Taller de pthreads

Javier Pimás

DC - FCEyN - UBA

Sistemas Operativos (taller), 2do cuatrimestre de 2010.

Repaso: ¿Qué son los *threads*?

¿V/F? "Son como procesos *light*."

V → En general un proceso involucra "más" estado que un thread.

Repaso: ¿Qué son los *threads*?

¿V/F? "Son como procesos *light*."

V → En general un proceso involucra "más" estado que un thread.

¿V/F? "Son como *mini-procesos* dentro de un proceso."

Repaso: ¿Qué son los *threads*?

¿V/F? "Son como procesos *light*."

V → En general un proceso involucra "más" estado que un thread.

¿V/F? "Son como *mini-procesos* dentro de un proceso."

V → Todo thread "vive en" (léase: es parte de) un proceso particular.

Breve paneo por la realidad

Rapidito y sin dolor (ni mucho detalle): qué tipos de threads hay actualmente "ahí afuera", de dónde vienen y para dónde parecen estar rumbeando.

- Threads Java.
- Threads Windows.
- Threads del siglo XX.
- Threads del siglo XXI $\rightarrow$ `pthreads`.
- Tendencias, industria, futuro.

Siglo XXI: "Portability" con P de POSIX.

- En 1995, la IE^3 logró incorporar los threads al standard.

  Versión vigente: IEEE POSIX 1003.1c (2004) + algo de hermenéutica (era inevitable).

Siglo XXI: "Portability" con P de POSIX.

- En 1995, la IE^3 logró incorporar los threads al standard.

  Versión vigente: IEEE POSIX 1003.1c (2004) + algo de hermenéutica (era inevitable).

- `pthreads` fue un paso crucial hacia la inter-compatibilidad.

  GNU/Linux, Free/Open/NetBSD, Mac OS X, AIX, HP-UX, Solaris, IRIX, Cygwin, Symbian OS ...

Siglo XXI: "Portability" con P de POSIX.

- En 1995, la IE^3 logró incorporar los threads al standard.

  Versión vigente: IEEE POSIX 1003.1c (2004) + algo de hermenéutica (era inevitable).

- `pthreads` fue un paso crucial hacia la inter-compatibilidad.

  GNU/Linux, Free/Open/NetBSD, Mac OS X, AIX, HP-UX, Solaris, IRIX, Cygwin, Symbian OS ...

- `pthreads` no es una implementación sino una especificación.

  Impone API común y semántica [casi] clara. Implementaciones varias, pero [casi] intercambiables.

Siglo XXI: "Portability" con P de POSIX.

- En 1995, la IE^3 logró incorporar los threads al standard.

  Versión vigente: IEEE POSIX 1003.1c (2004) + algo de hermenéutica (era inevitable).

- `pthreads` fue un paso crucial hacia la inter-compatibilidad.

  GNU/Linux, Free/Open/NetBSD, Mac OS X, AIX, HP-UX, Solaris, IRIX, Cygwin, Symbian OS ...

- `pthreads` no es una implementación sino una especificación.

  Impone API común y semántica [casi] clara. Implementaciones varias, pero [casi] intercambiables.

- En 2003, NPTL se afianzó como "la" implementación para Linux.

  Native POSIX Threads Library (donde "native" implica "con soporte a nivel del kernel").

Siglo XXI: "Portability" con P de POSIX.

- En 1995, la IE^3 logró incorporar los threads al standard.

  Versión vigente: IEEE POSIX 1003.1c (2004) + algo de hermenéutica (era inevitable).

- `pthreads` fue un paso crucial hacia la inter-compatibilidad.

  GNU/Linux, Free/Open/NetBSD, Mac OS X, AIX, HP-UX, Solaris, IRIX, Cygwin, Symbian OS ...

- `pthreads` no es una implementación sino una especificación.

  Impone API común y semántica [casi] clara. Implementaciones varias, pero [casi] intercambiables.

- En 2003, NPTL se afianzó como "la" implementación para Linux.

