
 UAB Universitat Autònoma de Barcelona	Sistemas Operativos	
Curso academico: 2015-2016		Código asignatura: 102747
Práctica: Threads		Centro: Escola d'Enginyeria

OBJETIVO

- Conocer el uso y funcionalidad de los threads, mutex y variables de condición.
- Razonar sobre problemas básicos de sincronización en programas concurrentes/paralelos.
- Conocer, desarrollar e implementar un patrón de sincronización Productor-Consumidor.
- Adquirir experiencia en la programación de POSIX threads y otros recursos de la librería Pthreads. Así como conocer las potencialidades de la API de Pthreads.
- Desarrollo de una pequeña aplicación en C que haga uso de las funcionalidades de la librería POSIX threads.

MATERIAL

- Comando **man** de unix: man [-s <sección>] <comando>
- Apuntes en el Campus Virtual
- Tutoriales/Manuales/Información en internet
 - POSIX Threads Programming. <https://computing.llnl.gov/tutorials/pthreads/>
 - Programming with POSIX Threads. David R. Butenhof. Addison-Wesley

EVALUACIÓN

Esta práctica se evalúa en las siguientes secciones:

- A. Trabajo previo (Sesión 1)
- B. Trabajo durante la sesión (Sesión 1)
- C. Trabajo previo (Sesión 2)
- D. Trabajo durante la sesión (Sesión 2)
- E. Trabajo fuera de la sesión e **informe de prácticas**

La suma de la nota obtenida en estas cinco secciones compone la nota final de la práctica 3. En total, el peso de esta práctica supone un tercio de la nota final de prácticas de la asignatura. El alumno debe tener bien presente que **el trabajo previo será evaluado durante la primera parte de cada sesión**, por ello es indispensable su elaboración previa a la práctica.

DESCRIPCIÓN

Se desea desarrollar una pequeña aplicación para simular el registro de peticiones de compra. Para ello, se implementará un patrón **productor-consumidor** donde los productores (clientes) enviarán las peticiones de compra al consumidor (servidor) a través de un buffer circular. La aplicación creará un thread-POSIX por cada cliente y servidor.

A cada **cliente** se le asignará un identificador numérico entre <1> y <NumClientes>. Cada cliente generará un total de 100 de peticiones de compra (ID de compra entre <1> ... <100>).

```
typedef struct {
    int id_cliente; // Identificador de cliente {1..N}
    int id_compra;  // Identificador de compra {1..100}
} peticion_t;

...

peticion_t mi_peticion;
mi_peticion.id_cliente = mi_id;
mi_peticion.id_compra = rand();
```

Se implementará un **buffer circular** de peticiones como un array estático. El buffer tendrá una capacidad máxima de 5 peticiones. Cada cliente almacenará su petición en el buffer de peticiones (en memoria compartida) siempre que el buffer no este lleno. En caso contrario, el cliente tendrá que esperar hasta que se produzca un hueco libre en el buffer.

```
#define TAM_BUFFER 5

typedef struct {
    int pos_lectura;    // Siguiete posicion a leer
    int pos_escritura;  // Siguiete posicion a escribir
    int num_peticiones; // Número de peticiones en el buffer
    peticion_t peticiones[TAM_BUFFER]; // Buffer de peticiones (circular)
} buffer_peticiones_t;
```

A su vez, el **servidor** leerá las peticiones pendientes del buffer y las almacenará en orden de llegada (FIFO) en el fichero 'compras.txt' como se muestra a continuación. En caso de estar el buffer vacío, el servidor tendrá que esperar hasta que se realice una petición. Cuando todos los clientes hayan realizado todas sus peticiones, el thread master realizará una petición con código <0,0> para que thread servidor finalice su ejecución.

```
> cat compras.txt

Cliente-1 1
Cliente-1 2
Cliente-2 1
Cliente-1 3
```

A. TRABAJO PREVIO (Sesión 1)

1. Implementar 1-thread cliente y 1-thread servidor

Como trabajo previo a la primera sesión, se desea que el alumno implemente un primer programa que cree 2 threads (un cliente y un servidor). Cada thread deberá ejecutar una función distinta que muestre por pantalla:

