Processi e Threads

Processi

Ogni attività può essere rappresentata tramite **processi**

I processi possono comunicare tra di loro perché possono aver bisogno di alcune risorse che gli dovranno essere fornite (es. dati); ci saranno delle policy per garantire l'accesso a chi ne ha il permesso

Un processo sarà formato da:

- codice di programmazione
- insieme di dati
- attributi che indicano il suo stato *durante l'esecuzione*

Un processo in esecuzione ha:

- identificatore dato dal sistema operativo (unico per ogni processo)
- stato
- priorità
- program counter → posizione dell'istruzione che deve essere eseguita in questo momento
- puntatori alle aree di memoria
- context data → salvataggio dei registri della CPU; serve perché se un processo viene interrotto salvo qui "a che punto stavo" per poi essere ricopiato quando viene ripreso il processo
- informazioni su dispositivi I/O
- informazioni di utilizzo varie

Tutto questo è salvato nel Process Control Block; creato dal sistema operativo al momento della creazione del processo; permette al sistema operativo di gestire più processi

Come si crea e gestisce un nuovo processo? 3 comandi principali:

1. *fork()*
2. *wait()*
3. *exit()*

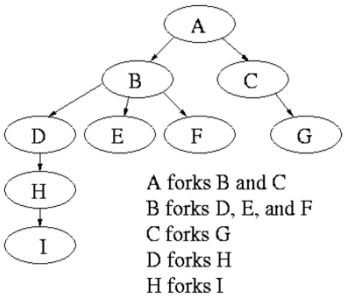
Un processo nasce sempre da un altro; il processo padre chiede al S.O. di creare un processo figlio, che è un duplicato quasi esatto del padre. I due processi diventano concorrenziali (N.B. non in parallelo)

Se ho un computer con 1 solo core solo uno dei due processi può essere eseguito (o padre o figlio); se anche ho più core il sistema operativo può comunque dover bloccare uno dei due

➔ non posso prevedere a priori ciò che succederà quindi devo programmare bene per poter gestire la sincronizzazione tra processi

Dalle varie fork risulta un albero di processi (Process Tree)

Ogni padre si "ricorda" dei propri figli (diretti); in caso della terminazione di un processo padre, il sistema operativo assegna i figli ad un altro processo → per evitare di terminare il processo padre prima dei figli (e quindi il processo padre potrebbe non acquisire i dati dei figli) uso *wait()*



➔ se A fa la *wait()* può aspettare che finisca o B o C

Quando accendo un computer ho un programma iniziale (bootstrap program) che inizializza i registri della CPU e la memoria, carica il sistema operativo e lo avvia

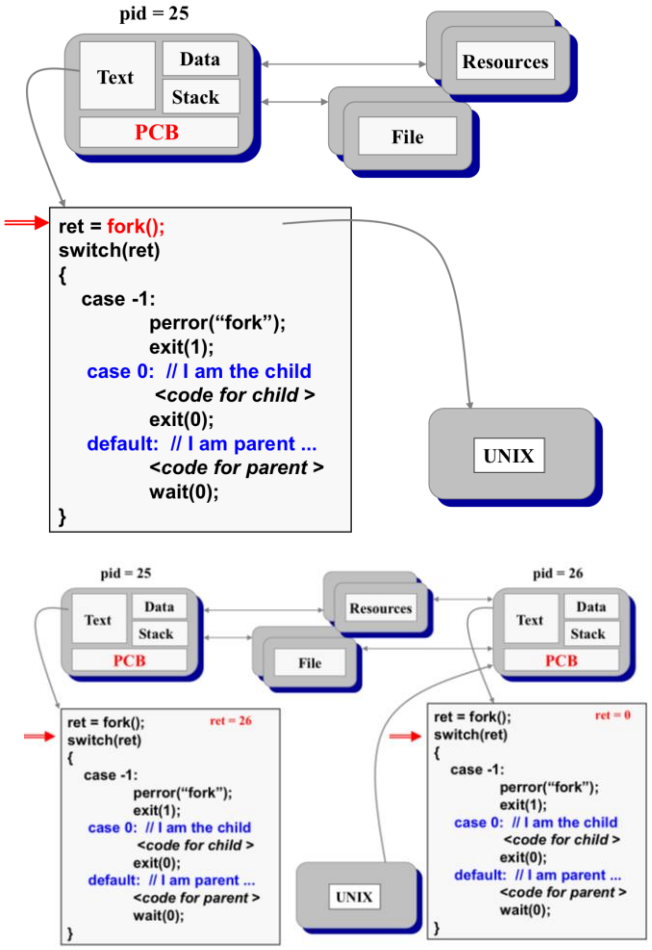
Il sistema operativo farà partire il primo processo ("init"); il sistema operativo poi aspetterà degli eventi (es. click sull'icona di Word → avvia il processo)

