



SISTEMAS OPERATIVOS

Grado en Desarrollo de Videojuegos
Universidad Complutense de Madrid

TEMA 5. Threads y Sincronización

Tema 5. Threads y Sincronización

5.1 Introducción

- Concurrencia y Paralelismo
- Sincronización y Comunicación

5.2 Procesos y Threads

- Threads
- Creación y terminación

5.3 Región Crítica

- Definición
- `pthread_mutex_t`
- Patrón de diseño: *Work Crew*

5.4 Variables de Condición

- Definición
- `p_thread_cont_t`
- Patrón de diseño: *Productor-Consumidor*

5.5 Soporte del Sistema

- Visibilidad de memoria
- Soporte de la CPU - ISA
- Soporte del SO



SISTEMAS OPERATIVOS

Grado en Desarrollo de Videojuegos
Universidad Complutense de Madrid

TEMA 5.1 Introducción

Concurrencia y Paralelismo

Concurrencia.

- Permite la compartición de recursos (p.ej. CPU) entre procesos (o threads) mientras no está siendo usado por el proceso actual (p.ej. E/S).
- Mecanismos
 - **Multiprogramación** (Tema 3)
 - **Programación multi-thread** (Tema 5)
 - Programación asíncrona (p.ej. corutinas C++, channels Golang, async/await Rust...)

Paralelismo. Uso simultáneo de los recursos del sistema (p.ej. cores de una CPU).

- Asociado al concepto de tiempo-compartido estudiado en la asignatura.
- Usa los mismos mecanismos que la concurrencia

Sincronización y Comunicación (I)

Sincronización. Coordinar la ejecución de varios procesos o threads

- **Mismo sistema**

- **Señales** (Tema 3)
- Ficheros, mediante el uso de cerrojos, p.ej. flock(2)
- **Mutex y variables de condición** (hilos de un proceso) (Tema 5)
- Semáforos (System V IPC, POSIX IPC)
- Colas de mensajes (System V IPC, POSIX IPC)
- Basados en UNIX domain socket(7) (Asignatura Redes)

- **Distintos sistemas**

- Basados en socket(7) TCP/UDP (Asignatura Redes)

Sincronización y Comunicación (II)

Comunicación. Compartir datos entre varios procesos o threads

- **Mismo sistema**

- **Threads** (hilos de un proceso) (Tema 5)
- **Memoria compartida** (System V IPC, POSIX IPC, Tema 4)
- **Tuberías con y sin nombre** (Tema 1)
- Colas de mensajes (System V IPC, POSIX IPC)
- **Basados en ficheros** (Tema 2)
- Basados en UNIX domain socket(7) (Asignatura Redes)

- **Distintos sistemas**

- Basados en socket(7) TCP/UDP (Asignatura Redes)



