Appunti Sistemi Operativi

Luca Seggiani
2025

1 Lezione del 23-09-25

1.1 Introduzione

Il corso di sistemi operativi riguarda l'ultima parte dello studio delle "architetture", che è partita con l'implementazione hardware in reti logiche, è continuata con lo studio del kernel in calcolatori elettronici, e termina appunto con lo studio dei sistemi operativi. Nello specifico, si considereranno sistemi operativi derivanti dalla famiglia **UNIX**.

Argomento del corso è la conoscenza di tecniche di programmazione usate nello sviluppo del sistema operativo **multiprogrammato** (più *processi* o più *thread*), con riferimento particolare alla programmazione **concorrente**, lo **scheduling** e la **memoria virtuale**.

Il corso mira a dare informazioni generiche utili allo studio di qualsiasi sistema operativo (anche non direttamente derivante da UNIX), in primis rivolte alla compresione di *come mai* una certa soluzione ad un problema è migliore di altre, e quali sono le tecniche che ci permettono di sviluppare soluzioni miglori.

1.1.1 Sistemi embedded e in tempo reale

Ci interesseremo anche ai sistemi **embedded** e sopratutto sistemi in **tempo reale**. Questi rappresentano sistemi *special-purpose* (per distinguere dai sistemi a scopo generale, *general-purpose*), dove dobbiamo rispettare coi nostri algoritmi di scheduling date **scadenze** temporali.

1.1.2 Programmazione concorrente

Con programmazione concorrente ci riferiamo alle tecniche che ci permettono di gestire più processi che si contendono le solite risorse, adottando politiche più o meno *eque* per i processi, o magari privilegiandole alcune. Obiettivo fondamentale sarà comunque quello di evitare *stalli* o **deadlock** dati da risorse occupate.

1.1.3 Programma del corso

Il corso è strutturato negli argomenti:

- **Concetti introduttivi** su sistemi operativi, architetture hardware e relativi cenni storici, in particolare ci interessano dettagli come la gestione della *pila* e le *interruzioni*;
- **Processi** e la loro gestione, inclusi gli algoritmi di *scheduling preemptive* e *non preemptive*, *prioritari* e *non prioritari* (FCFS, SJF, SRTF, RR). Inoltre si tratta la schedulazione nei sistemi **hard real-time** (RM, RDF);
- **Sincronizzazione** dei processi, quindi *programmazione concorrente*, *competizone* su risorse, e *scambio di informazioni fra processi* (IPC);
- Gestione della **memoria**, quindi *memoria virtuale*, *segmentazione* e *paginazione*;
- Gestione dei **dispositivi** di I/O, cioè i *driver*;

- **File system** su disco, cioè i componenti software che permettono la gestione di strutture di *file*, nella loro struttura sia *logica* che *fisica* di implementazione nei driver e nel sistema operativo;
- **Sicurezza**, quindi meccanismi di *protezione* fra processi, controlli sugli *accessi* sia in memoria che al filesystem, con riferimento al modello della *matrice degli accessi*.

1.1.4 Meccanismi e politiche

Una prima distinzione che possiamo fare è quella fra **meccanismo** e **politica**.

- Si dice **meccanismo** ciò che effettivamente implementato, in maniera sufficientemente veloce e compatta, nel kernel, per fornire il cosidetto *supporto architetturale* a risorse, dispositivi, ecc...
- Nei sistemi operativi ci interessano invece principalmente le **politiche**, cioè decisioni (che vanno poi implementate) su come gestire *a priori* date risorse, dispositivi, ecc...

1.1.5 Sistemi operativi

Un **sistema operativo** è in primo luogo un *programma software* che ha il compito di fare da intermediario fra l'*utente* e l'*hardware* di un calcolatore.

Far fronte ai bisogni dell'utente significa gestire e consentire l'accesso delle risorse ai *processi* di cui l'utente necessita. In questo individuiamo come obiettivi del sistema operativo:

- Eseguire i *programmi utente*;
- Rendere il sistema facile da usare;
- Utilizzare l'hardware in maniera efficiente.

1.1.6 Programmi

I programmi con cui abbiamo a che fare sono per noi *liste di istruzioni* (tralasciando il fatto che queste siano codificate in linguaggio macchina o in un suo linguaggio mnemonico), ordinate ma che possono presentare salti condizionali che cambiano il normale *flusso di esecuzione*.

Il **comportamento** e quindi i **risultati** di un programma dipendono sì dal codice, ma anche dai **dati** in ingresso allo stesso. In questo possiamo dire che il programma non esiste mai da solo ed è solo la parte **statica** di un processo.

1.1.7 Risorse

Iniziamo quindi a vedere quelle che sono le risorse che dobbiamo fornire ai programmi. Il modello che adottiamo è il più diffuso oggigiorno, cioè quello di *Von Neumann*. Questo modello comprende, a grandi linee:

- La **CPU** o *processore*, che ha il solo scopo di prelevare ed eseguire le istruzioni in maniera sequenziale, alterando il suo flusso come già detto solo nel caso di istruzioni condizionali, o come vedremo nel caso di interruzioni o altre situazioni simili;

- La **memoria principale**, che nell'architettura di Von Neumann contiene sia i dati che il programma in esecuzione, e che deve essere capace di fornire su richiesta alla CPU.

  Ricordiamo che questa è spesso *volatile*, cioè i suoi contenuti vengono sostanzialmente invalidati allo spengimento della macchina. Potremmo interrogarci sul motivo di tale decisione: principalmente diciamo che le ragioni sono economiche, in quanto mantenere l'informazione per lunghi periodi di tempi è solitamente più costosto e delegato a dispositivi (come i dischi) che offrono risparmi in cambio di grandi tempo di accesso (inusuali sulla memoria principale);

- Altre **risorse** che si aggiungono a CPU e memoria, comunque indispensabili per eseguire qualsiasi istruzione. Queste sono:

  – I **dispositivi**, che comprendono ad esempio la memoria di *archiviazione* (il **disco**) e le *periferiche* di interfaccia con l'utente;
  – Le risorse **logiche**, cioè determinate strutture dati in memoria che devono essere fornite in maniera più o meno esclusiva ai processi. Anche gli stessi *file* del *file system* sono risorse logiche.

Risulta chiaro come la gestione delle risorse hardware e logiche è fondamentale anche alla **portabilità** dei programmi, che magari vogliono avere accesso a funzionalità simili su più sistemi operativi (accesso alla tastiera, ai file, ecc...), senza dover necessariamente conoscere l'implementazione interna di tali sistemi operativi.

Abbiamo quindi l'obiettivo di implementare tutte quelle **interfacce** di cui il programma bisogna per presentare all'utente le sue funzioni. Questo include le interfacce grafiche, audio, ecc... per la realizzazione di ambienti visuali e interattivi nei sistemi moderni.

Dal nostro lato, quello del *sistema*, vorremo che le soluzioni tecniche che adottiamo non impattino in maniera negativa le prestazioni o comunque il funzionamento dei programmi che mandiamo in esecuzione.

1.1.8 Struttura stratificata del S/O

La struttura di un sistema operativa può dividersi in più livelli, fra cui:

- Il livello **hardware**, fornito come già detto da risorse come:

  – La **CPU**;
  – La **memoria principale**;
  – Le **periferiche**, fra cui *video*, *disco*, *interfacce di rete*, ecc...

  Il livello hardware offre la cosiddettà *intefaccia hardware*, data dalle specifiche secono cui interagiamo con i dispositivi hardware stessi;

- Il livello **sistema operativo** (o *S/O*), che implementa la gestione delle risorse che studieremo nel corso, e offre a sua volta altre risorse logiche. In particolare notiamo:

  – Gestione della **CPU**;
  – Gestione della **memoria**;
  – Gestione del **file system** e quindi dei *file*;

  - Gestione dei **dispositivi** attraverso i *driver*.

  Questo livello offre la sua interfaccia attraverso le **chiamate di sistema** o *primitive*, che implementano una certa **API** (*Application Programming Interface*) secondo le quali i programmi utente delegano all'S/O operazioni che non potrebbero normalmente portare avanti da soli (accesso a risorse, schedulazioni temporali, ecc...);

- Il livello delle **applicazioni**, che comprende i programmi utente.

Questa gerarchia implica chiaramente che ogni livello non conosce nulla riguardo al livello successivo, ma si preoccupa solo di fornire un'*interfaccia* conforme alle specifiche. A questo punto è compito del livello successivo stesso rispettare l'interfaccia e farne uso per i suoi scopi.

Il programmatore di **sistema** interagisce con i livelli *hardware* e *S/O*, mentre il programmatore di **applicazioni** interagisce con i livelli *S/O* e *applicazioni*.

Compito dell'*API* è quello di generare per i programmatori di applicazioni una macchina *astratta* più semplice da usare, più efficiente e più sicura (cioè realizzare gli obiettivi che ci eravamo posti in 1.1.5). Ricordiamo che per noi sicurezza significa *modelli* che controllano l'accesso da parte dei processi (altresì **soggetti**) alla memoria, e più in generale a tutte le risorse sistema (altresì **oggetti** dei programmi).

1.1.9 Definizione di S/O

Iniziamo a definire più nei dettagli cos'è un S/O.

- Un S/O è un **allocatore di risorse**, cioè gestisce *tutte* le risorse, e decide tra richieste conflittuali di accesso a tali risorse (inviate dai vari processi) al fine di garantirne un uso equo ed efficiente.
- Un S/O è però anche un **programma di controllo**, che controlla l'esecuzione dei programmi e lo stato dele risorse per prevenire usi impropri e stati inconsistenti.

Ricordiamo che in ogni caso l'unico programma effettivamente in esecuzione in ogni momento sulla macchina reale è il **kernel**, cioè nucleo, mentre il controllo viene temporaneamente passato fra programmi utente.

2 Lezione del 24-09-25

2.1 Cenni storici

Le prime macchine calcolatrici "moderne" nascono durante la seconda guerra mondiale, principalmente per scopi crittografici.

Fu nel periodo del secondo dopoguerra che diverse industrie, principalmente dal settore delle macchine da scrivere e di apparecchiature simili, decisero di sviluppare queste tecnologie per scopi di ricerca e commerciali.

Di pari passo diverse università iniziarono a loro volta a sviluppare architetture e macchine calcolatrici, in questo caso a puro scopo di ricerca. Un esempio locale è quello della **CEP** (*Calcolatrice Elettronica Pisana*), sviluppata dai dipartimenti di matematica e fisica di Pisa (sotto indicazione di Enrico Fermi) per aiutare i ricercatori nei loro calcoli.

Sempre a Pisa fu l'ingegnere Mario Tchou a lanciare, in collaborazione con Olivetti, il progetto che diventò nel 1959 l'**Elea 9003**, fra i primi calcolatori a transistor commerciali (di contro la CEP funzionava a valvole termoioniche).

2.1.1 Sistemi Batch

In queste prime macchine, anche se la possibilità della multiprogrammazione era disponibile, raramente si parlava di "sistemi operativi" veri e propri. I primi sistemi operativi nascono quindi per i mainframe degli anni '60, fra cui notiamo gli **IBM Sistema 360** (e i successivi Sistema 370).

Inizialmente, queste macchine venivano usate in modalità **batch** (più programmi di più utenti eseguiti in sequenza): i primi S/O nascono appunto per permettere l'uso simultaneo (*time-sharing*) della macchina da parte di più utenti.

In ogni caso, già nei primi sistemi batch monoprogrammati si necessitava di diversi componenti effettivamente assimilabili ad un rudimentale sistema operativo:

- Un sistema di programmazione in memoria di massa (all'epoca nastri magnetici);
- Una *Job Control Language* (**JCL**), che esprimeva direttive interpretate da un *Monitor* (antenato delle moderne *shell*);
- Un **BIOS** (*Basic Input Output System*), cioè un insieme di routine per l'interazione con le periferiche.

L'S/O era quindi composto da Monitor + BIOS, che poteva essere configurato per caricare programmi e mandarli in esecuzione. In ogni momento in memoria si trovavano comunque il S/O e al più un programma utente.

2.1.2 Sistemi di spooling

Il prossimo passo è quello dei sistemi di **spooling** (*Simultaneous Peripheral Operation On-Line*). Questi nascono per permettere al programma utente di restare in esecuzione mentre le periferiche (all'epoca molto lente) completano le loro operazioni, bufferrizzando quindi le operazioni di ingresso/uscita.

I sistemi operativi che implementavano lo spooling dovevano quindi arricchirsi per permettere questo tipo di funzionalità.

2.1.3 Sistemi multiprogrammati

Arriviamo quindi ai sistemi **multiprogrammati**, cioè che permettono la gestione contemporanea di più programmi nella memoria principale: per la prima volta oltre al sistema operativo possiamo caricare in memoria più di un singolo programma utente.

I sistemi operativi di questo tipo si dovranno quindi dotare di diverse funzionalità, fra cui *scheduling* dei processi, possibilità di fare **DMA** (*Direct Memory Access*) sulle periferiche, *preemption* dei programmi in esecuzione, *memoria virtuale* per permettere mappature in memoria localmente costanti per ogni programma, ecc...

2.1.4 Sistemi time-sharing

Lo sviluppo di sistemi di tipo multiprogammato è stato favorito dal fatto che i programmi utente che venivano sviluppati erano sempre più *interattivi*, quindi caratterizzati da fasi temporali distinte:

- **CPU-Burst**, dove il processore lavorava effettivamente sui dati;
- **I/O-Burst**, dove il processore attendeva operazioni I/O dalle periferiche, magari fornendosi del DMA.

Ci spostiamo quindi da un paradigma di esecuzione *sequenziale* ad un paradigma *multi-tasking*, dove il sistema operativo assegna ciclicamente istanti temporali (*quantum*) ai processi in esecuzione.

Il vantaggio dell'esecuzione multitasking è di poter avvicinare fra di loro i CPU-Burst, spostando il controllo della CPU da un processo all'altro quando si incorre in un I/O-Burst.

Per quanto ci riguarda, quindi, la tecnica del **time-sharing** non è che un modo per implementare il *multi-tasking*, cioè un caso particolare della *multiprogrammazione*, caratterizzato da processi in memoria che vengono eseguiti (o almeno hanno l'illusione di essere eseguiti) contemporaneamente. Ricordiamo che l'esistenza di più processi in memoria era di per sé caratteristica del sistema multiprogrammato.

L'idea di sviluppare diversi e sofisticati algoritmi di *scheduling* viene proprio dalla necessità di dover mantenere la CPU in piena attività, cioé eseguire più CPU-Burst possibile, scegliendo in maniera intelligente quali processi mandare in esecuzione (equivalentemente, a quali processi assegnare i quantum temporali).

Notiamo che il tempo che la CPU passa a realizzare lo scheduling e i cambi di contesto rappresenta effettivamente **overhead** per il sistema, cioè tempo non passato ad eseguire programmi utente, ma in qualche modo "sprecato" in altri modi. Questo overhead è giustificato solo nel caso in cui le virtualizzazioni che consente permettono una velocizzazione considerevole della macchina.

2.1.5 Sistemi in tempo reale

La storia dei sistemi operativi ha un'interessante tangente nei cosiddetti sistemi **real-time** (*in tempo reale*). Questi sono sistemi dove lo scheduling è *deterministico* e il tempo impiegato ad eseguire un dato processo può quindi essere stabilito prima che questo venga lanciato.

Sistemi di questo tipo sono utili nel caso di calcolatori che interagiscono con *ambienti operativi* reali attraverso **sensori** ed **attuatori**, dove la precisione temporale con cui vengono eseguite certe operazioni è effettivamente importante alla funzione della macchina.

In particolare notiamo due paradigmi possibili per i sistemi real-time:

- **Soft** real-time, che non assicurano ma si impegnano a mantenere le specifiche sopra descritte;
- **Hard** real-time, il cui funzionamento ha come priorità imprescindibile le specifiche sopra descritte.

3 Lezione del 25-09-25

3.1 Richiami architetturali

Riprendiamo alcuni aspetti architetturali di un sistema di elaborazione. L'architettura che consideriamo è quella di *Von Neumann*, modello ancora oggi in uso e composto da:

- La **CPU** (*Central Processing Unit*) o come abbiamo già detto *processore*. Rappresenta un circuito piuttosto complesso che ha però l'unica funzione di *esecutore di istruzioni*.

  Le istruzioni che questa esegue possono essere di tipo **CISC** (*Complex Instruction Set*), come ad esempio nell'architettura x86, o di tipo **RISC** (*Reduced Instruction Set*),

come ad esempio nell'architettura ARM. Ricordiamo comunque che nelle moderne implementazioni dell'x86 si traduce comunque in un instruction set RISC a livello architetturale per questioni di ottimizzazione.

Si può infatti dire che è inutile avere molte e complesse istruzioni (CISC) che richiedono molti cicli di clock, quandi si possono avere poche e semplici istruzioni (RISC) che ne richiedono pochi: eventuali istruzioni più complesse potranno essere implementate come *subroutine* che usano più istruzioni semplici.

Ricordiamo quindi che la CPU si limita ad eseguire istruzioni, e non conosce (non memorizza) il programma. La poca memoria che ha a disposizione (sotto forma di *registri*) viene usata per mantenere i dati che sta elaborando;

- La **RAM** o *memoria centrale*, o ancora come abbiamo visto *memoria principale*. Questa ha il compito di memorizzare *dati* e *programma* (questo il fulcro dell'architettura di Von Neumann) e di renderli disponibili alla CPU e, come vedremo, anche ad altri dispositivi.

  Abbiamo visto che è una memoria *volatile*, quindi che si mantiene solo finché il calcolatore è acceso, e che è una memoria ad *accesso diretto*, cioè si può accedere a qualsiasi locazione in tempo costante (a differenza di memorie di tipo *sequenziale*, ecc...).

  Le operazioni che possiamo svolgere sulla memoria sono *letture* e *scritture* su locazioni di memoria. Nelle memorie moderne le letture sono *non distruttive*, mentre le scritture (chiaramente) lo sono.

- Qualche tipo di complesso di **I/O**. Questo comprende periferiche come *tastiera*, *porte seriali/parallele*, *interfacce di rete*, ecc...

  Un dispositivo particolare che si trova nello spazio di I/O è il **disco** o *memoria secondaria*, a differenza della principale *persistente*, e usata per l'archiviazione di dati a lungo termine. Chiaramente, il tradeoff in questo caso è in termini di tempo (i dischi, anche allo stato solido, sono molto più lenti in tempo di accesso della RAM).

- Un **bus**, o *rete di interconnessione*, che permette a questi componenti di comunicare fra di loro.

  Questa comunicazione dovrà essere **bidirezionale**, in quanto ad esempio la CPU deve sia leggere che scrivere dalla RAM: abbiamo visto come bus di questo tipo possono essere implementati sfruttando la logica a 3 stati.

  Sperabilmente un bus dovrà contenere un numero consistente di linee. Torniamo all'esempio della CPU che legge in memoria: avremo bisogno di specificare l'*indirizzo* della locazione che vogliamo leggere, e vorremo vederci tornare una o più *parole* (cioè i dati che ci interessano) dalla memoria. Il modo più veloce per effettuare questa operazione è fornirsi di abbastanza linee per specificare sia gli indirizzi che i dati in **parallelo**: un bus *seriale* si dimostrerebbe molto più lento.

  A livello logico dobbiamo dire anche che c'è bisogno di un **protocollo**, o comunque una qualche *politica* di gestione del bus.

  - Ad esempio, la politica più semplice è quella dove la CPU è l'unica che può iniziare una transazione sul bus: questa è la classica configurazione *master-slave* dove la CPU rappresenta il *master* e memoria e I/O rappresentano gli *slave*;

  - Esistono però situazioni dove potremmo volere che i dispositivi (ad esempio il disco) scrivano in memoria, o viceversa sia la memoria a scrivere sui dispositivi. Questo è effettivamente il caso del *DMA*. Avere un bus che supporta più iniziatori di transazioni richiede necessariamente un protocollo che stabilisca chiaramente chi può iniziare in quale momento una data transazione.

  Le transazioni avvengono chiaramente in fasi, di cui ne individuiamo almeno 3 nel caso più semplice (singolo master, più slave):

  1. Una prima fase di richiesta della transazione da parte dell'*iniziatore*;
  2. Una fase di attesa da parte dell'iniziatore del responso dell'*obiettivo*;
  3. Una fase dove l'operazione viene effettivamente eseguita, in un determinato lasso di tempo.

  Nel caso di più master, abbiamo bisogno di meccanismi più sofisticati che implementino **mutua esclusione** e **sincronizzazione** delle risorse a cui i più iniziatori potrebbero voler accedere. Questo è vero sia a livello *logico* (su risorse logiche o comunque gestite dal S/O) che *elettrico* (2 o più componenti non pilotino mai le stesse linee contemporaneamente, pena fili bruciati).

Facciamo quindi una considerazione su come organizzare lo spazio di memoria e lo spazio dedicato ai registri delle periferiche. Esistono due configurazioni principali:

- **Memory-mapped I/O**: disponiamo i registri di I/O direttamente nello spazio di memoria, usando gli stessi indirizzi per indirizzare sia la memoria che i dispositivi;
- **Port-mapped I/O**: sfruttiamo due spazi, lo *spazio di memoria* e lo *spazio di I/O*, che mantengono separati i due tipi di informazione. Questo può essere fatto agilmente includendo un bit di selezione di spazio nel bus, ed è la soluzione adottata dall'architettura x86.

3.2 CPU

Vediamo nel dettaglio il primo componente, cioè la CPU.

3.2.1 Cicli CPU

Il funzionamento della CPU avviene in maniera **ciclica**: cogliamo più fasi che si ripetono nel tempo da quando questa viene accesa (reset) fino a quando viene spenta.

1. **Prelievo** o *fetch*: si legge la prossima istruzione in memoria, puntata dall'**IP** o **PC** (*Instruction Pointer* o *Program Counter*), e la si porta in un qualche registro interno al processore, pronta ad essere eseguita;
2. **Decodifica** o *decode*: si interpreta il significato dell'istruzione, cioè si individua qual'è effettivamente l'istruzione che dobbiamo eseguire, e si portano all'interno di registri gli eventuali *operandi sorgente* o gli indirizzi degli *operandi destinazione*;
3. **Esecuzione** o *execute*: si esegue effettivamente l'istruzione, direttamente attraverso la rete di controllo della CPU o sfruttando una o più **ALU** (*Arithmetic and Logic Unit*).

   Successivamente, il risultato viene (se necessario) riscritto in memoria attraverso un'operazione di *write-back*. Questa fase viene a volte considerata come a sé stante (ad esempio nelle pipeline delle architetture RISC).

3.2.2 Registri CPU

La CPU è dotata di una sua memoria interna formata da locazioni di memoria dette **registri**. Questi si dividono in registri **generali**, riservati alle elaborazioni, e **di stato**, riservati a compiti speciali.

Registri generali

Consideriamo un set estremamente generico di registri:

- **AX**, **BX**, **CX** e **DX** sono i classici registri programmatore a uso generale;
- **ESP** è utilizzato per indirizzare la **pila** o **stack**, ovvero una parte di memoria con disciplina LIFO che serve a gestire sottoprogrammi.

Registri di stato

Ricordiamo due registri di stato:

- L'**IP** o **PC** (*instruction pointer* o *program counter*). Viene usato per contenere l'indirizzo della locazione dalla quale sarà prelevata la prossima istruzione da eseguire. Il contenuto dell'EIP è fissato al reset iniziale, e impostato sulla prima istruzione da eseguire.
- L'**F** (registro dei *flag*). Consiste di una serie di elementi binari detti **flag**, fra cui ricordiamo:
  - **OF:** flag di overflow (traboccamento) delle operazioni aritmetiche, si imposta se l'ultima operazioni, presi gli operandi come interi, ha prodotto un risultato non rappresentabile su n bit;
  - **SF:** flag di segno, impostato quando l'ultima operazione restituisce un complemento a 2 con MSB $= 1$ (ergo negativo);
  - **ZF:** flag zero, che viene impostato quando l'ultima operazione restituisce qualcosa di nullo;
  - **CF:** flag di carry (riporto), che viene impostato quando l'ultima operazione richiede un riporto o un prestito, ergo presi gli operandi come naturali il risultato non è rappresentabile su n bit;
  - **IF:** flag di interruzioni attivate, quando è attivo il processore risponde alle interruzioni (che approfondiremo in seguito).

  Al reset i flag visti finora sono impostati a 0.

3.2.3 Instruction set

Consideriamo un set di istruzioni estremamente basilare. Innanzitutto, possiamo dividere le istruzioni in **operative** e **di controllo**. Possiamo quindi fare ulteriori suddivisioni al'interno di queste categorie:

- **Operative:**
  - Di trasferimento;
  - Aritmetiche;
  - Di traslazione/rotazione:

  - Logiche.

- **Di controllo:**
  - Di salto;
  - Di gestione di sottoprogrammi.

Queste categorie andranno quindi a definirsi nelle varie istruzioni `MOV`, `ADD`, ecc... a cui siamo abituati.

Una nota va fatta adesso sulla scomodità data dall'utilizzo di istruzioni CISC: queste si sono sviluppate storicamente secondo il pensiero che era meglio dare più strumenti possibile al programmatore, ma oggi che il codice macchina è quasi esclusivamente compilato la dimensione variabile degli opcode rende difficile tecniche di ottimizzazione come il pipelining.

In ogni caso, per informazioni più approfondite sulla struttura generale del processore considerato si rimanda ai testi specializzati o agli appunti in `https://raw.githubusercontent.com/seggiani-luca/appunti-rl/34228f66db395637bd1824d04f3130b977cc0ce4/master/master.pdf`.

3.3 RAM

Approfondiamo quindi il discorso della RAM. Nel sistema considerato inseriremo un elemento di **cache**, nello specifico fra la CPU e il bus (da cui si accede alla RAM). Il funzionamento della cache è dettagliato in `https://raw.githubusercontent.com/seggiani-luca/appunti-ce/638d3abf2e1d473632b575401582203c3b113c82/master/master.pdf`, e per quanto ci riguarda possiamo dire che funge da unità di "*memoizzazion*" dei dati (quando vengono richiesti), più veloce della RAM.

4 Lezione del 30-09-25

Continuiamo la discussione della memoria RAM.

Avevamo introdotto il meccanismo della cache come una sorta di "memoizzazione" dei dati in occasione del primo accesso. In verità, nei moderni processori (dal Pentium in poi) abbiamo due cache separate:

- La **I-cache**, cioè *cache istruzioni*;
- La **D-cache**, cioè *cache dati*;

Il vantaggio di distinguere fra cache istruzioni e cache dati è che la I-cache enon ha bisongo di essere ricopiata in memoria alla fine dell'utilizzo, e probabilmente deve mantenere zone di memoria molto specifiche, per cui ha senso non rallentarne l'operazione chiedendole di mantenere anche informazioni sui dati.

4.0.1 Gerarchie di memoria

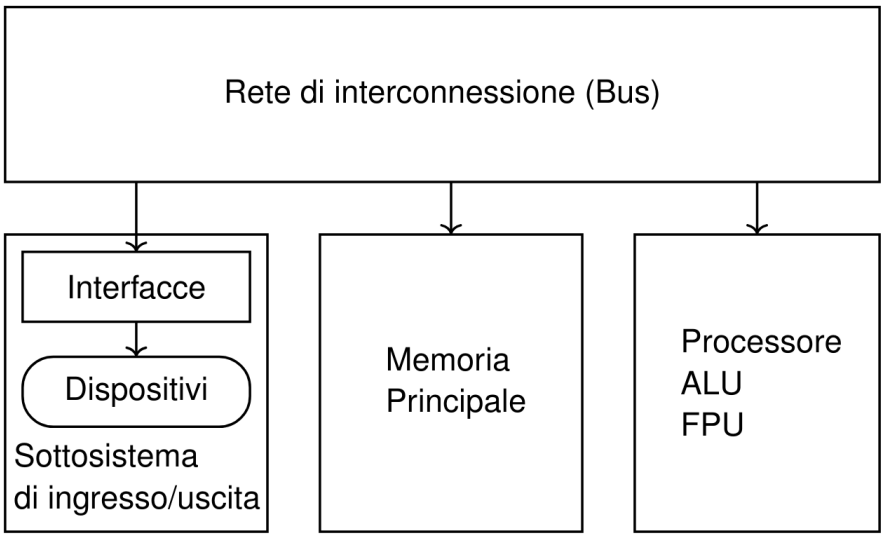
Fra registri, RAM, dispositivi a blocchi, ecc... abbiamo visto diverse fonti di *memoria* che un calcolatore può utilizzare. Potrebbe avere senso organizzare queste memorie in una struttura gerarchica, magari per *dimensione crescente* in avanti (e di conseguenza per [velocità] all'indietro):

1. Registri interni;

2. Cache;
3. RAM;
4. Dischi;
5. Nastri.

4.1 Schema a blocchi di un semplice calcolatore

Possiamo quindi, dopo aver visto tutte le componenti che lo compongono, vedere lo schema a blocchi di un semplice calcolatore:



Vediamo quindi come questi componenti comunicano fra di loro:

- La rete di interconnessione (bus) serve tutti (a scapito della direzione delle frecce, può supportare la comunicazione *da* e *a* componenti);
- Il processore e la RAM si trovano sul bus;
- I dispositivi, cioè i trasduttori col mondo esterno, comunicano con il sistema attraverso le loro *interfacce*, che obbedisce da un lato alle regole del bus e dall'altro alle specifiche del dispositivo stesso per permettere la comunicazione.

4.2 Interfacce

Abbiamo visto come fra il calcolatore ed ogni dispositivo si trovi un'apposita *interfaccia*.

Di base, ogni interfaccia è caratterizzata da più registri (accessibili nello spazio di I/O), che possono essere scritti o letti dal calcolatore per dare o ottenere informazioni dal dispositivo. Notiamo che letture e scritture sui registri delle interfacce possono essere distruttive: spesso il dispositivo implementa particolari funzioni che vengono lanciate da operazioni di questo tipo (un registro che si azzera dopo esser letto, ecc...).

Nel caso più semplice, in ogni caso, un'interfaccia dispone di almeno 3 registri:

- Registro di **stato**, che segnala lo stato corrente dell'interfaccia (se è di uscita può segnalare che è pronta a ricevere dati, se di entrata che ci sono dati pronti, ecc...);

- Registro di **controllo**, che permette al calcolatore di comandarne l'operazione (se di entrata può impedire che nuovi dati arrivino in ingresso, ecc...);
- Uno o più **buffer dati**, resi accessibili attraverso un registro di lettura. Solitamente si dice **TBR** (*Transfer Buffer Register*) il registro che accede al buffer di uscita e **RBR** (*Receive Buffer Register*) il registro che accede al buffer di entrata. Nel caso di interfacce di ingresso/uscita TBR e RBR stanno alla stessa porta dello spazio di I/O, e quale viene reso disponibile al processore varia in base al tipo di operazione che esso richiede (TBR per uscita, RBR per ingresso).

4.3 Interruzioni

Veniamo quindi al meccanismo dell'**interruzione**. Nella formulazione originale di Dijkstra queste servivano a risparmiare al processore l'attesa "attiva" (*busy wait*) dei bit di stato delle interfacce, delegando questo invece ad una segnalazione esplicita da parte dell'interfaccia, che viene *gestita* dal processore mettendo in esecuzione un determinato *handler* di interruzione.

Per gestire correttamente le interruzioni abbiamo bisogno di un po' di infrastruttura in più:

- Una nuova fase processore, successiva all'esecuzione, che si occupa di controllare le richieste di interruzioni in arrivo (nei sistemi x86 *la* richiesta, che è stata inoltrata da un sottosistema detto APIC);
- Una zona di memoria dove viene allocata la **IVT** (*Interrupt Vector Table*), che associa ad un indice (cioè il tipo di interruzione) l'inizio dell'handler relativo a tale tipo;
- Una nuova istruzione, `IRET`, che si occupa di ritornare da un gestore di interruzione.

  Non potremmo usare la semplice RET in quanto ogni interruzione salva dello stato aggiuntivo oltre al semplice IP sulla pila: di base, salveremo anche il registro dei FLAG.

4.3.1 Tipi di interruzione

Abbiamo visto nel corso di calcolatori elettronici come il meccanismo delle interruzioni può essere sfruttato per implementare molta più funzionalità di quelle relative alla gestione dei dispositivi. In particolare, i calcolatori moderni dispongono di più tipi di interruzioni:

- Interruzioni **esterne**, del tipo che abbiamo appena visto, che si distinguono ulteriormente in:
  - Interruzioni esterne **mascherabili**, cioè che possono essere ignorate variando il flag IF;
  - Interruzioni esterne **non mascherabili**, cioè che vengono sempre gestite;
- Interruzioni **interne**, cioè lanciate da situazioni interne al processore (eccezioni);
- Interruzioni **software**, che possono essere lanciate dal programmatore attraverso l'apposita istruzione `INT`.

4.4 Meccanismi di protezione

Veniamo quindi ai meccanismi tipici del S/O in sé per sé. Se vogliamo la separazione fra processi e S/O che gestisce quei processi (e quindi ha l'accesso prioritario alle risorse di sistema), dobbiamo separare l'operazione del calcolatore in due modalità principali:

- Modalità **utente**: usata per la normale esecuzione dei programmi, non è possibile accedere a tutte le risorse di sistema;
- Modalità **supervisor**: usata per lo svolgimento delle chiamate sistema (primitive), tutte le risorse di sistema sono disponibili.

Importante è che il passaggio da modo utente a modo supervisor richieda al programma in esecuzione in modo utente di "abbandonare" il controllo, cedendolo ad una primitiva sistema. Vedremo come questo si può implementare agilmente sfruttando il meccanismo di interruzione.

4.5 Componenti del S/O

Vediamo quindi, come abbiamo fatto per l'hardware, quelli che sono i **componenti** del S/O e come questi sono organizzati.

Prendiamo come riferimento un sistema *Unix*, in quanto più semplice ed elegante per i nostri scopi.

Adottando un approccio *top-down*, dove per *top* intendiamo lo spazio dell'utente, vediamo le seguenti componenti:

- L'*userspace*, cioè gli applicativi utente veri e propri;
- Gli strumenti che il S/O fornisce all'utente per la gestione del sistema, cioè:
  - La **shell**;
  - **Compilatore** e **linker**;
  - Le **librerie** sistema (che si rivolgono ad API, ecc...).
- Il **kernel**, cioè la parte del S/O che effettivamente gestisce il sistema. Qui troviamo:
  - Il sottosistema **file**, che gestisce il filesystem su uno o più dispositivi a blocchi;
  - A sua volta il sottosistema file interagisce con i **driver** dispositivo (in particolare coi driver dei dispositivi a blocchi), che hanno il compito di gestire a livello hardware il comportamento dei dispositivi;
  - Inoltre, troviamo il sottosistema **controllo processi**, composto da:
    * Funzionalità **IPC** (*Inter Process Communication*) per la comunicazione fra processi;
    * Lo **scheduler**, che decide quali processi mandare in esecuzione;
    * Il sottosistema di **gestione memoria**, che gestisce lo spazio in memoria principale allocato per ogni processo, interagendo col meccanismo della *memoria virtuale*.
- Infine, il *kernel* si appoggia all'**hardware** della macchina.

4.5.1 Modello gerarchico

Un modello più complesso per S/O potrebbe elaborare su questa struttura, prevedendo più livelli intermedi di kernel che implementano *macchine virtuali* via via più vicine all'hardware. Ognuna di queste fornirà al livello superiore funzioni (effettivamente chiamate sistema) sempre più astratte, che implementeranno nel complesso le chiamate sistema rese disponibili ai processi utente.

4.5.2 Modello client-server

Un'altro modello possibile per sistemi distribuiti in rete è quello di avere più *nodi* collegati alla stessa rete. Ogni nodo disporrà del suo kernel, e in esecuzione su quel kernel avrà uno specifico processo (utente o sistema).

La funzionalità del S/O sarà quindi implementata interrogando la rete per il servizio richiesto: sarà quindi compito della macchina su quella rete che effettivamente implementa tale servizio rispondere e fornire, appunto, il servizio.

In ogni caso non analizzeremo sistemi di questo tipo in questo corso, relativi più che altro a sistemi su *cloud*, e quindi all'ambito delle reti informatiche.

4.6 Gestione processi

Informalmente, il termine processo viene usato per indicare un programma in esecuzione sulla macchina.

- Rappresenta la *sequenza di eventi* generati dall'elaboratore durante l'esecuzione;
- Identifica la più piccola *unità di esecuzione* dentro un S/O multiprogrammato: questo consentirà l'esecuzione di *più* processi concorrenti;

Un processo va necessariamente *descritto*, cioè bisogna definire un **descrittore** che lo rappresenta. Del processo ci interessa:

- Il **codice** del programma che esegue;
- I **dati**;
- Il valore dell'**IP**;
- Lo stato dei **registri**;
- Lo **stack**.

Inoltre, ad un certo processo potranno essere associate delle risorse:

- **Memoria** utilizzata;
- **File** aperti;
- **Dispositivi** di I/O a cui ha accesso.

4.6.1 Processi in memoria

Il processo in memoria ha a disposizione il suo *spazio di indirizzamento virtuale*. Viene detto *virtuale* perché verrà allocato in una memoria centrale fisica, le cui locazioni potrebbero non corrispondere esattamente con la memoria offerta al processo (attraverso il meccanismo della *memoria virtuale*).

Partendo dal basso, le regioni di memoria fornite al processo nel suo spazio di indirizzamento saranno:

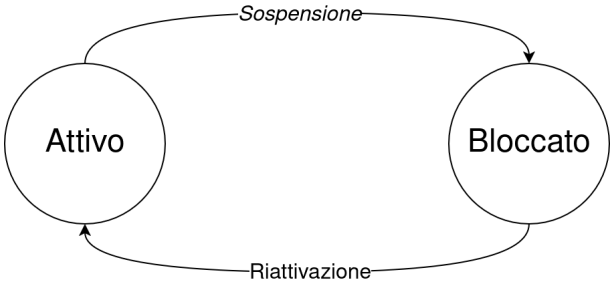
1. `text`: contiene il codice del processo;
2. `data`: contiene i dati statici del programa (sezione `data` e `bss`, che contiene lo spazio riservato a variabili statiche non allocate);
3. `heap`: l'heap del processo, dove vengono allocati oggetti in memoria dinamica;
4. `stack`: si sviluppa verso il basso, rappresenta la pila del processo in esecuzione.

5 Lezione del 02-10-25

Continuiamo la discussione dei processi:

5.0.1 Stato dei processi

In un **S/O monoprogrammato** il processo può trovarsi in uno di due stati:



- **Attivo**: il processo stato creato, è in esecuzione ed ha ancora istruzioni da eseguire.

  Creare un processo significa in primo luogo allocare le strutture dati che lo descrivono. In seguito, si deve allocare un po' di memoria al processo stesso per contenere i suoi dati (istruzioni, pila, ecc...) come visto in 4.6. Abbiamo però che allocare particolari descrittori al processo quando questo è l'unico in esecuzione sarebbe inutile: effettivamente un sistema monoprogrammato può essere tale solo se il processo in esecuzione è in qualche modo il S/O.

  A questo punto il processo è l'unico in esecuzione sulla CPU, e resterà tale fino alla fine del suo *lifetime*. Dovrà però *bloccarsi* se vuole accedere a risorse sistema: fara ciò usando una *chiamata a sistema*;

- **Bloccato**: il processo o qualche altro attore ha causato un qualche evento che ha determinato il passaggio di controllo al S/O (richiesta risorse di I/O, di risorse logiche, interruzioni esterne, ecc...).

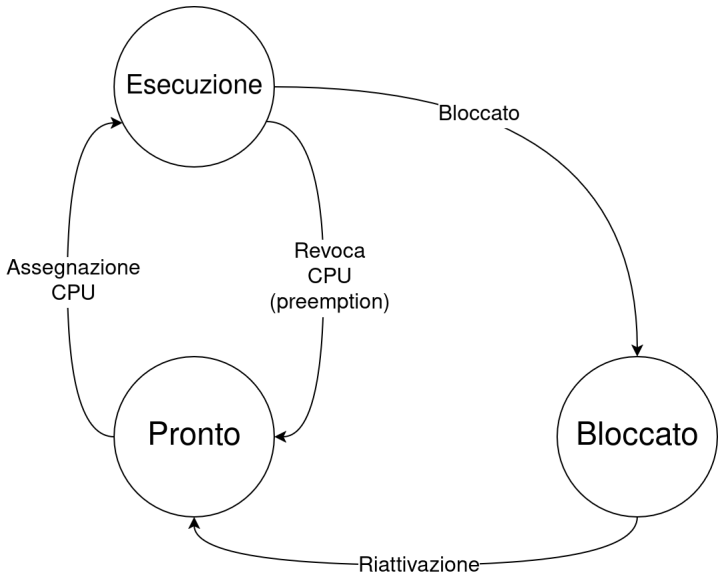
A questo punto sarà nuovamente un evento esterno (interruzione esterna, azione dell'utente) a riportare in esecuzione un processo, creandone uno nuovo se lo scorso aveva terminato, o rimettendo il corrente in esecuzione se si era bloccao su una operazione di I/O o simile.

Notiamo che la memoria del processo potrebbe cambiare anche questo è bloccato: ad esempio se si hanno dispositivi che operano in DMA.

La transizione fra attivo e bloccato è detta *sospensione*, mentre fra bloccato e attivo è è detta *riattivazione*.

Possiamo dire che gli stati *attivo* e *bloccato* sono in qualche modo in corrispondenza con le fasi descritte in 2.1.4 di **CPU-Burst** e **I/O-Burst**.

Se il numero di CPU è minore del numero dei processi, cioè in un sistema **monoprocessore**, ci dotiamo di più stati:



- **Pronto**: in questo caso il processo è in attesa del tempo del processore. Abbiamo che nella maggior parte delle implementazioni questo stato è rappresentato da una *coda* di descrittori di processo, la cosiddetta **coda pronti** (utile per implementare politiche prioritarie $\leftrightarrow$ code prioritarie);
- **Esecuzione**: il processo è in esecuzione sulla CPU;
- **Bloccato**: il processo è in attesa, come nell'esempio precedente.

  In questo caso, come nel precedente, la memoria processo potrebbe essere modificata da dispositivi in DMA. Potrebbero poi essere modificati, in particolari situazioni, i descrittori di processo: magari a causa di operazioni di altri processi col S/O (pipe fra processi, ecc...).

