



Sistemas Operativos Concurrencia



Existen DOS cosas muy difíciles de hacer en Ciencias de la Computación:

1. Poner nombres a las cosas.
2. La Concurrency.
3. Errar por uno.



Concurrencia



Concurrencia

Un detalle muy importante que hay que notar es que el individuo no está realizando todo en forma paralela, es decir al mismo tiempo:

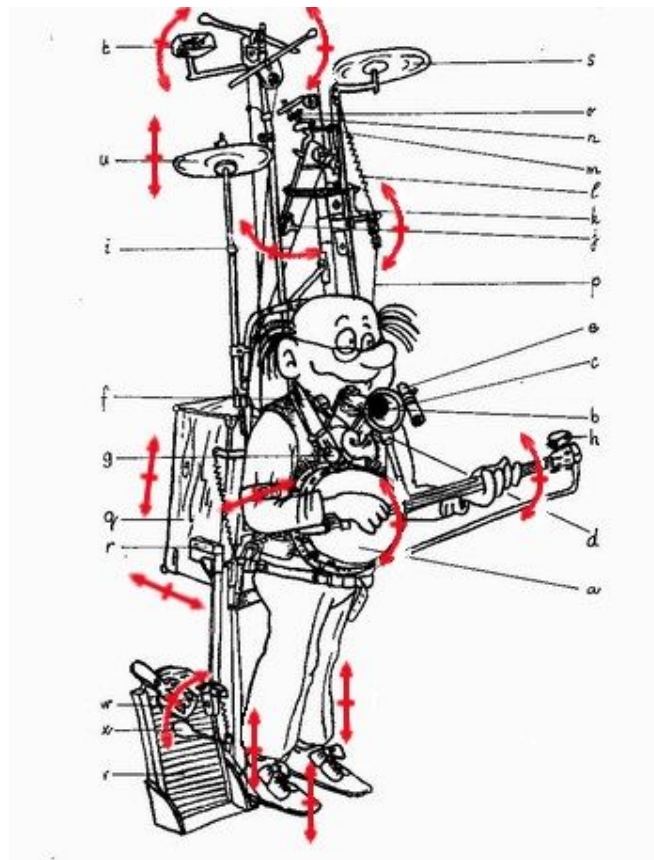
lee,

toma,

charla,

Mira

Todo en el mismo tiempo infinitesimal , sino que las operaciones se están llevando a cabo a la vez.



paralelismo



La Concurrency

El mundo de la concurrency se refiere a un conjunto de actividades que pueden suceder al mismo tiempo. Anderson-Dahlin pag. 129.

El correcto manejo de la concurrency es una de las claves en el desarrollo de los Sistemas Operativos Modernos. En este tema se verá cuál es la abstracción que maneja y disminuye la complejidad del problema de la Concurrency.



La Concurrency

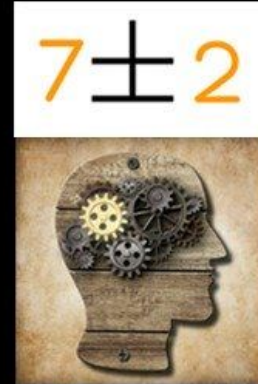
Uno de los aspectos más interesantes al enfrentarse con la Concurrency, es la manera en que nuestra mente es capaz de atacar un problema:

- En el caso de la mayoría de los programadores la forma más común de construir programas es la llamada forma secuencial: ejecutar una acción detrás de la otra. Esto es lo que la mayoría de los programadores realizan cotidianamente.
- Pensar en atacar un problema en el cual decenas de eventos pueden desencadenarse al mismo tiempo es aún muy complejo

La Concurrency

Además, un problema de los seres humanos se encuentra en nuestra limitación para manejar distinta información al mismo tiempo Miller GA Magical Seven Psych Review 1955 según este artículo de Miller de 1955 los seres humanos podemos manejar 7 ± 2 chunks of data .

The Magical Number Seven,
Plus or Minus Two



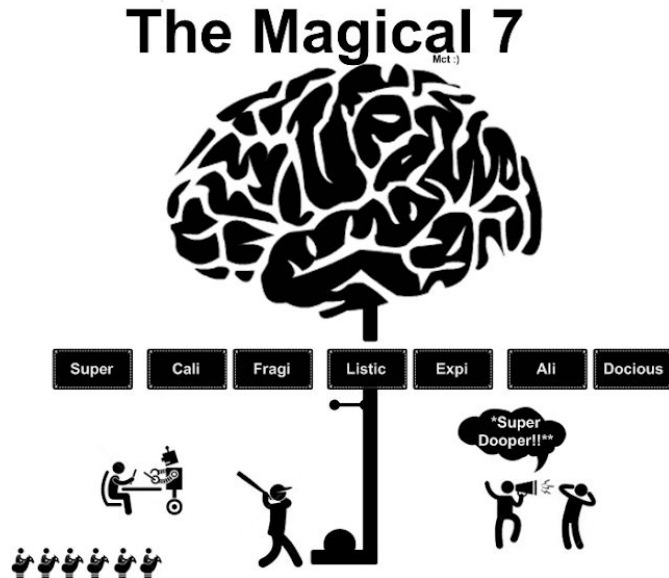
George A. Miller



La Concurrency

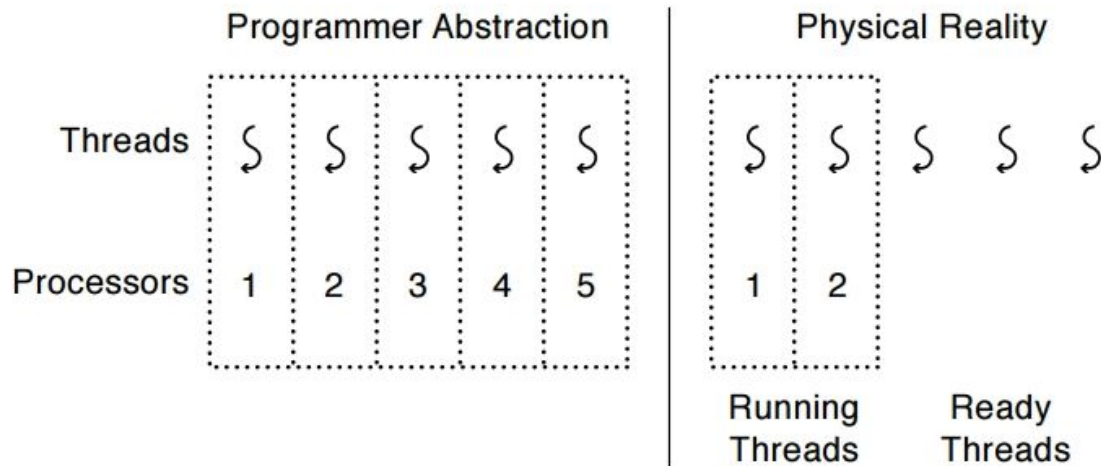
And finally, what about the magical number **seven**? What about the **seven wonders of the world**, the **seven seas**, the **seven deadly sins**, the **seven daughters of Atlas in the Pleiades**, the **seven ages of man**, the **seven levels of hell**, the **seven primary colors**, the **seven notes of the musical scale**, and the **seven days of the week**? What about the **seven-point rating scale**, the **seven categories for absolute judgment**, the **seven objects in the span of attention**, and the **seven digits in the span of immediate memory**? For the present I propose to withhold judgment. Perhaps there is something deep and profound behind all these sevens, something just calling out for us to discover it. But I suspect that it is only a pernicious, Pythagorean coincidence.

