# PCAD
# Programmazione Concorrente e Algoritmi Distribuiti

DIBRIS

Laurea Triennale Informatica, a.a. 2021/22

# Processi in UNIX

- Ogni processo UNIX (identificato dal PID - process identifier) ha uno spazio di indirizzamento separato e quindi non vede le zone di memoria dedicate agli altri processi.
- Un processo UNIX ha tre segmenti:
  - Stack
  - Dati
    - Dati statici
    - Heap
  - Codice
- L'indirizzamento è virtuale: codice, dati e heap sono memorizzati nella parte iniziale della memoria virtuale, lo stack nella parte finale

# Operazioni per creazione di processi

- L'unico modo di creare nuovi processi in Unix è attraverso la funzione di sistema *fork*
- La chiamata fork() dal processo $P$ (padre) crea un nuovo processo $F$ (figlio) che viene eseguito in parallelo con il padre
- Dopo la creazione del figlio, padre e figlio condividono lo stesso codice, inoltre il figlio ha una copia dei dati e del program counter del padre
- Il figlio proseguirà l'esecuzione a partire dall'istruzione che segue la fork (cioè lo stesso punto nel quale si trova il padre)

# Fork in Unix

- Per distinguere padre e figlio se la chiamata fork() ha successo allora restituisce:
  - il PID del figlio al processo padre
  - 0 al figlio

  Il valore di ritorno di fork viene usato quindi per ridefinire il comportamento del figlio
- Tipicamente infatti il figlio sostituirà l'immagine del processo padre con un nuovo programma attraverso la chiamata di sistema alla funzione execve che prende come parametro il nome di un file eseguibile
- La funzione execve() sostituisce dati, codice e stack con quelli del nuovo programma da eseguire nel contesto del processo figlio
- Il padre può aspettare la terminazione del figlio utilizzando la funzione di sistema waitpid

# Schema Fork

```
pid = fork();
if (pid < 0) {
   /* fork fallito */
} else if (pid > 0) {
   /* codice eseguito solo dal padre */
} else {
   /* codice eseguito solo dal figlio */
}
/* codice eseguito da entrambi */
```

# Esempio: ciclo fork/wait di una shell

```
while (1) {
 read_command(commands, parameters);
 if (fork() != 0) {     /* parent code */
   waitpid(-1, &status, 0);
 } else {               /* child code */
   execve(command, parameters, NULL);
 }
}
```

# Creazione di un processo

- La **fork** alloca spazio per il processo figlio
  - nuove tabelle per la memoria virtuale
  - nuova memoria segmenti dati e stack
  - dati, stack e u-structure vengono copiati da quelli del padre preservando file aperti, UID e GID, ecc.
  - il codice viene condiviso
- La **execve** non crea nessun nuovo processo: semplicemente, i segmenti dati, codice e stack vengono rimpiazzati con quelli del nuovo programma

# User and Kernel Mode

- I processi Unix operano in modo user e kernel: cioè il kernel esegue nel contesto di un processo le operazioni per gestire chiamate di sistema e interrupt
  - Alla partenza del sistema il codice del kernel viene caricato in memoria principale
  - Un processo in esecuzione in modo user non può accedere allo spazio di indirizzi del kernel
  - Quando un processo passa ad eseguire in modo kernel tale vincolo viene rilasciato per eseguire codice kernel (es. routine di gestione di interrupt/codice di una chiamata di sistema) nel contesto del processo utente
- Contesto utente: codice, dati, stack, registri,
- Contesto Kernel: entry nella tabella dei processi, u-area, stack kernel

# Livelli di contesto

- La parte dinamica del contesto di un processo (kernel stack, registri salvati) è organizzata a sua volta come stack con un numero di posizioni che dipende dai livelli di interrupt diversi ammessi nel sistema
- Ad esempio se il sistema gestisce interrupt software, interrupt di terminali, di dischi, di tutte le altre periferiche, e di clock: avremo al più sette livelli di contesto
  - Livello 0: User
  - Livello 1: Chiamate di sistema
  - Livelli 2-6: Interrupt (l'ordine dipende dalla priorità associata alle interrupt)

# Esempio di esecuzione nel contesto di un processo

- Il processo esegue una chiamata di sistema: il kernel salva il suo contesto (registri, program e stack pointer) nel livello 0 e crea il contesto di livello 1
- La CPU riceve e processa un interrupt di disco (il controllo viene fatto prima dell'esecuzione della prossima istruzione): il kernel salva il contesto di livello 1 (registri, stack kernel) e crea il livello 2 nel quale si esegue la routine di gestione dell'interrupt di disco
- La CPU riceve un interrupt di clock: il kernel salva il contesto di livello 2 (registri, stack kernel per la routine di gestione dell'interrupt disco) e crea il livello 3 nel quale si esegue la routine di gestione dell'interrupt di clock
- La routine termina l'esecuzione: il kernel recupera il livello di contesto 2 e così via
- Tutti questi passi vengono fatti sempre all'*interno dello stesso processo*: cambia solo la sua parte di contesto dinamica

# Algoritmo di gestione delle interruzioni

- L'algoritmo del kernel per la gestione di un'interrupt consiste dei seguenti passi:
  - salva il contesto del processo corrente
  - determina fonte dell'interrupt (trap/interrupt I/O/ ecc)
  - recupera l'indirizzo di partenza della routine digestione delle interrupt (dal vettore delle interrupt)
  - invoca la routine di gestione dell'interrupt
  - recupera il livello di contesto precedente
- Per motivi di efficienza parte della gestione di interruzioni e trap viene eseguita direttamente dalla CPU (dopo aver seguito un'istruzione): il kernel dipende quindi dal processore

