Sistemi Operativi
Da Unix a GNU/Linux (parte 1)

Docente: Claudio E. Palazzi

cpalazzi@math.unipd.it

Crediti per queste slides al Prof. Tullio Vardanega

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)

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 (Microsoft)
    - **4.x BSD** (*Berkeley Software Distribution*)
      - Incluso Virtual Memory e TCP/IP

Genesi – 4

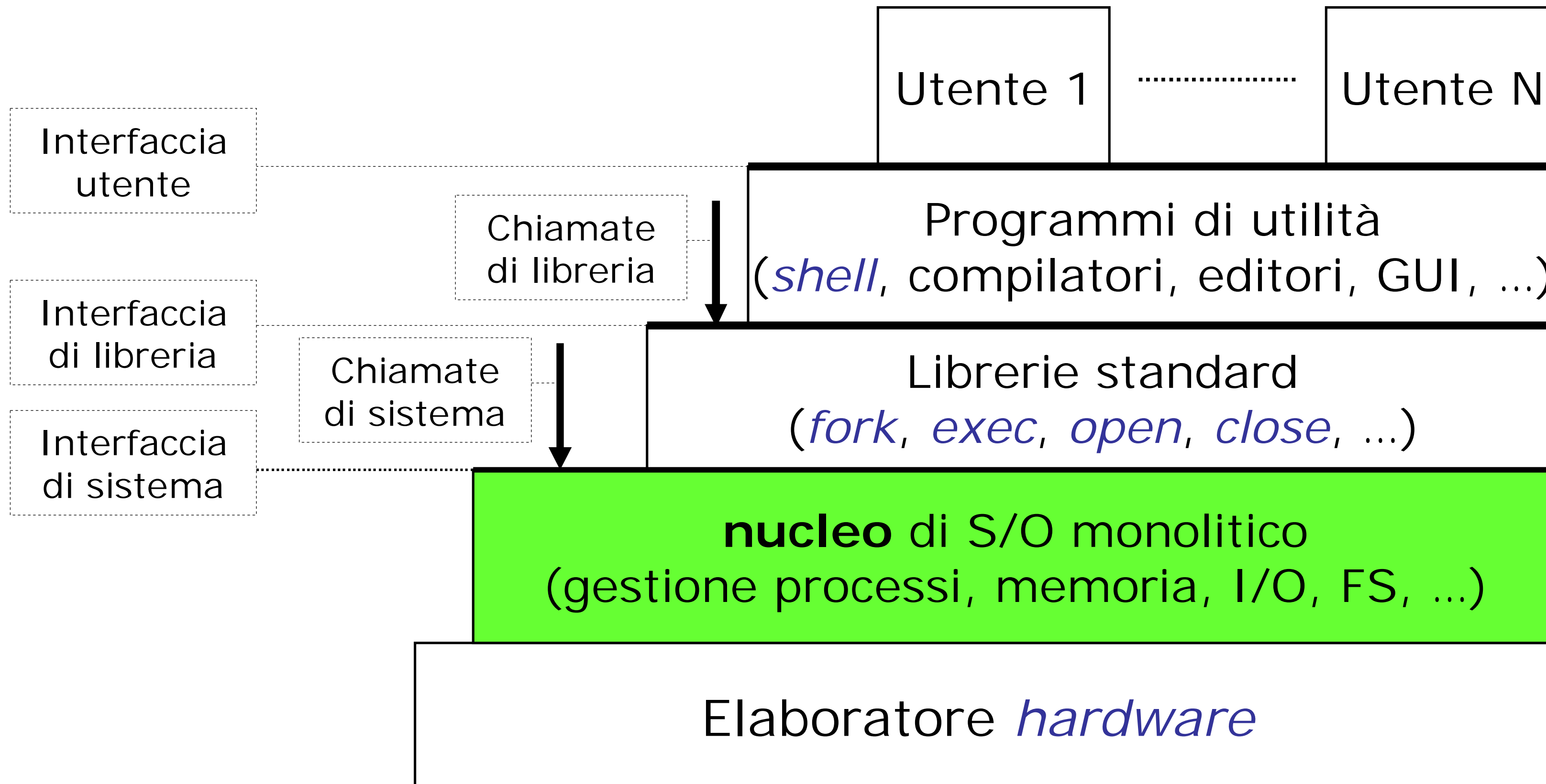
- **Standardizzazione** (1986 –)
  - **POSIX** (*Portable Operating System Interface for UNIX*) racchiude elementi selezionati di **System V** e **BSD**
    - I più maturi e utili secondo l'opinione di esperti "neutrali" incaricati da IEEE e ISO/IEC
    - Definisce l'insieme standard di **procedure di libreria** utili per operare su S/O compatibili
      - La maggior parte contiene chiamate di sistema
      - 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 (affidabile e efficiente)
    - 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 v3.8 di oggi)
    - Clone completo aderente a POSIX con qualche libertà
      - Il "meglio" di BSD e System V
    - Modello *open-source* (scritto nel C compilato da **gcc** – ***GNU*** *C compiler*)
  - Da non perdere: "Linux is obsolete"