La fork crea una una copia quasi esatta del processo:

- la funzione fork restituisce per il padre l'id del processo figlio, per il figlio restituisce 0 → il padre sa l'id del figlio, il figlio non sa l'id del padre
- il figlio eredita la copia della memoria (es. inizializzazione di variabili), i registri CPU, tutti i file aperti dal padre
- Il contesto di esecuzione nel PCB è una copia del contesto del padre al momento della chiamata; il PCB avrà diverso id e potrebbero avere differente stato

➔ come funziona la fork?

dal momento in cui il sistema operativo riceve la fork interrompe l'esecuzione del processo padre, crea una copia esatta della memoria (ma pid diverso) e anche del Process Control Block; le risorse del padre saranno ereditate al figlio (problemi di concorrenza)



nello switch avrò il risultato del fork (il processo capisce chi è):

1) caso -1: ci sono degli errori per cui la fork è fallita (difficile che avvenga però bisogna metterla perché devo sapere come gestire gli eventuali errori)
2) caso 0: figlio
3) default: padre

Se la fork è usata per creare un nuovo processo che deve eseguire un nuovo programma utilizzo il comando *exec()* → quando eseguito il sistema operativo rimpiazza la attuale immagine di processo (testo, dati, stack) con una nuova; il programma dovrà avere un main e non avere valori di ritorno

Per finire l'esecuzione, il figlio chiama *exit(status)*; con questa chiamata il sistema operativo:

- salva lo status della exit (di solito per indicare se è terminato correttamente)
- esegue tutte le funzioni specificate con *atexit(*fun*)* e *on_exit(*fun*)*
- esegue *fflush()* cosicché se c'erano delle print vengono eseguite
- chiude tutti i file e le connessioni (non condivise con altri processi)
- chiama *_exit(status)* → con questa chiamata il sistema operativo:
  - salva lo status
  - dealloca memoria
  - se il processo ha figli li assegna a init

- controlla se il padre è vivo:
  - se è vivo, mantiene il valore finché il padre non arriva alla wait (il figlio non muore ma diventa zombie)
  - se non è vivo, il figlio termina (cancello tutto)

Il padre potrebbe voler aspettare che i figli abbiano finito:

- *wait()* → il padre aspetta che uno qualsiasi dei figli abbia finito; la funzione restituisce l'id del figlio o -1 se non ci sono figli (ossia i figli sono già usciti)
- *waitpid()* → il padre aspetta che un figlio in particolare termini

➔ in entrambi i casi posso mettere un puntatore a intero *int* status* che indicherà lo stato di uscita del figlio

Gestione dei processi in Python → *import os* per lavorare con il sistema operativo; i comandi sono simili

```
import os

ret = os.fork()
if ret==0:
        # I am the child
        <code for child>
        os._exit(0)
else:
        # I am parent ...
        <code for parent >
        os.wait(0);
< ... >
```

Thread

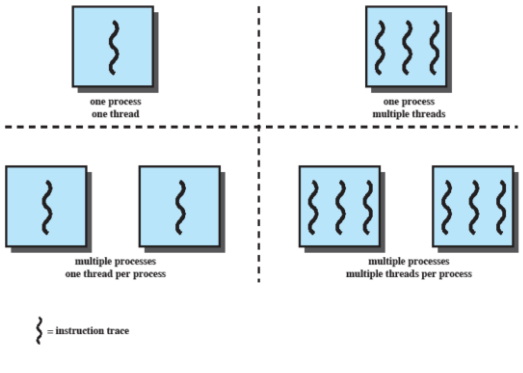
Un processo ha 2 caratteristiche:

1. proprietà delle risorse → il processo possiede un indirizzo virtuale per la propria immagine di processo (dati, risorse, …); l'unità del sistema operativo che la gestisce è detta **task**
2. esecuzione → il processo segue dei comandi che lo possono portare a interagire con altri processi; l'unità del sistema operativo che la gestisce è detta **thread**

➔ il sistema operativo gestisce le due caratteristiche in maniera indipendente

Posso avere più casi:

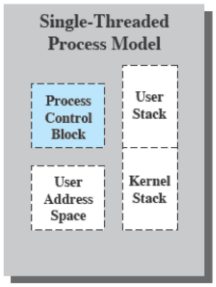
- 1 processo alla volta con 1 thread (es. MS-DOS)
- 1 processo alla volta con più threads (es. ambiente di esecuzione Java)
- più processi alla volta con un solo thread (es. macchine IoT che devono fare task semplici ma hanno poca memoria → il multi-thread richiede una gestione dei thread che occupa troppo in termini di memoria)
- più processi alla volta con più threads (es. maggior parte delle macchine moderne)