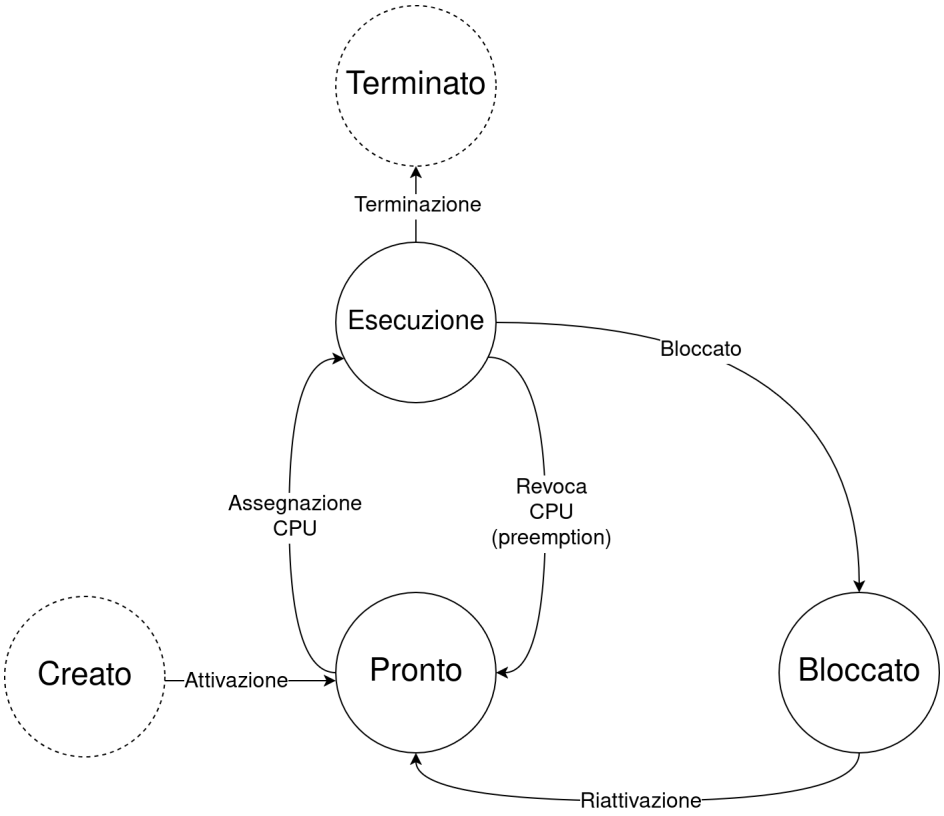
La transizione da pronto e esecuzione è detta *assegnazione CPU*, mentre la contraria è detta *revoca CPU*, o in inglese **preemption**. Notiamo che quest'ultima transizione non è prevista da tutti i sistemi.

L'operazione di assegnazione CPU è eseguita da un componente del S/O detto **scheduler**. Lo scheduler viene eseguito quando il processo in esecuzione cambia stato, quindi

in caso di *revoca CPU* o *sospensione*: sostanzialmente ogni volta che si richiede un nuovo processo da mettere in esecuzione. Questo comprende anche la situazione in cui il processo corrente *termina*.

Abbiamo poi che la transizioni allo stato bloccato sono le stesse dell'esempio precedente, con l'unica differenza che la *riattivazione* del processo non lo mette in esecuzione ma nello stato *pronto*.

C'è inoltre un'altro stato, oltre a quello di *terminazione* prima accennato, di cui dobbiamo tener conto: quello in cui il processo viene *creato*, cioè la transizione di un nuovo processo allo stato di *pronto* (detta *attivazione*):



In questo caso possiamo anticipare che in S/O che adottano politiche *prioritarie* allo scheduling potrebbe essere necessario eseguire lo scheduler (per mettere in esecuzione un nuovo processo di priorità più alta).

5.0.2 Descrittori di processo

Abbiamo introdotto come la gestione dei processi bisogna di apposite strutture dati dette *descrittori* di processo (in inglese **PCB**, *Process Control Block*).

Questa dovrà associare ad ogni processo:

- Nome del processo: questo è il solito **PID** (*Process IDentifier*), ed identifica univocamente ogni processo. Richiedere che i PID siano univici è una *politica* che li trasforma in una *risorsa*: l'S/O dovrà impegnarsi a gestire i PID in modo che non avvengano collisioni;
- Stato del processo: codifica uno degli stati definiti prima;

- Modalità di servizio: questo riguarda la priorità (se implementata) o il tipo di scheduling che si usa per gestire il processo;
- Informazioni sulla gestione della memoria: conterrà puntatori alla memoria dedicata al processo;
- Contesto del processo: l'immagine dei registri all'ultima sospensione del processo;
- Utilizzo delle risorse: conterrà puntatori alle risorse logiche e fisiche a cui ha accesso il processo;
- Idefntificazione del processo successivo: questo serve semplicemente ad implementare, come abbiamo accennato, le code prioritarie di processi (come ad esempio la *coda pronti*).

Una volta definito il descrittore di processo, potremo volerci fornire di:

- La coda dei processi pronti;
- Una o più code per i processi bloccati (solitamente una coda è associata a una risorsa su cui i processi si bloccano);
- Un registro di qualche tipo che contiene il processo attualmente in esecuzione.

5.0.3 Cambio di contesto

Il **cambio di contesto** è l'operazione attraverso cui l'uso del processore viene commutato da un processo all'altro. Questo consiste in:

1. Salvataggio del contesto del processo in esecuzione nel suo descrittore (cioè salvataggio di *stato*);
2. Inserimento del descrittore di processo corrente in coda *bloccati* o *pronti*.
3. Selezione di un nuovo processo da mettere in esecuzione e caricamento nel registro del descrittore del processo corrente (*short term schedulilng*);
4. Caricamento del contesto del nuovo processo e cessione a questi del controllo.

Notiamo che per realizzare il cambio di contesto al S/O (che si occupa poi di portare avanti queste operazioni) abbiamo bisogno di funzionalità implementate in hardware: il "processo" S/O è semplicemente il processo in esecuzione al tempo del cambio, che è forzato a passare al contesto sistema nell'istante in cui si mette ad eseguire codice sistema. Questo si riassume nella lista di operazioni ai passi (1) e (4).

Questo significa che per realizzare un S/O si necessita di un processore che implementi il cambio di contesto (x86 dal 286, ecc...).

Possiamo iniziare ad approfondire il modo in cui indicizziamo il processo. Dal punto di vista concettuale, vorremo usare una tripla:

$$S =< \text{PID}, \text{UID}, \text{GID} >$$

composta da **PID** (*Process IDentifier*, già visto), e **UID** e **GID** (*User IDentifier* e *Group IDentifier*), che rappresentano il proprietario del processo.

In caso di cambi di contesto al contesto sistema, quello che faremo è semplicemente cambiare UID e GID in modo da eseguire lo stesso processo, ma con privilegi diversi.

5.0.4 Creazione e terminazione di processi

Vediamo le operazioni che si possono svolgere sui processi.

Avremo che vorremo supportare *gerarchie* di procssi, dove un processo (padre) può richiedere la creazione di un nuovo processo (figlio). Questo significherà che ogni processo sarà figlio di un altro processa, e potra a sua volta essere padre di altri processi. Chiaramente, le informazioni relative alle relazioni parentali saranno mantenute da S/O nei descrittori di processi.

Se un processo termina, di base:

- Il padre può rilevare il suo stato di terminazione;
- Tutti i suoi figli terminano.

5.0.5 Processi concorrenti

Specifichiamo come più processi vengono eseguiti su una stessa macchina. Abbiamo che questi possono alternarsi (*interleaving*), tipico delle macchine a singolo processore, o eseguire effettivamente l'uno contemporaneamente all'altro. Questo è tipico delle macchine a più processori.

Nel caso di più processi in esecuzione contemporaneamente (diciamo processi *concorrenti*) vengono a verificarsi alcune problematiche:

- Processi **indipendenti**: se due processi $P1$ e $P2$ sono indipendenti, cioè non influenzano l'uno l'esecuzione dell'altra, dobbiamo assicurarci che questo resti vero, cioè dobbiamo risolvere il *problema della riproducibilità*. Visto che le risorse sistema sono condivise, dobbiamo infatti assicurarci che non ci siano effetti collaterali sensibili da un processo o dall'altro.
- Processi **interagenti**: se due processi $P1$ e $P2$ devono interagire fra di loro, possono farlo in maniera *esplicita* (per **cooperazione**), scambiandosi messaggi, o *implicita* (per **competizione**), magari competendo per la stessa risorsa in mutua esclusione.

6 Lezione del 07-10-25

6.1 Nucleo

Il **nucleo** o *kernel* è il cuore di un S/O, il componente che ha il compito di realizzare l'astrazione della *CPU virtuale*. Nel caso di sistemi monoprocessore, vogliamo dividere il tempo fra i processi per dargli l'illusione di essere gli unici in esecuzione sulla macchina.

6.1.1 Scheduling

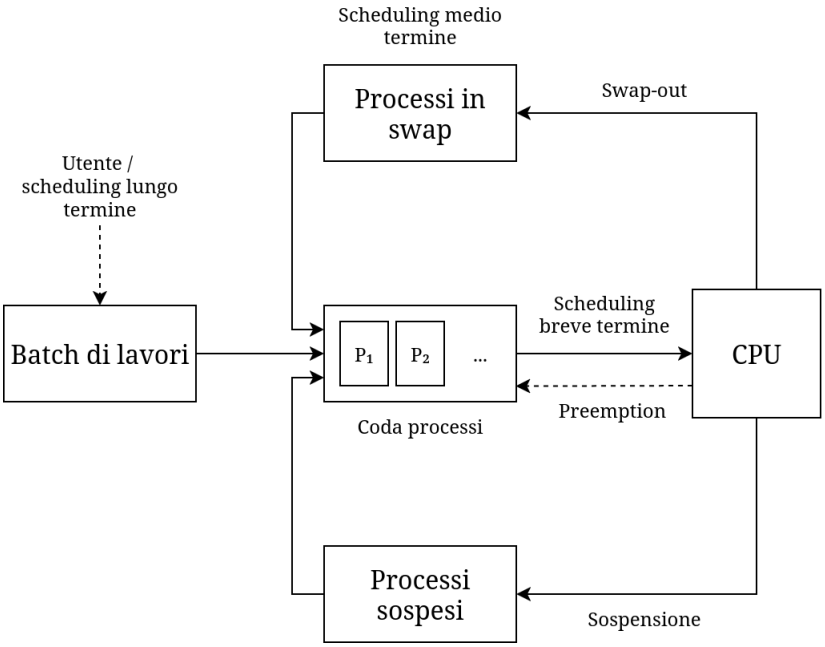
Lo scheduling è l'attività secondo la quale il sistema operativo effettua delle scelte fra quali processi caricare in memoria centrale e a quali assegnare la CPU.

Ci sono 3 diverse attività di scheduling:

- **Breve** termine: lo scheduling propriamente detto, il processo attraverso cui si assegna la CPU. Può essere *preemptive* e *non preemptive* (con o senza diritto di revoca). Solitamente viene invocato molto frequentemente (millisecondi);

- **Medio** termine (*swapping*): il trasferimento temporaneo in memoria secondaria dei processi. Si usa quando la memoria centrale dispone di memoria minore della somma di quella richiesta dai vari processi. Viene invocato più di rado (secondi, minuti);
- **Lungo** termine: la scelta di quali processi caricare dalla memoria secondaria in memoria centrale. Rappresenta un componente importante dei sistemi *batch* multiprogrammati, oggi come oggi quindi sui *server* e meno sulle macchine personali;

Vediamo quindi una schematica che mostra dove queste attività di scheduling avvengono nell'architettura vista:



I processi possono in genere classificarsi in:

- Processi vincolati da **I/O**: passano più tempo a fare I/O burst piuttosto che CPU burst (che sono tanti e piccoli);
- Processi vincolati da **CPU**: passano più tempo a fare calcoli, hanno pochi e lunghi CPU burst.

6.2 Algoritmi di scheduling

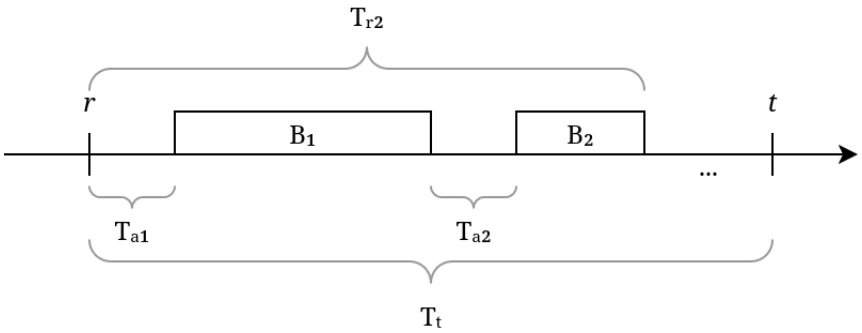
Gli algoritmi di scheduling che vedremo saranno:

- **FCFS** (*First Come First Served*): è *non prioritario* e *non preemptive*, consiste nel assegnare la CPU sempre al primo processo arrivato;
- **SJF** (*Shortest Job First*): è *prioritario* e *non preemptive*, consiste nell'assegnare la CPU al processo più breve;
- **SRTF** (*Shortest Remaining Time First*): è *prioritario* e *preemptive*, rappresenta sostanzialmente la versione con revoca del precedente;
- **RR** (*Round Robin*): è *non prioritario* e *preemptive*: si basa sull'assegnare quanti temporali ugualmente ad ogni processo;

- Schedulazione **su base prioritaria**: introdurremo qui l'idea di *priorità* per ogni processo;
- Schedulazione **a code multiple**: prevediamo più code, che possiamo distinguere usando gli algoritmi sopra descritti, o come vedremo sarà conveniente, assegnando una *priorità* ad ogni cosa;
- Schedulazione di sistemi **in tempo reale**. Questi sono algoritmi che devono assicurare la terminazione deterministicad ei processi. In questo vedremo gli algoritmi:
  - **RM** (*Rate Monotonic*);
  - **EDF** (*Earliest Deadline First*).

6.2.1 Valutazione degli algoritmi di scheduling

Iniziamo a vedere alcune metriche per la valutazione degli algoritmi di scheduling:



- **Utilizzazione** della CPU o *efficienza*, cioè definito ΔB_i il tempo di burst e T il quanto di tempo totale, vogliamo un efficienza E:

  $$E = \frac{\sum \Delta B_i}{T} < 1$$

  il più possibile vicina a 1;
- Tempo medio di **completamento** (o tempo di *turnaround*), cioè il tempo che passa prima che il processo possa completare la sua operazione (terminando). Prendiamo l'istante in cui il processo entra in coda pronti come r (da *richiesto*) e quello in cui termina termina come t (da *termina*). Il tempo di completamento T_t sarà ovviamente:

  $$T_t = t - r$$
- **Produttività** (o frequenza di *througput*), definita come il numero medio di processi completati nell'unità di tempo, cioè l'inverso del tempo medio di completamento:

  $$P = \frac{1}{T_t}$$
- Tempo di **risposta**, valutato dall'istante in cui un processo entra in coda pronti r all'istante in cui risponde, cioè termina un CPU burst (solitamente il primo). Purtroppo, non tutto il tempo di completamento T_c è dedicato al processo, ma

questo viene eseguito, come sappiamo, in più burst (diciamo B_1, B_2, ...). Il tempo di turnaround T_t sarà allora il tempo trascorso fra r e la fine di un burst B_i, cioè:

$$T_{ri} = \text{end}(B_i) - r$$

- Tempo di **attesa**, cioè la somma dei tempi di attesa posti fra i vari burst:

  $$T_a = \sum t_{\alpha i}$$

  Con riferimento ai sistemi interattivi, spesso ci interessa il tempo di attesa iniziale, cioè quello fra l'inserimento in coda pronti e l'inizio del primo CPU burst, o in relazione al nostro schema:

  $$T'_a = t_{\alpha 1} = r - \text{begin}(B_1)$$

  Dovrebbe essere chiaro che il tempo di *attesa* si distingue dal tempo di *risposta*, in quanto:

  – Il tempo di attesa è quello visto dal *processo* prima che questo possa iniziare la computazione;
  – Il tempo di risposta è quello visto dall'*utente* prima di vedersi tornare un primo risultato (si presume che alla fine dei CPU burst inizia un I/O burst che porta avanti qualche operazione di lettura o scritture da periferiche, tangibile per l'utente).

- Rispetto dei **vincoli temporali**, utile principalmente negli algoritmi di scheduling in *tempo reale*.

Fra queste metriche, tempo di **risposta** e di **attesa** sono relativi principalmente ai *sistemi interattivi*, mentre il rispetto dei **vincoli temporali** è relativo ai sistemi in *tempo reale*.

Chiameremo poi O_v l'**overhead** associato all'esecuzione dello scheduler. Ricordiamo che in ogni caso in questa fase stiamo parlando di scheduling a breve termine.

6.2.2 Algoritmo FCFS

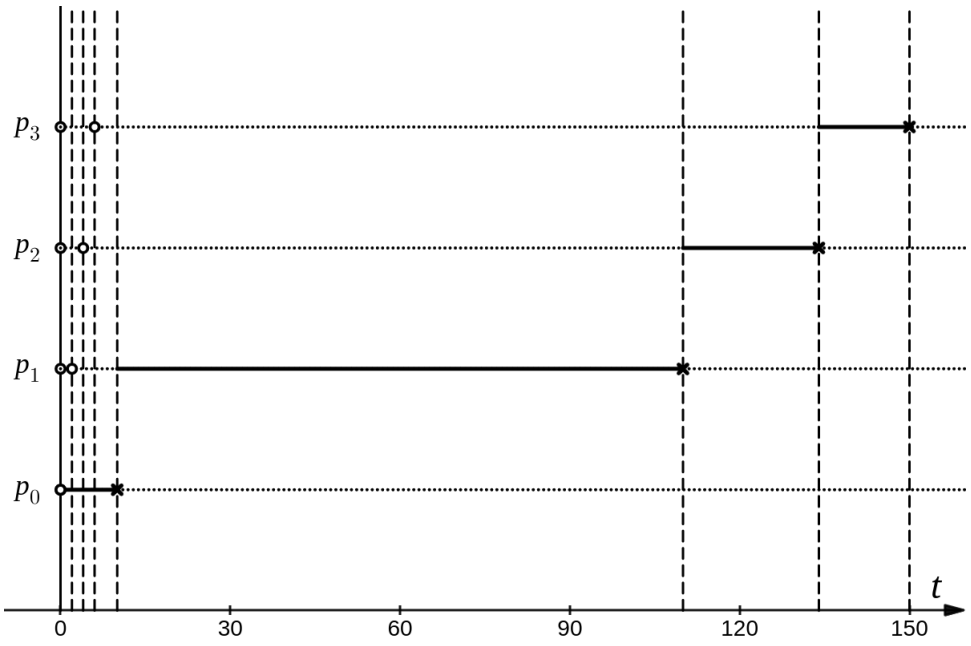
Nell'algoritmo **FCFS** (*First Come First Served*) assegnamo la CPU al primo processo in coda pronti. Sostanzialmente, trattiamo la coda pronti come una coda **FIFO** (*First In, First Out*). Questo lo rende non prioritario e non preemptive.

Quello che otteniamo è un efficienza teorica pari a $E \sim 1$ (c'è un piccolo overhead $O_v \sim 0$ dato dal cambio di contesto), ma generalmente prestazioni piuttosto limitate. Questo è dovuto al fatto che i tempi di attesa (e di conseguenza di completamento) dei processi sono completamente aleatori, e non si fa alcuna scelta informata mirata a minimizzarli.

Vediamo ad esempio il comportamento ottenuto con la sequenza di processi:

Processo	T richiesta	C esecuzione
p_0	0	10
p_1	2	100
p_2	4	24
p_3	6	16

Su un grafico con il tempo t alle ascisse, avremo:



Questo risulta in un tempo medio di attesa di:

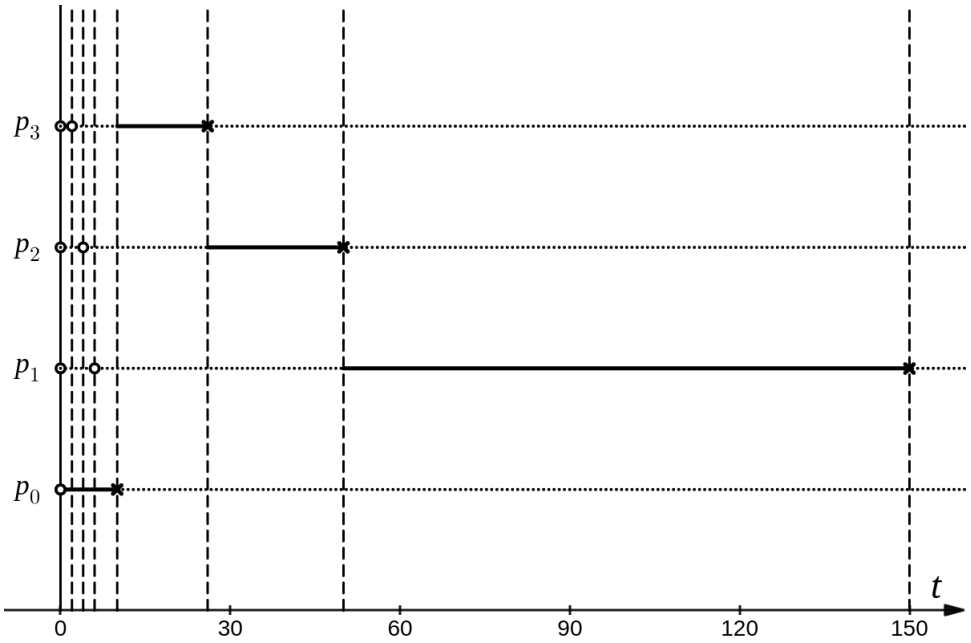
$$\tilde{t}_a = \frac{t_{a0} + t_{a1} + t_{a2} + t_{a3}}{4} = \frac{0 + 8 + 106 + 128}{4} = 60.5$$

Vediamo come questo comportamento può cambiare radicalmente cambiando l'ordine di richiesta dei processi. Poniamo infatti di avere la sequenza:

Processo	T richiesta	C esecuzione
p_0	0	10
p_1	6	100
p_2	4	24
p_3	2	16

dove semplicemente si è invertito l'ordine degli ultimi 3 processi.

