Processi

Un processo è un entità DINAMICA generata da un programma

è una serie di attività controllati da uno scheduler

Requirement of processes

- Possibilità di esecuzione alternata (compito di scheduler)
- Garanzia di risorse e policy di allocazione
- Creazione di processi utente e comunicazione inter-process

Da cosa è composto

- Codice
- set di dati
- stato

⟹

- id
- stato
- priorità
- P.C.
- puntatori
- stato I/O

tutte le informazioni di processo sono salvate nel Process Control Block, non aggiornate in real time.
È creato e gestito dall'OS, così da poter gestire avvii e arresti del processo

System call UNIX di gestione processo

fork() wait() exit()

da un processo si vuole creare un nuovo modulo di gestione, in modo che possano essere eseguite CONCORRENTEMENTE
il processo figlio è DUPLICATO del processo padre

da molteplici fork si crea un process tree

con la wait() è solitamente il padre ad aspettare il figlio

durante la fork il processo si sdoppia in padre e figlio

fork()

return:
- 0 se è il figlio
- pid se è il padre, pid è quello del figlio
- -1 se errore (fine risorse)

```
ret= fork()
switch (ret){
  case -1:
      return for
}
```

il processo figlio credita tutto
- file aperti
- RAM
- registri

exit(status)

la system call termina l'esecuzione del figlio poi:

- esegue la funzione specificata in atexit(fun) e on_exit(fun), con la differenza che atexit passa parametri
- fa l'fflush di tutti gli stream aperti
- chiude tutti i files aperti, ma non quelli condivisi
- chiama _exit()

La funzione _exit(status)

- dealloca memoria
- se il processo ha figli, assegna ad init
- check se padre è vivo
- se è vero tiene il valore finché il padre lo richiede, passando ad uno stato Zombie fino al wait del padre
- se padre è morto allora il processo muore

THREAD

Un processo ha due caratteristiche
- scheduling/esecuzione
- ownership

Il thread è l'unità che è assegnata all'esecuzione e allo scheduling del processo

Durante l'epoca dell' MS-DOS vi era un solo processo e un solo thread
Si punta ad un OS in MULTI THREADING (più finestre / programmi alla volta)

Un altro, es JAVA, è ad un UNICO processo multi thread

Nel caso multithread e multi-processor c'è bisogno di

- SPAZIO VIRTUALE di MEMORIA
- Accesso protetto a:
  - file
  - altri processi
  - memoria

Ogni thread ha:

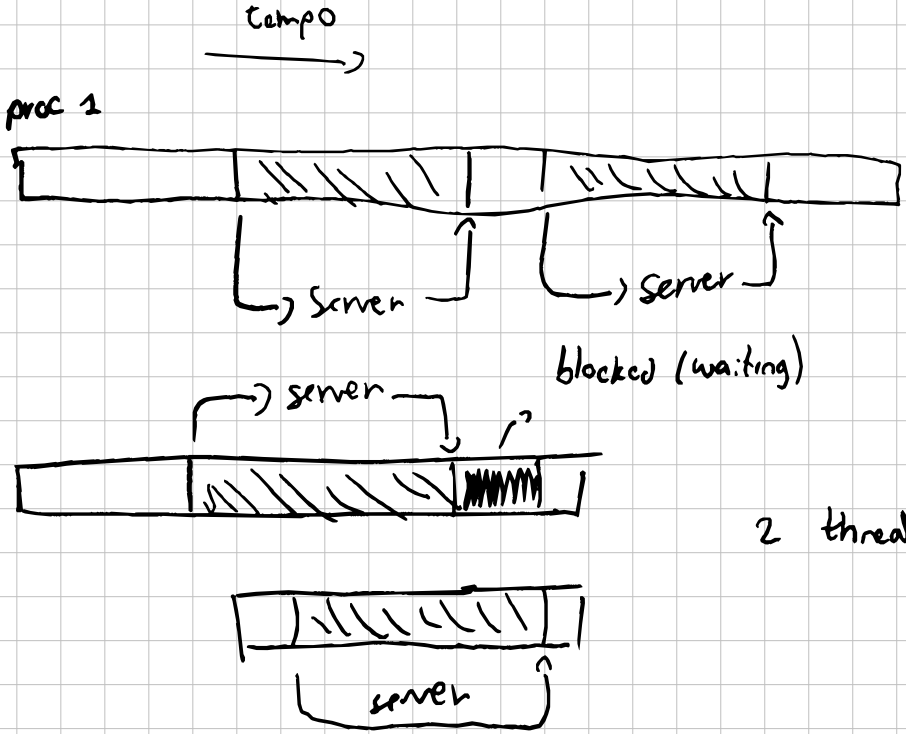
- stato di esecuzione (running, ready, blocked)
- contesto del thread (TCB)
- stack di esecuzione
- spazio di memoria per variabili locali

VANTAGGI THREAD:

- I thread possono comunicare meglio l'un l'altro
- lo switch del thread richiede meno TEMPO di uno relativo dei processi
- perfetto per processi asincroni (es. mentre si aspetta file non blocca l'app)
- semplifica la modularità dell'applicazione

Esempio di multithread

Chiamata RPC



con multithread è possibile avere un organizzazione più precisa e "veloce" dei processi che possono ad esempio delegare richieste I/O senza che un programma si blocchi

↳ UTILIZZO DI TIME FRAME

IMPLEMENTAZIONI

- User - Level Thread (ULT)
- Kernel - Level Thread (KLT)

ULT

La differenza è che

- Tutta la gestione thread è in mano all'applicazione
- Il Kernel non è a conoscenza dei thread creati

GUARDARE slide 1.48

KLT

- Il kernel mantiene informazioni di contesto sul thread, e non da la gestione in mano all'applicazione
- Lo scheduling è fatto thread - basis

windows è quel tipo

Perche scegliere uno o l'altro?

ULT

Vantaggi

- Lo scheduling è in mano unicamente all' app
- Non è necessaria un'elevazione di privilegi
- Gira su ogni os, è implementato attraverso librerie

Svantaggi

- Una syscall bloccante su un thread blocca tutti i thread!
- Non viene sfruttata l'architettura multicore/multithread del processore

KLT

Vantaggi

- Il kernel può far girare più thread in contemporanea sfruttando più core
- Se un thread è bloccato ne può far girare un altro dallo stesso processo
- Le routine del kernel possono essere messe in multithread

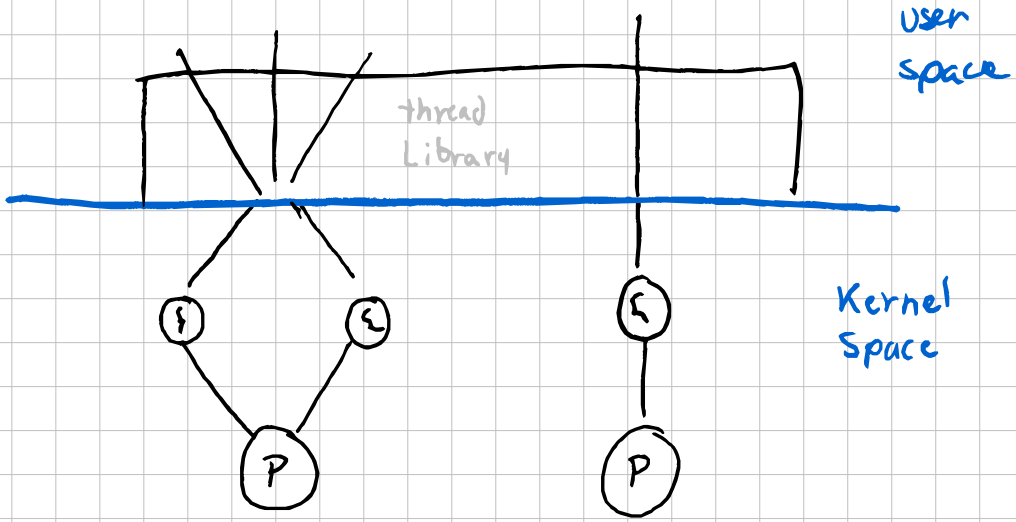
Svantaggi

- Il trasferimento da un thread ad un altro ha bisogno un mode switch al kernel

C'è la possibilità di avere un approccio combinato

.. APPROCCIO COMBINATO

- La creazione dei thread è fatta a livello USER-SPACE
- La maggioranza delle sincronizzazioni dei thread è fatto dall'applicazione
- un numero u di ULT sono mappati in k di KLT



Un possibile arrangiamento Thread processo sono nelle slide 1.55

POSIX THREADS (Pthread)

- esiste uno standard di creazione e gestione thread, appunto PTHREADS
  I thread sono anche molto più performanti del semplice fork
- Sono tipi definiti nell' Header <Pthread.h> in C

Master / slave model

Una funzione, "Mapper", assegna compiti ai thread "worker"

La DIPELINE è quello su cui vengono fatti girare in contemporanea : thread, seguendo : time slice

I THREAD hanno accesso alla stessa aria di memoria, ma hanno delle zone protette

Thread safety

Un codice si dice "thread safe" quando più thread possono essere eseguiti assieme senza problemi.

↳ Necessaria sincronizzazione porta a potenziali problemi di accesso simultaneo

IMPLEMENTAZIONE e CREAZIONE

Si usa come detto, la libreria pthread

- pthread_create() crea un nuovo thread e lo rende eseguibile.
  Ci sono limiti implementativi
  I thread sono capaci di creare altri thread

Come terminare un thread?

- Il thread ha finito il lavoro, return
- pthread_exit() viene chiamato quando il thread ha completato il lavoro e viene distrutto
  N.B. Non c'è garbage collection
- pthread_cancel()