Livelli gerarchici

- Sistema **multi-programmato** multi-utente
- Architettura a livelli gerarchici



Interfaccia utente

- UNIX nasce con I/F per linea di comando (*shell*)
  - Più potente e flessibile di GUI ma destinato a 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 uscita di comandi ad ingresso di altri mediante | (*pipe*) e combinati in sequenze interpretate (*script*)
  - In modalità normale la *shell* esegue un comando alla volta
  - Comandi possono essere inviati all’esecuzione liberando la *shell* (&, *background*)

Gestione dei processi (UNIX) – 1

- **Processo**
  - La principale entità attiva nel sistema
  - Inizialmente definito come sequenziale
    - Ossia dotato di un singolo flusso di controllo interno
  - Concorrenza a livello di processi
    - Molti processi attivati direttamente dal sistema (*daemon*)
      - appuntamenti
    - Creazione mediante **`fork()`**
      - Clone con stessa memoria all'inizio e accesso a file aperti
    - La discendenza di un processo costituisce un "gruppo"
    - Comunicazione mediante scambio messaggi (*pipe*) e invio di segnali (*signal*) entro un gruppo
  - Processi figli hanno memoria identica quella del processo genitore <u>solo all'inizio</u>
    - Poi **indipendente** alla prima modifica

Gestione dei processi (UNIX) – 2

- Processi con più flussi di controllo interni
  - Detti *thread*
  - La creazione di un *thread* gli assegna identità, attributi, compito e argomenti

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

  - Thread <u>condividono tutte le risorse</u> logiche e fisiche del processo genitore (inclusi valori di variabili in uso)

