Threads

IPC

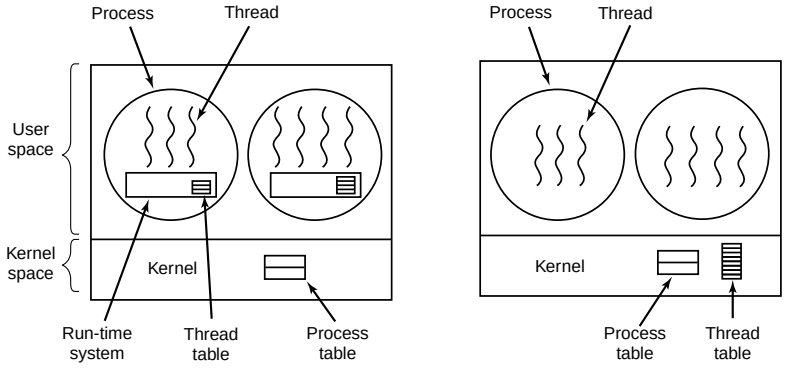
Due maniere di implementare i thread

supportati dal kernel (Windows NT/95/98, Mach, Linux)

implementati in spazio utente (Java Virtual Machine, Ada, Posix Threads,...)

ibrido: thread sia nel kernel che in spazio utente (Solaris)

Implementare i thread in spazio utente vs. nel kernel



Thread in spazio utente

Realizzati in una libreria
thread table

Il context switch non richiede di saltare in modo kernel
⇒ molto veloce!

Posso avere i thread su SO che non li supportano

Problema: system call bloccanti
⇒ se eseguo una chiamata bloccante si bloccano *tutti* i thread

Problena: non posso sfruttare più CPU

Problema: context switch sempre volontario

Chiamate di sistema bloccanti e user-level threads

Soluzione: wrapper code around system calls

Occorre modificare la system call library

Esempio: il programma contiene
`read(fd, buf, len)`

Il wrapper esegue select(2) per sapere se read(2) bloccherebbe o no

read(2) viene eseguita solo quando non blocca

Kernel-level threads

Il SO gestisce i thread

La thread table si affianca alla process table

(In Linux esiste una tabella sola)

Le chiamate bloccanti sono gestite dal kernel

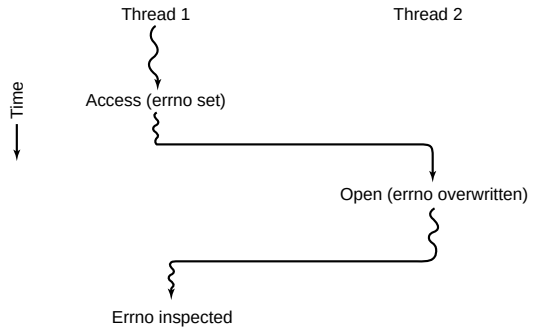
I thread e le variabili globali

La globale `errno(3)` indica il motivo del fallimento di una syscall

```
if (-1 == open("foo.txt", O_RDONLY)) {
  fprintf(stderr, "error in open: %d\n", errno);
  exit(EXIT_FAILURE);
}
```

Nota: errno(3) è parte della libreria standard C
⇒ per usare i thread *occorre modificare la lib std C*

Conflitto sull'uso della globale errno



Thread-local storage

Una variabile TLS ha un valore diverso per ciascun thread

Ogni thread ha la sua copia

Un solo nome!

TLS usato nelle implementazioni thread-safe della lib std C

(non solo errno; anche malloc(3) ...)

Creare thread in Linux

```
int clone(int (*fn)(void *), void *child_stack, int flags, void *arg);
```

creates a new process, just like fork(2). [...] Unlike fork(2), these calls allow the child process to share parts of its execution context with the calling process, such as the memory space, the table of file descriptors, [...]

non è standard!

Esempio:

```
#define STK_SIZE (512*1024)
int f(void * pv) {  printf("hello, threaded world!\n"); }
...
void * stack = malloc(STK_SIZE);
clone(f, stack + STK_SIZE - 4, CLONE_FS | CLONE_FILES | CLONE_VM, NULL);
```


Creare thread in Windows

```
int _beginthread(
   void( *start_address )( void * ),
   unsigned stack_size,
   void *arglist
   );
```

... ma c'è anche CreateThread con 6 parametri ...