The Magical 7



A qué apuntamos

El concepto clave es escribir un programa concurrente como una secuencia de streams de ejecución o threads que interactúan y comparten datos en una manera muy precisa. El concepto básico es el siguiente:





La Abstraccion

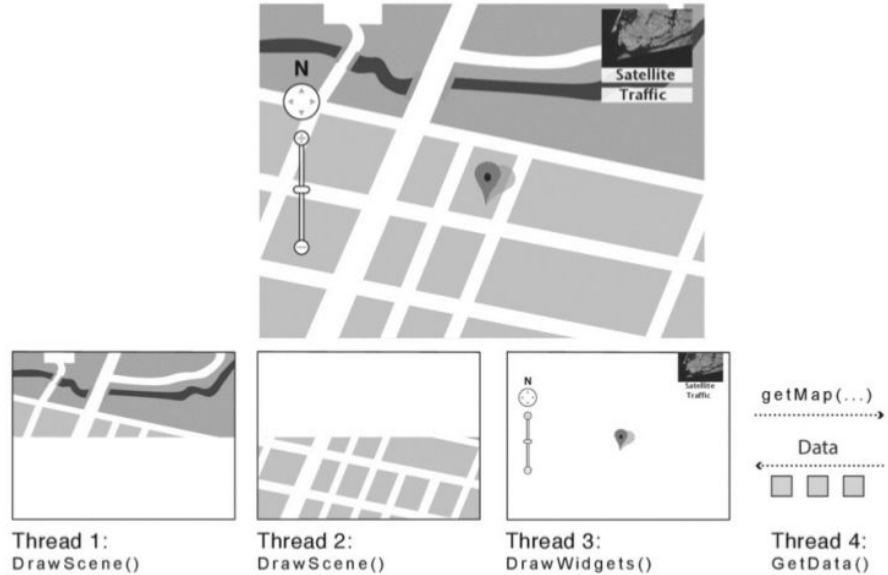
Un thread es una secuencia de ejecución atómica que representa una tarea planificable de ejecución

- Secuencia de ejecución atómica: Cada thread ejecuta una secuencia de instrucciones como lo hace un bloque de código en el modelo de programación secuencial.
- tarea planificable de ejecución: El sistema operativo tiene injerencia sobre el mismo en cualquier momento y puede ejecutarlo, suspenderlo y continuarlo cuando él desee.

La Abstraccion

Un thread es una secuencia de ejecuc

- Secuencia de ejecución atómica: un bloque de código en el modo
- tarea planificable de ejecución: momento y puede ejecutarlo, s



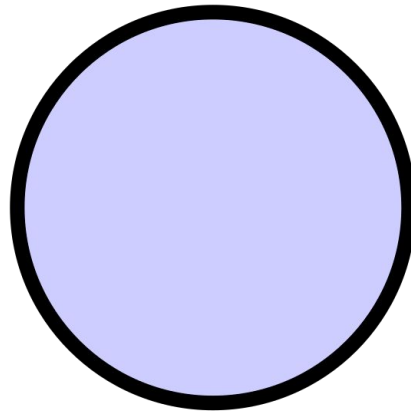


Threads vs procesos

Proceso: un programa en ejecución con derechos restringidos.

Thread: una secuencia independiente de instrucciones ejecutándose dentro de un programa.

Process



Thread





Threads

Esta abstracción, el thread, se caracteriza por :

- Thread id
- un conjunto los valores de registros
- stack propio
- una política y prioridad de ejecución
- un propio errno
- datos específicos del thread

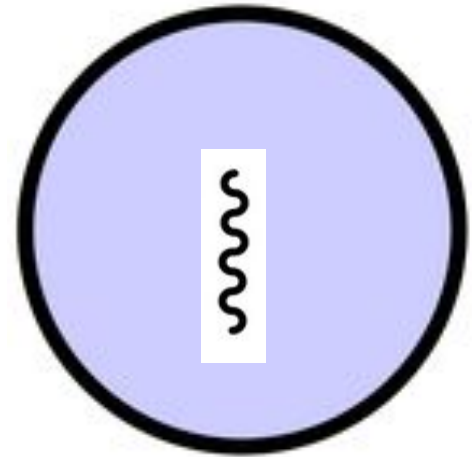
Thread



Threads

One thread per process: un proceso con una única secuencia de instrucciones ejecutándose de inicio a fin. Para los que vieron Pascal o C sería el equivalente a un bloque de instrucciones delimitado por begin-end o { }. Lo que todos los programadores de modelo secuencial conocemos.