# Trap/Interrupt vs Context switch

- Il modo user/kernel permette al kernel di lavorare nel contesto di un altro processo senza dover creare nuovi processi kernel
- Con questo meccanismo un processo in modo kernel può svolgere funzioni logicamente collegate ad altre processi (ad es. la gestione dei dati restituiti da un lettore di disco) e non necessariamente collegate al processo che *ospita* momentaneamente il kernel
- Nota: La gestione di trap/interrupt si basa su una sorta di context switch all'interno di un processo: il controllo non passa ad un'altro processo ma è necessario salvare la parte corrente del contesto dinamico del processo all'interno dello stesso processo

# Context Switch

- Il kernel vieta context switch arbitrari per mantenere la consistenza delle sue strutture dati
- Il controllo può passare da un processo all'altro in quattro possibili scenari:
  - Quando un processo si sospende
  - Quando termina
  - Quando torna a modo user da una chiamata di sistema ma non è più il processo a più alta priorità
  - Quando torna a modo user dopo che il kernel ha terminato la gestione di un'interrupt a non è più il processo a più alta priorità
- In tutti questi casi il kernel lascia la decisione di quale processo da eseguire allo scheduler

# MultiThreading

# Dai processi...

I processi dei Sistemi Operativi hanno le seguenti caratteristiche:

- Rappresentano unità di allocazione risorse: codice eseguibile, dati allocati staticamente (variabili globali) ed esplicitamente (heap), risorse mantenute dal kernel (file, I/O, workind dir), controlli di accesso ...
- Rappresentano unità di esecuzione: un percorso di esecuzione attraverso uno o più programmi: stack di attivazione (variabili locali), stato (running, ready, waiting,...), priorità, parametri di scheduling,...

Queste due componenti si possono considerare in maniera separata

# . . . ai thread

- Un *thread* (o *processo leggero*, *lightweight process*) è una unità di esecuzione:
  - program counter, insieme registri
  - stack del processore
  - stato di esecuzione
- Un thread condivide con i thread suoi pari una unità di allocazione risorse:
  - il codice eseguibile
  - i dati
  - le risorse richieste al sistema operativo
- un *task* = una unità di risorse + i thread che vi accedono

# Condivisione di risorse tra i thread

- Vantaggi: maggiore efficienza
  - Creare e cancellare thread è più veloce (100–1000 volte): meno informazione da duplicare/creare/cancellare (e a volte non serve la system call)
  - Lo scheduling tra thread dello stesso processo è molto più veloce che tra processi
  - Cooperazione di più thread nello stesso task porta maggiore throughput e performance
    (es: in un file server multithread, mentre un thread è bloccato in attesa di I/O, un secondo thread può essere in esecuzione e servire un altro client)

# Condivisione di risorse tra thread (Cont.)

- Svantaggi:
  - Maggiore complessità di progettazione e programmazione
    - i processi devono essere "pensati" paralleli
    - minore information hiding
    - sincronizzazione tra i thread
    - gestione dello scheduling tra i thread può essere demandato all'utente
  - Inadatto per situazioni in cui i dati devono essere protetti
- Ottimi per processi cooperanti che devono condividere strutture dati o comunicare (e.g., produttore–consumatore, server, ...): la comunicazione non coinvolge il kernel

# Esempi di applicazioni multithread

- Lavoro foreground/background: mentre un thread gestisce l'I/O con l'utente, altri thread operano sui dati in background. Spreadsheets (ricalcolo automatico), word processor (reimpaginazione, controllo ortografico,. . . )
- Elaborazione asincrona: operazioni asincrone possono essere implementate come thread. Es: salvataggio automatico.
- Task intrinsecamente paralleli: vengono implementati ed eseguiti più efficientemente con i thread. Es: file/http/dbms/ftp server, . . .

# User Level Thread

**User-level thread (ULT):** stack, program counter, e operazioni su thread sono implementati in librerie a livello utente. Vantaggi:

- efficiente: non c'è il costo della system call
- semplici da implementare su sistemi preesistenti
- portabile: possono soddisfare lo standard POSIX 1003.1c (pthread)
- lo scheduling può essere studiato specificatamente per l'applicazione

# User Level Thread (Cont.)

Svantaggi:

- non c'è scheduling automatico tra i thread
  - non c'è prelazione dei thread: se un thread non passa il controllo esplicitamente monopolizza la CPU (all'interno del processo)
  - system call bloccanti bloccano tutti i thread del processo: devono essere sostituite con delle routine di libreria, che blocchino solo il thread se i dati non sono pronti (*jacketing*).
- L'accesso al kernel è sequenziale
- Non sfrutta sistemi multiprocessore
- Poco utile per processi I/O bound, come file server

Esempi: thread CMU, Mac OS $\leq 9$, alcune implementazioni dei thread POSIX

# Kernel Level Thread

**Kernel-level thread (KLT):** il kernel gestisce direttamente i thread. Le operazioni sono ottenute attraverso system call. Vantaggi:

- lo scheduling del kernel è per thread, non per processo $\Rightarrow$ un thread che si blocca non blocca l'intero processo
- Utile per i processi I/O bound e sistemi multiprocessor

Svantaggi:

- meno efficiente: costo della system call per ogni operazione sui thread
- necessita l'aggiunta e la riscrittura di system call dei kernel preesistenti
- meno portabile
- la politica di scheduling è fissata dal kernel e non può essere modificata

Esempi: Unix

# Implementazioni ibride ULT/KLT

**Sistemi ibridi:** permettono sia thread livello utente che kernel.
Vantaggi:

- tutti quelli dei ULT e KLT
- alta flessibilità: il programmatore può scegliere di volta in volta il tipo di thread che meglio si adatta

Svantaggio: portabilità Es: Solaris 2 (thread/pthread e LWP), Linux (pthread e cloni), Mac OS X, Windows NT, ...