

Informazioni

- Docente: Prof. Bruno Carpentieri

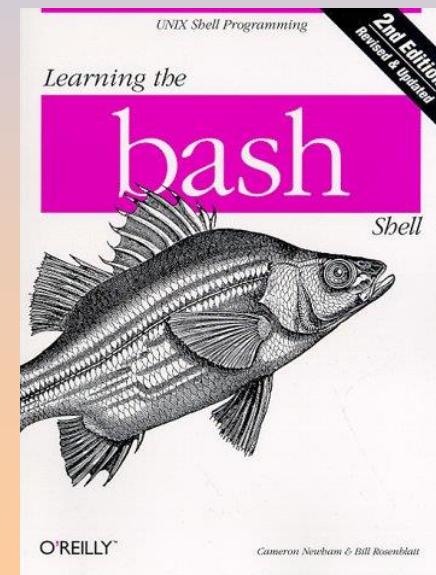
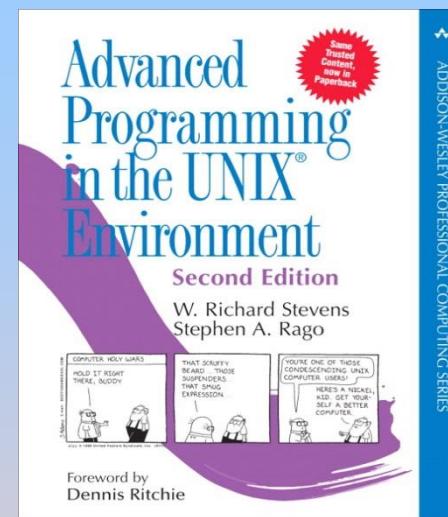
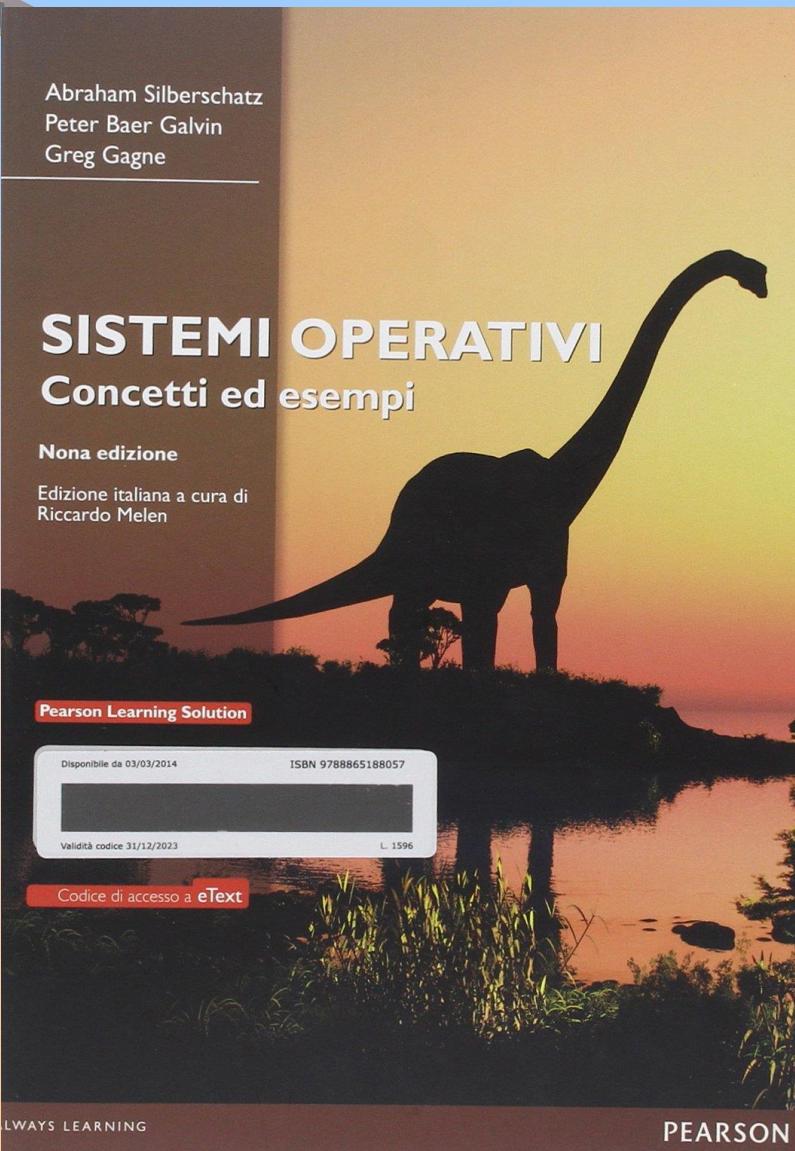
- e-mail: bc@dia.unisa.it

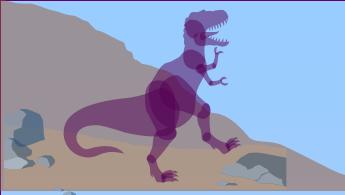
(nelle e-mail indicare, come subject, il nome del corso, cioè Sistemi Operativi e firmarsi sempre con nome, cognome e matricola)

- Queste slide:

<http://www.di.unisa.it/professori/bc/wrapupSO2018.pdf>

Principali libri di testo





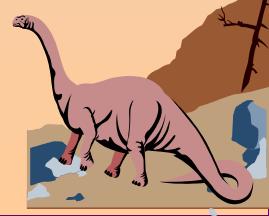
Principali libri di testo (II)

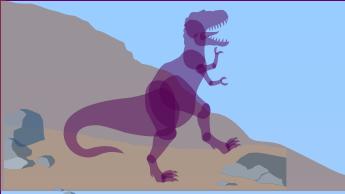
Per superare l'esame è necessario studiare **sui libri !!!**
(e non solo su appunti, registrazioni, "bignami", slide, etc.)

A. Silberschatz, G. Gagne, P. Galvin,
Sistemi Operativi,
Pearson, Nona Edizione, 2014.

B. W. R. Stevens, S. A. Rago,
Advanced Programming in the UNIX Environment,
Wiley, Seventh Edition, 2005.

C. C. Newham, B. Rosenblatt,
Learning the bash shell,
O'Reilly, Third Edition.





Capitolo 1: Introduzione

- Che cosa fa un Sistema Operativo
- Organizzazione di un sistema elaborativo
- Architettura degli elaboratori
- Struttura del sistema operativo
- Attività del sistema operativo
- Gestione dei processi
- Gestione della memoria
- Gestione della memoria di massa
- Protezione e sicurezza
- Strutture dati del kernel
- Ambienti di elaborazione
- Sistemi operativi open-source



Cos'è un Sistema Operativo?