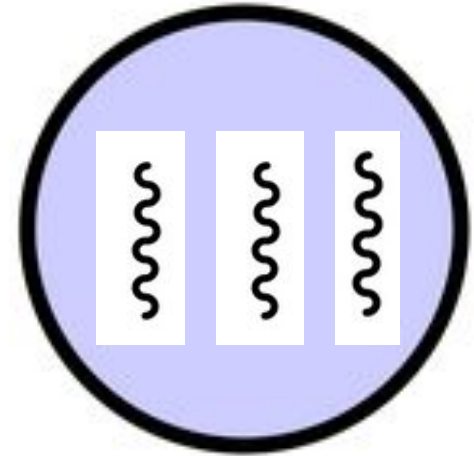
Process

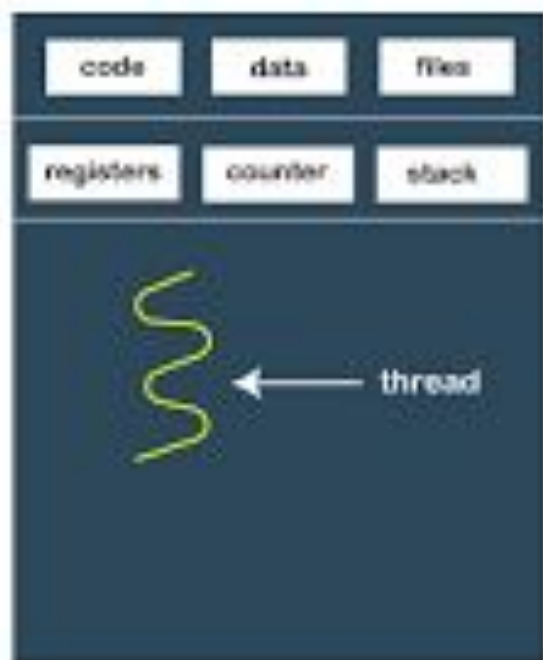


Threads

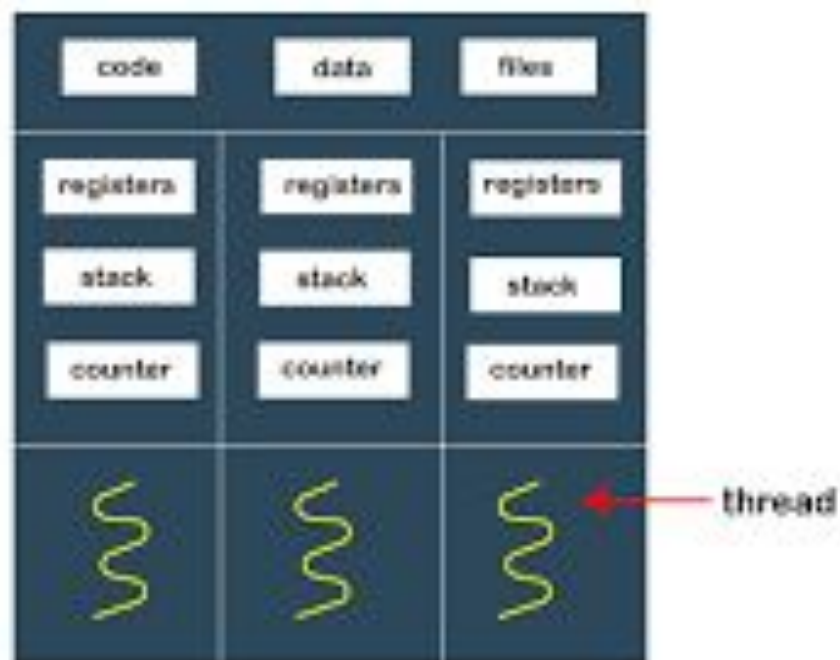
- Many thread per process: un programa es visto como threads ejecutándose dentro de un proceso con derechos restringidos. En dado un ti algunos threads pueden estar corriendo y otros estar suspendidos. Cuando se detecta por ejemplo una operación de I/O por alguna interrupción, el kernel desaloja (preempt) a algunos de los threads que están corriendo, atiende la interrupción, y al terminar de manejar la interrupción vuelve a correr el thread nuevamente.

Process





Single-threaded process



Multi-threaded process



Thread Scheduler

¿Cómo hace el S.O. para crear la ilusión de muchos threads con un número fijo de procesadores?

Obviamente es necesario un planificador de thread o threads scheduler, ya que el S.O. podría estar trabajando con un único procesador. El cambio entre threads es transparente, es decir que el programador debe preocuparse de la secuencia de instrucciones y no el cuando éste debe ser suspendido o no.

Por ende los Threads **proveen un modelo de ejecución en el cual cada thread corre en un procesador virtual dedicado (exclusivo) con una velocidad variable e impredecible** Anderson-Dahlin, pag 138.



Thread Scheduler

Esto quiere decir que desde el punto de vista del thread cada instrucción se ejecuta inmediatamente una detrás de otra. Pero el que decide cuando se ejecuta es el planificador de threads o thread scheduler. Por ejemplo:

```
...  
...  
x = x + 1;  
y = x + y;  
z = x + 5y;  
...  
...
```



Thread Scheduler

Entonces
base a
antedicho,
pueden
encontrar
siguientes
escenarios
ejecución:

Programmer's View

.
.
.
x = x + 1;
y = y + x;
z = x + 5y;
.
.
.

Possible Execution #1

.
.
.
x = x + 1;
y = y + x;
z = x + 5y;
.
.
.

Possible Execution #2

.
.
.
x = x + 1;
.....
Thread is suspended.
Other thread(s) run.
Thread is resumed.
.....
y = y + x;
z = x + 5y;

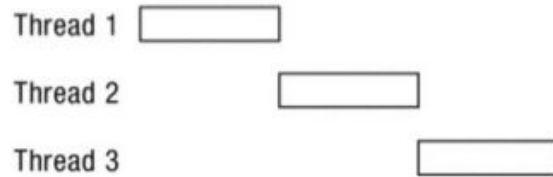
Possible Execution #3

.
.
.
x = x + 1;
y = y + x;
.....
Thread is suspended.
Other thread(s) run.
Thread is resumed.
.....
z = x + 5y;

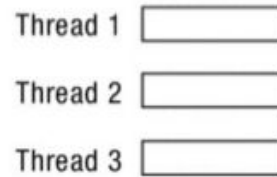
Thread Scheduler

Y los
siguientes
interleaves o
entrelazado
pueden
suceder, con
estos distintos
threads

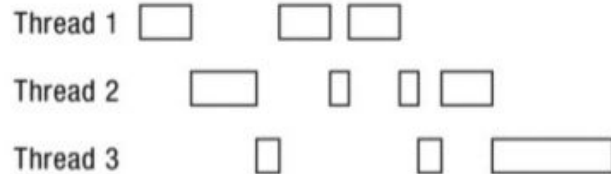
One Execution



Another Execution



Another Execution





Thread Scheduler

En la actualidad hay dos formas de que los threads se relacionan entre sí:

- Multi-threading Cooperativo: no hay interrupción a menos que se solicite.
→ ¿Problemas?
- Multi-threading Preemptivo: Es el más usado en la actualidad. Consiste en que un threads en estado de running puede ser movido en cualquier momento.

El API de Threads



Creacion

thread: Es un puntero a la estructura de tipo pthread_t, que se utiliza para interactuar con el threads.

attr: Se utiliza para especificar los ciertos atributos que el thread deberia tener, por ejemplo, el tamaño del stack, o la prioridad de scheduling del thread. En la mayoría de los casos es NULL.

start_routine: Sea tal vez el argumento más complejo, pero no es más que un puntero a una función, en este caso que devuelve void.

arg: Es un puntero a void que debe apuntar a los argumentos de la función.

```
#include<pthread.h>
int pthread_create(pthread_t * thread, const pthread_attr_t * attr,
                  void * (start_routine) (void *), void * arg)
```




Terminación de un thread

Muchas veces es necesario esperar a que un determinado thread finalice su ejecución, para ello se utiliza la función `pthread_join()`, que toma dos argumentos

1. `thread` es el thread por el que hay que esperar y es de tipo `pthread_t`.
2. `value_ptr` es el puntero al valor esperado de retorno.

```
int pthread_join(pthread_t thread, void **value_ptr )
```



```
#include <pthread.h>
#include <stdio.h>

void *mythread(void *arg) {

    printf("%s\n", (char *) arg);
    return NULL;
}

int main(int argc, char *argv[]) {
    pthread_t p1, p2;
    int rc;

    printf("inicia main() \n");
    rc=pthread_create(&p1, NULL, mythread, "A");
    rc=pthread_create(&p2, NULL, mythread, "B");

    rc=pthread_join(p1, NULL);
    rc=pthread_join(p2, NULL);

    printf("termina main() \n");
    return 0;
}
```



Threads: Estructura

Como se ha visto, cada thread es la representación de una secuencia de ejecución de un conjunto de instrucciones. El S.O. provee la ilusión de que cada uno de estos threads se ejecutan en su propio procesador, haciendo de forma transparente que se ejecuten o paren su ejecución.

