



<http://sites.google.com/site/utnfrbactalks/>

Oradores

Matías Dumrauf
Facundo Viale

C Talks III

C Common Libraries

- Threads
- Semáforos
- Signals
- Sockets

Threads

Threads

¿Que es un Thread? ¿Para qué un Thread?

Un Thread (Hilo) de ejecución es la unidad más chica de procesamiento que puede ser planificada. Permite la ejecución de tareas de manera **concurrente**, es decir, en paralelo.

Los Threads comparten una serie de recursos tales como el espacio de memoria, los archivos abiertos, situación de autenticación, etc. Esta técnica permite simplificar el *diseño* de una aplicación que debe llevar a cabo distintas funciones simultáneamente.

Lo que es propio de cada hilo es el contador de programa, la pila de ejecución y el estado de la CPU (incluyendo el valor de los registros).

El proceso sigue en ejecución mientras al menos uno de sus hilos de ejecución siga activo. Cuando el proceso finaliza, todos sus hilos de ejecución también han terminado. Asimismo en el momento en el que todos los hilos de ejecución finalizan, el proceso no existe más y todos sus recursos son liberados.

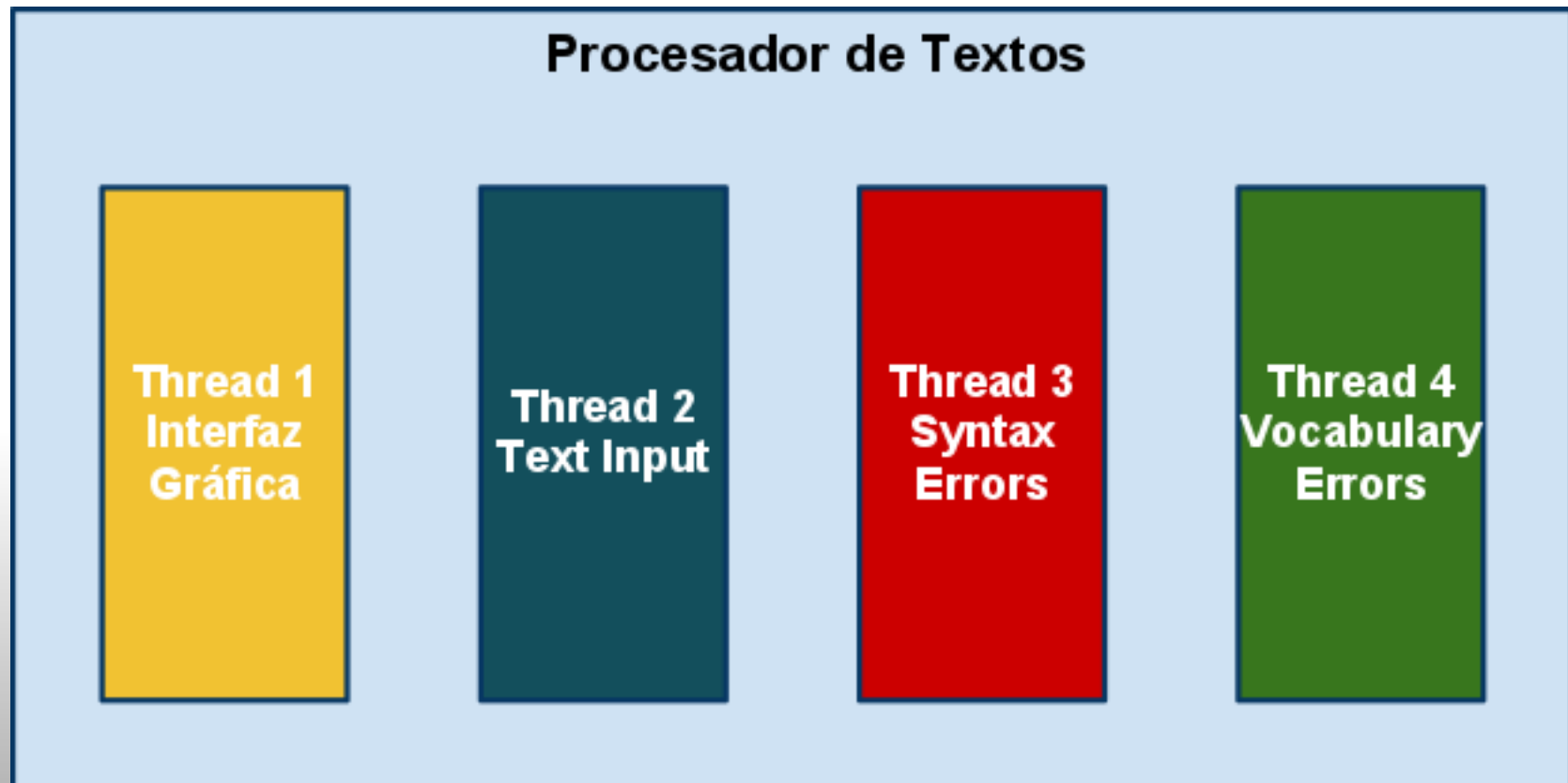


Threads

Un caso simple ilustrativo:

Un Procesador de Textos necesita ejecutar las siguientes tareas concurrentemente:

1. Mostrar una interfaz gráfica, que administre menú, botones e íconos.
2. Permitir el ingreso de texto por teclado.
3. Marcar con rojo errores en la sintaxis del lenguaje que se esté escribiendo, en Tiempo Real.
4. Marcar con verde errores de vocabulario que no concuerden con un diccionario local, en Tiempo Real.



APIs del Sistema Operativo

WinAPI Threads - Linkea internamente con la api Win32

- `_beginthreadex()`
- `_endthreadex()`
- `CreateThread()`
- `TerminateThread()`
- `WaitForSingleObject()`

PThreads - Se debe linkear explícitamente con lpthreads

- `pthread_create()`
- `pthread_exit()`
- `pthread_atexit()`
- `pthread_join()`

Threads

Posix Threads - Explicación de la API

Para que nuestra aplicación pueda utilizar Threads, necesitamos una API que nos permita "decirle" al Sistema Operativo que queremos lanzar uno, o bien terminarlo.

PThreads es un Standard POSIX para Threads. El estándar *POSIX.1c (IEEE Std 1003.1c-1995)* define una API para la creación y manipulación de los mismos.

La interfaz en sistemas Unix está definida en el header [pthread.h](#).

- **Creación:**

```
int pthread_create(pthread_t * thread_id,  
                  const pthread_attr_t * attr,  
                  void * (*start_routine)(void *),  
                  void * arg);
```

- **(Esperar a la) Terminación:**

```
int pthread_join(pthread_t thread_id, void **value_ptr);
```


Posix Threads - Explicación de la API

```
int pthread_create(pthread_t          * thread_id,  
                  const pthread_attr_t * attr,  
                  void               * (*start_routine) (void *),  
                  void               * arg);
```

Descripción: Crea un Thread.

Argumentos:

- *thread_id*: retorna el tid del Thread lanzado.
- *attr*: Si se setea con NULL, define los atributos por default (recomendado).

Para conocer los valores seteables:

<http://www.yolinux.com/TUTORIALS/PosixThreads/CreationTermination>

- *start_routine*: Puntero a función que será la rutina de ejecución del Thread. Debe poseer como único argumento un puntero a void.
- *arg*: Puntero al único argumento de la función. Para pasar más de un argumento utilizar una estructura.

Return Value:

If successful, the pthread_create() function returns zero. Otherwise, an error number is returned to indicate the error.

Posix Threads - Explicación de la API

```
int pthread_join(pthread_t *thread_id, void **value_ptr);
```

Descripción:

Suspende la ejecución del Thread *llamante* hasta que el Thread indentificado por su *thread_id* termine su ejecución.

Argumentos:


- *thread_id*: el tid del Thread a esperar.
- *value_ptr*: Si no es NULL, se retorna el valor retornado por el Thread con la función pthread_exit().

Return Value:

If successful, the pthread_join() function returns zero. Otherwise, an error number is returned to indicate the error.

Posix Threads - Ejemplo I

```
8#include <stdio.h>
9#include <stdlib.h>
10#include <pthread.h>
11
12void start_routine_example(void *arg);
13
14int main(void){
15    char msg[] = "Hello Posix Threads!";
16    pthread_t tid;
17
18    if( pthread_create(&tid, NULL, (void*)&start_routine_example, msg) ){
19        perror("Ocurrio un error durante la creacion del Thread.");
20        return EXIT_FAILURE;
21    }
22
23    pthread_join(tid, NULL);
24
25    return EXIT_SUCCESS;
26}
27
28void start_routine_example(void *arg){
29    int i;
30    char *msg = (char*) arg;
31
32    for(i = 0; i < 5; i++)
33        printf("%s\n", msg);
34}
35
```



Threads

Linking with PThreads

Para poder linkear con la Biblioteca se le debe pasar como argumento al GCC:

-pthread ó **-lpthread** (es lo mismo).

Ejemplo:

```
matias@laptop:~/workspace/C/c-talks/threads$ ls
main_threads_example1.c  main_threads_example2.c  main_threads_example3.c
matias@laptop:~/workspace/C/c-talks/threads$ gcc main_threads_example1.c -lpthread -o ThreadExample
matias@laptop:~/workspace/C/c-talks/threads$ ./ThreadExample
Hello Posix Threads!
Hello Posix Threads!
Hello Posix Threads!
Hello Posix Threads!
Hello Posix Threads!
matias@laptop:~/workspace/C/c-talks/threads$
```

Posix Threads - Ejemplo II

```
8#include <stdio.h>
9#include <stdlib.h>
10#include <pthread.h>
11
12void start_routine_example2(void *arg);
13
14int main(void){
15    int i;
16    char *msg[] = { "I'm Thread 1!", "I'm Thread 2!", "I'm Thread 3!",
17                    "I'm Thread 4!", "I'm Thread 5!"};
18
19    pthread_t threads[5];
20
21    for(i = 0; i < 5; i++){
22        if( pthread_create(&threads[i], NULL, (void*)&start_routine_example2, msg[i]) ){
23            perror("Ocurrio un error durante la creacion del Thread.");
24            return EXIT_FAILURE;
25        }
26        for(i = 0; i < 5; i++){
27            pthread_join(threads[i], NULL);
28        }
29        return EXIT_SUCCESS;
30}
31
32void start_routine_example2(void *arg){
33    char *msg = (char*) arg;
34    printf("Thread ID: <%u> | %s\n", (unsigned int)pthread_self(), msg);
35}
36
```

Posix Threads - Ejemplo III: Pasando Structs como argumentos

main_threads_example3.c

```
1 8#include <stdio.h>
2 9#include <stdlib.h>
3 10#include <pthread.h>
4 11
5 12#include "../tads/persona.h"
6 13
7 14
8 15void do_something(void *arg);
9 16
10 17int main(void){
11 18    pthread_t  tid_mati;
12 19    t_persona *p_mati = persona_crear("Dumrauf", "Matias", "DNI", 33123456, 'M', 22);
13 20
14 21    printf("Che Main, soy %s %s. Vine a dar la charla.\n", p_mati->nombre, p_mati->apellido);
15 22    puts("Main: Pasa pasa. Te estan esperando.");
16 23
17 24    if( pthread_create(&tid_mati, NULL, (void*)&do_something, (void*)p_mati) ){
18 25        perror("Ocurrio un error durante la creacion del Thread.");
19 26        return EXIT_FAILURE;
20 27    }
21 28
22 29    puts("Main: Esperando la finalizacion del Thread.");
23 30    pthread_join(tid_mati, NULL);
24 31
25 32    puts("Main: Perdona, pero como te llamabas???");
26 33    printf("%s %s, por? =P\n", p_mati->nombre, p_mati->apellido);
27 34    puts("Main: O.o No, por nada..");
28 35
29 36    persona_matar(p_mati);
30 37
31 38    return EXIT_SUCCESS;
32 39}
33 40
```

Posix Threads - Ejemplo III: Pasando Structs como argumentos

```
40
41 void do_something(void *arg){
42     t_persona *p = (t_persona*) arg;
43
44     persona_falsificarIdentidad(p, "McClure", "Troy");
45     printf("\nHola, soy %s %s\n", p->nombre, p->apellido);
46     printf("Tal vez me recuerden de otras charlas como: \n"
47           "C Talks I, C Talks II y C Talks III: \"La venganza de las C Talks!\"\n"
48           "En nuestra charla de hoy hablaremos de Threads.\n\n");
49
50     persona_falsificarIdentidad(p, "Dumrauf", "Matias");
51 }
52
```

Console X

<terminated> c-talks [C/C++ Application] /home/matias/workspace/C/c-talks/Debug/c-talks (14/09/10 03:51)

Che Main, soy Matias Dumrauf. Vine a dar la charla.

Main: Pasa pasa. Te estan esperando.

Main: Esperando la finalizacion del Thread.

Hola, soy Troy McClure

Tal vez me recuerden de otras charlas como:

C Talks I, C Talks II y C Talks III: "La venganza de las C Talks!"

En nuestra charla de hoy hablaremos de Threads.

Main: Perdona, pero como te llamabas???

Matias Dumrauf, por? =P

Main: O.o No, por nada..

Threads

Windows Threads - Explicación de la API

CreateThread() o _beginthreadex()???

Una diferencia es que la primera viene de WinNT (permitiendo más atributos de seguridad) y la segunda se adapta más -según MSDN- al estilo "puro" de C. A partir de Win9k, se comportan prácticamente igual.

