



SAPIENZA  
UNIVERSITÀ DI ROMA

# Appunti di Sistemi Operativi I

Colacel Alexandru Andrei

## Disclaimer

---

Le fonti sono le slides del Prof. Tolomei, appunti di colleghi presi a lezione e integrazioni con il libro "*I moderni sistemi operativi IV edizione*" di Andrew S. Tanenbaum. Se diversamente verrà indicato a pié di pagina.

**Nota:** è vietata assolutamente la vendita di questo materiale in qualsiasi forma senza il mio consenso.

---

# Indice

<b>1 Introduzione</b>	<b>3</b>
1.1 Kernel/User mode e protezione della memoria . . . . .	3
1.2 Che cos'è un Sistema Operativo . . . . .	3
1.2.1 L'OS come macchina estesa . . . . .	3
1.2.2 L'OS come gestore delle risorse . . . . .	3
1.3 Analisi dell'hardware . . . . .	4
1.3.1 System Bus . . . . .	4
1.3.2 Dispositivi di I/O . . . . .	4
1.3.3 Processori . . . . .	5
1.3.4 Protezione delle system call . . . . .	7
1.3.5 Passaggio di parametri . . . . .	7
1.3.6 Timer, Istruzioni atomiche, Virtual Memory . . . . .	7
1.4 Progettazione e implementazione del sistema operativo . . . . .	9
<b>2 Gestione dei Processi e thread</b>	<b>10</b>
2.1 Stato di esecuzione del processo . . . . .	11
2.2 Process Control Block (PCB) . . . . .	12
2.3 Creazione di processi . . . . .	13
2.3.1 Risorse del padre vs figlio . . . . .	14
2.3.2 Esecuzione del padre vs figlio . . . . .	14
2.4 Terminazione di processi . . . . .	17
2.5 Scheduling dei processi . . . . .	17
2.5.1 Il Context Switch . . . . .	18
2.6 Comunicazione dei processi . . . . .	18
2.7 Scheduling dei processi . . . . .	19
2.8 Scheduling Preemptive e Non-Preemptive . . . . .	20
2.8.1 Problemi . . . . .	20
2.9 Il Dispatcher . . . . .	20
2.10 Definizioni utili . . . . .	20
2.11 Criteri/metriche di scheduling . . . . .	22
2.12 Algoritmi di Scheduling . . . . .	23
2.12.1 First-Come-First-Serve (FCFS) . . . . .	23
2.12.2 Esempi . . . . .	24
2.12.3 Round - Robin (RR) . . . . .	28
2.12.4 Esempi . . . . .	28
2.12.5 Shortest-Job-First (SJF) . . . . .	31
2.12.6 Esempi . . . . .	31
2.12.7 Priority Scheduling . . . . .	34
2.12.8 Multilevel Queue (MLQ e MLFQ) . . . . .	35
2.12.9 Esempio di suddivisione tramite MLQ e MLFQ: . . . . .	35
2.12.10 Esempio . . . . .	36
2.12.11 Extra: Lottery scheduling . . . . .	37
2.13 Thread e Multi-threading . . . . .	38
2.13.1 Vantaggi . . . . .	38
2.13.2 Programmazione Multi-core . . . . .	39
2.13.3 Tipi di parallelismo . . . . .	39
2.13.4 Kernel threads vs User threads . . . . .	39
2.13.5 Modelli di Multi-threading . . . . .	42
<b>3 Sincronizzazione tra Processi/Thread</b>	<b>45</b>
3.1 I Lock . . . . .	46
3.2 I Semafori . . . . .	49
3.2.1 Esempi . . . . .	50
3.2.2 Problemi semafori . . . . .	51
3.3 I Monitor . . . . .	52
3.3.1 Problemi di Lettura e Scrittura . . . . .	54
3.3.2 (First) Readers-Writers Problem: Soluzione I . . . . .	54
3.3.3 Soluzione II: usare i Monitor . . . . .	54

3.4 Deadlock . . . . .	55
3.4.1 Cos'è un deadlock . . . . .	56
3.4.2 Resource Allocation Graph (RAG) . . . . .	56
3.4.3 Modellazione dei deadlock . . . . .	56
3.4.4 Esempio . . . . .	57
3.5 Prevenire ed evitare un deadlock . . . . .	58
3.5.1 Stato sicuro . . . . .	59
3.5.2 Archi di pretesa . . . . .	59
3.5.3 Esempi . . . . .	59
<b>4 Gestione della Memoria</b>	<b>61</b>
<b>5 Gestione dei Sistemi di I/O</b>	<b>62</b>
<b>6 File System</b>	<b>63</b>
<b>7 Advanced Topics</b>	<b>64</b>
<b>8 Esercizi</b>	<b>65</b>
<b>9 Formulario</b>	<b>66</b>

# 1 Introduzione

Il programma con cui si interagisce può essere formato testo (quindi la **\$SHELL**) oppure modalità **GUI** (Graphical User Interface) quando ci sono le icone. La maggior parte dei computer ha due modalità operative: kernel e utente. Il sistema operativo è il componente software di maggior importanza e viene eseguito in modalità kernel (detta anche Supervisor). In questo modo ha accesso a tutto l'hardware e può eseguire qualunque istruzione che la macchina sia in grado di svolgere. La modalità utente ha a sua disposizione solo un sottoinsieme di istruzioni.

Esiste una differenza tra il sistema operativo (che troverete scritto anche OS) e il normale software in modalità utente. Infatti l'utente non è libero di scrivere E.g un gestore degli interrupt ecc... perchè esistono protezioni hardware dell'OS stesso

Questa differenza è meno evidente nei sistemi embedded o sistemi integrati (che possono non avere la modalità kernel) o sistemi interpretati che usano interpreti e non hardware per separare i componenti

## 1.1 Kernel/User mode e protezione della memoria

Alcune istruzioni eseguite dalla CPU risultano essere più sensibili di altre. Affinché tali istruzioni privilegiate vengano utilizzate esclusivamente dal sistema operativo, la CPU può essere impostata in due modalità specifiche a seconda del programma in esecuzione. La CPU può essere quindi impostata in:

- Kernel mode ossia in modalità senza alcuna restrizione, permettendo l'esecuzione di qualsiasi istruzione (utilizzata dal sistema operativo)
- User mode, dove non sono possibili:
  - Accedere agli indirizzi riservati ai dispositivi di I/O
  - Manipolare il contenuto della memoria principale
  - Arrestare il sistema
  - Passare alla Kernel Mode
  - ...

Per poter impostare una delle due modalità, viene utilizzato un bit speciale salvato in un registro protetto: se impostato su 0 la CPU sarà in Kernel mode, mentre se impostato su 1 la CPU sarà in User mode

## 1.2 Che cos'è un Sistema Operativo

Il Sistema Operativo esegue fondamentalmente due funzioni non correlate. Da una parte fornisce ai programmati funzioni di applicazioni un insieme di risorse astratte e dall'altra le risorse hardware.