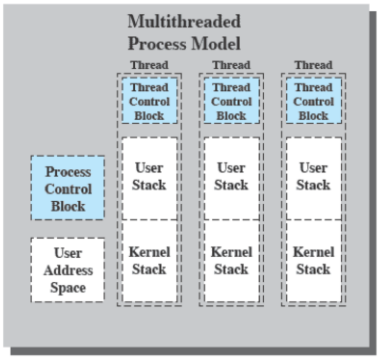


In un sistema operativo che supporta il multi-thread:

- c'è uno spazio degli indirizzi di memoria con l'immagine del processo (che a sua volta avrà più thread)
- è garantito l'accesso protetto ai processori, ad altri processi, ai file e alle risorse di I/O

Quando ci sono 1+ thread in un processo, ogni thread ha:





- stato di esecuzione (running, ready, blocked)
- context data salvato quando non è running
- stack di esecuzione
- aree di memoria statiche per le variabili locali
- accesso alla memoria e alle risorse del suo processo (condiviso con gli altri eventuali thread)

➔ differenza con i processi: nei processi la memoria tra padre e figlio non è condivisa ma copiata

posso vedere un thread come un program counter che opera all'interno di un processo

➔ perché usare thread?

1. modularità
2. più veloci da creare e terminare
3. switching tra i thread è più veloce
4. i processi sono indipendenti (comunicazione tramite sistema operativo che deve fermare i processi per gestire questa comunicazione) mentre i thread parlano tra loro senza chiamare il kernel
5. avere più lavori in esecuzione (background e foreground; es. test della rete mentre compio un altro task)
6. velocità di esecuzione

Ci sono una serie di azioni che possono influire sulle attività di un thread → il sistema operativo deve gestirle (es. la sospensione di un processo implica la sospensione di tutti i suoi thread perché hanno lo stesso spazio degli indirizzi di memoria)

3 stati di esecuzione per i thread:

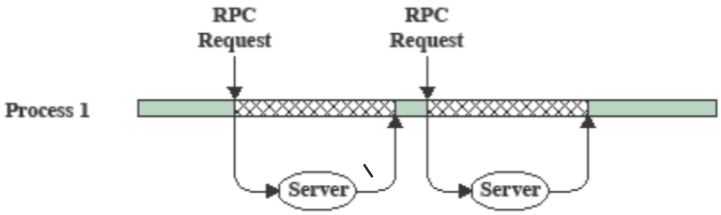
1. running
2. ready
3. blocked

Per cambiare lo stato dei thread posso:

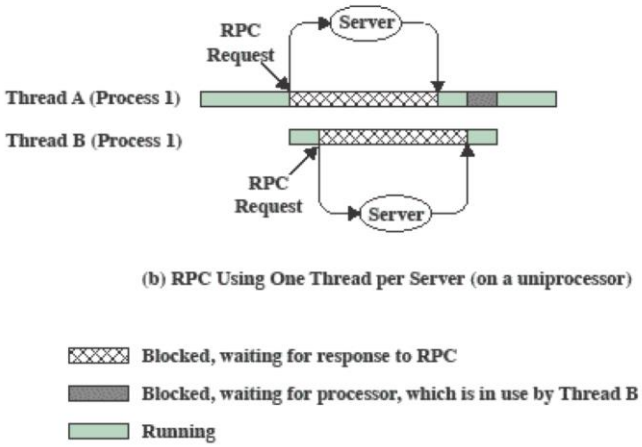
- creare un altro thread
- bloccare il thread (posso bloccare il singolo thread o il processo; es. attesa informazione da altro thread blocca il singolo thread / chiamata a sistema operativo blocca l'intero processo)
- sbloccare il thread
- chiudere il thread

es. programma con 2 Remote Procedure Call (chiamate di funzioni in esecuzione su un server) che vadano su due host differenti e che combinano alla fine il risultato

➔ usando un singolo thread:



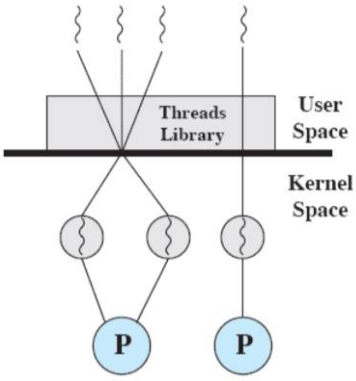
➔ 1 thread per server:



Come gestisco i thread?