Sul grafico avremo:



Questo risulta in un tempo medio di attesa di:

$$\tilde{t}_a = \frac{t_{a0} + t_{a1} + t_{a2} + t_{a3}}{4} = \frac{0 + 44 + 22 + 8}{4} = 18.5$$

Chiaramente molto meglio del caso precedente.

Abbiamo quindi che l'algoritmo è utile per sistemi batch, dove l'unica cosa che ci interessa è uso massimo della CPU (che ci assicura), ma largamente da evitare per sistemi interattivi, e sopratutto per sistemi real-time. Questo è vouto all'aleatorietà legata al momento della richiesta dei processi, che rende impossibile rispondere celeremente o fare qualsiasi tipo di promessa sul tempo di turnaround.

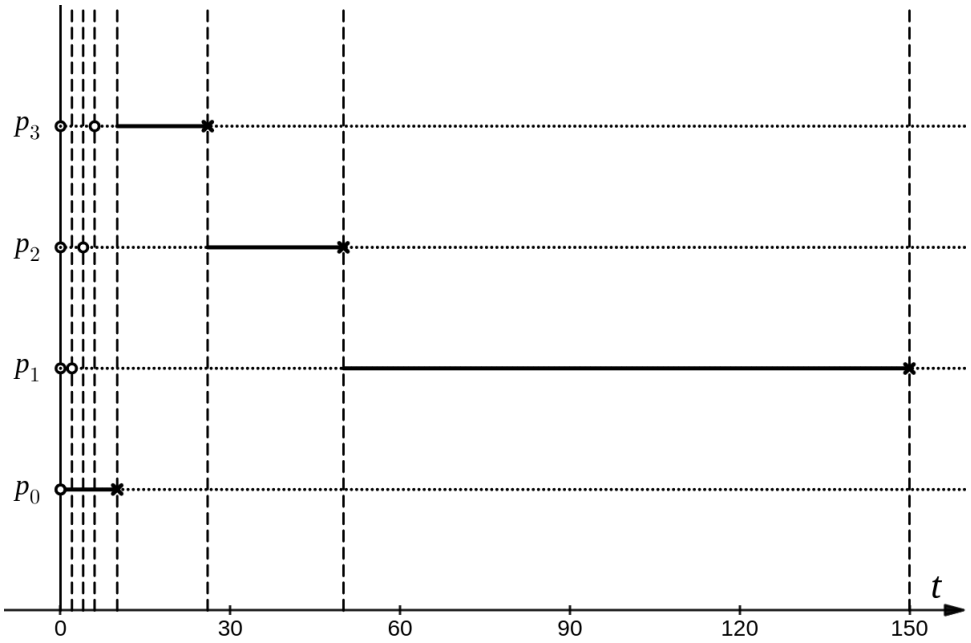
6.2.3 Algoritmo SJF

L'algoritmo **SJF** (*Shortest Job First*) implementa una **priorità statica**: si fa l'ipotesi di conoscere il **tempo di CPU** utilizzato da ogni processo, e assegnare priorità maggiori a processi con tempo di esecuzione minore. Di contro, non è preemptive, cioè una volta assegnata la CPU non può revocarla. Su come il S/O conosce il tempo di esecuzione non facciamo per adesso ipotesi.

Simuliamo anche questo algoritmo, usando la stessa tabella del primo caso in 6.2.2:

Processo	T richiesta	C esecuzione
p_0	0	10
p_1	2	100
p_2	4	24
p_3	6	16

Osserviamo che applicando l'algoritmo si ottiene il flusso di esecuzione:



Cioè esattamente quello che avevamo visto come caso ottimo del FCFS (che ricordiamo aveva tempo medio $\tilde{T}_a = 18.5$), senza che i processi siano arrivati necessariamente nell'ordine ottimo.

Facciamo una nota sulla priorità statica: ad ogni chiamata dello scheduler questo può sapere solo i tempi di esecuzione dei processi attualmente in esecuzione, cioè si potrebbe mandare in esecuzione un processo con tempo di esecuzione maggiore quando ne entra in coda pronti ne entra uno con tempo minore. In questo caso, per la natura *non preemptive* dell'algoritmo, bisogna lasciare che questo esegua prima di mettere il nuovo arrivato in esecuzione.

Adoperando questo algoritmo si minimizza (nel senso matematicamente ottimo) il tempo di attesa medio dei processi, in quanto si cerca di arrivare il prima possibile al processo sucessivo (svolgendo adesso il più veloce). Uno svantaggio sarà chiaramente che i processi che dimostrano tempi di esecuzione lunghi verranno eseguiti sempre per ultimi.

6.2.4 Algoritmo SRTF

Abbiamo introdotto l'algoritmo **SRTF** (*Shortest Remaining Time First*) come una versione *preemptive* del SJF (in questo rimane comunque prioritario).

Visto che è preemptive, viene eseguito *ogni volta* che cambiano le condizioni di scelta (non soltanto quando la CPU è libera, come nel caso dei non preemptive, ma ogni che un nuovo processo entra in esecuzione). Può per questo motivo eseguire senza innescare cambi di contesto.

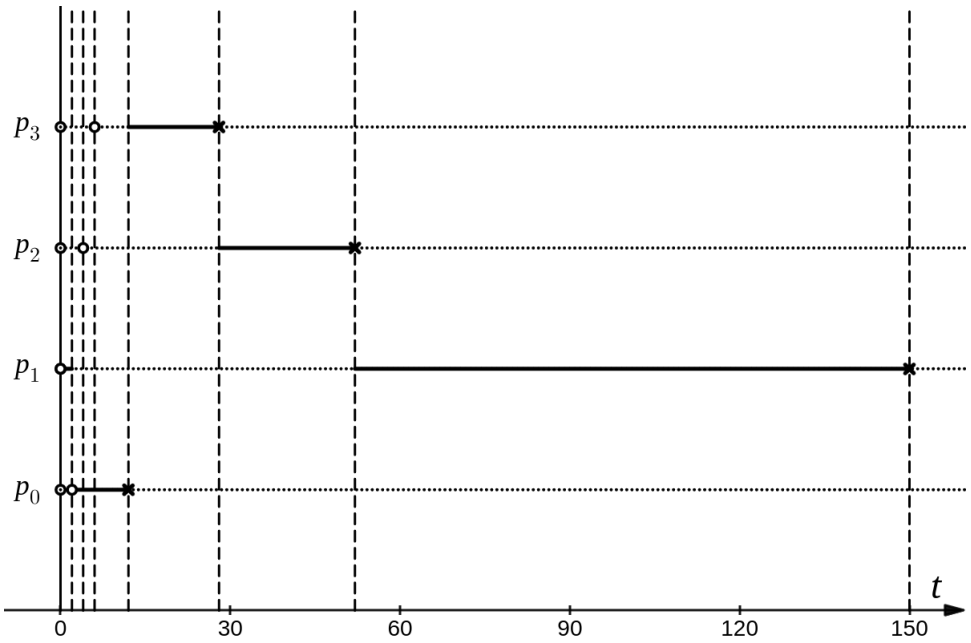
Dovremmo adattare la nostra ipotesi di conoscenza del tempo di CPU ad un'ipotesi di conoscenza del **tempo rimanente** per ogni processo: in questo caso se il processo appena entrato ha tempo rimanente minore di quello del processo attualmente in esecuzione, conviene sfruttare la preemption. Di nuovo, per adesso non facciamo assunzioni su come ricaviamo tale euristica.

L'unica considerazione che ci conviene fare è che aggiornare le previsioni temporali ad ogni evento che cambia le condizioni di scelta chiede allo scheduler di fare più conti, e quindi aumenta leggermente l'overhead O_v. Ampia letteratura dimostra che l'approccio è comunque conveniente.

Simuliamo questo algoritmo con la tabella, modificata dalle precedenti mandando per primo in esecuzione il processo p_1, con tempo di esecuzione maggiore:

Processo	T richiesta	C esecuzione
p_0	2	10
p_1	0	100
p_2	4	24
p_3	6	16

Osserviamo che applicando l'algoritmo si ottiene il flusso di esecuzione:



Questo risulta in un tempo medio di attesa di:

$$\tilde{t_a} = \frac{t_{a0} + t_{a1} + t_{a2} + t_{a3}}{4} = \frac{2 + 0 + 28 + 12}{4} = 10.5$$

prima del primo burst.

Ciò che è importante in questo caso è che il processo p_1 viene sospeso con preemption per permettere l'esecuzione dei CPU burst di p_0, p_2 e p_3, molto più veloci. In questo modo il sistema risulta nel complesso molto più responsivo.

L'SRTF migliora la risposta in tempo reale del SJF, permettendo una riduzione sia dei tempi di turnaround che dei tempi di attesa medi in caso di richieste di esecuzione di processi non ottimali.

Un problema del SRTF, come avevamo visto nel SJF, e la **process starvation**: in genere, negli algoritmi in base prioritaria, si rischia che i processi a priorità minore (in questo caso quelli con tempo rimanente maggiore) non vengano mai serviti e rimangano a lungo in coda pronti, rendendo il sistema meno responsivo.

Anche qui la letteratura ci rende noto che la priorità dei processi in SRTF è **monotona crescente**: man di mano che i processi eseguono, il loro tempo rimanente diminuisce e quindi la priorità aumenta. Questo effetto aiuta a ridurre la process starvation.

6.2.5 Stima dei tempi in SJF e SRTF

Chiariamo la questione di come si possono fare previsioni informate sul **tempo di esecuzione** (in *SJF*) e il **tempo rimanente** (in *SRTF*).

Un'approccio, preso ad esempio il caso del **tempo di esecuzione**, è quello di usare la tecnica della **esponenziale**. Si fa una stima iniziale s_i del tempo di burst t_i esimo. Preso un parametro a con $0 < a < 1$, si aggiorna ad ogni terminazione del processo (quindi facendo delle *osservazioni* per ogni esecuzione del processo) la stima come:

$$s_{n+1} = at_n + (1 - a)s_n$$

Presi $a \sim 0$ si ha che le stime deviano malvolentieri da quella iniziale, mentre con $a \sim 1$ si ha che le stime sono molto volubili rispetto alle osservazioni fatte.

Modeli statistici più complessi possono dare previsioni più accurate, sempre tenendo conto del fatto che lo scheduler deve eseguire con overhead $O_v \sim 0$, o almeno il più piccolo possibile.

Una volta noto il *tempo di esecuzione*, il **tempo rimanente** si può stimare considerando il tempo che il processo ha impiegato finora e sottraendolo dal tempo di esecuzione totale (se non rinunciando al tener conto se tale tempo è stato usato in CPU o I/O burst, purtroppo un'euristica è un'euristica).

7 Lezione del 08-10-25

Continuiamo la discussione degli algoritmi di scheduling.

7.0.1 Algoritmo RR

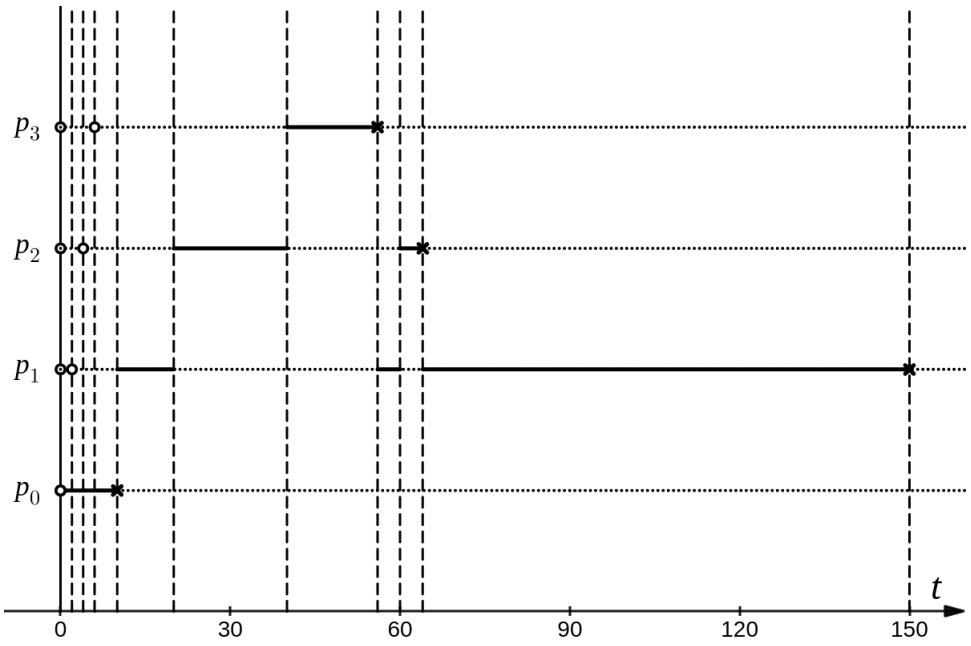
L'algoritmo **RR** (*Round Robin*) è preemptive e non prioritario, e si basa su un meccanismo molto semplice: ad ogni processo viene dato un quanto temporale prefissato, e via preemption si passa al processo successivo quando tale quanto viene esaurito.

Questo lo rende molto efficiente: l'overhead O_v è minimo, quasi al pari di FCFS (leggermente più alto in quanto i cambi di contesto sono più frequenti, uno ogni quanto temporale).

Simuliamo questo algoritmo usando la sequenza di processi richiesti già vista in 6.2.2, e imponendo un quanto temporale di $\Delta T = 20$. La tabella dei tempi di richiesta e esecuzione dei processi sarà:

Processo	T richiesta	C esecuzione
p_0	0	10
p_1	2	100
p_2	4	24
p_3	6	16

In questo caso la *timeline* avrà un aspetto del genere:



Vediamo che il tempo medio, calcolato come:

$$\tilde{t_a} = \frac{t_{a0} + t_{a1} + t_{a2} + t_{a3}}{4} = \frac{0 + 10 + 20 + 40}{4} = 17.5$$

sarà più o meno nella norma rispetto a SJR o STRF.

La cosa importante è, come lo era stato per STRF, la *responsività* che l'algoritmo riesce a realizzare: un processo non resta mai in coda pronti per un tempo maggiore al quanto temporale moltiplicato per i processi in esecuzione meno 1.

Dal punto di vista implementativo, organizzaremo i quanti temporali sfruttando una componente hardware detta **timer**: questo è considerato a tutti gli effetti una periferiche, ed invia (dopo un'opportuna configurazione) interruzioni esterne periodiche. Ogni volta che il processore riceve tale interruzione, mette in esecuzione lo scheduler, che provvede a cambiare il contesto al prossimo processo. Chiaramente, lo scheduler può comunque essere messo in esecuzione da eventi comuni come la terminazione di processi.

Per realizzare l'esecuzione ciclica si usa una semplice coda pronti dove lo scheduler estreae sempre dalla testa e inserisce sempre in fondo alla coda.

Come abbiamo visto dall'esempio, l'RR rende molto semplice stimare il tempo di attesa: se ci sono N processi in esecuzione, $N - 1$ saranno in coda pronti in qualsiasi momento, quindi un dato processo aspetterà:

$$T_a = (N - 1)q$$

dove q è il quanto di tempo.

Chiaramente il T_a diventa troppo grande se ci sono troppi processi.

7.1 Code multilivello

Abbiamo quindi discusso i 4 algoritmi di scheduling fondamentali che avevamo introdotto in 6.2. Ognuno di questi ha pro e contro distinti, ed è più adeguato in una o un altra situazione.

Per questo motivo nei sistemi operativi moderni si preferisce implementare lo scheduling attraverso **più algoritmi** di scheduling, che gestiscono ognuno la situazione che più conviene.

Un modo elegante di realizzare ciò è mantenere **più code**, una per ogni algoritmo di scheduling. Sistemi di questo tipo vengono detti **multilevel queue** o *code multilivello*.

7.1.1 Code di feedback

Una variante interessante delle code multilivello è dato dalle *code di feedback*. Queste nascono per gestire più gerarchie di processi dai requisiti di esecuzione diversi (I/O bound, più interattivi, e CPU bound, più lenti).

In questo caso si prevede una struttura di code, ad esempio come la seguente:

1. Coda RR *veloce*, con quanto $\Delta T = 10$;
2. Coda RR *più lenta*, con quanto $\Delta T = 20$;
3. Coda FCFS, la più *lenta*.

Le code vengono ordinate per priorità decrescente.

Un processo richiesto viene messo nella coda RR più veloce: se al momento della prima revoca CPU non è riuscito a terminare il suo primo CPU burst, viene spostato nella coda RR più lenta. La procedura si ripete finché il processo non è giudicato come *non interattivo* e spostato nella coda FCFS.

In questo modo si riescono a sviluppare sistemi che si *"adattano"* in qualche modo a diversi tipi di processi, scegliendo per ognuno la coda (e quindi la politica di schedulazione) più adatta.

Facciamo qualche altra considerazione: poniamo che ci sia un processo molto interattivo (magari il processo che si occupa di disegnare l'ambiente grafico) nella prima coda, e un processo CPU bound molto lento (magari un software di calcolo scientifico) in coda FCFS.

Avremo che il processo interattivo farà molti I/O burst (scrittura a video, lettura dati da mouse e tastiera, ecc...) e probabilmente i suoi CPU burst finiranno prima del quanto temporale offerto dalla coda RR. Sarà in questi istanti che il processo lento potrà entrare in esecuzione nella coda FCFS.

L'approccio non è ideale in quanto presenta alcuni difetti:

- Potrebbero verificarsi casi dove il processo in coda FCFS non riesce ad eseguire di fronte a una grande massa di processi veloci che arrivano in coda RR (*starvation*): per questo motivo gli S/O moderni implementano altri meccanismi, come l'*aging*;
- Un'altra problematica è data dal fatto che l'ultima coda è non preemptive: quando un processo dal lungo CPU Burst entra in coda FCFS vi resta finche non ha finito di eseguire, bloccando il resto del sistema.

  Possiamo risolvere questo problema rendendo l'FCFS vagamente *preemptive*: visto che ci sono altre due code prima di essa, possiamo cogliere l'occasione del lancio di un nuovo processo per rimettere in esecuzione lo scheduler. Questo meccanismo non è propriamente necessaria per le prime 2 code in quanto il quanto di tempo è limitato (e quindi prima o poi il nuovo processo viene servito), mentre è fondamentale per la coda FCFS che potrebbe bloccare anche per diverso tempo.

Un ultima considerazione che vogliamo fare è se *conviene* riportare i processi dalle code di livello inferiore (più lente) nelle code di livello superiore (più veloci) quando queste si sgombrano: in generale, abbiamo che la letteratura non lo trova particolarmente vantaggioso. Questo perché un processo che è finito in una coda meno interattiva è probabilmente *meno interattivo*, per cui può godere di CPU burst più lunghi e deve restare nella coda dove si trova.

In caso di *aging*, di contro, questo processo è necessario e più che naturale: spostiamo i processi che sono da molto tempo in code inferiori verso le code superiori per forzarne l'esecuzione.

8 Lezione del 14-10-25

8.1 Schedulazione real-time

Veniamo quindi a come implementare la schedulazione nei sistemi in **tempo reale**. Avevamo detto che questi erano sistemi principalmente di tipo *embedded*, cioè incorporati, non general-purpose ma *special-purpose* atti a governare sistemi esterni (sistemi di controllo per veicoli, macchinari industriali, ecc...).

8.1.1 Esecuzione ciclica

Abbiamo che la caratteristica principale di sistemi di questo tipo è il tipo di periferiche con cui interagiscono: invece di periferiche multiple e variabili (come nei sistemi general-purpose), avremo un insieme fisso di dispositivi di ingresso (detti *sensori*) e di uscita (detti *attuatori*).

Questo porta ad un paradigma di esecuzione fortemente periodico: si campiona il sistema esterno attraverso i sensori, compie una qualche elaborazione, e aggiornano gli attuatori per rispondere a quanto rilevato.

Ciò significa che i processi messi in esecuzione devono rispettare il periodo dell'esecuzione ciclica, e produrre il loro risultato entro date *scadenze* date dal periodo corrente e il numero di altri processi in esecuzione. Occorre allora avere un controllo preciso e granulare sul tempo che impiegano a terminare.

8.1.2 Deadline

In questo caso prevederemo, dopo l'istante di richiesta r di un processo, una certa deadline d, calcolata come:

$$d = r + \Delta d$$

dove Δd è il tempo massimo di esecuzione del processo.