### 1.2.1 L'OS come macchina estesa

L'architettura è l'insieme delle istruzioni, organizzazione della memoria, I/O, e struttura dei bus. Per la gestione dell'hardware si utilizza un software chiamato **driver** che fornisce l'interfaccia per la lettura e la scrittura senza che il programmatore si occupi dei dettagli. L'astrazione è la chiave per risolvere la complessità, una buona astrazione suddivide un'attività complessa in due attività più gestibili. La prima riguarda la definizione e l'implementazione delle astrazioni. La seconda riguarda l'impiego di queste astrazioni per risolvere problemi reali.

### 1.2.2 L'OS come gestore delle risorse

La gestione delle risorse include il multiplexing (condivisione) delle risorse in due modalità diverse: nel tempo e nello spazio. Quando una risorsa è condivisa temporalmente programmi o utenti diversi fanno a turno ad usarla in un certo arco di tempo finito. L'altro tipo di multiplexing è nello spazio. Pressupponendo che vi sia abbastanza memoria per gestire parecchi programmi in memoria contemporaneamente, specialmente se necessita solo di una piccola frazione del totale. Tutto questo solleva però problemi di equità, protezione ecc... risolvibili dall'OS.

## 1.3 Analisi dell'hardware

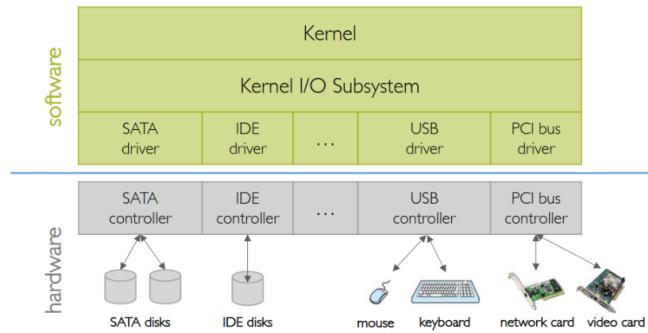


Figura 1: Alcuni dei componenti di un semplice personal computer

### 1.3.1 System Bus

Inizialmente veniva usato un unico bus per gestire tutto il traffico. Combina le funzioni di:

- Data bus effettivo di informazioni
- Address bus per determinare dove tali informazioni devono essere inviate
- Control bus per indicare quali operazioni dovrebbero essere eseguite

Sono stati aggiunti più bus dedicati per gestire il traffico da CPU a memoria e I/O, PCI, SATA, USB, ecc.

### 1.3.2 Dispositivi di I/O

Ogni dispositivo I/O è composto da due parti fondamentali il dispositivo fisico stesso e il controller del dispositivo (chip o set di chip che controllano una famiglia di dispositivi fisici).

Il sistema operativo comunica con un controller del dispositivo utilizzando un driver di dispositivo specifico.

La comunicazione con i "Dispositivi di Controllo" ciascun controller del dispositivo ha un numero di registri dedicati per comunicare con esso:

- **Registri di stato:** forniscono alla CPU informazioni sullo stato del dispositivo I/O (ad esempio, inattivo, pronto per l'input, occupato, errore, transazione completata)
- **Registri di configurazione/controllo:** utilizzati dalla CPU per configurare e controllare il dispositivo
- **Registri dei dati:** utilizzati per leggere o inviare dati al dispositivo I/O

La CPU riesce ad indirizzare mettendo l'address di un byte di memoria nel address bus, specificando il segnale READ sul controllo del bus. Alla fine, la RAM risponderà con il contenuto della memoria sul Data bus.

La CPU può comunicare con un controller del dispositivo in due modi:

- I/O mappati alle porte che fanno riferimento ai registri del controller utilizzando un I/O separato spazio degli indirizzi.  
Il registro di ciascun controller del dispositivo I/O è mappato su una porta specifica (indirizzo). Richiede classi speciali di istruzioni CPU (ad es. IN/OUT). L'istruzione IN legge da un dispositivo I/O, mentre OUT scrive. Quando si usano le istruzioni IN o OUT, l' M/#IO non viene asserito, quindi la memoria non risponde e il chip I/O sì.
- I registri del controller del Memory-Mapped I/O utilizzano lo stesso spazio di indirizzamento utilizzato dalla memoria principale.  
Il Memory-mapped I/O "spreca" parte dello spazio degli indirizzi ma non necessita di istruzioni speciali. Per le porte del dispositivo I/O della CPU sono proprio come i normali indirizzi di memoria. La CPU utilizza istruzioni simili a MOV per accedere ai registri del dispositivo I/O. In questo modo viene asserito l' M/#IO indicando l'indirizzo richiesto dalla CPU riferito alla memoria principale

Per eseguire le attività di I/O, vengono utilizzate due modalità di gestione:

- Polling
- La CPU periodicamente verifica lo stato dei task delle attività di I/O
- Interrupt-driven dove la CPU riceve un segnale di interrupt dal controller una volta che la task I/O viene completata
- La CPU riceve un interrupt dal controller (device o DMA<sup>1</sup>) una volta finito il task dell'I/O (con successo o in modo anomalo)
- La CPU svolge il lavoro effettivo di spostamento dei dati
- La CPU delega il lavoro a un controller DMA dedicato

### 1.3.3 Processori

La CPU preleva le istruzioni della memoria (esegue il **fetch**) e le esegue. La CPU si occupa poi di decodificare per determinare il tipo e gli operandi, eseguirli e poi prelevare, decodificare ed eseguire le successive. Poiché accedere alla memoria richiede molte risorse tutte le CPU contengono all'interno dei **registri** per memorizzare variabili importanti o risultati temporanei. Oltre ai registri i computer contengono il **program counter (PC)**, contenente l'indirizzo di memoria in cui si trova la successiva istruzione da seguire, dopodichè viene il PC viene aggiornato per posizionarsi sulla successiva.

Un altro registro è lo **stack pointer** che punta alla cima dello stack attuale. Un altro registro è il **program status word (PSW)** che contiene i bit di condizione, impostati da istruzioni di confronto, la priorità della CPU, la modalità (kernel o utente) e altri bit di controllo. Nelle architetture più moderne, vengono implementati più livelli di protezione, detti **protection rings**. Ogni ring aggiunge restrizioni sulle istruzioni eseguibili dalla CPU.