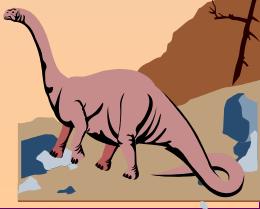
- Un programma che agisce come tramite tra l'utente e gli elementi fisici del calcolatore.
- E' un insieme di programmi (software) che:
 - gestisce gli elementi fisici di un calcolatore (hardware),
 - fornisce una piattaforma ai programmi di applicazione
 - agisce da intermediario tra l'utente e la struttura fisica del calcolatore





Cos'è un Sistema Operativo? (II)

- Scopi di un sistema operativo:
 - Eseguire programmi utente e rendere più semplice la soluzione dei problemi dell'utente.
 - Rendere conveniente ed efficiente l'utilizzo del sistema di calcolo.
- Un sistema operativo deve assicurare il corretto funzionamento di un calcolatore.
- Funzioni del Sistema Operativo
 - Estendere e astrarre l'hardware (per semplificare la programmazione, per rendere i programmi portabili, etc..).
 - (ad es. un “file” è un astrazione)
 - Gestire le risorse
 - (ad es. suddividere stampanti, dischi, tempo di CPU fra più programmi)





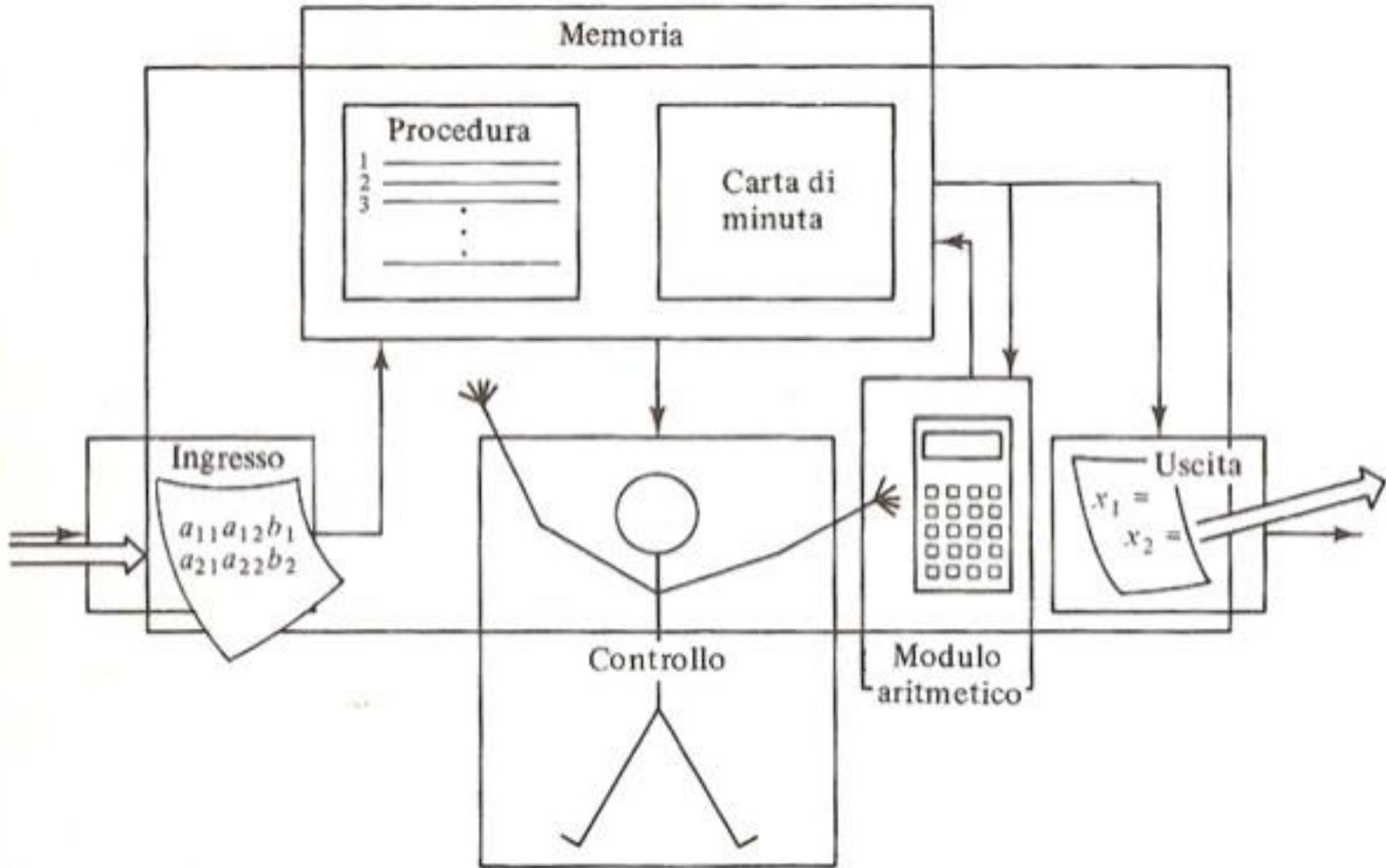
Architetture a singolo processore

- Questi sistemi sono dotati di un singolo processore che esegue un set di istruzioni general-purpose.
- Spesso usano anche processori special purpose,
 - ad es. i processori di I/O, che muovono velocemente dati tra le componenti (disk-controller, processori associati alle tastiere, etc.).
- A volte la CPU principale comunica con questi processori.
- Altre volte, essi sono totalmente autonomi.





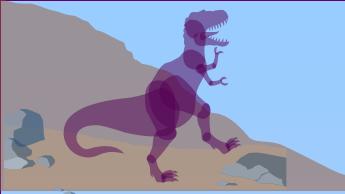
Architettura di Von Neumann





Funzionamento di un sistema di calcolo

- I dispositivi di I/O e la CPU possono operare in modo concorrente.
- Ciascun controllore di dispositivo è responsabile di un dispositivo di uno specifico tipo.
- Ciascun controllore di dispositivo ha un buffer locale.
- La CPU sposta dati dalla/alla memoria principale verso/da i buffer locali.
- L' I/O è tra il dispositivo ed il buffer locale del suo controllore.
- I controllori di dispositivo informano la CPU della fine di una operazione di I/O tramite un segnale di interruzione (*interrupt*).



