
Parte I

Introduzione ai Sistemi Operativi

Sistemi Operativi - prof. Silvio Salza - a.a. 2008-2009

I - 1

Docente

Prof. Silvio Salza

Dip. Di Informatica e Sistemistica

Via Ariosto 25, Il piano stanza B211

salza@dis.uniroma1.it (*usare con criterio*)

<http://www.dis.uniroma1.it/~salza/>

Ricevimento

Martedì 11.30 - 13.30

(verificare su web)

Sistemi Operativi - prof. Silvio Salza - a.a. 2008-2009

I - 2

Orario

Lunedì	15.45 -17.15	aula	21
Martedì	15.45 -17.15	aula	21

Materiale didattico

- Testo di riferimento

A.S. Tanenbaum, *Modern Operating Systems*, second edition, Prentice Hall International, 2001

traduzione italiana

A.S. Tanenbaum, *I Moderni Sistemi Operativi*, seconda edizione, Jackson Università, 2002

- Copia delle trasparenze del docente, e altro materiale: sul sito web

<http://www.dis.uniroma1.it/~salza/>

Obiettivi del corso

- *Studiare la struttura dei sistemi operativi, a livello di principi di funzionamento, strategie operative e meccanismi interni*
- *Comprendere l'interazione fra software applicativo e sistema operativo, e quindi le modalità di sviluppo di applicazioni complesse*
- *Comprendere l'impatto che il sistema operativo e la sua configurazione hanno sul funzionamento e le prestazioni delle applicazioni*
- *Poter dedurre dall'analisi delle prestazioni il tipo e la misura delle azioni correttive da adottare nella configurazione dell'hardware e del software*

Organizzazione del corso

1. Introduzione
 - Obiettivi e finalità del SO
 - Evoluzione dei SO
 - Concetti fondamentali
 - Struttura del SO
2. Processi e thread
 - Struttura ed implementazione
 - Stati dei processi
 - Comunicazione tra processi
 - Scheduling
3. Deadlock
 - Definizioni e problematiche
 - Metodi per il controllo del deadlock

Organizzazione del corso

4. Gestione della Memoria

- Partizioni
- Memoria virtuale
- Problematiche di implementazione

5. File Systems

- Organizzazione dei file
- Accesso con indici
- Implementazione del File System
- Esempi di file system

6. SO Multimediali

- File multimediali
- Scheduling dei processi
- Caching e disk scheduling

Organizzazione del corso

7. Analisi delle prestazioni

- Sistemi di congestione
- Analisi delle utilizzazioni
- Capacity planning

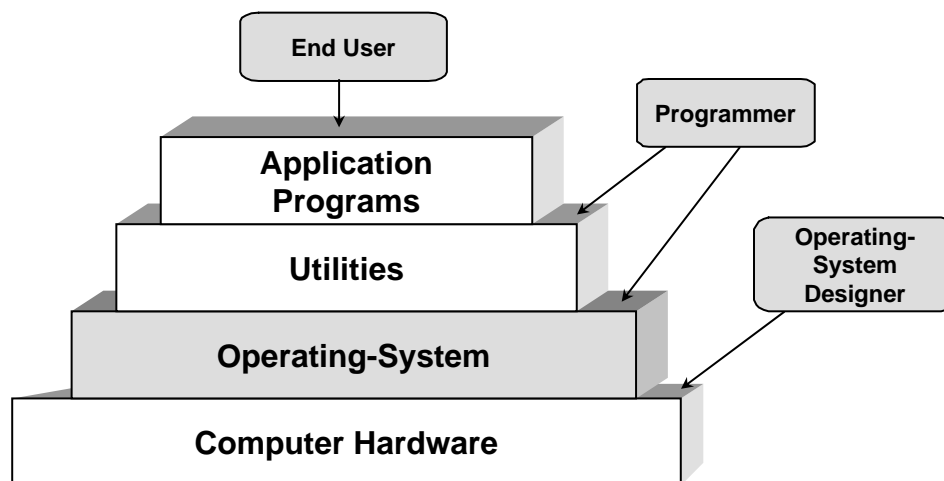
8. Sicurezza e protezione

- Richiami di crittografia
- Autenticazione degli utenti e dei file
- Attacchi e difese

9. Windows

- Struttura del sistema
- Processi e thread
- Gestione della memoria
- Il File System

Il Sistema Operativo



Un Sistema Operativo gestisce le risorse hardware con la finalità di offrire un insieme di servizi agli utenti di un sistema di elaborazione.