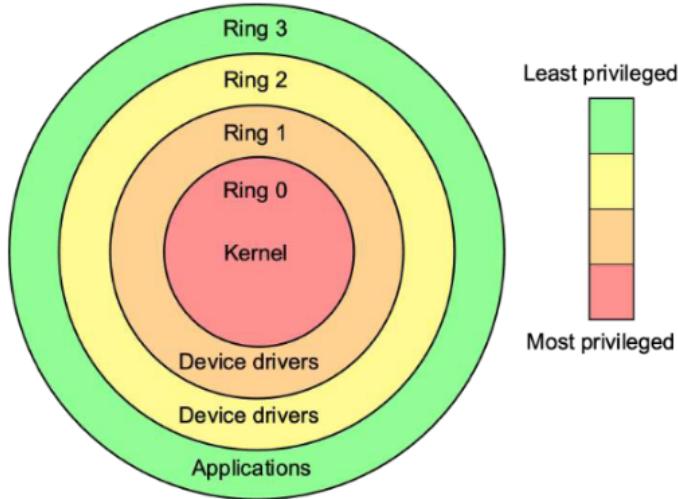


Figura 2: Protection Rings

Un modo semplice per avere una protezione alla memoria è quello di avere due registri dedicati:

- **Base** → contenente il primo address di partenza
- **Limite** → contenente l'ultimo address valido

Il sistema operativo carica i registri di base e limite all'avvio del programma mentre la CPU controlla che ogni indirizzo di memoria a cui fa riferimento il programma utente rientri tra i valori base e limite.

<sup>1</sup>Direct Memory Access

Per ottenere servizi dall'OS un programma utente deve fare una **system call (chiamata di sistema)** che entra nel kernel e richiama l'OS. Quando il lavoro è stato completato il controllo è restituito al programma utente.

L'istruzione TRAP cambia la modalità da utente a kernel e avvia l'OS.

- System call (software TRAP), ossia la richiesta di un servizio dell'OS, svolte in modo sincrono e innescate dai software
- Exception (fault), ossia la gestione di errori dovuti ad eventi inattesi, svolte in modo sincrono e innescate dai software
- Interrupt, ossia il completamento di una richiesta in attesa, svolte in modo asincrono e innescate dal hardware

Esistono sei categorie importanti di System call:

- **Gestione dei processi** include end, abort, load, execute, creazione e terminazione di processi, get/set attributi di processi, attesa, signal event, e allocazione di memoria libera. Quando un processo si interrompe o si interrompe, è necessario avviare o riprenderne un altro. Quando i processi si arrestano in modo anomalo, potrebbe essere necessario fornire core dump e/o altri strumenti diagnostici o di ripristino
- **Gestione dei file** crea file, elimina file, apri, chiudi, leggi, scrivi, riposiziona, ottieni attributi di file e imposta attributi di file. Queste operazioni possono essere supportate anche per directory e file ordinari. L'effettiva struttura della directory può essere implementata utilizzando file ordinari sul file system o tramite altri mezzi (ne parleremo più avanti)
- **Gestione dei dispositivi** include dispositivi di richiesta, dispositivi di rilascio, lettura, scrittura, riposizionamento, recupero/ impostazione degli attributi del dispositivo e collegamento o scollegamento logico dei dispositivi. I dispositivi possono essere fisici (ad es. unità disco) o virtuali/ astratti (ad es. file, partizioni e dischi RAM). Alcuni sistemi rappresentano i dispositivi come file speciali nel file system, in modo che l'accesso al "file" richieda il driver di dispositivo del sistema operativo appropriato. E.g la directory /dev su qualsiasi sistema UNIX
- **Informazioni di sistema** include le chiamate per ottenere/impostare l'ora, la data, i dati di sistema e gli attributi di processo, file o dispositivo. I sistemi possono anche fornire la possibilità di eseguire il dump della memoria in qualsiasi momento. Programmi a passo singolo che interrompono l'esecuzione dopo ogni istruzione e tracciano il funzionamento dei programmi (debug).
- **Comunicazione** include creazione/eliminazione di connessioni di comunicazione, invio/ricezione di messaggi, trasferimento di informazioni sullo stato e collegamento/scollegamento di dispositivi remoti. Esistono due modelli di comunicazione:
  - **scambio di messaggi**<sup>2</sup>, il modello di trasmissione dei messaggi deve supportare le chiamate a:
    - \* Identificare un processo remoto e/o un host con cui comunicare
    - \* Stabilire una connessione tra i due processi
    - \* Aprire e chiudere la connessione secondo necessità
    - \* Trasmettere messaggi lungo la connessione
    - \* Attendere i messaggi in arrivo, in stato di blocco o non blocco
    - \* Eliminare la connessione quando non è più necessaria
  - **memoria condivisa**<sup>3</sup>, il modello di memoria condivisa deve supportare le chiamate a:
    - \* Creare e accedere alla memoria condivisa tra processi (e thread)
    - \* Fornire meccanismi di blocco che limitano l'accesso simultaneo
    - \* Liberare memoria condivisa e/o alloCarla dinamicamente secondo necessità

<sup>2</sup>Nota: Più semplice e facile (in particolare per le comunicazioni tra computer) e generalmente appropriato per piccole quantità di dati

<sup>3</sup>Più veloce e generalmente l'approccio migliore in cui devono essere condivise grandi quantità di dati. Ideale quando la maggior parte dei processi deve leggere i dati anziché scriverli

### 1.3.4 Protezione delle system call

Fornisce meccanismi per controllare quali utenti/processi hanno accesso a quali risorse di sistema. Le chiamate di sistema consentono di regolare i meccanismi di accesso secondo necessità. Agli utenti non privilegiati può essere concesso temporaneamente un accesso elevato autorizzazioni in circostanze specifiche. Quando un programma utente richiede l'esecuzione di una syscall tramite un'API fornita dal sistema operativo, tale richiesta viene prima convertita in linguaggio macchina, per poi venir gestita dal **System call Handler**, il quale si occuperà di salvare lo stato precedente dei registri, i quali verranno poi alterati durante l'esecuzione della syscall, per poi essere ripristinati.