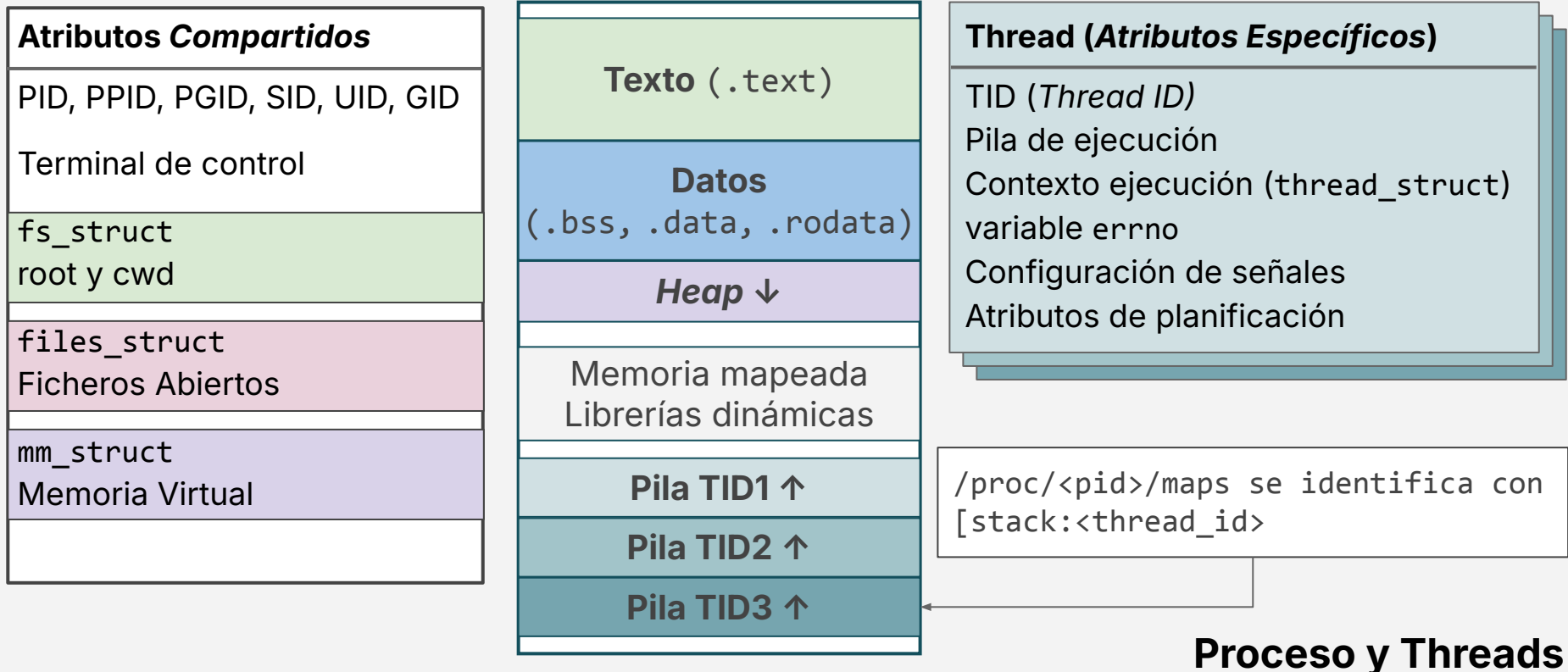
SISTEMAS OPERATIVOS

Grado en Desarrollo de Videojuegos
Universidad Complutense de Madrid

TEMA 5.2 Procesos y Threads

Threads (I)

- **Proceso** Unidad de ejecución con su espacio de memoria virtual y recursos asociados (tabla de ficheros, gestión de señales...)
- **Thread** Es un “*proceso ligero*” en el contexto de un proceso con el comparte algunos recursos (**especialmente el espacio de direcciones**) pero mantiene su propio contexto de ejecución.



Threads (II)

Aplicaciones multiproceso y multithread

- Los programas multi-thread ofrecen
 - Menor coste de creación
 - Cambios de contexto más eficientes
 - Modelo de compartición sencillo
 - Un mecanismo eficaz para aprovechar el **paralelismo** dentro de un proceso, especialmente para aplicaciones intensivas en CPU.
- Sin embargo:
 - Requieren de mecanismos de sincronización
 - Deben usar versiones *thread-safe* o *reentrantes* (*_r*) de las funciones de librería.
 - Difícil desarrollo y depuración (interbloqueos, condiciones de carrera...)
 - La programación asíncrona es más eficiente para aplicaciones intensivas en E/S (**concurrency**).

Threads (III)

Linux & POSIX Threads

- Procesos y Threads se representan por el mismo objeto del kernel `task_struct`.
- Se crean con la misma llamada `clone(2)` especificando los recursos que se comparten (ver `CLONE_THREAD`)
- Todos los threads de un proceso tienen un identificador único (TID), coincide con el PID de `task_struct` y tienen el mismo **Thread Group ID** (TGID).
- Para el thread principal `TID = TGID`
- TGID es el identificador mostrado por `ps` como PID
- **API POSIX Threads** define un conjunto de interfaces de programación (POSIX.1) para aplicaciones multi-thread.

pthreads(7)

```
$ ps -aL -o pid,tid,tgid,pgid,cmd
  PID   TID   TGID  PGID CMD
15607 15607 15607 15607 ./prod_cons 1 3 ← main thread
15607 15608 15607 15607 ./prod_cons 1 3
15607 15609 15607 15607 ./prod_cons 1 3
15607 15610 15607 15607 ./prod_cons 1 3
```

Creación y Terminación (I)

<pthread.h>

```
int pthread_create(pthread_t *thread,  
                  const pthread_attr_t *attr,  
                  typeof(void *(void *)) *start,  
                  void *arg);
```

- **pthread_t** Tipo de datos que representa al thread en el proceso.
 - Además de en la creación se puede obtener con `pthread_self(3)`
 - Tipo *opaco*, usar `pthread_equal(3)` para comparar
- **start** Función ejecutada por el thread `void *(void *)`. El thread terminará:
 - Retorne de la función `start`
 - Llame a la función `pthread_exit(3)`
 - Sea cancelado por otro thread `pthread_cancel(3)` (**no usar** en general)
 - El proceso termina (de cualquiera de las formas habituales) que terminará todos los threads del proceso.
- **arg** Puntero a los argumentos. **Importante:** El tiempo de vida debe ser igual al del thread.

Creación y Terminación (II)

<pthread.h>

Atributos de creación del thread `pthread_attr_t`

- Permiten fijar características sobre la planificación y **terminación** (ver `pthread_attr_init(3)`).
- **PTHREAD_CREATE_JOINABLE** (defecto). Permite la sincronización por otros threads.
 - `pthread_join(3)` espera la terminación de un thread y devuelve el valor de retorno.
 - Los recursos son destruidos cuando el thread se sincroniza.
- **PTHREAD_CREATE_DETACHED**
 - Cuando no es necesario controlar los threads creados.
 - Los recursos son destruidos cuando el thread termina.

Creación y Terminación (III)

<pthread.h>

```
int pthread_join(pthread_t thread, void **retval);
```

- `retval` dirección de la variable de retorno:
 - El ciclo de vida de la variable debe superar la vida del thread
 - El valor se fija mediante la función `pthread_exit(3)`
- `thread` especifica el thread que se espera
 - Cualquier thread puede realizar la sincronización de otro
 - No se puede esperar a "cualquier thread" como en `waitpid(2)`

Creación y Terminación (IV)

Ejemplo. Ejercicio 1

Escribir un programa que cree un número de threads indicado por el primer argumento, de forma que:

- Cada thread se le asignará un identificador 0,1,2... que imprimirá por la salida estándar y usará para hacer un `sleep(3)` de los mismos segundos.
- El thread principal esperará a que terminen todos los threads.