POSIX threads

Problema: molti linguaggi non prevedono i thread (C, C++, ...)

Le syscall per i thread non sono standard

POSIX threads: uno standard per scrivere programmi portabili

Possono essere user-level o kernel-level (implementation detail)

In Linux: kernel-level

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

creates a new thread of control that executes concurrently with the calling thread — manuale di pthread_create(3)

Esempio con pthread_create

```
#include <pthread.h>
#include <unistd.h>
#include <stdio.h>
void * fn (void *pv) {
  int i;
  char * msg = pv;
  for (i=0; i<3; i++) {
    printf("%s\n", msg);
    sleep(1);
  }
  return NULL;
}
main() {
  pthread_t id0, id1;
  int err;
  err = pthread_create(&id0, NULL, fn, "Sono il primo!");
  err = pthread_create(&id1, NULL, fn, "Sono il secondo!");
  return 0;
}
```


L'output non corrisponde a quello che ci aspetteremmo...

```
$ gcc -Wall pthread_create_example.c -lpthread
$ ./a.out
Sono il primo!
Sono il secondo!
$
```

Solo due scritte, poi termina

Il thread principale termina subito
⇒ occorre attendere che tutti i thread terminino


```
int pthread_join(pthread_t th, void **thread_return);
```

suspends the execution of the calling thread until the thread identified by th *terminates* — manuale di pthread_join(3)

```
main() {
  pthread_t id0, id1;
  int err;
  void * retval;
  err = pthread_create(&id0, NULL, fn, "Sono il primo!");
  err = pthread_create(&id1, NULL, fn, "Sono il secondo!");
  pthread_join(id0, &retval);
  pthread_join(id1, &retval);
  return 0;
}
```

Ora funziona...

```
$ gcc -Wall pthread_create_example.c -lpthread
$ ./a.out
Sono il primo!
Sono il secondo!
Sono il primo!
Sono il secondo!
Sono il primo!
Sono il secondo!
$
```


Thread-Local storage (ovvero Thread-Specific Data)

Creare una nuova variabile TLS:

```
int pthread_key_create(pthread_key_t *key,
void (*destr_function) (void *));
```

Leggere e scrivere il valore della variabile:

```
int pthread_setspecific(pthread_key_t key, const void *pointer);
void * pthread_getspecific(pthread_key_t key);
```

Esempio:

```
pthread_key_t pippo;
pthread_key_create(&pippo, NULL);
...
pthread_setspecific(pippo, 123);
```


Come mantenere compatibilità con errno?

due problemi:

assegnare ad errno: `errno = 123;`

leggere errno: `printf("%d\n", errno);`

Frammento di /usr/include/bits/errno.h

```
#  if !defined _LIBC || defined _LIBC_REENTRANT
/* When using threads, errno is a per-thread value.  */
#   define errno (*__errno_location ())
#  endif
```

dunque i due frammenti di prima diventano

```
(*__errno_location()) = 123;
printf("%d\n", (*__errno_location()));
```

La variabile TLS contiene *l'indirizzo* di una var intera

```
int * __errno_location() {
  int * p;
  pthread_getspecific(errno_key, &p);
  return p;
}
```

Esercizi!

Scrivere programmi C che sfruttino

a) tutte le syscall che abbiamo visto

b) i thread con la libreria pthreads

fateli!!!

Nuovo argomento

Comunicazione fra thread

Un ciclo che incrementa `global` fino a 10 (?!)

```
int global = 0;
fn () {
  while (global < 10) global++;
}
```

Cosa succede se `fn` è eseguita da più thread?

Una esecuzione "sfortunata"

Thread A	**Thread B**
test: `global < 10`?	

Una esecuzione "sfortunata"

Thread A	**Thread B**
test: `global < 10`?	
	test: `global < 10`?

Una esecuzione "sfortunata"

Thread A	**Thread B**
test: `global < 10`?	
	test: `global < 10`?
sì, dunque eseguo `global++`	

Una esecuzione "sfortunata"

Thread A	**Thread B**
test: `global < 10`?	
	test: `global < 10`?
sì, dunque eseguo `global++`	
	sì, dunque eseguo `global++`

risultato: global == 11 !!!