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

- **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

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

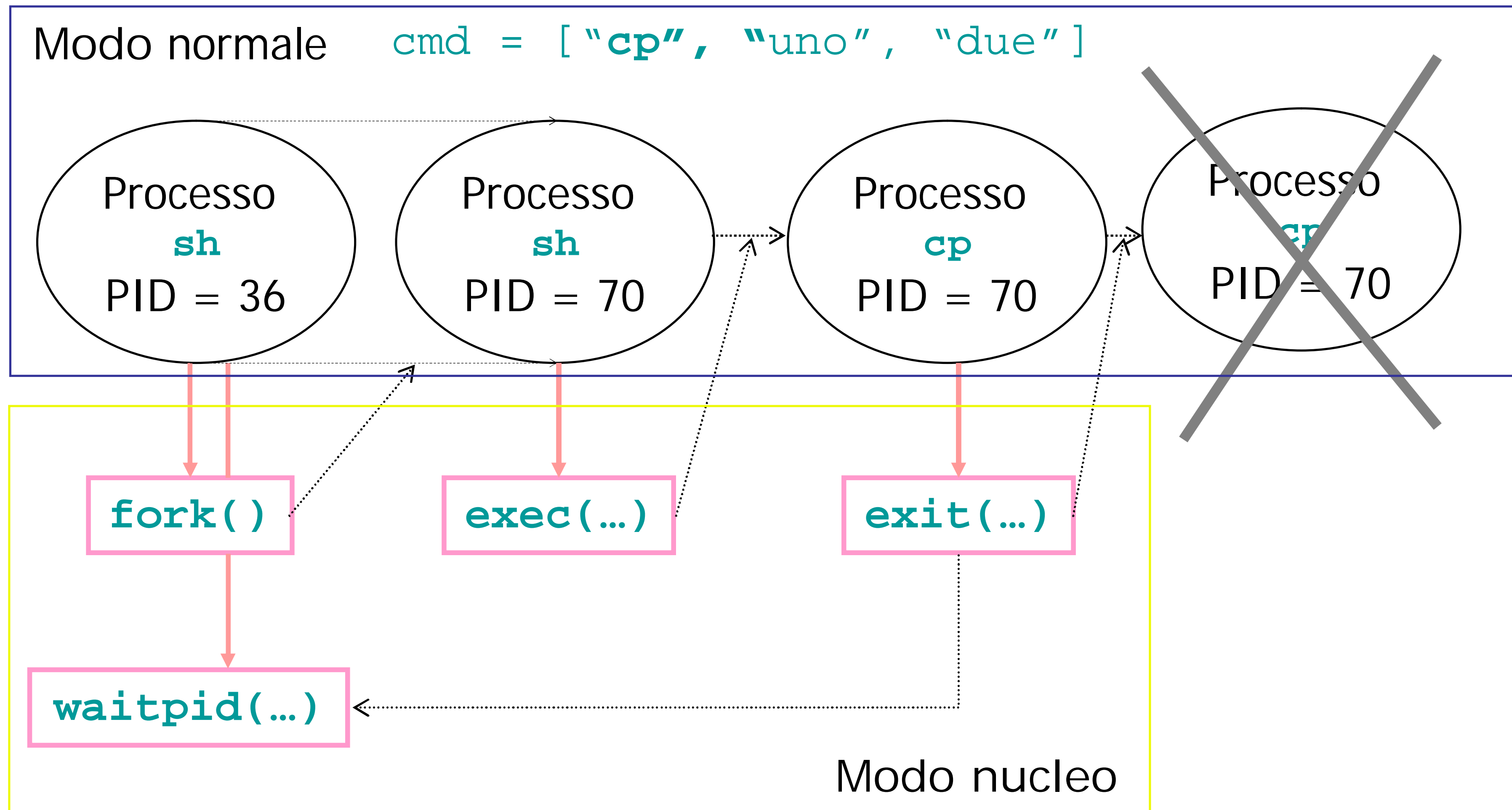
1

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 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

Esecuzione di comando di *shell* – 3



Vedi anche Fig. 10.8 nel libro di testo (terza edizione)

Gestione dei processi (UNIX) – 5

- `fork()` duplica il processo chiamante creando un processo figlio uguale ma distinto
  - Che accade se questi include più *thread*?
- Vi sono 2 possibilità
  - **Tutti i *thread* del padre vengono clonati**
    - Difficile gestire il loro accesso concorrente ai dati e alle risorse condivise con *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
    - Come decidere il *thread* destinazione di un segnale inviato a un processo?
    - Un thread era bloccato da lettura da tastiera, i nuovi thread anche? E chi riceverà i caratteri digitati?