  Native POSIX Threads Library (donde "native" implica "con soporte a nivel del kernel").

$\Rightarrow$ El uso de threads se volvió aceptable para muchos más proyectos.

Referencias

- IEEE Online Standards: POSIX

  `http://standards.ieee.org/catalog/olis/arch_posix.html`

  `http://www.unix.org/version3/ieee_std.html`

- Tutorial del LLNL sobre pthreads

  `https://computing.llnl.gov/tutorials/pthreads/`

¿Cómo se define *thread* en el standard?

En la sección *Base Definitions* del IEEE 1003.1c leemos:

Thread (3.393) A single flow of control within a process.

Each thread has its own thread ID, scheduling priority and policy, errno value, thread-specific key/value bindings, and the required system resources to support a flow of control.

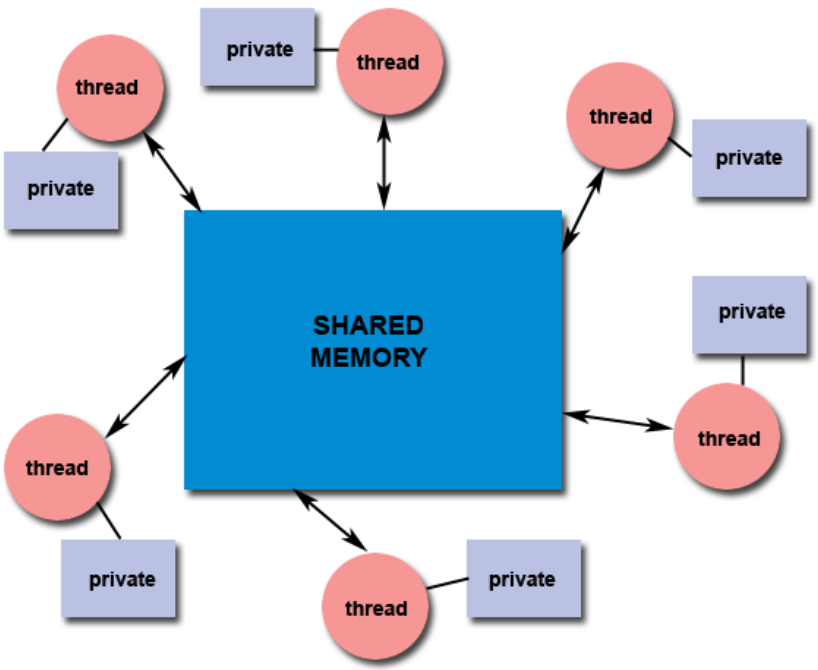
Anything whose address may be determined by a thread, including but not limited to static variables, storage obtained via `malloc()`, directly addressable storage obtained through implementation-defined functions, and automatic variables, are accessible to all threads in the same process.

Thread ID (3.394) Each thread in a process is uniquely identified during its lifetime by a value of type `pthread_t` called a thread ID.

Lo mío, lo tuyo, lo propio, lo ajeno.

¿Cómo describir el modelo de acceso a memoria que muestra el diagrama?

