas.



Ese filósofo que entra en una panadería y se encuentra con un banquero... Jarlll!

IPC: Señales

Las señales (signals) son un mecanismo de comunicación asíncrona del kernel con los procesos. Funcionan de forma parecida a una interrupción al nivel del software de usuario. Si el proceso receptor está preparado para atenderla, ejecuta la función de servicio de esa señal y en el retorno prosigue su flujo normal. Si no dispone de una función específica, se ejecuta la acción por defecto (muerte, para la mayoría).



El proceso A tiene preparado (armado) el manejador para la señal (*signal handler*)

El proceso B no tiene preparado el manejador de señal N
cuya acción por defecto es terminar el proceso

IPC: Señales

Listado de señales definidas en el estándar POSIX. Cada señal tiene un número asignado pero la señal no transporta más información. En este cuadro, cuando en Portable Number aparece como N/A significa que esa cifra depende del SO.

Las señales SIGUSR1 y SIGUSR2 no tienen un significado asignado de antemano, pueden ser usadas por el usuario con libertad.

Un proceso puede enviar una señal a otro, siempre que pertenezcan al mismo usuario o grupo, usando la función `kill()`, que por motivos históricos mantiene ese nombre pero sirve para cualquier señal. Lo mismo sucede con el comando `kill` de la shell.

| Signal | Portable number | Default action | Description |
|-----------|-----------------|-----------------------|---|
| SIGABRT | 6 | Terminate (core dump) | Process abort signal |
| SIGALRM | 14 | Terminate | Alarm clock |
| SIGBUS | N/A | Terminate (core dump) | Access to an undefined portion of a memory object |
| SIGCHLD | N/A | Ignore | Child process terminated, stopped, or continued |
| SIGCONT | N/A | Continue | Continue executing, if stopped |
| SIGFPE | 8 | Terminate (core dump) | Erroneous arithmetic operation |
| SIGHUP | 1 | Terminate | Hangup |
| SIGILL | 4 | Terminate (core dump) | Illegal instruction |
| SIGINT | 2 | Terminate | Terminal interrupt signal |
| SIGKILL | 9 | Terminate | Kill (cannot be caught or ignored) |
| SIGPIPE | 13 | Terminate | Write on a pipe with no one to read it |
| SIGPOLL | N/A | Terminate | Pollable event |
| SIGPROF | N/A | Terminate | Profiling timer expired |
| SIGQUIT | 3 | Terminate (core dump) | Terminal quit signal |
| SIGSEGV | 11 | Terminate (core dump) | Invalid memory reference |
| SIGSTOP | N/A | Stop | Stop executing (cannot be caught or ignored) |
| SIGSYS | N/A | Terminate (core dump) | Bad system call |
| SIGTERM | 15 | Terminate | Termination signal |
| SIGTRAP | 5 | Terminate (core dump) | Trace/breakpoint trap |
| SIGTSTP | N/A | Stop | Terminal stop signal |
| SIGTTIN | N/A | Stop | Background process attempting read |
| SIGTTOU | N/A | Stop | Background process attempting write |
| SIGUSR1 | N/A | Terminate | User-defined signal 1 |
| SIGUSR2 | N/A | Terminate | User-defined signal 2 |
| SIGURG | N/A | Ignore | Out-of-band data is available at a socket |
| SIGVTALRM | N/A | Terminate | Virtual timer expired |
| SIGXCPU | N/A | Terminate (core dump) | CPU time limit exceeded |
| SIGXFSZ | N/A | Terminate (core dump) | File size limit exceeded |
| SIGWINCH | N/A | Ignore | Terminal window size changed |

IPC: Señales

SIGINT: El proceso recibe esta señal cuando el usuario pulsa CTRL+C



SIGKILL: No enmascarable, el resultado es la muerte del proceso



SIGALRM: Ha vencido un temporizador



SIGCHLD: Proceso hijo muerto, no pasa nada si se ignora.



IPC: Memoria compartida

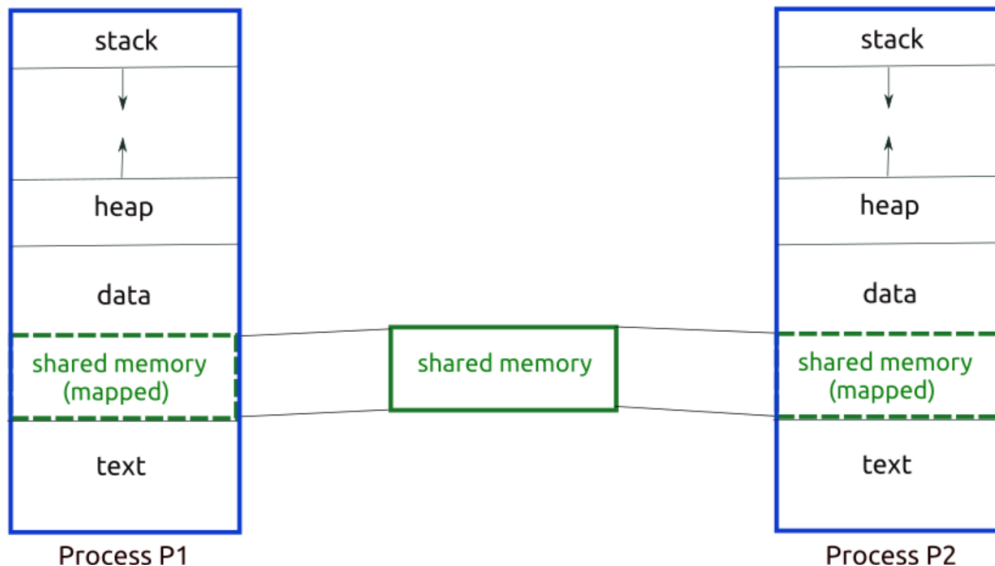
Una de las razones de que existan los sistemas operativos es separar los espacios de memoria de los procesos para que X no pueda leer ni modificar los datos de Y. Parece que no tiene mucho sentido hablar de memoria compartida entre procesos y, sin embargo, es uno de los mecanismos de comunicación en Unix.

La memoria compartida es un servicio que proporciona el kernel a los procesos. Uno de ellos debe pedirla de forma explícita. El Sistema reserva en RAM un rango que mapea como otro cualquiera en el mapa de memoria virtual del peticionario. El acceso a esta memoria requiere una clave que el peticionario puede compartir con otro proceso. Cuando esto ocurre, el sistema operativo mapea esos marcos también en las páginas de memoria virtual de este segundo proceso. A partir de ese instante, ambos procesos pueden acceder tanto en escritura como en lectura al buffer de memoria compartida. La compartición no se limita a dos procesos, puede haber más con permiso de acceso.