Para que la ilusión sea creíble, el sistema operativo debe guardar y cargar el estado de cada thread. Como cualquier thread puede correr en el procesador o en el kernel, también debe haber estados compartidos, que no deberían cambiar entre los modos.

Para poder entender la abstracción hay que comprender que existen dos estados:

- El estado per thread.

- El estado compartido entre varios threads.

El Estado Per-thread y Threads Control Block (TCB)



Cada thread debe tener una estructura que represente su estado. Esta estructura se denomina Thread Control Block (TCB), se crea una entrada por cada thread. La TCB almacena el estado per-thread de un thread:

El estado del Cómputo que debe ser realizado por el thread.

Para poder crear múltiples threads y pararlos y arrancarlos, el S.O. debe poder almacenar en la TCB el estado actual del bloque de ejecución:

El puntero al stack del thread.

Una copia de sus registros en el procesador.



Metadata del thread

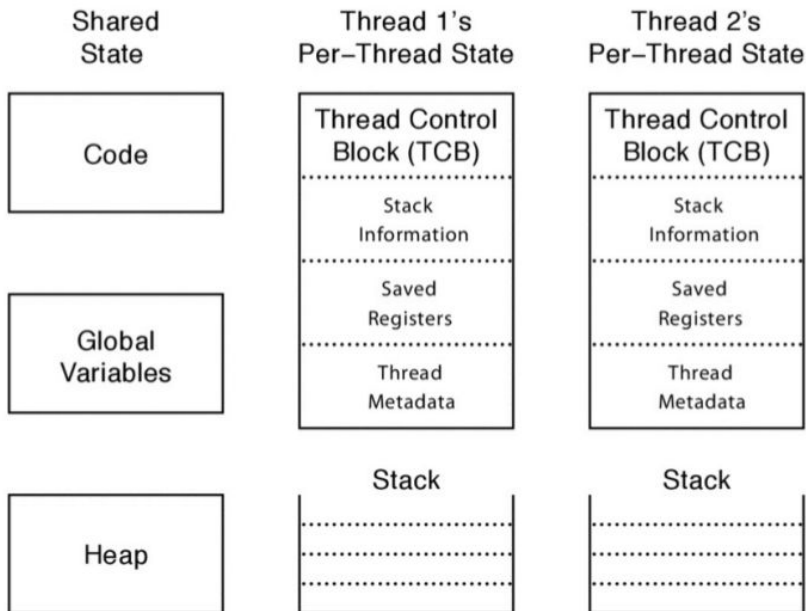
Por cada thread se debe guardar determinada información sobre el mismo:

- ID
- Prioridad de scheduling
- Status

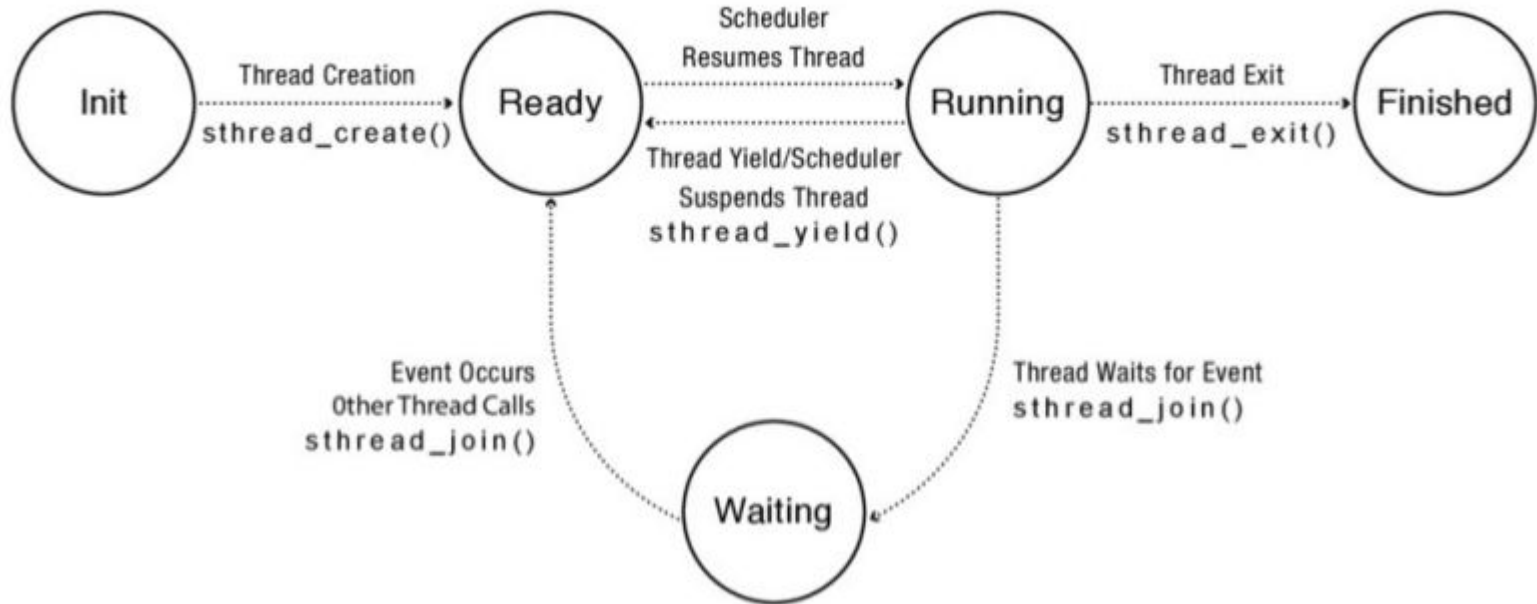
Metadata del thread

De forma contraria al per-thread state se debe guardar cierta información que es compartida por varios Threads:

- El Código
- Variables Globales
- Variables del Heap



Estados





Estados

Init: Un thread se encuentra en estado INIT mientras se está inicializando el estado per-thread y se está reservando el espacio de memoria necesario para estas estructuras. Una vez que esto se ha realizado el estado del thread se setea en READY. Además se lo pone en una lista llamada ready list en la cual están esperando todos los thread listos para ser ejecutados en el procesador.

Ready: Un thread en este estado está listo para ser ejecutado pero no está siendo ejecutado en ese instante. La TCB está en la ready list y los valores de los registros están en la TCB. En cualquier momento el thread scheduler puede transicionar al estado RUNNING.



Estados

Running: Un thread en este estado está siendo ejecutado en este mismo instante por el procesador. En este mismo instante los valores de los registros están en el procesador. En este estado un RUNNING THREAD puede pasar a READY de dos formas:

- El scheduler puede pasar un thread de su estado RUNNING a READY mediante el desalojo o preemption del mismo mediante el guardado de los valores de los registros y cambiando el thread que se está ejecutando por el próximo de la lista.
- Voluntariamente un thread puede solicitar abandonar la ejecución mediante la utilización de `thread_yield`, por ejemplo.



