

Processo

Intenzione di esecuzione di un programma

Cosa ha un proprio stato, si risponde alle interazioni e lo stato evolve.

A un processo succede:

- codice
- dati
- stack

Spazio festi intituzz
(rantizionamento delle
RAM riservate all'intizzo)

Copia registro CPU

File aperti (ancora il
file pointer)

alarm globale PTR

Process componenti

Spazio lib
inizializzato

Max



Traono queste
informazioni
nella Tabella
dei processi

Stack e Heap sono le aree di memoria che crescono, lo Stack è memoria statica mentre l'Heap è dinamica.

Crescendo gli address tocconi

DATA = dati già contenuti all'intante 0, è una area di memoria statica

Il File pointer è una testina che si muove dentro al file

Tabelle dei Processi

Ci sono Tabelle di tutti i processi in esecuzione.
Qui sono contenute le informazioni in file seguenti, strettamente legate ai processi implementati.

Lo immaginiamo come una Tabella lineare di processi, a ogni processo è associato il Process Control Block che è la sua "ID"

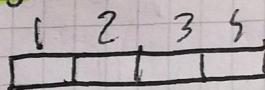
La Tabella è gestita in maniera dinamica.

Dentro al record del processo trovo o le informazioni in maniera diretta o puntatori alle informazioni.

In pratica il PCB è un punto di accesso a tutto quello che c'è nel processo.

Potremo considerare come identificativo del processo l'indice della Tabella.

L'identificativo è il PID



I sistemi sono multiprogrammati per cercare lo pseudo-parallelismo, dunque si implementa quello che è il concetto di CPU virtuale.

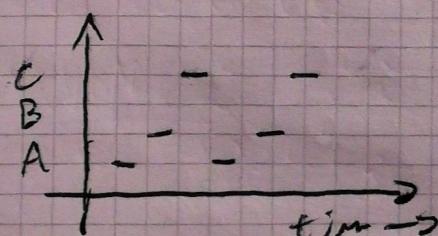
Invece di eseguire i processi in maniera sequenziale facciamo a "turno".

Ma se il processo A e B usano contemporaneamente la CPU come si gestiscono i registri?

Nel PCB c'è appunto la lista dei registri.

E se sequenziale: $A \rightarrow B \rightarrow C \rightarrow D \dots$

Pseudo-parallelismo



Inglese mentito il concetto di CPU virtuale, ogni processo vede la CPU come se gli fosse interamente dedicata.

• Process switch: "Cambiare" processo a cui si debba la CPU

Creatiōne Processo

Come si crea?

Nella fase di boot/avvio del sistema

Da parte di un altro processo/utente

Metodi di creazione

• Sbaggiamento padre: fork e exec (UNIX)

• Creazione attorno al programma: CreateProcess (WIN)

↳ Processo base: INIT, si occupa della gestione di tutti i processi, è sempre attivo.

Ero è l'antenato di tutti i processi

(In questo albero se un processo è creato è aggiunto come figlio di chi lo ha creato)

E' cre l'albero di processi con radice l'INIT

Se termina un processo che aveva figli il processo INIT li eredita.

Perciò diciamo l'albero?

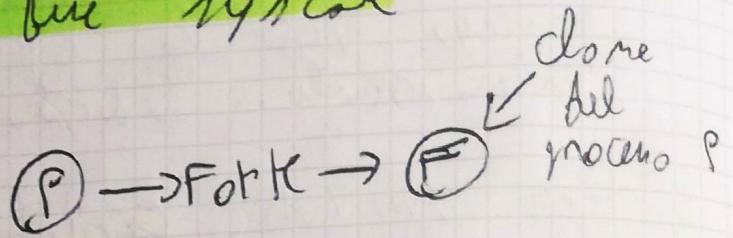
Perciò alcuni processi sono strettamente legati al padre, se il padre muore anche lui dovranno morire

Fork e exec: sono due sys call

Crea nuovo processo,

cosa è richiesto da un altro processo, non ha parametri.

Avviene una biforcazione



La Fork tocca 0 nel contesto del figlio e il PID nel contesto del padre per gestire la domenica.

