Genesi – 1

- **DTSS** (*Dartmouth College Time Sharing System*, 1964)
  - Il primo elaboratore multi-utente a divisione di tempo
  - Programmato in BASIC e ALGOL
  - Presto soppiantato da
- **CTSS** (*MIT Compatible Time Sharing System*, in versione sperimentale dal 1961)
  - Enorme successo nella comunità scientifica
  - Induce MIT, Bell Labs e GE alla collaborazione nel progetto di
- **MULTICS** (*Multiplexed Information and Computing Service*, 1965)
  - Quando Bell Labs abbandona il progetto, Ken Thompson, uno degli autori di MULTICS, ne produce in **assembler** una versione a utente singolo
- **UNICS** (UNiplexed "ICS", 1969) **UNIX**

Genesi – 2

- **1974**
  - Nuova versione di UNIX per PDP-11 completamente riscritta in **C** con Dennis Ritchie
    - PDP-11 (*Programmed Data Processor*)
      - 2 KB *cache*, 2 MB RAM
    - Linguaggio C definito appositamente come evoluzione del rudimentale **BCPL** (*Basic Combined Programming Language*)
  - Enorme successo grazie alla diffusione di PDP-11 nelle università
- **1979**
  - Rilascio di **UNIX v7**, "la" versione di riferimento
    - Perfino Microsoft lo ha inizialmente commercializzato!
      - Sotto il nome di **Xenix**, ma solo a costruttori dell'*hardware* degli elaboratori (p.es.: Intel)

Genesi – 3

- **Portabilità di programmi**
  - Programma scritto in un linguaggio ad alto livello dotato di compilatore per più elaboratori
    - È desiderabile che anche il compilatore sia portabile
  - Dipendenze limitate ad aspetti specifici della architettura di destinazione
    - Dispositivi di I/O, gestione interruzioni, gestione di basso livello della memoria
- **Diversificazione degli idiomi** (1979 – 1986)
  - Avvento di **v7** e divaricazione in due filoni distinti
    - **System V** (AT&T → Novell → **S**anta **C**ruz **O**peration)
      - Incluso Xenix …
    - **4.x BSD** (*Berkeley Software Distribution*, da Bill Joy che poi fonda Sun Microsystems nel 1982)
      - 4.0 rilasciato nell'ottobre 1980, poi evoluto sino a 4.4 Lite R2 (1995)

Genesi – 4

- **Standardizzazione** (1986 –)
  - **POSIX** (*Portable Operating System Interface for UNIX*) racchiude elementi selezionati di **System V** e **BSD**
    - I più maturi e utili secondo l'opinione di esperti "neutrali" incaricati da IEEE e ISO/IEC
    - Definisce l'insieme standard di **procedure di libreria** utili per operare su S/O compatibili
      - La maggior parte contiene chiamate di sistema standard
      - Servizi utilizzabili da linguaggi ad alto livello

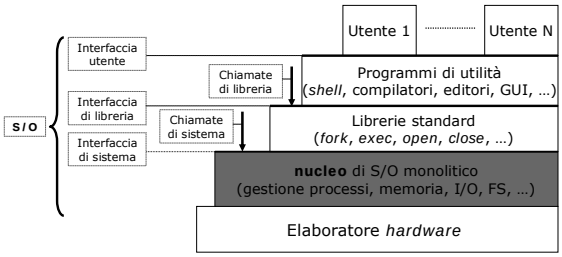
Genesi – 5

- **Scelte architetturali per cloni UNIX**
  - ***Micro-kernel*** : MINIX (Tanenbaum, 1987)
    - Nel nucleo **solo** processi e comunicazione
    - Il resto dei servizi (p.es. : FS) realizzato in processi utente
      - MINIX non copre tutti i servizi UNIX
  - **Nucleo monolitico** : GNU/Linux (Linus Torvalds, da v0.01 nel maggio 1991 a v2.6 di oggi)
    - Clone completo aderente a POSIX con qualche libertà
      - Il "meglio" di BSD per esecuzione su PC
    - Modello *open-source* (scritto nel C compilato da **gcc** – ***GNU** C compiler*)
    - Parallelo al progetto **GNU** (GNU is NOT UNIX!) `http://www.gnu.org`