Estados

Waiting: En este estado el Thread está esperando que algún determinado evento suceda. Dado que un thread en WAITING no puede pasar a RUNNING directamente, estos thread se almacenan en la lista llamada waiting list. Una vez que el evento ocurre el scheduler se encarga de pasar el thread del estado WAITING a RUNNING, moviendo la TCB desde el waiting list a la ready list.

Finished: Un thread que se encuentra en estado FINISHED nunca más podrá volver a ser ejecutado. Existe una lista llamada finished list en la que se encuentran las TCB de los threads que han terminado.



Tabla de equivalencia entre procesos y threads:

Process primitive	Thread primitive	Description
fork	pthread_create	crea un nuevo flujo de control
exit	pthread_exit	sale de un flujo de control existente
waitpid	pthread_join	obtiene el estado de salida de un flujo de control
atexit	pthread_cleanup	función a ser llamada en el momento de salida de un flujo de control
getpid	pthread_self	obtiene el id de un determinado flujo de control
abort	pthread_cancel	terminación anormal de un flujo de control



Threads y Linux Diferencias Proceso/Thread

Los threads

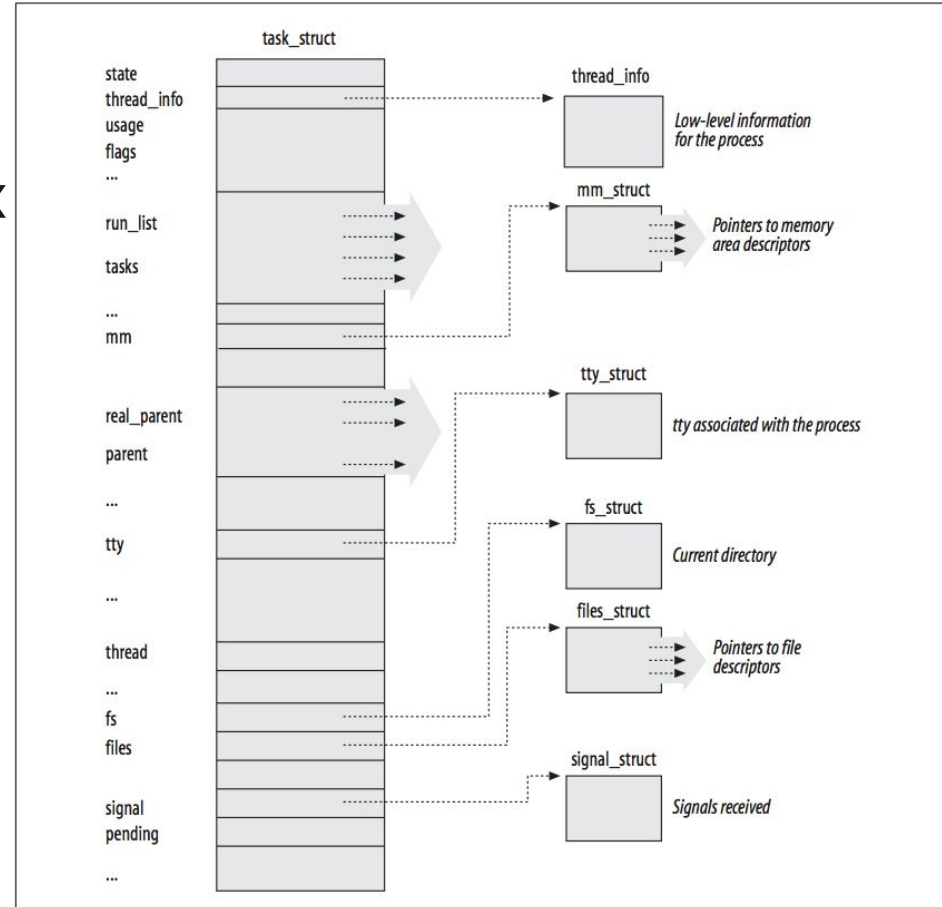
- Por defecto comparten memoria
- Por defecto comparten los descriptores de archivos
- Por defecto comparten el contexto del filesystem
- Por defecto comparten el manejo de señales

Los Procesos

- Por defecto no comparten memoria
- Por defecto no comparten los descriptores de archivos
- Por defecto no comparten el contexto del filesystem
- Por defecto no comparten el manejo de señales

Modelo de thread en linux

Linux utiliza un modelo 1-1 (proceso-thread), con lo cual dentro del kernel no existe distinción alguna entre thread y proceso – todo es un tarea ejecutable.





Creación de Threads en Linux

Creacion de Threads (LinuxThreads)

```
clone(CLONE_VM | CLONE_FS | CLONE_FILES | CLONE_SIGHAND, 0);
```

fork()

```
clone(SIGCHLD, 0);
```

vfork()

```
clone(CLONE_VFORK | CLONE_VM | SIGCHLD, 0);
```

Memoria en un proceso multithread

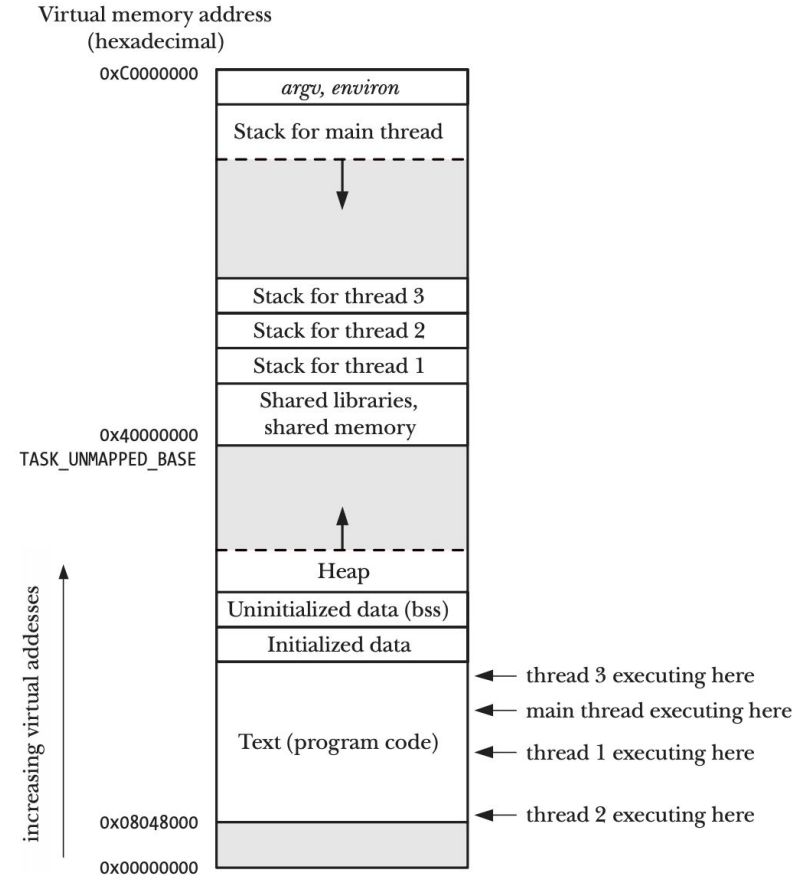


Figure 29-1: Four threads executing in a process (Linux/x86-32)