If (fork() == 0)

FIGLIO

else

PADRE

In questo modo gestiamo cosa deve fare il figlio e cosa il padre.

EXEC

è una chiamata a sistema con dei parametri exec(c, arg1, arg2 ...)

La exec inizia il comando e ha varie di argomenti.

La exec riviste l'ambiente di esecuzione/risparmio di indirizzamento e gli piazza il C

L'exec usa lo spazio di indirizzamento per piazzare il codice chiamato C, preso dal DSO, ed eseguirlo con i vari parametri

La exec permette di eseguire un nuovo programma.

Si può usare la Fork e non la exec facendo attenzione alla comunicazione tra processi.

La CreateProcess di WIN è già completa da sola, così non molti processi.

Terminazione del Processo

- Uscita normale : exit (UNIX), EXIT Process (WIN)

Il processo regrada al SO che vuole auto terminarsi.

Exit status : è un valore che se è 0 regrade che l'uscita non ha avuto problemi,

VOLONTARIA

- Uscita su errore

VOLONTARIA

Il codice chiede di terminare perché se che non può eseguire svolto.

- errore critico

IN VOLONTARIO

Legge di anomalie di germezi, istruzioni illegali o rinviate, divisione per 0.

Alcuni errori critici sono gestibili, altri no.

Il processo gestirebbe richiedere di avviare una routine per gestire il problema.

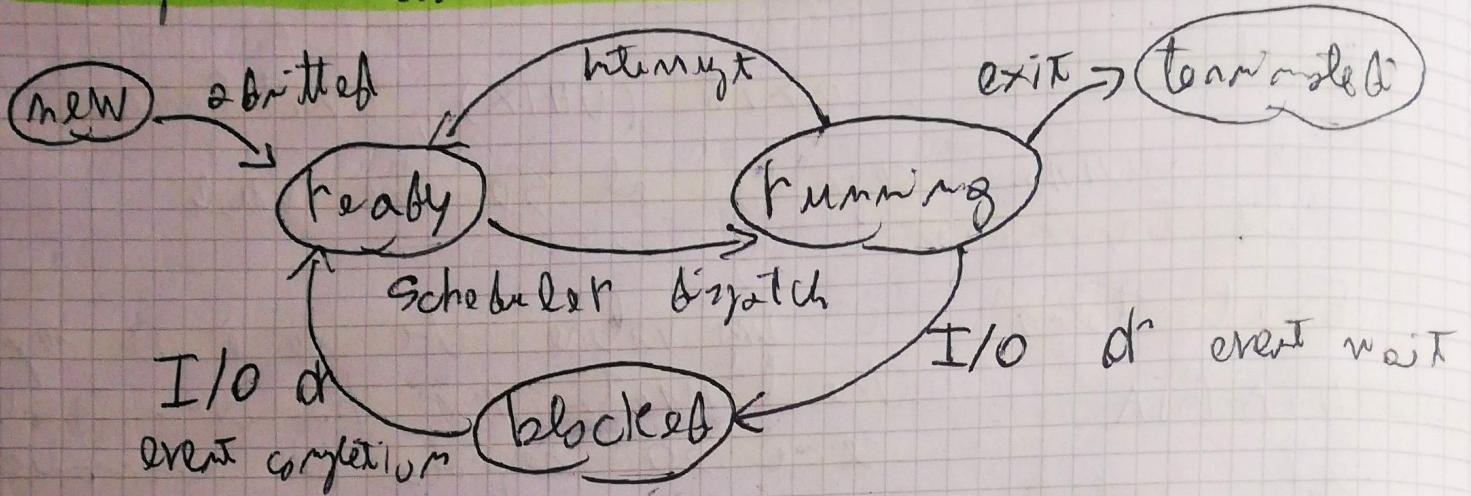
Se molti sono gestibili, questi errori causano chiunque interrompe il processo.