- In un sistema *soft real-time* si cerca di fare il possibile per assicurare che il processo termini prima di d;
- In un sistema *hard real-time* la terminazione del processo prima di d è prerogativa dell'integrità dell'intero sistema.

In particolare, riguardo al paradigma di esecuzione ciclica accennato nello scorso paragrafo, avremo che per un processo che deve eseguire ciclicamente con periodo Δt, per ogni istante di richiesta r_i l'istante di richista successivo sarà calcolato come:

$$r_{i+1} = r_i + \Delta t$$

In questo caso sarà fondamentale rispettare la diseguaglianza:

$$d_i < r_{i+1} \Leftrightarrow \Delta d_i < \Delta t$$

data d_i come deadline dopo la richiesta r_i.

Estendo il concetto a sistemi multiprogrammati, avremo che dati n processi il periodo T di aggiornamento simultaneo di ogni processo in esecuzione sarà:

$$T = \text{MCM}\left(\Delta t_i\right)$$

con t_i i periodi di ogni processo: semplicemente si prende il minimo comune multiplo.

Ritornando all'idea dei CPU burst, se il processo si svolge in più CPU burst C_1, C_2, ecc... dovrà quindi essere che:

$$T_e = \sum C_i < \Delta d_i$$

cioè che almeno il tempo di esecuzione del processo sia minore del tempo massimo di esecuzione per rispettare la deadline corrente.

8.1.3 Algoritmo RM

Iniziamo quindi a vedere alcuni algoritmi di scheduling per sistemi real-time. Il primo che vediamo è l'**RM** (*Rate Monotonic*). Questo consiste semplicemente ad assegnare una priorità statica *monotonica crescente* ai processi in base al *rate*, cioè la frequenza, del loro ciclo di esecuzione. Questo equivale ad assegnare una proprietà inversamente proporzionale al periodo t del processo:

$$p \propto \frac{1}{t} = f$$

Visto che la proprietà è statica, chiaramente l'algoritmo è non preemptive.

Facciamo quindi l'esempio dell'esecuzione dell'algoritmo, ipotizzando due processi p_a e p_b:

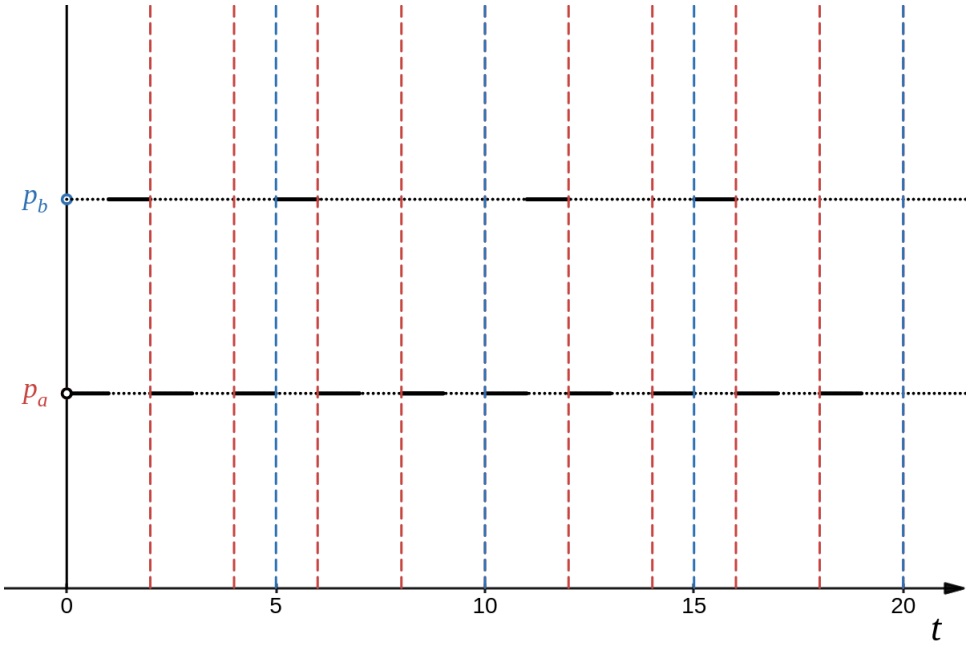
Processo	**Δt periodo**	**C esecuzione**
p_a	2	1
p_b	5	1

Da questo è fra l'altro immediato che, con $\Delta t_a = 2$ e $\Delta t_b = 5$, il periodo complessivo di sistema T è:

$$T = \text{MCM}(\Delta t_a, \Delta t_b) = \text{MCM}(2, 5) = 10$$

Vedremo come questo periodo determina anche il periodo dell'attività dello scheduler.

Simulando l'esecuzione si ha, colorando in rosso le deadline di p_a e in blu quelle di p_b:



Vediamo quindi come riusciamo a rispettare tutte le deadline. Un problema è che, ad esempio all'istante 10, si sono fatti 3 cicli da un unità temporale a vuoto, cioè l'efficienza E è:

$$E = \frac{10 - 3}{10} = 70\%$$

Questo non è immediatamente sbagliato: significa solo che il sistema ha abbastanza risorse da soddisfare ampiamente le richieste in arrivo. Potrebbe diventare un problema quando vogliamo *"stringere"* le temporizzazioni in modo da far fronte ad un maggior numero di processi, o processi con CPU burst più consistenti.

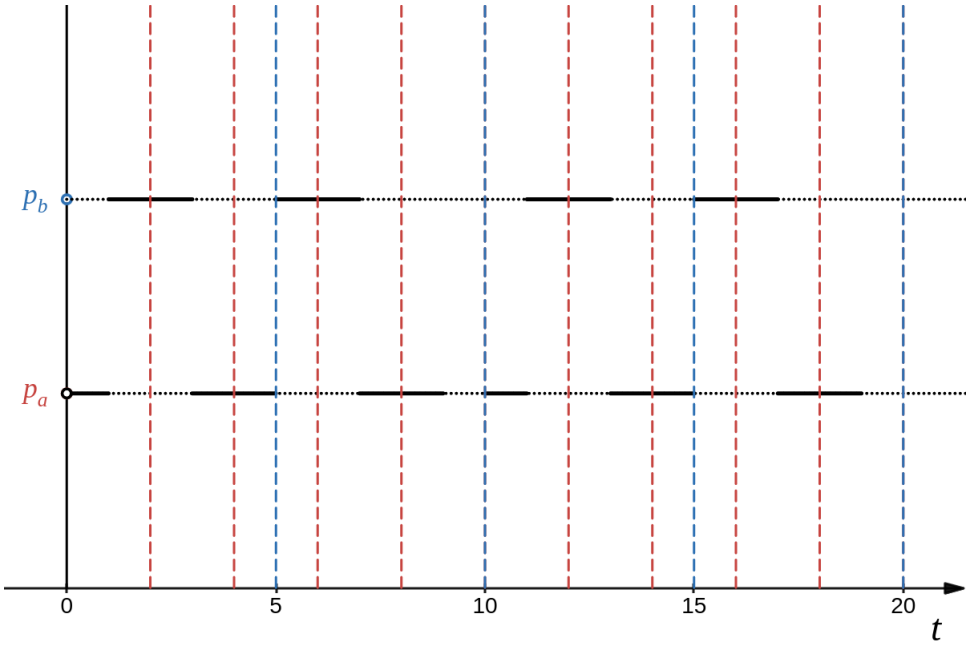
9 Lezione del 15-10-25

Continuiamo la discussione dell'algoritmo **RM** (*Rate Monotonic*).

Volevamo vedere gli effetti che si ottenevano quando si aumentava la pressione sulla CPU sfruttando questo algoritmo. Prendiamo allora gli stessi processi p_a e p_b della scorsa lezione, ma raddoppiamo il tempo di esecuzione del processo p_b:

Processo	**Δt periodo**	**C esecuzione**
p_a	2	1
p_b	5	2

Simulando l'esecuzione si ha, colorando le linee di periodo come nello scorso esepmio:

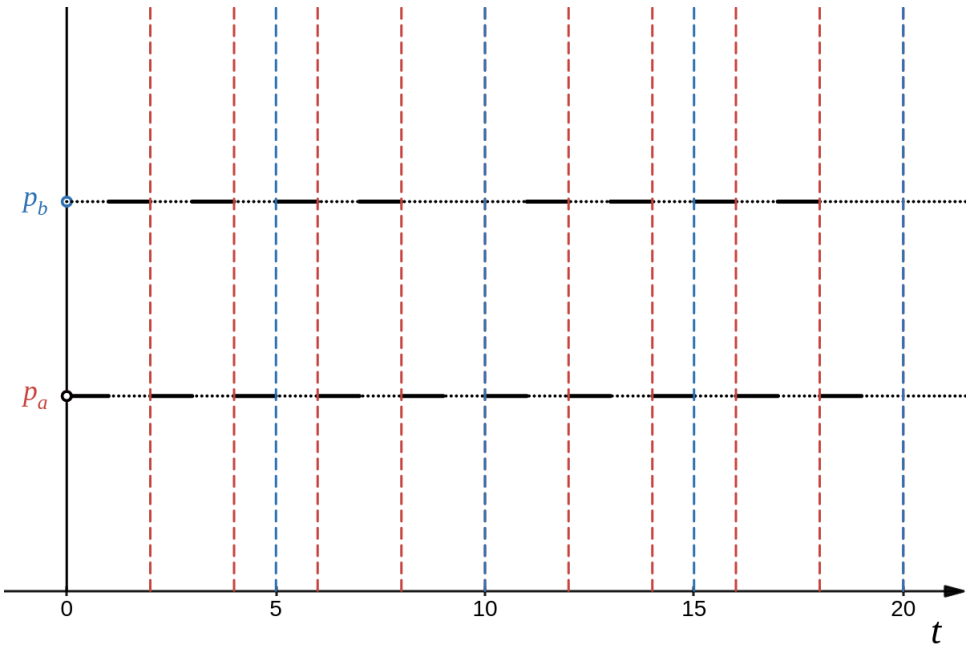


Vediamo quindi come siamo arrivati al 90% di utilizzo della CPU, e come il fatto che l'algoritmo è non preemptive significa che quando p_b accede alla CPU, la tiene anche oltre la linea di periodo di p_a (che ha comunque tempo di eseguire prima della linea successiva).

9.0.1 Algoritmo RM "preemptive"

Potremmo introdurre la preemption nell'algoritmo RM. In questo caso, ad ogni periodo riportiamo in esecuzione il processo con priorità più alta.

Vediamo quindi la timeline che otteniamo applicando questa versione con preemption all'esempio della scorsa sezione:



Notiamo che questo algoritmo di scheduling introduce un overhead maggiore della versione non preemptive, dato dai maggiori cambi di contesto. Inoltre, almeno in questo caso, non varia particolarmente in utilizzo CPU o in efficacia generale.

Comunque, possiamo osservare che mantiene i processi ancora più lontani dalla deadline, che è generalmente un comportamento desiderabile.

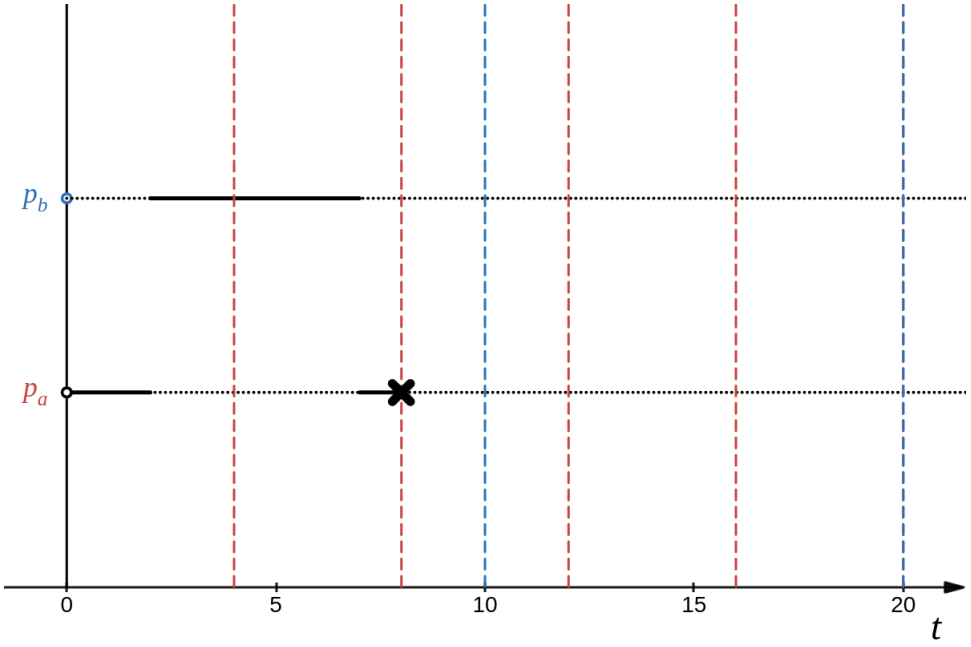
Abbiamo quindi che per gli algoritmi a priorità *static*, il RM è ottimo: se un insieme di processi è schedulabile a priorità statica in real-time, allora lo è con RM. Di contro, se non è schedulabile con RM, non esiste nessun algoritmo a priorità statica che può schedularlo.

9.1 Processi non schedulabili staticamente

Approfondiamo cosa significa, per un insieme di processi, essere *schedulabili a priorita statica*. Prendiamo la tabella di processi:

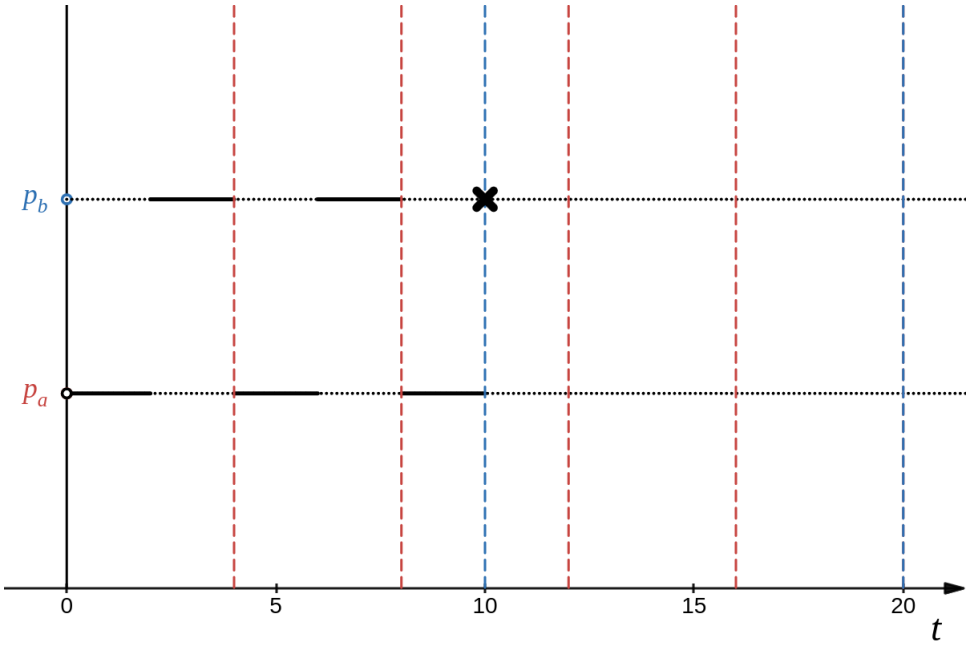
Processo	Δt periodo	C esecuzione
p_a	4	2
p_b	10	5

Vediamo come ogni processo chiede di eseguire per metà del suo periodo.
Se usiamo la schedulazione RM in questo caso, otteniamo:



Dove all'istante $t = 8$ non siamo riusciti a completare il CPU burst di p_a, e si è quindi giunti ad un *overflow*: un fallimento della schedulazione che non ha rispettato la deadline.

Pensiamo allora di utilizzare l'algoritmo RM "preemptive" visto nella scorsa szione. In questo caso, si avrà:



A questo punto è p_b ad andare in overflow! All'istante $t = 10$ infatti non siamo riusciti a completare le sue 5 unità temporali per completare l'esecuzione.

Abbiamo chiaramente incontrato un insieme di processi non schedulabili con priorità statica, e nemmeno introducendo la preemption abbiamo risolto il problema: dovremo trovare una qualche altra soluzione.

9.1.1 Trattazione matematica

Abbiamo che, nel caso dei due processi p_a e p_b, il minimo che dobbiamo rispettare per poter in primo luogo eseguire i processi nel periodo di sistema è:

$$n_a C_a + n_b C_b \leq T$$

dove ricordiamo T è il m.c.m. fra i periodi t_a e t_b, e n_a e n_b sno rispettivamente il numero di volte in cui i processi p_a e p_b entrano in esecuzione per periodo di sistema. In particolare, questi valori si possono calcolare dai perodi dei processi Δt_a, Δt_b, come:

$$n_a = \frac{T}{\Delta t_a}, \quad n_b = \frac{T}{\Delta t_b}$$

Sostituendo, si ha quindi:

$$\frac{T}{\Delta t_a} C_a + \frac{T}{\Delta t_b} C_b \leq T \implies \frac{C_a}{\Delta t_a} + \frac{C_b}{\Delta t_b} \leq 1$$

Possiamo quindi generalizzare quanto trovato alla (ovvia) legge:

$$U = \sum_{i=0}^{n-1} \frac{C_i}{T_i} \leq 1$$

per n processi arbitrari, dove U viene detto **fattore di utilizzazione**.

Nell'esempio considerato finora, questo valore è:

$$U = \frac{2}{4} + \frac{5}{10} = 1$$

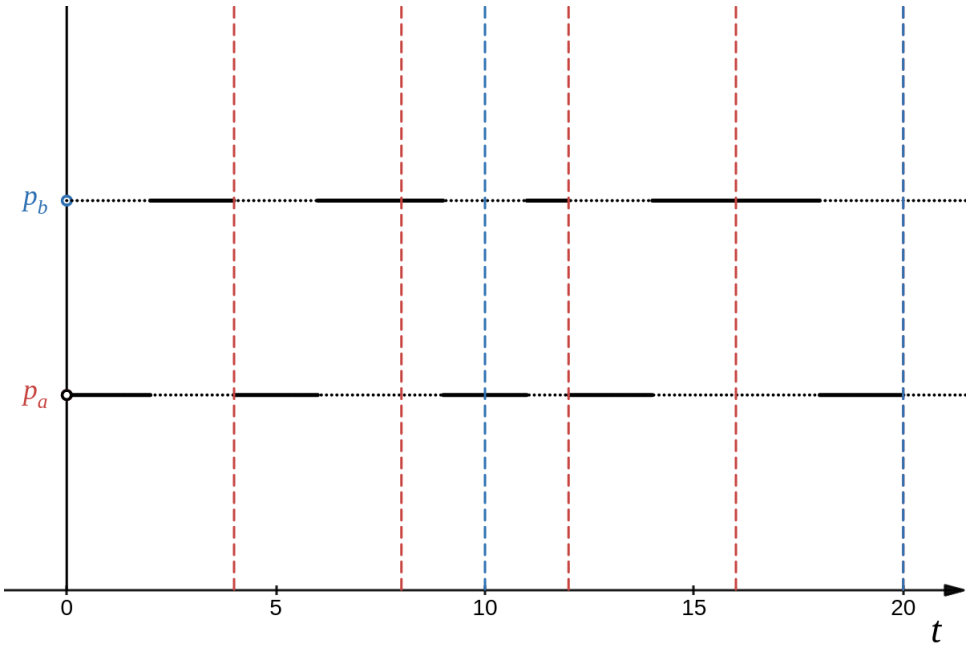
per cui i processi sono schedulabili. Ci manca da trovare un algoritmo che li sappia schedulare.

9.2 Algoritmo EDF

L'algoritmo **EDF** (*Earliest Deadline First*) è un algoritmo di schedulazione real-time, preemptive e a priorità dinamica.

Consiste nel mandare in esecuzione il processo che è più vicino alla sua deadline. Come in tutti gli algoritmi a priorità dinamica, lo scheduler viene messo in esecuzione al cambio dei criteri di scelta (quindi quando un processo entra in coda pronti). In ogni caso, se due processi so trovano ugualmente vicini alla deadline al momento dell'esecuzione dello scheduler, si opta per ridurre i cambi di contesto al minimo e mantenere in esecuzione quello che sta già eseguendo.

Vediamo come questo algoritmo si applica all'esempio schedulabile visto finora:



Vediamo come otteniamo il 100% dell'utilizzazione CPU, e riusciamo a schedulare i processi senza overflow.

All'istante $t = 8$, infatti, il processo p_b è più vicino di p_a alla deadline, e quindi viene mantenuto in esecuzione (fino a $t = 9$ dove termina). In questo modo si riesce ad evitare che il suo CPU burst venga *"tagliato"* prima che esso possa rispettare la sua deadline.

Abbiamo quindi trovato un'algoritmo che risolve i problemi che avevamo incontrato con RM: possiamo anticipare che questo algoritmo è ottimo fra gli algoritmi di schedulazione in real-time a priorità dinamica.

Una considerazione può essere fatta sull'overhead che introduciamo, almeno per l'esempio sopra. Abbiamo detto che sì, si cerca di mantenere al minimo i cambi di contesto, ma c'è comunque un certo overhead dato dall'esecuzione dello scheduler ad ogni creazione di processo.

In questo, potremmo aggiornare il modello introdotto in 9.1.1 come segue:

$$U = ... \leq 1 - O_v$$

dove O_v è un fattore temporale che tiene conto dell'overhead.