La memoria compartida es un recurso crudo, no tiene ningún mecanismo de sincronización. Nada garantiza que mientras el proceso B está leyendo una cadena de caracteres de ese espacio el proceso A no los modifique, por lo que se usa en combinación con señales o candados para controlar los accesos concurrentes.

IPC: Memoria compartida

Puede haber N procesos usando la misma memoria compartida, así que el Sistema Operativo tiene que llevar la cuenta de cuántos de ellos están activos y no liberar el espacio hasta que no quede ninguno cargado.

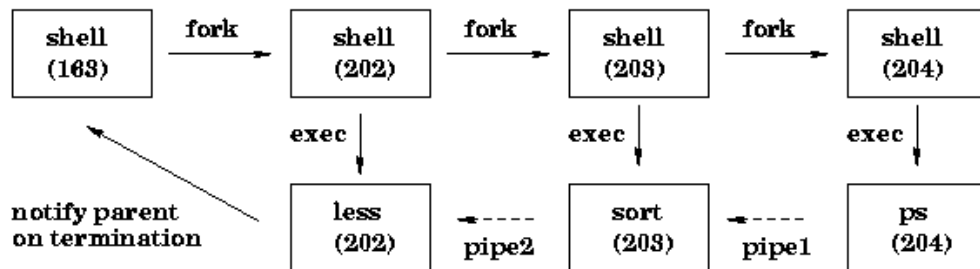


IPC: Pipes

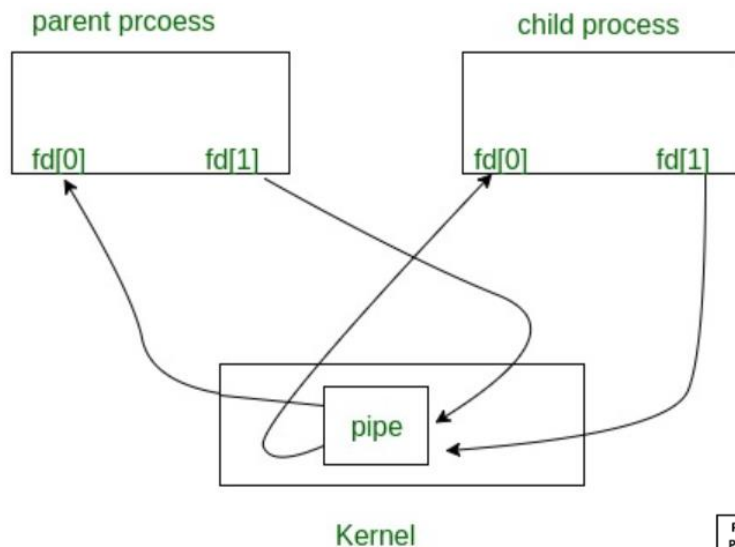
Los **pipes** (tuberías) nacieron durante el desarrollo de Unix para poder alimentar la salida estándar de un proceso a la entrada estándar del siguiente. Son un medio de comunicación secuencial y no estructurado, el lado receptor está a la escucha y recibe un flujo de bytes en el mismo orden en que se envían.

Es unidireccional y sin contención, esto significa que no puede pararse al emisor desde el receptor salvo que se use otro mecanismo como una señal. Las tuberías estándar en Unix son anónimas y duran mientras existan los dos extremos de la comunicación.

ps | sort | less

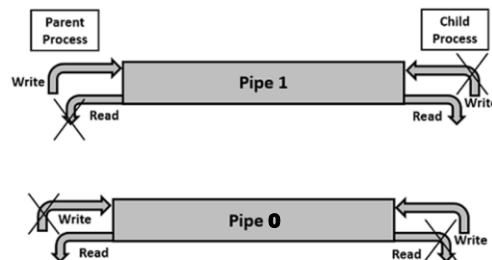


IPC: Pipes



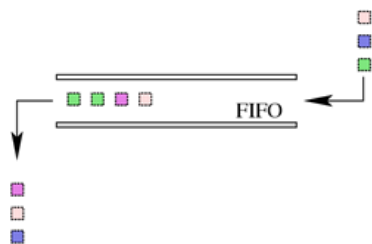
Cuando hacemos un `fork()`, el hijo hereda los descriptores abiertos del padre.

Una manera de poner a ambos en comunicación es crear dos pipes (no nombrados) desde el padre (`fd[0]` y `fd[1]`). El padre envía datos al hijo por `fd[1]` y el hijo contesta por `fd[0]`.



IPC: Pipes

Una extensión funcional de esta idea son las **tuberías con nombre** (named pipes), también llamadas **FIFO**. Las tuberías con nombre se manejan de forma muy parecida a los ficheros (tienen una entrada especial) con path, nombre y comandos de apertura, escritura, lectura y cierre. Son persistentes, hay que borrarlas de forma explícita. Tienen algunas diferencias con un fichero normal, no pueden abrirse en lectura y escritura de forma simultánea, la lectura y escritura son bloqueantes y no puede moverse el puntero con una instrucción del tipo `fseek()`, por eso son unidireccionales y FIFO. Suelen usarse en arquitecturas tipo cliente/servidor.



```

stack@undercloud-0:~/test
File Edit View Search Terminal Help
-rw-rw-r--. 1 stack stack 0 Aug 8 13:51 file5
[stack@undercloud-0 test]$ tail -n 2 < contents.txt
-rw-rw-r--. 1 stack stack 0 Aug 8 13:51 file4
-rw-rw-r--. 1 stack stack 0 Aug 8 13:51 file5
[stack@undercloud-0 test]$ ls
contents.txt file1 file2 file3 file4 file5
[stack@undercloud-0 test]$ clear

[stack@undercloud-0 test]$ mkfifo my-named-pipe
[stack@undercloud-0 test]$ ls -al
total 8
drwxrwxr-x. 2 stack stack 112 Aug 8 13:58 .
drwx----- 11 stack stack 4096 Aug 8 13:15 ..
-rw-rw-r--. 1 stack stack 416 Aug 8 13:53 contents.txt
-rw-rw-r--. 1 stack stack 0 Aug 8 13:51 file1
-rw-rw-r--. 1 stack stack 0 Aug 8 13:51 file2
-rw-rw-r--. 1 stack stack 0 Aug 8 13:51 file3
-rw-rw-r--. 1 stack stack 0 Aug 8 13:51 file4
-rw-rw-r--. 1 stack stack 0 Aug 8 13:51 file5
prw-rw-r--. 1 stack stack 0 Aug 8 13:58 my-named-pipe
[stack@undercloud-0 test]$ ls -al > my-named-pipe
[stack@undercloud-0 test]$
  