### 1.3.5 Passaggio di parametri

Esistono tre metodi usati per passare parametri :

- Salvare i parametri in registri
- Salvare parametri in **block** o **table** di un'area di memoria dedicata
- Parametri inseriti nello stack dal programma e estratti dallo stack dal sistema operativo (più complessi a causa dei diversi spazi degli indirizzi)

I metodi block e stack non limitano il numero o la lunghezza dei parametri passati

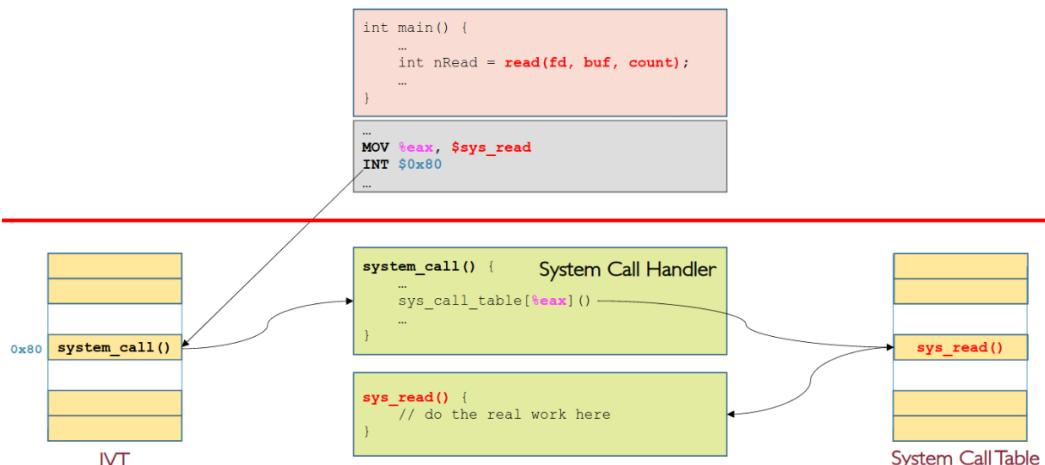


Figura 3: System call Handler

### 1.3.6 Timer, Istruzioni atomiche, Virtual Memory

Il timer è una funzionalità hardware per abilitare la scheduling della CPU. Nei sistemi multi-tasking, permette alla CPU di non essere monopolizzata da processi "egoistici". Il timer genera un interrupt e ad ogni sua interruzione, lo scheduler della CPU prende il sopravvento e decide quale processo da eseguire successivamente. Gli interrupt possono verificarsi in qualsiasi momento e interferire con i processi in esecuzione e l'OS deve essere in grado di sincronizzare le attività di cooperazione, simultanei processi, garantire che brevi sequenze di istruzioni (ad es. lettura-modifica-scrittura) vengano eseguite atomicamente da disabilitare gli interrupt prima della sequenza e riabilitarli successivamente.

La virtualizzazione della memoria è un'astrazione (dell'effettiva memoria principale fisica) e conferisce ad ogni processo l'illusione che la memoria fisica sia solo contigua spazio degli indirizzi (spazio degli indirizzi virtuali). Permette di eseguire programmi senza che vengano caricati interamente nella memoria principale. Può essere implementata sia in HW (**MMU**) che in SW (OS).

- **MMU**, associa gli indirizzi virtuali a quelli fisici tramite una tabella delle pagine gestita dal sistema operativo. Utilizza una cache denominata Translation Look-aside Buffer (TLB) con "mappature recenti" per ricerche più rapide. Il sistema operativo deve sapere quali pagine sono caricate nella memoria principale e quali su disco

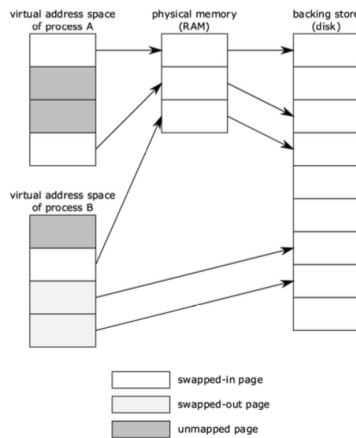


Figura 4: Virtual vs. Physical Address Space

- Il sistema operativo è responsabile della gestione degli spazi di indirizzi virtuali

Lo spazio degli indirizzi virtuali è tipicamente suddiviso in blocchi contigui della stessa dimensione (ad esempio, 4 KiB), chiamati pagine (**pages**). Ciascuna pagina che non sono caricate nella memoria principale vengono memorizzate su disco.

## 1.4 Progettazione e implementazione del sistema operativo

La struttura interna dei diversi sistemi operativi può variare notevolmente e gli obiettivi del sistema è facilitare l'usabilità rispetto al progettare/implementare. È fondamentale separare le politiche dai meccanismi della policy ovvero **cosa** sarà fatto e il meccanismo che indica **come** farlo. Il disaccoppiamento della logica della politica dal meccanismo sottostante è un principio di progettazione generale nell'informatica, in quanto migliora il sistema:

- flessibilità: l'aggiunta e la modifica delle politiche possono essere facilmente supportate
- riusabilità: i meccanismi esistenti possono essere riutilizzati per implementare nuove politiche
- stabilità: l'aggiunta di una nuova politica non destabilizza necessariamente il sistema

Le modifiche ai criteri possono essere facilmente regolate senza riscrivere il codice.

I primi sistemi operativi sviluppati in linguaggio assembly, e uno dei vantaggi era il controllo diretto sull'HW (alta efficienza) mentre uno svantaggio può essere il legame con uno specifico HW (bassa portabilità). Oggi abbiamo un mix di linguaggi e ai livelli più bassi troveremo assembly il corpo principale in C e i programmi di sistema in C, C++, linguaggi di scripting come PERL, Python, ecc.

Il sistema operativo dovrebbe essere suddiviso in sottosistemi separati, ciascuno con compiti, input/output e caratteristiche prestazionali accuratamente definiti. Esistono vari modi di strutturare un sistema operativo:

- **Struttura semplice**, dove non vi è alcun sottosistema e non vi è separazione tra kernel e user mode (esempio: il sistema MS-DOS). Semplice da implementare ma estremamente insicuro e poco rigido.
- **Struttura a Kernel Monolitico**, dove l'intero sistema operativo opera in kernel mode e solo i software utente lavorano in user mode (esempio: il sistema UNIX). Semplice da implementare ed efficiente, ma ancora poco sicuro e rigido
- **Struttura a livelli** (come MULTICS) dove l'OS è suddiviso in N livelli ed ogni livello L usa funzionalità implementate dal livello  $L_1$  ed espone nuove funzionalità al livello  $L + 1$ . Per via della struttura a livelli, il sistema è molto modulare, portabile e semplice da debuggere, rendendo tuttavia più complessa per la comunicazione tra di essi.
- **Struttura a Microkernel**, dove il kernel contiene solo le funzionalità di base, mentre tutte le altre funzionalità dell'OS e i programmi utente vengono eseguiti in user mode. Tale struttura porta ad una maggiore sicurezza, affidabilità ed estensibilità, ma anche ad un'efficienza ridotta.
- **Struttura a Moduli del Kernel caricabili (LKM)**, dove l'OS utilizza dei moduli tramite cui accedere alle funzionalità del kernel
- **Sistema a Kernel Ibrido**, dove viene utilizzato un approccio intermedio al kernel monolitico e al microkernel, ottenendo i vantaggi di entrambi gli approcci

## 2 Gestione dei Processi e thread

Un programma è un file eseguibile che risiede nella memoria persistente (ad es. disco) e contiene solo l'insieme di istruzioni necessarie per eseguire un lavoro specifico. Un processo è l'astrazione del sistema operativo di un programma in esecuzione (unità di esecuzione). Il processo è dinamico, mentre un programma è statico (solo codice e dati). Diversi processi possono eseguire lo stesso programma (ad esempio, multiple istanze di Google Chrome) ma ognuna ha il proprio stato. Un processo esegue un'istruzione alla volta, in sequenza.