Hay una salvedad: CreateThread() efectúa muchas validaciones de control internamente antes de lanzar un thread. Ésto genera mucho **OVERHEAD**. Internamente, luego de la validaciones, CreateThread() llama a _beginthreadex(). Es por eso que vamos a preferir usar _beginthreadex().

- **Creación:**

```
uintptr_t _beginthreadex(  
    void *security,  
    unsigned stack_size,  
    unsigned ( *start_address )( void * ),  
    void *arglist,  
    unsigned initflag,  
    unsigned *thrdaddr  
);
```

- **(Esperar a la) Terminación:**

```
DWORD WINAPI WaitForSingleObject(  
    __in HANDLE hHandle,  
    __in DWORD dwMilliseconds  
);
```


Windows Threads - Explicación de la API

Ejemplo de uso:

```
#define DEFAULT_INITIAL_PAGES_SIZE 1024*1024
```

```
void ftp_server_run(t_ftp_server *server){
    t_socket_client *new_client;
    t_ftp_client *newFtpClient;
    HANDLE controlThread, dataThread;

    log_write(server->log, "FTP_SERVER", "MAIN", MSG_INFO, "FTP Server en ejecucion ...");

    while(1){
        new_client = sockets_accept(server->server_socket);

        if( new_client != NULL){
            log_write(server->log, "FTP_SERVER", "MAIN", MSG_INFO,
                "Nueva conexion cliente entrante. Descriptor: %d", new_client->socket->desc);

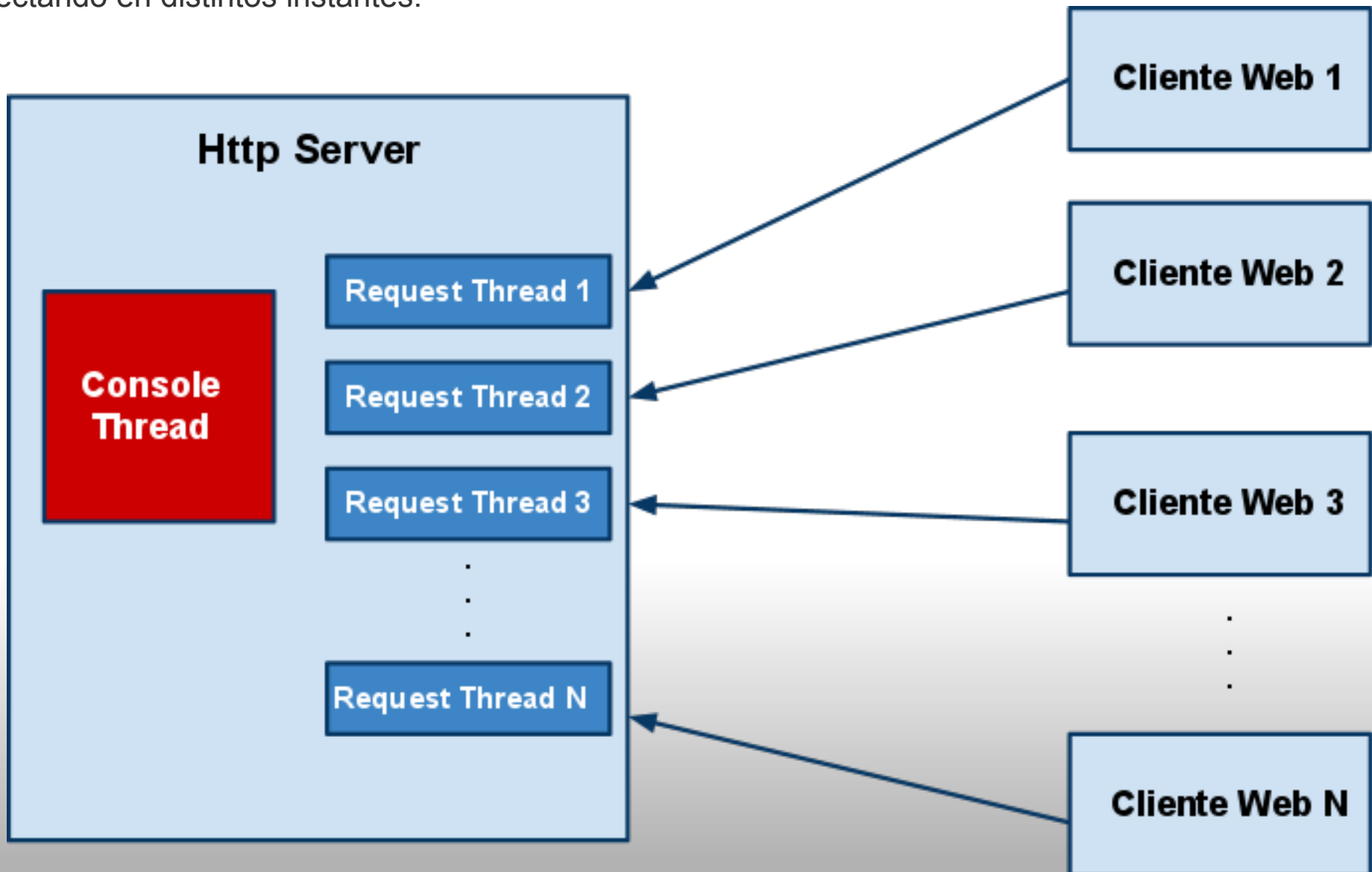
            newFtpClient = ftp_client_create(
                client, server->config->server_ip, server->config->data_port, server->config,
                server->config->default_dir, server->config->ftp_time_out, server->log);

            controlThread = _beginthreadex(NULL, DEFAULT_THREAD_STACK_SIZE, (LPVOID)ftp_server_newControlThread, newFtpClient, 0, NULL);
            dataThread = _beginthreadex(NULL, DEFAULT_THREAD_STACK_SIZE, (LPVOID)ftp_server_newDataThread, newFtpClient, 0, NULL);
        }
    }
}
```

Threads

Ejemplo "complejo":

Un Servidor Web debe poseer una consola que esté bloqueada a la espera de un comando ingresado por el Usuario. *Al mismo tiempo*, tiene que atender peticiones asincrónicas de distintos Clientes que se le van conectando en distintos instantes.



Heaps

Heaps

Explicación

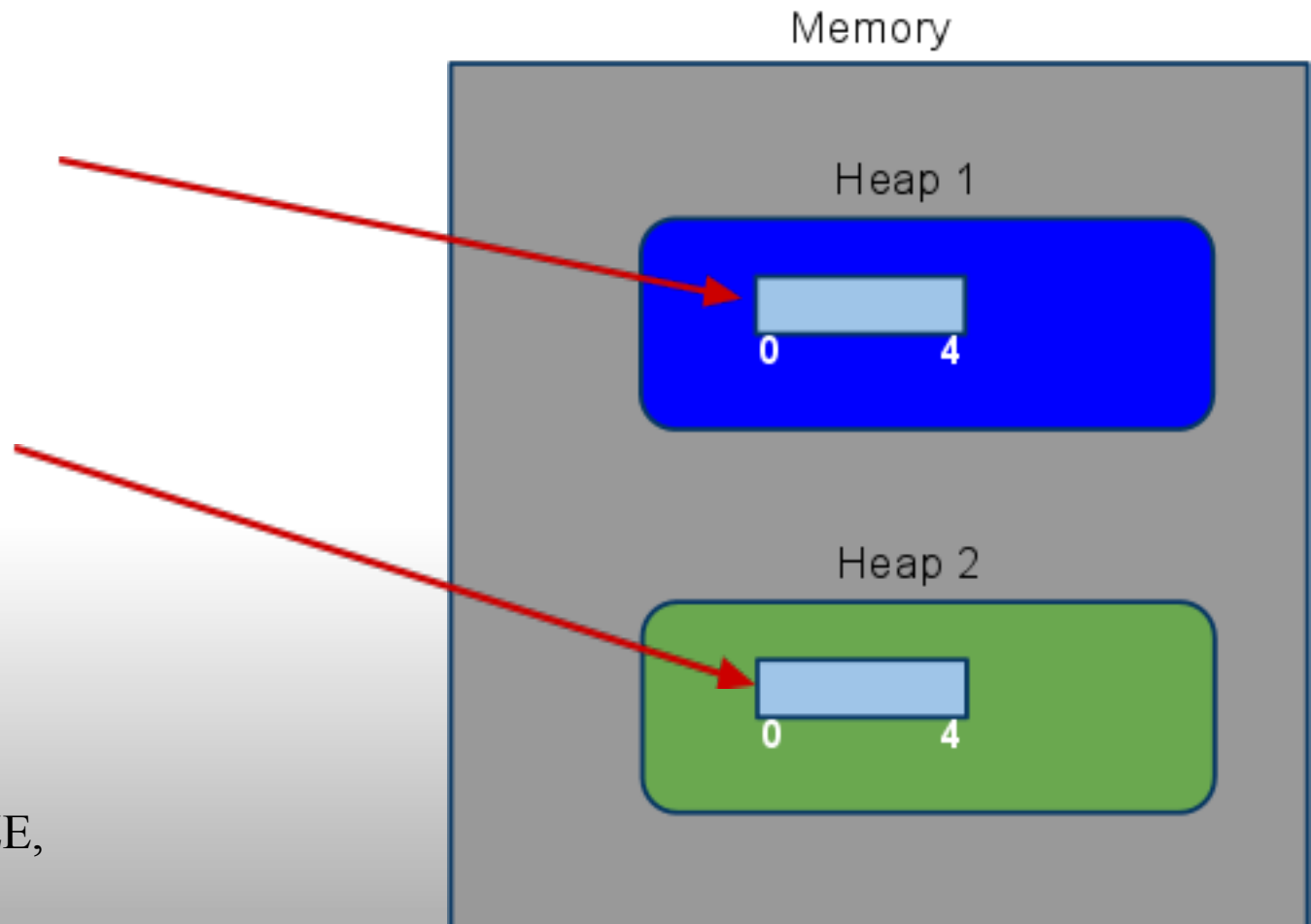
[Dynamic memory allocation \(Wikipedia\)](#)

In [computer science](#), **dynamic memory allocation** (also known as **heap-based memory allocation**) is the allocation of [memory](#) storage for use in a [computer program](#) during the [run-time](#) of that program. It can be seen also as a way of distributing ownership of limited memory resources among many pieces of data and code.

Dynamically allocated memory exists until it is released either explicitly by the programmer, or by the [garbage collector](#). This is in contrast to [static memory allocation](#), which has a fixed duration. It is said that an object so allocated has a *dynamic lifetime*.

```
char *name = malloc(4+1);
```

```
char *name = HeapAlloc(  
    GetProcessHeap(),  
    HEAP_NO_SERIALIZE,  
    4+1);
```

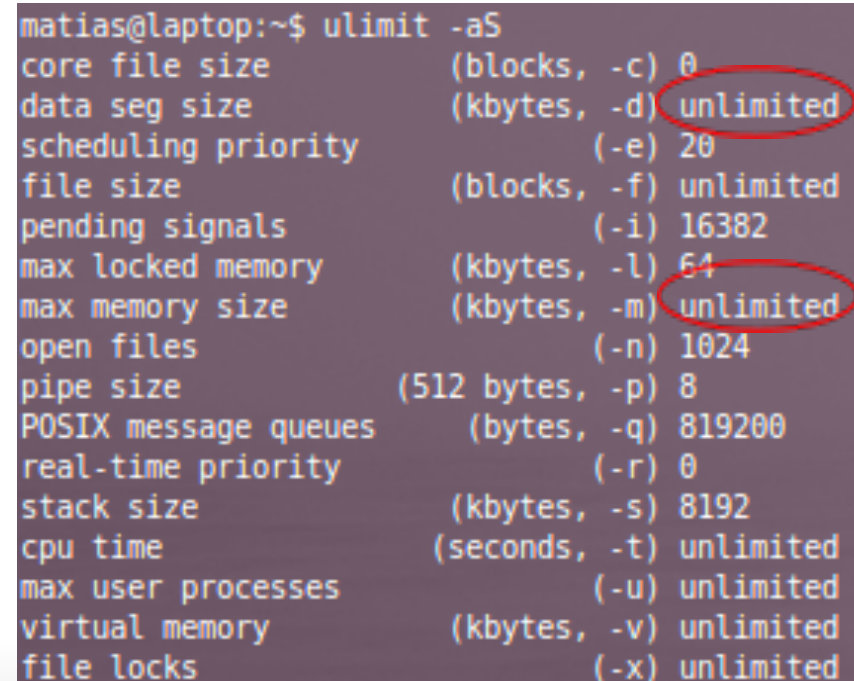


Unix Memory Allocation

En estos sistemas es transparente para el programador como se administra la memoria. Ésto se debe que sus APIs respetan estándares multiplataforma, e.g. ANSI, POSIX, ISO, que implican que el sistema operativo sea el encargado de su manejo.

Lo que no quiere decir que no exista un Heap.

Cantidad de memoria que un proceso puede alocar en su Heap



```
matias@laptop:~$ ulimit -aS
core file size          (blocks, -c) 0
data seg size           (kbytes, -d) unlimited
scheduling priority     (-e) 20
file size               (blocks, -f) unlimited
pending signals         (-i) 16382
max locked memory       (kbytes, -l) 64
max memory size         (kbytes, -m) unlimited
open files              (-n) 1024
pipe size               (512 bytes, -p) 8
POSIX message queues    (bytes, -q) 819200
real-time priority      (-r) 0
stack size              (kbytes, -s) 8192
cpu time                (seconds, -t) unlimited
max user processes      (-u) unlimited
virtual memory          (kbytes, -v) unlimited
file locks              (-x) unlimited
```

ANSI - declarados en <stdio.h>

malloc, calloc, realloc, free,

POSIX - declarados en

Windows Memory Allocation

En estos sistemas se proveen varios mecanismos para el manejo de memoria. Más allá de que la reserva la siga administrando el sistema operativo, existen mecanismos que permiten manipular la memoria explícitamente, aún a nivel de páginas.