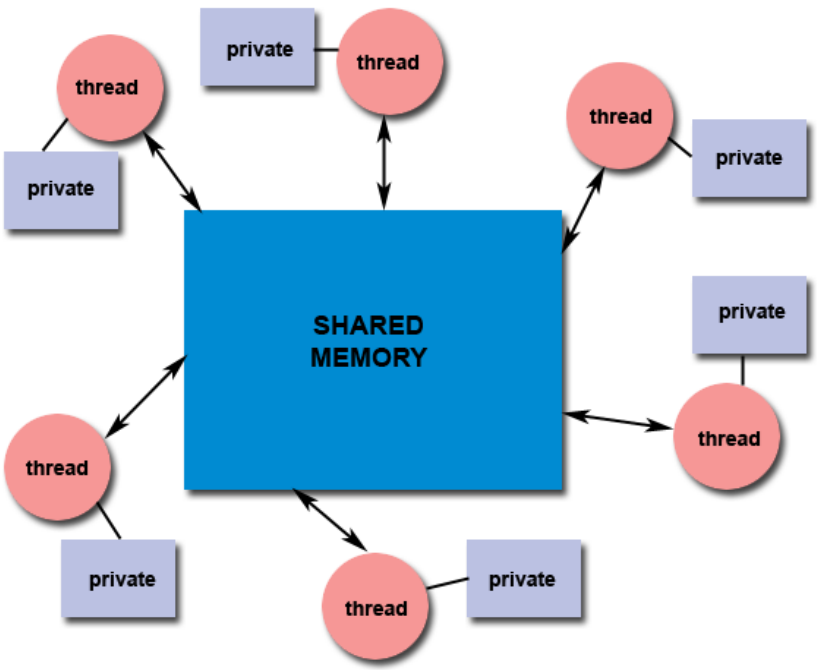
¿Compartida? ¿Privada? ¿Distribuida? ¿Un híbrido . . . ?



Lo mío, lo tuyo, lo propio, lo ajeno.

¿Cómo describir el modelo de acceso a memoria que muestra el diagrama?

¿Compartida? ¿Privada? ¿Distribuida? ¿Un híbrido . . . ?

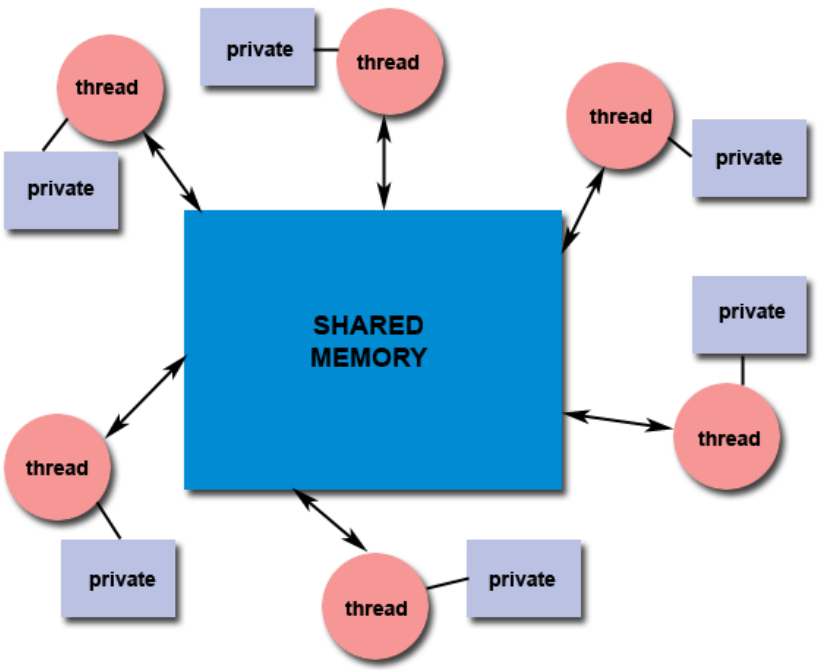


- ¿Cuántos threads hay ahí?
- ¿Procesos? ¿Máquinas?

Lo mío, lo tuyo, lo propio, lo ajeno.

¿Cómo describir el modelo de acceso a memoria que muestra el diagrama?

¿Compartida? ¿Privada? ¿Distribuida? ¿Un híbrido . . . ?



- ¿Cuántos threads hay ahí?
- ¿Procesos? ¿Máquinas?

⇒ No puede ser distribuida.

- Algo de compartida hay.
- Algo de privada también.
- ¿Qué predomina?

¿Por qué threads y no procesos?

Por razones de eficiencia/performance

- `fork()` es considerablemente más caro que `pthread_create()`.

¿Por qué threads y no procesos?

Por razones de eficiencia/performance

- `fork()` es considerablemente más caro que `pthread_create()`.
- El context-switching también es más pesado y costoso entre procesos.