Velocidad de creación de threads

Table 28-3: Time required to create 100,000 processes using *fork()*, *vfork()*, and *clone()*

Method of process creation	Total Virtual Memory					
	1.70 MB		2.70 MB		11.70 MB	
	Time (secs)	Rate	Time (secs)	Rate	Time (secs)	Rate
<i>fork()</i>	22.27 (7.99)	4544	26.38 (8.98)	4135	126.93 (52.55)	1276
<i>vfork()</i>	3.52 (2.49)	28955	3.55 (2.50)	28621	3.53 (2.51)	28810
<i>clone()</i>	2.97 (2.14)	34333	2.98 (2.13)	34217	2.93 (2.10)	34688
<i>fork()</i> + <i>exec()</i>	135.72 (12.39)	764	146.15 (16.69)	719	260.34 (61.86)	435
<i>vfork()</i> + <i>exec()</i>	107.36 (6.27)	969	107.81 (6.35)	964	107.97 (6.38)	960

Thread Groups: CLONE_THREAD

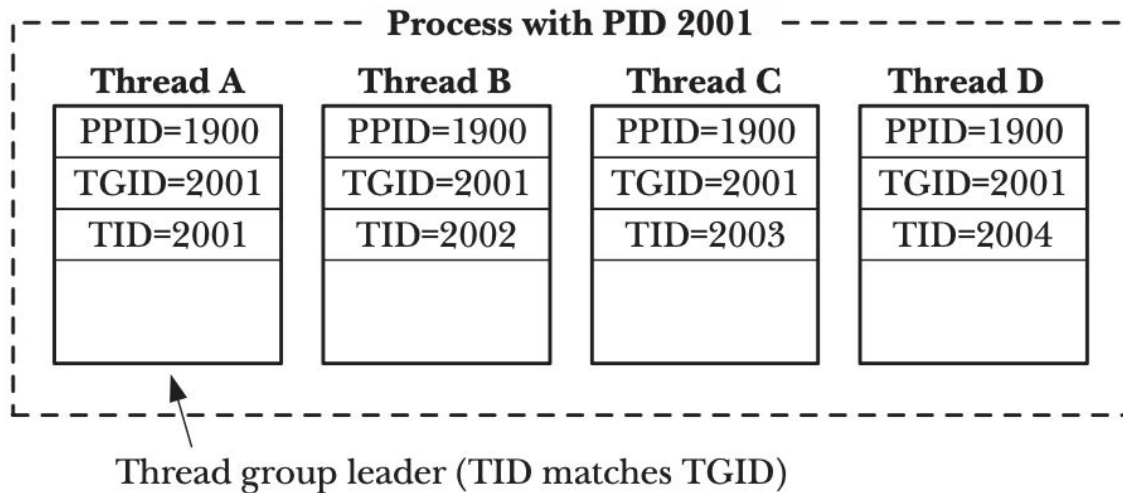


Figure 28-1: A thread group containing four threads



Native POSIX Threads Library (NPTL)

```
clone(CLONE_VM | CLONE_FILES | CLONE_FS | CLONE_SIGHAND |  
CLONE_THREAD | CLONE_SETTLS | CLONE_PARENT_SETTID |  
CLONE_CHILD_CLEARPID | CLONE_SYSVSEM, 0)
```

Se puede comprobar si se está usando en el sistema ejecutando:

```
$ /lib/libc.so.6
```



Viendo los Threads: ps m

```
$ ps m
  PID TTY          STAT TIME  COMMAND
 3587 pts/3        -    0:00 bash❶
    -  -           Ss    0:00 -
 3592 pts/4        -    0:00 bash❷
    -  -           Ss    0:00 -
12534 tty7          - 668:30 /usr/lib/xorg/Xorg -core :0❸
    -  -          Ssl+ 659:55 -
    -  -          Ssl+  0:00 -
    -  -          Ssl+  0:00 -
    -  -          Ssl+  8:35 -
```

Listing 8-1: Viewing threads with ps m

This listing shows processes along with threads. Each line with a number in the PID column (at ❶, ❷, and ❸) represents a process, as in the normal ps output. The lines with dashes in the PID column represent the threads associated with the process. In this output, the processes at ❶ and ❷ have only one thread each, but process 12534 at ❸ is multithreaded, with four threads.

Viendo los Threads: ps -T



```
prateekjangid@prateekjangid: ~  
prateekjangid@prateekjangid:~$ ps -T -p 1904  
  PID     SPID TTY          TIME CMD  
  1904     1904 ?          00:00:00 gjs  
  1904     1926 ?          00:00:00 gmain  
  1904     1932 ?          00:00:00 gdbus  
  1904     1940 ?          00:00:00 JS Helper  
  1904     1941 ?          00:00:00 JS Helper  
prateekjangid@prateekjangid:~$
```

The image shows a terminal window with a dark background. The title bar at the top reads 'prateekjangid@prateekjangid: ~'. The terminal content shows the command 'ps -T -p 1904' being executed. The output is a table with columns: PID, SPID, TTY, TIME, and CMD. It lists five threads of the process with PID 1904, each with its own SPID and TTY, and their respective commands: gjs, gmain, gdbus, and two instances of JS Helper. The prompt 'prateekjangid@prateekjangid:~\$' is visible at the bottom of the terminal output.

Sincronización

Sincronización



La programación multihilo extiende el modelo secuencial de programación de un único hilo de ejecución. En este modelo se pueden encontrar dos escenarios posibles:

- Un programa está compuesto por un conjunto de threads independientes que operan sobre un conjunto de datos que están completamente separados entre sí y son independientes.
- Un programa está compuesto por un conjunto de threads que trabajan en forma cooperativa sobre un set de memoria y datos que son compartidos.

Sincronización



En un programa que utiliza un modelo de programación de threads cooperativo, la forma de pensar secuencial no sirve:

1. La ejecución del programa depende de la forma en que los threads se intercalan en su ejecución, esto influye en los accesos a la memoria de recursos compartidos.
2. La ejecución de un programa puede no ser determinística. Diferentes corridas pueden producir distintos resultados, por ejemplo debido a decisiones del scheduler. Qué pasa con el debugging?
3. Los compiladores y el procesador físico pueden reordenar las instrucciones. Los compiladores modernos pueden reordenar las instrucciones para mejorar la performance del programa que se está ejecutando, este reordenamiento es generalmente invisible a los ojos de un solo thread

Sincronización



Teniendo en cuenta lo anterior, la programación multithreading puede incorporar bugs que se caracterizan por ser:

- útiles
- no determinísticos
- no reproducibles

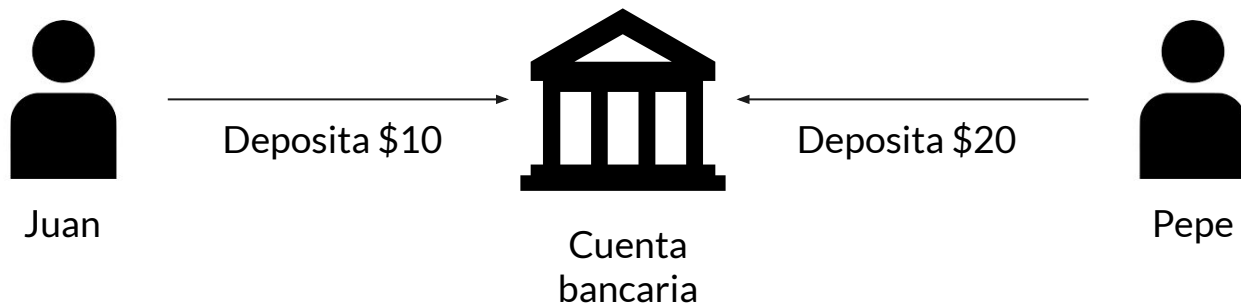
El approach a seguir en estos casos es: (1) estructurar el programa para que resulte fácil el razonamiento concurrente y (2) utilizar un conjunto de primitivas estándares para sincronizar el acceso a los recursos compartidos.