Segnali di interruzione

- Un segnale di interruzione causa il trasferimento del controllo all'appropriata procedura di servizio dell'evento ad esso associato.
- La gestione di un'interruzione deve essere molto rapida.
- Dato che il numero di possibili interrupt è predefinito, per gestire le interruzioni si utilizza una tabella di puntatori (*vettore delle interruzioni*):
 - che contiene gli indirizzi delle procedure di servizio associate all'interruzione.
- E' inoltre necessario:
 - salvare l'indirizzo dell'istruzione che viene interrotta
 - Disabilitare la possibilità di ulteriori interruzioni fino a quando la gestione dell'interrupt corrente non è terminata
- Un *segnale di eccezione (trap)* è un interrupt software causato da un errore o da una richiesta specifica dell'utente
 - *chiamata del sistema o del supervisore*



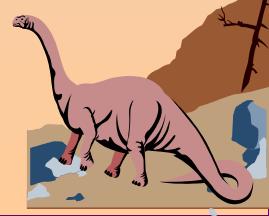
Attività del S.O.: gestione delle interruzioni

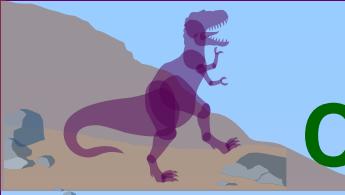
- I sistemi operativi moderni sono caratterizzati dal fatto di essere *guidati dalle interruzioni (interrupt driven)*.
 - se non ci sono processi da eseguire, dispositivi di I/O da servire o utenti con cui interagire, il S.O. resta inattivo nell'attesa che accada qualcosa.
- In presenza di una interruzione:
 - il sistema operativo preserva lo stato della CPU salvando lo stato dei registri e del contatore di programma prima di servire l'interruzione.
 - Determina di che tipo sia l'interruzione.
 - Segmenti diversi di codice determinano quale azione debba essere presa per ciascun tipo di interrupt.
 - Dopo aver servito l'interruzione il S.O. ripristina lo stato della CPU (ad es. i registri) e del contatore di programma originali.



Capitolo 2: Strutture dei sistemi operativi

- Servizi di un sistema operativo
- Interfaccia con l'utente del sistema operativo
- Chiamate di sistema
- Categorie di chiamate del sistema
- Programmi di sistema
- Progettazione e realizzazione di un sistema operativo
- Struttura del sistema operativo
- Debugging dei sistemi operativi
- Generazione di sistemi operativi
- Avvio del sistema

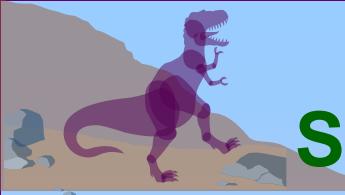




Chiamate del sistema (*system calls*)

- Le system calls costituiscono l'interfaccia tra un processo ed il sistema operativo.
- Sono generalmente disponibili in forma di istruzioni in linguaggio assembly.
- In alcuni sistemi le chiamate possono essere invocate direttamente tramite funzioni scritte in programmi ad alto livello (C, C++).
- Tipicamente ad ogni chiamata di sistema è associato un numero.
 - Il sistema mantiene una tabella, indicizzata da questi numeri.





System Calls e funzioni di libreria in Unix

- Tutti i sistemi operativi provvedono service points attraverso i quali i programmi richiedono servizi dal kernel.
- Unix prevede un ben definito, limitato numero di entry points direttamente nel kernel, chiamati **system calls**.
- Questi entry points in Unix sono direttamente accessibili al programmatore C.
 - A differenza di sistemi operativi più antiquati in cui erano accessibili solo attraverso routines in linguaggio macchina.
- La tecnica usata è questa:
 - per ogni system call esiste una funzione con lo stesso nome nella libreria standard di C.
 - La chiamata a questa funzione C invocherà l'appropriato servizio kernel usando qualsiasi tecnica sia appropriata in quello specifico sistema.





System Calls e funzioni di libreria in Unix (II)

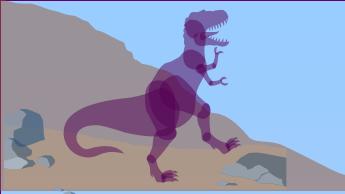
- Dal punto di vista del programmatore sia system calls che funzioni di libreria appaiono come normali funzioni C.
 - Mentre però sarà sempre possibile riscrivere le funzioni di libreria, ovviamente non sarà possibile rimpiazzare o riscrivere le system-calls.
- In `<sys/types.h>` sono presenti alcune definizioni di tipi di dati dipendenti dalla macchina usati dal sistema.
- Questi tipi di dati sono detti ***primitive system data types***
 - la maggior parte di questi tipi di dati finisce in `_t`
 - Ad es. `off_t`, `dev_t`, etc.





Introduzione alla Bash (Bourne Again Shell)

(Rosenblatt)



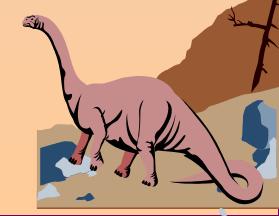
Shell in Unix

- La shell e' un interprete di comandi.
 - I comandi digitati dall'utente vengono letti dalla shell, interpretati e inviati al kernel per essere eseguiti.
 - I comandi vengono digitati dall'utente sulla riga comandi della shell.
 - La riga comandi e' una riga digitabile che comincia dal prompt.
 - Il prompt e' un carattere o un insieme di caratteri che possono essere personalizzati dall'utente.
 - Il prompt dell'utente root (il superuser) inizia con il carattere '#' (cancelletto), mentre il prompt di un utente qualsiasi inizia con il carattere '\$'.
 - La shell viene eseguita quando facciamo login e termina al logout
- 



Lab: shell

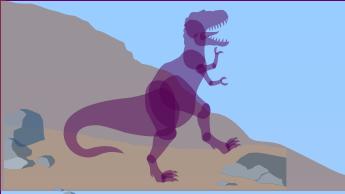
- vi e gcc
- BASH
- Prompt di comandi e compiti della shell.
- L'ambiente shell.
- Comandi, argomenti e opzioni
- File e directory
- Filename e wildcard
- I/O e ridirezione
- Pipeline
- Processi in background
- I/O in processi in background
- Caratteri speciali e backslash escaping
- help, echo, read
- history
- Personalizzare l'ambiente
- Script



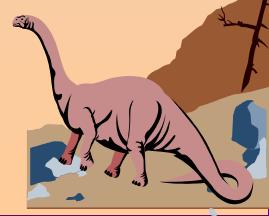


Capitolo 3: Processi

- Concetto di processo.
- Scheduling dei processi.
- Operazioni sui processi.
- Comunicazione tra processi.
- Esempi di sistemi per la IPC.
- Comunicazione nei sistemi client/server.