9.3 Thread

Introduciamo semplicemente il concetto di **thread** (o *processo leggero*). Avevamo detto che un processo è al contempo:

- Un elemento che possiede delle *risorse*;
- Un elemento a cui viene *assegnata* la CPU (si conserva lo *stato* e si usa uno *scheduler* per decidere quando caricarlo).

Possiamo separare questi due aspetti:

- Definiamo **processo leggero**, o *thread*, l'elemento a cui viene assegnata la CPU;
- Di contro, definiamo **processo pesante**, o *task*, l'elemento che possiede le risorse.

Un processo pesante può essere composto da più thread, ognuno dei quali rappresenta effettivamente un flusso di esecuzione a sé stante. Tutti i thread possono però accedere alle risorse del loro processo pesante (incluso *spazio di indirizzamento*, file aperti, ecc...).

10 Lezione del 21-10-25

10.1 Tassonomia di Flynn

Prima di venire alla sincronizzazione dei processi, vediamo brevemente la **classificazione delle architetture** attraverso la *tassonomia di Flynn*.

Questa è una classificazione che vede un sistema di elaborazione da 2 punti di vista ortogonali:

- La capacità di avere più flussi di **esecuzione**: si possono distinguere **SI** (*Single Instruction Stream*) e **MI** (*Multiple Instruction Stream*). Questo concetto è vicino a quello di *thread* visto nella scorsa sezione;
- La capacità di avere più flussi di **dati**: si possono distinguere **SD** (*Single Data Stream*) e **MD** (*Multiple Data Stream*);

Abbiamo quindi la prima distinzione:

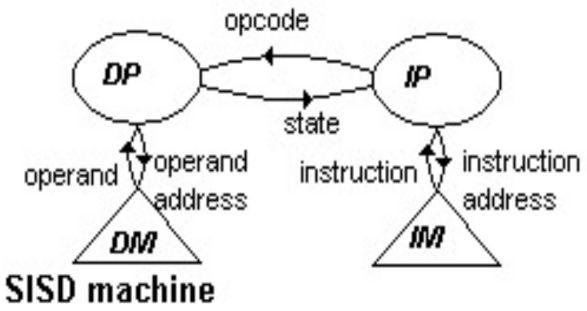
	SI (*Single Instruction Stream*)	**MI (*Multiple Instruction Stream*)**
SD (*Single Data Stream*)	Macchine SISD	Macchine MISD
MD (*Multiple Data Stream*)	Macchine SIMD	Macchine MIMD

Iniziamo a vedere dove si collocano le macchine che conosciamo.

10.1.1 Macchine SISD

Le macchine SISD rappresentano le tradizionali macchine *sequenziali* e *monoprocessore* definite dall'architettura di Von Neumann. In questo caso si ha un solo flusso di istruzioni, ciascuna agente su al più un flusso dati, e ad ogni istante temporale si esegue una singola istruzione.

Vediamo una schematizzazione di questa architettura coerente con Flynn:



Da questa schematizzazione notiamo:

- Un'unita di elaborazione **dati**, detta **DP** (*Data Processor*), che interagisce ottenendo operandi e fornendo indirizzi di operandi (e sperabilmente dati) con uno stream **dati**, detto **DM** (*Data Memory*);
- Un'unità di elaborazione **istruzioni**, detta **IP** (*Instruction Processor*), che interagisce ottenendo istruzioni e fornendo indirizzi di operazioni con uno stream **istruzioni**, detto **IM** (*Instruction Memory*).

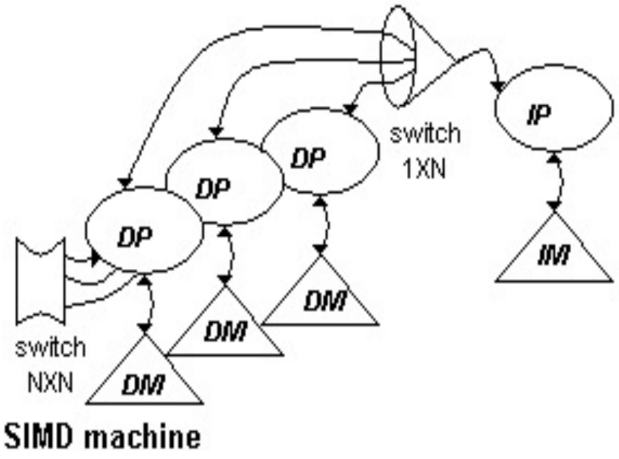
IP e DP interagiscono scambiandosi codifiche di istruzioni (IP $\rightarrow$ DP) e variazioni di stato (della memoria dati, DP $\rightarrow$ IP).

Possiamo notare che la cosiddetta *architettura Harvard* è un architettura che prevede, come dalla schematizzazione sopra, una forte separazione fra stream istruzioni e dati (di contro alla Von Neumann, che prevede un unica fonte di memoria per dati e istruzioni). Entrambe le architetture sono classificate come SISD, la Harvard è più usata in sistemi real-time mentre la Von Neumann è ancora oggi più usata nei sistemi general purpose.

10.1.2 Macchine SIMD

Le macchine SIMD sono formate da unità di elaborazione multiple, che eseguono le stesse istruzioni *contemporaneamente*, ma su flussi di dati differenti.

Una schematizzazione simile a quella sopra riportata è la seguente:



Abbiamo che questa moltiplicazione dei flussi dati su cui si elabora si ha moltiplicando le unità di elaborazione dati (i **DP**), facendole obbedire ad una singola unità di elaborazione istruzion (l'**IP**). Vediamo innanzitutto come si realizza la sincronizzazione fra questi DP: si prevede uno *switch* $1 \times N$ che porta le codifiche di istruzioni dall'IP a tutti i DP.

Per l'interazione fra le DP prevediamo poi uno switch $N \times N$ che le collega. Chiaramente questo switch sarà inefficiente, e vorremo usarlo il meno possibile.

Architetture di questo tipo possono essere *regolari* o create *ad hoc* sulla base della struttura del problema: nel caso di architetture regolari (cioè che rispettano la topologia fisica) non si hanno conflitti, e questo le rende efficienti e poco costose.

Le applicazioni di un'architettura di questo tipo sono nel caso di operazioni fortemente vettorizzate, come ad esempio nelle applicazioni grafiche e multimediali. Inoltre, questo è il tipo di architettura che incontriamo spesso per macchine che devono portare avanti moli massiccie di computazione come i *supercomputer*.

Riassumendo, possiamo dire che l'architettura SIMD prevede 2 tipi di parallelismo:

- **Parallelismo temporale**: c'è un meccanismo di *pipeline*, cioè fasi diverse di un'unica istruzione sono eseguite in parallelo in differenti moduli connessi in cascata.
- **Parallelismo spaziale**: i medesimi passi sono eseguiti contemporaneamente su un array di processori perfettamente uguali, sincronizzati da un solo controllore.

Lato programmatore possiamo prevedere due paradigmi per la compilazione di programmi pensati per l'esecuzione su macchine SIMD:

- Il primo modo è non riscrivere il codice, sapendo che un programma pensato come scalare su macchina sequenziale (SISD), in esecuzione su macchina SIMD sarà vettoriale.

  Avremo quindi che, ad esempio:

```
// somma scalari
c = a + b
```

  su macchina SIMD diventerà:

```
// soma vettori (!)
C = A + B
```

- Nel caso si voglia essere più espliciti nel tipo di operazioni che facciamo, possiamo implementare un **compilatore vettoriale**: l'idea è che questo riconosca automaticamente quando le istruzioni SIMD potrebbero tornare utili per parallelizzare delle operazioni vettoriale, e inserisca quindi le operazioni necessarie.

  Ad esempio, potremmo volere:

```
for(int i = 0; i < 100; i++) {
    c[i] = a[i] + b[i];
}
// qui riconosciamo che il ciclo e' vettorizzabile, e quindi lo
    vettorizziamo
```

10.1.3 Macchine MISD

In una macchina MISD vogliamo avere più flussi di istruzioni che lavorano contemporaneamente su un unico flusso dati.

Abbiamo che questa categoria è sostanzialmente vuota: il parallelismo fra più istruzioni in esecuzione sullo stesso flusso dati si ha effettivamente nei processori moderni solo attraverso il meccanismo della **pipeline**.

Se prevediamo che il processore debba svolgere per ogni istruzione più fasi, fra cui ad esempio:

1. Prelievo istruzione;
2. Decodifica;
3. Prelievo operandi;
4. Esecuzione;
5. Scrittura.

Avremo che questo potrà, disponendo di più unità di elaborazione atte a completare ognuna di queste fasi, parallelizzare come segue:

	Prelievo istruzione	Decodifica	Prelievo operandi	Esecuzione	Scrittura
t_0	i				
t_1	$i+1$	i			
t_2	$i+2$	$i+1$	i		
t_3	$i+3$	$i+2$	$i+1$	i	
t_4	$i+4$	$i+3$	$i+2$	$i+1$	i

In questo, a regime (nella tabella tempo t_4) si avranno più di un unità di elaborazione a lavoro contemporaneamente, e potremmo dire di aver realizzato in qualche modo il paradigma MISD.

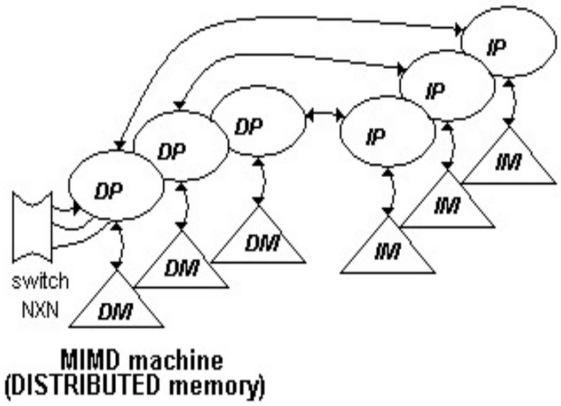
Facciamo tra l'altro una nota sull'efficienza: se l'esecuzione di un'istruzione senza pipeline richiedeva un tempo Δt, e il troughput era $\frac{1}{T}$, prevedendo una pipeline ad n stadi si riesce ad arrivare a $\frac{1}{T} \times n$ (assunto che ogni stadio richieda lo stesso tempo e si riesca a ridurre l'overhead dato da n troppo grandi).

10.1.4 Macchine MIMD

Le macchine MIMD rappresentano per noi la categoria più interessante più flussi di istruzioni sono in esecuzione contemporaneamente su più processori, elaborando insiemi di dati distinti, privati o condivisi.

Ne prevediamo due tipologie principali:

- **DM-MIMD** (*Distributed Memory MIMD*), cioè a *memoria distribuita*. Queste si schematizzano come segue:



In questo caso abbiamo più coppie IP-DP (con relative memorie IM e DM), che rappresentano sostanzialmente più macchine SISD. Uno switch $N \times N$ permette quindi la comunicazione fra le unità.

Abbiamo quindi che tra i nodi non esiste memoria condivisa e ogni nodo esegue indipendentemente un flusso di istruzioni su un differente insieme di dati, memorizzati su spazi differenti. La comunicazione è realizzata mediante una sottorete dedicata (appunto, lo switch).

Applicazioni di questa architettura sono ad esempio una qualsiasi *rete di calcolatori* (come sia sviluppato lo switch non è specificato dal modello, e per noi potrebbe essere anche un router internet). Questo è il caso dei *cluster* di workstation, realizzati solitamente attraverso *Ethernet*. In questo richiediamo dalle macchine che compongono la rete 2 caratteristiche principali:

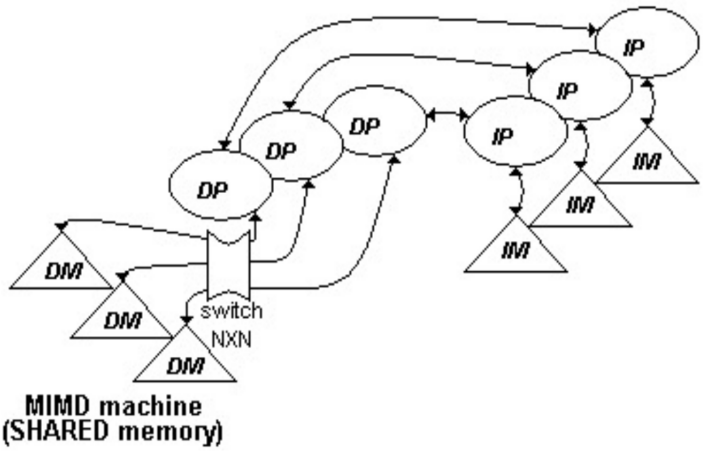
1. **High-availability**: in caso di guasti, la computazione può migrare da un nodo all'altro;
2. **Load-balancing**: i task da eseguire sono allocati nei nodi che hanno il minor carico.

Architetture MIMD più stabili sono poi le rete di interconnessione regolari e dirette (ipercubi, mesh, torus), attraverso cui i nodi si scambiano informazioni secondo il paradigma del *message passing* (*"scambio di messaggi"*). Queste macchine sono molto scalabili e si prestano bene ad algoritmi ad elevata località: sono stati costruiti cluster composti anche da milioni di unità sequenziali.

Una variante del DM-MIMD è il **DM-MIMD MPP** (*Massively Parallel Processing*). Questo è un paradigma utile in applicazioni scientifiche e particolari contesti di calcolo commerciale-finanziario. In un sistema MPP si ha:

  - Migliaia di nodi (CPU standard, ognuna con la propria memoria e la propria copia del SO)
  - Una rete di interconnessione custom molto potente (larga banda e bassa latenza). Affinché l'elaborazione MPP dia effettivi vantaggi occorre disporre di software capace di partizionare il lavoro e i dati su cui opera tra i vari processori.

- **SM-MIMD** (*Single Memory MIMD*), cioè a *memoria condivisa*. Queste si schematizzano come segue:



  Sono quindi macchine sempre *multiprocessore*, ma dove le varie unità di elaborazione dati comunicano attraverso uno switch $N \times N$ con un *pool* unico di memoria (formato anche da più flussi di memoria, ma visti allo stesso modo da ogni unità di elaborazione dati).

  Questo approccio è meno scalabile, realizza la comunicazione fra processori condividendo aree di memoria, e richiede una rete di interconnessione (lo switch $N \times N$ estremamente efficiente). Vogliamo che il numero N di processori sia piccolo ($N < 100$), affiché si possa avere stretto accompiamento fra i nodi. Si incorre chiaramente in problemi di competizione fra unità di elaborazione (mutua esclusione e sincronizzazione) che potrebbero impattare le prestazioni.

10.1.5 Confronto fra SIMD e MIMD

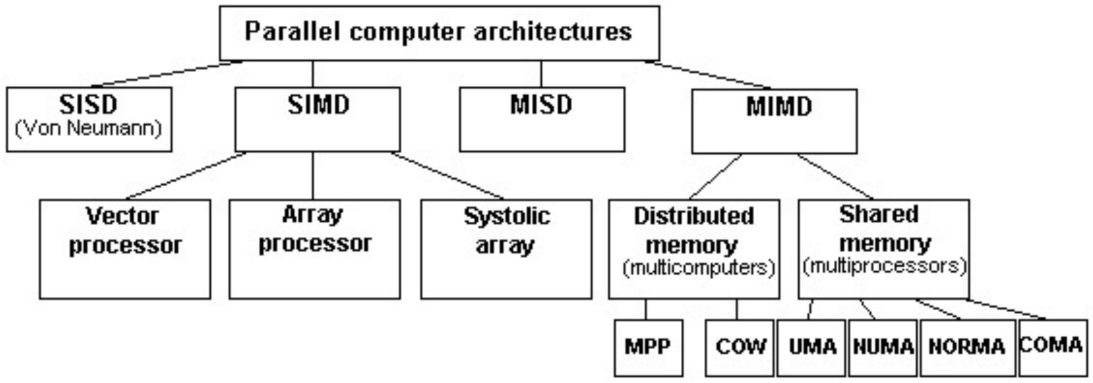
Possiamo fare quindi un confronto fra le architetture apperentemente simili, SIMD e MIMD:

- Le SIMD richiedono meno hardware delle MIMD (c'è un unica unità di controllo, o *elaborazione istruzioni*);
- Le MIMD usano spesso processori general-purpose, quindi costano meno delle SIMD (che richiedono processori o ISA particolari, quindi meno hardware ma più specializzato);
- Le SIMD usano meno memoria delle MIMD (una sola copia del programma in memoria);

- Le MIMD godono di una grande flessibilità in termini di modelli computazionali supportati (si pensi *client-server*, *P2P*, ecc...). Di contro, è piuttosto semplice modificare un programma seqeunziale perché esegua su architettura SIMD in maniera vettorizzata.

10.1.6 Sintesi della tassonomia di Flynn

Possiamo quindi vedere un grafico riassuntivo delle tassonomie viste:

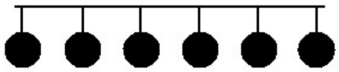


Dove si notano le 4 categorie principali (SISD, SIMD, MISD e MIMD) ed alcune sottocategorie (non abbiamo parlato di tutte le sottocategorie, per una trattazione più completa si rimanda alla letteratura).

10.2 Tipologie di interconnessione

Dopo aver discusso le architetture descritte dalla tassonomia di Flynn, vediamo i tipi principali di **interconnessione** che si possono avere fra unità di elaborazione (o in generale nodi).

10.2.1 Bus



Il **bus** è la più semplice rete di interconnessione che abbiamo visto. Rappresenta una configurazione semplice ed affidabile (a meno che non si rompa il bus tutto funziona), dove ogni nodo a *grado* 1 (tutti i nodi sono direttamente connessi al bus e a nient'altro), e il *diametro* è 1 (la distanza massima è data dal solo bus). Il numero totale di *link* di cui abbiamo bisogno è sempre 1: il bus stesso è l'unico link di cui necessitiamo (i nodi devono solo collegarsi a tale link).

Il problema è chiaramente la *competizione* sull'accesso al mezzo, che è massima: si hanno spesso problemi di mutua esclusione sulle stesse risorse, ad esempio quando più nodi vogliono accedere alla stessa risorsa contemporaneamente e devono farlo attraverso un unico bus.

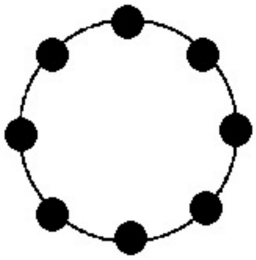
10.2.2 Array lineare

L'**array lineare** espande in qualche modo l'idea del bus: in un certo senso vogliamo partizionare il bus in tanti link nodo a nodo. In questo caso il grado del primo e dell'ultimo nodo sarà 1, mentre quello dei restanti nodi sarà 2. Il diametro sara $n - 1$ per n nodi, e il numero totale di link $n - 1$ (quelli necessari a legare ogni nodo con i nodi adiacenti).

Il vantaggio di questo approccio è la competizione, che viene ridotta al minimo (ogni coppia adiacente di nodi può comunicare indipendentemente dagli altri). Più nello specifico, nel caso ideale possiamo avere fino a $\frac{N}{2}$ competizioni contemporanee e parallele (appunto, una per ogni coppia).

I nodi dovranno quindi fornire servizi di *routing*, cioè permettere a nodi da un capo dell'array di comunicare con nodi dall'altro capo, prendendosi a carico in qualche modo il messaggio da comunicare. Questo è chiaramente a scapito della robustezza: se un nodo si rompe due parti dell'array lineare rimangono separate.

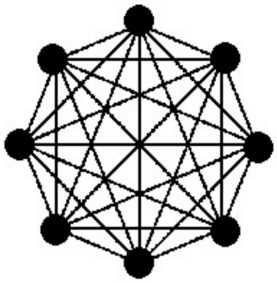
10.2.3 Ring



Il **ring** è un array lineare chiuso su stesso (dove si sono collegati i nodi estremi, cioè quelli con grado 1). Il grado diventa quindi 2 per tutti i nodi. Il diametro subisce la prima caratteristica fondamentale del ring: per raggiungere un dato nodo si hanno a disposizione due direzioni anziché una, per cui il diametro complessivo è $\frac{N}{2}$.

Anche la tolleranza ai guasti migliora, in quanto un nodo guasto non pregiudica necesssariamente l'integrità del sistema (ne servono 2 per isolare una parte della rete).

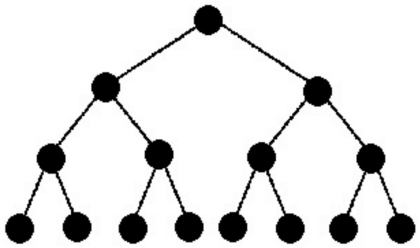
10.2.4 Connessione completa



La **connessione completa** (o *tutti-a-tutti*) è la rete di interconnessione più, appunto, *connessa*, che vediamo. In questo caso ogni nodo comunica direttamente con ogni altro nodo: il grado è per tutti i nodi $N - 1$ e il diametro è 1 (si arriva direttamente al nodo desiderato). Svantaggioso è chiaramente il numero totale di link, che cresce come $N\frac{N-1}{2}$: l'approccio chiaramente non è scalabile!

Dobbiamo quindi trovare modelli per reti di interconnessione che presentino parte dei vantaggi della connessione completa (alta tolleranza ai guasti, bassissimo diametro), riducendo però il numero di link e quindi aumentando la scalabilità.