- Rol del thread (productor o consumidor)
- El ID asignado/enviado por el master thread
- El TID asignado por el SO (gettid()) y el PID del proceso (getpid())

[Hints] Para la realización de este apartado sera útil conocer:

- Creación de threads POSIX
- Funciones gettid() y getpid()
- Anexo. Ejemplo 1 y Ejemplo 2

B. TRABAJO DURANTE LA SESIÓN (Sesión 1)

2. Implementar el buffer de peticiones

Se deberá implementar las funciones del TAD buffer circular de peticiones. A continuación se presenta la interfaz del TAD buffer con el prototipo de las funciones a implementar.

```
// Inicializa la estructura del buffer y sus centinelas @pos_lectura y @pos_escritura
void buffer_peticiones_inicializar(buffer_peticiones_t* buffer_peticiones);

// Devuelve 1 si el buffer esta completamente lleno, 0 en caso contrario
int buffer_peticiones_lleno(buffer_peticiones_t* buffer_peticiones);

// Devuelve 1 si el buffer esta completamente vacio, 0 en caso contrario
int buffer_peticiones_vacio(buffer_peticiones_t* buffer_peticiones);

// Encola @peticion en el buffer
void buffer_peticiones_encolar(
    buffer_peticiones_t* buffer_peticiones, peticion_t* peticion);

// Retira del buffer la peticion mas antigua (FIFO) y la copia en @peticion
void buffer_peticiones_atender(
    buffer_peticiones_t* buffer_peticiones, peticion_t* peticion);
```

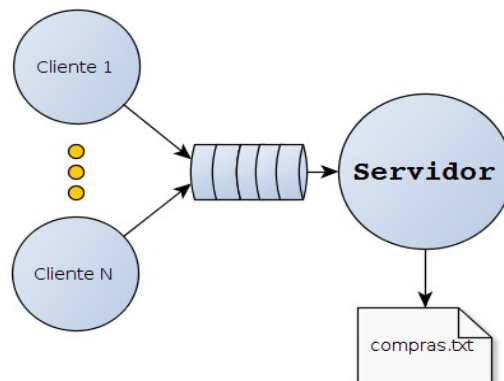
[Hints] Para la realización de este apartado sera útil:

- Programación fluida en C
- Se recomienda testear la funcionalidad del buffer antes de proceder a implementar sucesivos apartados de la práctica. Es conveniente cerciorarse de que cualquier error que pueda surgir no es debido a una incorrecta implementación de las funciones del buffer.

3. Productor-Consumidor N-1 (utilizando mutex)

Extender el programa anterior para la creación de N threads 'Clientes' y 1 thread 'Servidor'. La aplicación permitirá introducir por línea de comandos el número de clientes que se crearán. Cada **cliente (productor)** generará un total de 100 peticiones de compra (ID entre 1 y 100) y las encolará en el buffer (siempre que no este lleno). Por su lado, el **servidor (consumidor)** atenderá las peticiones del buffer (mientras queden peticiones sin atender en el buffer) y guardará todas las peticiones en el fichero 'compras.txt' (en orden de llegada; FIFO).

En el contexto de este problema, consideramos que el buffer de peticiones es un **recurso compartido** por todos los threads. Se desea implementar un código **libre condiciones de carrera** que puedan amenazar la coherencia del estado del buffer. Para garantizar un acceso controlado y correcto al buffer se utilizaran variable(s) mutex. De esta forma, todas las operaciones sobre el buffer se realizarán en exclusión mutua entre threads.



[Hints] Para la realización de este apartado sera útil conocer:

- Uso de mutex POSIX
- Anexo. Ejemplo 4

C. TRABAJO PREVIO (Sesión 2)

4. Variables de condición (POSIX Threads)

Como trabajo previo a la segunda sesión, será obligatorio documentarse sobre el uso y funcionamiento de las variables de condición de la librería POSIX Threads (Condition Variables). Se propone copiar, compilar y ejecutar el '**Ejemplo 5**' de este guión. Es imprescindible entender el funcionamiento del ejemplo con el fin de responder a las siguientes cuestiones.