Cada proceso posee su propio espacio de direcciones. Todos los Threads de un proceso pueden acceder al mismo, y existen varios mecanismos para las reservas dinámicas de memoria.

También es posible compartir memoria entre varios procesos.

- Virtual Address Space
- Memory Pools
- Memory Performance Information
- Virtual Memory Functions
- **Heap Functions**
- File Mapping
- Large Memory Support
- Global and Local Functions
- Standard C Library Functions
- Comparing Memory Allocation Methods

Windows Heaps

Por default, como vimos antes, cada proceso/thread posee su propio Heap. Utilizando la WinAPI, se puede obtener este Heap utilizando la función **GetProcessHeap()**. Windows recomienda crear un Heap privado por cada nuevo proceso/thread ya que el Heap por default tiene un *memory leak* de 80 bytes!!!

```
HANDLE WINAPI HeapCreate( __in DWORD flOptions, __in SIZE_T dwInitialSize, -----> Crear un
Heap. __in SIZE_T dwMaximumSize );
BOOL WINAPI HeapDestroy( __in HANDLE hHeap -----> Destruye un Heap. );
LPVOID WINAPI HeapAlloc( __in HANDLE hHeap, __in DWORD dwFlags, -----> Alocar memoria de
un Heap. __in SIZE_T dwBytes );
BOOL WINAPI HeapFree( __in HANDLE hHeap, __in DWORD dwFlags, -----> Liberar memoria
previamente alocada, al Heap. __in LPVOID lpMem );
```

Windows Heaps

Ejemplo de uso:

LE INDICAMOS AL SO QUE NO SE ENCARGUE DE LA
SERIALIZACION DENTRO DEL HEAP



```
/* Crear un Heap */  
HANDLE heap = HeapCreate(HEAP_NO_SERIALIZE, 1024*1024, 0);  
  
/* Alocar memoria dinamicamente */  
char *var = HeapAlloc(heap, HEAP_NO_SERIALIZE, strlen("Hello World") + 1);  
  
strcpy(var, "Hello World");  
puts(var);  
  
/* Liberar la memoria */  
HeapFree(heap, HEAP_NO_SERIALIZE, var);  
  
/* Destruir el Heap */  
HeapDestroy(heap);
```


Windows Heaps

Consideración sobre HeapReAlloc: el "equivalente" de realloc().

```
LPVOID WINAPI HeapReAlloc( __in HANDLE hHeap,  
__in DWORD dwFlags, __in LPVOID lpMem, __in SIZE_T  
dwBytes );
```

Dado que es posible compartir un Heap entre dos o más threads o procesos, si no seteamos el flag `HEAP_NO_SERIALIZE`, vamos a dejar que el SO maneje la serialización. Ésto, como dijimos antes, nos da un coste de performance. Si utilizamos `HEAP_NO_SERIALIZE`, Windows nos recomienda que para asegurar que no haya corrupción en el Heap y que se garantice la mutua exclusión, sólomente usemos `HeapReAlloc()` bajo las siguientes condiciones:

- El proceso posee sólomente un Thread.
- El proceso posee múltiples Threads, pero cada Thread posee su propio Heap privado.
- El proceso posee múltiples Threads y un único Heap, y él mismo se encarga de garantizar la mutua exclusión.

Windows Heaps

Ejemplo de uso con Threads:

```
public void thread_function(PVOID arg){

    /* Crear un Heap */
    HANDLE heap = HeapCreate(HEAP_NO_SERIALIZE, 1024*1024, 0);

    /* Alocar memoria dinamicamente */
    char *var = HeapAlloc(heap, HEAP_NO_SERIALIZE, strlen("Hello World") + 1);

    strcpy(var, "Hello World");
    puts(var);

    /* Liberar la memoria */
    HeapFree(heap, HEAP_NO_SERIALIZE, var);

    /* Destruir el Heap */
    HeapDestroy(heap);

    _endthreadex(0);

}
```

Semáforos

Semáforos

¿Que es un Semáforo? ¿Para qué un Semáforo?

Un Semaforo es un **Tipo Abstracto de Datos** que permite:

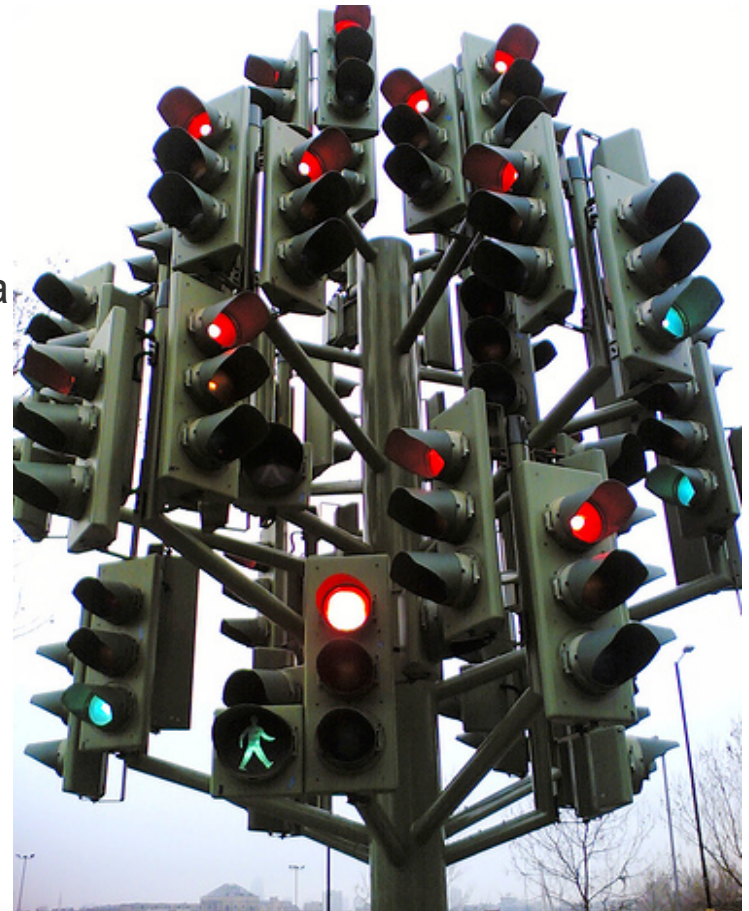
- Sincronizar ejecución ordenada de Procesos.
- Sincronizar el acceso de dos o más Procesos a una Región Crítica.
- Señalizar un Recurso que posee/puede poseer más de una unidad (Recurso Crítico).

Los **valores** que puede tomar un Semáforo pueden ser:

- 1
- 0
- > 0 (para semáforos contadores)

Las **operaciones** que se pueden efectuar con un Semáforo, son *atómicas* (consumen 1 unidad de tiempo):

- Wait(*sem*):
 - decrementa el valor de *sem* en 1.
 - si *sem* < 0 (despues del wait), el Proceso/Thread que ejecutó la operación queda bloqueado y se mete en la "Cola de Bloqueados" (FIFO) de ese semáforo, sino sigue ejecutando.
- Signal(*sem*):
 - incrementa el valor de *sem* en 1.
 - si *sem* < 0 (antes del signal), quiere decir que hay Procesos/Threads en la "Cola de Bloqueados". Entonces despierta al primero.



Semáforos

Tipos de semáforos

Se distinguen *conceptualmente* tres tipos:

- **MUTEX**: garantizan la mútua exclusión de un recurso entre dos o más procesos, mediante locks (cerrojos). Sus valores posibles son 1 o 0. Ej de uso: una estructura compartida entre dos threads.
- **BINARIOS**: son esencialmente lo mismo que el MUTEX. Pueden tomar valores 1 o 0, pero el único que puede incrementar el semáforo (signal) es el mismo que lo decrementó (wait).
- **CONTADORES**: permiten representar recursos con más de una unidad disponible. El valor del semáforo se inicializa con cantidad de unidades de un mismo recurso. Si al hacer un wait no hubieran recursos disponibles, el Proceso/Thread quedaría bloqueado hasta que alguno se libere. Ej de uso: hay 5 impresoras disponibles y 10 Procesos que las utilizan (toman-imprimen-libera).

1) Sincronizar Ejecución

Supongamos que ejecutamos el Ejemplo II de Threads varias veces:

```
matias@laptop:~/workspace/C/c-talks/threads$ gcc main_threads_example2.c -pthread -o ThreadExample2
matias@laptop:~/workspace/C/c-talks/threads$ ./ThreadExample2
Thread ID: <1195084048> | I'm Thread 1!
Thread ID: <1186691344> | I'm Thread 2!
Thread ID: <1178298640> | I'm Thread 3!
Thread ID: <1169905936> | I'm Thread 4!
Thread ID: <1161513232> | I'm Thread 5!
matias@laptop:~/workspace/C/c-talks/threads$ ./ThreadExample2
Thread ID: <649763088> | I'm Thread 1!
Thread ID: <616192272> | I'm Thread 5!
Thread ID: <632977680> | I'm Thread 3!
Thread ID: <624584976> | I'm Thread 4!
Thread ID: <641370384> | I'm Thread 2!
matias@laptop:~/workspace/C/c-talks/threads$ ./ThreadExample2
Thread ID: <2566535440> | I'm Thread 1!
Thread ID: <2541357328> | I'm Thread 4!
Thread ID: <2558142736> | I'm Thread 2!
Thread ID: <2549750032> | I'm Thread 3!
Thread ID: <2532964624> | I'm Thread 5!
matias@laptop:~/workspace/C/c-talks/threads$
```

Que pasó? Por qué no ejecutó en el orden en que fueron lanzados??

Semáforos

1) Sincronizar Ejecución

Como no *establecimos* el **Orden de Ejecución**, y todos los Threads se crean en el "mismo instante", cualquiera pueda empezar a ejecutar en cualquier momento.

Si queremos que se ejecuten en orden -> hay que Sincronizar la ejecución con Semáforos para que **"cuando termine de ejecutar uno, habilite la ejecución del otro"**

Para ésto, se emplean semáforos **MUTEX**.

Supongamos que queremos que los Threads se ejecuten **SIEMPRE** en el orden: T1, T2, T3, T4, T5

Estado Inicial de los Semáforos

S1 = 1 S2 = 0 S3 = 0 S4 = 0 S5 = 0

| Thread 1 | Thread 2 | Thread 3 | Thread 4 | Thread 5 |
|-------------|-------------|-------------|-------------|-------------|
| lock(S1); | lock(S2); | lock(S3); | lock(S4); | lock(S5); |
| ejecutar(); | ejecutar(); | ejecutar(); | ejecutar(); | ejecutar(); |
| unlock(S2); | unlock(S3); | unlock(S4); | unlock(S5); | |

lock() y unlock() son equivalentes a wait() y signal() respectivamente.

Semáforos

2) Sincronizar acceso a Región Crítica

Región Crítica: porción de memoria compartida entre dos o más procesos/threads que puede ser accedida concurrentemente por varios de ellos en un determinado instante.

Recurso Crítico: recurso compartido entre dos o más procesos/threads que puede ser accedido concurrentemente por varios de ellos en un determinado instante.

Ej: `int x = 0;`

Thread 1

`printf("%d\n", x);`

`x = x + 2;`

Thread 2

`printf("%d\n", x);`

`if(x == 0)`

`x = x + 1;`

`printf("%d\n", x);`

-----> **CHAN !**

En un determinado momento, habiendo lanzado los dos Threads, los dos están en dentro de la Región Crítica y acceden a la variable 'x'.

Semáforos

2) Sincronizar acceso a Región Crítica

Solución:

Estado Inicial de los Semáforos

MUTEX_1 = 0 MUTEX_2 = 1

int x = 0;

Thread 1

Thread 2

```
Thread 1: while( 1 ){  
    lock(MUTEX_1);  
    printf("%d\n", x);  
    x = x + 2;  
    unlock(MUTEX_2);  
}  
  
Thread 2: while( 1 ){  
    lock(MUTEX_2);  
    printf("%d\n", x);  
    if( x == 0 )  
        x = x + 1;  
    printf("%d\n", x);  
    unlock(MUTEX_1);  
}
```

Los locks **SIEMPRE** se cruzan.

Semáforos

3) Señalizar un Recurso Crítico que posee/puede poseer de más de una unidad

Supongamos que tuviéramos 2 impresoras accedidas simultáneamente por 3 Procesos o Threads. En un determinado momento podría ocurrir que sólo hay 1 impresora disponible y que dos Procesos/Threads quisieran imprimir.

Para ésto se utilizan semáforos **CONTADORES**. Se inicializan con la cántidad de unidades de un recurso.

Imaginemos que no se Señalizara el recurso: saldría la impresión "un poco de documento", "un poco de otro", "cortado", etc.