Visione generale – 1

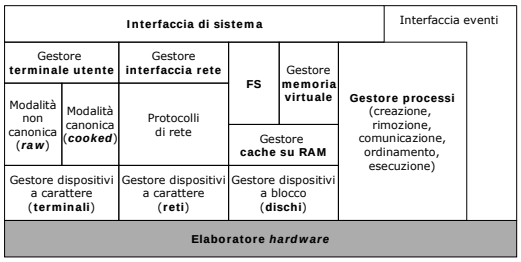
- Sistema **multi-programmato** multi-utente
  - Orientato allo **sviluppo** di applicazioni
  - Progettato per semplicità, eleganza e consistenza
    - **NO** a complessità derivante da compromessi di compatibilità verso il passato
    - **SI** a specializzazione e flessibilità
      - Ogni servizio fa una sola cosa (bene)
      - Facile combinare liberamente più servizi per realizzare azioni più sofisticate
- Architettura a livelli gerarchici

Livelli gerarchici



Architettura di nucleo UNIX

Struttura monolitica = insieme staticamente fissato di moduli oggetto



Interfaccia utente

- UNIX nasce con I/F per linea di comando (*shell*)
  - Più potente e flessibile di GUI ma solo per utenti esperti
  - Una *shell* per ogni terminale utente (**`xterm`**)
    - Lettura dal *file* "*standard input*"
    - Scrittura sul *file* "*standard output*" o "*standard error*"
      - Inizialmente associati al terminale stesso (visto come *file*)
    - Possono essere re-diretti
      - < per `stdin`, > per `stdout`
  - Caratteri di *prompt* indicano dove editare il comando
  - Comandi compositi possono associare l'uscita di comandi all'ingresso di altri mediante | (*pipe*)
    - Oppure essere combinati in sequenze interpretate (*script*)
  - In modalità normale la *shell* esegue un comando alla volta
    - Ma i comandi possono essere inviati all'esecuzione liberando la *shell* (&, *background*)

Gestione dei processi (UNIX) – 1

- **Processo**
  - La sola entità attiva nel sistema
  - Inizialmente definito come sequenziale
    - Dotato di un singolo flusso di controllo interno
  - Concorrenza a livello di processi
    - Molti processi attivati direttamente dal sistema (*daemon*)
    - Creazione mediante **`fork`**`()`
    - La discendenza di un processo costituisce un "gruppo"
    - Comunicazione mediante scambio messaggi (*pipe*) e invio di segnali (*signal*) entro un gruppo

Gestione dei processi (UNIX) – 2

- Processi con più flussi di controllo interni
  - Detti *thread*
    - Apportano maggior capacità di calcolo entro un processo
  - 2 scelte realizzative possibili per gestire *thread*
    - Nello spazio di nucleo → deve cambiare il nucleo!
    - Nello spazio di utente
      - Più facile da aggiungere
      - Il nucleo però ordina solo i processi
  - POSIX non impone una particolare scelta
  - Un *thread* può svolgere compiti specializzati
  - La creazione di un *thread* gli assegna identità, attributi, compito e argomenti

```
res = pthread_create( &tid , attr , fun , arg )
```

Gestione dei processi (UNIX) – 3

- Completato il proprio lavoro il *thread* termina se stesso volontariamente
  - Invocando la procedura **`pthread_exit`**
- Un *thread* può sincronizzarsi con la terminazione di un suo simile
  - Invocando la procedura **`pthread_join`**
- L'accesso a risorse condivise viene sincronizzato mediante semafori a mutua esclusione
  - Tramite le procedure **`pthread_mutex{_init, _destroy}`**
- L'attesa su condizioni logiche (p.es. : risorsa libera) avviene mediante variabili speciali simili a *condition variables* (ma senza *monitor*)