Concetto di processo

- Un S.O. esegue una varietà di “programmi”:
 - i sistemi a lotti (*batch*) eseguivano *lavori* (*job*)
 - un sistema a partizione di tempo esegue *programmi utente* (*task*)
 - Spesso i termini *lavoro* e *processo* sono utilizzati in modo intercambiabile.
 - Informalmente un processo può essere considerato come un programma in esecuzione.
 - Un programma di per sé non è un processo.
 - Il programma, anche detto *sezione testo*, è un’entità *passiva*, mentre il processo è un’entità *attiva*.
 - Un processo include:
 - Contatore di programma (program counter)
 - Pila (stack)
 - Sezione di dati (data section)
- 

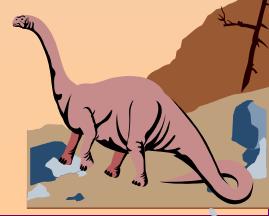
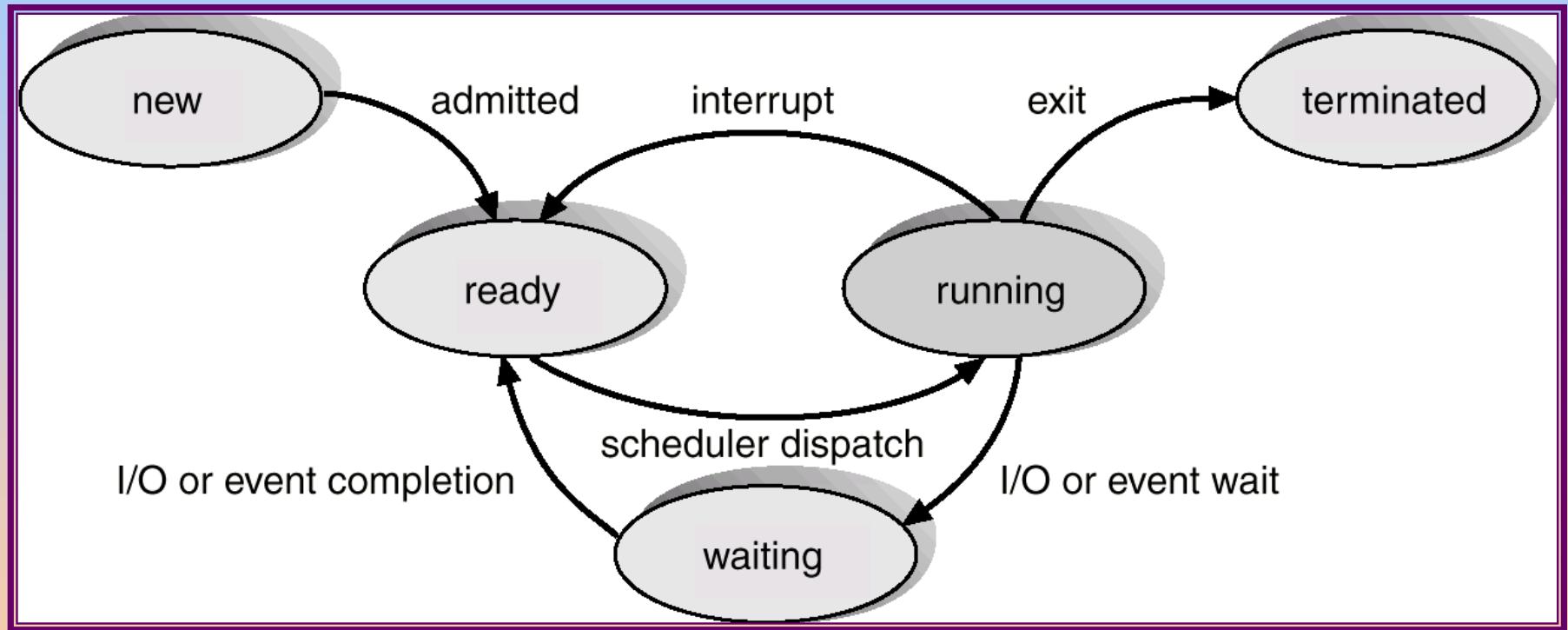


Stato del processo

- L'esecuzione di un processo deve progredire in maniera sequenziale.
- Mentre un processo è in esecuzione è soggetto a cambiamenti di *stato*.
- Ogni processo può trovarsi in uno tra i seguenti stati:
 - **Nuovo (new)**: Il processo viene creato.
 - **In esecuzione (running)**: quando è in memoria ed ha il controllo della CPU;
 - **In attesa (waiting)**: quando è temporaneamente sospeso in attesa di un evento, quale la terminazione di I/O, lo scadere di un timer, la ricezione di un messaggio etc.;
 - **Pronto (ready)**: quando è in memoria e pronto per l' esecuzione, ma non ha il controllo della CPU e attende di essere assegnato ad un'unità di elaborazione.
 - **Terminato (terminated)**: quando termina l'esecuzione e abbandona il sistema.



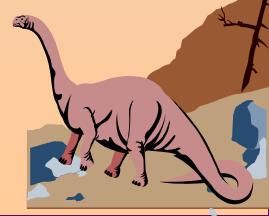
Diagramma di transizione degli stati di un processo

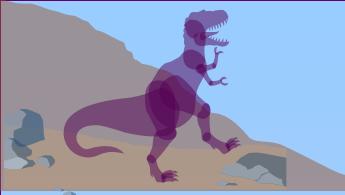




Process Descriptor (PID)

- In un sistema operativo ogni processo è rappresentato da un descrittore di processo:
 - *process descriptor – PD o PID* - detto anche blocco di controllo di un processo (*process control block - PCB*).
- Nel PCB sono contenute informazioni connesse ad uno specifico processo:
 - **Stato del processo:** tra new, ready, running, waiting, terminated.
 - **Contatore di programma:** indica l'indirizzo della successiva istruzione da eseguire.
 - **Registri di CPU:** accumulatori, registri indice, puntatori alla cima delle strutture a pila (stack pointer), registri d'uso generale e registri contenenti informazioni relative ai codici di condizione.
 - **Informazioni sullo scheduling di CPU:** priorità del processo, puntatori alle code di scheduling e altri parametri di schedulazione.





Blocco di controllo dei processi (II)

- **Informazioni sulla gestione della memoria:** registri di base e di limite, tabelle delle pagine in memoria o dei segmenti (a seconda della tecnica usata dal S.O.).
- **Informazioni di contabilizzazione delle risorse:** tempo di CPU, tempo reale di CPU, numero del processo, etc..
- **Informazioni di I/O:** lista dei dispositivi di I/O assegnati al processo, elenco file aperti, etc..

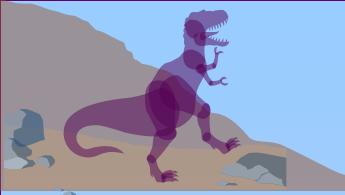




Operazioni: creazione di un processo

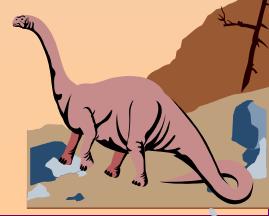
- Durante la sua esecuzione un processo può creare altri processi.
- Il processo creante è chiamato *padre*, mentre il processo creato è chiamato *figlio*.
- La gerarchia che esiste in questa struttura è ricorsiva, per cui un processo figlio può essere, a sua volta, padre di altri processi.
- Così facendo si viene a creare un *albero dei processi*.
- Risorse
 - Padre e figlio condividono tutte le risorse.
 - Il figlio condivide un sottinsieme delle risorse del padre.
 - Padre e figlio non condividono risorse.
- Esecuzione
 - Il padre continua l'esecuzione, concorrentemente al processo figlio.
 - Il padre attende che i processi figli terminano la loro esecuzione.