```

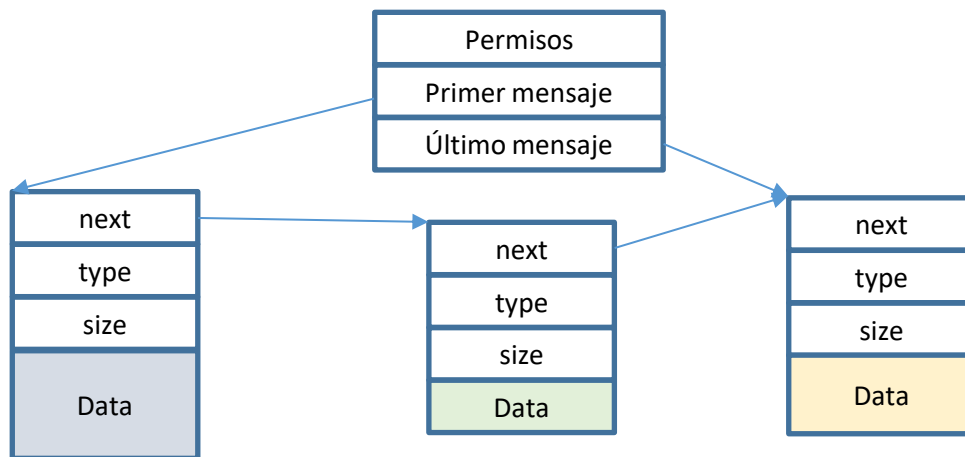
Creación desde la shell con `mkfifo`

Entrada en el directorio

IPC: Colas de mensajería

Las colas de mensajería son un mecanismo de comunicación asíncrono. El envío y recepción no son bloqueantes. Utilizan el [patrón publisher/suscribir](#). Existen varias implementaciones de las que veremos en la práctica las dos más usadas en los sistemas Unix: System V (más antigua) y POSIX.

Las colas se crean y manejan como ficheros especiales y pueden tener nombre o ser anónimas. El nombre sirve para que los procesos sepan a qué cola concreta deben acceder.



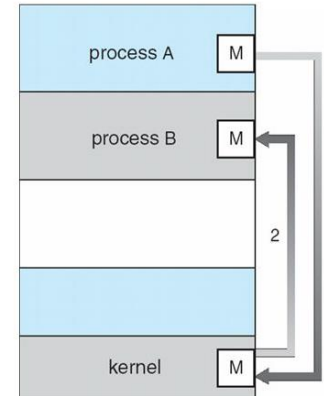
Modelo lógico de cola de mensajería

IPC: Colas de mensajería

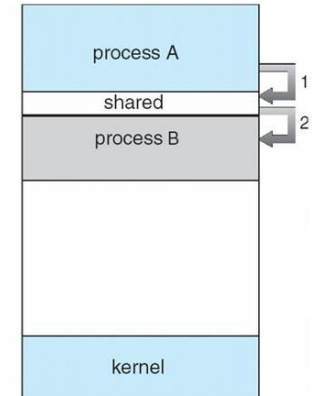
La cola de mensajes se instancia como una estructura de datos en el espacio del kernel, que es el que se encarga de sincronizar los accesos para evitar colisiones.

Las colas se tienen que crear expresamente, pueden definirse como solo de lectura, solo de escritura o bidireccionales. Lo normal es que sea el servidor el que la cree. Los procesos que hayan habilitado una cola determinada pueden enviar y recibir mensajes por ella indistintamente, en esto también se distinguen de los pipes nombrados.

Message Passing



Shared Memory



Diferencias entre memoria compartida y cola de mensajería, Silberschatz, Galvin & Gagne (2013)

IPC: Sockets

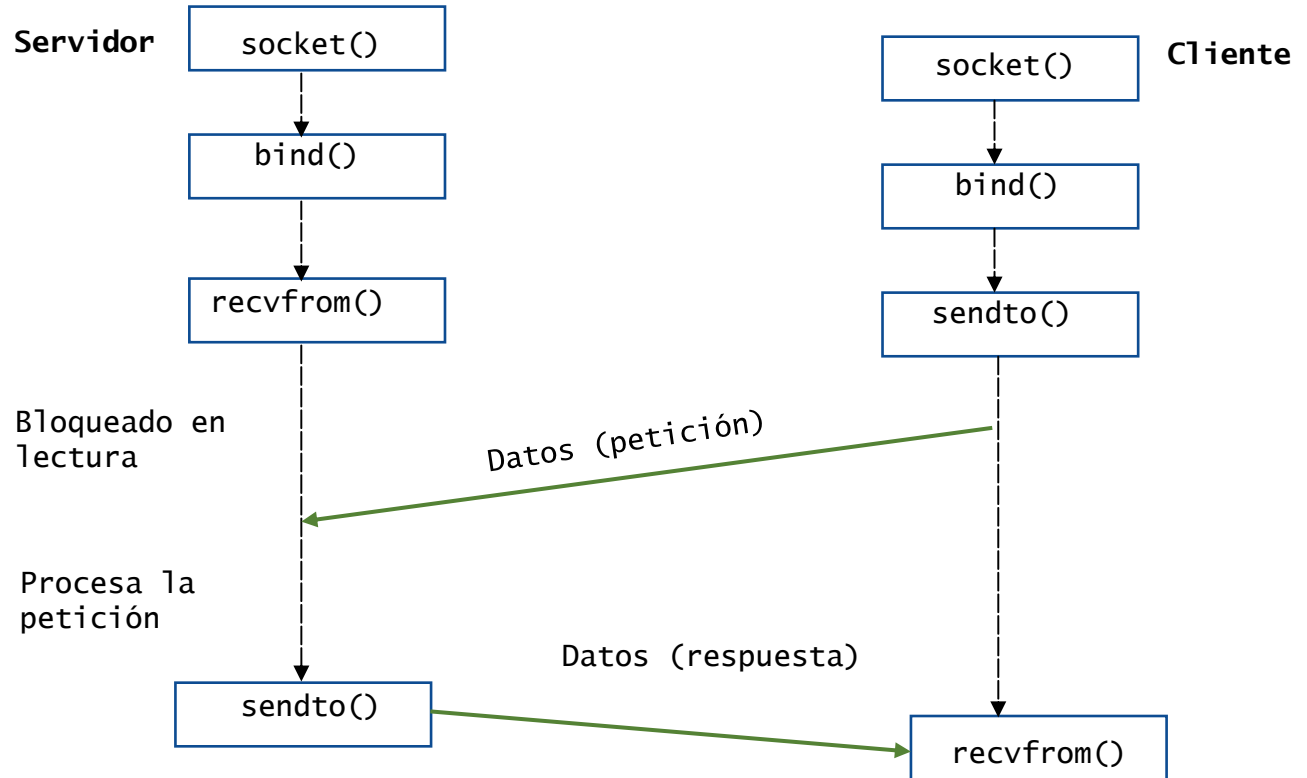
En las versiones iniciales de Unix no se usaban comunicaciones TCP/IP. En 1982, se definieron los sockets en la versión 4.1 de BSD. Son puntos lógicos de terminación de un canal comunicación entre procesos, soportado por una red IP. La expansión de internet hizo que los sockets se convirtieran en un sistema de comunicación universal que acabaron adoptando todos los sistemas operativos (en Microsoft, Windows 95 y Windows NT fueron las primeras versiones en soportarlo).