Race Conditions



Una race condition se da cuando el resultado de un programa depende en como se intercalaron las operaciones de los threads que se ejecutan dentro de ese proceso. De hecho los threads juegan una carrera entre sus operaciones, y el resultado del programa depende de quién gane esa carrera.

Ejemplo





Ejemplo

Thread 1: saldo = saldo + 10	(saldo inicialmente vale 100)
load r1, saldo	saldo=100 r1=100
add r1, r1, 10	saldo=100 r1=110
store saldo, r1	saldo=110 r1=110



Ejemplo

Thread 2: saldo = saldo + 20	(saldo inicialmente vale 100)
<code>load r1, saldo</code>	saldo=100 r1=100
<code>add r1, r1, 20</code>	saldo=100 r1=120
<code>store saldo, r1</code>	saldo=120 r1=120



Ejemplo

load	r1, saldo	saldo=100 r1=100
add	r1, r1, 10	saldo=100 r1=110
store	saldo, r1	saldo=110 r1=110
load	r1, saldo	saldo=110 r1=110
add	r1, r1, 20	saldo=110 r1=130
store	saldo, r1	saldo= 130 r1=130



Ejemplo

load r1, saldo	saldo=100 r1=1
add r1, r1, 20	saldo=100 r1=120
store saldo, r1	saldo=120 r1=120
load r1, saldo	saldo=120 r1=120
add r1, r1, 10	saldo=120 r1= 130
store saldo, r1	saldo=130 r1=130



Ejemplo

load r1, saldo	saldo=100 r1=100
add r1, r1, 20	saldo=100 r1=120
load r1, saldo	saldo=100 r1=100
store saldo, r1	saldo=100 r1=100
add r1, r1, 10	saldo=100 r1=110
store saldo, r1	saldo=110 r1=110 Se perdieron \$20!!!

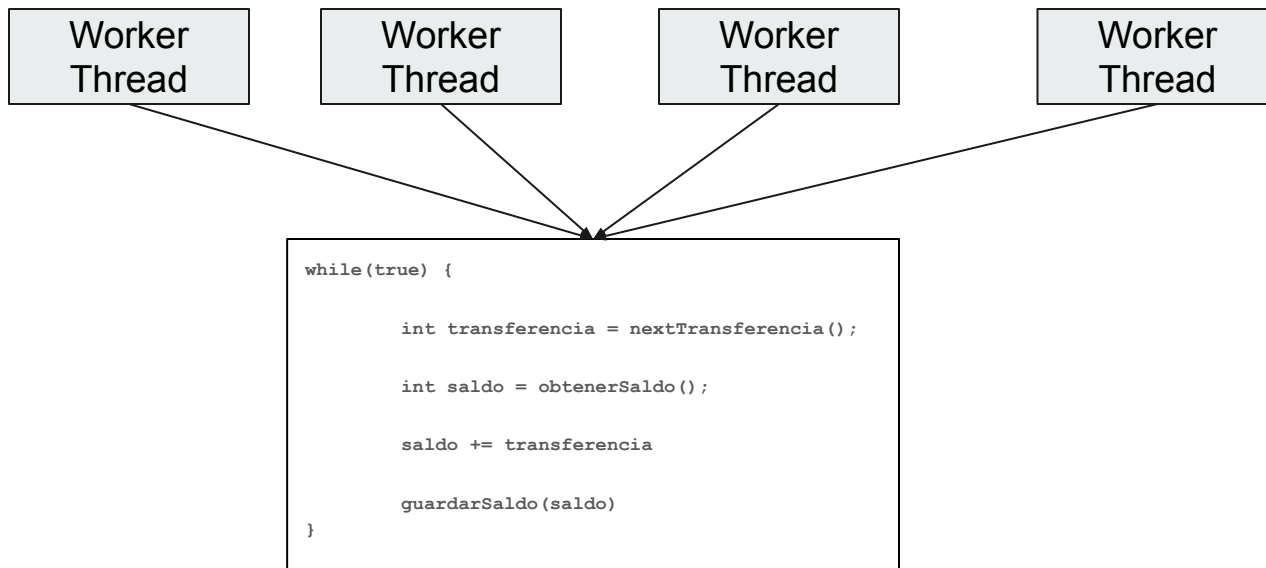


Un típico proceso para procesar transacciones

```
while(true) {  
    int transferencia = nextTransferencia();  
    int saldo = obtenerSaldo();  
    saldo += transferencia  
    guardarSaldo(saldo)  
}
```



Un típico proceso para procesar transacciones





Seccion Critica

```
while(true) {  
  
    int transferencia = nextTransferencia();  
  
    int saldo = obtenerSaldo();  
  
    saldo += transferencia  
  
    guardarSaldo(saldo)  
  
}
```

Mientras que solo un thread a la vez
entre a esta zona, esta todo bien!

Cómo se implementa una sección crítica



Cómo se implementa una sección crítica

```
while(true) {  
    int transferencia = nextTransferencia();
```

Entrar a la Seccion Critica



```
        int saldo = obtenerSaldo();  
  
        saldo += transferencia  
  
        guardarSaldo(saldo)
```

Salir de la seccion Critica

```
}
```

Race Conditions: Cual es el valor de las variables



Thread A

```
x=1;
```

Thread B

```
x=2;
```

Thread A

```
x=y+1;
```

Thread B

```
y= y * 2;
```

Race Conditions: Cual es el valor de las variables



Thread A

`x = x + 1;`

Thread B

`x = x + 2;`

```
load    x,r1
add     r2,r1,1
store   x,r2
```

```
load r1,x
add  r2,r1,2
store x,r2
```

```
load r1,x
add  r2,r1,1
store x,r2
```

```
load r1,x
add  r2,r1,2
store x,r2
```

```
load r1,x
add  r2,r1,1
store x,r2
```


```
load r1,x
add  r2,r1,2
store x,r2
```

La Cerveza



```
if ( beer ==0 ) {           //si no hay beer
    if (nota==0) {          //si no hay nota
        nota=1;             //dejar nota
        HayBeer++;          //comprar beer
        nota=0;             //sacar nota
    }
}
```


La Cerveza



```
if ( beer ==0 ) {  
    if ( beer ==0 ) {  
        if (nota==0) {  
            nota=1;  
            beer++;  
            nota=0;  
        }  
    }  
}
```

```
if (nota==0) {
```



Locks

Operaciones Atómicas



En el ejemplo anterior se desensambló y se ejecutó el programa en assembler con operaciones atómicas, este tipo de operaciones no pueden dividirse en otras y se garantiza la ejecución de la misma sin tener que intercalar ejecución.