Il sistema operativo fornisce la stessa quantità di spazio di indirizzi virtuali a ciascun processo. Lo spazio degli indirizzi virtuali è un'astrazione dello spazio degli indirizzi della memoria fisica. L'intervallo di indirizzi virtuali validi che un processo può generare dipende dalla macchina. Avremo perciò:

- **Text** contenente le istruzioni dell'eseguibile
- **Data** Variabili globali e statiche inizializzate
- **BSS** variabile globale e statica (non inizializzata o inizializzata a 0)
- **Stack** Struttura LIFO utilizzata per memorizzare tutti i dati necessari a una chiamata di funzione (stack frame)
- **Heap** usata per l'allocazione dinamica

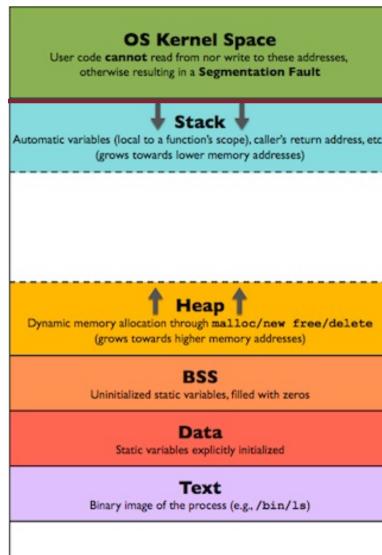


Figura 5: Virtual Address Space Layout

Sullo stack sono definite due operazioni: **push** e **pop**. Il push viene usato per inserire degli elementi, mentre pop per rimuoverli. Usa un registro dedicato (e.g. `esp`) il cui contenuto è l'indirizzo nella memoria principale della parte superiore dello stack. La memoria dello stack cresce convenzionalmente dall'alto verso il basso, cioè dagli indirizzi di memoria più alti a quelli più bassi. Ogni funzione utilizza una porzione dello stack, chiamata **stack frame** e in ogni momento possono esistere contemporaneamente più stack frame, a causa di diverse chiamate di funzioni nidificate, ma solo una è attiva. Lo stack frame per ogni funzione è diviso in tre parti:

- parametri della funzione + indirizzo di ritorno
- back-pointer allo stack frame precedente
- variabili locali

Il primo è impostato dal chiamante, il secondo e il terzo vengono impostati dal chiamato. Il puntatore `esp` viene sempre aggiornato man mano che lo stack cresce ed è difficile per il chiamato accedere ai parametri effettivi senza un riferimento fisso nello stack. Per risolvere questo problema invece di utilizzare un singolo puntatore in cima allo stack usa un puntatore aggiuntivo alla parte inferiore (base) dello stack (`%ebp`) lascia che `esp` sia libero di cambiare tra diverse chiamate di funzione, mentre tiene `%ebp` fissato all'interno di ogni stack frame.



Figura 6: Function Parameters + Return

## 2.1 Stato di esecuzione del processo

In ogni momento un processo può trovarsi in uno dei seguenti cinque stati:

- **New**, il processo appena avviato
- **Ready**, il processo è pronto per essere eseguito ma attende di essere programmato sulla CPU
- **Running**, il processo sta effettivamente eseguendo istruzioni sulla CPU
- **Waiting**, il processo è sospeso in attesa che una risorsa sia disponibile o un evento da completare/verificare (ad es. input da tastiera, accesso al disco, timer, ecc.)
- **Terminated**, il processo è terminato e il sistema operativo può distruggerlo

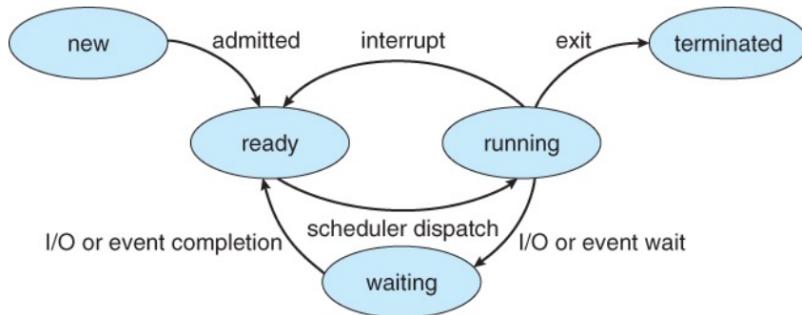


Figura 7: Process Execution State Diagram

Durante l'esecuzione, il processo passa da uno stato all'altro in base alle azioni del programma (ad esempio, chiamate di sistema) come azioni del sistema operativo (ad es. scheduling) o azioni esterne (ad es. interruzioni).

La maggior parte delle chiamate di sistema (ad esempio quelle di I/O) sono bloccanti. Il processo chiamante (spazio utente) non può fare nulla fino al ritorno della chiamata di sistema.

Il sistema operativo (spazio del kernel) si occupa di:

- impostare il processo corrente in uno stato di attesa (ovvero, in attesa del ritorno della chiamata di sistema)
- pianifica un diverso processo pronto per evitare che la CPU sia inattiva

Una volta che la chiamata di sistema ritorna, il processo precedentemente bloccato è pronto per essere schedulato nuovamente per l'esecuzione.<sup>4</sup>

Lo stato del processo è costituito da:

- il codice del programma in esecuzione
- i dati statici del programma in esecuzione
- il program counter (PC) che indica la prossima istruzione da eseguire
- Registri CPU
- la catena di chiamate del programma (stack) insieme ai puntatori di frame e stack
- lo spazio per l'allocazione dinamica della memoria (heap) e al suo puntatore, le risorse in uso (ad es. file aperti)
- lo stato di esecuzione del processo (pronto, in esecuzione, ecc.)

## 2.2 Process Control Block (PCB)

Il **PCB** (o detto anche Tabella dei Processi), è la struttura dati principale utilizzata dal sistema operativo per tenere traccia di qualsiasi processo. Il PCB tiene traccia dello stato di esecuzione e della posizione di un processo. Il sistema operativo assegna un nuovo PCB alla creazione di un processo e lo inserisce in una coda di stato e viene deallocated non appena termina il processo associato.

