# Informática I
# Threads

Alejandro Furfaro

28 de Setiembre 2010

# Agenda

1. Concepto de programación paralela.
2. Thread ¿Que es?.
3. Concurrencia => Race Conditions.
4. Sincronización.
5. Funciones

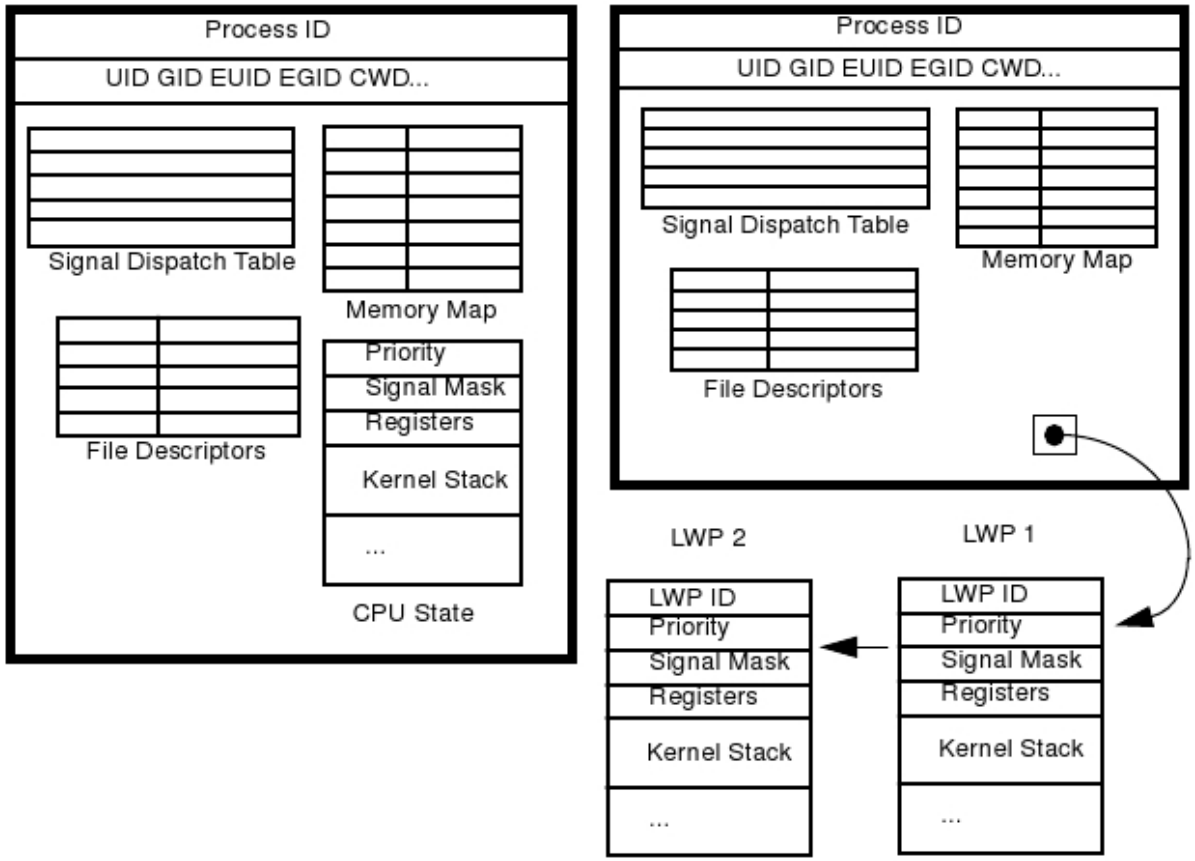
# Concepto de Programación Paralela

- **Computación paralela**
  Técnica de programación en la que muchas instrucciones se ejecutan simultáneamente.
  Se basa en el principio de que los problemas grandes se pueden dividir en partes más pequeñas que pueden resolverse de forma concurrente, es decir, “en paralelo”.
  Al principio para aprovechar esta posibilidad los motherboards deb’ian incluir mas de un procesador.
  Actualmente un procesador puede tener hasta 12 CPUs (Core i7 Extreme Edition).
- **Multithreading**
  Técnica que permite a un programa la posibilidad de hacer varias operaciones en forma concurrente.