Ojo en una arquitectura de 32 bit load y store de una palabra son atómicas, en otras arquitecturas eso no sucede por ejemplo en 64 bits.

Lock

Un lock es una variable que permite la sincronización mediante la **exclusión mutua**, cuando un thread tiene el candado o lock ningún otro puede tenerlo.

La idea principal es que un proceso asocia un lock a determinados estados o partes de código y requiere que el thread posea el lock para entrar en ese estado. Con esto se logra que sólo un thread acceda a un recurso compartido a la vez.

Esto permite la exclusión mutua, todo lo que se ejecuta en la región de código en la cual un thread tiene un lock, garantiza la atomicidad de las operaciones.





API de Locks

```
obtener(lock);  
if ( cerveza ==0 ) {  
    cerveza++;  
}  
dejar(lock);
```



API de Locks

```
pthread_mutex_t lock;  
int pthread_mutex_init(&lock,NULL);  
int pthread_mutex_lock (pthread_mutex_t * mutex);  
int pthread_mutex_unlock (pthread_mutex_t *mutex);  
int pthread_mutex_trylock(pthread_mutex_t * mutex);  
int pthread_mutex_timedlock(pthread_mutex_t * mutex, struct timespec *abb_timeout);
```



Algunas Propiedades Formales

Un lock debe asegurar :

Exclusión mutua: como mucho un solo Thread posee el lock a la vez.

Progress: Si nadie posee el lock, y alguien lo quiere ... alguno debe poder obtenerlo.

Bounded waiting: Si T quiere acceder al lock y existen varios threads en la misma situación, los demás tienen una cantidad finita (un límite) de posible accesos antes que T lo haga.

La Sección Crítica es aquella sección del código fuente que se necesita que se ejecute en forma atómica. Para ello esta sección se encierra dentro de un lock.



Comunicacion entre procesos: IPC

Son mecanismos que disponibiliza el sistema operativo para que procesos (no threads) puedan compartir información.

Con Threads el problema es trivial: todo es compartido por defecto. Con procesos los recursos a compartir tienen que ser explícitos mediante diferentes primitivas que el sistema operativo expone.

Estas primitivas se las llama colectivamente Interprocess Communication (IPC)

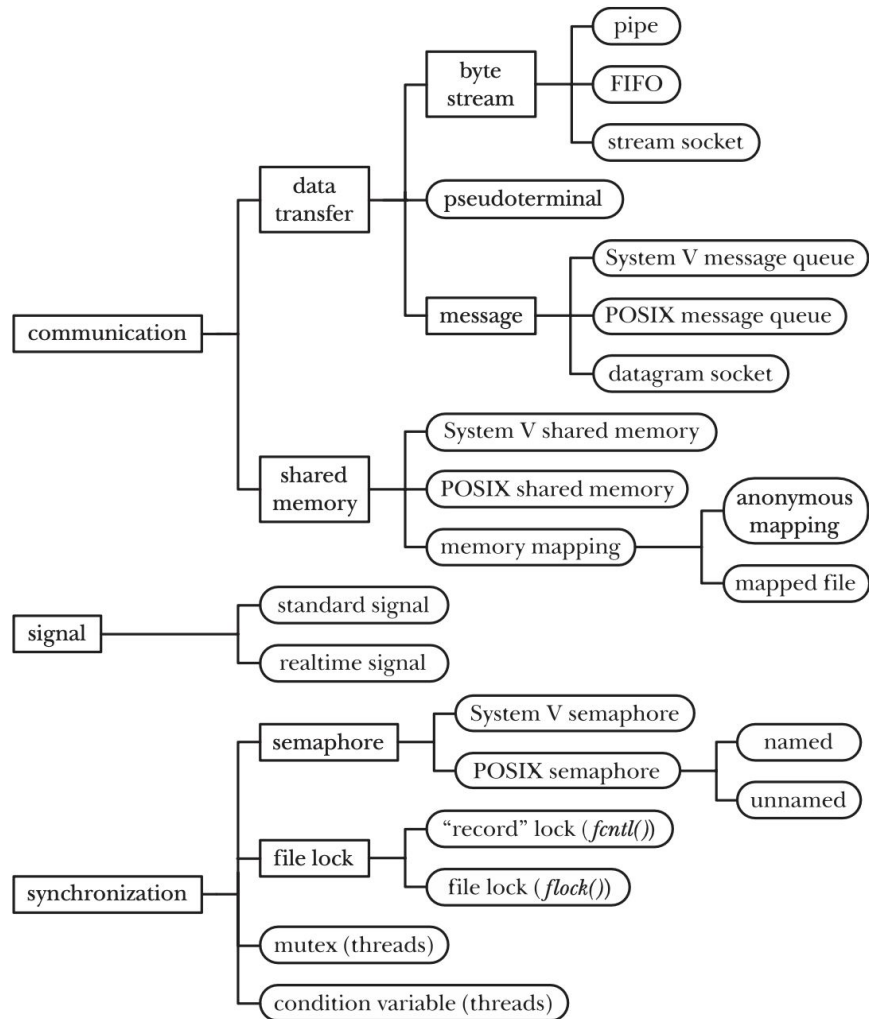


Figure 43-1: A taxonomy of UNIX IPC facilities

Ejemplo: Memoria compartida

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/mman.h>

void* create_shared_memory(size_t size) {
    // Our memory buffer will be readable and writable:
    int protection = PROT_READ | PROT_WRITE;

    // The buffer will be shared (meaning other processes can access it), but
    // anonymous (meaning third-party processes cannot obtain an address for it),
    // so only this process and its children will be able to use it:
    int visibility = MAP_SHARED | MAP_ANONYMOUS;

    // The remaining parameters to `mmap()` are not important for this use case,
    // but the manpage for `mmap` explains their purpose.
    return mmap(NULL, size, protection, visibility, -1, 0);
}
```

```
#include <string.h>
#include <unistd.h>

int main() {
    char parent_message[] = "hello"; // parent process will write this message
    char child_message[] = "goodbye"; // child process will then write this one

    void* shmem = create_shared_memory(128);

    memcpy(shmem, parent_message, sizeof(parent_message));

    int pid = fork();

    if (pid == 0) {
        printf("Child read: %s\n", shmem);
        memcpy(shmem, child_message, sizeof(child_message));
        printf("Child wrote: %s\n", shmem);
    } else {
        printf("Parent read: %s\n", shmem);
        sleep(1);
        printf("After 1s, parent read: %s\n", shmem);
    }
}
```