Associata a ogni classe di I/O c'è una posizione (solitamente in una collocazione fissa vicino alla base della memoria) chiamata **vettore di interrupt**<sup>5</sup>. Contiene l'indirizzo della procedura di servizio dell'interrupt. Ad ogni interrupt il computer salta all'indirizzo specificato nel vettore dell'interrupt. Il PCB contiene:

- Stato del processo pronto, in attesa, in esecuzione, ecc.
- Numero di processo (ovvero identificatore univoco)
- Program Counter (PC) + Stack Pointer (SP) + registri di uso generale
- Informazioni sullo scheduling della CPU priorità e puntatori alle code di stato
- Informazioni sulla gestione della memoria tabelle delle pagine
- Informazioni sull'account, ossia il tempo utilizzato dalla CPU del kernel e dell'utente, stato I/O del proprietario
- Elenco dei file aperti

---

<sup>4</sup>NOTA: l'intero sistema non è bloccato, solo il processo che ha richiesto la chiamata bloccata è!

<sup>5</sup>In informatica, un interrupt vector (vettore delle interruzioni) è un indirizzo di memoria del gestore di interrupt, oppure un indice ad un array, chiamato interrupt vector table, il quale può essere implementato tramite una dispatch table. La tabella degli interrupt vector contiene gli indirizzi di memoria dei gestori di interrupt. Quando si genera una interruzione, il processore salva il suo stato di esecuzione con il context switch, ed inizia l'esecuzione del gestore di interruzione all'interrupt vector (questo procedimento avviene quando l'interruzione ha carattere sincrono). Infatti vi è una wait instruction (istruzione d'attesa) che obbliga la CPU ad effettuare cicli vuoti fino a che non arriva il prossimo interrupt. Nel caso in cui si dovesse verificare una interruzione asincrona il programma provvede a lanciare un interrupt con la richiesta di I/O e nell'attesa del dato continua ad effettuare operazioni logico-aritmetiche. Quando arriverà nel momento in cui gli servirà il risultato si fermerà e inizierà ad aspettare. L'I/O asincrono serve a far sì che alcuni programmi possano anticipare la richiesta di un dato così nel caso in cui esso dovesse servire lo possono già utilizzare. Fonte: Wikipedia

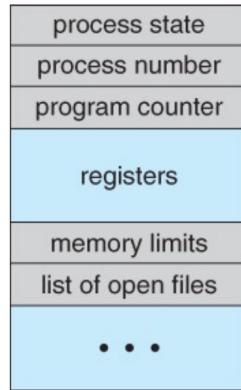


Figura 8: Process Execution State Diagram

## 2.3 Creazione di processi

I processi possono creare altri processi tramite specifiche chiamate di sistema. Il processo creatore è chiamato genitore del nuovo processo, chiamato figlio. Il genitore condivide risorse e privilegi con i suoi figli. Un genitore può aspettare che un figlio finisca o continuare in parallelo.

A ogni processo viene assegnato un identificatore intero (noto anche come identificatore di processo o PID) e per ogni processo viene memorizzato anche il PID genitore (PPID).

Sui tipici sistemi UNIX lo scheduler del processo è denominato `sched` e riceve il PID 0. La prima cosa che fa all'avvio del sistema è lanciare `init`, che dà a quel processo il PID 1. Successivamente `init` avvia tutti i daemons di sistema e i login degli utenti e diventa il genitore ultimo di tutti gli altri processi. I processi vengono creati attraverso la chiamata di sistema `fork()`.

Tutti i processi sono uguali. L'unico indizio di gerarchia dei processi è che quando viene creato un processo, al genitore viene dato uno speciale gettone o token (detto **handle**) che può controllare il figlio invalidando la gerarchia.

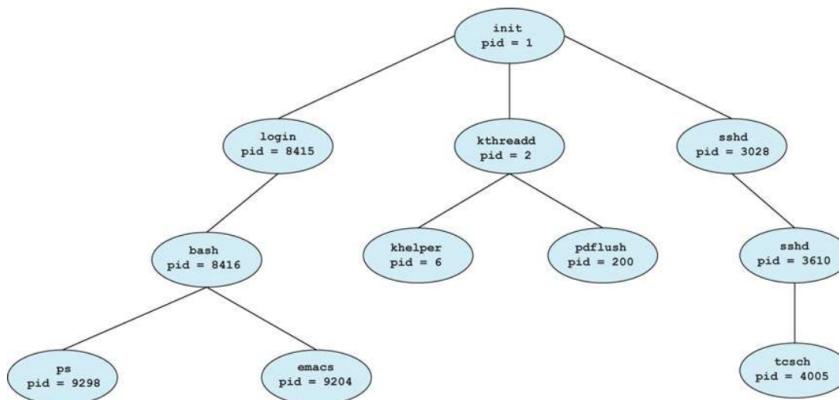


Figura 9: Process Creation: UNIX/Linux

### 2.3.1 Risorse del padre vs figlio

Abbiamo due possibilità per lo spazio degli indirizzi del figlio rispetto al genitore:

- Il bambino può essere un duplciato esatto del genitore, condividendo lo stesso programma e gli stessi segmenti di dati in memoria
- Ciascuno avrà il proprio PCB, inclusi program counter, registri e PID
- Questo è il comportamento della chiamata di sistema `fork()` in UNIX
- Il processo figlio può avere un nuovo programma caricato nel suo spazio degli indirizzi, con tutti i nuovi segmenti di codice e dati
- In Windows le chiamate di sistema saranno create con il comando `spawn()`
- I sistemi UNIX implementano ciò come secondo passaggio, utilizzando la chiamata di sistema `exec.`

### 2.3.2 Esecuzione del padre vs figlio

Abbiamo due opzioni per il processo genitore dopo aver creato il figlio:

- Attendere che il processo figlio termini prima di procedere emettendo un `wait` chiamata di sistema, per un figlio specifico o per qualsiasi figlio
- Oppure eseguire contemporaneamente al figlio, continuando l'elaborazione senza essere bloccato (quando una shell UNIX esegue un processo come attività in background utilizzando "&")

Consideriamo ora questo esempio:

```
#include <sys/types.h>
#include <stdio.h>
#include <unistd.h>
int main(){
    pid_t pid;
    /* fork a child process */
    pid = \texttt{fork()};
    if (pid < 0) {
        /* if the returned PID is < 0, an error occurred */
        fprintf(stderr, "Fork Failed");
        exit(-1);
    }
    else if (pid == 0) {
        /* execute child process code */
        execlp("/bin/ls", "ls", NULL);
    }
    else {
        /* execute parent process code */
        \dots
        /* wait for child to terminate */
        wait(NULL);
        printf("Child terminated");
        exit(0);
    }
}
```



Figura 10: Albero decisionale del primo esempio

Consideriamo ora quest'altro esempio:

```

int pid = \texttt{fork()};
if(pid == 0) {           // A's child (B)
    pid = \texttt{fork()};
    if(pid == 0) {         // B's child (C)
        \dots
        execp(\dots);
    }
    else { // B
        \dots
    }
}
else { // A
    \dots
}
  