¿Por qué threads y no procesos?

Por razones de eficiencia/performance

- `fork()` es considerablemente más caro que `pthread_create()`.
- El context-switching también es más pesado y costoso entre procesos.
- El overhead total usualmente varía en 1 orden de magnitud.

¿Por qué threads y no procesos?

Por razones de eficiencia/performance

- `fork()` es considerablemente más caro que `pthread_create()`.
- El context-switching también es más pesado y costoso entre procesos.
- El overhead total usualmente varía en 1 orden de magnitud.

¿Por qué threads y no procesos?

Por razones de eficiencia/performance

- `fork()` es considerablemente más caro que `pthread_create()`.
- El context-switching también es más pesado y costoso entre procesos.
- El overhead total usualmente varía en 1 orden de magnitud.

Demo: `time forks` vs. `time pthreads`

- ...

¿Por qué podría querer procesos y no threads?

Un ejemplo: cuando la promiscuidad amenaza la seguridad

- Ejemplo: algo pequeño pero muy sensible, tipo `ssh-agent`.

Otro similar: cuando va en detrimento de la confiabilidad

¿Por qué podría querer procesos y no threads?

Un ejemplo: cuando la promiscuidad amenaza la seguridad

- Ejemplo: algo pequeño pero muy sensible, tipo `ssh-agent`.
- Idea: usando threads, compartir 1 tal proceso entre N usuarios.

Otro similar: cuando va en detrimento de la confiabilidad

¿Por qué podría querer procesos y no threads?

Un ejemplo: cuando la promiscuidad amenaza la seguridad

- Ejemplo: algo pequeño pero muy sensible, tipo `ssh-agent`.
- Idea: usando threads, compartir 1 tal proceso entre N usuarios.
- ¡Ni a palos! ¡Esa memoria es mía, mía, mía!

Otro similar: cuando va en detrimento de la confiabilidad

¿Por qué podría querer procesos y no threads?

Un ejemplo: cuando la promiscuidad amenaza la seguridad

- Ejemplo: algo pequeño pero muy sensible, tipo `ssh-agent`.
- Idea: usando threads, compartir 1 tal proceso entre N usuarios.
- ¡Ni a palos! ¡Esa memoria es mía, mía, mía!

Otro similar: cuando va en detrimento de la confiabilidad

- Ejemplo: tengo una 2da instancia de cierto *daemon* crítico.

¿Por qué podría querer procesos y no threads?

Un ejemplo: cuando la promiscuidad amenaza la seguridad

- Ejemplo: algo pequeño pero muy sensible, tipo `ssh-agent`.
- Idea: usando threads, compartir 1 tal proceso entre N usuarios.
- ¡Ni a palos! ¡Esa memoria es mía, mía, mía!

Otro similar: cuando va en detrimento de la confiabilidad

- Ejemplo: tengo una 2da instancia de cierto *daemon* crítico.
- Idea: configurarlo como un segundo thread del mismo proceso.

¿Por qué podría querer procesos y no threads?

Un ejemplo: cuando la promiscuidad amenaza la seguridad

- Ejemplo: algo pequeño pero muy sensible, tipo `ssh-agent`.
- Idea: usando threads, compartir 1 tal proceso entre N usuarios.
- ¡Ni a palos! ¡Esa memoria es mía, mía, mía!

Otro similar: cuando va en detrimento de la confiabilidad

- Ejemplo: tengo una 2da instancia de cierto *daemon* crítico.
- Idea: configurarlo como un segundo thread del mismo proceso.
- ¡Ni loco! Quiero un proceso aparte. Idealmente, en otro equipo.

¿Por qué podría querer procesos y no threads?

Un ejemplo: cuando la promiscuidad amenaza la seguridad

- Ejemplo: algo pequeño pero muy sensible, tipo `ssh-agent`.
- Idea: usando threads, compartir 1 tal proceso entre N usuarios.
- ¡Ni a palos! ¡Esa memoria es mía, mía, mía!