Gestione dei processi (UNIX) – 4

- Un *thread* può avere 2 "modi" operativi
  - Modo **normale**
    - Esecuzione nello spazio di utente con diritti, memoria virtuale, risorse rigidamente distinti da quelli del nucleo
  - Modo **nucleo**
    - Esecuzione con privilegi, memoria virtuale e risorse di nucleo
  - Il **cambio di modo** consegue a una **chiamata di sistema**
    - Chiamata incapsulata in una **procedura di libreria** per consentirne l'invocazione da programmi scritti in linguaggi ad alto livello (p.es. C)

Gestione dei processi (UNIX) – 5

- Il nucleo impiega 2 strutture per gestire processi
  - **Tabella dei processi**
    - **Permanentemente in RAM** e per **tutti** i processi
      - Parametri di ordinamento (p.es. : priorità, tempo di esecuzione cumulato, tempo di sospensione in corso, …)
      - Descrittore della memoria virtuale del processo
      - Lista dei segnali significativi e del loro stato
      - Stato, identità, relazioni di parentela, gruppo di appartenenza
  - **Descrittore degli utenti**
    - In RAM **solo** per i processi attualmente **attivi**
      - Parte del contesto (immagine dei registri dell'elaboratore)
      - Stato dall'ultima chiamata di sistema (parametri, risultato)
      - Tabella dei descrittori dei *file* aperti
      - Crediti del processo
      - *Stack* da usare in modo nucleo

Esecuzione di comando di *shell* – 1

Codice semplificato di un processo *shell*

```
while (TRUE) {
   type_prompt(); // mostra prompt sullo schermo
   read_command(cmd, par); // legge linea comando
1  pid = fork();
   if (pid < 0) {
      printf("Errore di attivazione processo.\n");
      continue; // ripeti ciclo
   };
   if (pid != 0) {
     waitpid(-1, &status, 0); // attende la terminazione
                              // di qualunque figlio
   } else {
2    execve(cmd, par, 0);
   }
}
```

Codice eseguito dal padre

Codice eseguito dal figlio

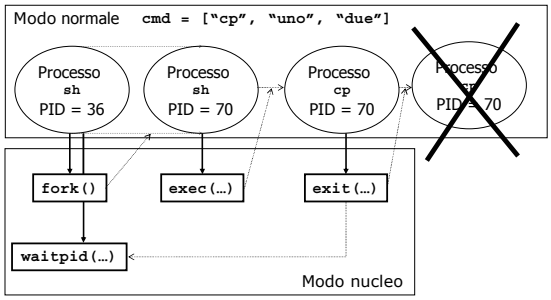
Esecuzione di comando di *shell* – 2

1. Il processo chiamante passa in **modo nucleo** e prova a inserire i propri dati per il figlio nella **Tabella dei Processi** (incluso il PID). Se ci riesce, alloca memoria per *stack* e dati del figlio. A questo punto il codice del figlio è ancora **lo stesso** del padre

2. La linea di comando emessa dall'utente viene passata al processo figlio come *array* di stringhe. La **exec**, che opera in **modo nucleo**, localizza il programma da eseguire e lo sostituisce al codice del chiamante, passandogli la linea di comando e le "**definizioni di ambiente**" specifiche per il nuovo processo

Le aree dati e codice sono copiate per il figlio solo se questi le modifica (***copy-on-write***) e caricate solo quando riferite (***paging-on-demand***)

Esecuzione di comando di *shell* – 3



Gestione dei processi (UNIX) – 6

- `fork()` **duplica** il processo chiamante creando un **processo figlio** uguale a se stesso ma distinto
  - Che accade se il processo creante comprende più *thread* ?
- Vi sono 2 possibilità
  - **Tutti i *thread* del padre vengono clonati**
    - Difficile gestire il loro accesso concorrente a dati e risorse condivise con i *thread* del padre
  - **Solo un *thread* del padre viene clonato**
    - Possibile sorgente di inconsistenza rispetto alle esigenze di cooperazione con le *thread* non clonate
- Dunque il *multi-threading* aggiunge gradi di complessità
  - Al FS
    - Più difficile assicurare consistenza nell'uso concorrente di *file*
  - Alla comunicazione tra entità attive
    - Quale *thread* deve essere il destinatario di segnali inviati al processo?
- `fork()` non era stata pensata per uso da processi con *thread* !