SISTEMAS OPERATIVOS

Grado en Desarrollo de Videojuegos
Universidad Complutense de Madrid

TEMA 5.3 Región Crítica

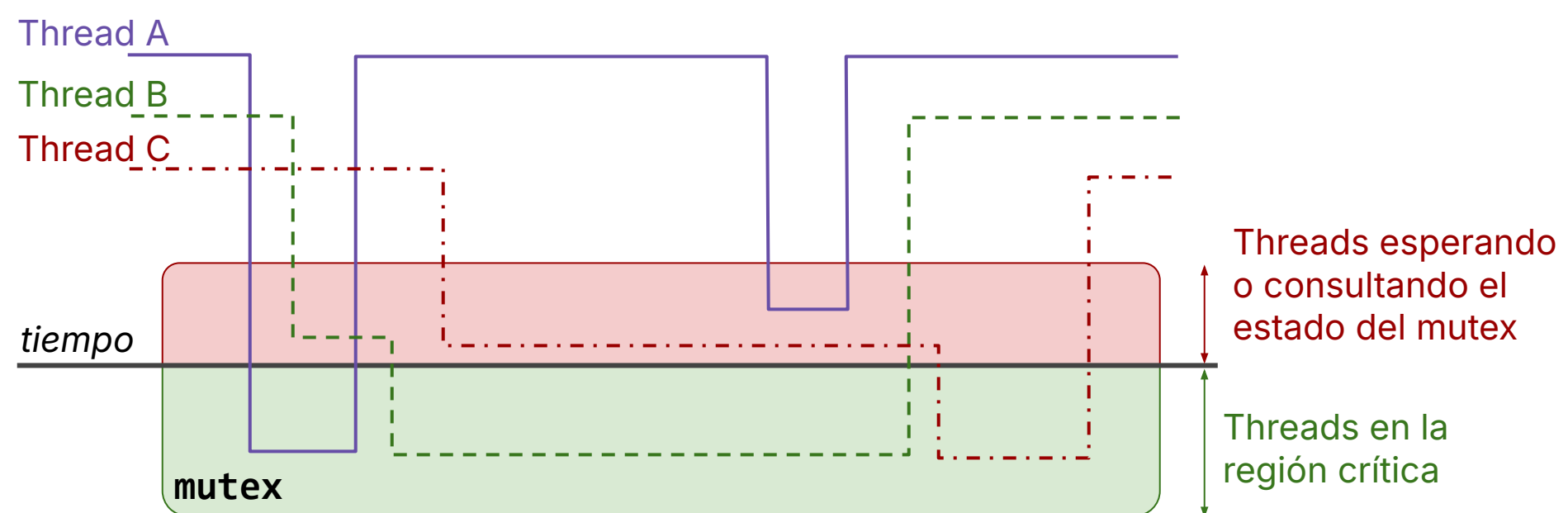
Región Crítica (I)

- **Invariante.** Condición sobre el estado **compartido** que debe satisfacerse cuando se observa fuera de la región crítica.
 - Ejemplo: *En una lista cada elemento debe contener un puntero válido al siguiente elemento menos el último que será NULL*
- **Región Crítica.**
 - Áreas de código donde se modifica o se consulta el estado compartido del programa.
 - Las regiones críticas preservan los invariantes del programa permitiendo la ejecución **exclusiva** de un thread.
 - Ejemplo: *Eliminar un elemento de la lista*
- **Predicados.** Expresión lógica que describen el estado de una invariante. El predicado indica que un thread puede ejecutarse.
 - Ejemplo: *La lista está vacía*

Región Crítica (II)

Mutex. Permiten implementar regiones críticas asegurando que un único thread ejecuta el código de la región a la vez (*mutual exclusion*)

- Un mutex tiene dos estados cerrado (*locked*) y abierto (*unlocked*)
- La región crítica es la sección de código comprendida entre las llamadas lock y unlock
- Caso especial de los semáforos [Dijkstra 1968] (más *difíciles* de usar)



Región Crítica (III)

```
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
```

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

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

- attr Define el comportamiento del mutex respecto a los errores:
 - *deadlock* Un thread no debe hacer lock() doble de un mutex
 - Un thread no debe hacer unlock() de mutex de otros threads
 - No hacer de unlock() the un mutex en estado unlock()
- La función destroy sólo se puede llamar cuando otros threads no estén esperando en el mutex y el mutex está unlock

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

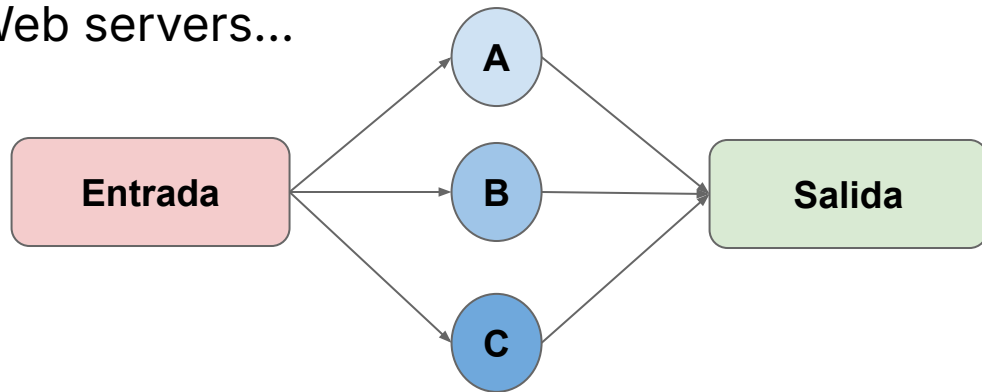
```
int pthread_mutex_trylock(pthread_mutex_t *mutex);
```

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

Patrón: *Work Crew / Thread Pool*

Work Crew

- Los threads cooperan mediante la ejecución independiente de tareas.
- La tarea ejecutada por cada thread puede ser diferente
- Requieren la sincronización en los datos de entrada y salida de tarea
- Ejemplos: Game engines, Web servers...