Obiettivi del Sistema Operativo

Il SO controlla l'esecuzione di programmi applicativi e svolge funzione di interfaccia tra l'utente e l'hardware

- **Semplicità**

Rende l'uso del computer più semplice, mascherando la complessità della piattaforma hardware

- **Efficienza**

Ottimizza l'uso delle risorse da parte dei programmi applicativi

- **Flessibilità**

Garantisce la trasparenza verso le applicazioni di modifiche dell'hardware, e quindi la portabilità del software

Virtualizzazione delle risorse

- All'utente ed alle applicazioni vengono mostrate *risorse virtuali*, più semplici da usare rispetto alle *risorse reali*
- La corrispondenza tra risorse virtuali e risorse reali è mantenuta dal SO in modo *trasparente* (mascherandone la struttura)
- Le risorse reali nella maggioranza dei casi possono essere solo assegnate in *uso esclusivo*, limitando il parallelismo nella esecuzione delle applicazioni
- La disponibilità di una molteplicità di risorse virtuali, rimuove i vincoli e favorisce l'esecuzione concorrente di più applicazioni

Risorse virtuali

- **CPU**

Tramite la multiprogrammazione più programmi condividono l'uso della CPU, alternandosi

- **Memoria**

Tramite la memoria virtuale lo spazio di memoria dei programmi non è limitato dalla memoria fisica

- **I/O**

L'accesso ai dispositivi di I/O è semplificato, ed il vincolo di uso esclusivo è rimosso

Esempio (gestione di dischi)

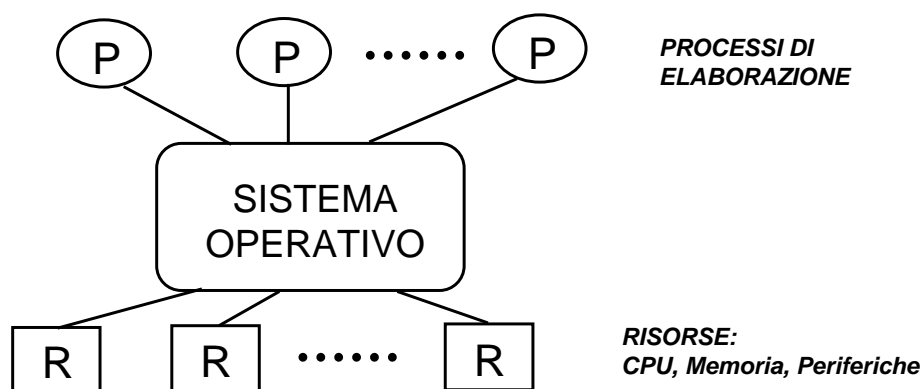
Risorsa reale \Rightarrow drive per floppy disk

Risorsa virtuale \Rightarrow file di I/O

Senza la virtualizzazione occorrerebbe:

- conoscere l'organizzazione fisica del floppy e il linguaggio di comando del controller
- specificare gli indirizzi fisici
- specificare i comandi del controller
 - accendi il motore
 - sposta le testine di n cilindri
 - etc

Gestione delle risorse



- Molteplicità di processi elaborati *concorrentemente*
- Più processi possono necessitare la stessa risorsa
- Il SO svolge ruolo di intermediazione tra processi e risorse