- terminazione da un altro processo

IN VOLONTARIO

metodi : kill (UNIX), terminate process (WIN)

Il processo come un automa ha uno stato



- **RUNNING** : CPU effettivamente in uso per il processo
- **Ready** : processo eseguibile ma attualmente in sospensione
- **Blocked** : bloccato finché non intrinseca un evento utente

Trasmissione

RUNNING → **Blocked** : il processo si blocca in attesa di input

RUNNING → **READY** : lo scheduler prende un altro processo

Ready → **Running** : lo scheduler prende questo processo

Blocked → **Ready** : input è ora disponibile

Schedulatore : componenti / algoritmo per la gestione dei processi

Dato che ci sono tanti processi in stato Ready
creare una lista di tali processi

Tabella dei processi

Abbiamo in ogni record il PCB che contiene
informazioni sul process

- state
- number
- program counter
- registers
- memory limits
- open files
- ...

Structures Tables

Process				
0	1	2	3	...
Scheduler				

Gestione interrupt per parcheggio di Processo:

- 1) Salva nello stack il PC e la PSW che sono nello stack attuale.
- 2) Si carica nel registro degli interrupt l'indirizzo della procedura associata.
- 3) Salva i registri e impila un nuovo stack.
- 4) Si esegue la procedura di servizio dell' interrupt.
- 5) Quando finisce l' interrupt si ripete con quale processo si è registrato.
- 6) Si aggiorna del PCB lo stato del processo.
- 7) Si riavvia il processo.

Come struttura finita per l'esecuzione i processi vengono de CODE

Un processo gestisce in modo più siste

Coda dei processi READY: sotto insieme delle tabelle di processi

Coda dei Disponibili

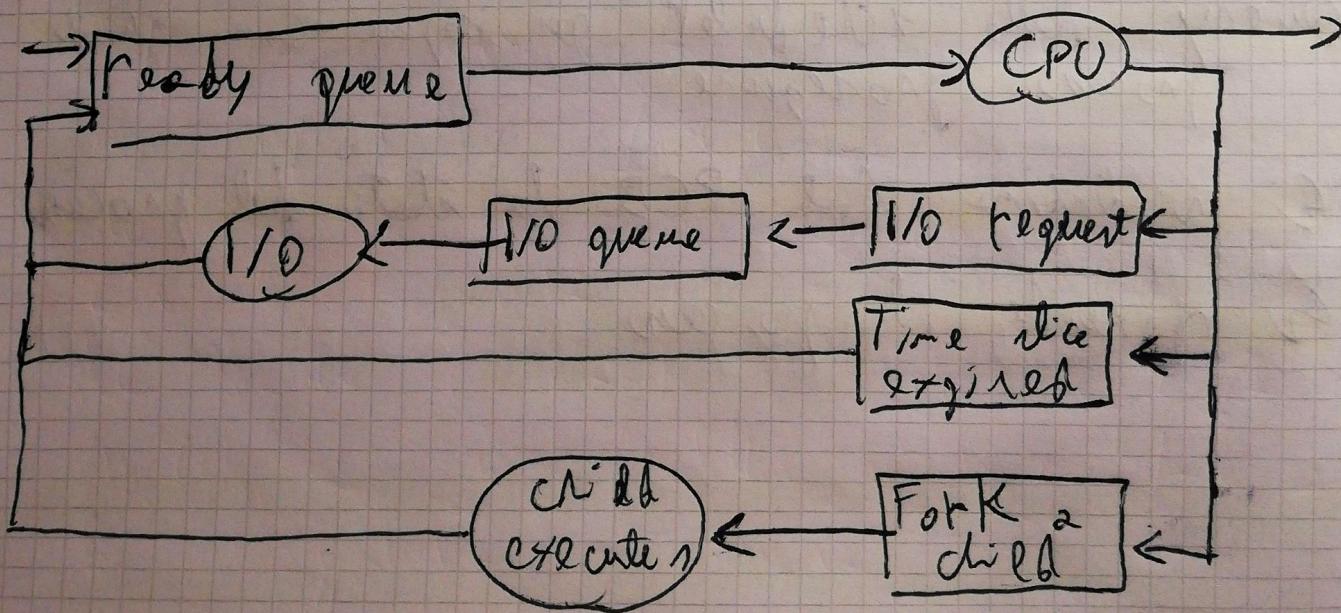
Coda verso cui altri processi stanno fatto tabelle di processi, e implementata con doppio LINKING

Coda con dei processi bloccati che devono essere usciti dai disponibili

Abbiamo una coda per ogni controller

Quindi il processo fornito è già pronto nella coda del controller, verrà fissato quando dovrà essere eseguito nella ready queue.