```



Figura 11: Albero decisionale del secondo esempio

Consideriamo ora questo esempio:

```
int pid = \texttt{fork()};
if(pid == 0) { // A's child (B)
    \dots
    execp(\dots);
}
else { // A
    pid = \texttt{fork()};
    if(pid == 0) { // A's child (C)
        \dots
        execp(\dots);
    }
}
```

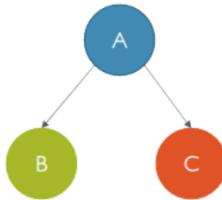


Figura 12: Albero decisionale del secondo esempio

Consideriamo ora quest'altro esempio:

```
for( int i=0;i<n; i++) {
    if(\texttt{fork()} == 0) { // A0's child
        \dots
        execp(\dots);
    }
}
// back in the parent A0
// wait for all children to terminate

for( int i=0;i<n; i++) {
    wait(NULL);
}
```



Figura 13: Albero decisionale del secondo esempio

Riepilogo delle chiamate di sistema viste:

- **fork()** genera un nuovo processo figlio come copia esatta del genitore
- **execp** sostituisce il programma del processo corrente con il inserire il programma denominato
- **sleep** sospende l'esecuzione per un certo numero di secondi
- **wait/waitpid** attende che uno o più processi specifici terminino l'esecuzione

## 2.4 Terminazione di processi

I processi possono richiedere la propria terminazione effettuando la chiamata di sistema `exit`, tipicamente restituendo un *int*. Questo *int* viene passato al genitore se sta eseguendo un'attesa. Solitamente è 0 in caso di completamento con successo e un diverso da zero in caso di problemi.

I processi possono anche essere terminati dal sistema per una serie di motivi:

- L'incapacità del sistema di fornire le risorse di sistema necessarie
- In risposta a un comando `kill` o ad un altro interrupt di processo non gestito
- Uscita normale (condizione volontaria)
- Uscita su errore (condizione volontaria)
- Terminato da un altro processo (condizione involontaria)

Un genitore può uccidere i suoi figli se il compito loro assegnato non è più necessario, il sistema però può consentire (o meno) al figlio di continuare senza un genitore. Sui sistemi UNIX, i processi orfani sono generalmente ereditati da `init`, che poi li uccide.

Quando un processo termina, tutte le sue risorse di sistema vengono liberate, i file aperti svuotati e chiusi, ecc. Lo stato di terminazione del processo dei tempi di esecuzione vengono restituiti al genitore se è in attesa che il figlio termini o `init` se il processo diventa orfano.

I processi sono detti **zombie** se stanno tentando di terminare ma non possono perché il loro genitore non li sta aspettando oppure ereditati da `init` come orfani e uccisi.

## 2.5 Scheduling dei processi

Gli obiettivi principali del sistema di schedulazione dei processi:

- mantenere la CPU sempre occupata
- fornire tempi di risposta "accettabili" per tutti i programmi, in particolare per quelli interattivi

Lo scheduler del processo deve soddisfare questi obiettivi implementando politiche adeguate per lo scambio dei processi dentro e fuori la CPU. Questi obiettivi però possono essere contrastanti, e ogni volta che il sistema operativo interviene per scambiare i processi, impiega tempo sulla CPU per farlo, che viene quindi "perso" dal fare qualsiasi lavoro produttivo utile.

Il sistema operativo mantiene i PCB di tutti i processi nelle code di stato. C'è una coda per ciascuno dei cinque stati in cui può trovarsi un processo e una per ogni dispositivo I/O (dove i processi attendono che un dispositivo diventi disponibile o consegni i dati). Quando il sistema operativo cambia lo stato di un processo, il PCB viene scollegato dalla coda corrente e spostato in quella nuova. Il sistema operativo può utilizzare criteri diversi per gestire ciascuna coda di stato.

La coda di esecuzione è vincolata dal numero di core disponibili sul sistema. Ogni volta, è possibile eseguire un solo processo su una CPU. Non esiste un limite teorico al numero di processi negli stati `new/ready/waiting/terminated`.

Esistono diversi scheduler (che in un sistema efficiente utilizza un mix tra questi):

- Uno **scheduler a lungo termine** viene eseguito raramente ed è tipico di un sistema batch o di un sistema molto carico
- Uno **scheduler a breve termine** viene eseguito molto frequentemente (circa ogni 100 millisecondi) e deve sostituire molto rapidamente un processo dalla CPU e inserirne un altro
- Alcuni sistemi utilizzano anche uno **scheduler a medio termine**: quando i carichi del sistema diventano elevati, questo scheduler consente ai lavori più piccoli e più veloci di terminare rapidamente e liberare il sistema

### 2.5.1 Il Context Switch

Il **Context Switch** procedura utilizzata dalla CPU per sospendere il processo attualmente in esecuzione al fine di eseguirne uno pronto. È un'operazione molto costosa in quanto l'arresto del processo in corso implica il salvataggio di tutto il suo stato interno (PC, SP, altri registri, ecc.) al suo PCB e l'avvio di un processo pronto consiste nel caricare tutto il suo stato interno (PC, SP, altri registri, ecc.) dal suo PCB.

Il Context Switch si usa quando ci sono delle chiamate TRAP<sup>6</sup> come system call, eccezioni, o interruzioni hardware. Ogni volta che arriva una TRAP la CPU ha il compito di creare uno stato di salvataggio dello stato corrente dei processi attivi. Passare alla modalità kernel per gestire l'interrupt ed eseguire un ripristino dello stato del processo interrotto.

Per evitare che i processi legati alla CPU monopolizzino la CPU, anche il Context Switch viene attivato tramite interrupt del timer hardware (**slice**). Lo **slice** ha quantità massima di tempo tra due cambi di contesto per garantire che si verifichi almeno un cambio di contesto. Può accadere più frequentemente di così (ad esempio, a causa di richieste di I/O). Lo slice è facilmente implementabile in hardware tramite interrupt timer. È un meccanismo utilizzato dai moderni sistemi operativi multitasking time-sharing per aumentare la reattività del sistema (pseudo-parallelismo). <sup>7</sup>

### Il tempo nel Context Switch

Il tempo impiegato per completare un cambio di contesto è solo tempo di CPU sprecato. Un intervallo di tempo inferiore comporta cambi di contesto più frequenti. Un intervallo di tempo maggiore comporta cambi di contesto meno frequenti. Ha anche il compito di minimizzare lo spreco di tempo della CPU, massimizzando quindi l'utilizzo e la reattività della CPU.