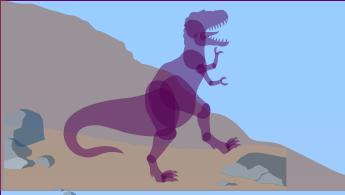




Creazione di un processo (II)

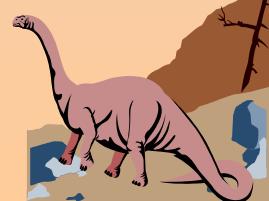
- Spazio di indirizzi
 - Il processo figlio è un duplicato del padre
 - Nel processo figlio si carica un programma
- UNIX
 - fork
 - exec dopo fork carica un programma nel processo figlio.





Creazione di un processo in Unix

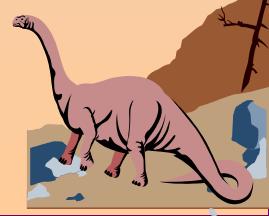
```
# include <stdio.h>
void main (int argc, char *argv[ ])
{ int pid;
    /* genera un nuovo processo */
    pid = fork ();
    if (pid < 0)    { /* errore */
                      fprintf(stderr, "errore");
                      exit(-1);
    }
    else    if (pid == 0)    { /* processo figlio */
                      execvp ("/bin/ls", "ls", NULL);
    }
    else    { /* processo genitore */
                      wait (NULL);
                      printf("Il figlio ha terminato");
                      exit(0);
    }
}
```

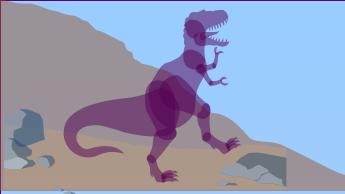




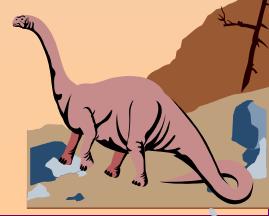
(Stevens)

Capitolo 8: Process Control





Lab: processi

- Studio di alcune system call di Linux relative ai processi:
 - ad. es. fork, exec, wait, waitpid, getpid, getppid, etc.
 - Utilizzo di queste system call in semplici programmi C.
 - Esercitazioni su creazione di processi, processi concorrenti, etc..
- 



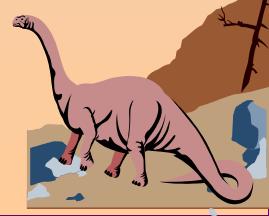
Capitolo 4: Thread

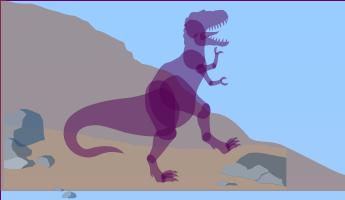
- Introduzione
- Programmazione multicore
- Modelli di supporto al multithreading
- Librerie dei thread
- Threading implicito
- Problematiche di programmazione multithread
- Esempi di Sistemi Operativi



Thread

- In alcune situazioni una singola applicazione deve poter gestire molti compiti simili tra loro.
- In altre una singola applicazione può dover gestire più compiti diversi, a volte eseguibili concorrentemente.
- Una possibile soluzione è quella della creazione di più processi tradizionali.
- Nel modello a processi, l'attivazione di un processo o il cambio di contesto sono operazioni molto complesse che richiedono ingenti quantità di tempo per essere portate a termine.
- Tuttavia a volte l'attività richiesta ha vita relativamente breve rispetto a questi tempi.
 - Ad es. l'invio di una pagina html da parte di un server web: applicazione troppo leggera per motivare un nuovo processo.
- Possibile soluzione: threads.

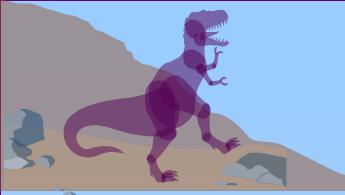




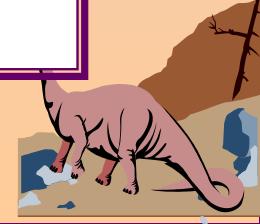
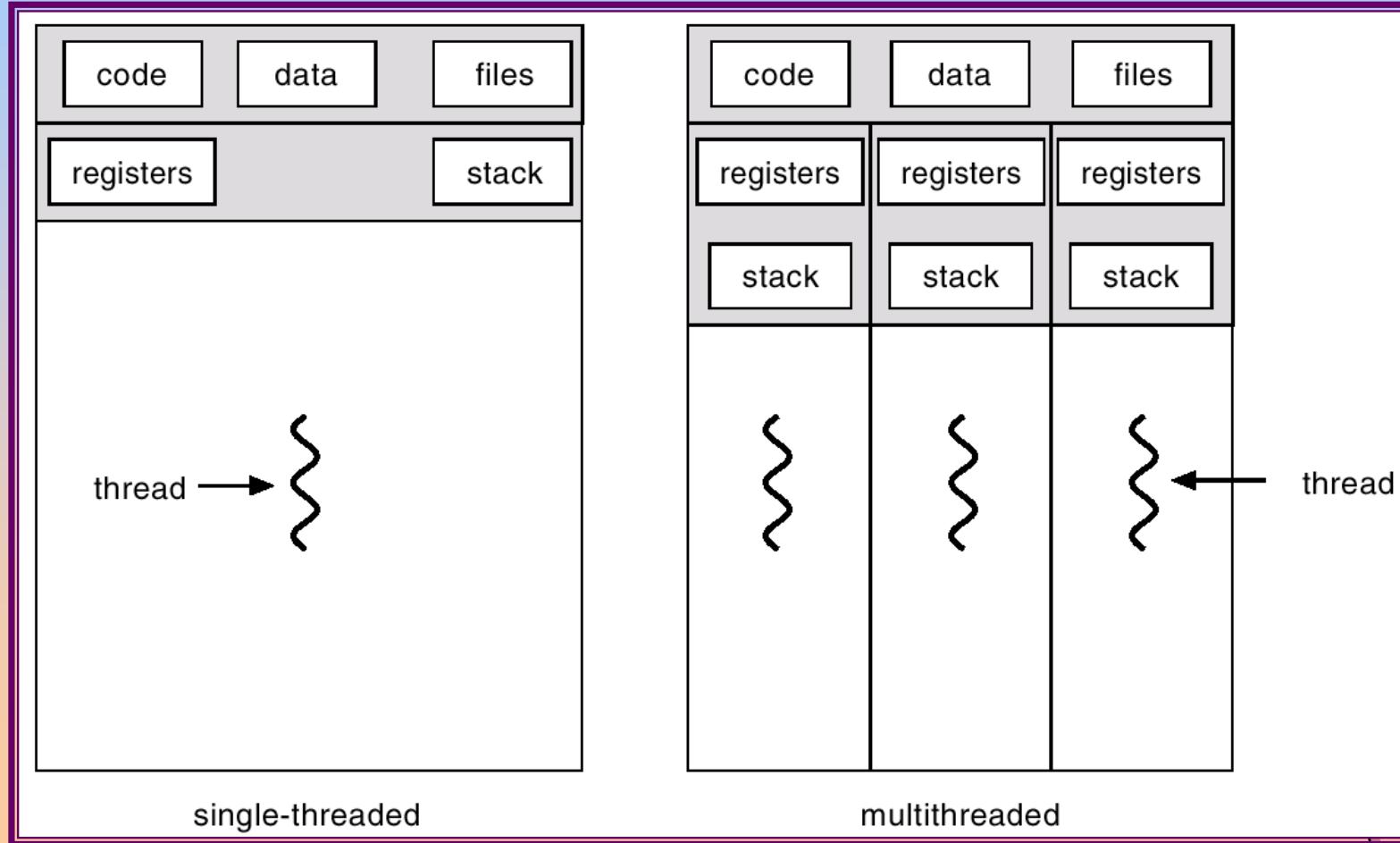
Thread (II)