Obiettivi della gestione

CORRETTEZZA

- *L'elaborazione di ciascun processo deve svolgersi indipendentemente da quella degli altri*
- *I risultati devono essere gli stessi che si avrebbero elaborandolo ciascun processo in isolamento*

EFFICIENZA

- *La gestione delle risorse deve essere ottimizzata rispetto ad una serie di indici di prestazione:*
 - Utilizzazione delle risorse
 - Tempi di risposta
 - Affidabilità

Trasparenza e Portabilità

TRASPARENZA

- Il SO assicura la *trasparenza* rispetto alla *piattaforma hardware*
- Le applicazioni fanno riferimento non alla *piattaforma hardware*, ma alla *piattaforma software*, cioè al SO

PORTABILITÀ

- Un'applicazione scritta per un certo SO (es. Windows NT, Linux..) può essere eseguita su qualsiasi piattaforma hardware su cui sia disponibile quel SO
- Su una stessa piattaforma hardware possono essere rese disponibili diversi SO

Servizi del Sistema Operativo

- **Sviluppo di programmi**
Traduttori, collegatori, librerie ecc.
- **Esecuzione di programmi**
Caricamento in memoria, allocazione delle risorse
- **Accesso ai dispositivi di I/O**
Gestione di un'interfaccia logica verso i dispositivi di I/O
- **Accesso alla memoria di massa**
File System, protezione e controllo degli accessi
- **Gestione degli errori**
Recovery automatico, protezione delle applicazioni
- **Accounting**
Addebito dell'uso delle risorse nei contesti multiutente

Evoluzione dei sistemi operativi

Procede parallela all'evoluzione dell'hardware

- **Prima generazione 1945-1955**
 - Tecnologia a valvole e pannelli cablati
 - Sistemi stand-alone
- **Seconda generazione 1955-1965**
 - Tecnologia a transistor
 - Sistemi batch uniprogrammati
- **Terza generazione 1965-1980**
 - Circuiti integrati
 - Multiprogrammazione e memoria virtuale
- **Quarta generazione 1980 ad oggi**
 - Personal Computers

Sistemi di prima generazione

SISTEMI STAND-ALONE

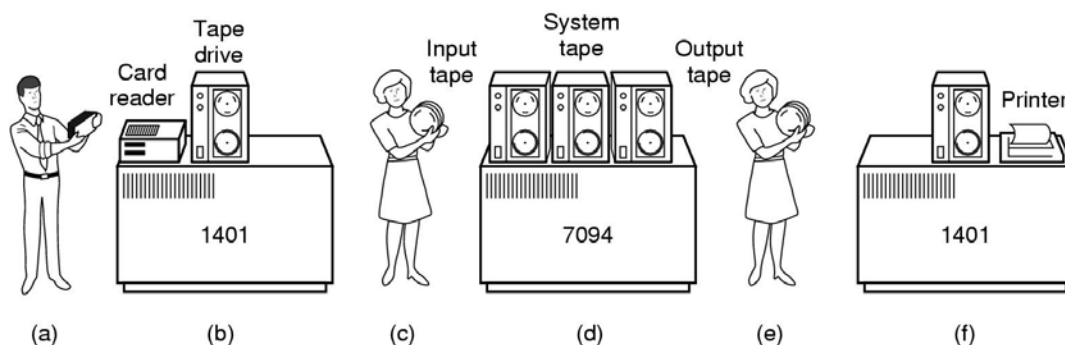
- I primi sistemi non avevano assolutamente sistema operativo
- I programmi utente interagivano direttamente con le risorse hardware

Problemi

- Sviluppo e messa a punto delle applicazioni molto costosa
- Bassissima utilizzazione delle risorse, accesso al sistema con prentazione tipo 'campo da tennis'