Hay dos tipos de socket, **datagram** o no orientados a sesión, soportados en UDP, y **stream** u orientados a conexión, que usan TCP. La abstracción socket define el punto de acceso y los procedimientos para engancharse (*bind*) y gestionar el intercambio de información.

Los sockets soportan una gran cantidad de protocolos de aplicación que usamos a diario como HTTP/HTTPS, SSH, SMTP/POP3, SNMP, etc.

IPC: Sockets

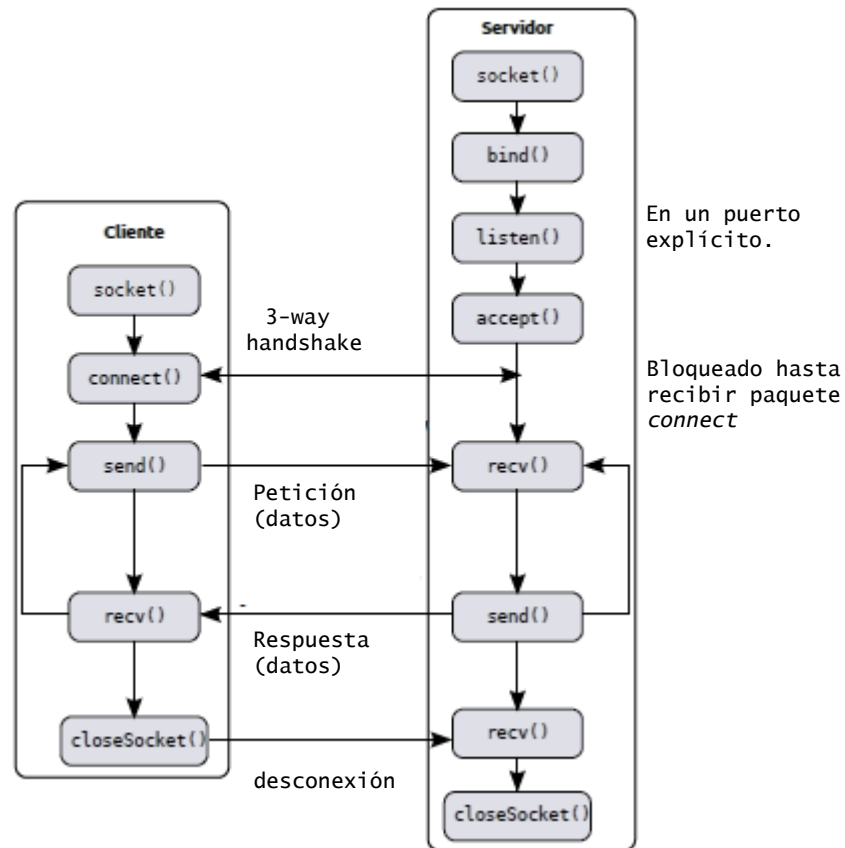
Datagram Socket (UDP)



IPC: Sockets

Stream Socket (TCP)

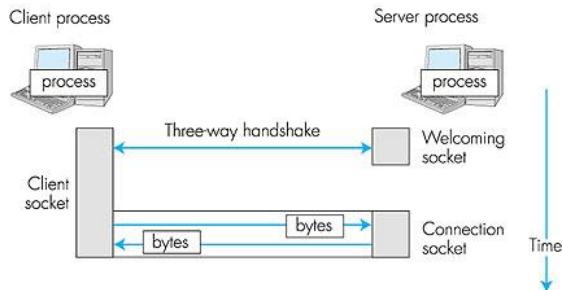
La diferencia con los *datagrams sockets* es que en los primeros no hay concepto de conexión. El cliente hace una petición y el servidor contesta pero no hay garantía de entrega. TCP es un protocolo orientado a conexión que tiene que establecerse y cerrarse explícitamente. Mientras dura, garantiza los reenvíos si hay fallo y la recepción en el mismo orden que se enviaron. Una conexión TCP está siempre asociada a lo que se denomina un puerto.



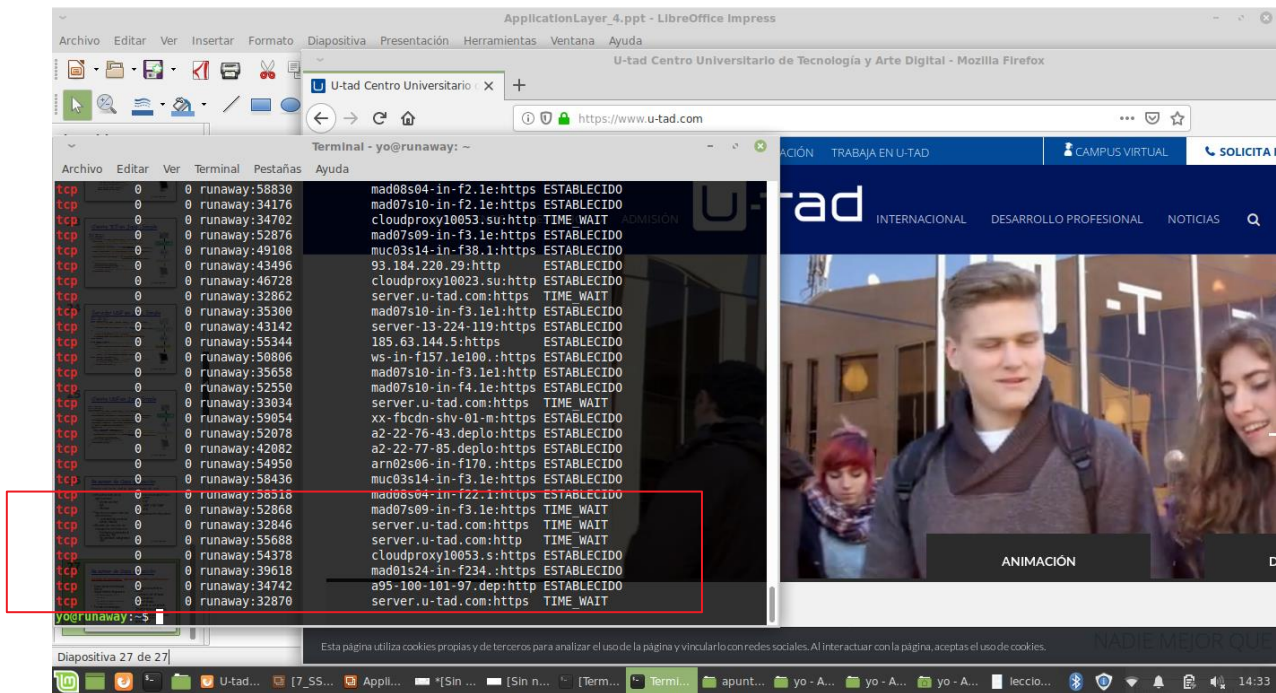
IPC: Sockets

La comunicación vía sockets sigue el modelo cliente servidor y funciona entre dos máquinas cualesquiera siempre que haya conectividad IP. El proceso servidor debe ejecutarse primero, y crear un socket en el puerto correspondiente. Por ejemplo, el puerto 80 es el estándar para recibir peticiones HTTP y es el que abre el demonio httpd si no se indica lo contrario en su configuración. El servidor queda bloqueado a la espera de recibir conexiones.

El cliente crea un socket en su máquina local y se dirige al servidor usando la dirección IP y el puerto de este con el que quiere conectar. Una vez que el servidor recibe la petición de conexión, hace un `fork()`, crea un socket local y es el que se utilizará para comunicarse con el cliente. Este tipo de solución permite atender múltiples clientes. El proceso servidor que está a la escucha solo ejecuta el `fork()` y vuelve de inmediato a esperar nuevas peticiones, las peticiones de servicio las atienden los hijos.



IPC: Sockets



Con netstat podemos ver los puertos abiertos de nuestro equipo (cliente) y los de la máquina servidora (u-tad.com)

Concurrencia

La concurrencia se produce cuando se pide el uso de un recurso limitado por parte de varios peticionarios. Es habitual a nivel hardware, cuando distintos dispositivos compiten por el uso de un bus o por una línea de interrupción. Existen protocolos para arbitrar esas situaciones.

En el software sucede lo mismo cuando dos procesos o threads tienen que escribir en un mismo puerto, socket, fichero, etc.

Los recursos pueden ser:

- **Físicos**, como un registro de un controlador, o **lógicos**, como un campo de una base de datos.
- **Reutilizables**, cuando la operación no los destruye (un fichero), o **consumibles** (el contenido de un mensaje que veremos en esta lección)
- **Exclusivos** (un temporizador) o **compartidos** (un fichero).
- **Expropiables** (una página de memoria RAM) o **no expropiables** (un mutex)

Concurrencia

Los procesos son **cooperantes** cuando están diseñados de forma premeditada para colaborar entre ellos o **independientes**, cuando la concurrencia es fortuita (varios procesos escribiendo en el mismo fichero de log).

La cooperación puede ser **implícita**, cuando la orquesta el sistema operativo de forma opaca al programador. Por ejemplo, cuando ordena los accesos a la RAM o a los puertos de E/S. La cooperación es explícita cuando sucede al nivel de procesos de usuario y se emplean los servicios de sincronización nativos del sistema operativo.

Se dice que una operación es **atómica** cuando para el resto del sistema concurrente ocurre de forma instantánea. Las únicas instrucciones atómicas son las de código máquina, que ya sabemos que se ejecutan de principio a fin en la CPU, incluso aunque se produzca una interrupción hardware. Un fragmento de código de dos líneas de ensamblador no es atómico al nivel de la CPU, puesto que puede llegar una interrupción y el SO podría expulsar al proceso de la CPU.

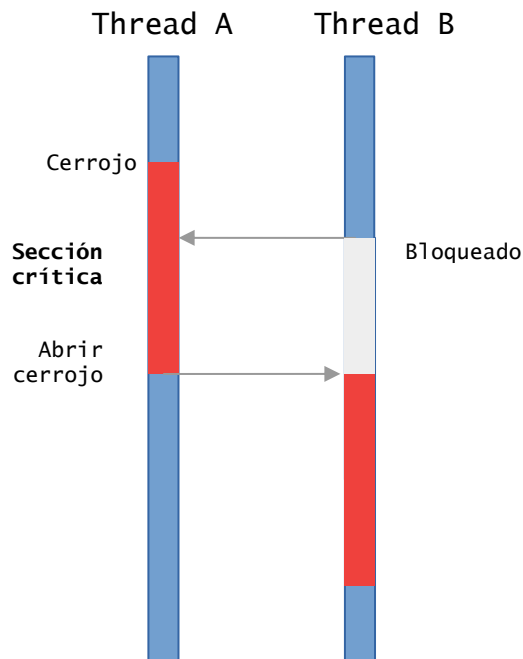
La atomicidad se usa en el kernel para construir los mecanismos de sincronización.

Concurrencia

En asuntos de concurrencia es más operativo hablar de **sección crítica**, un fragmento de código que se ejecuta de forma ininterrumpida desde el punto de vista de los procesos/threads concurrentes.

En este sencillo ejemplo hay dos threads de un mismo proceso, con una sección de código crítico. El primer hilo que llega a ese punto de ejecución pide el “cerrojo” del SO que protege dicha sección y se hace con él, empezando a ejecutarla.

El proceso B llega al punto de pedir el cerrojo mientras A está dentro de la sección crítica y el sistema operativo lo pasa a bloqueo. No saldrá de él mientras A no termine la sección crítica y libere el cerrojo. La petición y liberación del cerrojo son operaciones que incluye el usuario en el código, por lo que el diseño de aplicaciones concurrentes debe ser muy cuidadoso.