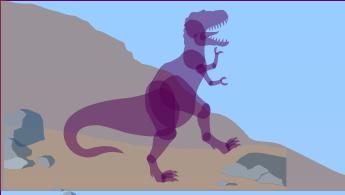
- Un thread è talvolta chiamato processo leggero (*lightweight process*).
- Condivide con gli altri thread che appartengono allo stesso processo la sezione del codice, la sezione dei dati e altre risorse di sistema, come i file aperti e i segnali.
- Processi tradizionali = singolo thread.
- Processi multithread = più thread.
- Molti kernel sono ormai multithread,
 - con i singoli thread dedicati a servizi specifici come la gestione dei dispositivi periferici o delle interruzioni.
- Molti programmi per i moderni PC sono predisposti per essere eseguiti da processi multithread.
- Un'applicazione, solitamente, è codificata come un processo a sè stante comprendente più thread di controllo.
- Il multithreading è la capacità di un sistema operativo di supportare thread di esecuzione multipli per ogni processo.





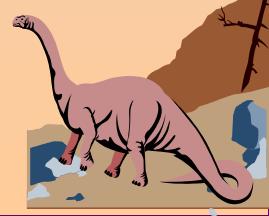
Processi a singolo thread e multithread





Capitolo 5: Sincronizzazione dei processi

- Introduzione
- Problema della sezione critica
- Soluzione di Peterson
- Hardware per la sincronizzazione
- Lock mutex
- Semafori
- Problemi tipici di sincronizzazione
- Monitor
- Esempi di sincronizzazione
- Approcci alternativi

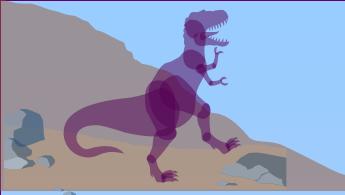




Problema della sezione critica

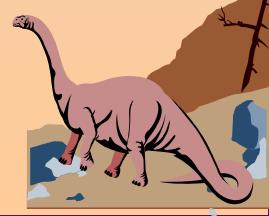
- Per evitare il problema delle race condition si introduce l'idea di **sezione critica**:
 - una **sezione critica** è un segmento di codice nel quale il processo può modificare variabili comuni, scrivere un file, aggiornare tabelle, etc.
- Quando un processo è in esecuzione nella propria sezione critica non si deve consentire a nessun altro processo di entrare in esecuzione nella propria sezione critica.
- L'esecuzione delle sezioni critiche da parte dei processi è *mutuamente esclusiva nel tempo*.

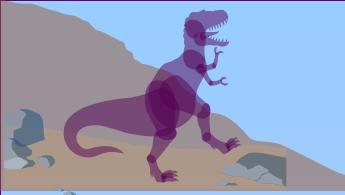




Soluzione al problema della sezione critica

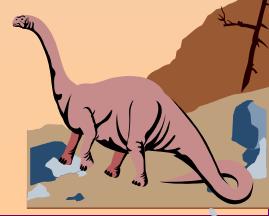
- Per risolvere il problema della sezione critica, occorre soddisfare i tre seguenti requisiti:
 1. **Mutua esclusione.** Se il processo P_i è nella sua sezione critica, allora nessun altro processo può essere nella sua sezione critica.
 2. **Progresso.** Se nessun processo è in esecuzione nella sua sezione critica ed esiste qualche processo che desidera entrare nella propria sezione critica, allora la decisione riguardante la scelta del processo che potrà entrare per primo nella propria sezione critica non può essere rimandata indefinitamente.
 3. **Attesa Limitata.** Se un processo ha già richiesto l'ingresso nella sua sezione critica, esiste un limite al numero di volte che si consente ad altri processi di entrare nelle rispettive sezioni critiche prima che si accordi la richiesta del primo processo.
 - Assumiamo che ogni processo sia eseguito a una velocità diversa da 0
 - Nessuna ipotesi sulla *velocità relativa* degli n processi

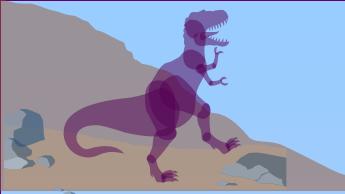




Capitolo 6: Scheduling della CPU

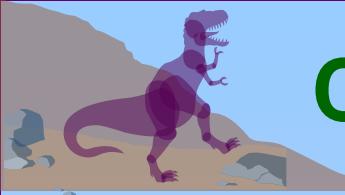
- Concetti fondamentali.
- Criteri di scheduling.
- Algoritmi di scheduling.
- Scheduling dei thread.
- Scheduling per sistemi multiprocessore
- Scheduling real-time della CPU
- Esempi di sistemi operativi. (solo Linux)
- Valutazione degli algoritmi.





Capitolo 7: Stallo dei processi

- Modello del sistema
- Caratterizzazione delle situazioni di stallo
- Metodi per la gestione delle situazioni di stallo
- Prevenire le situazioni di stallo
- Evitare le situazioni di stallo
- Rilevamento delle situazioni di stallo
- Ripristino da situazioni di stallo

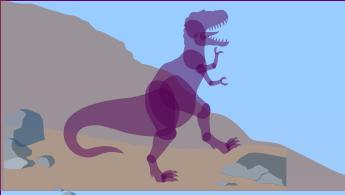


Caratterizzazione dei deadlock

Una situazione di deadlock può verificarsi solo se valgono **tutte e quattro** le seguenti condizioni simultaneamente :

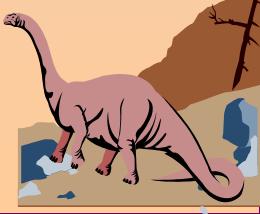
- **Mutua esclusione:** almeno una risorsa deve essere “non condivisibile”, cioè può essere usata da un solo processo alla volta. Se un altro processo richiede questa risorsa, si deve bloccare il processo richiedente fino al rilascio della risorsa.
- **Possesso ed attesa:** un processo, in possesso di almeno una risorsa, attende di acquisire ulteriori risorse già in possesso di altri processi.
- **Impossibilità di prelazione:** non esiste un diritto di prelazione. Una risorsa può essere rilasciata dal processo che la possiede solo volontariamente, al termine del suo utilizzo.
- **Attesa circolare:** deve esistere un insieme $\{P_0, P_1, \dots, P_n\}$ di processi in attesa, tali che P_0 è in attesa di una risorsa che è posseduta da P_1 , P_1 è in attesa di una risorsa posseduta da P_2 , ..., P_{n-1} è in attesa di una risorsa posseduta da P_n , e P_n è in attesa di una risorsa posseduta da P_0 .