ACCODAMENTO (metti in cui avvengono gli inserimenti e estrazioni dalla coda)



Thread

Evoluzione del modello di processo

Facciamo combiniada e più flussi di esecuzione
lo stessi spazio di memoria.

Facciamo condivisi tutti i flussi in un unico processo

P1 P2 P3
① ② ③

P record unico

Qui ogni process ha
solo CPU virtuale

Qui ho già CPU virtuale
per un solo processo

Può essere utile aprire molti file nel vari flussi
senza usare di altri dati.

Thread: flussi di esecuzione. Nel 1° modello
ne abbiamo di più, nel 10 modelli
abbiamo già processi, ognuno con un solo
thread.

Un thread può creare un altro, ma non c'è
affidabilità, sono protelli

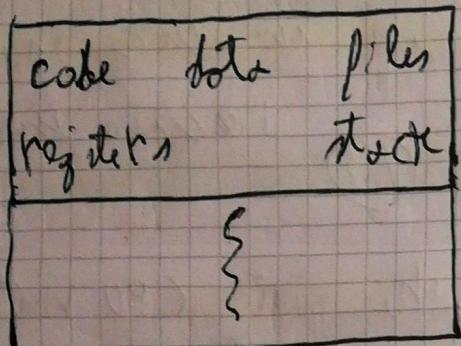
Vantaggio: se un thread fa una chiamata
bloccante, il SO non blocca tutto il
processo, cari nel 10 modelli ma solo
quel thread. (liberalmente)

Il modello a Thread cerca un modo
per avvicinare ancora di più il parallelismo

Thread è combinazione di

- PC, Register, Stack, file
- Il resto è tutto comune

2° modello



1° modello

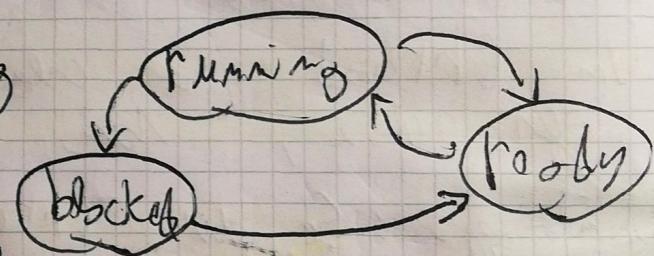
code	data	files
Reg.	Reg.	Reg.
stack	stack	stack
{	{	{

Ogni thread ha il suo PC fatti che siano
delle fare una certa routine.

E applica il comune degli stati anche ai thread
quindi viene fatto scheduling, con un cambio
di contesto più veloce.

Il problema dello scheduling
è rendere le rotte del

thread, non già del processo



Poi se abbiamo switchare un thread,
l'efficienza per cambiare immediatamente se i
due thread sono fratelli o no.

Dato che se si fa lo switch tra fratelli non
ci lo mappante in memoria, solo la gestione
dei register e stack, invece se si fa lo
switch tra non fratelli si deve fare anche
la mappatura in memoria.

Poiché molti thread in certi momenti si preferisce lo switch verso processi.

Thread: "processi leggeri"

Operazioni thread

thread-create: un thread si crea in altro

thread-exit: il thread chiamante termina

thread-join: un thread si incarica di un altro thread

thread-yield: il thread chiamante libera volontariamente la CPU.

Programmazione multicores

I thread permettono maggiore realibilità con cui che permettono multithreading e con sintesi con architetture multicore.