Estado Inicial del Semáforo

S_IMPRESORA = 2

Ej:

Thread 1

```
while( 1 ){  
    preparar_doc_muy_largo();  
    wait(S_IMPRESORA);  
    imprimir_doc_muy_largo();  
    signal(S_IMPRESORA);  
}
```

Thread2

```
while( 1 ){  
    preparar_doc_largo();  
    wait(S_IMPRESORA);  
    imprimir_doc_largo();  
    signal(S_IMPRESORA);  
}
```

Thread2

```
while( 1 ){  
    preparar_doc_tranki();  
    wait(S_IMPRESORA);  
    imprimir_doc_tranki();  
    signal(S_IMPRESORA);  
}
```

APIs del Sistema Operativo

WinAPI - Linkea internamente con la api Win32

- CreateSemaphore()
- CreateMutex()
- WaitForSingleObject()
- ReleaseSemaphore()
- ReleaseMutex()
- CloseHandle()

POSIX - Se debe linkear explícitamente con lthreads

- sem_init()
- pthread_mutex_init()
- pthread_mutex_lock()
- pthread_mutex_unlock()
- pthread_mutex_destroy()
- sem_wait()
- sem_post()
- sem_destroy()

Semáforos

Explicación de la API POSIX

La API POSIX provee dos interfaces de semáforos:

- MUTEX:

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

```
#include <time.h>
```

```
int pthread_mutex_init(pthread_mutex_t *mutex, const pthread_mutexattr_t *mutexattr);
```

```
int pthread_mutex_lock(pthread_mutex_t *mutex);
```

```
int pthread_mutex_trylock(pthread_mutex_t *mutex);
```

```
int pthread_mutex_timedlock(pthread_mutex_t *mutex, const struct timespec *abs_timeout);
```

```
int pthread_mutex_unlock(pthread_mutex_t *mutex);
```

```
int pthread_mutex_destroy(pthread_mutex_t *mutex);
```

Return Value

pthread_mutex_init always returns 0. The other mutex functions return 0 on success and a non-zero error code on error.

Explicación de la API

Error Codes

.

The **pthread_mutex_lock** function returns the following error code on error:

EINVAL: the mutex has not been properly initialized.

EDEADLK: the mutex is already locked by the calling thread (“error checking” mutexes only).

The **pthread_mutex_trylock** function returns the following error codes on error:

EBUSY: the mutex could not be acquired because it was currently locked.

EINVAL: the mutex has not been properly initialized.

The **pthread_mutex_timedlock** function returns the following error codes on error:

ETIMEDOUT: the mutex could not be acquired before the *abs_timeout* time arrived.

EINVAL: the mutex has not been properly initialized.

The **pthread_mutex_unlock** function returns the following error code on error:

EPERM: the calling thread does not own the mutex (“error checking” mutexes only).

EINVAL: the mutex has not been properly initialized.

The **pthread_mutex_destroy** function returns the following error code on error:

EBUSY: the mutex is currently locked.

Semáforos

Explicación de la API

- **SEMAPHORE**: (para contadores y binarios, que funcionan como Mutex)

#include <semaphore.h>

```
int sem_init(sem_t *sem, int pshared, unsigned int value);  
int sem_wait(sem_t * sem);  
int sem_timedwait(sem_t * sem, const struct timespec *abstime);  
int sem_trywait(sem_t * sem);  
int sem_post(sem_t * sem);  
int sem_post_multiple(sem_t * sem, int number);  
int sem_getvalue(sem_t * sem, int * sval);  
int sem_destroy(sem_t * sem);
```

Return Value

All semaphore functions return 0 on success, or -1 on error in which case they write an error code in **errno**.

Explicación de la API

Error Codes

The **sem_init** function sets **errno** to the following codes on error:

EINVAL: *value* exceeds the maximal counter value **SEM_VALUE_MAX**.

ENOSYS: *pshared* is not zero.

The **sem_timedwait** function sets **errno** to the following error code on error:

ETIMEDOUT: if *abstime* arrives before the waiting thread can resume following a call to **sem_post** or **sem_post_multiple**.

The **sem_trywait** function sets **errno** to the following error code on error:

EAGAIN: if the semaphore count is currently 0.

The **sem_post** and **sem_post_multiple** functions set **errno** to the following error code on error:

ERANGE: if after incrementing, the semaphore count would exceed **SEM_VALUE_MAX**
(the semaphore count is left unchanged in this case)

The **sem_destroy** function sets **errno** to the following error code on error:

EBUSY: if some threads are currently blocked waiting on the semaphore.

Semáforos

Explicación de la API - WinAPI

La WinAPI provee, también, dos interfaces:

- MUTEX:

```
HANDLE WINAPI CreateMutex( __in_opt LPSECURITY_ATTRIBUTES lpMutexAttributes, __in BOOL bInitialOwner,
__in_opt LPCTSTR lpName );
BOOL WINAPI ReleaseMutex( __in HANDLE hMutex);
```

- SEMAPHORE (contadores):

```
HANDLE WINAPI CreateSemaphore( __in_opt LPSECURITY_ATTRIBUTES lpSemaphoreAttributes, __in LONG lInitialCount, __in LONG lMaximumCount, __in_opt LPCTSTR lpName );
BOOL WINAPI ReleaseSemaphore( __in HANDLE hSemaphore, __in LONG lReleaseCount, __out_opt LPLONG lpPreviousCount -----> );
```

A pointer to a variable to receive the previous count for the semaphore. This parameter can be NULL if the previous count is not required.

Semáforos

Explicación de la API - WinAPI

Para destruir el semáforo y para hacerle un **down**, la WinAPI utiliza las mismas dos funciones tanto para MUTEX como para Contadores.

DWORD WINAPI WaitForSingleObject(__in HANDLE hHandle, __in DWORD dwMilliseconds ----->);

The time-out interval, in milliseconds. If a nonzero value is specified, the function waits until the object is signaled or the interval elapses.

If *dwMilliseconds* is zero, the function does not enter a wait state if the object is not signaled; it always returns immediately. If *dwMilliseconds* is **INFINITE**, the function will return only when the object is signaled.

BOOL WINAPI CloseHandle(__in HANDLE hObject);

Explicación de la API - WinAPI: MUTEX

```
HANDLE mutex = CreateMutex(NULL, FALSE, NULL);
```

```
if ( mutex == NULL ){  
    /* Ocurrio un error durante la creacion */  
  
    return NULL;  
}
```

```
WaitForSingleObject(mutex, INFINITE);
```

```
ReleaseMutex(mutex);
```

```
CloseHandle(mutex);
```

Explicación de la API - WinAPI: Semaphore

```
HANDLE mutex = CreateSemaphore(NULL, 1, 1, NULL);
```

```
if ( mutex == NULL ){  
    /* Ocurrio un error durante la creacion */  
    return NULL;  
}
```

```
WaitForSingleObject(mutex, INFINITE);
```

```
ReleaseSemaphore(mutex, 1, NULL);
```

```
CloseHandle(mutex);
```

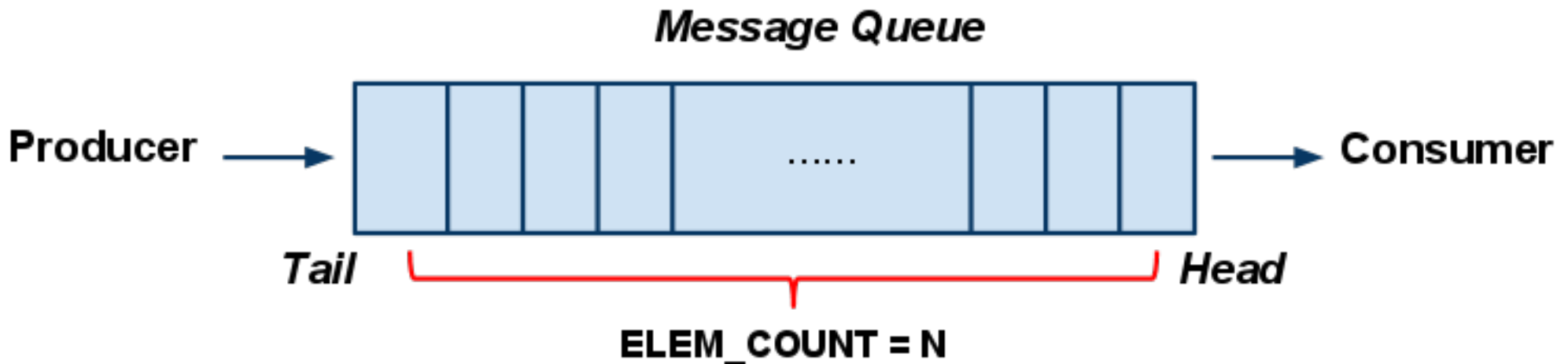
Semáforos

Ejemplo clásico: el Productor-Consumidor

Se tiene una Cola de Mensajes infinita, sin tamaño fijo, y dos o más Threads/Procesos en ejecución, cada uno cumpliendo un rol determinado.

Producer: Genera nuevos mensajes y los **Deposita** en la Cola.

Consumer: Retira los nuevos mensajes de la Cola (si los hay) y los **Consume**.



La Cola no es más que un buffer infinito de almacenamiento intermedio.

Es un método de IPC (Inter Process Communication) llamado, justamente, **Message Queueing**.

El **Consumidor** se encuentra bloqueado a la espera de que se ingresen nuevos Mensajes en la Cola, para luego retirarlos desde la Cabeza de la misma (**Head**) y consumirlos de la manera más conveniente. De llegar a haber varios mensajes, los irá retirando de a uno por orden FIFO.

De la misma manera, el **Productor** genera nuevos mensajes y los va depositando en la Cola, desde su *cola* (**Tail**), es decir, desde la parte de atrás.

Para el diseño de este modelo se requiere de un semáforo **CONTADOR**, que llamaremos **ELEM_COUNT**.

Semáforos

Ejemplo clásico: el Productor-Consumidor

Consideremos que vamos a diseñar nuestra propia API (Application Programming Interface) para manejar Colas de Mensajes (es decir, nuestra propia biblioteca), a partir de un TAD que llamaremos Queue.

Esencialmente vamos a necesitar dos funciones:

1. `queue_push()`: para meter un nuevo mensaje en la cola.
2. `queue_pop()`: para obtener un mensaje de la cola.

El Estado Inicial del semáforo contador **ELEM_COUNT** sería 0, ya que la Cola va a estar vacía.

Lo único que nos queda por definir, a grandes rasgos, son las dos funciones de los Procesos/Threads:

3. `consumir()`: que consume el mensaje.
4. `producir()`: que produce el mensaje.

El pseudo-código de especificación de la lógica de los dos Procesos/Threads sería el siguiente:

```
producer() {  
    while(1) {  
        msg = producir();  
        queue_push(cola, msg);  
        signal(ELEM_COUNT);  
    }  
}
```

```
consumer() {  
    while(1) {  
        wait(ELEM_COUNT);  
        msg = queue_pop(msg);  
        consumir(msg);  
    }  
}
```

Semáforos

Ejemplo clásico: el Productor-Consumidor

Para implementar este modelo Producer-Consumer, vamos a necesitar Especificar, Diseñar e Implementar el TAD Queue.

Especificación:

TAD_{queue} = (Queue, OP_{queue})

Queue = { Q / Q = (Head, Tail, Elem_Count) y

Head : puntero a **t_link_element** (primer elemento de la Queue)

Tail : puntero a **t_link_element** (ultimo elemento de la Queue)

Elem_Count : sem_t (semáforo contador de elementos de la Queue)

}

t_link_element = struct link_element

struct link_element = (Data, Next) donde

Data = puntero a **void** (generico)

Next = puntero a **struct link_element**

(forma recursiva de definir el tipo abstracto del nodo de la Queue)

Ejemplo clásico: el Productor-Consumidor

Especificación:

$OP_{\text{queue}} = \{ \text{queue_create}, \text{queue_push}, \text{queue_pop}, \text{queue_destroy} \}$

donde

```
queue_create    :: void          -> t_queue*
queue_push      :: t_queue* X void* -> void
queue_pop       :: t_queue*       -> void*
queue_destroy   :: t_queue*       -> void
}
```

Ejemplo clásico: el Productor-Consumidor

Diseño

```
1
2 #ifndef QUEUE_H
3 #define QUEUE_H
4
5     #include <semaphore.h>
6
7     typedef struct link_element{
8         void *data;
9         struct link_element *next;
10    } t_link_element;
11
12    typedef struct{
13        t_link_element *head;
14        t_link_element *tail;
15        sem_t          elem_count;
16    } t_queue;
17
18    t_queue *queue_create();
19    void     queue_push(t_queue *queue, void *data);
20    void     *queue_pop(t_queue *queue);
21    void     queue_destroy(t_queue *queue);
22
23 #endif /* QUEUE_H */
24
```


Semáforos

Ejemplo clásico: el Productor-Consumidor - Implementación

```
queue.c
8#include <stdlib.h>
9#include "queue.h"
10
11t_queue *queue_create(){
12    t_queue *queue = malloc(sizeof(t_queue));
13    queue->head = NULL;
14    queue->tail = NULL;
15
16    if (sem_init(&queue->elem_count, 0, 0) == -1) {
17        free(queue);
18        return NULL;
19    }
20    return queue;
21}
22
23void queue_push(t_queue *queue, void *data){
24    t_link_element *new_elem = malloc(sizeof(t_link_element));
25    new_elem->data = data;
26    new_elem->next = NULL;
27
28    if (queue->head == NULL)
29        queue->head = new_elem;
30    else
31        queue->tail->next = new_elem;
32
33    queue->tail = new_elem;
34    sem_post(&queue->elem_count);
35}
36
37void *queue_pop(t_queue *queue){
38    t_link_element *elem;
39    void *data;
40
41    sem_wait(&queue->elem_count);
42    elem = queue->head;
43    data = elem->data;
44    queue->head = queue->head->next;
45    if (queue->head == NULL)
46        queue->tail = NULL;
47
48    free(elem);
49    return data;
50}
51
52void queue_destroy(t_queue *queue){
53    sem_destroy(&queue->elem_count);
54    free(queue);
55}
```