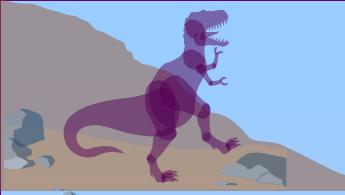




Metodi per la gestione dei deadlock

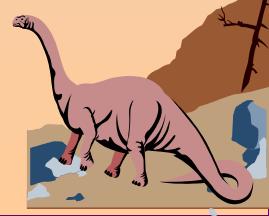
- Assicurare che il sistema non entri *mai* in uno stato di deadlock.
 - **Prevenire i deadlock:**
 - evitare che contemporaneamente si verifichino mutua esclusione, possesso e attesa, impossibilità di prelazione e attesa circolare ⇒ basso utilizzo risorse e throughput ridotto.
 - **Evitare i deadlock:**
 - evitare gli stati del sistema a partire dai quali si può evolvere verso un deadlock.
- Permettere al sistema di entrare in uno stato di deadlock, quindi ripristinare il sistema.
 - **Determinare la presenza di un deadlock.**
 - **Ripristinare il sistema da un deadlock.**
- Ignorare il problema e “fingere” che i deadlock non avvengano mai nel sistema; impiegato dalla maggior parte dei S.O., incluso UNIX.

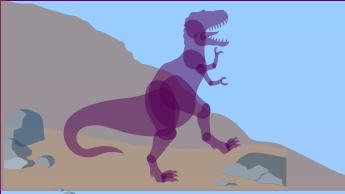




Capitolo 8: Memoria centrale

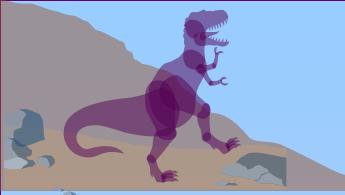
- Introduzione
- Avvicendamento dei processi (swapping)
- Allocazione contigua della memoria
- Segmentazione
- Paginazione
- Struttura della tabella delle pagine
- Esempio: le architetture Intel a 32 e 64 bit
- Esempio: architettura ARM





Paginazione

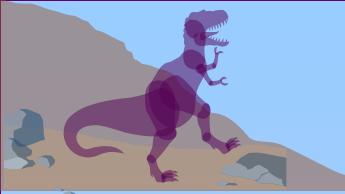
- Con il metodo della paginazione la memoria viene suddivisa in blocchi di dimensione fissa.
 - Lo spazio indirizzi logico di un processo può essere allocato in blocchi anche non contigui di memoria fisica.
 - I blocchi della memoria fisica sono detti **frame**, mentre i blocchi della memoria logica sono detti **pagine**.
 - Quando un processo è pronto per essere eseguito si caricano le sue pagine nei blocchi di memoria disponibile, prendendole dalla memoria ausiliaria, che è divisa in blocchi di dimensione fissa uguale a quella dei frame.
 - Si risolve il problema della frammentazione esterna (si potrà avere invece frammentazione interna).
 - Bisogna mantenere traccia dei frame liberi: per eseguire un processo di n pagine ci sarà bisogno di n frame liberi in cui caricare il processo.
 - Una *tabella delle pagine* verrà utilizzata per tradurre l'indirizzo logico in un indirizzo fisico.
- 



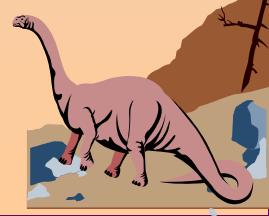
Capitolo 9: Memoria Virtuale

- Introduzione
- Paginazione su richiesta
- Copiatura su scrittura
- Sostituzione delle pagine
- Allocazione dei frame
- Attività di paginazione degenere (trashing)
- File mappati in memoria
- Allocazione di memoria del kernel
- Altre considerazioni
- Esempi di sistemi operativi





Paginazione su richiesta

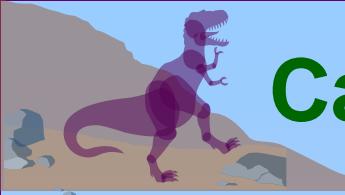
- La maggior parte dei sistemi operativi utilizza il metodo della paginazione su richiesta.
 - Anziché caricare in memoria l'intero processo, come nella paginazione, si carica una pagina in memoria solo quando essa è necessaria.
 - Per caricare un programma utente in memoria è necessario un minore tempo di I/O (aumentando la velocità di esecuzione).
 - Nessun vincolo sulla quantità di memoria fisica necessaria ad un processo.
 - Aumento del grado di multiprogrammazione grazie al minore utilizzo della memoria fisica.
 - Una pagina è necessaria \Rightarrow c'è stato un riferimento ad essa.
 - Se il processo tenta di accedere ad una pagina che non era stata caricata nella memoria, il sistema genera una *eccezione di pagina mancante* (page fault trap).
 - Il riferimento non era valido \Rightarrow si termina il processo (abort)
 - Il riferimento era valido \Rightarrow si porta la pagina in memoria
- 



Capitolo 10:

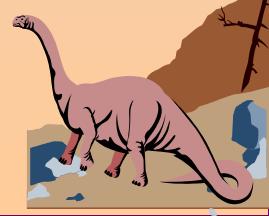
Memoria secondaria

- Struttura dei dispositivi di memorizzazione
- Struttura dei dischi
- Connessione dei dischi
- Scheduling del disco
- Gestione dell'unità a disco
- Gestione dell'area di avvicendamento
- Strutture RAID
- Realizzazione della memoria stabile



Capitolo 11: Interfaccia del File-System

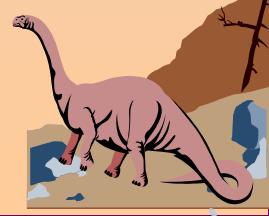
- Concetto di file
- Metodi di accesso
- Struttura della directory e del disco
- Montaggio di un file system
- Condivisione di file
- Protezione

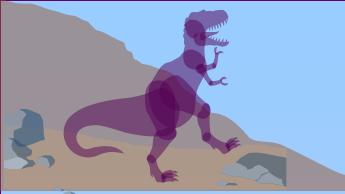




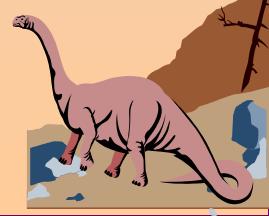
(Stevens)