Otro similar: cuando va en detrimento de la confiabilidad

- Ejemplo: tengo una 2da instancia de cierto *daemon* crítico.
- Idea: configurarlo como un segundo thread del mismo proceso.
- ¡Ni loco! Quiero un proceso aparte. Idealmente, en otro equipo.

¿Por qué podría querer procesos y no threads?

Un ejemplo: cuando la promiscuidad amenaza la seguridad

- Ejemplo: algo pequeño pero muy sensible, tipo `ssh-agent`.
- Idea: usando threads, compartir 1 tal proceso entre N usuarios.
- ¡Ni a palos! ¡Esa memoria es mía, mía, mía!

Otro similar: cuando va en detrimento de la confiabilidad

- Ejemplo: tengo una 2da instancia de cierto *daemon* crítico.
- Idea: configurarlo como un segundo thread del mismo proceso.
- ¡Ni loco! Quiero un proceso aparte. Idealmente, en otro equipo.

Hay muchos más ejemplos (ejercicio: pensar algunos más), pero sigamos adelante.

La API es grande pero el "core" azón es chico.

tipo de datos (tid)	`pthread_t`
crear nuevo thread	`pthread_create(thread, attr, startfn, arg)`
terminar un thread	`pthread_exit(status)`
crear atributos	`pthread_attr_init(attr)`
destruir atributos	`pthread_attr_destroy(attr)`

Por concisión hemos omitido aquí las demás primitivas (unas 90) y abstraido bastante los tipos de los parámetros (casi todos son punteros-a-eso, etc). Para los detalles escabrosos de cada tipo y función, véase `man 3 pthread`.

Pasando parámetros y usando atributos.

crear atributos	`pthread_attr_init(attr)`
crear nuevo thread	`pthread_create(thread, attrs, startfun, arg)`

Pasando parámetros y usando atributos.

crear atributos `pthread_attr_init(attr)`

crear nuevo thread `pthread_create(thread, attrs, startfun, arg)`

attr Atributos. NULL $\Rightarrow$ todos los attrs en valores por defecto.

Pasando parámetros y usando atributos.

crear atributos	`pthread_attr_init(attr)`
crear nuevo thread	`pthread_create(thread, attrs, startfun, arg)`

attr	Atributos. NULL ⇒ todos los attrs en valores por defecto.
startfun	Puntero a función que recibe 1 puntero a `void`. No puede ser NULL. (¡el thread necesita un punto de entrada!)

Pasando parámetros y usando atributos.

crear atributos	`pthread_attr_init(attr)`
crear nuevo thread	`pthread_create(thread, attrs, startfun, arg)`

attr	Atributos. NULL ⇒ todos los attrs en valores por defecto.
startfun	Puntero a función que recibe 1 puntero a `void`. No puede ser NULL. (¡el thread necesita un punto de entrada!)
arg(s)	Instancia de `void*` que recibirá `startfun(void* arg)`. NULL ⇒ OK, si `startfun()` no lo necesita . . . .

Pasando parámetros y usando atributos.

crear atributos	`pthread_attr_init(attr)`
crear nuevo thread	`pthread_create(thread, attrs, startfun, arg)`

attr	Atributos. NULL ⇒ todos los attrs en valores por defecto.
startfun	Puntero a función que recibe 1 puntero a `void`. No puede ser NULL. (¡el thread necesita un punto de entrada!)
arg(s)	Instancia de `void*` que recibirá `startfun(void* arg)`. NULL ⇒ OK, si `startfun()` no lo necesita . . . .

Para pasar estructuras más complejas

1. definimos una `struct` con campos a gusto
2. al crear un thread, le pasamos un puntero-a-eso
3. el nuevo thread recibe ese puntero y . . . lo *castea* a lo macho.