Ejemplo clásico: el Productor-Consumidor

Implementación
de un Programa
Message Queueing

para un

Producer de Msgs
y un Consumer
que los imprime
por Standard Out.

```
Message_Queueing.c
8#include <stdio.h>
9#include <stdlib.h>
10#include <string.h>
11#include <pthread.h>
12#include "../lib/queue.h"
13
14void consumer(void *arg);
15
16int main(void) {
17    t_queue    *queue = queue_create();
18    pthread_t   consumer_tid;
19    char        buffer[20];
20    char        *msg_rcv;
21
22    if (queue == NULL) {
23        perror("Error al intentar crear la Queue.");
24        return EXIT_FAILURE;
25    }
26    if (pthread_create(&consumer_tid, NULL, (void*)&consumer, (void*)queue)) {
27        perror("Ocurrio un error durante la creacion del Thread.");
28        return EXIT_FAILURE;
29    }
30    while (1) {
31        /* Leer una cadena desde la Entrada Standard */
32        fgets(buffer, 20, stdin);
33        msg_rcv = malloc(strlen(buffer) + 1);
34        strcpy(msg_rcv, buffer);
35        queue_push(queue, msg_rcv);
36
37        if (!strcmp(buffer, "exit", 4)) break;
38    }
39    pthread_join(consumer_tid, NULL);
40    queue_destroy(queue);
41
42    return EXIT_SUCCESS;
43}
```

Ejemplo clásico: el Productor-Consumidor

Lógica del Thread Consumer:

```
45 void consumer(void *arg){
46     t_queue *queue = (t_queue*) arg;
47     char *new_msg;
48     int EXIT_CMD_RECV = 0;
49
50     while (!EXIT_CMD_RECV){
51         new_msg = queue_pop(queue);
52         printf("Mensaje recibido: %s\n", new_msg);
53
54         if (!strncmp(new_msg, "exit", 4)) EXIT_CMD_RECV = 1;
55
56         free(new_msg);
57     }
58 }
```

Signals

¿ Que es una Signal ?

Es una forma limitada de realizar IPC (Inter-Process Communication) propia de los sistemas Unix, la cual consiste en una forma de notificación asincronica que informa a un proceso de la ocurrencia de un evento. Cuando un proceso recibe una señal, detiene su ciclo de ejecución para atender a esta.

Todas las signals tiene un handler (manejador/función de tratamiento) el cual tiene definido un comportamiento a realizar cuando se le envié la signal al proceso. Por defecto todas las signals ya tienen un handler asignado cuando el proceso entra en ejecución.

El programador puede re-escribir los handlers e incluso indicar que la signal sea ignorada (Aunque esto no es posible para todas).

Signals

Existe muchos tipos de Signals los cuales están definidos en signal.h, estos son algunos ejemplos:

| Signal | Descripción | Tratamiento por Defecto | Es Ignorable? |
|---------|--|----------------------------------|---------------|
| SIGSEGV | Segmentation Violation.Salta con dirección de memoria ilegal | exit + volcado de memoria | SI |
| SIGCHLD | Proceso hijo terminado, detenido (*o que continúa) | ignorar | SI |
| SIGPIPE | Se genera al escribir sobre la pipe sin lector | exit | SI |
| SIGINT | Interrupción, se genera al pulsar "^c" durante la ejecución | exit | SI |
| SIGKILL | Destrucción inmediata del proceso | exit (<i>No reprogramable</i>) | NO |
| SIGSTOP | Detiene el proceso | exit (<i>No reprogramable</i>) | NO |
| SIGUSR1 | User defined 1. Signal definido por el usuario | exit | SI |

Ejemplo:

```
#include <stdio.h>
#include <stdlib.h>
#include <signal.h>
#include <unistd.h>

void print(int signal){
    printf("SIGUSR1!!\n");
}

int main(int argc, char **argv) {

    signal(SIGUSR1, print);

    raise(SIGUSR1);

    while(1){
        usleep(1000);
    }

    return EXIT_SUCCESS;
}
```

```
facundoviale@ARBADCL051:~/Development/C$ ./Signal &
[2] 20691
facundoviale@ARBADCL051:~/Development/C$ SIGUSR1!!
```

```
facundoviale@ARBADCL051:~/Development/C$ ps -fea | grep Signal
1000      20691 20580  0 14:32 pts/1    00:00:00 ./Signal
1000      20693 20580  0 14:32 pts/1    00:00:00 grep Signal
facundoviale@ARBADCL051:~/Development/C$ kill -SIGUSR1 20691
facundoviale@ARBADCL051:~/Development/C$ SIGUSR1!!
```

Signals y TADs el problema de los Scopes

Como se puede observar el handler solo recibe un entero, y es ejecutado internamente. Supongamos el caso de que queremos enviar una signal y hacer que esta nos imprima el nombre del usuario que esta logeado en nuestra aplicación. Este dato es algo que esta dentro de nuestro TAD de la aplicación y no queremos declarar una variable global.

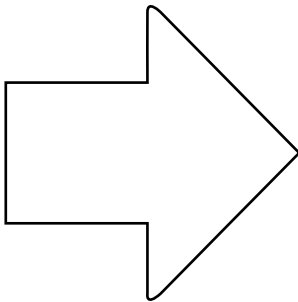
Solución:

```
20 void print_user(int signal){
21     static t_user *user = NULL;
22
23     if( user == NULL ){
24         user = (t_user*)signal;
25         return;
26     }
27
28     printf("Usuario Logeado: %s\n", user->name);
29 }
30
31 int main(int argc, char **argv) {
32     t_user *user = malloc( sizeof(t_user) );
33     user->name = "John Doe";
34
35     print_user((int)user);
36
37     signal(SIGUSR1, print_user);
38     signal(SIGUSR2, SIG_IGN);
39
40     raise(SIGUSR1);
41
42     return EXIT_SUCCESS;
43 }
```


Consideraciones en el uso de Fork

Si creamos un proceso hijo, al finalizar este manda un SIGCHLD al proceso padre, libera todos sus recursos, dejando lo suficiente como para mantener un esqueleto del proceso, y entra en un estado a la espera de que el proceso padre lea el estado con el que finalizo el proceso. Por defecto el proceso padre ignora la SIGCHLD, por lo cual los procesos hijo siempre quedan en estado **ZOMBIE!**

Solucion



```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <signal.h>
4 #include <unistd.h>
5 #include <sys/wait.h>
6
7
8 void sigchld_handler(int signal){
9     while( waitpid(-1, 0, WNOHANG) ); /* eliminate zombies */
10 }
11
12 int main(int argc, char **argv) {
13
14     signal(SIGCHLD, sigchld_handler);
15
16     int child_pid = fork();
17
18     if (child_pid > 0) { /* parent process */
19         while(1) usleep(1000);
20     }
21
22     return EXIT_SUCCESS;
23 }
```

Sockets

¿ Que es un Socket ?

Es una forma de realizar IPC (Inter-Process Communication), pero se destaca por no solo comunicar procesos de una misma PC sino que también lo puede hacer entre distintas PCs. Esto es porque su funcionamiento trabaja a través de un protocolo red (Ej: Protocolo IP).

Un Socket esta definido por:

- Una dirección IP destino
- Un numero de puerto destino (Max. 65535)
- Un protocolo:
 - TCP (Transmission Control Protocol):
 - Transporte fiable de flujo de bits, es decir internamente gestión la retransmisiones de paquetes, pérdida de paquetes, orden en el que llegan los paquetes, paquetes duplicados, etc ...)
 - Para llevar a cabo las gestiones anteriores se tiene que añadir bastante información a los paquetes que se envían.
 - UDP (User Datagram Protocol):
 - Proporciona un nivel de transporte no fiable, ya que apenas añade la información necesaria para la comunicación extremo a extremo al paquete. No introduce retardos para establecer una conexión, no mantiene estado de conexión alguno y no realiza seguimiento de estos parámetros.
- Opcionalmente también se puede especificar una IP origen y un puerto origen.

Algunas consideraciones

- Toda la información enviada o recibida a través de un socket se encuentra encapsulada dentro del protocolo a través del cual funciona el Socket, como seria el Protocolo TCP/IP. Esto ocurre de manera transparente para el programador, y es gestionado por parte del sistema operativo.
- Los Puertos que usan los sockets van de 0 a 65535 (2 bytes de direccionamiento) y estos existen para cada IP de la PC. Por cada par IP-Puerto son comunes a todos los procesos, por lo que no puede haber 2 procesos usando un mismo par IP-Puerto.
 - Los puertos inferiores al 1024 son puertos reservados para el sistema operativo y usados por "protocolos conocidos". Ej: FTP (20 y 21), HTTP (80), POP3 (110), etc ...
- Si un proceso trata de usar un puerto que ya esta en uso por otro proceso, este lanza un error en tiempo de ejecución, informando que el puerto que uno selecciono ya esta en uso o "bindeado".
- Cuando un proceso que creo sockets finaliza sin antes haberlos cerrado, estos son cerrados por el OS cuando se liberan los recursos de este.
- La IP 127.0.0.1 (localhost o loopback) es una dirección reservada que tienen todas las computadoras, router o dispositivo independientemente de que disponga o no de una Placa Red y es utilizada para pruebas de retroalimentación. Esto quiere decir cuando queremos que un proceso se conecte a otro proceso de la misma PC a través de una IP local fija.

Funcionalidad de los Sockets

Esta reseña no es referente a una plataforma en particular sino que habla en rasgos generales de las cosas que se pueden hacer con los sockets.

- **Crear:** Tal como dice, es la creación de un socket. Esta función no tiene que devolver algo que nos referencia al socket creado. En lenguajes de bajo nivel como C se retorna un Descriptor ID, que simplemente un numero entero que identifica al socket creado.
- **Ligar:** Es una funcionalidad que nos permite definir explícitamente una IP y puerto a los cuales queremos asociar nuestro socket. Esto puede ser necesario ya que durante la creación del socket el puerto e IP al que esta asociado el socket son elegidos por el SO.
- **Escuchar:** El socket queda "en modo servidor", especificándole a través de que puerto va a recibir conexiones.
- **Conectar:** Especificando una IP y Puerto destino el socket intenta conectarse.
- **Aceptar:** Esta función es propia de cuando el socket esta "en modo servidor" y sirve para aceptar la conexión de nuevos clientes.
- **Enviar:** Envían una n cantidad de información, a través de un socket
- **Recibir:** Lee una n cantidad de información que halla llegado a través de un socket.
- **Cerrar:** El socket queda destruido por lo que el puerto que tenia asociado queda a disposición de otro proceso.

Funcionalidad de los Sockets

| Funcionalidad | Posix | WinAPI |
|---------------|---------|-------------|
| Crear | socket | socket |
| Ligar | bind | bind |
| Escuchar | listen | listen |
| Conectar | connect | connect |
| Aceptar | accept | accept |
| Enviar | send | send |
| Recibir | recv | recv |
| Cerrar | close | closesocket |

Posix:

<sys/socket.h> : http://www.opengroup.org/onlinepubs/9699919799/basedefs/sys_socket.h.html

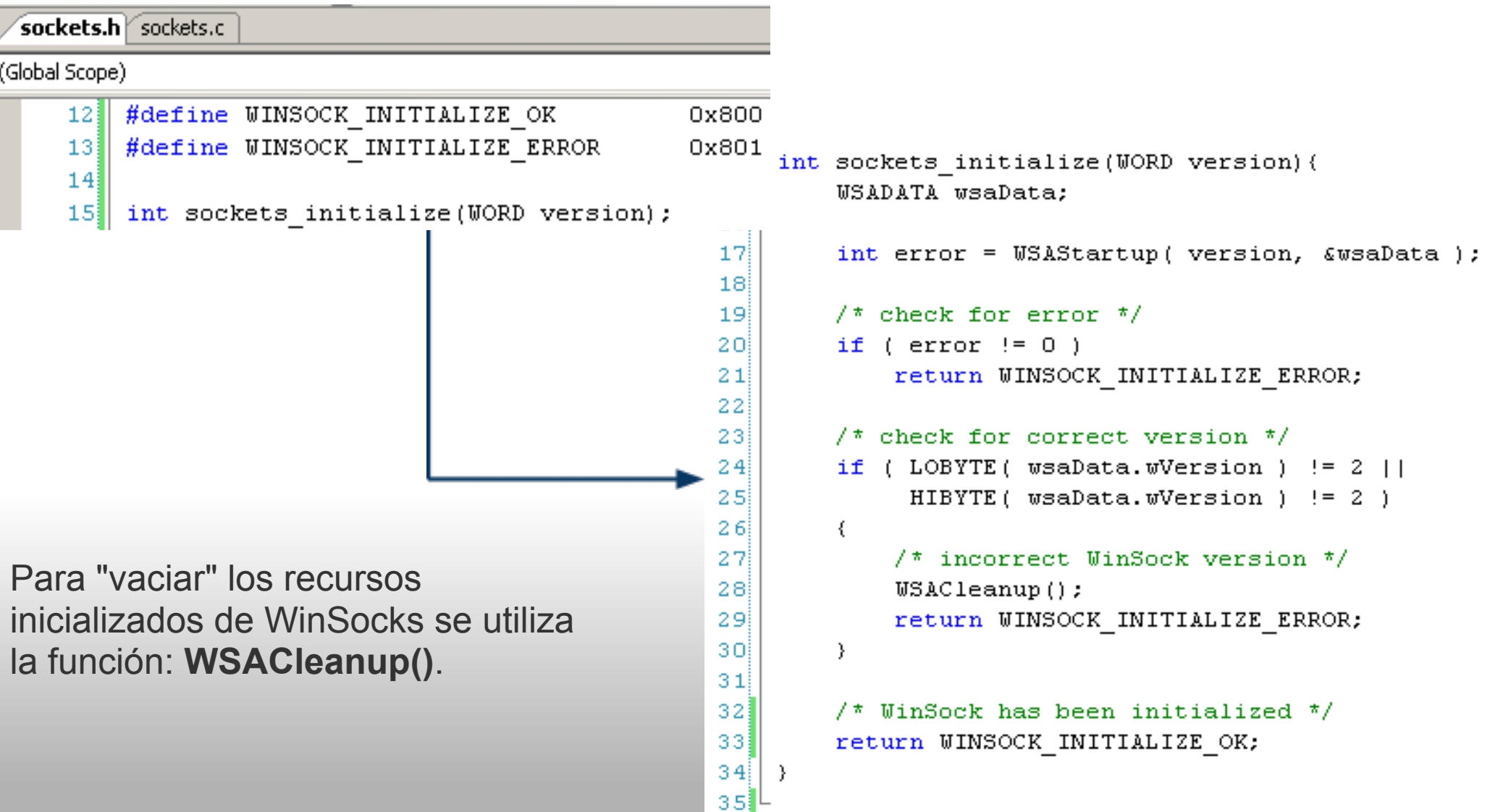
WinAPI:

<winsock2.h> : <http://msdn.microsoft.com/en-us/library/ms741394%28v=VS.85%29.aspx>

Sockets

Particularidad de la WinAPI - Inicializar y Finalizar biblioteca WinSocks2

La WinAPI para sockets (WinSocks) requiere que se inicialice **con qué versión** de la biblioteca se va a trabajar.



```
sockets.h sockets.c
(Global Scope)

12 #define WINSOCK_INITIALIZE_OK 0x800
13 #define WINSOCK_INITIALIZE_ERROR 0x801
14
15 int sockets_initialize(WORD version);

17
18
19 /* check for error */
20 if ( error != 0 )
21     return WINSOCK_INITIALIZE_ERROR;
22
23 /* check for correct version */
24 if ( LOBYTE( wsaData.wVersion ) != 2 ||
25     HIBYTE( wsaData.wVersion ) != 2 )
26 {
27     /* incorrect WinSock version */
28     WSACleanup();
29     return WINSOCK_INITIALIZE_ERROR;
30 }
31
32 /* WinSock has been initialized */
33 return WINSOCK_INITIALIZE_OK;
34 }
35
```

Para "vaciar" los recursos inicializados de WinSocks se utiliza la función: **WSACleanup()**.

Sockets

Particularidad de la WinAPI - Inicializar y Finalizar biblioteca WinSocks2

MUY IMPORTANTE:

- WinSocks se debe inicializar una **única vez por proceso**. Y **WSACleanup()** al final de la ejecución del mismo (o cuando se quiera forzar el fin de la ejecución. p.e.: un error esperado).
- *windows.h* internamente incluye al header *winsock.h*, es decir, a la versión anterior de la api, por lo cual fuerza al compilador a utilizar esa versión.
- Si se quiere utilizar la versión 2, *winsock2.h*, hay que seguir este orden de includes para que el compilador *descarte* el include anteriormente citado y no se generen errores de compilación.

```
9  #include <stdio.h>  /* printf */
10 #include <stdlib.h> /* perror  EXIT_FAILURE  EXIT_SUCCESS */
11
12 #include <winsock2.h>
13 #pragma comment(lib, "Ws2_32.lib")
14
15 #include <windows.h>
```

Directiva para el linker.
Le indica que "linkee con esta biblioteca"

Particularidad de la WinAPI - Inicializar y Finalizar biblioteca WinSocks2

Ejemplo:

```
1 #include <stdio.h> /* printf */
2 #include <stdlib.h> /* perror EXIT_FAILURE EXIT_SUCCESS */
3
4 #include <winsock2.h>
5 #pragma comment(lib, "Ws2_32.lib")
6
7 #include <windows.h>
8 #include "sockets.h"
9
10 int wmain(void) {
11     if( sockets_initialize(MAKEWORD(2, 2)) == WINSOCK_INITIALIZE_ERROR ){
12         perror("#No se pudo inicializar WinSocks.");
13         return EXIT_FAILURE;
14     }
15
16     /* Utilizar api WinSock2*/
17
18     WSACleanup();
19
20     return EXIT_SUCCESS;
21 }
```

Sockets Bloqueantes y No-Bloqueantes

Los sockets tiene 2 modalidades de funcionamiento Bloqueantes o No-Bloqueantes, siendo Bloqueantes el modo por defecto con el que se inician estos:

- *Bloqueantes*: Las funciones de los sockets quedan bloqueadas, es decir detienen el hilo de ejecución, a la espera de que algo las desbloquee. Por Ej:
 - La función `recv` queda bloqueada hasta que algo le llegue.
 - La función `accept` queda bloqueada hasta que una conexión nueva llegue

Para lidiar con el problema del bloqueo se suele trabajar con threads o con la función `select`.

- *No Bloqueante*: Las funciones de los sockets no bloquean el hilo de ejecución, si no hay información nueva la funciona no hace nada y el hilo de ejecución continua. Por Ej:
 - La función `recv` chequea si hay información para leer, si no la hay retorna -1 y se continua la ejecución del programa.
 - La función `accept` chequea si hay nuevas conexiones, si no las hay retorna -1 y se continua la ejecución del programa.

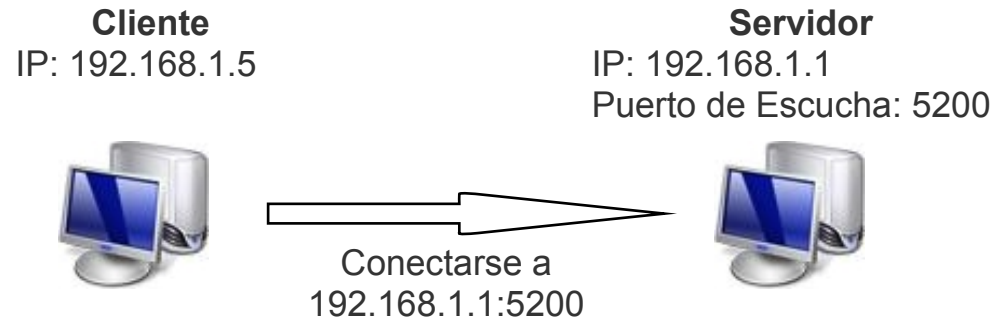
El ser no bloqueantes, puede llegar a generar mucho overhead si no son manejadas adecuadamente (Podría transformarse en una espera activa que consume mucho CPU).

La decisión de usar una modalidad u otra depende pura y exclusivamente de lo que se este desarrollando y las necesidades de la aplicación.

Sockets

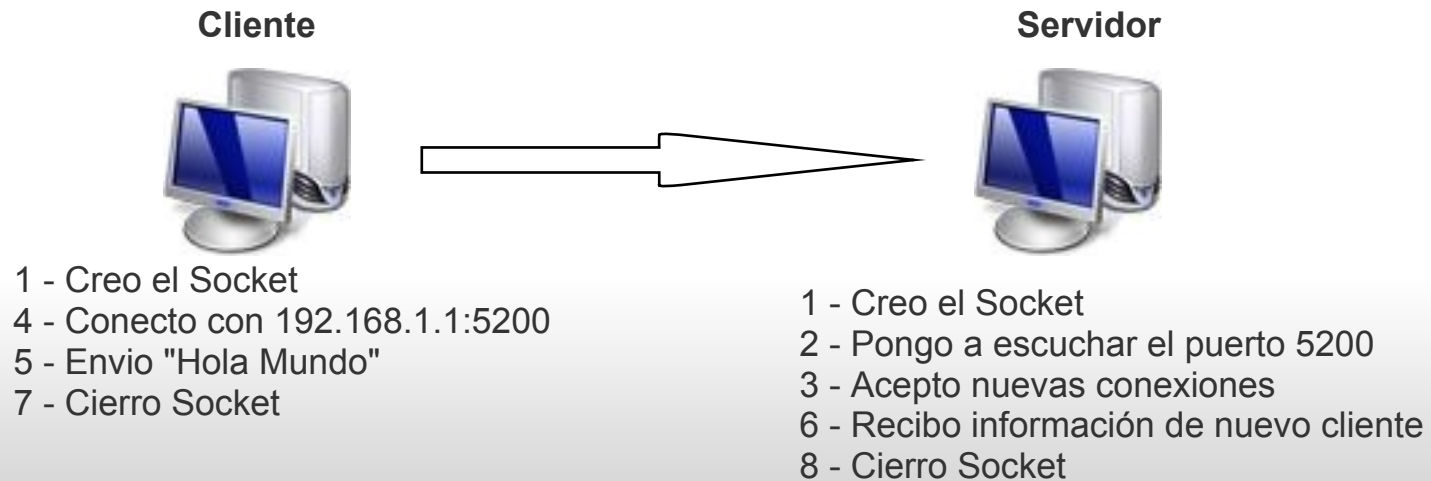
Arquitectura Cliente-Servidor

Es la forma a través de la cual funcionan los Sockets. Siempre existe "alguien" a la espera de conexiones y "alguien" que intenta conectarse.



Hay que tener en cuenta que un servidor puede recibir y aceptar conexiones de varios clientes.

Conexión mediante Sockets



Ejemplo en Código (Posix) de un Servidor:

```
11#include <stdlib.h>
12#include <string.h>
13#include <stdio.h>
14#include <sys/select.h>
15#include <sys/types.h>
16#include <sys/socket.h>
17#include <arpa/inet.h>
18#include <unistd.h>
19
20#define SOCKET_MAX_BUFFER 100
21
22int main(int argc, char **argv) {
23    int addrlen = sizeof(struct sockaddr_in);
24    char buffer[SOCKET_MAX_BUFFER];
25
26    /* Creo el Socket: SOCK_STREAM para TCP y SOCK_DGRAM par UDP */
27    int descriptor = socket(AF_INET, SOCK_STREAM, 0);
28    int remote_client;
29
30    /* Direccion Local */
31    struct sockaddr_in *local_address = malloc(sizeof(struct sockaddr_in));
32    /* Direccion remota ( a la que me quiero conectar )*/
33    struct sockaddr_in *remote_address = malloc(sizeof(struct sockaddr_in));
34
35    { /* Con esto fuerzo a que el puerto local sea el 5201 y que tome la IP por defecto de la PC */
36        local_address->sin_family = AF_INET;
37        local_address->sin_addr.s_addr = inet_addr("127.0.0.1");
38        local_address->sin_port = htons(5300);
39
40        bind(descriptor, (struct sockaddr *)local_address, sizeof(struct sockaddr_in));
41    }
42
43    /* Activo la escucha de conexiones entrantes a traves del puerto 5200 y como maximo 100 conexiones.*/
44    listen(descriptor, 100);
45
46    /* Acepto nueva conexion entrante */
47    remote_client = accept(descriptor, (struct sockaddr *)remote_address, (void *)&addrlen);
48
49    /* Leo e imprimo la informacion recibida a traves de la nueva conexion */
50    recv(remote_client, buffer, SOCKET_MAX_BUFFER, 0);
51    printf("%s", buffer);
52
53    /* Cierro el socket y por ende la conexion */
54    close(descriptor);
55
56    return EXIT_SUCCESS;
57}
```

Ejemplo en Código (Posix) de un Cliente:

```
11 #include <stdlib.h>
12 #include <string.h>
13 #include <sys/select.h>
14 #include <sys/types.h>
15 #include <sys/socket.h>
16 #include <arpa/inet.h>
17 #include <unistd.h>
18
19 int main(int argc, char **argv) {
20
21     /* Creo el Socket: SOCK_STREAM para TCP y SOCK_DGRAM para UDP */
22     int descriptor = socket(AF_INET, SOCK_STREAM, 0);
23
24     /* Direccion Local */
25     struct sockaddr_in *local_address = malloc(sizeof(struct sockaddr_in));
26     /* Direccion remota ( a la que me quiero conectar )*/
27     struct sockaddr_in *remote_address = malloc(sizeof(struct sockaddr_in));
28
29     { /* Con esto fuerzo a que el puerto local sea el 5201 y que tome la IP por defecto de la PC */
30         local_address->sin_family = AF_INET;
31         local_address->sin_addr.s_addr = INADDR_ANY;
32         local_address->sin_port = htons(5301);
33
34         bind(descriptor, (struct sockaddr *)local_address, sizeof(struct sockaddr_in));
35     }
36
37     { /* Con esto indico que me quiero conectar al puerto 5200 de la IP 127.0.0.1 (localhost) */
38         remote_address->sin_family = AF_INET;
39         remote_address->sin_addr.s_addr = inet_addr("127.0.0.1");
40         remote_address->sin_port = htons(5300);
41     }
42
43     /* Me conecto al servidor */
44     connect(descriptor, (struct sockaddr *)remote_address, sizeof(struct sockaddr_in));
45
46     /* Le envio un Hola Mundo! */
47     send(descriptor, "Hola Mundo!", strlen("Hola Mundo!") + 1, 0);
48
49     /* Cierro el socket y por ende la conexion */
50     close(descriptor);
51
52     return EXIT_SUCCESS;
53 }
```

Ejemplo en Código (WinSocks2) de un Servidor:

Snapshots en breves (cuando se deje de colgar mi VMWare Player =P).