Gestione dei processi (GNU/Linux) – 6

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

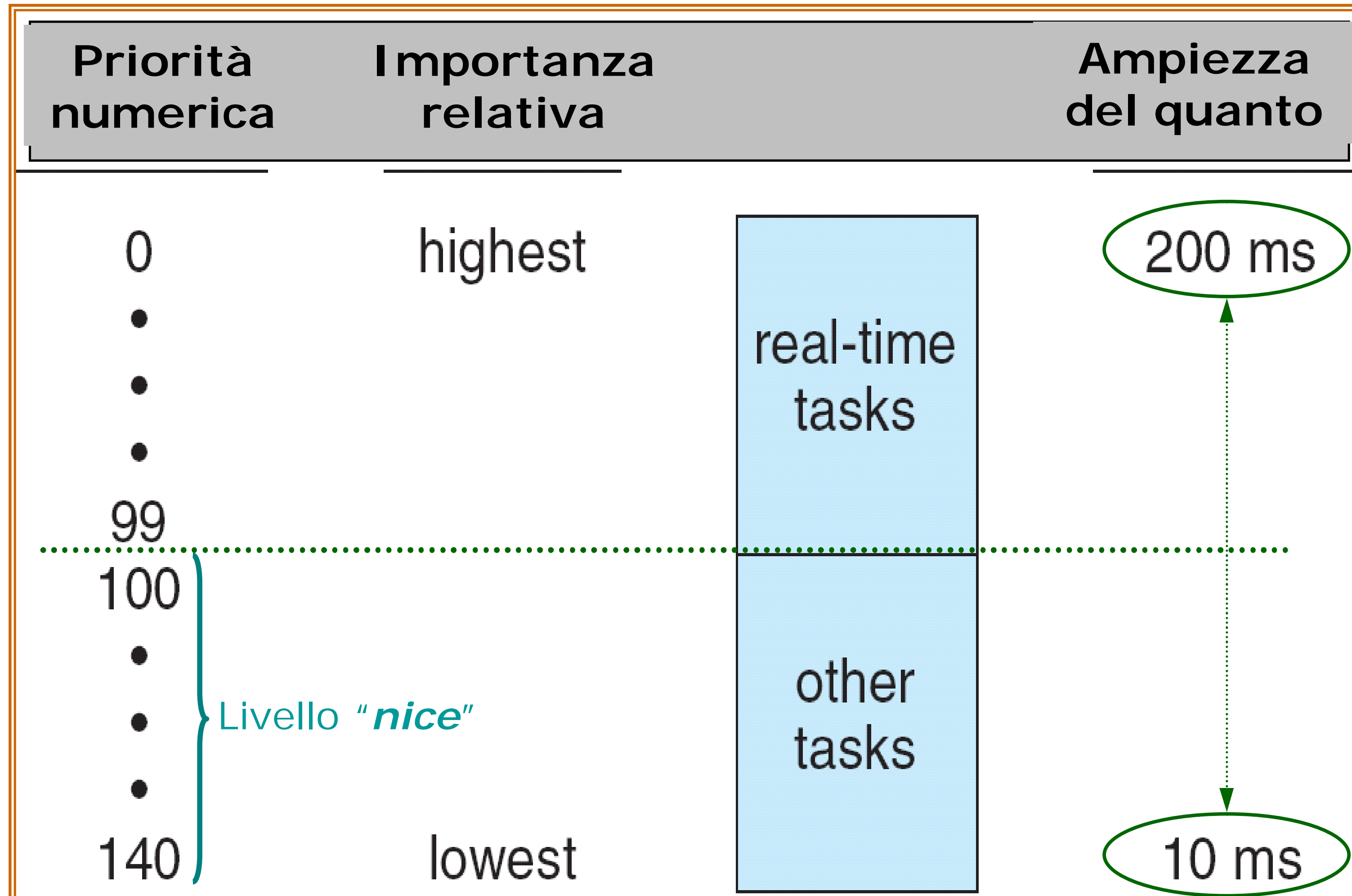
```
pid = clone(function, stack_ptr, ctrl, arg);
```

  - `function` = programma da eseguire nel nuovo "*task*" (processo o *thread*) con argomento `par`
  - `Stack_ptr` = 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
    - Es. Solo copia o stesso address space?