Compilando código que usa pthreads

Fácil; basta con . . .

```
CFLAGS=-pthread
```

(agregar al Makefile)

```
gcc -pthread -o test test.c
```

```
g++ -pthread -o test test.cpp
```

Eso agrega los `-I` necesarios para compilar, los `-L` para linkear la biblioteca, etc.

API básica para exclusión mutua.

tipo de datos	`pthread_mutex_t`
crear mutex	`pthread_mutex_init(mutex, attr)`
destruir mutex	`pthread_mutex_destroy(mutex)`
espera bloqueante	`pthread_mutex_lock(mutex)`
intento no bloqueante	`pthread_mutex_trylock(mutex)`
liberación (signal)	`pthread_mutex_unlock(mutex)`

API básica para variables de condición.

tipo de datos	`pthread_cond_t`
crear VC	`pthread_cond_init(cond, attr)`
destruir VC	`pthread_cond_destroy(mutex)`
crear atributos	`pthread_condattr_init(attr)`
destruir atributos	`pthread_condattr_destroy(mutex)`
wait	`pthread_cond_wait(cond, mutex)`
signal	`pthread_cond_signal(cond)`
broadcast	`pthread_cond_broadcast(cond)`

Modo de uso de VCs y mutexes.

- Notar que una VC siempre se usa con un mutex asociado.
  Hay reglas de juego ahí (usar o no el mismo mutex acá o allá, etc).
  Es responsabilidad del programador conocer y respetar estos contratos.

Modo de uso de VCs y mutexes.

- Notar que una VC siempre se usa con un mutex asociado.
  Hay reglas de juego ahí (usar o no el mismo mutex acá o allá, etc).
  Es responsabilidad del programador conocer y respetar estos contratos.

- Cuando un thread **–que debe tener el mutex ya tomado–** llama a `wait()`, suelta el mutex y entra en espera bloqueante.

Modo de uso de VCs y mutexes.

- Notar que una VC siempre se usa con un mutex asociado.
  Hay reglas de juego ahí (usar o no el mismo mutex acá o allá, etc).
  Es responsabilidad del programador conocer y respetar estos contratos.

- Cuando un thread –**que debe tener el mutex ya tomado**– llama a `wait()`, suelta el mutex y entra en espera bloqueante.

- Cuando un thread llama a `signal()`, *otro* thread en espera, de haberlo, se despierta de *su* `wait()` **con el mutex ya adquirido**.

Modo de uso de VCs y mutexes.

- Notar que una VC siempre se usa con un mutex asociado.
  Hay reglas de juego ahí (usar o no el mismo mutex acá o allá, etc).
  Es responsabilidad del programador conocer y respetar estos contratos.

- Cuando un thread –**que debe tener el mutex ya tomado**– llama a `wait()`, suelta el mutex y entra en espera bloqueante.

- Cuando un thread llama a `signal()`, *otro* thread en espera, de haberlo, se despierta de *su* `wait()` **con el mutex ya adquirido**.

- Si no hay ningún thread esperando a esa VC, tanto los `signal()` como los `broadcast()` se ignoran: no tienen efecto ni se acumulan.

Modo de uso de VCs y mutexes (cont.)

Sea `count` una variable global que lleva la cuenta de algo. Interesa que sea incrementada ante cierto evento, detectable por alguno de N threads activos; léase, uno cualquiera de ellos en cada ocurrencia del evento en cuestión.

Modo de uso de VCs y mutexes (cont.)

Sea count una variable global que lleva la cuenta de algo. Interesa que sea incrementada ante cierto evento, detectable por alguno de N threads activos; léase, uno cualquiera de ellos en cada ocurrencia del evento en cuestión.

varcond.c

```
pthread_mutex_lock(&count_mutex);

if(count < COUNT_LIMIT) {
    pthread_cond_wait(&count_threshold_cv, &count_mutex);
    printf("count(): thread %ld: cond. signal received.\n", my_id);
    count += 125;
    printf("count(): thread %ld: count is now %d.\n", my_id, count);
}

pthread_mutex_unlock(&count_mutex);
pthread_exit(NULL);
```