Ejemplo en Código (WinSocks2) de un Cliente:

Snapshots en breves (cuando se deje de colgar mi VMWare Player =P).

Socket Unix

Un socket de dominio UNIX (UDS) o socket IPC (socket de comunicación interprocesos) es un socket virtual, similar a un socket de internet que se utiliza en los sistemas operativos POSIX para comunicación entre procesos. Estas conexiones aparecen como flujos de bytes, al igual que las conexiones de red, pero todos los datos se mantienen dentro de la computadora local. Los Sockets de dominio UNIX utiliza el sistema de archivos como su dirección espacio de nombres, es decir, son vistos por los procesos como archivos de un sistema de archivos. Esto permite que dos procesos distintos referenciar y abrir el mismo socket con el fin de comunicarse. Sin embargo, la comunicación real (el intercambio de datos) no utiliza el sistema de ficheros, sino buffers de memoria del núcleo.

Los beneficios de este tipo de socket son que generan menos context switch y tienen mucha mejor performance.

Socket Unix

```
#include <sys/un.h>

#define SOCK_PATH "echo_socket"

int main(void)
{
    int s, len;
    struct sockaddr_un local;

    if ((s = socket(AF_UNIX, SOCK_STREAM, 0)) == -1) {
        perror("socket");
        exit(1);
    }

    local.sun_family = AF_UNIX;
    strcpy(local.sun_path, SOCK_PATH);
    unlink(local.sun_path);
    len = strlen(local.sun_path) + sizeof(local.sun_family);
    if (bind(s, (struct sockaddr *)&local, len) == -1) {
        perror("bind");
        exit(1);
    }
}
```

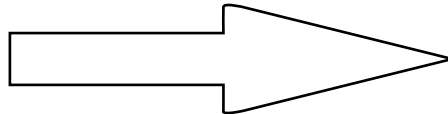
Sockets

Pensemos en TADs

Nuestro objetivo con el TAD es encapsular toda la información y variables utilizadas, bajo un solo concepto o tipo de dato.

Definimos sockets.
h

Nuestro Tipo de Dato Abstracto



```
20 #define DEFAULT_BUFFER_SIZE      2048
21 #define DEFAULT_MAX_CONEXIONS  100
22 #define SELECT_USEC_TIMEOUT     500
23
24 typedef enum {
25     SOCKETSTATE_CONNECTED,
26     SOCKETSTATE_DISCONNECTED
27 } e_socket_state;
28
29 typedef enum {
30     SOCKETMODE_NONBLOCK      = 1,
31     SOCKETMODE_BLOCK         = 2
32 } e_socket_mode;
33
34
35 typedef struct {
36     int desc;
37     struct sockaddr_in* my_addr;
38     e_socket_mode mode;
39 } t_socket ;
40
41 typedef struct {
42     t_socket* socket;
43     t_socket* serv_socket;
44     e_socket_state state;
45 } t_socket_client ;
46
47 typedef struct {
48     t_socket *socket;
49     int maxconexions;
50 } t_socket_server ;
51
52 typedef struct {
53     char data[DEFAULT_BUFFER_SIZE];
54     int size;
55 } t_socket_buffer ;
```

Funciones asociadas en TADs

Simplemente pensemos encapsular todo el código anteriormente usado, de una forma modular, genérica y reutilizable que abstraiga al usuario de la Biblioteca de como funcionan los sockets internamente a través de nuestro nuevo tipo de dato.

```
63 void sockets_bufferDestroy(t_socket_buffer *tbuffer);
64
65 void sockets_setMode(t_socket *sckt, e_socket_mode mode);
66 e_socket_mode sockets_getMode(t_socket *sckt);
67 int sockets_isBlocked(t_socket *sckt);
68
69 t_socket_client *sockets_createClient(char *ip, int port);
70 int sockets_isConnected(t_socket_client *client);
71 int sockets_equalsClients(t_socket_client *client1, t_socket_client *client2);
72 e_socket_state sockets_getState(t_socket_client *client);
73 void sockets_setState(t_socket_client *client, e_socket_state state);
74 int sockets_connect(t_socket_client *client, char *server_ip, int server_port);
75 int sockets_send(t_socket_client *client, void *data, int datalen);
76 int sockets_sendBuffer(t_socket_client *client, t_socket_buffer *buffer);
77 int sockets_sendString(t_socket_client *client, char *str);
78 t_socket_buffer *sockets_recv(t_socket_client *client);
79 int sockets_recvInBuffer(t_socket_client *client, t_socket_buffer *buffer);
80 void sockets_destroyClient(t_socket_client *client);
81
82 t_socket_server *sockets_createServer(char *ip, int port);
83 void sockets_setMaxConexions(t_socket_server* server, int conexions);
84 int sockets_getMaxConexions(t_socket_server* server);
85 int sockets_listen(t_socket_server* server);
86 t_socket_client *sockets_accept(t_socket_server* server);
87 void sockets_destroyServer(t_socket_server* server);
```

Ej: Cliente-Servidor con nuestro TAD

Cliente

```
7 #include <stdlib.h>
8 #include "clib/sockets.h"
9
10 int main(int argc, char **argv) {
11
12     t_socket_client *client = sockets_createClient("127.0.0.1", 5301);
13
14     sockets_connect(client, "127.0.0.1", 5300);
15
16     sockets_sendString(client, "Hola Mundo!");
17
18     sockets_destroyClient(client);
19
20     return EXIT_SUCCESS;
21 }
```

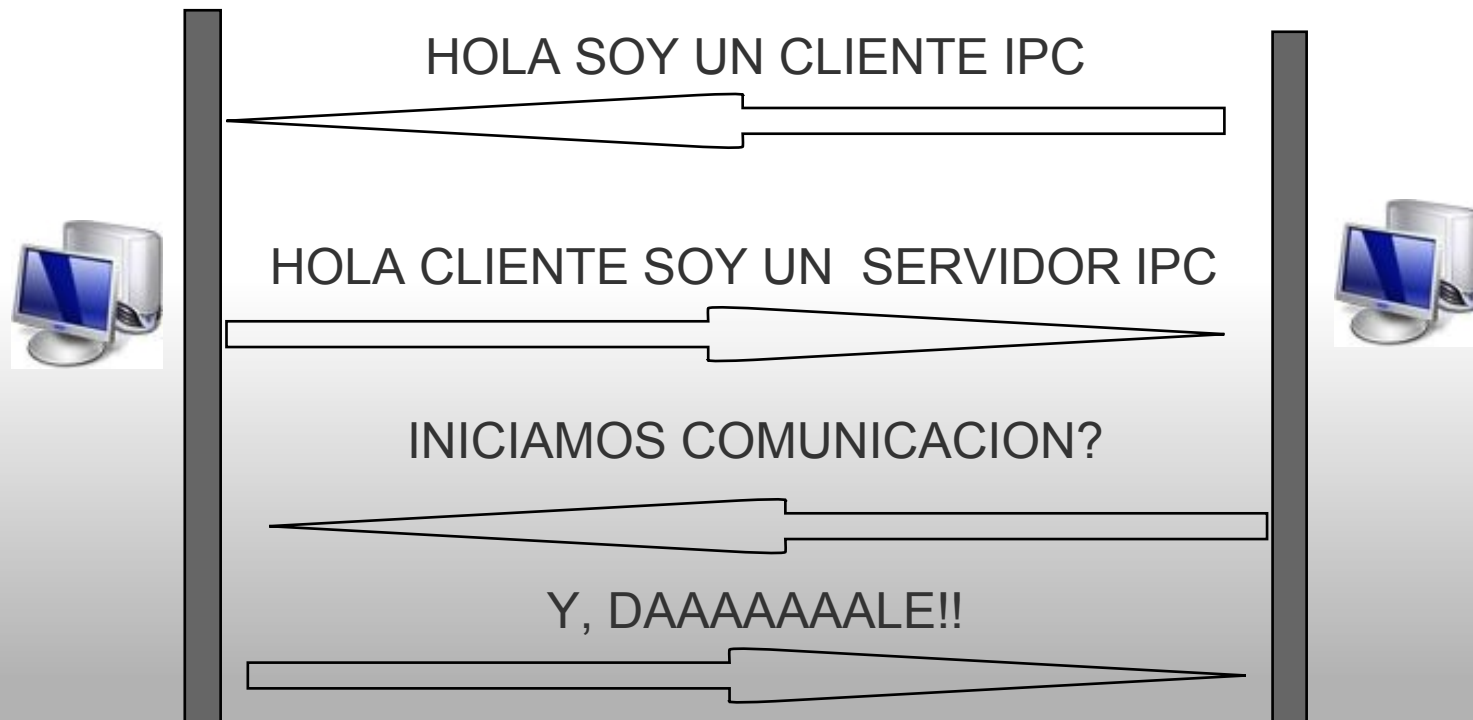
Servidor

```
8 #include <stdlib.h>
9 #include "clib/sockets.h"
10
11 int main(int argc, char **argv) {
12     t_socket_server *server = sockets_createServer("127.0.0.1", 5200);
13     t_socket_client *client;
14     t_socket_buffer *buffer;
15
16     sockets_listen(server);
17
18     client = sockets_accept(server);
19
20     buffer = sockets_recv(client);
21
22     printf("%s\n", buffer->data);
23
24     sockets_bufferDestroy(buffer);
25
26     sockets_destroyClient(client);
27
28     sockets_destroyServer(server);
29
30     return EXIT_SUCCESS;
31 }
```

Handshake

Handshaking ("apretón de manos") es un proceso automatizado de negociación que establece de forma dinámica los parámetros de un canal de comunicaciones establecido entre dos entidades antes de que comience la comunicación normal por el canal. Por lo general, un proceso que tiene lugar cuando un equipo está a punto de comunicarse con un dispositivo exterior a establecer normas para la comunicación.

Esto quiere decir que una vez establecida la conexión y antes de empezar con la comunicación normal, tanto el cliente como el servidor se identifican y negocian parámetros de comunicación.



Función select

Los sockets bloqueantes, nos fuerzan a utilizar threads si es que queremos realizar otras tareas mientras esperamos que algo ocurra en estos. Una solución single-thread para esto es utilizar la función select.

La función select recibe un conjunto de descriptores (para nuestro caso sockets) y opcionalmente un timeout.

Existen algunas funciones equivalentes select que son parte de Posix como es el caso de la función poll(). Este varia principalmente en su forma de manejarlo.

Fuera de Posix, Linux tiene su propia implementación llamada epoll() la cual es mucho mas rápida y eficiente que select o poll. Esta es usada por Nginx o lighttpd, que son servidores HTTP de alto rendimiento.

Función select

Los parámetros que recibe la función `select()` son los siguientes:

- **int** con el valor del descriptor más alto que queremos tratar más uno. Cada vez que abrimos un fichero, socket o similar, se nos da un descriptor de fichero que es entero. Estos descriptors suelen tener valores consecutivos. El 0 suele estar reservado para la `stdin`, el 1 para la `stdout`, el 2 para la `stderr` y a partir del 3 se nos irán asignando cada vez que abramos algún "fichero". Aquí debemos dar el valor más alto del descriptor que queramos pasar a la función más uno.
- **fd_set *** es un puntero a los descriptors de los que nos interesa saber si hay algún dato disponible para leer o que queremos que se nos avise cuando lo haya. También se nos avisará cuando haya un nuevo cliente o cuando un cliente cierre la conexión.
- **fd_set *** es un puntero a los descriptors de los que nos interesa saber si podemos escribir en ellos sin peligro. Si en el otro lado han cerrado la conexión e intentamos escribir, se nos enviará una señal `SIGPIPE` que hará que nuestro programa se caiga.
- **fd_set *** es un puntero a los descriptors de los que nos interesa saber si ha ocurrido alguna excepción.
- **struct timeval *** es el tiempo que queremos esperar como máximo. Si pasamos `NULL`, nos quedaremos bloqueados en la llamada a `select()` hasta que suceda algo en alguno de los descriptors. Se puede poner un tiempo cero si únicamente queremos saber si hay algo en algún descriptor, sin quedarnos bloqueados.

Función select

Estos `fd_set` son unos punteros un poco raros. Para rellenarlos y ver su contenido tenemos una serie de macros:

- **FD_ZERO (`fd_set *`)** nos vacía el puntero, de forma que estamos indicando que no nos interesa ningún descriptor de fichero.
- **FD_SET (`int`, `fd_set *`)** mete el descriptor que le pasamos en `int` al puntero `fd_set`. De esta forma estamos indicando que tenemos interés en ese descriptor. Llamando primero a `FD_ZERO()` para inicializar el contenido del puntero y luego a `FD_SET()` tantas veces como descriptors tengamos, ya tenemos la variable dispuesta para llamar a `select()`.
- **FD_ISSET (`int`, `fd_set *`)** nos indica si ha habido algo en el descriptor `int` dentro de `fd_set`. Cuando `select()` sale, debemos ir interrogando a todos los descriptors uno por uno con esta macro.
- **FD_CLEAR (`int`, `fd_set *`)** elimina el descriptor dentro del `fd_set`.