Ordinamento dei processi (GNU/Linux) – 1

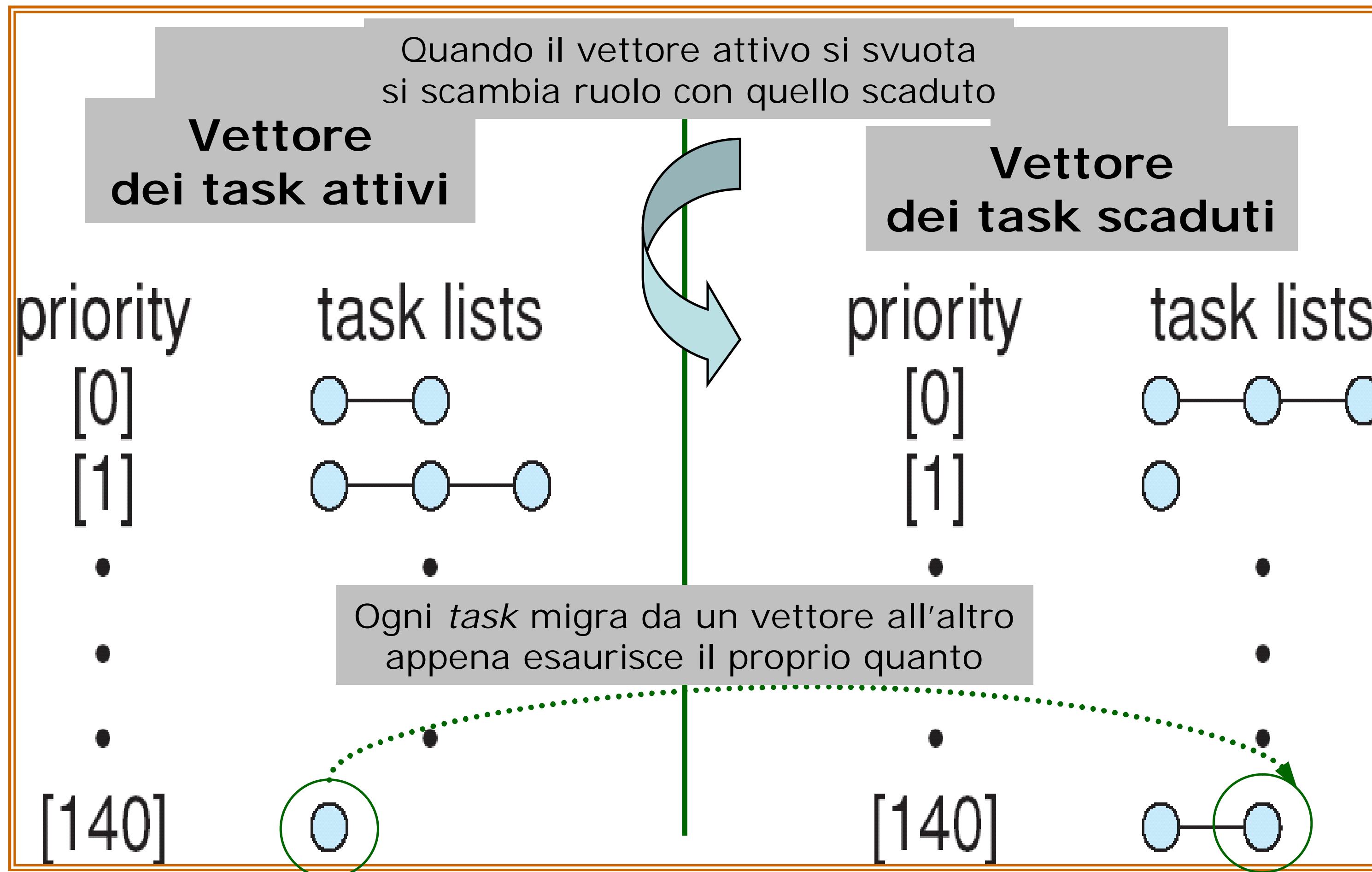
- I *thread* sono gestiti direttamente dal nucleo
  - Ordinamento per *task* (*thread* o processo indistintamente)
  - Selezione distinta tra classi distinte
  - Prerilascio per fine quanto o per attesa di evento
- 3 classi di priorità di *task*
  - **Tempo reale con politica FCFS a priorità senza prerilascio**
    - 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 (*Timesharing*)**
    - 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

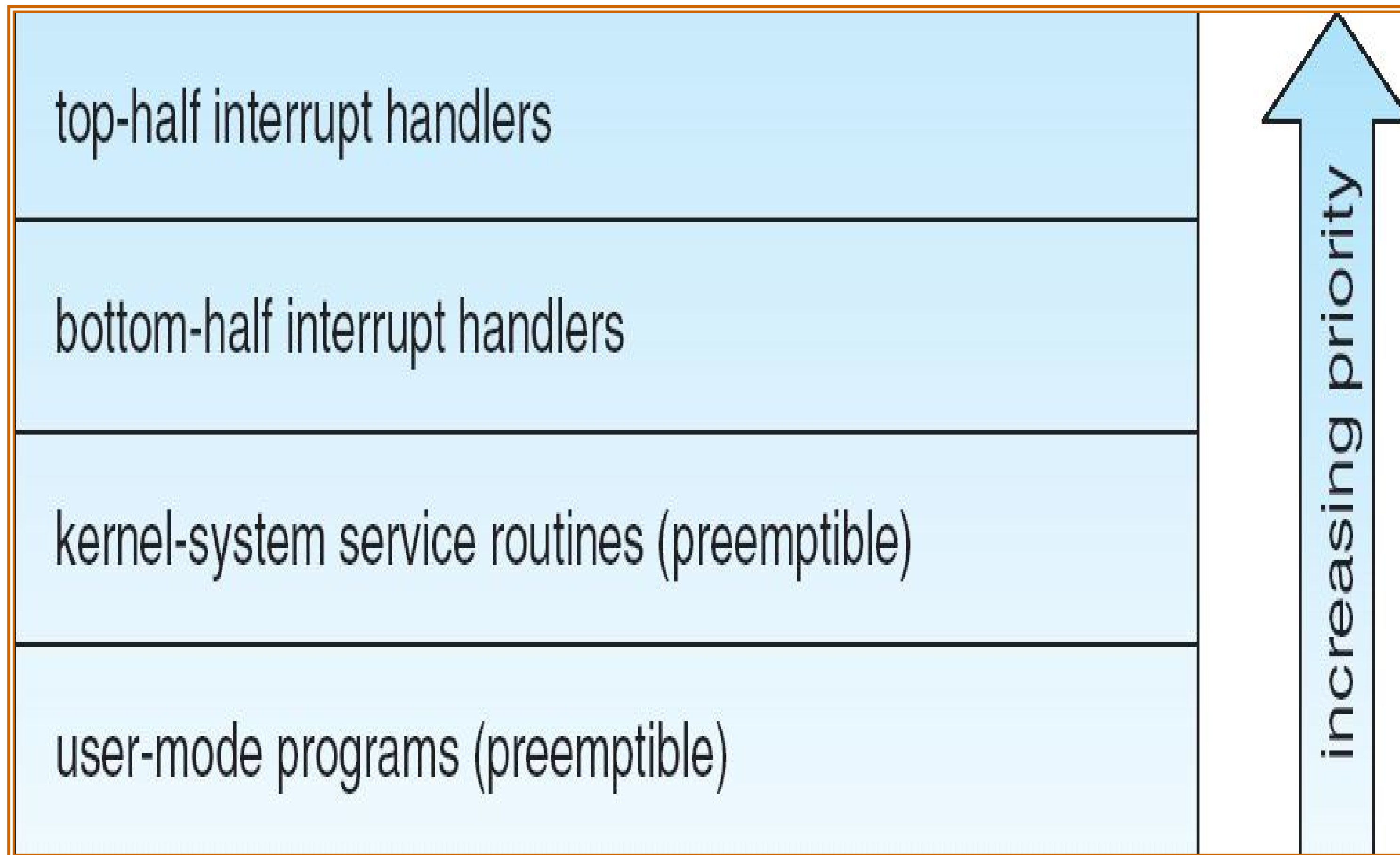


Silberschatz, Galvin and Gagne ©2005

Ordinamento dei processi (GNU/Linux) – 4

- Per versione < 2.6 l'attività dei processi in modo nucleo **non** ammetteva prerilascio
  - Ma questo naturalmente causava pesanti 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 (*top half*) dei gestori disabilita
    - La parte differita (*bottom half*) non disabilita ma il completamento della sua esecuzione viene preferito a ogni altra
      - Tranne che di altre parti immediate

Ordinamento dei processi (GNU/Linux) – 5



Silberschatz, Galvin and Gagne ©2005

Inizializzazione (GNU/Linux) – 1

- BIOS carica l'MBR (*Master Boot Record*) da primo settore su disco di boot 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
  - 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 (criptate in `/etc/passwd`)
  - Infine **`exec`**`("`**`shell`**`")`

Inizializzazione (GNU/Linux) – 3