10.2.5 Albero binario



Si può pensare di ordinare i nodi secondo un **albero binario**. In questo caso vorremo definire l'*altezza* dell'albero come $h = \log_2(N)$ per N nodi e i gradi come:

- 2 per la radice;
- 1 per le foglie;
- 3 per tutti gli altri nodi.

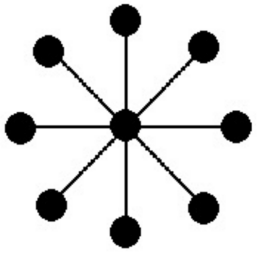
Il diametro sarà (senza dimostrazione) $2 \times (h-1)$ e il numero totale di link $N-1$ ($O(N)$ anziché $O(N^2)$, già migliore).

Questo tipo di rete permette una facile comunicazione fra nodi sugli stessi sottoalberi, mentre per comunicazioni fra sottoalberi distinti porta alla *congestione dei rami alti*: questo la rende poco scalabile. In particolare, più ci avviciniamo alla radice minori saranno i link (e quindi maggiore il carico sul singolo link). Inoltre, i nodi dovranno fare da router, e quindi più ci avviciniamo alla radice più i nodi hanno responsabilità di router sempre maggiori. Questo culmina sulla radice stessa, che chiaramente è sottoposta ad un carico non indifferente e rappresenta il punto più debole dell'architettura ad albero binario.

Per quanto riguarda la tolleranza ai guasti, vale lo stesso discorso: più in alto (verso la radice) avviene il guasto, maggiori sono le conseguenze per il sistema. Come caso limite, se si guasta la radice si incorre in un partizionamento in due dell'intero sistema.

Soluzioni alternative si possono avere sfruttando alberi, anziche binari, *n-ari*, cioè con n figli per nodo.

10.2.6 Stella



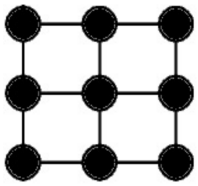
Le reti di interconnessione a **stella** prevedono un singolo nodo centrale che connette $n-1$ nodi periferici. Il grado di tale nodo centrale sarà quindi $n-1$, mentre i nodi periferici

avranno 1. Il diametro sarà 2 (dobbiamo passare sempre dal nodo centrale, a meno che non si voglia parlare col nodo centrale stesso). Il numero di link è ridotto ($N - 1$), e quindi da questo punto di vista il sistema è vantaggioso.

Il difetto più grande è chiaramente la presenza di un singolo nodo centralizzato soggetto a guasti o sovraccarichi. Questo è il classico problema del *single point of failure* delle architetture client-server: possiamo infatti intendere il nodo centrale come un *server* e i nodi periferici come *client* di tale server.

Abbiamo quindi che per quanto si possa rendere potente il nodo server, questo dovrà portare tutto il carico della rete (bassa scalabilità), e un guasto del server rappresenterà una mancanza di servizio per tutti i nodi client (bassa robustezza).

10.2.7 Mesh bidimensionale



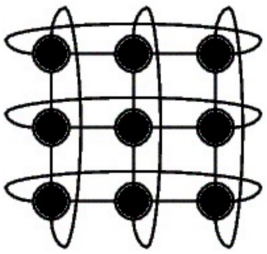
La struttura a **mesh bidimensionale** prevede di disporre i nodi su una griglia bidimensionale. In questo caso il lato della griglia sarà $r = \sqrt{N}$ e il grado dei nodi sarà:

- 2 per i nodi ai vertici,
- 3 per in nodi *"centrali"* ai lati (diciamo nodi agli *spigoli*);
- 4 per tutti gli altri nodi.

Il diametro sarà $2 \times (r - 1)$, e il numero totale di link ($2 \times (n - 2) \times r$).

La resistenza ai guasti di queste configurazioni è buona, ma può essere milgiorata. Vediamo come.

10.2.8 Toro bidimensionale



Collegando i nodi ai lati opposti di una struttura a mesh bidimensionale si ottiene una struttura a **toro bidimensionale**. In questo caso il grado è 4 per tutti i nodi, ma il diametro migliora: va sostanzialemente come r (dove r è il lato calcolato come $r = \sqrt{N}$, uguale alla rete bidimensionale). Il numero totale di link è invece stabile a $2N$.

Questa topologia risulta ben scalabile e notevolmente resistente ai guasti: sostanzialmente rappresenta per la mesh bidimensionale quello che il ring rappresenta per l'array lineare.

Architetture come le ultime viste (mesh, tori, ecc...) possono essere estese a più dimensioni, portando a *ipercubi*, *ipertori*, ecc...

10.3 Metriche di prestazione

Finiamo questa sezione del programma parlando di alcune **metriche** per le **prestazioni** di architetture descritte secondo la tassonomia di Flynn.

- Chiamiamo **speed-up** (S) il rapporto fra il tempo di esecuzione sequenziale sul tempo di esecuzione parallelo (SIMD o MIMD, cioè):

  $$S = \frac{T_1}{T_N}$$

  Questa metrica rappresenta quindi il *guadagno* in velocità che si ha passando ad un sistema multiprocessore. Idealmente vorremmo $S = N$, cioè *lineare* col numero di unità di elaborazione, ma come vedremo a causa dell'overhead introdotto da più processori dobbiamo accontentarci di $S < N$, con $S \approx N$.

  l valore dello speed-up dipende dalle applicazioni, ma anche dall'architettura: nelle SIMD spesso $S \approx N$, mentre nelle MIMD è difficile far crescere S (non è facile far lavorare pienamente tutte le CPU, o trasferire efficientemente fra di esse).

- L'**efficienza** (E) è invece definita come lo speed-up sul numero di unità di elaborazione, cioè:

  $$E = \frac{S}{N}$$

  Come prima, vorremmo $E = 1$, ma dobbiamo accontentarci di $E < 1$, con $E \approx 1$.

Per giustificare come mai non si può mai avere $S = N$ (o equivalentemente, $E = 1$), ci viene incontro la *legge di Amdhal*. Questa dice che un parallelismo *"perfetto"* (nelle varie attività compiute da un calcolatore) non è mai raggiungibile in quanto saranno *sempre* presenti sequenza di programmi *intrinsecamente seriali*. Semplificando, si può dire che ci saranno sempre degli intervalli di tempo sequenziale (T_{seq}), impiegati ad eseguire istruzioni non parallelizzabili come operazioni di I/O, costrutti condizionali, algoritmi intrinsecamente sequenziali, ecc....

Più nello specifico, la legge di Amdahl ridefinisce lo speed-up come:

$$S = \frac{T_1}{T_{\text{seq}} + \frac{T_1 - T_{\text{seq}}}{N}}$$

cioè il guadagno di speed-up è dato solo dalle parti parallelizzabili del programma ($T_1 - T_{\text{seq}}$), mentre le parti seriali (T_{seq}) non hanno grandi guadagni.

Il problema è che, prendendo il limite per $N \to \infty$, si ha:

$$\lim_{N\to\infty} S = \lim_{N\to\infty} \frac{T_1}{T_{\text{seq}} + \frac{T_1 - T_{\text{seq}}}{N}} = \frac{T_1}{T_{\text{seq}}}$$

cioè a dominare lo speed-up sono le parti sequenziali (le parti non sequenziali vengono abbattute velocemente, quelle sequenziali mai e rimangono come overhead complessivo dell'intero sistema).

10.3.1 Multitasking

In conclusione, vogliamo dire che il *multitasking* è quasi sempre vantaggioso (anche intuitivamente), ed è di notevole importanza anche nelle macchine per mantere alto lo sfruttamento delle CPU (ciò che avevamo chiamato *efficienza CPU*).

Il vincolo che vogliamo rispettare sarà, quindi, di base:

$$P >> N$$

cioè avere più processi che unità di elaborazione.

10.4 Sincronizzazione processi

Iniziamo quindi a vedere la **sincronizzazione** fra processi, in particolare con riferimento ai *tipi* di interazione che ci possono essere fra processi, e i problemi di **mutua esclusione** e **sincronizzazione**.

10.4.1 Tipi di interazione

Ricordiamo che i processi possono interagire fra di loro secondo 2 modalità:

- **Cooperazione**: quindi per sincronizzazione diretta o esplicita, cioè definita dai programmi;
- **Competizione**: quindi per sincronizzazione indiretta o implicita, non definita dal codice dei programmi ma causata da tentativi di accesso *simultaneo* a risorse limitati.

Nominiamo poi l'**interferenza**, rappresentata da errori dipendenti dal tempo.
Per l'interazione fra processi faremo riferimento a 2 modelli principali:

- A **memoria comune**: in questo caso prevediamo n processi con risorse private (cioè spazi di indirizzamento privati per ognuno), ma che possono accedere a risorse *condivise* in un terzo spazio di indirizzamento comune per tutti.

  Se vogliamo ricondurci alla tassonomia di Flynn, questo è un esempio di macchina multiprocessore (o monoprocessore in *timesharing*...) di tipo MIMD (se vogliamo SM-MIMD): più unità di elaborazione sugli stessi dati (tralasciando il fatto che ogni unità ha poi i suoi dati privati);

- A **scambio di messaggi**: in questo caso prevediamo n processi in esecuzione su unità di elaborazione distinte (ancora, multiprocessore o monoprocessore in *timesharing*) con le loro risorse (memorie) locali, che possono comunicare fra di loro attraverso un meccanismo di *scambio di messaggi*.

  Notiamo di nuovo che non è necessario che le unità siano necessariamente distinte, e ricordiamo che non sono i processi a comunicare in sé per sé, ma le unità di elaborazione a supportare un meccanismo che permette tale comunicazione.

  Questo è sempre un esempio di architettura MIMD (se vogliamo DM-MIMD), simile ad esempio a quella che ci permettono i sistemi multicalcolatore connessi in rete (più macchine distinte, con le loro risorse, che comunicano fra di loro).

11 Lezione del 22-10-25

11.1 Mutua esclusione

Analizziamo il problema della **mutua esclusione** studiando il seguente pseudocodice C, implementante una semplice struttura *stack*:

```
T stack[n];
int top = -1;

// inserisci in cima
void insert(T y) {
  top++;
  stack[top] = y;
}

// estrai dalla cima
T extract() {
  T temp = stack[top];
  top--;
  return temp;
}
```

Se le variabili `stack` e `top` si trovano in memoria condivisa, o in altre la struttura stack creata si trova in memoria condivisa, potremmo incorrere in situazioni dove più operazioni sullo stack vengono iniziate contemporaneamente (le funzioni che operano sullo stack vengono chiamate contemporaneamente), e lo scheduler interlaccia le operazioni in un modo che rende lo stack inconsistente.

Ad esempio, con 2 processi p_1 e p_2, il primo chiamante `insert()` e il secondo chiamante `extract()` potremmo avere la timeline di esecuzione:

```
t0: top++;              // p1
t1: temp = stack[top]; // p2
t2: top--;              // p2
t3: stack[top] = y;     // p1
```

con conseguenze chiaramente disastrose! Si estre da una zona della pila non allocata e si inserisce sovrascrivendo la cima.

Chiaramente, sappiamo che nella maggior parte dei sistemi reali le istruzioni `top++` , ecc... non saranno eseguite atomicamente, ma lo saranno le istruzioni assembler che implementano tali istruzioni di alto livello. In ogni caso, se il sistema può fallire visto dal livello alto, possiamo essere sicuri che fallirà anche più probabilmente al livello più basso.

Siamo quindi di fronte al classico problema della mutua esclusione, dove vogliamo limitare l'accesso ad una determinata risorsa ad un solo processo per volta, o equivalentemente vogliamo rendere *atomiche* le operazioni su tale risorsa.

11.1.1 Soluzione (scorretta) software

Una prima idea potrebbe essere, *in software*, di introdurre un prologo alle funzioni che dovranno essere atomiche. Tale prologo avrà il compito di controllare un determinato flag di *"prenotazione"* sulla risorsa, ad esempio come:

```
prologo:
  while(occupato == 1);
  occupato = 1;
  // sezione critica
```

```
epilogo:
  occupato = 0;
```

Dobbiamo però renderci conto che a questo punto i due processi si trovano a condividere una nuova variabile in memoria condivisa, cioè il flag `occupato` stesso. Questo significa che possiamo incorrere nuovamente in situazioni dove lo stato diventa inconsistente, ad esempio, assumendo `occupato` inizialmente uguale a 0:

```
t0: while(occupato == 1); // p1, passa
t1: while(occupato == 1); // p2, passa
t2: occupato = 1          // p2
t3: occupato = 1          // p1
// p1 e p2 sono entrambi in sezione critica!
```

Notiamo inoltre che quello che abbiamo implementato è effettivamente una *busy wait* (attesa attiva) sulla variabile occupato, che nella programmazione di sistemi operativi è inaccettabile (rappresenta overhead inutile che potrebbe essere dedicato ad altri processi).

11.1.2 Soluzione hardware

Prevediamo allora una modifica *hardware* che ci permetta di risolvere il problema: magari una nuova istruzione assembler detta *test-and-set* (con mnemonica TSL). La TSL accetterà un registro e un indirizzo in memoria, e il suo funzionamento sarà il seguente:

- Carica il valore all'indirizzo nel registro;
- Imposta il valore nell'indirizzo a 1.

Immaginiamo che realizzare questo tipo di istruzione in sistemi multiprocessore richiederà l'aggiunta al bus di una nuova linea, detta *lock*, che permette ad un processore di bloccare il bus finché non ha terminato la sua operazione.

Quello che potremo fare è quindi implementare 2 routine assembler:

```
lock(x):
  TSL registro, x
  CMP registro, 0
  JNE lock
  RET // torna al chiamante, entra in sezione critica

unlock(x):
  MOVE x, 0
  RET
```

Il fatto che la TSL imposta subito, e sopratutto atomicamente, il valore all'indirizzo `x` a 1, cioè blocca subito la risorsa, ci permette di stare sicuri che nessun'altro avrà l'opportunità di *"rubare"* la risorsa occupata. Anche schedulando a livello istruzioni assembler (che è ciò che si fà nella realta), dopo la TSL lo stato del sistema è consistente: il processo che ha diritto alla risorsa vi accederà, gli altri no.

I vantaggi di questo approccio sono che funziona, e funziona anche su sistemi multiprocessoe (assunta memoria condivisa e bus dotato di linea lock). Gli svantaggi sono che siamo comunque costretti a fare una busy wait, anche se questa è più tollerabile della precedente (la combinazione TSL, CMP e JNE si sbriga in pochi cicli di clock).

In questo caso, prologo ed epilogo visti da un linguaggio di alto livello come il C avranno il seguente aspetto:

```
prologo:
  lock(x)
  // sezione critica
epilogo:
  unlock(x)
```

11.2 Semafori

I semafori rappresentano strumenti generali per la soluzione di problemi di sincronizzazione.

Un semaforo `s` è un oggetto alla quale è associato un intero non negativo, `s.value`, con valore iniziale $s_0 \geq 0$. Al semaforo è associata una lista di attesa, `s.queue`, nella quale sono posti i descrittori dei processi che attendono l'autorizzazione a procedere.

Le primitive sul semaforo coinvolgono il contatore `s.value`, e sono 2:

- `wait(s)`: questa si occupa di decrementare, se > 0, `s.value`. Altrimenti mette il chiamante in attesa. In pseudocodice:

```
void wait(s) {
  if(s.value == 0) {
    // metti il chiamante in attesa
    insert(s.queue, /* chiamante */);
  } else {
    s.value--;
  }
}
```

- `signal(s)`: questa si occupa di incrementare `s.value`, e se c'erano processi in attesa risvegliarne uno. In pseudocodice:

```
void signal(s) {
  if(!isEmpty(s.queue)) {
    primo = extract(s.queue); // e' una coda fifo
    // inserici primo in coda pronti
  } else {
    s.value++;
  }
}
```

11.2.1 Semafori di mutua esclusione

I semafori permettono di realizzare la mutua esclusione. Chiamiamo questo tipo di semafori **mutex**.

Per realizzare un mutex basta inizializzare un semaforo col valore $s_0 = 1$. In questo caso basterà inserire prologo ed epilogo nelle funzioni che vogliamo rendere atomiche:

```
prologo:
  wait(mutex);
  // sezione critica
epilogo:
  signal(mutex);
```

Quello che succede è che alla prima `wait()` il semaforo si svuota e tutti i processi successivi dovranno aspettare nella coda `mutex.queue`. Quando il processo finisce la sua operazione, esegue la `signal()`, liberando la risorsa per il prossimo processo in attesa.

12 Lezione del 23-10-25

12.0.1 Atomicità delle primitive semaforiche

Ora che il problema della mutua esclusione è effettivamente risolto, interroghiamoci su come rendere effettivamente atomiche le `wait()` e `signal()`.

- In ambiente monoprocessore, basterà rendere tali funzioni *primitive* di sistema, e quindi eseguirle con le interruzioni disattivate, per assicurarne l'atomicità.
- In ambienti multiprocessore, si potrebbero invece avere collisioni fra le `wait()` e `signal()` chiamate da processi concorrenti in esecuzione parallela.

  Approfondiamo il problema: posto un semaforo di mutex, ad esempio, questo si troverà in memoria condivisa. Più di un processore potrà accedere alla memoria condivisa attraverso il bus. Se ci limitiamo a rendere "atomiche" le primitive `wait()` e `signal()` disattivando le interruzioni su *un* processore, non risolviamo il caso dove più processori vogliono accedere al semaforo contemporaneamente.

  Possiamo risolvere il problema usando la tecnica introdotte in 11.1.2, cioè dotandoci di un meccanismo hardware di *bloccaggio* del bus, fornito dall'istruzione `TSL`. Potremo infatti usare le primitive `lock()` e `unlock()` per bloccarci sulla risorsa semaforo, cioè dicendo:

```
prologo:
  {
    lock(mutex);
    wait(mutex);
    unlock(mutex);
  }
  // sezione critica
epilogo:
  {
    lock(mutex);
    signal(mutex);
    unlock(mutex);
  }
```

  Notiamo che questa soluzione non è propriamente corretta: infatti il processo non chiamerà la `unlock()` nel prologo finché non verrà svegliato dalla `wait()`: questo significa che potrebbe tenersi il lock sul semaforo, impedendo ad altri processi di segnalare sul semaforo e liberarlo!

  La soluzione corretta sarà quindi quella di ridefinire le primitive semaforiche come segue:

  - `wait(s)`:

```
void wait(s) {
  lock();
  if(s.value == 0) {
    // metti il chiamante in attesa
    insert(s.queue, /* chiamante */);
  } else {
    s.value--;
  }
  unlock();
}
```

– `signal(s)`:

```
void signal(s) {
  lock();
  if(!isEmpty(s.queue)) {
    primo = extract(s.queue); // e' una coda fifo
    // inserici primo in coda pronti
  } else {
    s.value++;
  }
  unlock();
}
```

12.1 Produttori e consumatori

Ipotizziamo adesso una situazione dove:

- Un processo, detto **produttore**, deposita un messaggio in un *buffer*;
- Un'altro processo, detto **consumatore**, preleva il messaggio dal *buffer*.

La policy sul buffer sarà la seguente:

- Il produttore non deve inserire un messaggio nel buffer se questo è pieno;
- Il consumatore non deve prelevare un messaggio dal buffer se questo è vuoto.

Possiamo usare 2 semafori per realizzare una prima soluzione:

- `spazio_disponibile`, con $s_0 = 1$, segnalerà quando il buffer è vuoto;
- `messaggio_disponibile`, con $s_0 = 0$, segnalerà quando il buffer è pieno.
- A questo punto il processo produttore dovrà controllare che il buffer sia vuoto (e aspettare che lo sia se non lo è), inserire il messaggio e segnalare che un nuovo messaggio è disponibile. In pseudocodice:

```
// produttore
do {
  // produci messaggio
  wait(spazio_disponibile);
  buffer.insert(messaggio);
  signal(messaggio_disponibile);
} while(!fine);
```

- Il consumatore dovrà invece controllare che il buffer abbia un nuovo messaggio (e aspettare che lo abbia se non lo ha), prelevare il messaggio e segnalare che il buffer è nuovamente vuoto. In pseudocodice:

```
// consumatore
do {
  wait(messaggio_disponibile);
  messaggio = buffer.extract();
  signal(spazio_disponibile); // prima segnala e poi consuma!
  // consuma messaggio
} while(!fine)
```

12.1.1 Più produttori e consumatori

Complichiamo la situazione: introduciamo un buffer ad n elementi, e prevediamo la presenza contemporanea di più produttori e consumatori.

In questo caso dovremmo assicurare, oltre che la sincronizzzazione coi due semafori appena visti, la mutua esclusione attraverso un mutex. Inoltre, dovremmo prevedere che il semaforo `spazio_disponibile` abbia $s_0 = n$, e non 1. Possiamo fidarci che il meccanismo dei semafori assicura il corretto ordinamento dei processi (per ogni messaggio che inseriamo, si libera uno e un solo consumatore).

- Avremo quindi che lo pseudocodice del produttore sarà:

```
// produttore
do {
  // produci messaggio
  wait(spazio_disponibile);
  {
    wait(mutex);
    buffer.insert(messaggio);
    signal(mutex);
  }
  signal(messaggio_disponibile);
} while(!fine);
```

- Mentre lo pseudocodice del consumatore sarà:

```
// consumatore
do {
  wait(messaggio_disponibile);
  {
    wait(mutex);
    messaggio = buffer.extract();
    signal(mutex);
  }
  signal(spazio_disponibile); // prima segnala e poi consuma!
  // consuma messaggio
} while(!fine)
```

12.1.2 Semafori distinti

Accorgiamoci che in questo sistema, i produttori si bloccano su `spazio_disponibile`, i consumatori si bloccano su `messaggio_disponibile`, ed entrambi si possono bloccare sul `mutex`. Questo non è particolarmente elegante e può portare a situazioni di rallentamento.