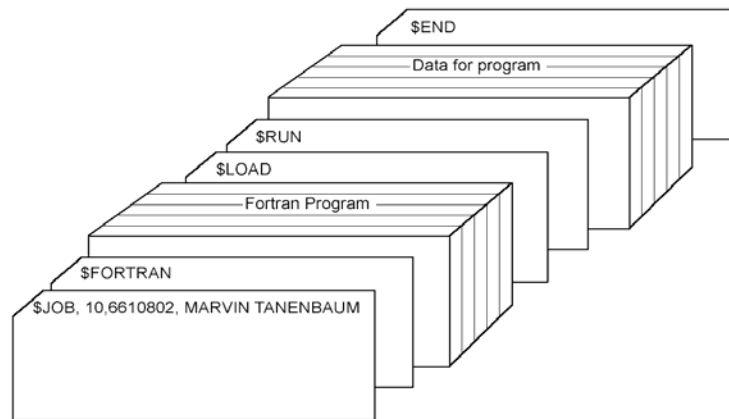
Sistemi di seconda generazione

SISTEMI BATCH



- Sistemi ancillari per la gestione dell'I/O
- Lavori sottomessi a lotti (*batch*)
- Migliore utilizzazione del sistema centrale

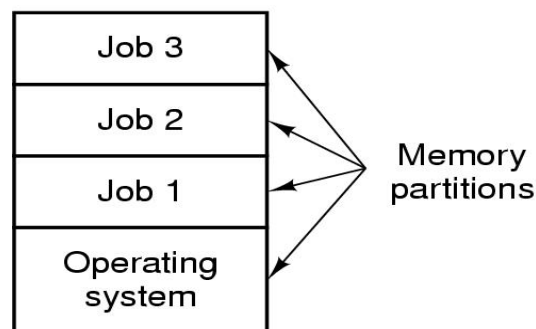
Struttura di un 'job'



- Lavori (*job*) preparato fuori linea e poi sottomesso al sistema
- Le 'schede di controllo' corrispondono a comandi per il sistema operativo

Sistemi di terza generazione

MULTIPROGRAMMAZIONE A PARTIZIONI



- Più lavori (oltre al SO) presenti contemporaneamente in memoria centrale
- CPU concessa a rotazione
- Recupero dei tempi morti di I/O

Multiprogrammazione

- Pool di job contemporaneamente presenti in memoria centrale
- Ciascun job è in condizioni di essere eseguito subito
- Se un job esegue un'operazione di I/O la CPU rimane inattiva per un tempo molto lungo
- La CPU viene immediatamente riassegnata ad un altro dei job del pool
- Eliminazione dei tempi morti di attesa fine I/O
- Migliore utilizzo della CPU
- Numero di job limitato dal numero di partizioni, e dalle dimensioni della memoria centrale

Memoria Virtuale

- Spazio di lavoro dei programmi trasferito dinamicamente in memoria centrale
- Non più necessario caricare i programmi interamente in memoria centrale
- Ciascun programma vede uno spazio di indirizzamento più ampio di quello effettivamente disponibile
- Possibile avere un maggior numero di programmi in memoria centrale
- Maggiore 'livello di multiprogrammazione'
- Migliore utilizzazione delle risorse

Time sharing

- Sistemi operativi per utenza di tipo interattivo
- Necessari tempi di risposta molto bassi
- Pochi utenti residenti in memoria centrale
- Si alternano sulla CPU con tempi molto stretti
- Utenti non residenti sono trasferiti quando necessario da disco in memoria centrale e vice versa (*swapping*)
- Gli utenti hanno l'illusione di avere tutto il sistema a loro disposizione
- Sfrutta i tempi morti degli utenti (*think time*)
- Possibile gestire anche molte decine di utenti

Sistemi di quarta generazione

- Inizialmente l'hardware dei PC non può supportare sistemi operativi evoluti
- Sistemi monoutente e senza multitasking (CPM e DOS)
- L'evoluzione dell'hardware consente di introdurre le feature principali dei sistemi di terza generazione
 - multitasking
 - memoria virtuale
 - file system evoluti
 - supporto multiutente
- Windows, Linux, Mac Os
- Sistemi evoluti ma monoutente (o usati come tali)
- Si sono progressivamente evoluti verso SO di server