Concurrencia

La programación concurrente se enfrenta a dos problemas básicos, la condición de carrera y el *deadlock* (interbloqueo). La **condición de carrera** se produce cuando varios procesos o threads acceden a un mismo recurso (por ejemplo una memoria compartida) y el orden de llegada es puramente aleatorio, el que determinen la carga de la máquina y las decisiones del *scheduler* en cada momento. Imaginemos que tenemos un proceso con dos threads que acceden a la variable entera cuenta.

```
int cuenta; //Variable compartida
```

```
cuenta = 8;
```

```
// Thread A  
cuenta++;
```

```
// Thread B  
cuenta--;
```

```
LD A,(cuenta)  
ADD A,1  
LD (cuenta),A  
LD A,(cuenta)  
DEC A,1  
LD (cuenta),A
```

A se ejecuta de forma completa
antes que B (o al revés).
Resultado esperado, cuenta=8

```
LD A,(cuenta)  
ADD A,1  
LD A,(cuenta)  
DEC A,1  
LD (cuenta),A  
LD (cuenta),A
```

A es expulsado por el S0
después de la instrucción
ADD. El resultado global es
cuenta = 7, porque el efecto
del primer ADD
no se guarda en memoria

```
LD A,(cuenta)  
DEC A,1  
LD A,(cuenta)  
ADD A,1  
LD (cuenta),A  
LD (cuenta),A
```

B es expulsado por el S0
después de la instrucción
DEC. El resultado global es
cuenta = 9

Concurrencia

Otro ejemplo que pone en evidencia los conflictos que genera una condición de carrera. Romeo quiere invitar a Julieta al concierto de Taylor Swift como regalo de cumpleaños y está comprando las entradas por internet. Quedan solo dos, Romeo feliz, le envía un whatsapp a Julieta “Solo queda un par de entradas! Ahora mismo las compro”. Le da al botón de compra y se lanza la transacción contra su banco. Lo que no saben es que, a la vez, un alumno de INSO de U-tad, secreto admirador de la Swift, está también intentando conseguir una entrada. Mientras Romeo pierde el tiempo con su whatsapp, la transacción del alumno U-tad se procesa, el sitio web emite la entrada y el cómputo global de entradas queda en 1. Cuando el banco de Romeo autoriza la compra ya solo queda una entrada y Romeo y Julieta entran en crisis de pareja. Vale, sí, el servidor es muy malo porque no ha reservado las entradas antes de comprobar la solvencia de los compradores, el programador no previó esta condición de carrera y ha puesto en peligro una bonita historia de amor.



Concurrencia

El **interbloqueo** es la pesadilla de la programación concurrente y ocurre cuando un proceso retiene el recurso X y no lo libera a la espera de que otro proceso libere el recurso Y, que retiene pero no puede usar porque está a la espera del recurso X. Es una espera circular, en la que cada proceso “ni come ni deja comer”.

Imagina una carretera de doble sentido en obras. Un operario en cada extremo va dando paso de forma alternativa por el único carril. En un momento de despiste no se comunican bien y abren el tráfico a la vez. Las dos filas de coches, se encuentran de frente y frenan porque son buenos ciudadanos y respetan el límite de velocidad. El carril es estrecho, es imposible girar y hay un kilómetro de espera en cada sentido, no puede darse marcha atrás. Eso es un deadlock.



Atasco de tráfico en una glorieta, un deadlock en la vida real

Concurrencia

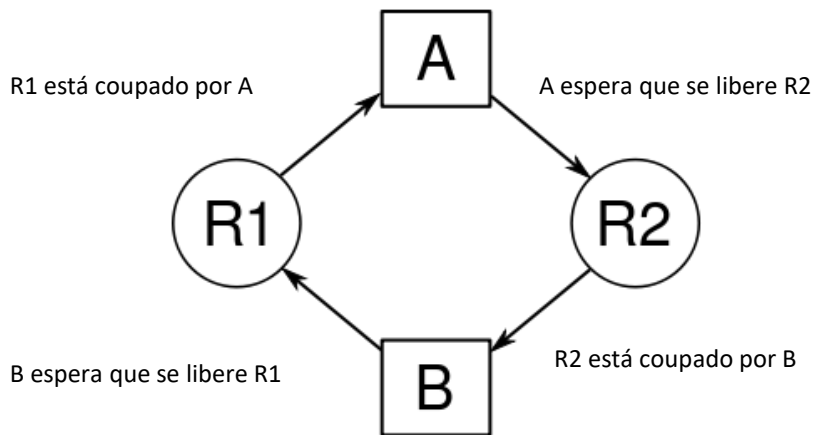
Para que ocurra un interbloqueo se tienen que dar las cuatro condiciones de Coffman, que caracterizó esta situación en un artículo de 1971.

- **Exclusión mutua.** Hay un recurso por el que contienden los procesos y solo puede usarlo uno de ellos. En el ejemplo, el recurso es el único carril en la zona de obras.
- **Condición de retención y espera.** Los procesos mantienen la posesión de los recursos ya asignados a ellos mientras esperan recursos adicionales. El carril es único pero tiene dos sentidos. El flujo en sentido este no puede convivir con el flujo en sentido oeste. Para que funcione bien la ocupación por uno de los flujos exige que el otro espere.
- **Condición de no expropiación.** Cuando un proceso ocupa el recurso no se le puede quitar. En el ejemplo, no podemos sacar los coches volando del carril. En la realidad, el Sistema Operativo evita todas las posibles condiciones de bloqueo en recursos compartidos como la memoria o la CPU porque tiene la posibilidad de expropiarlos. Los procesos de usuario no tienen esa capacidad.
- **Condición de espera circular.** En el ejemplo los dos sentidos de tráfico forman una condición de espera circular, puesto que el tráfico oeste tiene que esperar a que se libere el sentido este y viceversa.

Concurrencia

Interbloqueo

Diagrama con el interbloqueo más simple. El proceso A está esperando a que se libere el recurso R2 y tiene en propiedad el recurso R1. R2 es propiedad de B que espera a que se libere R1. En esto consiste la espera circular, esta situación se conoce en la literatura como *ABBA deadlock*.



ABBA[®]

Sincronización

El concepto de sincronización es sencillo de entender con la siguiente analogía:



En un establecimiento público hay un aseo en el que solo puede entrar un cliente. La puerta solo se puede abrir con una llave. El cliente que termina de usar el aseo sale, la puerta se cierra de forma automática, devuelve la llave al encargado y este la sortea entre los que están esperando. El aseo es el recurso compartido y la llave el mecanismo que lo regula. Con esta solución se evita que alguien abra la puerta con un cliente dentro (sección crítica) y contienda por el lavabo o el inodoro.

La llave funciona como un **mecanismo de exclusión mutua**.