Possiamo risolvere questo problema usando, anzichè uno, 2 semafori di mutex.

- In questo caso lo pseudocodice del produttore sarà:

```
// produttore
do {
  // produci messaggio
  wait(spazio_disponibile);
  {
    wait(mutex_produttore);
    buffer.insert(messaggio);
    signal(mutex_produttore);
  }
  signal(messaggio_disponibile);
} while(!fine);
```

- Mentre lo pseudocodice del consumatore sarà:

```
// consumatore
do {
  wait(messaggio_disponibile);
  {
    wait(mutex_consumatore);
    messaggio = buffer.extract();
    signal(mutex_consumatore);
  }
  signal(spazio_disponibile); // prima segnala e poi consuma!
  // consuma messaggio
} while(!fine)
```

Questo chiaramente ci porta a dover fare delle considerazioni sulle modalità in cui si implementa il buffer. In particolare, vorremo che le operazioni `insert()` ed `extract()` siano completamente disaccoppiate e non possano collidere: questo perché la configurazione adottata permette a queste di essere eseguite contemporaneamente (l'una dal produttore e l'altra dal consumatore, che si bloccano su semafori diversi).

Se si adotta la classica implementazione ad array, questo problema non si pone. Se si usa una struttura più sofisticata come una lista, la situazione è più complicata. Vediamo nel dettaglio.

La soluzione che possiamo immaginare è di avere una lista con puntatore alla coda, dove le inserzioni (produttore) si fanno in coda, e le estrazioni (consumatore) si fanno in testa.

In questo caso, per liste con più di un elemento, operazioni di estrazione ed inserzione agiranno su oggetti completamente distinti in memoria, e non avremo problemi. Il problema sarebbe però quando si vuole avere un inserzione ed un estrazione parallela su una lista con un solo elemento.

12.2 Primitive di comunicazione

Tralasciamo per adesso i sistemi in memoria condivisa, e parliamo dei sistemi distribuiti, composti da nodi con memorie locali. Questo è ad esempio il caso delle reti di calcolatori.

Il problema che ci poniamo è come sfruttare un certo **canale di comunicazione** orientato per realizzare due primitive, la primitiva `send(destinazione, messaggio)`, e la primitiva `receive(origine, messaggio)`. Notiamo che la `send()` è *asincrona* (o ugualmente, non *sincrona* o non *bloccante*): quando si ritorna dalla chiamata, non si può avere la sicurezza che il messaggio sia stato recapitato. Una primitiva *bloccante*, di contro, avrebbe sospeso il chiamante fino all'arrivo del messaggio: questo chiaramente implica l'attesa di un ACK da parte del destinatario.

La `receive()` è invece necessariamente bloccante: il processo chiamante viene messo in attesa finché un messaggio non è stato effettivamente ricevuto e può essere recapitato.

Una soluzione più esplicita per il programmatore a cui forniamo le `send()` e `receive()` potrebbe essere quella di imporre il ricevimento dell'ACK, cioè:

```
// invia
send(destinazione, messaggio);
// aspetta l'ACK, e' bloccante
ack = receive(destinazione);
```

Questo sfrutta il fatto che la `receive()` è bloccante e effettivamente risolve il nostro problema, permettendoci di mantenere la `send()` asincrona. Chiaramente, però, richiede al programmatore di scrivere codice più complicato (e corretto!).

12.2.1 Formato del messaggio

Per dotarci di primitive di comunicazione, abbiamo bisogno di stabilire un **formato standard** per i messaggi che andiamo ad inviare. Questo è solitamente diviso in:

- **Intestazione**: contiene informazioni su:
  - **Origine** del messaggio;
  - **Destinazione** del messaggio;
  - **Tipo** del messaggio;
  - **Lunghezza** (in byte) del messaggio;
  - *Informazioni di controllo* varie sul messaggio.
- **Corpo**: contiene il messaggio vero e proprio, o *payload*, che vogliamo trasmettere.

12.2.2 Produttori e consumatori remoti

Vediamo un primo esempio di come le primitive di comunicazione potrebbero essere usate, ad esempio per realizzare un sistema produttore e consumatore.

In particolare, vediamo 2 varianti di comunicazione, **diretta simmetrica** e **diretta asimmetrica**:

- Comunicazione **diretta simmetrica**:
  - Lato produttore si avrà:

```
pid consumatore = /* ... */;
main() {
  msg mess;
  do {
    produci(&mess);
    send(consumatore, mess);
  } while(!fine);
}
```

  - Lato consumatore si avrà:

```
pid produttore = /* ... */;
main() {
  msg mess;
  do {
    receive(produttore, &mess);
    consuma(M);
  } while(!fine);
}
```

  Vediamo come in questo tipo di comunicazione non è necessario prevedere un *buffer*: si invia un messaggio per volta e si aspetta, lato consumatore, per ogni messaggio.

- Comunicazione **diretta asimmetrica**:
  - Lato produttore si avrà:

```
pid consumatore = /* ... */;
main() {
  msg mess;
  do {
    produci(&mess);
    send(consumatore, mess);
  } while(!fine);
}
```

– Lato consumatore si avrà:

```
main() {
  msg mess;
  pid produttore;
  do {
    receive(&produttore, &mess);
    consuma(M);
  } while(!fine);
}
```

In questo caso il produttore non è già noto al consumatore, che invece si mette in ascolto per il primo messaggio disponibile.

12.2.3 Modello client-server

Il discorso fatto finora su produttori e consumatori può essere sviluppato introducendo il paradigma (probabilmente già noto) **client-server**.

In questo caso prevediamo più processi, detti **client** (o *clienti*) che richiedono servizi ad un solo processo, detto **server** (o *servitore*). I client accedono al server tramite una determinata **porta**, che per quanto ci riguarda si occupa anche di *bufferizzazione* delle richieste dei client prima che queste arrivino al processo server vero e proprio.

13 Lezione del 28-10-25

13.1 Lettori e scrittori

Sempre sull'argomento della sincronizzazione fra processi, vediamo l'esempio di più **scrittori** che voglioni scrivere su una risorsa che è letta da più **lettori**.

Iniziamo a vedere quali politiche vogliamo assicurare:

1. Chiaramente, fra gli scrittori c'è una stretta politica di mutua esclusione: non si può scrivere in 2 o più contemporaneamente.
2. Anche fra lettori e scrittori deve esserci mutua esclusione (non possiamo leggere ciò che è inconsistente perché ci si sta scrivendo);
3. In tutti gli altri contesti, vorremmo permettere di avere più lettori contemporanei.

Per risolvere il problema della mutua esclusione fra scrittori (1) prevediamo un semaforo di mutex `sem wrt = 1`, che viene prelevato in fase di scrittura:

```
proc writer {
  wait(wrt); // mutex

  // scrivi

  signal(wrt); // mutex
}
```

Il semaforo `wrt`, inizialmente pensato per la mutua esclusione fra scrittori (1), può essere usato anche dai lettori per risolvere il problema (2). Il problema in questo caso sarà che non assicureremo la politica (3) di letture contemporanee: facendo la `wait()` su `wrt` sblocchiamo infatti un lettore per volta.

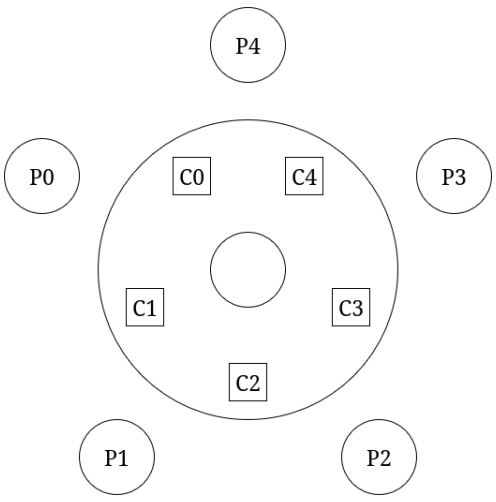
Dotiamoci quindi di un contatore `int readCount = 0`, che tiene conto dei processi lettori che attualemente stanno leggendo la risorsa. Proteggiamo quindi il contatore con un nuovo semaforo di mutex `sem mutex = 1`.

```
proc reader {
  wait(mutex);
  {
    readCount++;
    if(readCount == 1) wait(wrt); // solo il primo aspetta
  }
  signal(mutex);

  // lettura

  wait(mutex);
  {
    readCount--;
    if(readCount == 0) signal(wrt); // solo l'ultimo rilascia
  }
  signal(mutex);
}
```

In questo caso saranno solo rispettivamente il primo e l'ultimo lettore a prendersi la briga di fare la `wait()` e quindi la successiva `signal()` sul semaforo condiviso con gli scrittori.

13.2 Problema dei 5 filosofi

Veniamo quindi ad un esempio celebre di programmazione concorrente.



Ipotizziamo una situazione dove 5 filosofi $p_0, p_1, ...$ sono seduti ad una tavola circolare, al centro della quale è posta una scodella di riso. Sulla tavola, una alla destra di ogni filosofo, ci sono esattamente 5 bacchette $c_0, c_1, ...$ Ogni filosofo per mangiare ha bisogno di due bacchette. Il problemaè: come possono i filosofi coordinarsi per mangiare tutti, e quindi ottenere tutti ciclicamente le 2 bacchette?

Contemporaneamente, possono mangiare al massimo 2 filosofi: ad esempio, se sta mangiando p_0, possono mangiare contemporaneamente solo p_2 o p_3.

Vediamo il comportamento del singolo filosofo. Questo potrà trovarsi in uno di 3 stati:

```
enum State {
  THINKING, // non ha fame
  HUNGRY,   // sta cercando di ottenere 2 bacchette
  EATING    // sta mangiando
}
```

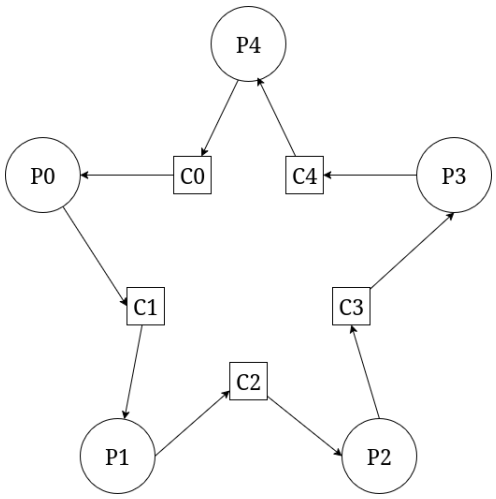
1. Un primo approccio può essere quello di dotarsi di un semaforo di mutex per bacchetta, cioè avere `sem chopstick[5] = 1`. In questo caso lo pseudocodice del filosofo sarà:

   ```
   proc philosopher {
     // pensa

     // aspetta bacchette
     wait(chopstick[i]); // sx
     wait(chopstick[(i + 1) % 5]); // dx

     // mangia

     // rilascia bacchette
     signal(chopstick[(i + 1) % 5]); // dx
     signal(chopstick[i]); // sx
   }
   ```

   Questo approccio può però portare a *deadlock* nel caso in cui tutti i processi riescano ad effettuare la prima `wait()`: in questo caso si troveranno con la bacchetta a sinistra presa da loro, e quella a destra presa dal vicino, col risultato che nessuno può procedere e i filosofi muoiono di fame. Peccato!

   Possiamo modellizzare questa situazione con un grafo, dove nodi circolari rappresentano i **filosofi** (processi), e nodi quadrati rappresentano le **bacchette** (*risorse*). Useremo le frecce da processi e risorse per rappresentare l'**attesa** di una risorsa (chiamata di `wait()`), e le frecce da risorse a processi per rappresentare il **possesso** di una risorsa (`wait()` terminata).

   Vediamo che la situazione di deadlock appena descritto in questo caso è rappresentata da un **ciclo** nel grafo processi-risorse:



Per risolvere questo problema, vorremo prima approfondire il concetto di deadlock, e quindi implementare appropiate tecniche di deadlock **detection** (*rilevamento* di deadlock) e deadlock **avoidance** (*risoluzione* o *prevenzione* di deadlock).

13.3 Monitor

I **monitor** rappresentano un'astrazione di alto livello che permettono la sincronizzazione di processi. Sostanzialmente, sono strutture dati contenenti funzioni (*operazioni*) che vengono eseguite in mutua esclusione (cioè in maniera *atomica*) all'interno di un certo contesto (dove si condivide codice di *inizializzazione* e *dati*).

All'interno di un monitor prevediamo variabili di **condizione**, su cui sono permesse operazioni di `wait()` e `signal()`. Chiaramente queste variabili di condizioni saranno visibili solo all'*interno* del monitor, cioè del codice delle operazioni definite dal monitor.

- La `wait()` mette in attesa un processo finché un altro non esegue una `signal()`;
- La `signal()` sveglia i processi che erano in wait sulla variabile di condizione. La particolarità della `signal()` è che non ha effetti se nessuno è in stato di `wait()`.

13.3.1 Gestione delle variabili di condizione

Assumiamo che un processo P invochi `x.signal()` sulla variabile di condizione `x`, mentre un altro processo Q si trova nello stato `x.wait()`. Cosa dovrebbe succedere a questo punto?

Ci sono due opzioni:

- **Signal and wait**: P aspetta che Q esca dal monitor o si metta in attesa di un'altra condizione: è il caso *preemptive*;
- **Signal and continue**: Q aspetta che P esca dal monitor o si metta in attesa di un'altra condizione.

Noi adotteremo la soluzione *signal and wait*.

13.3.2 Monitor per problema dei 5 filosofi

Vediamo quindi come un monitor può essere usato per risolvere il problema dei 5 filosofi.

```
monitor Philosophers {
  enum State {
    THINKING, // non ha fame
    HUNGRY,   // sta cercando di ottenere 2 bacchette
    EATING    // sta mangiando
  }
  State state[5];

  // variabili di condizione
  condition self[5];

  void pickup(int i) {
    state[i] = HUNGRY; // hai fame
    test(i); // puoi mangiare?
    if(state[i] != EATING) self[i].wait(); // se puoi mangia
  }

```

```
  void putdown(int i) {
    state[i] = THINKING; // non stai piu' mangiando
    test((i - 1) % 5); // sx puo' mangiare?
    test((i + 1) % 5); // dx puo' mangiare?
  }

  void test(int i) {
    if(
      (state[(i - 1) % 5] != EATING) && // sx non sta mangiando?
      (state[i] == HUNGRY) &&           // hai fame?
      (state[(i + 1) % 5] != EATING)    // dx non sta mangiando?
    ) {
      state[i] = EATING; // mangia
      self[i].signal();
    }
  }

  initialization_code() {
    for(int i = 0; i < 5; i++) {
      state[i] = THINKING;
    }
  }
}
```

A questo punto il processo filosofo sarà molto semplice:

```
process philospher {
  Philosophers.pickup();

  // mangia

  Philosophers.putdown();
}
```

Abbiamo che questo approccio è privo di deadlock. Infatti, i processo filosofi non provano ad ottenere le bacchette finché non hanno la sicurezza di poter prendere *entrambe* le bacchette.

14 Lezione del 29-10-25

14.1 Implementazione di un monitor

Veniamo quindi a come si può effettivamente implementare un monitor come descritto nella scorsa lezione.

Avere più funzioni in mutua esclusione significa effettivamente usare un semaforo di mutex `sem mutex = 1`, e avere il seguente prologo ed epilogo di funzione per ogni funzione interna al monitor:

```
func_monitor() {
  // prologo
  wait(mutex);

  // corpo func

  // epilogo
  signal(mutex);
}
```

Il problema diventa quindi la gestione delle variabili di condizione:

- Ricordiamo che `x.wait()` vuole che il processo attuale si sospenda;
- `x.signal()` potrebbe invece bloccare il processo e passare ad un altri (*signal and wait*) oppure continuare col processo corrente (*signal and continue*). Noi, come anticipato in 13.3.1, useremo la prima politica.

Avremo quindi un semaforo inizializzato a zero su ogni variabile di condizione (ad esempio `sem x_sem = 0`) per il blocco, e un contatore dei processi bloccati sulla variabile (ad esempio `int x_count = 0`).

- A questo punto la `x.wait()` sarà:

```
x.wait() {
  x_count++;

  signal(mutex); // devo sbloccare il mutex
  wait(x_sem);
}
```

Il problema è che facciamo una `signal(mutex)`, quando in verità vorremmo segnalare di proseguire ai processi già interni al monitor. Modifichiamo allora il moniotr, introducendo un semaforo `sem next = 0` e un contatore `int next_count = 0` per i processi *"bloccati"* al suo interno.

Prologo ed epilogo saranno allora:

```
func_monitor() {
  // prologo
  wait(mutex);

  // corpo func

  // epilogo
  if(next_count > 0) {
    signal(next); // prima fai uscire i processi interni
  } else {
    signal(mutex); // poi apri il monitor ad altri
  }
}
```

A questo punto il codice della `x.wait()` potrà essere:

```
x.wait() {
  x_count++;

  if(next_count > 0) { // c'e' qualcuno in attesa
    signal(next);
  } else { // non c'e' nessuno, sblocca il monitor
    signal(mutex);
  }

  wait(x_sem);
  x_count--; // uscito da wait()
}
```

- Implementiamo quindi la `x.signal()` secondo la politica signal and wait:

```
x.signal() {
  if(x_count > 0) {
    signal(x_sem);
```

```
4
5       next_count++;
6       wait(next); // sono uno dei processi del monitor
7       next_count--;
8     }
9   }
```

Ci dovrebbe quindi essere chiaro il funzionamento del monitor come un ambiente "ristretto" per i processi del sistema dove lo scheduling non è necessariamente FCFS (o qualsiasi fosse l'algoritmo usato dallo scheduler del sistema).

14.1.1 Conditional wait

Un'altra possibile politica che si può adottare all'interno dei monitor è la cosiddetta **conditional wait**, nella forma `x.wait(c)` dove `c` è un *numero di priorità*. I processi con numero di priorità più piccolo (priorità più alta) vengono schedulati per primi.

Un esempio dove potrebbe essere utile usare tale costrutto è il seguente, dove si implementa un monitor con il compito di allocare una certa risorsa:

```
monitor ResourceAllocator {
  boolean busy;
  condition x;
  void acquire(int time) {
    if(busy) x.wait(time);
    busy = TRUE;
  }

  void release() {
    busy = FALSE;
    x.signal();
  }

  initialization code() {
    busy = FALSE;
  }
}
```

In questo caso prendiamo come argomento `time`, cioè il tempo per cui occupiamo la risorsa (meno tempo $\rightarrow$ più priorità).

14.2 Deadlock

Veniamo quindi alla trattazione vera e propria dei **deadlock**, o *blocchi critici*, che avevamo introdotto in 13.2.

Di base, questi sono situazioni dove ciascun processo, in un insieme di processi, detiene una risorsa e ne desidera una di un altro. Sul grafo di allocazione, equivalentemente, significa che abbiamo un *ciclo*.

Ricordiamo che ci eravamo posti di implementare appropiate tecniche di deadlock **detection** (*rilevamento* di deadlock) e deadlock **avoidance** (*risoluzione* o *prevenzione* di deadlock).

Notiamo adesso che in verità esistono casi dove si possono avere cicli nel grafo di allocazione, ma non avere deadlock: questo è il caso di risorse con **istanze multiple**. In questocaso, un ciclo in un grafo di allocazione simboleggia la *possibilità* di aver deadlock, ma la situazione potrebbe comunque risolversi (lo si verifica per osservazione del grafo).

Una soluzione banale al problema del deadlock è obbligare il programmatore a dichiarare subito tutte le risorse di cui il programma ha bisogno: a questo punto, lato S/O, si potrà realizzare via mutex su tali risorse un sistema di bloccaggio che eviterà sempre i deadlock. Il problema è chiaramente che tale vincolo è estremamente restrittivo, e renderebbe non solo molto scomodo per il programmatore programmare una data applicazione, ma in generale abbatterebbe l'efficienza dell'intero sistema.

14.2.1 Condizioni di deadlock

Esistono 4 condizioni necessarie affinché si verifichi un deadlock:

1. **Mutua esclusione**: solo un processo per volta può usare una data risorsa;
2. **Hold and wait**: un processo che ha ottenuto almeno una risorsa si mette in attesa di altre risorse ottenute da altri processi;
3. **No preemption**: una risorsa può essere rilasciata solo *volontariamente* dai processi che la ottengono, quando questi completano la loro operazione sulla stessa;
4. **Attesa circolare**: esiste un insieme $\{p_0, p_1, ..., p_n\}$ di processi in attesa tali che p_0 aspetta una risorsa di p_1, p_1 aspetta una risorsa di p_2, ..., p_{n-1} aspetta una risorsa di p_n.

Si ha che difficilmente si può lavorare sulle prime 3 condizioni: la muta esclusione deriva dal fatto che ci sono fisicamente risorse limitate nel sistema, e non vogliamo imporre al programmatore vincoli su come e quali risorse vogliono ottenere, o constringerli a programmare meccanismi di recupero dal ritiro di risorse (*preemption* su risorse).