# Threads => Flujos de instrucciones

- As'i como un sistema operativo puede manejar múltiples proceso en forma concurrente, un proceso puede manejar múltiples flujos de instrucciones en forma concurrente.
- Un **Thread** es justamente eso: Un flujo de instrucciones que se ejecuta en paralelo con el resto del proceso.
- Un **Thread** es parte de un proceso, con menos estructura de control.
  - Si un proceso contiene Datos, Código, Estado del Kernel, Contexto de la CPU, un **Thread** está representado por el contexto de la CPU.
  - Pero un **Thread** es solo esa parte. Y es la que lo hace diferente de los demás **Threads** que pueda crear el proceso.
  - El resto de la estructura de control del proceso es común a todos los **Threads** (Tablas de descriptores de archivos,

# Estruturas


Process ID
UID GID EUID EGID CWD...
Signal Dispatch Table
Memory Map
File Descriptors
Priority
Signal Mask
Registers
Kernel Stack
...
CPU State
Process ID
UID GID EUID EGID CWD...
Signal Dispatch Table
Memory Map
File Descriptors
LWP 2
LWP 1
LWP ID
Priority
Signal Mask
Registers
Kernel Stack
...
LWP ID
Priority
Signal Mask
Registers
Kernel Stack
...

# Procesos Livianos

- Se pueden tener varios en una misma área de memoria.
- El tiempo de creación es 10 a 100 veces menor comparado con la creación de procesos pesados.
- Se destruyen con mayor rapidez.
- El context switch es más rápido.
- Se comunican entre si con mayor facilidad.
- **Reducen la seguridad.**

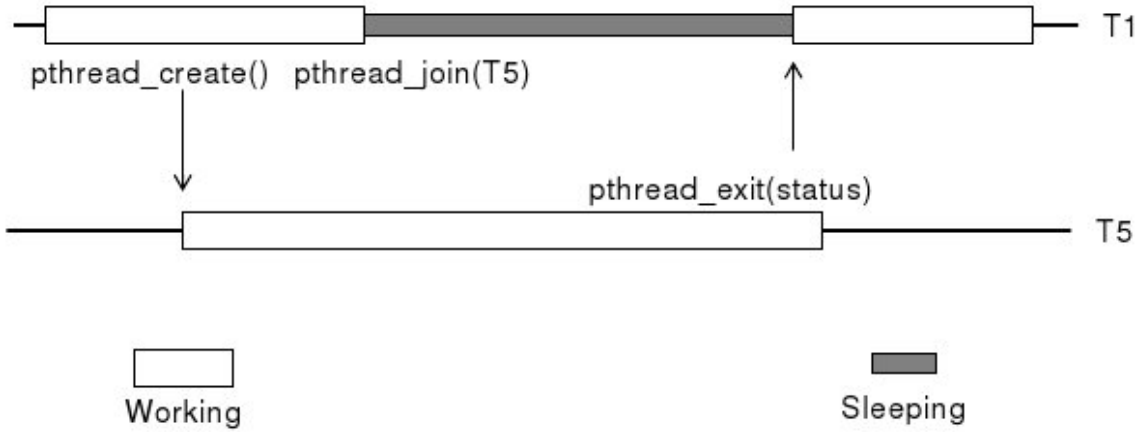
# Para que sincronizar?

- Los **Threads** comparten todo su mapa de memoria, variables, y recursos, por lo tanto se crean race condition constantemente.
- Al igual que los semáforos en los procesos se debe prevenir la escritura y lectura concurrente en los **Threads**.
- Podemos (y en ocasiones debemos!) bloquear un **Thread** hasta que se de una condición espec'ifica.