Race condition

Definizione

una *race condition* si ha quando
il risultato di una computazione concorrente
dipende dalla velocità di esecuzione dei processi

Le race condition sono un *grosso* problema

Il difetto si manifesta "raramente"

... magari solo in produzione!

Difficili da debuggare perché difficili da riprodurre

Possono solo capitare in programmi concorrenti

Il disastro del Therac-25

Macchina da radioterapia

Prima macchina del suo genere interamente controllata in software

Rilasciata nel 1983

Uso sospeso nel 1987

tre morti per sovradosaggio; e tre feriti

causa del problema: una *race condition*

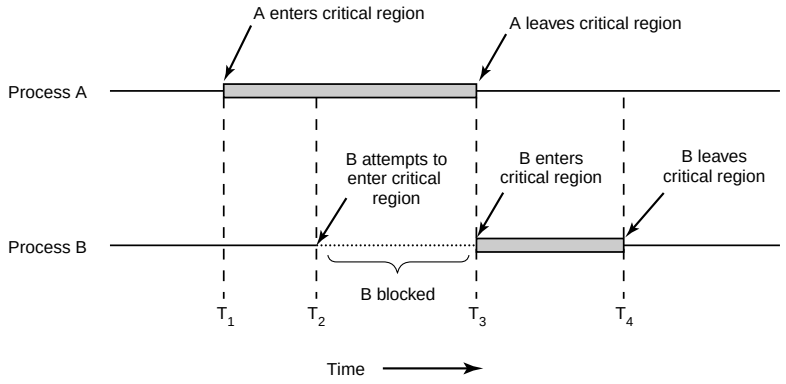
Abbiamo bisogno della *mutua esclusione*

regione critica: le parti di programma che accedono alla risorsa condivisa

Quattro condizioni

1. non più di un thread all'interno della regione
2. nessuna assunzione sulle velocità dei processi
3. nessun processo fuori della sezione critica può impedire ad un altro di entrare
4. accesso alla sezione critica consentito entro un tempo finito

Mutua esclusione con regioni critiche



Soluzione: disabilitare gli interrupt

può essere fatto solo in kernel mode

solo per brevissimi periodi!

e se ho più di una CPU?

⇒ solo nel kernel, solo per poche istruzioni

Non-soluzione con variabile di lock

```
int g_lock = 0; /* variabile globale */
...
/* codice eseguito da tutti i thread: */
while(1) {
  DoSomeWork();
  while (g_lock != 0)
    ;   /* busy wait */
  g_lock = 1;
  CriticalSection();
  g_lock = 0;
}
```

- Purtroppo, NON funziona!

Quasi soluzione: accesso alternato

```
/* var globale */
int g_turn = 0;

/* codice del thread n. 0 */
  while(1) {
    DoSomeWork();
    while (g_turn != 0)
      ;  // busy wait
    EnterCriticalSection();
    g_turn = 1;
  }
```

```
/* codice del thread n. 1 */
  while(1) {
    DoSomeWork();
    while (g_turn != 1)
      ;  // busy wait
    EnterCriticalSection();
    g_turn = 0;
  }
```


Quasi-soluzione: accesso alternato

Garantisce la mutua esclusione;

ma permette a un thread fuori CS di impedire a un altro l'accesso alla CS

Viola la condizione **??**, quindi *non* è una buona soluzione!

Una soluzione hardware

- Istruzione TSL reg,addr (Test and set)
  - copia il contenuto della cella addr nel registro reg
  - mette nella cella addr il valore 1
  - tutto ciò in maniera atomica

```
int tsl(int *g) {
  int old = g;
  g = 1;
  return old;
}
...
/* codice eseguito da ogni thread: */
while(1) {
  DoSomeWork();
  while (tsl(&g_lock) != 0)
    ;   /* busy wait */
  EnterCriticalSection();
  g_lock = 0;
}
```


Il busy wait è cattivo

- Peterson e TSL usano busy wait
- Il busy wait sciupa CPU
- Priority inversion
- Preferire l'uso di primitive fornite dal kernel
- Per es.: Suspend e Resume