Sincronización

La idea de la llave aparentemente es sencilla de realizar. Usamos una variable booleana como llave, el thread se bloquea con espera activa mientras la llave está ocupada. Cuando se libera, entra en la zona crítica, bloquea la llave, ejecuta la sección crítica y la libera al final. Es el paralelismo SW del protocolo del aseo.

```
int llave_libre;  
  
llave_libre = True;  
  
// Seccion de código del thread  
  
while (!llave_libre); // espera activa  
{  
    llave_libre = False;  
    seccion_critica();  
    llave_libre = True;  
}
```



Sincronización

En efecto, hay truco. Vamos a ignorar que la espera activa es una estrategia pésima para el rendimiento de la CPU, todos los procesos/threads en espera estarían en un bucle infinito. Aun así, no funciona. Si el proceso es expulsado en los puntos indicados, el estado queda incoherente.

```
int llave_libre;
```

```
llave_libre = True;
```

```
// Seccion de código del thread
```

```
while (!llave_libre); // espera activa
{
    llave_libre = False;
    seccion_critica();
    llave_libre = True;
}
```

Proceso expulsado, la llave sigue libre, cualquier otro thread que consiga CPU entra en la sección crítica

Proceso expulsado, la llave sigue ocupada e impide a otro thread ejecutarse

El problema es que la adquisición y liberación del candado no son operaciones atómicas.

Sincronización

Mutex

La solución que se ofrece a nivel de Sistema es el mutex, un mecanismo bloqueante que funciona de forma atómica utilizando instrucciones de ensamblador (como XCHG en x8086). El mutex garantiza que el thread que adquiere la propiedad ejecuta la zona crítica sin interrupción por otros hilos del mismo sistema concurrente. Cualquier otro hilo que intenta adquirir la propiedad del mutex ocupado es pasado a estado de bloqueo por el kernel y no saldrá de él hasta que el propietario libere el mutex.

```
//Pthreads Mutex Example
```

```
THREAD 1
```

```
pthread_mutex_lock (&mut);
```

```
a = data;
```

```
a++;
```

```
data = a;
```

```
pthread_mutex_unlock (&mut);
```

```
THREAD 2
```

```
pthread_mutex_lock (&mut);
```

```
/* blocked */
```

```
/* blocked */
```

```
/* blocked */
```

```
/* blocked */
```

```
b = data;
```

```
data = b;
```

```
b--;
```

```
pthread_mutex_unlock (&mut);
```

Ejemplo mínimo con mutex

Sincronización

Spin lock

El candado o cerrojo propiamente dicho en Unix (spin lock) es un mecanismo de bajísimo nivel del kernel que inhibe el *scheduler* y parte las interrupciones y permite la ejecución atómica casi pura. Tiene mucho riesgo si no se maneja con cuidado, puesto que un thread que intenta capturar un spin lock quedará en espera activa si está ocupado, y si la tarea que lo ocupó no lo libera, el sistema no reaunda el *scheduler* correctamente.

Los candados son recomendables para secciones críticas muy breves, como el incremento de un contador o recorrer en lectura un vector. Se usan de forma habitual en la programación del kernel. Resultan desaconsejables cuando la tarea va a llevar un tiempo impredecible, por ejemplo expulsar una página de RAM a swap o leer datos del adaptador de red.

Como norma general, si no eres un programador muy experto, usa mejor mutex.

Sincronización

Candado en xv6

```


// Acquire the lock.
// Loops (spins) until the lock is acquired.
// Holding a lock for a long time may cause
// other CPUs to waste time spinning to acquire it.
void
acquire(struct spinlock *lk)
{
    pushcli(); // disable interrupts to avoid deadlock.
    if(holding(lk))
        panic("acquire");


    // The xchg is atomic.
    while(xchg(&lk->locked, 1) != 0)


    // Tell the C compiler and the processor to not move loads or stores
    // past this point, to ensure that the critical section's memory
    // references happen after the lock is acquired.
    __sync_synchronize();

    // Record info about lock acquisition for debugging.
    lk->cpu = mycpu();
    getcallerpcs(&lk, lk->pcs);
}

```

 Enmascarar todas las interrupciones !!!! DANGER !!!!

 Si el proceso ya tenía el candado algo muy grave ha ocurrido, panic

 Instrucción atómica que permite poner a 1 el candado (cerrado)

Sincronización

Candado en xv6

```
// Release the lock.
void
release(struct spinlock *lk)
{
    if(!holding(lk))
        panic("release");
```

Si el proceso ya tenía el candado algo muy grave ha ocurrido, panic

```
lk->pcs[0] = 0;
lk->cpu = 0;
```

```
// Tell the C compiler and the processor to not move loads or stores
// past this point, to ensure that all the stores in the critical
// section are visible to other cores before the lock is released.
// Both the C compiler and the hardware may re-order loads and
// stores; __sync_synchronize() tells them both not to.
__sync_synchronize();
```

```
// Release the lock, equivalent to lk->locked = 0.
// This code can't use a C assignment, since it might
// not be atomic. A real OS would use C atomics here.
```

```
asm volatile("movl $0, %0" : "+m" (lk->locked) : );
```

Poner a 0 el candado (cerrado). No muy atómica

```
popcli();
```


Desenmascarar todas las interrupciones

```
}
```

Sincronización

Candado en xv6

Ejemplo de uso de candado en xv6, en la función scheduler (fichero proc.c). El scheduler adquiere el candado antes seleccionar el proceso y lo libera al final. La función es sencilla en extremo.



¿Pero qué scheduling policy es esta?
¿El primero que pillá?

```
void
scheduler(void)
{
    struct proc *p;
    struct cpu *c = mycpu();
    c->proc = 0;

    for(;;){
        // Enable interrupts on this processor.
        sti();

        // Loop over process table looking for process to run.
        acquire(&ptable.lock);
        for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
            if(p->state != RUNNABLE)
                continue;

            // Switch to chosen process. It is the process's job
            // to release ptable.lock and then reacquire it
            // before jumping back to us.
            c->proc = p;
            switchvm(p);
            p->state = RUNNING;

            switch(&(c->scheduler), p->context);
            switchkvm();

            // Process is done running for now.
            // It should have changed its p->state before coming back.
            c->proc = 0;
        }
        release(&ptable.lock);
    }
}
```

Sincronización

Mutex

El uso de mutex es muy eficaz, pero también puede presentar problemas. Uno de ellos es la **estampida** (thundering herd). Si hay muchos threads bloqueados, la liberación del mutex lleva a todos al estado de Listo para ejecución, el scheduler elegirá a uno, que adquirirá el mutex, y después irán entrando todos los demás para quedar de nuevo bloqueados, pagando el precio de múltiple cambios de contexto improductivos. La solución pasa por una cola en la que el kernel mantiene los threads bloqueados por un mismo mutex y solo despierte al que lleve más tiempo en espera.