Sockets

Ejemplo:

```
14  #define MAX_NUMBER_OF_CLIENTS 10
15
16  fd_set read_descriptors;
17  int socket_server;
18  int socket_clients[MAX_NUMBER_OF_CLIENTS];
19
20
21
22  int max_desc = socket_server;
23
24  FD_ZERO (&read_descriptors);
25  FD_SET (socket_server, &read_descriptors);
26
27  for (i=0; i<MAX_NUMBER_OF_CLIENTS; i++){
28
29      FD_SET (socket_clients[i], &read_descriptors);
30
31      if( socket_clients[i] > max_desc)
32          max_desc = socket_clients[i];
33  }
34
35  select (max_desc+1, &read_descriptors, NULL, NULL, NULL);
36
37  /* Se trata los sockets clientes */
38  for (i=0; i<MAX_NUMBER_OF_CLIENTS; i++){
39
40      if (FD_ISSET (socket_clients[i], &read_descriptors)){
41          /* Si entramos aca es porque uno de los clientes recibio informacion */
42      }
43
44  }
45
46  /* Se trata el socket servidor */
47  if (FD_ISSET (socket_server, &read_descriptors)){
48      /* Si entramos aca es porque llego una nueva conexion */
49  }
```

Estos son los sockets con los que vamos a trabajar.

Luego de crear e inicializar los sockets meto los descriptores en el fd_set y llamo al select

Luego del select recorro el fd_set para verificar si llego una nueva conexión o si algún cliente recibió información

Consideraciones sobre el select

De acuerdo a las especificaciones de select, cualquiera de sus argumentos no esta seguro de ser modificado internamente. Esto quiere decir que una vez que pasemos un fd_set o un struct timeval *, estos tienen que volver a ser creados para ser utilizados en la función.

¿ Que es un Stream ?

Un stream es un flujo o secuencia de bytes enviados a través de un canal de comunicación. Los sockets envían la informas como streams a través de la red.

La función send de la librería de sockets recibe un puntero a un bloque de memoria y la longitud de este. Por lo que a fin de cuentas lee el contenido del bloque de memoria y lo envía tal cual por la red (obviamente encapsulado dentro del protocolo TCP/IP u UDP/IP).

Si solo enviamos cadenas la situación no es muy complicada, pero muchas veces requerimos de enviar información con un formato especifico el cual deja de ser una cadena.

Sockets

Nuestro Problema, Como enviar un TAD como un Stream?

Es decir, como transformar esto:

```
69     typedef struct {
70         unsigned int dni;
71         unsigned char edad;
72         char *nombre;
73         char *apellido;
74     } t_persona ;
```

en esto:

| | | | | | | | | |
|-------------|------------|-------------|-------------|-----------|------------|-----------|-----------|-----------|
| 0000 0000 | 0000 0000 | 0000 0000 | 0000 0001 | 0001 1000 | 0000 0100 | 0100 1010 | 0110 1111 | 0110 1000 |
| DNI [1byte] | DNI [byte] | DNI [3byte] | DNI [4byte] | Edad[24] | Long. Nom. | Nombre[J] | Nombre[o] | Nombre[h] |

| | | | | |
|------------|------------|--------------|--------------|--------------|
| 0110 1110 | 0000 0011 | 0100 0100 | 0110 1111 | 0100 0101 |
| Nombre [n] | Long. Ape. | Apellido [D] | Apellido [o] | Apellido [e] |

La respuesta es: **Serializando!**

Que es la Serializacion?

Consiste en un proceso de transformación de un Objeto(programación orientada a objetos) u Estructura que se encuentra en un medio de almacenamiento (como puede ser un archivo, o un buffer de memoria), con el fin de transmitirlo a través de una conexión en red como una serie de bytes. La serie de bytes o el formato pueden ser usados para crear un nuevo objeto u estructura que es idéntico en todo al original, en donde es recibido.

En lenguajes de alto nivel como Java, C#, Python, etc ... este proceso es alto transparente y automático para el programador. Para el caso de C, esta funcionalidad debe ser realizada manualmente por el programador usando funciones como memset, memcpy, etc ...

La estructuras estáticas son muy fáciles de serializar, mientras que las estructuras con varios campos dinámicos requieren un poco mas de trabajo.

Sockets

Serializacion de una Estructura Estática

Supongamos:

```
69  typedef struct {
70      unsigned int dni;
71      unsigned char edad;
72      char nombre[20];
73      char apellido[20];
74  } t_persona ;
```

t_persona unaPersona;

&unaPersona => 0x100

&unaPersona.dni => 0x100

&unaPersona.edad => 0x104

&unaPersona.nombre => 0x105

&unaPersona.apellido => 0x119

Las estructuras que contienen campos estáticos y son creadas estáticamente, en memoria se crean de manera contigua por lo que la estructura ya se encontraría "serializada". Bastaría con hacer un:

send(descriptor, &unaPersona, sizeof(t_persona));

| Direccion | Contenido | Longitud |
|-----------|-----------|----------|
| 0x100 | 1 | 4 byte |
| 0x104 | 24 | 1 byte |
| 0x105 | 'J' | 1 byte |
| 0x106 | 'o' | 1 byte |
| 0x107 | 'h' | 1byte |
| 0x108 | 'n' | 1 byte |
| 0x109 | '\0' | 1 byte |
| 0x10A | '\0' | 1byte |
| : | : | : |
| : | : | : |
| 0x119 | 'D' | 1 byte |
| 0x11A | 'o' | 1 byte |
| 0x11B | 'e' | 1 byte |
| 0x11C | '\0' | 1 byte |
| : | : | : |
| 0x12D | '\0' | 1 byte |

Serializacion de una Estructura Dinámica

```
12 typedef struct{
13     unsigned int dni;
14     unsigned char age;
15     char *nombre;
16     char *apellido;
17 } t_persona;
```

```
t_persona *unaPersona = malloc( sizeof(t_persona) );
unaPersona->nombre = "John";
unaPersona->apellido = "Doe";
unaPersona          => 0x100
&unaPersona->dni    => 0x100
&unaPersona->edad    => 0x104
unaPersona->nombre   => 0x200
unaPersona->apellido => 0x250
```

Como se puede ver la memoria esta dispersa y no es un bloque contiguo, es por eso que nosotros debemos manualmente transformarlo en un bloque contiguo.

| Direccion | Contenido | Longitud |
|-----------|-----------|----------|
| 0x100 | 1 | 4 bytes |
| 0x104 | 24 | 1 byte |
| 0x105 | 0x200 | 4 bytes |
| 0x106 | 0x250 | 4 bytes |
| : | : | : |
| : | : | : |
| 0x200 | 'J' | 1 byte |
| 0x201 | 'o' | 1 byte |
| 0x202 | 'h' | 1 byte |
| 0x203 | 'n' | 1 byte |
| 0x204 | '\0' | 1 byte |
| : | : | : |
| : | : | : |
| 0x250 | 'D' | 1 byte |
| 0x251 | 'o' | 1 byte |
| 0x252 | 'e' | 1 byte |
| 0x253 | '\0' | 1 byte |
| : | : | : |

Serializador

```
6 typedef struct {
7     int dni;
8     char *name;
9     char *lastname;
10 } t_person;
11
12 typedef struct {
13     int length;
14     char *data;
15 } t_stream;
16
17
18 t_stream *Person_serializer(t_person *self) {
19     char *data = malloc( sizeof(int) + strlen(self->name) + 1 + strlen(self->lastname) + 1);
20     t_stream *stream = malloc( sizeof(t_stream) );
21     int offset = 0, tmp_size = 0;
22
23     memcpy(data, &self->dni, tmp_size = sizeof(int));
24
25     offset = tmp_size;
26     memcpy(data + offset, self->name, tmp_size = strlen(self->name) + 1);
27
28     offset += tmp_size;
29     memcpy(data + offset, self->lastname, tmp_size = strlen(self->lastname) + 1);
30
31     stream->length = offset + tmp_size;
32     stream->data = data;
33
34     return stream;
35 }
```


Des-Serializador

```
37 t_person *Person_deserializer(t_stream *stream) {
38     t_person *self = malloc(sizeof(t_person));
39     int offset = 0, tmp_size = 0;
40
41     memcpy(&self->dni, stream->data, tmp_size = sizeof(int));
42
43     offset = tmp_size;
44     for (tmp_size = 1; (stream->data + offset)[tmp_size-1] != '\0'; tmp_size++);
45     self->name = malloc(tmp_size);
46     memcpy(self->name, stream->data + offset, tmp_size);
47
48     offset += tmp_size;
49     for (tmp_size = 1; (stream->data + offset)[tmp_size-1] != '\0'; tmp_size++);
50     self->lastname = malloc(tmp_size);
51     memcpy(self->lastname, stream->data + offset, tmp_size);
52
53     return self;
54 }
```

Serializador & Des-Serializador

La serializacion es muy fácil de testear y con suficiente practica implementarla no lleva mucho tiempo. Y nos garantiza forma apropiada y correcta de crear un Stream de información sin mucha lógica, al coste de un mayor dominio en cuanto al conocimiento de la memoria.

```
56 int main(void) {
57     t_person *juan1 = malloc(sizeof(t_person));
58     t_person *juan2;
59     t_stream *stream;
60
61     juan1->dni = 30234980;
62     juan1->name = malloc( sizeof("JUAN") );
63     strcpy(juan1->name, "JUAN");
64     juan1->lastname = malloc( sizeof("PEREZ") );
65     strcpy(juan1->lastname, "PEREZ");
66
67     stream = Person_serializer(juan1);
68
69     juan2 = Person_deserializer(stream);
70
71     assert( juan1->dni == juan2->dni );
72     assert( strcmp(juan1->name, juan2->name) == 0 );
73     assert( strcmp(juan1->lastname, juan2->lastname) == 0 );
74
75     return EXIT_SUCCESS;
76 }
```

Sockets

Consideración: Directiva `#pragma pack(1)` & `'__attribute__((__packed__))'`

Hasta ahora hemos visto como es que las estructuras son almacenadas en memoria, pero hay algo a destacar. No es del todo cierto!

Como bien mencionamos el compilador es encargado de transformar un struct de código a un formato de bytes dispuestos de tal manera que cumplan con esta estructura. El problema es que el compilador puede llegar a optar, por agregar mas bytes de los correspondientes por cuestiones internas. *Esto lleva a que una estructura pueda resultar mas grande de lo que realmente es!*

Para evitar esto simplemente agregamos la directiva `#pragma pack(1)`, esto le especifica al compilador que no agregue bytes de mas a la estructura. Esta directiva es propia de Windows pero actualmente es soportada por GCC también.

```
10 #pragma pack(1)
11
12 typedef struct {
13     unsigned int dni;
14     unsigned char age;
15     char name[20];
16     char lastname[20];
17 } t_persona;
18
19 t_persona a = {20000000, 24, "John", "Doe"};
```

Sockets

La directiva ' __attribute__ ((__packed__))'

La directiva propia de GCC es `__attribute__ ((__packed__))` la cual debe ser colocada en cada una de las estructuras que deseamos, con esto le especifica al compilador que para esta estructura en particular no agregue los bytes extra.

```
11 typedef struct {
12     unsigned int dni;
13     unsigned char age;
14     char name[20];
15     char lastname[20];
16 } t_persona_unpack;
17
18 t_persona_unpack a = {2000000, 24, "John", "Doe"};
```

Codigo en ASM

a:

```
.long 2000000
.byte 24
.string "John"
.zero 15
.string "Doe"
.zero 16
.zero 3
```

`sizeof(t_persona_unpack) = 48`

```
20 typedef struct {
21     unsigned int dni;
22     unsigned char age;
23     char name[20];
24     char lastname[20];
25 } __attribute__ ((__packed__)) t_persona_pack;
26
27 t_persona_pack b = {2000000, 24, "John", "Doe"};
```

Codigo en ASM

b:

```
.long 2000000
.byte 24
.string "John"
.zero 15
.string "Doe"
.zero 16
```

`sizeof(t_persona_pack) = 45`

Como se puede observar el compilador para la estructura `t_persona_unpack` agrega 3 bytes mas. Estos bytes extra vienen el campo edad que es un char y ocupa 1 solo byte por lo que internamente decide agregar mas para completar la longitud de un int.

Consideraciones Finales

- El enviar estructuras no es del todo seguro, ya que si compilamos con 64 o 32 bits el tamaño de algunos tipos de dato varían. Como solución, se podría hacer un memcpy mas manual donde el tamaño de los tipos de datos sea siempre fijo. Tambien se podria usar en vez de int el tipo int32_t;
- Si bien acá dimos ejemplos para t_persona, es común que tengamos que mandar diferentes tipos de estructuras. Por lo que deberíamos diseñar un protocolo que primero tenga un campo el cual indique el tipo de estructura y a continuación este la estructura.

Bibliografía utilizada

- http://en.wikipedia.org/wiki/Thread_%28computer_science%29
- <http://www.yolinux.com/TUTORIALS/LinuxTutorialPosixThreads.html#CREATIONTERMINATION>
- http://opengroup.org/onlinepubs/007908799/xsh/pthread_create.html
- http://opengroup.org/onlinepubs/007908799/xsh/pthread_join.html
- <http://es.wikipedia.org/wiki/Mutex>
- http://es.wikipedia.org/wiki/Sem%C3%A1foro_%28inform%C3%A1tica%29
- http://sourceware.org/pthreads-win32/manual/pthread_mutex_init.html
- http://sourceware.org/pthreads-win32/manual/sem_init.html
- <http://www.chuidiang.com/clinix/sockets/socketselect.php>
- <http://en.wikipedia.org/wiki/Epoll>
- <http://sig9.com/articles/gcc-packed-structures>
- <http://gcc.gnu.org/onlinedocs/gcc/Type-Attributes.html>
- http://es.wikipedia.org/wiki/Interfaz_de_programaci%C3%B3n_de_aplicaciones
- http://en.wikipedia.org/wiki/Dynamic_memory_allocation
- **man ulimit**
- <http://uw713doc.sco.com/en/man/html.2/mmap.2.html>