Single-core $\boxed{T_1 \ T_2 \ \overline{T_3} \ T_3 \ T_1 \ T_2 \ T_3 \dots}$

Multi-core 0 $\boxed{T_1 \ T_3 \ T_1 \ T_3 \ \overline{T_1} \dots}$

1 $\boxed{\overline{T_2} \ T_3 \ T_2 \ T_3 \dots}$

Nel single-core: esecuzione intercalata

Nel multicore: parallelo (sia più)

Il multithreading (o "genetizzazione") solo se i due thread caricati sulla CPU sono fratelli, poiché condividono lo stesso spazio di memoria

Applicazioni basate sul multi core comportano

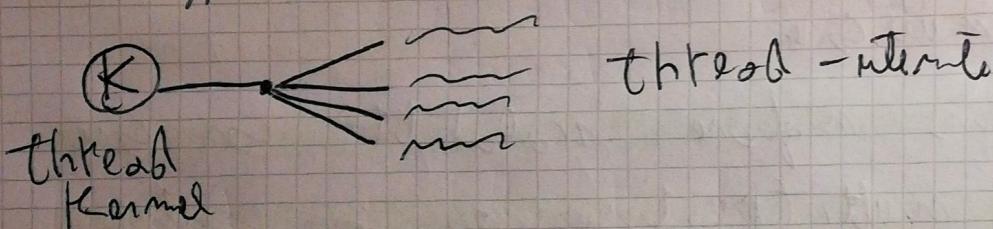
delle cose in più da ottimizzare

- Separare i task
- suddividere i dati
- test e debugging (già complicato)
- bilanciamento (di task)
- dipendenze dei dati

Threads a livello utente

- 1 o più thread

- può essere utile se il Kernel non supporta i thread



Ci sono librerie che implementano switch run-time che gestiscono le tabella di thread all processo

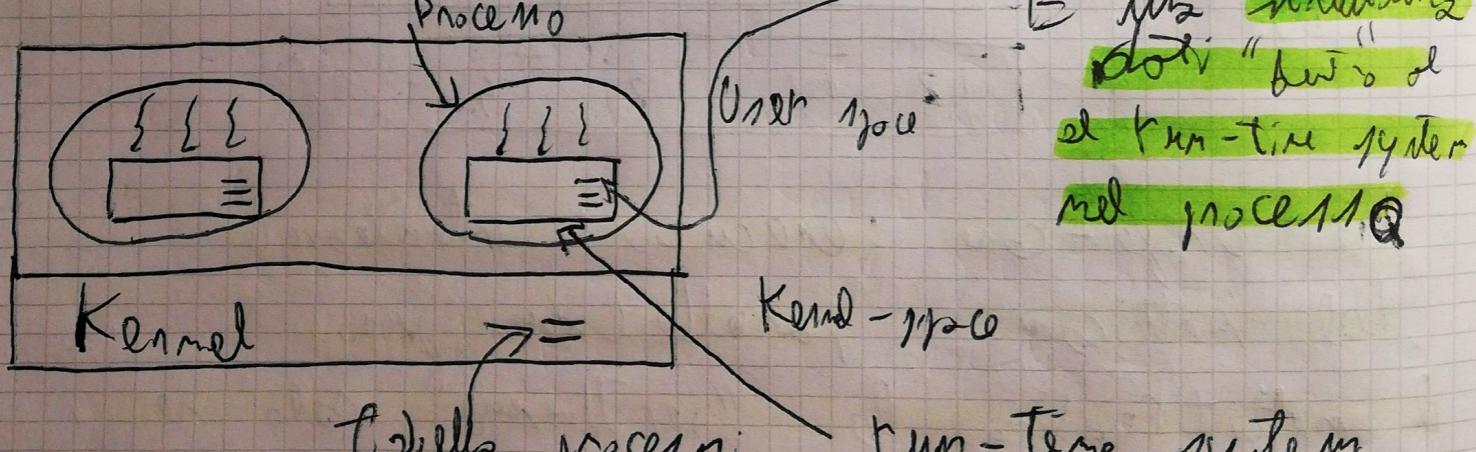


Tabella processi run-time system

Come avviene lo switch di thread se il Kernel non ne ha nulla? Su base spontanea, il thread prima o poi lascierà la CPU

Pro: Dispatching non richiede Traz = context switch
Schedulering generalizzabile

Contro: Chiamate bloccanti | Si blocca tutto il processore,
possibilità di non rilasciare CPU | fino che il kernel non fa
della modifica in thread