El otro problema, que es especialmente grave en entornos de tiempo real, es la **inversión de prioridad**. Supongamos que hay dos procesos diferentes usando el mismo mutex porque comparten cierto dato. Uno de ellos tiene prioridad alta y el otro baja. Cuando este segundo se apropia del mutex, el proceso de prioridad alta queda bloqueado por uno de baja prioridad, lo que va en contra de cualquier algoritmo de *scheduling*. La solución en este caso es incrementar la prioridad de la tarea propietaria al mismo nivel que la de la tarea de mayor prioridad bloqueada por ese mutex. Cuando la de baja prioridad finaliza la sección crítica y libera el mutex vuelve a su prioridad original.

Sincronización

Semáforos

Los semáforos son un mecanismo definido por Edsger Dijkstra, inspirado en la señalización ferroviaria. A diferencia del mutex, que como indica su nombre, es de exclusión mutua, el semáforo sirve para arbitrar el acceso de M peticionarios a N recursos de forma simultánea, donde $M > N$.

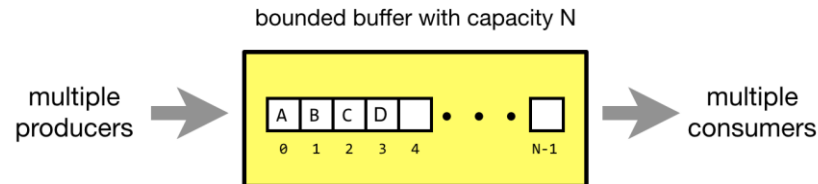
Volviendo a la analogía, en el establecimiento hay dos o más aseos (N), cada uno con su llave. Si llega un nuevo usuario podrá acceder siempre que el número de llaves ocupadas sea $< N$, de lo contrario espera a que cualquiera se desocupe.



Sincronización

Semáforos

Problema del productor-consumidor en su fórmula más sencilla, con una única cola. Podemos imaginar que los productores son threads que reciben tramas de varios canales de comunicación de datos y los consumidores se encargan de reencaminarlos en función de la dirección de destino. Es una situación habitual. Los productores van escribiendo paquetes de tamaño fijo, en una cola circular de N elementos. Mientras no termina la escritura no se puede leer, pero podría llegar otro paquete y un segundo productor escribirlo en otro bloque. Los consumidores van extrayendo los paquetes completos, pero tienen que hacerlo a una velocidad tal que no haya nunca más de N paquetes en la cola. Si no se controla este extremo, un nuevo productor sobrescribiría un paquete no leído y esa información se perdería. En caso de llegar a N , los productores entran en contención. Para mejorar el rendimiento se pueden añadir consumidores, aumentar la longitud del buffer o poner más colas en paralelo.



Sincronización

Semáforos

El semáforo se implementa como un contador entero (número de llaves) que puede variar entre 0 y N. Cuando un proceso ocupa un recurso, el valor del contador se incrementa, cuando lo desocupa, se resta. La operación de ocupar se denomina wait y la de liberar signal, esto hace que en numerosos textos se indique se los semáforos funcionan con operaciones wait/signal y eso es muy confuso porque las operaciones wait y signal de este objeto no tienen ninguna relación con las funciones wait() y signal() de Unix. Es preferible referirse a ellas como semaphore_wait() y semaphore_signal().

El máximo valor que puede alcanzar el contador del semáforo se define en su creación. Un semáforo binario solo puede tomar los valores 0 y 1, en cuyo caso es equivalente en operación a un mutex. En general, un semáforo no es un mecanismo de exclusión mutua (lo que garantiza la ejecución de la sección crítica) sino una respuesta al problema productor-consumidor. K productores colocan el recurso que sea en una cola y M consumidores van extrayendo y procesando. Si la cola está vacía los consumidores tienen que esperar a que los productores generen nuevos recursos.

Sincronización

Semáforos

De nuevo, la implementación de los semáforos parece trivial.

```
int contador_semaforo = 0;

void semaphore_wait(int *cs){
    while (*cs <= 0);           //espera activa
    *cs = *cs - 1;
}

void semaphore_signal(int *cs){
    *cs = *cs + 1;
}
```

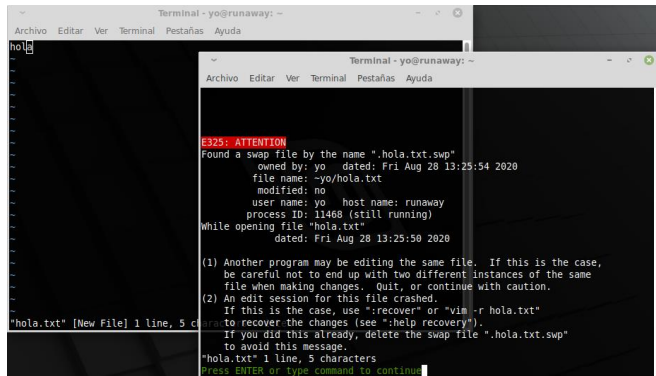
La realidad ya sabemos que es más complicada. Las operaciones de suma y resta del contenido de `contador_semaforo` **no son atómicas**, se puede conseguir protegiendo ambas operaciones como secciones críticas con un mutex.

```
void semaphore_signal(int *cs){
    pthread_mutex_lock(&num);
    *cs = *cs + 1;
    pthread_mutex_unlock(&num);
}
```

Sincronización

Candados sobre ficheros

Todos los sistemas operativos incorporan un mecanismo de protección de ficheros. Es una situación muy habitual, se intenta borrar una carpeta en Windows y el sistema lo impide porque hay una aplicación usándolo (un documento Word o PDF, un fuente que se está editando). Esto se consigue usan candados (locks), que conceden acceso exclusivo en escritura/borrado al proceso que se apropia de ellos. El resto de procesos autorizados pueden seguir leyendo pero no están autorizados a realizar modificaciones. En las primeras versiones de Unix los candados de ficheros eran consultivos y se confiaba en la pericia de los programadores para no modificar un fichero compartido sin consultar antes si existía un candado.



```
Terminal - yo@runaway: ~
hola

C325: ATTENTION
Found a swap file by the name ".hola.txt.swp"
  owned by: yo   dated: Fri Aug 28 13:25:54 2020
  file name: ~yo/hola.txt
  modified: no
  user name: yo   host name: runaway
  process ID: 11468 (still running)
While opening file ".hola.txt"
  dated: Fri Aug 28 13:25:50 2020

(1) Another program may be editing the same file. If this is the case,
    be careful not to end up with two different instances of the same
    file when making changes. Quit, or continue with caution.
(2) An edit session for this file crashed.
    If this is the case, use ":recover" or "vim -r hola.txt"
    to recover the changes (see ":help recovery").
    If you did this already, delete the swap file ".hola.txt.swp"
    to avoid this message.
".hola.txt" 1 line, 5 characters
Press ENTER or type command to continue
```