Gestione dei processi (GNU/Linux)

- Maggior granularità nel trattamento della condivisione di strutture di controllo nella creazione di processi e *thread* figli
- Chiamata di sistema alternativa a **`fork`**()

```
pid = clone(fun, stack, ctrl, par);
```

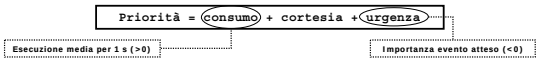
  - `fun` = programma da eseguire nel nuovo "*task*" (processo o *thread*) con argomento `par`
  - `stack` = indirizzo dello *stack* assegnato al nuovo *task*
  - `ctrl` = grado di condivisione desiderato tra il nuovo *task* e l'ambiente del chiamante
    - Spazio di memoria, FS, *file*, segnali, identità

Ordinamento dei processi (UNIX) – 1

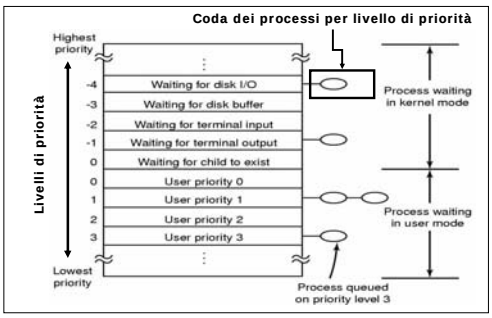
- **Sistema destinato a uso interattivo**
  - Politica di ordinamento con prerilascio
  - Deve garantire buoni tempi di risposta e soddisfare le aspettative degli utenti
- **Algoritmo di ordinamento a 2 livelli**
  - Livello basso (***dispatcher***)
    - Seleziona un processo tra quelli pronti (in RAM)
      - Secondo le indicazioni fornite dallo *scheduler*
  - Livello alto (***scheduler***)
    - Assicura che tutti i processi abbiano opportunità di esecuzione spostando processi tra RAM e disco

Ordinamento dei processi (UNIX) – 2

- **Politica di ordinamento**
  - Selezione per priorità decrescente
    - Priorità alta (< 0) per esecuzione in modo nucleo
    - Priorità bassa (> 0) per esecuzione in modo normale
  - Più processi possono avere la stessa priorità
    - 1 coda di processi ∀ livello di priorità
    - Selezione *round robin* da testa della coda
    - Esecuzione a quanti con ritorno in fondo alla coda
    - Aggiornamento periodico della priorità dei processi



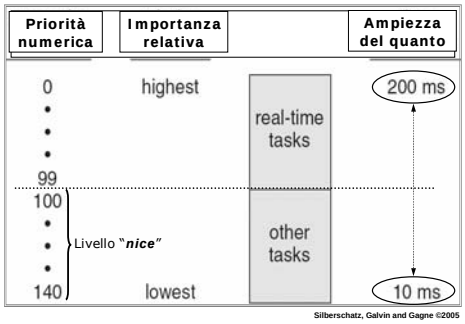
Ordinamento dei processi (UNIX) – 3



Ordinamento dei processi (GNU/Linux) – 1

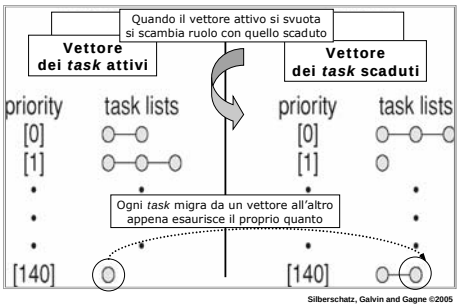
- I *thread* sono gestiti direttamente dal nucleo
  - Ordinamento per "*task*" (*thread* o processo indistintamente)
  - Selezione distinta tra classi distinte
  - Prerilascio per fine quanto o per attesa di evento
- 3 **classi di priorità** di *task*
  - **Tempo reale con politica FCFS a priorità senza prerilascio**
    - A priorità uguale viene scelto il *task* in attesa da più tempo
  - **Tempo reale con politica RR a priorità**
    - Prerilascio per quanti con ritorno in fondo alla coda
  - **Divisione di tempo RR a priorità**
    - Priorità dinamica con premio o penalità rispetto al grado di interattività con I/O (alta → premio, bassa → penalità)
    - Nuovo valore assegnato al *task* all'esaurimento del suo quanto corrente