Tipologie di SO

- ***Sistemi Operativi per Mainframe***
- ***Sistemi Operativi per Server***
- ***Sistemi Operativi per Multiprocessori***
- ***Sistemi Operativi per Personal Computers***
- ***Sistemi Operativi in Tempo Reale***
- ***Sistemi Operativi Embedded***
- ***Sistemi Operativi per Smart Card***

SO per mainframes

- Gestione di una grande periferia di I/O
- Tipiche applicazioni
 - Grandi server
 - Sistemi centralizzati (banche, linee aeree..)
- Supportano efficacemente tre tipi di servizio
 - Batch
 - Transazionale
 - Time sharing
- Tipico rappresentante MVS, OS/390

SO per server

- Configurazioni 'robuste' di PC o di workstation
- Servono molti utenti in contesti di rete, mettendo a disposizione accessi concorrenti e servizi
 - Stampa, dischi
 - Accesso a Internet (gateway), posta elettronica etc.
- Tipici rappresentanti: UNIX, Windows 2000, Linux
- Versioni particolari per piattaforme parallele
 - SMP (Symmetric MultiProcessing)
 - Asimmetrico
- Windows 2000 Datacenter, versioni speciali di UNIX

SO real-time

- Sistemi altamente specializzati
- Gestione di applicazioni con forti vincoli real-time
 - Controllo di processo
 - Sistemi di telecomunicazione e multimedia
- Scadenze (*deadlines*) definite per i task
- Hard real-time
 - Hard deadlines
 - Piattaforme hardware ad hoc
- Soft real-time
 - Sporadicamente possono mancare una deadline

Sistemi embedded e Smart Card

- Sistemi embedded
 - PDA (Personal Digital Assistent)
 - Telefoni cellulari
 - Elettrodomestici
- Risorse estremamente limitate
- Palm OS, Windows CE
- SM (Smart Cards)
 - Risorse ancora più scarse
 - SO basati su Java
 - Anche esecuzione concorrente di più applets

Sistemi proprietari e sistemi aperti

- Sistema proprietario: progettato da un costruttore con riferimento ad una sua particolare piattaforma hardware
 - Sfruttano le peculiarità dell'HW
 - Non consentono la portabilità del software
 - *Esempi*: IBM-MVS, DEC-VMS
- Sistema aperto: architettura indipendente dalla piattaforma hardware:
 - Disponibili su diverse piattaforme
 - Favoriscono la portabilità del software
 - Esempi: UNIX, NT4

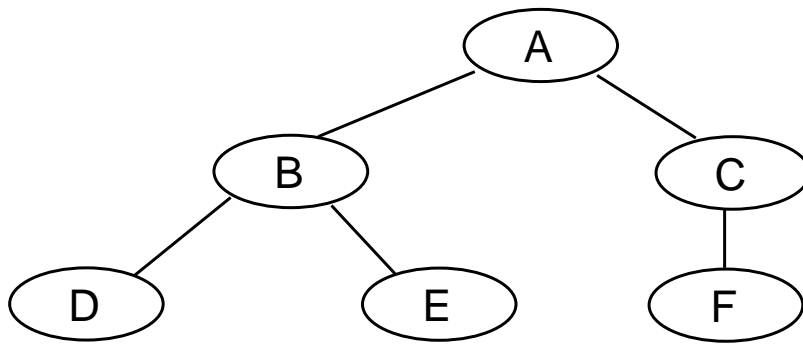
Processi di elaborazione

- Concetto fondamentale su cui si basano tutti i moderni sistemi operativi
- Introdotti per monitorare e controllare in modo sistematico l'esecuzione dei programmi
- Un processo è un programma eseguibile con associati:
 - I dati su cui opera
 - Un contesto di esecuzione: cioè le informazioni necessarie alla CPU per eseguirlo, ed al SO per schedularlo

Gestione dei processi

- Il SO utilizza particolari strutture per mantenere tutta l'informazione relativa a ciascun processo, ed il suo stato
- Quando un processo va in esecuzione sulla CPU può interagire sia con il SO che con altri processi
- Tutte le applicazioni non banali sono costituite da più processi: modularità e parallelismo
- Anche il SO è costituito da un insieme di processi
- Nelle applicazioni di rete i processi possono risiedere su sistemi diversi

Gerarchie di processi



- Un processo può creare altri processi (figli) demandandogli parte dei propri compiti
- In base ai rapporti di parentela esistono gerarchie di processi

Chiamate di sistema (system call)

- Costituiscono l'interfaccia tra i programmi utente ed il Sistema Operativo
- Corrispondono a tutte le operazioni 'delicate' che ai programmi non è consentito effettuare direttamente
 - Operazioni di I/O
 - Allocazione di risorse
 - Interazioni tra processi
- Le operazioni vengono effettuate in loro vece dal SO, che mantiene così il controllo della situazione

Lo standard POSIX

- Le chiamate di sistema sono disponibili sia a livello di linguaggio assembler che come librerie per linguaggi ad alto livello (es. C)
- Le librerie differiscono da SO a SO, e quindi precludono la portabilità del software
- POSIX (ISO 9945-1) definisce un'insieme di chiamate standard (circa 100)
- Molti SO sono compatibili (*compliant*) con POSIX
- Il riferimento a POSIX garantisce la portabilità delle applicazioni

Esempio: lettura da file

- Funzione di libreria chiamata da programma C
$$\text{count} = \text{read}(\text{fd}, \text{buffer}, \text{nbytes})$$
- **fd** e **nbytes** passati per valore, **buffer** per riferimento
- Alla funzione C corrisponde direttamente una chiamata di sistema, con esattamente gli stessi parametri
- Il codice macchina corrispondente alla funzione
 - Prepara i parametri
 - Provoca una trap: il controllo passa al SO
 - Il processo chiamante viene sospeso

POSIX: gestione di processi

Process management

Call	Description
pid = fork()	Create a child process identical to the parent
pid = waitpid(pid, &statloc, options)	Wait for a child to terminate
s = execve(name, argv, environp)	Replace a process' core image
exit(status)	Terminate process execution and return status

- Sono le principali chiamate di sistema
- Corrispondono al classico meccanismo UNIX di creazione dei processi
- Un processo genera una sua replica (*figlio*) tramite la **fork**
- Il figlio muta il suo comportamento tramite la **execve**

fork

pid = fork()

- La **fork** crea un duplicato del processo chiamante (stesso codice, registri, file etc.)
- Dopo la **fork** le strade si separano, ciascuno ha le sue variabili, solo il codice è in comune
- **pid** torna 0 per il *figlio* e il PID (*process identifier*) del figlio per il *genitore*
- Informazione è utilizzata nel codice per differenziare il comportamento del figlio da quello del genitore
- La 'mutazione' viene fatta dal figlio eseguendo una **execve**

execve

`s = execve(name,argv,environp)`

- Una di varie chiamate di libreria corrispondenti all'unica system call **exec**
- La **execve** rimpiazza il codice del processo con quello contenuto nel file **name**
- In genere il figlio esegue la mutazione subito dopo la **fork** in base ad informazioni passategli dal genitore
 - **argv** punta ad un array di parametri
 - **environp** punta ad un array contenente informazioni di contesto

waitpid

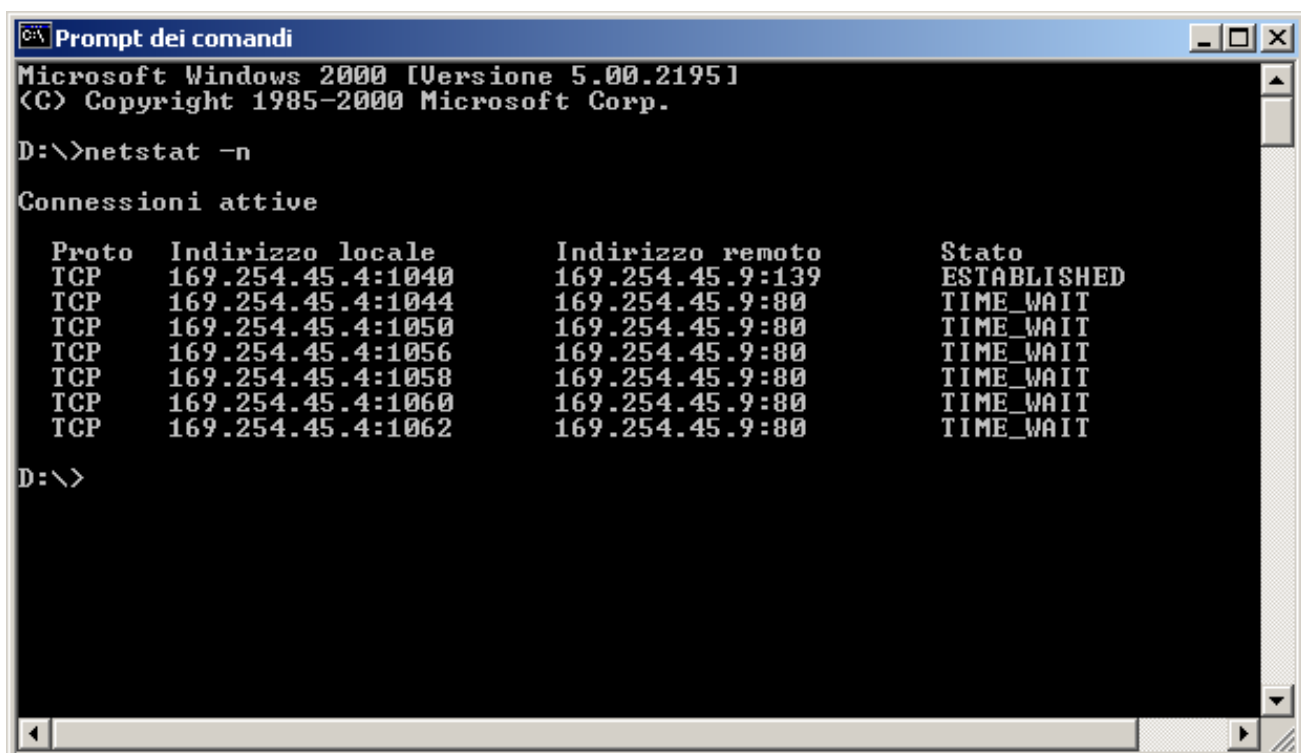
`pid = waitpid(pid,&statloc,options)`

- La **waitpid** blocca il processo chiamante e lo mette in attesa della terminazione del figlio di cui specifica il **pid**
- Quando il genitore crea un figlio e gli demanda un compito, prima o poi esegue una **waitpid**
 - **pid** indica il processo che si vuole aspettare (qualsiasi figlio se vale -1)
 - **statloc** ritorna con lo stato del figlio all'atto della terminazione (normale o meno)
 - **options** specifica opzioni di vario tipo

Esempio: una semplice 'shell'

- La *shell* è l'interprete dei comandi nella classica interfaccia UNIX a linea di comando
- Qualcosa di simile è costituito dall'interprete dei comandi del DOS, presente anche in Windows
- È il processo che gestisce l'interfaccia utente
- Percorre indefinitamente un ciclo
 - Accetta una linea contenente un nuovo comando
 - Crea un figlio cui demanda l'esecuzione del comando
 - Attende la terminazione del figlio
 - Stampa il *command prompt* e ricomincia il ciclo

Il command prompt di Windows



```
Microsoft Windows [Versione 5.00.2195]
(C) Copyright 1985-2000 Microsoft Corp.

D:\>netstat -n

Connessioni attive

Proto  Indirizzo locale          Indirizzo remoto          Stato
TCP    169.254.45.4:1040         169.254.45.9:139        ESTABLISHED
TCP    169.254.45.4:1044         169.254.45.9:80         TIME_WAIT
TCP    169.254.45.4:1050         169.254.45.9:80         TIME_WAIT
TCP    169.254.45.4:1056         169.254.45.9:80         TIME_WAIT
TCP    169.254.45.4:1058         169.254.45.9:80         TIME_WAIT
TCP    169.254.45.4:1060         169.254.45.9:80         TIME_WAIT
TCP    169.254.45.4:1062         169.254.45.9:80         TIME_WAIT

D:\>
```

Una semplice shell

```
while (TRUE) {                               /* repeat forever */
    type_prompt( );                           /* display prompt */
    read_command (command, parameters)       /* input from terminal */

if (fork() != 0) {                             /* fork off child process */
    /* Parent code */
    waitpid( -1, &status, 0);                /* wait for child to exit */
} else {
    /* Child code */
    execve (command, parameters, 0);         /* execute command */
}
}
```

POSIX: gestione dei file

File management

Call	Description
fd = open(file, how, ...)	Open a file for reading, writing or both
s = close(fd)	Close an open file
n = read(fd, buffer, nbytes)	Read data from a file into a buffer
n = write(fd, buffer, nbytes)	Write data from a buffer into a file
position = lseek(fd, offset, whence)	Move the file pointer
s = stat(name, &buf)	Get a file's status information

POSIX: gestione del file system

Directory and file system management

Call	Description
s = mkdir(name, mode)	Create a new directory
s = rmdir(name)	Remove an empty directory
s = link(name1, name2)	Create a new entry, name2, pointing to name1
s = unlink(name)	Remove a directory entry
s = mount(special, name, flag)	Mount a file system
s = umount(special)	Unmount a file system

POSIX: chiamate miscellanee

Miscellaneous

Call	Description
s = chdir(dirname)	Change the working directory
s = chmod(name, mode)	Change a file's protection bits
s = kill(pid, signal)	Send a signal to a process
seconds = time(&seconds)	Get the elapsed time since Jan. 1, 1970

- La **kill** permette di mandare 'segnali' ai processi
- Se il processo destinatario è in attesa di un segnale, questo gli viene comunicato e la sua esecuzione riprende
- Se il destinatario non è in attesa viene 'ucciso', donde il macabro nome della system call

Win32 API

- Win32 API (Application Program Interface) è la libreria di Windows tramite la quale si accede alle system call
- Supportata (parzialmente) da tutte le versioni di Windows
- Non a tutte le procedure, in tutte le implementazioni, corrispondono in realtà chiamate di sistema
- Molte chiamate non coinvolgono il kernel, cioè vengono gestite nello *user space*
- Molte migliaia di chiamate
- Molte chiamate sono relative alla GUI (Graphical User Interface), creazione di finestre, menu, scrollbar etc.

Win32 API (esempi)

UNIX	Win32	Description
fork	CreateProcess	Create a new process
waitpid	WaitForSingleObject	Can wait for a process to exit
execve	(none)	CreateProcess = fork + execve
exit	ExitProcess	Terminate execution
open	CreateFile	Create a file or open an existing file
close	CloseHandle	Close a file
read	ReadFile	Read data from a file
write	WriteFile	Write data to a file
lseek	SetFilePointer	Move the file pointer
stat	GetFileAttributesEx	Get various file attributes
mkdir	CreateDirectory	Create a new directory
rmdir	RemoveDirectory	Remove an empty directory
link	(none)	Win32 does not support links
unlink	DeleteFile	Destroy an existing file
mount	(none)	Win32 does not support mount
umount	(none)	Win32 does not support mount
chdir	SetCurrentDirectory	Change the current working directory
chmod	(none)	Win32 does not support security (although NT does)
kill	(none)	Win32 does not support signals
time	GetLocalTime	Get the current time