4.1 Cuestión 1. Explicar de manera razonada el propósito del bucle while en la línea 18. ¿Sería válido sustituir el 'while' por un condicional 'if'? ¿Por qué?

```
17. pthread_mutex_lock(&mutex);
18. while (tickets==0) {
19.     pthread_cond_wait(&tickets_ready,&mutex);
20. }
21. --tickets;
22. pthread_mutex_unlock(&mutex);
```

4.2 Cuestión 2. El código del 'Ejemplo 5' no siempre termina (compruébalo!). Explica e implementa las modificaciones necesarias para evitar la situación de **deadlock**.

[Hints] Para la realización de este apartado será útil conocer:

- Anexo. Ejemplo 5
- **POSIX Threads Programming.** <https://computing.lln.gov/tutorials/pthreads/>

D. TRABAJO DURANTE LA SESIÓN (Sesión 2)

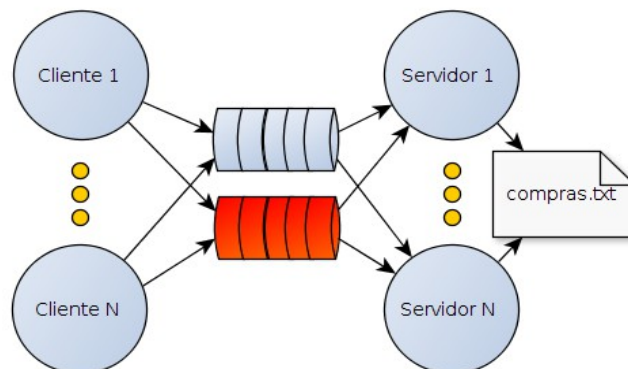
5. Productor-Consumidor N-1 (utilizando variables de condición)

Extender el programa del punto 4 utilizando variables de condición para el correcto acceso al buffer de peticiones y así evitar posibles **esperas activas**.

6. [Extra] Productor-Consumidor N-N con 1 buffers extra con alta prioridad

Como apartado extra, se plantea la extensión del apartado anterior para múltiples servidores (consumidores) y la implementación de un segundo buffer de alta prioridad. De forma similar a los apartados anteriores, los clientes encolarán aleatoriamente sus peticiones en cualquiera de las dos colas. No obstante, los servidores darán prioridad absoluta a las peticiones en la cola de alta prioridad. Solo atenderán peticiones normales en caso de que la cola de alta prioridad este vacía.

Destacar que se desea una implementación **sin esperas activas** (se debe descartar cualquier solución en la que un servidor este continuamente consultando el estado de ambas colas activamente).



ANEXO. EJEMPLOS

1. Creación de POSIX-threads

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

#define NUM_THREADS 5

// (RAW) void* pthread_function(void* thread_number)
void* pthread_function(int thread_id) {
    printf("I'm thread number %d\n", thread_id);
    pthread_exit(NULL);
}

int main(int argc, char** argv) {
    // Create threads
    long int i;
    pthread_t thread[NUM_THREADS];
    for (i=0; i<NUM_THREADS; ++i) {
        // (RAW) pthread_create(thread+i, NULL, pthread_function, i);
        pthread_create(thread+i, NULL, (void* (*)(void*))pthread_function, (void*)(i));
    }
    // Wait for threads
    for (i=0; i<NUM_THREADS; ++i) {
        pthread_join(thread[i], NULL);
    }
    return 0;
}
```

2. Paso de parametros y devolución de resultados de un thread a través de la API

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

#define NUM_THREADS 5

void* pthread_function(int thread_id) {
    printf("I'm thread number %d\n", thread_id);
    printf("Thread number %d is about to exit\n", thread_id);
    // Return value
    int *return_value = malloc(sizeof(int));
    *return_value = 5;
    pthread_exit(return_value);
}

int main(int argc, char** argv) {
    // Create threads
    long int i;
    pthread_t thread[NUM_THREADS];
    for (i=0; i<NUM_THREADS; ++i) {
        pthread_create(thread+i, NULL, (void* (*)(void*))pthread_function, (void*)(i));
    }
    // Wait for threads
    int* exit_status;
    for (i=0; i<NUM_THREADS; ++i) {
        pthread_join(thread[i], (void**)&exit_status);
        printf("Thread exited with %d code\n", *exit_status);
        free(exit_status);
    }
    return 0;
}
```