- **User Level Thread (ULT)** → gestiti dall'applicazione; es. per gestione: Pthread in C; caratteristiche:
  - non viene mai chiamato il kernel (che vede un singolo processo → non mi dà più core ma **singolo core**)
  - posso scegliere la politica di scheduling
  - possono essere caricati su qualsiasi sistema operativo (si tratta di caricare una libreria)
  - chiamata bloccante blocca tutti i thread
- **Kernel Level Thread (KLT)** → gestiti dal kernel (sistema operativo); caratteristiche:
  - mantiene le informazioni di contesto
  - fa scheduling
  - il kernel può fare scheduling di più thread di un processo su più core
  - se un thread è bloccato, il kernel può dare il core non utilizzato ad un altro thread dello stesso processo
  - l'intervento del sistema operativo può portare ritardi

➔ posso anche avere un approccio combinato



Threads:Processes	Description	Example Systems
1:1	Each thread of execution is a unique process with its own address space and resources.	Traditional UNIX implementations
M:1	A process defines an address space and dynamic resource ownership. Multiple threads may be created and executed within that process.	Windows NT, Solaris, Linux, OS/2, OS/390, MACH
1:M	A thread may migrate from one process environment to another. This allows a thread to be easily moved among distinct systems.	Ra (Clouds), Emerald
M:N	Combines attributes of M:1 and 1:M cases.	TRIX

Pthread

I Pthread sono dei tipi del linguaggio C inclusi nella libreria pthread.h

- sono standard POSIX
- hanno bisogno di meno risorse di sistema per essere eseguiti
- sono più veloci

Platform	fork()			pthread_create()		
	real	user	sys	real	user	sys
Intel 2.6 GHz Xeon E5-2670 (16 cores/node)	8.1	0.1	2.9	0.9	0.2	0.3
Intel 2.8 GHz Xeon 5660 (12 cores/node)	4.4	0.4	4.3	0.7	0.2	0.5
AMD 2.3 GHz Opteron (16 cores/node)	12.5	1.0	12.5	1.2	0.2	1.3
AMD 2.4 GHz Opteron (8 cores/node)	17.6	2.2	15.7	1.4	0.3	1.3
IBM 4.0 GHz POWER6 (8 cpus/node)	9.5	0.6	8.8	1.6	0.1	0.4
IBM 1.9 GHz POWER5 p5-575 (8 cpus/node)	64.2	30.7	27.6	1.7	0.6	1.1
IBM 1.5 GHz POWER4 (8 cpus/node)	104.5	48.6	47.2	2.1	1.0	1.5
INTEL 2.4 GHz Xeon (2 cpus/node)	54.9	1.5	20.8	1.6	0.7	0.9
INTEL 1.4 GHz Itanium2 (4 cpus/node)	54.5	1.1	22.2	2.0	1.2	0.6

(Runtime con 50000 operazioni)

Per sfruttare i Pthread, un programma deve essere organizzato in task indipendenti e concorrenti

Quando progetto un programma con più threads, ci sono 2 possibili modelli di sviluppo:

- Manager/Worker: un thread manager gestisce gli input e assegna il lavoro agli altri thread (1+ thread worker)
- Pipeline: divido il task in tanti sotto-task che vengono eseguiti in serie ma concorrenzialmente da thread differenti

Chi programma deve far attenzione all'accesso alla memoria condivisa e che essa sia sincronizzata

➔ un programma è thread-safe quando più thread possono essere eseguiti simultaneamente senza interazioni inattese

es. no sovrascrizioni, no gare tra thread

Come creare un thread con Pthread? *pthread_create()* crea un nuovo thread e lo rende eseguibile; i thread creati sono peers e possono creare a loro volta altri thread

Come termino un thread con Pthread? ho più modi:

- il thread è completo → la funzione che lo ha chiamato fa return
- *pthread_exit()* → il thread ha completato il suo lavoro; questa chiamata non libera le risorse
- *pthread_cancel()* da un altro thread
- *exit()* → chiude il processo!
- il main termina senza aver eseguito *pthread_exit()* [se però eseguo *pthread_detach()* non termina]

In Java lavoro con i thread in 2 modi:

- faccio una sottoclasse di Thread
- faccio una classe che implementa runnable

➔ il codice del thread è nella ridefinizione del metodo *run()*

creo un nuovo thread con *new()*; eseguo thread con *start()* [che richiama *run()*]; un thread può aspettarne un altro con *join()*

Symmetric MultiProcessing (SMP)

Il computer è sempre stato visto come una macchina sequenziale (processore esegue le istruzioni in maniera sequenziale) ma si può usare anche il parallelismo

Tassonomia di Flynn

- Single Instruction Single Data → un singolo processore esegue una singola istruzione per operare con dati immagazzinati in una singola memoria
- Single Instruction Multiple Data → ogni istruzione è eseguita su differenti insiemi di dati da processori diversi
- Multiple Instruction Single Data → una sequenza di dati è trasmessa a un insieme di processori, ognuno dei quali esegue una diversa sequenza di istruzioni
- Multiple Instruction Multiple Data → un insieme di processori eseguono simultaneamente delle diverse sequenze di istruzioni su differenti insiemi di dati

Classica organizzazione di un SMP: