Programmazione e Architetture (Modulo B)

Lezione 11
Scheduling

Luca Manzoni

Tipologie di scheduler

Per sistemi batch e interattivi

- Per sistemi batch
  - First-come first-served
  - Shortest job first
  - Shortest remaining time next
- Per sistemi interattivi
  - Round-robin
  - Scheduling con priorità
  - Shortest process next
  - Guaranteed scheduling
  - Lottery scheduling
  - Fair-share scheduling

First-Come, First-Served

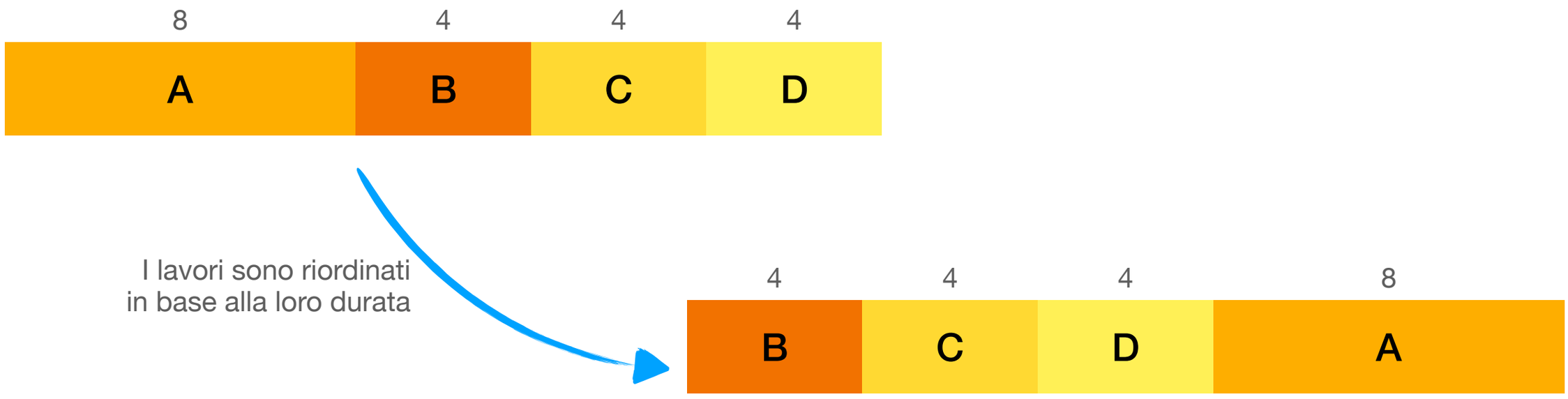
Esecuzione in ordine di arrivo

- I processi sono assegnati alla CPU nell'ordine di arrivo (quindi si ha una unica coda di processi)
- Quando un processo esegue una operazione bloccante viene eseguito il successivo nella coda
- Quando un processo passa diventa *ready* viene inserito in fondo alla coda, come se fosse un processo nuovo
- È molto semplice da implementare ma certamente non minimizza il tempo di attesa per il completamento dei lavori (e.g., pensate se il primo lavoro è lunghissimo e non ha operazioni bloccanti se non alla fine)

Shortest Job First

Esegue il lavoro che richiede meno tempo

- Supponiamo che i tempi necessari per completare ogni lavoro siano noti a priori
- Diventa possibile riordinare i lavori prima di eseguirli
- Minimizza il tempo di attesa medio per i risultati **se** tutti i lavori sono noti a priori



Shortest remaining time next

Esegue il lavoro che terminerà prima

- Una versione preemptive di **shortest job first** è **shortest remaining time next**.
- Quando un nuovo lavoro arriva il suo tempo di completamento è comparato col tempo rimanente degli altri processi
- Se il nuovo processo ha un tempo di completamento minore di quello attualmente in esecuzione il processo corrente è sospeso e l'esecuzione prosegue col nuovo processo arrivato

Round-Robin Scheduling

Condividere la CPU tra più processi

- Uno degli scheduler più semplici per sistemi interattivi
- A ogni processo viene allocato un intervallo temporale, detto **quanto** (*quantum*)
- Vi è una coda di processi e viene selezionato il primo della coda per eseguire per un quanto di tempo
- Al termine del quanto di tempo se il processo non è terminato (o non ha eseguito operazioni bloccanti) perde comunque l'utilizzo del processore e viene messo in fondo alla coda
- Dato che il context switch non è eseguito in tempo zero, diventa importante scegliere la dimensione del quanto: più è piccolo maggiori sono l'overhead e la reattività, più è grandi minori sono overhead e reattività

Round-Robin Scheduling

Condividere la CPU tra più processi

Scheduling con Priorità

Fornire diverse priorità a lavori diversi

- Non tutti i processi hanno la stessa priorità, vorremmo poter eseguire alcuni processi più spesso e altri meno spesso

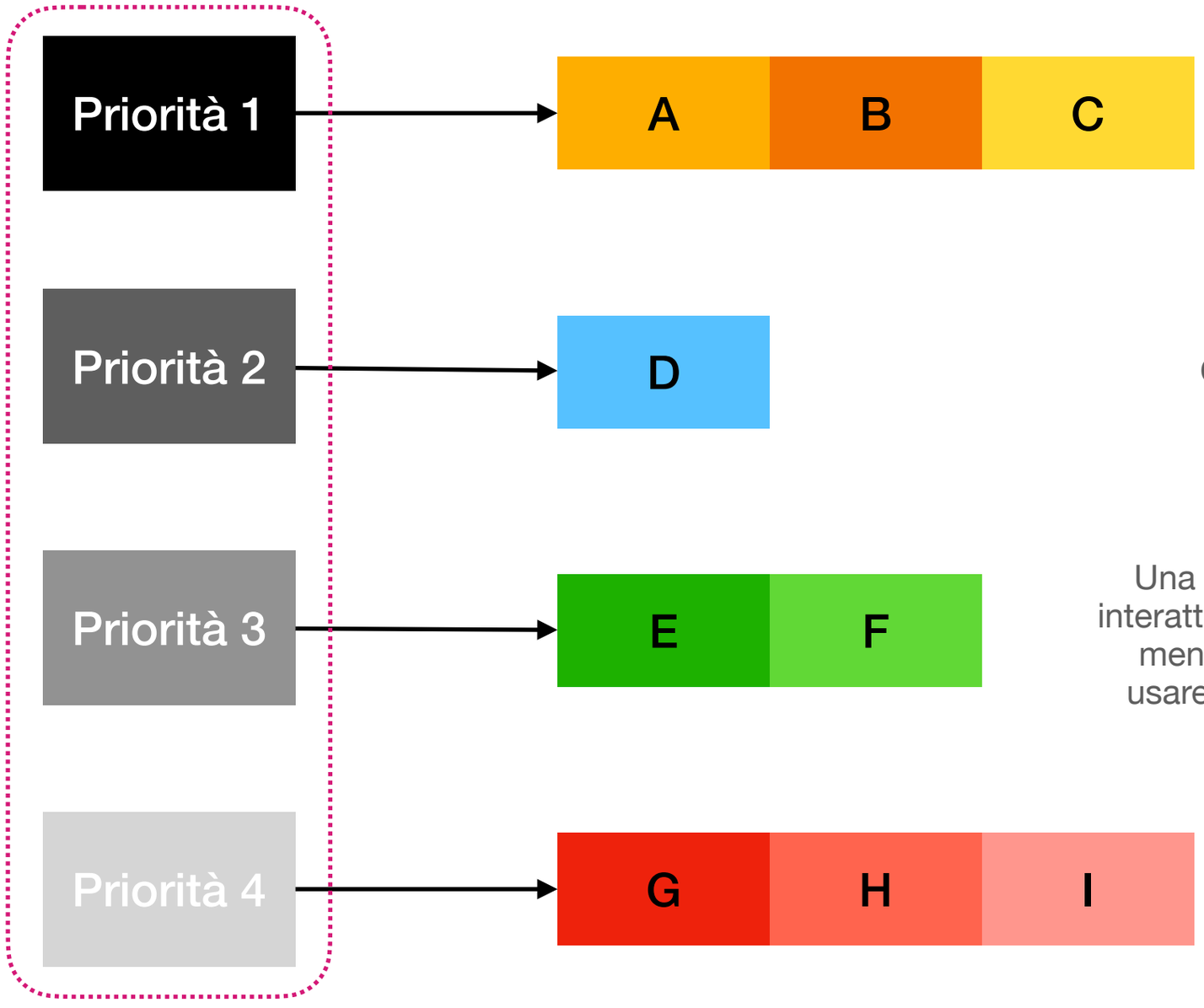
- Per garantire che ogni processo esegua (e non venga sempre “superato” da uno a priorità maggiore) ci sono diverse strategie:

  - Scelta di una policy per cui ogni priorità ottiene del tempo su CPU (questo consente di non cambiare le priorità)

  - Un processo che esegue ha la sua priorità diminuita. Quando non è più il processo a massima priorità si passa al successivo (che ora è quello a priorità massima). In questo caso le priorità cambiano nel tempo

Classi di priorità

Combinare priorità con altri scheduler



All’interno di ogni classe di priorità viene usato uno scheduler come round-robin

Classi di priorità diverse possono avere, per esempio, durata dei quanti diversa.

Una osservazione è che processi I/O bound interattivi beneficiano di tempi di risposta minori, mentre processi CPU bound beneficiano di usare la CPU per intervalli di tempo maggiori

I processi sono divisi in classi di priorità

Shortest Process Next

Shortest Job First per sistemi interattivi

- Nei sistemi batch “shortest job first” produce il miglior tempo di risposta
- Per trasferire l’idea a sistemi interattivi dovremmo sapere quale processo richiede tempo minore per terminare tra quelli ready
- Una possibilità è quella di stimare il tempo a partire da precedenti esecuzioni:
  - Inizieremo con una stima $T_S \leftarrow T_0$
  - Alla successiva esecuzione otteniamo un tempo T_1, aggiorniamo la nostra stima come $T_S \leftarrow \alpha T_0 + (1-\alpha)T_1$ con $\alpha \in [0,1]$
  - In generale prendiamo la nostra stima attuale e, ottenuto un nuovo tempo di esecuzione T_i aggiorniamo la stima come $T_S \leftarrow \alpha T_S + (1-\alpha)T_i$
  - Facile da implementare, specialmente con $\alpha = 0.5$

Guaranteed Scheduling

Garantire una certa quantità di risorse

- Possiamo voler garantire che ognuno di n processi riceva una frazione $1/n$ del tempo del processore
- Possiamo calcolare il rapporto $\frac{\text{tempo CPU consumato}}{\text{tempo CPU a cui si ha diritto}}$
- Se il rapporto è < 1 allora dobbiamo dedicare più tempo a quel processo
- Se il rapporto è > 1 allora allora dobbiamo dedicare meno tempo a quel processo
- Possiamo eseguire il processo col rapporto minore (e cambiarlo con un altro processo quando non è più tale)

Lottery Scheduling

Una semplice implementazione per garantire risorse

- Implementare il guaranteed scheduling non è banale
- Una possibile soluzione è assegnare a ogni processo un certo numero di “biglietti”
- Per decidere il prossimo processo che potrà fare uso della CPU si estrae uno dei biglietti. Il processo che lo possiede avrà accesso alla CPU
- In questo caso la frazione *attesa* di uso della CPU dipende dal numero di biglietti di un processo rispetto al totale

Fair Share Scheduling

Dividere il processore tra più *utenti*

- Fino ad ora abbiamo visto come il tempo su CPU è diviso tra più processi
- In sistemi multi-utente possono esserci più utenti ognuno con un diverso numero di processi in esecuzione
- E.g., Utente 1 con 20 processi e Utente 2 con 2 processi. Non vogliamo che l'utente 1 riceva 10 volte più tempo su CPU dell'utente 2
- Per questo il fair share scheduling utilizza la conoscenza degli utenti a cui appartengono i processi per garantire equità tra gli *utenti,* non solo tra i processi

Creazione di processi con fork

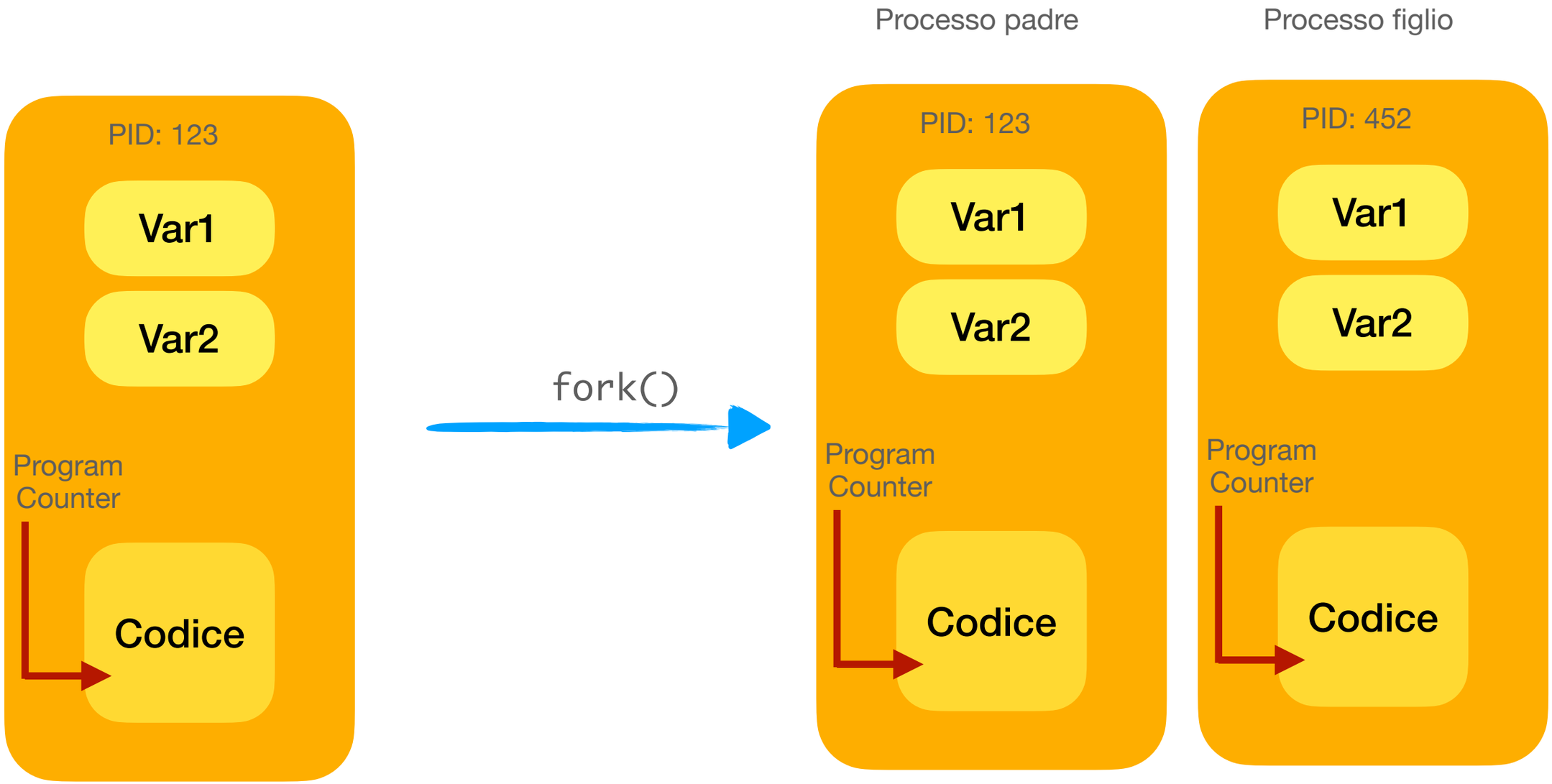
Processi in unix

E creazione di nuovi processi

- In unix ogni processo ha un identificativo detto PID - **P**rocess **ID**entifier
- Il primo processo che viene avviato ha solitamente PID 1
- È possibile creare altri processi tramite la chiamata di sistema “fork”
- Fork duplica un processo (e tutte le risorse associate):
  - Una copia manterrà il PID originario (questo è il processo padre)
  - Una copia avrà un nuovo PID (questo è il processo figlio)
  - Esclusa questa differenza le due copie sono identiche

Fork

Rappresentazione grafica



L’intero spazio degli indirizzi è duplicato. Questo significa che ogni processo ha una sua copia di tutta la memoria (ed un suo program counter indipendente)

Fork

E creazione di nuovi processi

- Per utilizzare fork in C è necessario includere unistd.h
- Questo rende disponibile la funzione fork()
- Il valore ritornato dalla chiamata a fork, di tipo pid_t è:
  - -1 se la chiamata è fallita
  - 0 nel processo figlio
  - Il pid del processo figlio nel processo padre
- Notate come fork ritorni **due** volte in due processi diversi con valori diversi

Wait

Interazione tra processi

- È possibile richiedere a un processo padre di attendere che un processo figlio termini
- Includendo sys/wait.h diventa disponibile la funzione wait
- “Wait” prende come argomento un puntatore a un intero
- “Wait” ritorna quando uno dei processi figli è terminato con valore di ritorno:
  - -1 se vi è stato un errore
  - Altrimenti è il PID del processo figlio che è terminato