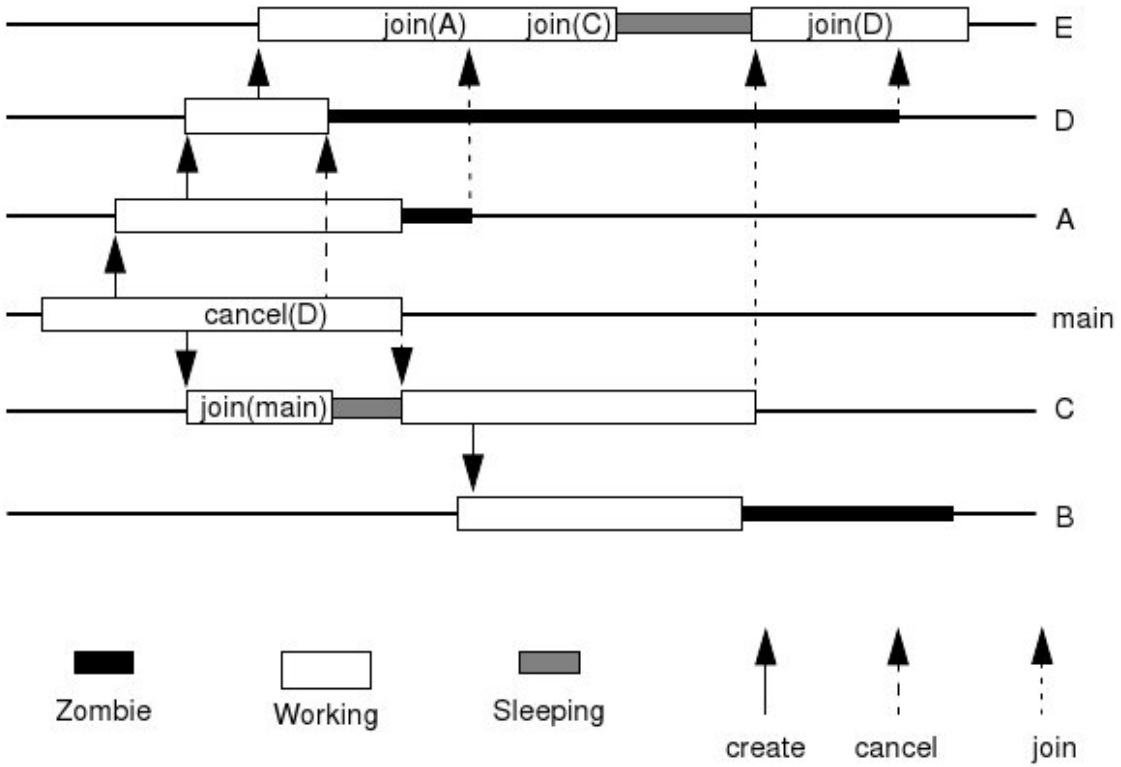
# Mecanismos de Sincronización

- **Mutex** (**Mut**ual **ex**clusion lock)
  Similar al semáforo para procesos (ipc). Bloquea el flujo hasta que es liberado.
  `pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER`
- **Join**. Espera a que termine un **Thread** espec'ifico.
- **Condition Variables** Bloquean un **Thread** hasta que otro cumple la condición. Puede despertar a un **Thread** que esté esperando, o a todos aquellos que estén bloqueados por la condición.
  `pthread_cond_t condition_cond = PTHREAD_COND_INITIALIZER`

# Join


T1
pthread_create()
pthread_join(T5)
pthread_exit(status)
T5
Working
Sleeping

# Join


join(A)
join(C)
join(D)
E
D
A
cancel(D)
main
join(main)
C
B
Zombie
Working
Sleeping
create
cancel
join

# Funciones 1/4

- **Crear un thread**
  ```
  int pthread_create(pthread_t *tid ,pthread_attr_t
  *attr,void *(*func)(void*), void *arg);
  ```
- **join**. Espera por la finalización de un **Thread** que no sea detach.
  ```
  int pthread_join(pthread_t tid, void **status);
  ```
- **detach**. Configura al thread para liberar los recursos usados al terminar.
  ```
  int pthread_detach(pthread_tic);
  ```
- **Terminar** la ejecución.
  ```
  void pthread_exit(void *status);
  ```
- Devuelve el thread id.
  ```
  pthread_t pthread_self(void);
  ```

# Funciones 2/4

- **Copia** los argumentos por defecto a atributo.
  `int pthread_attr_init(pthread_attr_t* atributo);`
- Destruye la varible atributo
  `int pthread_attr_destroy(pthread_attr_t * atributo);`
- Compara dos **Threads** id (devuelve cero si son distintos).
  `int pthread_equal(pthread_t tid1, pthread_t tid2);`
- **Cancela** la ejecución del **Thread** indicado.
  `int pthread_cancel(pthread_t tid);`

# Funciones 3/4

- Si el **mutex** está libre lo bloquea, sino detiene su ejecución hasta que esté libre.

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

- Trata de bloquar el mutex, si esta ya esta tomado devuelve un EBUSY.

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

- Si el **Thread** está bloqueado lo libera.

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

- **Destruye** el mutex..

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

# Funciones 4/4

- **Bloquea** hasta que se cumple la condición.
```
int pthread_cond_wait (pthread_cond_t *cond,
pthread_mutex_t *mutex);
```
- Despierta a un **Thread** que este dormido por cond_wait.
```
int pthread_cond_signal (pthread_cond_t *cond);
```
- Despierta a todos los **Threads** dormidos.
```
int pthread_cond_broadcast (pthread_cond_t
*cond);
```
- **Destruye** a la variable cond.
```
int pthread_cond_destroy (pthread_cond_t *cond);
```