Ordinamento dei processi (GNU/Linux) – 2



Silberschatz, Galvin and Gagne ©2005

Ordinamento dei processi (GNU/Linux) – 3

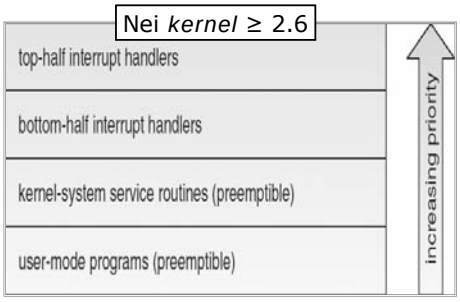


Silberschatz, Galvin and Gagne ©2005

Ordinamento dei processi (GNU/Linux) – 4

- Per versione < 2.6 l'attività dei processi in modo nucleo **non ammetteva** prerilascio
  - Ma questo causava gravi problemi di **inversione di priorità**
- Con versione ≥ 2.6 si usa granularità più fine
  - Inibizione selettiva di prerilascio
    - Per sezioni critiche corte
  - Uso di semafori convenzionali
    - Per sezioni critiche lunghe
  - Uso minimale di disabilitazione delle interruzioni
    - La parte **immediata** dei gestori (*top half*, molto breve) disabilita
    - La parte **differita** (*bottom half*, più lunga) non disabilita ma la sua esecuzione viene privilegiata rispetto a ogni altra attività
      - Tranne che ad ogni eventuale altra parte immediata

Ordinamento dei processi (GNU/Linux) – 5

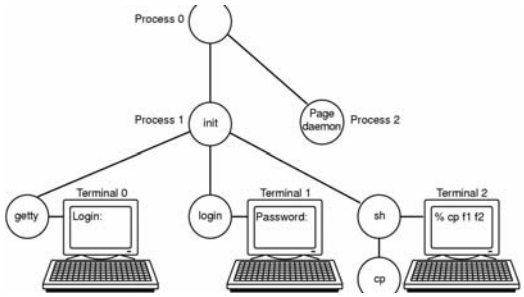


Silberschatz, Galvin and Gagne ©2005

Inizializzazione (GNU/Linux) – 1

- BIOS carica l'MBR in RAM e lo "esegue"
  - MBR = 1 settore di disco = 512 B
- L'MBR carica il programma di *boot* dal corrispondente blocco della partizione attiva selezionata
  - Lettura della struttura di FS, localizzazione e caricamento del nucleo di S/O
- Il programma di inizializzazione del nucleo è scritto in *assembler* (specifico per l'elaboratore!)
  - Poche azioni di configurazione di CPU e RAM
  - Il controllo passa poi al `main` di nucleo
    - Configurazione del sistema con caricamento **dinamico** dei gestori dei dispositivi rilevati
    - Inizializzazione e attivazione del processo 0

Inizializzazione (GNU/Linux) – 2

- **Processo 0**
  - Configurazione degli orologi, installazione del **FS di sistema**, creazione dei processi **1** (`init`) e **2** (`daemon` delle pagine)
- **Processo 1**
  - Configurazione modo utente (singolo, multi)
    - Esecuzione *script* di inizializzazione `shell` (`/etc/rc` etc.)
    - Lettura numero e tipo terminali da `/etc/ttys`
    - `fork()` ∀ terminale abilitato ed `exec("getty")`
- **Processo `getty`**
  - Configurazione del terminale e attivazione del *prompt* di *login*
  - Al *login* di utente, `exec("/usr/bin/login")` con verifica della *password* d'accesso (memorizzate in `/etc/passwd`)
  - Infine `exec("shell")` e poi si procede come mostrato alle diapositive 269-271

Inizializzazione (GNU/Linux) – 3