Ejemplo. Ejercicio 3

Escribir un programa cuyo primer argumento indicará el número de threads y el segundo el tamaño de bloque (T_b):

- Cada thread se le asignará un identificador ($i = 0, 1, 2, \dots$) y sumará T_b enteros en el rango $[i * T_b - (i+1) * T_b - 1]$
- El thread principal sincronizará todos los threads y mostrará la suma.

Patrón: *Lector - Escritor (I)*

Lector-Escritor

- En este patrón de sincronización se asume un recurso compartido:
 - Los accesos de lectura son mucho más frecuentes que los de escritura
 - Se permite el acceso concurrente al recurso por varios *lectores*
 - El acceso de los *escritores* debe ser exclusivo para mantener la consistencia de los datos.
 - La tarea ejecutada por cada thread puede ser diferente
- Ejemplos de aplicación:
 - Caché compartidas
 - Consulta del estado del global del juego
- **Problemas de Inanición** si se favorecen las lecturas o escrituras. En linux (pthread_rwlock_t) se favorece la productividad priorizando las lecturas.

Patrón: *Lector - Escritor (II)*

```
pthread_rwlock_t rwlock = PTHREAD_RWLOCK_INITIALIZER;  
  
int pthread_rwlock_init(pthread_rwlock_t *rwlock,  
                        const pthread_rwlockattr_t *attr);  
  
int pthread_rwlock_destroy(pthread_rwlock_t *rwlock);
```

- Mismas consideraciones en la creación y destrucción que un pthread_mutex_t

```
int pthread_rwlock_[rd|wr]lock(pthread_rwlock_t *lock);  
  
int pthread_rwlock_try[rd|wr]lock(pthread_rwlock_t *lock);  
  
int pthread_rwlock_unlock(pthread_rwlock_t *lock);
```

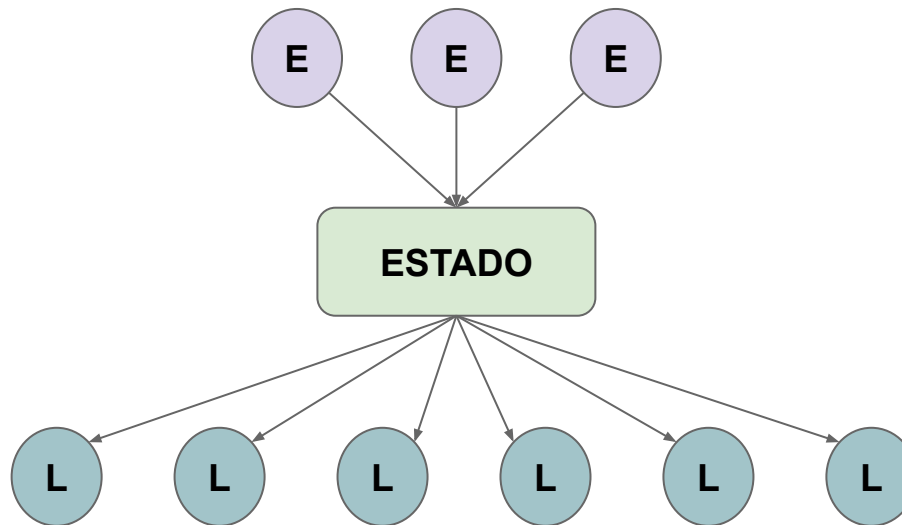
- Bloqueo de lectura (rdlock), el thread obtiene el mutex si no está bloqueado por un escritor ni hay escritores esperando (se puede bloquear varias veces)
- Bloqueo de escritura (wrlock), el thread obtiene el mutex si ningún otro thread tiene un bloqueo de cualquier tipo (rd o wr).

Patrón: *Lector - Escritor* (III)

Ejemplo. Ejercicio 6.

Escribir un programa que cree *L* lectores (primer argumento) y *E* escritores (segundo argumento), de forma que:

- Los threads *Lector* imprimirán por pantalla un entero compartido y esperarán 0.1s con la llamada `usleep(3)`. Este acceso lo repetirán 5 veces.
- Los threads *Escritor* incrementarán en 1 la variable y esperarán 0.25s. Este acceso lo repetirán 3 veces.
- El thread principal arrancará primero los escritores.





SISTEMAS OPERATIVOS

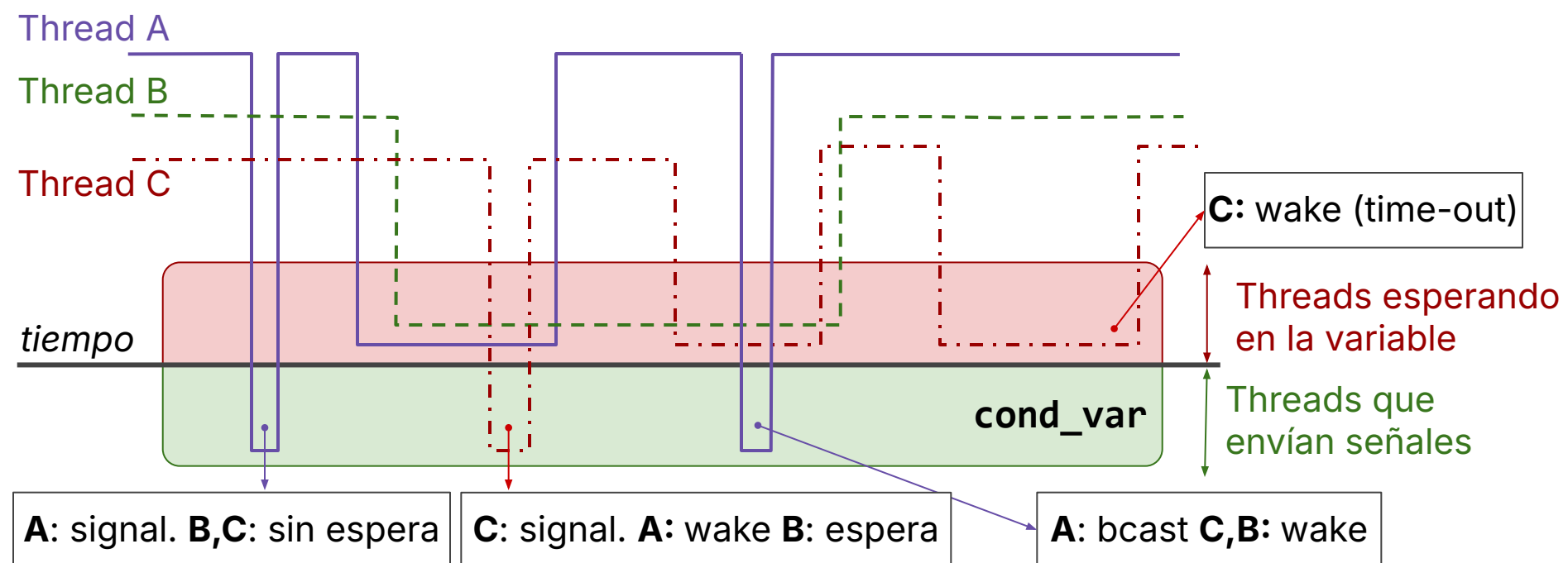
Grado en Desarrollo de Videojuegos
Universidad Complutense de Madrid

TEMA 5.3 Variables de Condición

Variables de Condición (I)

Variable de Condición. Abstracción que permite bloquear y despertar threads según la evaluación de un predicado sobre variables (*estado*) compartido:

- **Estado compartido**, representado por variables del programa. Ejemplo: *número de elementos en la lista*
- **Mutex**, para actualizar el estado en una región crítica. Un thread **despierta** de la espera siempre con el **mutex bloqueado** (*lock*)
- **Predicado**, condición sobre el estado que se espera/señaliza. Ejemplo: *elementos > 0*



Variables de Condición (II)

```
pthread_cond_t cond = PTHREAD_COND_INITIALIZER;
```

```
int pthread_cond_init(pthread_cond_t *cond,  
                      const pthread_condattr_t *attr);
```

```
int pthread_cond_destroy(pthread_cond_t *cond);
```

- La función destroy sólo se puede llamar cuando otros threads no estén esperando el la variable de condición

```
int pthread_cond_wait(pthread_cond_t *cond, pthread_mutex_t *mutex);
```

- Versión timedwait permite especificar un timeout (struct timespec).
- El thread vuelve de la llamada con mutex en estado lock
- El thread debe **comprobar el predicado** **antes** de entrar en la espera **y después** de salir de ella.

```
int pthread_cond_signal(pthread_cond_t *cond);
```

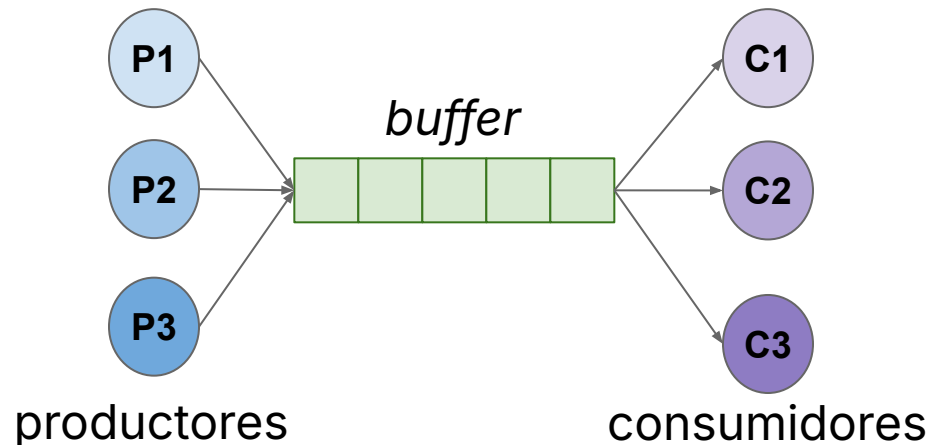
```
int pthread_cond_broadcast(pthread_cond_t *cond);
```

- Despierta a un thread (*cualquiera*) esperando en la variable de condición. La versión broadcast despierta todos.

Patrón de Diseño: Productor - Consumidor (I)

Productor - Consumidor

- Productores, generan datos que se colocan en un buffer compartido
- Consumidores, recogen los datos del buffer y los procesan.
- Variables de condición y mutex para controlar el buffer.
- Predicados: *Buffer no está vacío* (se puede consumir), *Buffer no está lleno* (se puede producir).



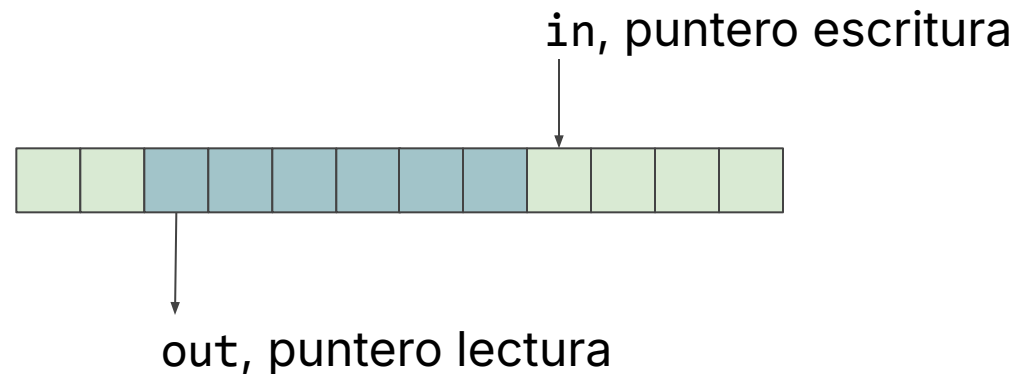
- Generalización en el patrón *pipeline*, con múltiples etapas de productor-consumidor.

Patrón de Diseño: Productor - Consumidor (II)

Ejemplo. Ejercicio

Escribir un programa con P threads productores (argumento 1) y C threads consumidores (argumento 2).

- Los tiempos de producción y consumición serán de 1 y 2 segundos respectivamente.
- El buffer compartido será un array de tamaño fijo con dos índices (*in* - producción, y *out* - consumición).



- Implementar el sistema con dos variables de condición y predicados:
 - consumidor: `elements > 0`
 - productor: `elements < buffer_size`
- Cada productor producirá un número fijo de elementos (`NUM_ELEMENTS`)

Patrón de Diseño: Productor - Consumidor (II)

Ejemplo

- Añadir una condición de finalización para los consumidores:
 - Opción 1. Contar el número de elementos totales consumidos, y modificar el predicado y los envíos de señales (broadcast) de los consumidores.
 - Opción 2 (patrón *píldora envenenada*). Escribir un elemento de finalización cuando se hayan producido todos los elementos, si un consumidor lee del buffer el elemento de finalización, terminará.



SISTEMAS OPERATIVOS

Grado en Desarrollo de Videojuegos
Universidad Complutense de Madrid

TEMA 5.4 Soporte del Sistema Operativo

Visibilidad de Memoria (I)

- El compilador y el hardware pueden **reordenar libremente las instrucciones** que se emiten si mantienen la **causalidad** aparente del programa.

```
x = 3;  
...  
c = 2 * x + 1  
...  
y = c;
```

relación de orden observable entre operaciones de memoria.

load, add, store
se pueden reordenar siempre que en este punto $y=7$

- En situaciones **asíncronas** (p.ej. manejadores de señales, multi-thread) no se puede asegurar esta **causalidad**:
 - Cache privadas por procesador (los protocolos de coherencia no aseguran valores consistentes hasta que se propagan las invalidaciones)
 - Ejecución *fuera de orden y especulativa* (que afecta temporización y cache)
 - Optimizaciones del compilador
 - Modelo de **consistencia** de memoria (reordenación de las operaciones store/load)

Visibilidad de Memoria (II)

Las barreras de memoria (*memory barriers*) imponen un orden parcial aparente de las operaciones en cada lado de la barrera.

Tipos de Barreras de Memoria

- **Lectura:** Las lecturas (load) antes de la barrera se completan antes de las operaciones de lectura después de la barrera.
- **Escritura:** Las escrituras (store) antes de la barrera se completan antes de las operaciones de escritura después de la barrera.
- **General/Completa:** afectan tanto a las operaciones load y store. Las llamadas `pthread_mutex_lock/unlock` actúan de este modo.

C++11. <atomic>

- Ofrece un interfaz para gestionar el acceso a datos compartidos en programas multi-thread.
- Permite definir el modelo de memoria de forma que el programador puede establecer un orden bien definido de accesos.

Visibilidad de Memoria (III)

pthread: Reglas de Visibilidad

- **Creación de threads (`pthread_create`)**
 - Un thread recién creado observa todos los efectos de memoria realizados por el thread creador antes de la llamada a `pthread_create()`.
- **Mutex (`pthread_mutex_lock` / `pthread_mutex_unlock`)**
 - Todas las operaciones de memoria realizadas antes de `pthread_mutex_unlock()` por un thread son visibles para cualquier thread que posteriormente adquiere el mismo mutex mediante `pthread_mutex_lock()`.
- **Finalización de threads (`pthread_join`)**
 - Todos los efectos de memoria de un thread finalizado son visibles para el thread que ejecuta `pthread_join()` sobre él.
- **Variables de condición (`pthread_cond_signal` / `pthread_cond_broadcast`)**
 - Todos los efectos de memoria previos a un `signal` o `broadcast` son visibles para los threads que se despiertan a consecuencia de la señalización.

Estos puntos requieren de una [barrera de memoria implícita](#) en las llamadas.

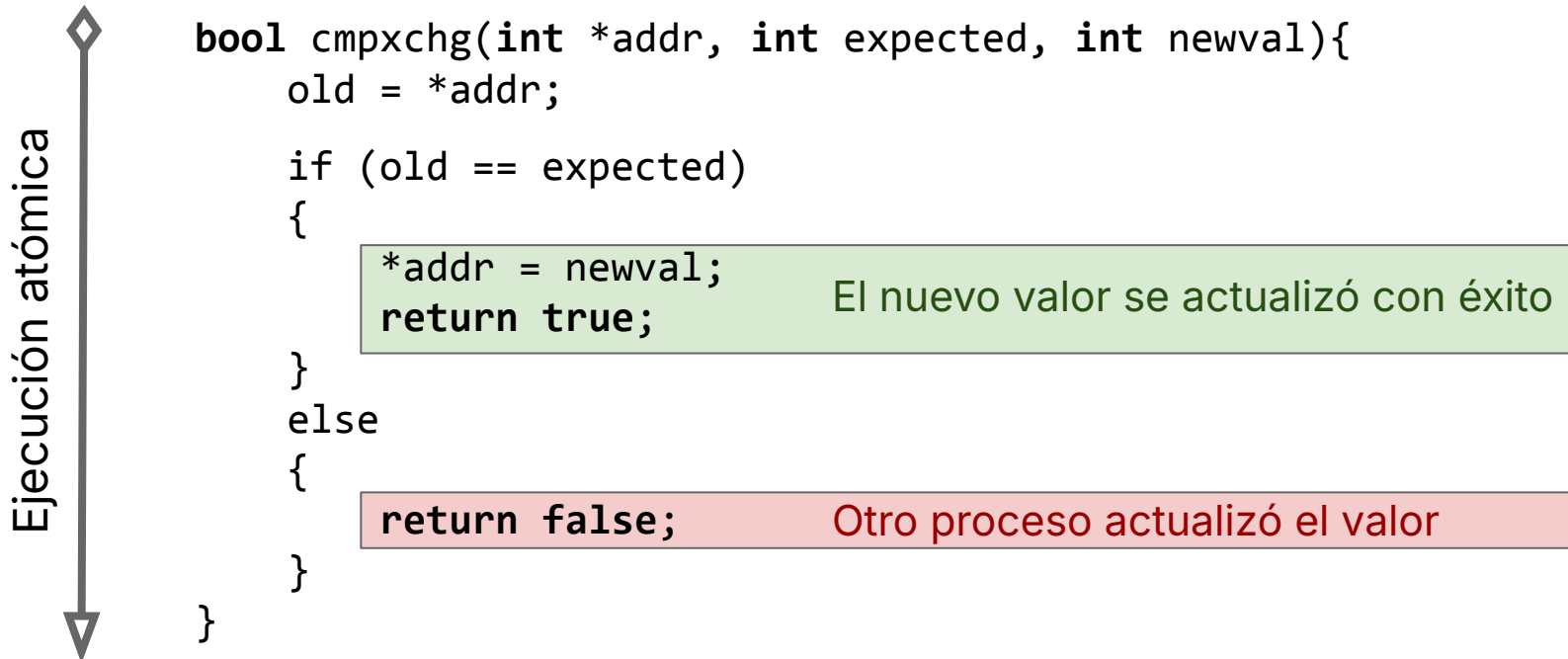
Soporte del Sistema: CPU ISA

Barreras de Memoria

- Soporte del procesador para implementar las barreras de memoria:
 - x86: `mfence`, `sfence`, `lfence`
 - ARM: `dmb`, `dsb`, `isb`

Instrucciones atómicas

- La arquitectura del repertorio de instrucciones ofrece operaciones de lectura-modificación-escritura atómicas (libres de condiciones de carrera)
- Ejemplo: *Compare-and-Swap*. Intel x86 instrucción `cmpxchg`



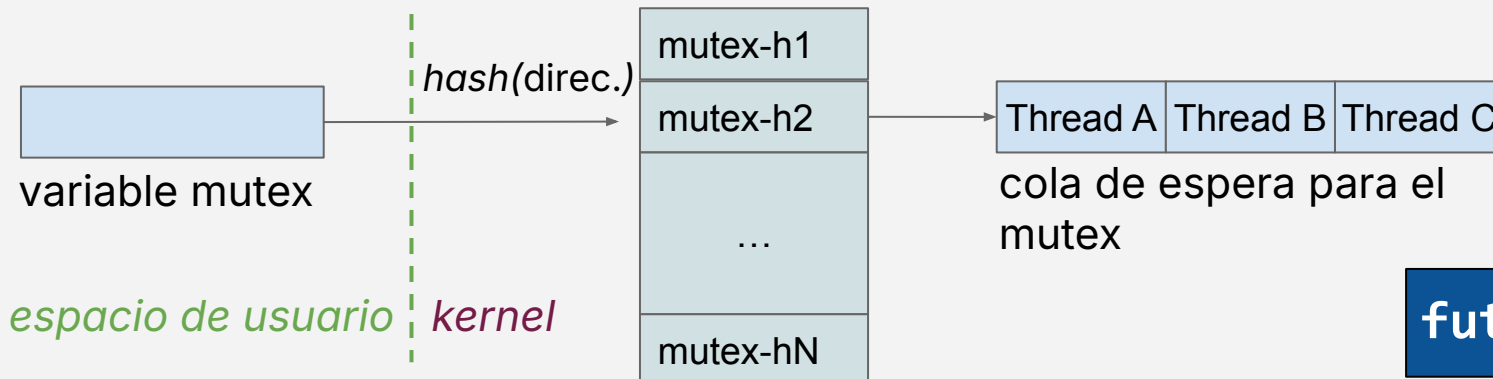
Soporte del Sistema: SO (I)

Subsistemas del SO

- Memoria virtual compartida entre threads (Tema 4)
- Suspensión y reanudación de la ejecución de threads (Tema 3)
- Implementación de colas de espera asociadas a mutex y variables de condición
- Linux y Windows implementan el soporte mediante la estructura futex (*fast user space mutex*)

futex

- Representados por una variable compartida por procesos (mmap - shared) o threads.
- La variable se modifica siempre mediante operaciones atómicas
- Las operaciones **sin contienda** se realizan en el espacio de usuario. Si varios threads entran en contienda por un mutex el kernel arbitra el acceso.



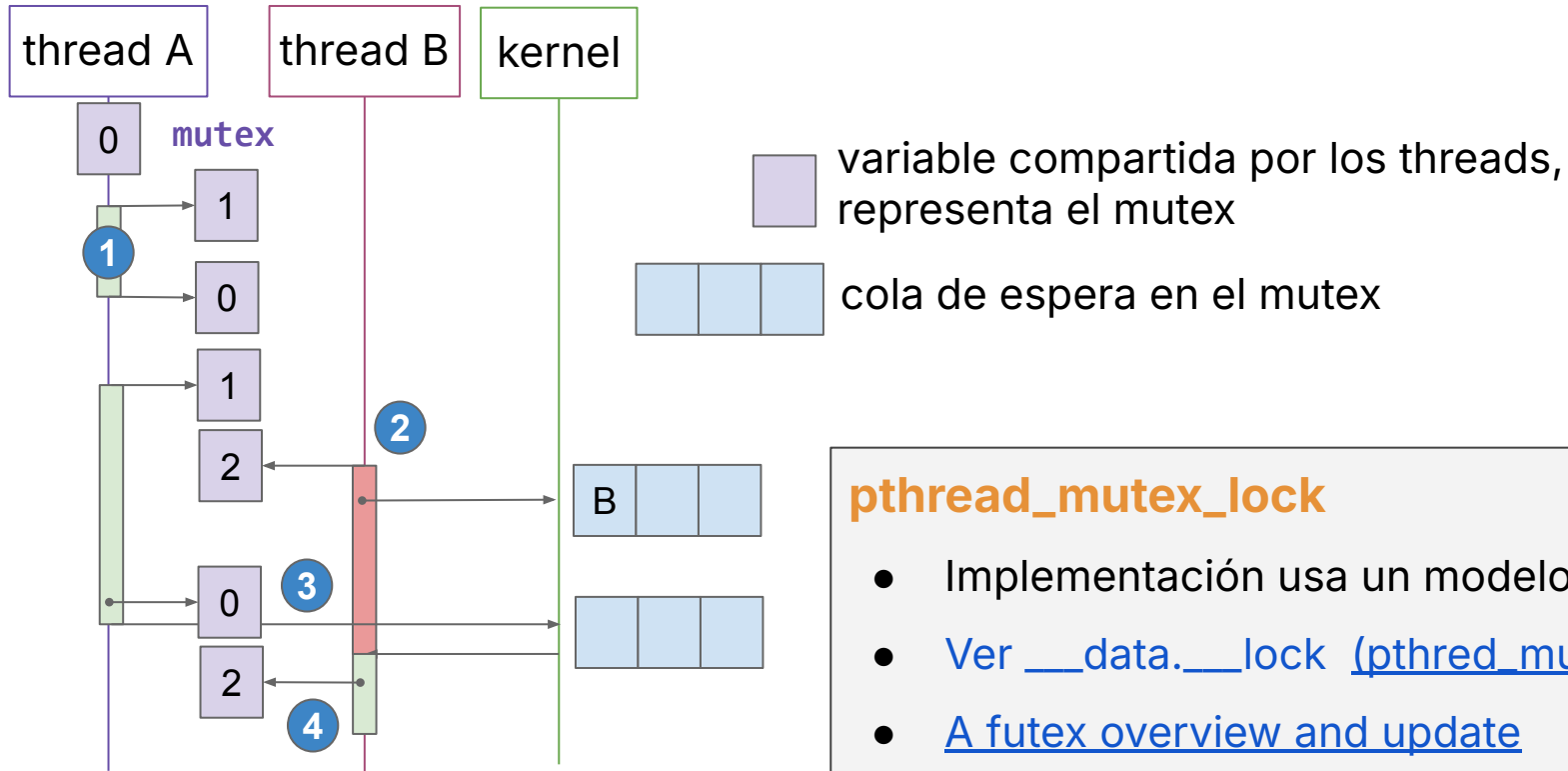
Soporte del Sistema: SO (II)

mutex lock

```
if ((c = cmpxchg(val, 0, 1)) != 0) ①
do
{
    if (c == 2 || cmpxchg(val, 1, 2) != 0)
        futex_wait(&val, 2); ②
} while ((c = cmpxchg(val, 0, 2)) != 0); ④
```

mutex unlock

```
if ((atomic_dec(val, 0, 1)) != 0) ③
{
    val = 0;
    futex_wake (&val, 1);
}
```



pthread_mutex_lock

- Implementación usa un modelo similar
- Ver `__data.__lock` ([pthread_mutex_lock.c](#))
- [A futex overview and update](#)