Selez. garante gestione
page-fault Mo chiamata a sister

Selez. garante di provare se una read operazione in
blocco

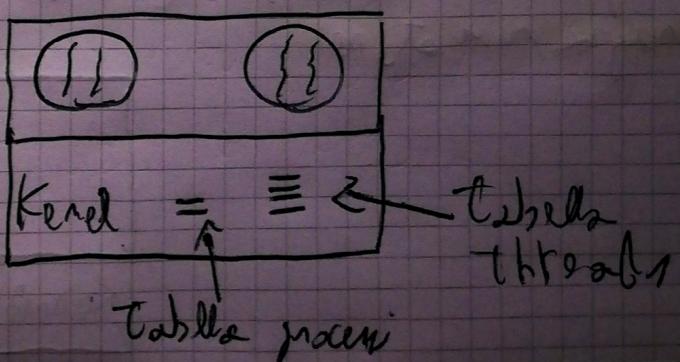
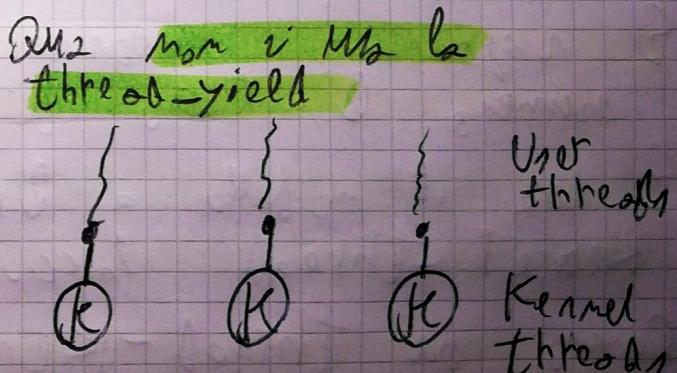
Page fault | se il programma fa una chiamata o
solto un'istruzione non in
memoria (grazie) il SO prende
l'istruzione e le viene dal
Disk.

Se un thread causa un page-fault il Kernel
blocca tutto il processo e procede a quello
traferimento I/O momento C per fare
altro thread disponibile

Threads & Diretta Kernel

- modelli 1 a 1
- richiede supporto
specifico del Kernel

Abbiamo una sola
tabella di thread
nel Kernel



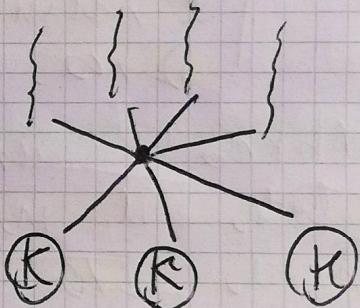
Pro: un thread se ha una chiamata
blockante non infastida gli altri.

- Contro:
- Creazione e distruzione threads più costose
 - Il numero di threads diversi limitato, è possibile non necessario ricorrere
 - Context-switch più lento, richiede la swap

In realtà lo switch se ci tra modelli è più lento, è più veloce se i thread sono meno modelli.
(ci intende riportare al modello thread a livello utente)

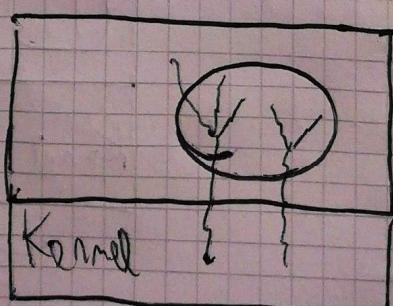
Thread modello ibrido

- molti o molti
- Prende i vantaggi degli altri due modelli



E risparmia risorse del sistema quando un numero limitato di thread Kernel.

E usa il modello a livello Kernel, già abilitato.
Kernel-thread faccio gestire da thread utente.
Ognuno dei thread-kernel è associato a 1 o più processi utenti, l'esecuzione è decisa dal programma utente.



Oggi

- Tutti i SO prevedono i thread a livello Kernel, ciò che fornisce alle gerarchie di programmazione di avere i thread stante
 - Ci sono anche le librerie di accesso ai Thread (e pressoché dal web molti)
- | Pthreads & Posix (con specifiche per il particolare sistema)
- | threads WIN32
- | thread in Java