3. Comunicación entre threads a través de memoria compartida

```
#define NUM_THREADS    3

typedef struct {
    int in;
    int out;
} thread_data_t;

pthread_t thread[NUM_THREADS];
thread_data_t thread_data[NUM_THREADS];

void* pthread_function(int thread_id) {
    // Return value
    thread_data[thread_id].out = 2*thread_data[thread_id].in;
    pthread_exit(NULL);
}

int main(int argc, char** argv) {
    // Create threads
    srand(time(0));
    long int i;
    for (i=0; i<NUM_THREADS; ++i) {
        thread_data[i].in = rand()%10;
        pthread_create(&thread[i], NULL, (void*) (*)(void*) pthread_function, (void*)(i));
    }
    // Wait for threads
    for (i=0; i<NUM_THREADS; ++i) {
        pthread_join(thread[i], NULL);
        printf("Thread %ld exited with data %d -> %d\n",
            i, thread_data[i].in, thread_data[i].out);
    }
    return 0;
}
```

4. POSIX Mutex (Mutual exclusion)

```
#define NUM_THREADS    5

// Mutex
pthread_mutex_t mutex;

void* pthread_function(int thread_id) {
    pthread_mutex_lock(&mutex);
    printf("I'm thread number %d in mutual exclusion\n", thread_id);
    pthread_mutex_unlock(&mutex);
    pthread_exit(NULL);
}

int main(int argc, char** argv) {
    // Init mutex
    pthread_mutex_init(&mutex, NULL);
    // Create threads
    pthread_t thread[NUM_THREADS];
    long int i;
    for (i=0; i<NUM_THREADS; ++i) {
        pthread_create(&thread[i], NULL, (void*) (*)(void*) pthread_function, (void*)(i));
    }
    // Wait for threads
    for (i=0; i<NUM_THREADS; ++i) {
        pthread_join(thread[i], NULL);
    }
    return 0;
}
```

5. POSIX Condition Variable (EJEMPLO CON DEADLOCK)

Acceso a una región crítica de programa (como máximo 2 threads a la vez). El programa implementa el acceso a una región crítica pidiendo un 'ticket' y devolviéndolo una vez finalizado el trabajo.

```
23. #include <pthread.h>
24. #include <stdio.h>
25. #include <unistd.h>
26.
27. #define NUM_THREADS      5
28.
29. // Mutex & CV
30. pthread_mutex_t mutex;
31. pthread_cond_t tickets_ready;
32.
33. // Global counter
34. int tickets = 2;
35.
36. // Thread function
37. void* pthread_function(int thread_id) {
38.     // IN (Get ticket)
39.     pthread_mutex_lock(&mutex);
40.     while (tickets==0) {
41.         pthread_cond_wait(&tickets_ready,&mutex);
42.     }
43.     --tickets;
44.     pthread_mutex_unlock(&mutex);
45.     // DO STUFF ...
46.     printf("Thread %d working\n",thread_id);
47.     sleep(1);
48.     // OUT (Release ticket)
49.     pthread_mutex_lock(&mutex);
50.     if (tickets==0) {
51.         pthread_cond_signal(&tickets_ready);
52.     }
53.     ++tickets;
54.     pthread_mutex_unlock(&mutex);
55.     pthread_exit(NULL); // Exit
56. }
57.
58. // Main
59. int main(int argc,char** argv) {
60.     // Init mutex & CV
61.     pthread_mutex_init(&mutex,NULL);
62.     pthread_cond_init(&tickets_ready,NULL);
63.     // Create threads
64.     long int i;
65.     pthread_t thread[NUM_THREADS];
66.     for (i=0;i<NUM_THREADS;++i) {
67.         pthread_create(&thread[i],NULL,
68.             (void* (*)(void*))pthread_function,(void*)(i));
69.     }
70.     // Wait for threads
71.     for (i=0;i<NUM_THREADS;++i) {
72.         pthread_join(thread[i],NULL);
73.     }
74.     pthread_mutex_destroy(&mutex);
75.     pthread_cond_destroy(&tickets_ready);
76.     return 0;
77. }
```