Capitolo 3: File I/O





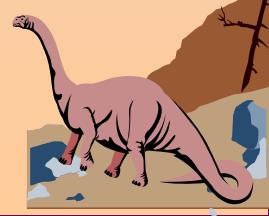
Lab: I/O di base

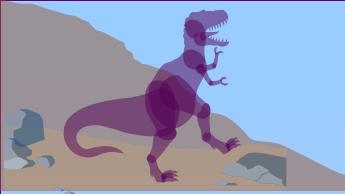
- Studio di alcune system call di Linux relative all'I/O:
 - ad. es. open, creat, close, lseek, read, write, etc.
 - Utilizzo di queste system call in semplici programmi C.
- 



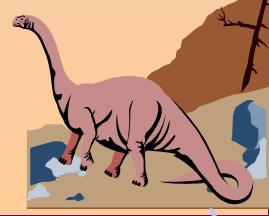
(Stevens)

Capitolo 4: Files and Directories





Files e Directories

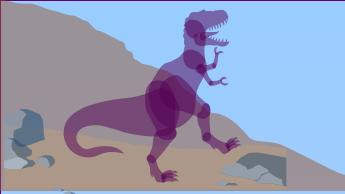
- stat, fstat e lstat
 - opendir e closedir
 - mkdir
 - rmdir
 - chdir
 - getcwd
- 



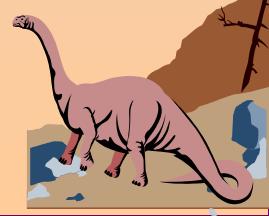
Capitolo 12:

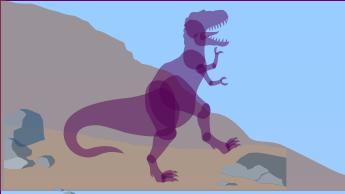
Realizzazione del File System

- Struttura del file system
- Realizzazione del file system
- Realizzazione delle directory
- Metodi di allocazione
- Gestione dello spazio libero
- Efficienza e prestazioni
- Ripristino
- File system con annotazione delle modifiche
- NFS
- Esempio: il file system WAFL

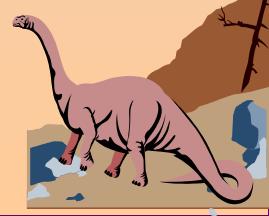


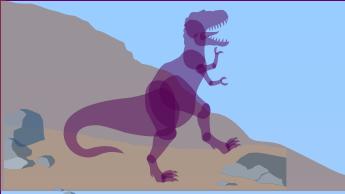
File system

- Tutte le applicazioni informatiche hanno bisogno di memorizzare e recuperare informazioni.
 - Abbiamo tre requisiti essenziali per la memorizzazione delle informazioni a lungo termine:
 - Si deve potere memorizzare un'enorme quantità di informazioni.
 - Le informazioni devono sopravvivere alla terminazione del processo che le usa.
 - Più processi devono poter accedere alle informazioni in modo concorrente.
 - Il *file system* è l'insieme delle strutture dati e dei metodi che ci permettono la registrazione e l'accesso a dati e programmi presenti in un sistema di calcolo.
 - Il file system risiede permanentemente nella memoria secondaria, progettata per mantenere in maniera non volatile grandi quantità di dati.
- 



Capitolo 13: Sistemi di I/O

- Introduzione
 - Hardware di I/O
 - Interfaccia di I/O per le applicazioni
 - Sottosistema per l'I/O del kernel
 - Trasformazione delle richieste di I/O in operazioni hardware
 - STREAMS
 - Prestazioni
- 



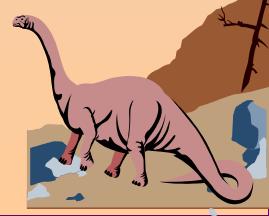
I/O

- I due compiti di un computer sono l'I/O e l'elaborazione.
- Il ruolo di un sistema operativo nell'I/O di un computer è quello di gestire e controllare le operazioni e i dispositivi di I/O.
- I dispositivi di I/O sono diversi per funzioni e velocità, quindi abbiamo diversi metodi di controllo.
- Questi metodi rappresentano il **sottosistema di I/O** del kernel.
- Questo sottosistema separa il resto del nucleo dalla complessità di gestione dei dispositivi di I/O.
- Esiste una grande varietà di dispositivi di I/O.
- I **driver dei dispositivi** offrono al sottosistema di I/O un'interfaccia uniforme per l'accesso ai dispositivi,
 - così come le system call forniscono un'interfaccia uniforme tra le applicazioni ed il sistema operativo.



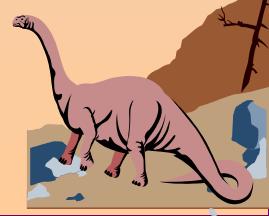
(Stevens)

Capitolo 10: Signals





Segnali

- Un segnale è un interrupt software che permette di gestire eventi asincroni,
 - come ad esempio un CTRL-C inviato da un utente seduto ad un terminale o la fine prematura di un processo per un errore di sistema.
 - Un segnale può essere generato in qualsiasi istante,
 - a volte da un processo utente, più spesso dal kernel a seguito di un errore software o hardware o comunque di un evento eccezionale.
 - Ogni segnale ha un nome che comincia con SIG (ad es. SIGABRT, SIGALARM) a cui viene associato una costante intera ($\neq 0$) positiva definita in signal.h
 - I segnali non contengono informazioni aggiuntive,
 - in particolare chi li riceve non può scoprire l'identità di chi li manda.
- 



Segnali (II)

- Il segnale e' un evento asincrono;
 - esso puo' arrivare in un momento qualunque ad un processo ma non necessariamente quando il segnale viene ricevuto verrà fatta qualcosa.
 - Vi sono azioni di default che vengono compiute
 - oppure il processo può scegliere di ignorare il segnale o gestirlo in maniera diversa dal default.
- Quindi le azioni associate ad un segnale sono le seguenti:
 - Ignorare il segnale
 - tranne che per SIGKILL e SIGSTOP che non possono essere ignorati perché sono il modo che il superuser ha di terminare o stoppare momentaneamente qualsiasi processo
 - Catturare il segnale
 - equivale ad associare all'occorrenza del segnale l'esecuzione di una funzione utente;
 - ad es. se il segnale SIGTERM e' catturato possiamo voler ripulire tutti i file temporanei generati dal processo
 - Eseguire l'azione di default associata
 - terminazione del processo per la maggior parte dei segnali



Lab: segnali

- Segnali Unix.
 - Studio di alcune system call di Linux relative ai segnali:
 - ad. es. signal, kill, raise, sleep, abort, pause, etc.
 - Utilizzo di queste system call in semplici programmi C.
- 