## 2.6 Communicazione dei processi

I processi possono essere **indipendenti** o **cooperanti**:

- Processi **indipendenti** operano contemporaneamente su un sistema e possono né influenzare né essere influenzati da altri processi
- I processi **cooperanti** possono influenzare o essere influenzati da altri processi al fine di raggiungere un compito comune

Possono esserci diversi processi che richiedono l'accesso allo stesso file (ad es pipeline). Abbiamo bisogno di questa cooperazione tra processi e ci sono diverse caratteristiche importanti. Abbiamo la **velocità** di calcolo è un problema che può essere risolta più velocemente se può essere suddiviso in sotto-attività da risolvere simultaneamente. La **modularità** definisce l'architettura più efficiente può essere quella di scomporre un sistema in moduli cooperanti. Infine abbiamo la **convenienza** infatti anche un singolo utente può essere multitasking, come modificare, compilare, stampare ed eseguire lo stesso codice in finestre diverse. Ci sono due modi possibili per comunicare tra processi cooperanti:

- **Memoria condivisa**<sup>8</sup>
  - Più complicato da configurare e non funziona altrettanto bene su più computer
  - Più veloce una volta configurato, poiché non sono necessarie chiamate di sistema
  - Preferibile quando (una grande quantità) di informazioni deve essere condivisa sullo stesso computer
- **Scambio di messaggi**
  - Più semplice da configurare e funziona bene su più computer
  - Più lento in quanto richiede chiamate di sistema (syscall) per ogni trasferimento di messaggi

<sup>6</sup>richiesta di un servizio dell'OS, svolte in modo sincrono e innestate dai software

<sup>7</sup>Diverso invece quando si parla di **multiprocessore** che hanno due o più CPU che condividono la stessa memoria fisica, vero parallelismo hardware

<sup>8</sup>Per far sì che un processo possa richiedere un'area di memoria da condividere con altri processi c'è, ovviamente, necessità di effettuare una chiamata di sistema (a livello utente, non sarebbe possibile completare quest'operazione). Tuttavia, una volta "pagato questo prezzo iniziale", le interazioni successive tra processi che condividono un'area di memoria può avvenire senza ulteriori chiamate di sistema. **Fonete: il Prof**

- Preferibile quando la quantità e/o la frequenza dei trasferimenti di dati è piccola o quando sono multipli i computer sono coinvolti

La memoria da condividere è inizialmente all'interno dello spazio degli indirizzi di un particolare processo e bisogna effettuare chiamate di sistema per rendere quella memoria pubblicamente disponibile ad altri processi. Altri processi devono effettuare le proprie chiamate di sistema per collegare la memoria condivisa al proprio spazio degli indirizzi.

Il **Message Passing Systems** deve supportare almeno chiamate di sistema per l'invio e la ricezione di messaggi e stabilire un collegamento di comunicazione tra i cooperanti processi prima che i messaggi possano essere inviati.

Esistono però tre problemi da risolvere nello scambio di messaggi:

- Comunicazione diretta o indiretta (es. naming)

- Nella **Comunicazione diretta** il mittente deve conoscere il nome del destinatario a cui desidera inviare un messaggio. Deve conoscere anche il collegamento uno a uno tra ogni coppia mittente-destinatario per la comunicazione simmetrica e il destinatario deve conoscere anche il nome del mittente.
- La **Comunicazione indiretta** utilizza mailbox o porte condivise dove più processi possono condividere la stessa mailbox o porta. Solo un processo può leggere un determinato messaggio in una mailbox. Il sistema operativo deve fornire chiamate di sistema per creare ed eliminare mailbox e per inviare e ricevere messaggi da/per mailbox.

- Comunicazione sincrona o asincrona
- Buffering automatico o esplicito

Scegliere se utilizzare o meno una queue per i messaggi:

- **Zero capacity**, dove i messaggi non possono essere salvati in una queue, dunque il mittente rimane bloccato finché il destinatario non riceve il messaggio.
- **Bounded capacity**, dove vi è una queue di capienza predefinita, permettendo ai mittenti di non bloccarsi a meno che la queue non sia piena.
- **Unbounded capacity**, dove la queue ha teoricamente capienza infinita, implicando che i mittenti non debbano mai bloccarsi.

## 2.7 Scheduling dei processi

Definiamo come CPU burst lo stato in cui un processo è eseguito dalla CPU, mentre definiamo come I/O burst lo stato in cui un processo è in attesa che i dati vengano trasferiti dentro o fuori dal sistema. In generale, ogni processo alterna costantemente tra CPU burst e I/O burst.

Lo scheduling dei processi è una **politica** che stabilisce quali processi devono essere eseguiti dalla CPU. Lo scheduler della CPU è un processo che, a seconda di una policy di scheduling stabilita, si occupa di selezionare un processo dalla ready queue da eseguire in ogni momento in cui la CPU è inattiva. La struttura dati utilizzata per la ready queue e l'algoritmo usato per selezionare il processo successivo è basato su una queue FIFO (First In, First Out).

- Concetti di base dello scheduling. Quasi tutti i programmi hanno alcuni cicli alternati di calcoli della CPU e attesa di I/O. Anche un semplice recupero dalla memoria principale richiede molto tempo rispetto alla velocità della CPU. (Consideriamo per ora la multiprogrammazione sui sistemi con un unico processore)
- Algoritmi di scheduling
- Concetti avanzati di scheduling.

Ogni volta che la CPU diventa inattiva lo **scheduler** della CPU seleziona un altro processo dalla coda pronta per l'esecuzione successiva. La struttura dei dati per la coda pronta e l'algoritmo utilizzato per selezionare il processo successivo non sono però necessariamente basati su una coda FIFO. Le **decisioni di scheduling** della CPU avvengono in una delle quattro condizioni:

1. Quando un processo passa dallo stato in **esecuzione** allo stato di **attesa**
2. Quando un processo passa dallo stato in **esecuzione** allo stato **pronto**
3. Quando un processo passa dallo stato di **attesa** allo stato **pronto**
4. Quando un processo viene **creato** o **terminato**

Nel 1° e il 4° caso è necessario selezionare un nuovo processo, mentre nel 2° e 3° caso si continua con il processo corrente o bisogna selezionarne uno nuovo.

## 2.8 Scheduling Preemptive e Non-Preemptive

Un algoritmo di scheduling **non-preemptive** assegna la CPU a un processo e non lo interrompe finché non termina o finché non rilascia volontariamente la CPU. Ciò significa che un processo che richiede una quantità considerevole di tempo della CPU può bloccare altri processi che sono in attesa di esecuzione.

Un algoritmo di scheduling **preemptive**<sup>9</sup>, al contrario, consente al sistema operativo di interrompere l'esecuzione di un processo in qualsiasi momento per consentire l'esecuzione di un altro processo in coda. Questo significa che i processi con un tempo di esecuzione più breve possono essere eseguiti più rapidamente, anche se ci sono altri processi in attesa di esecuzione.

### 2.8.1 Problemi

Preemption potrebbe causare problemi se si verifica mentre il kernel è impegnato nell'implementazione di una system call (ad esempio, l'aggiornamento dei dati critici del kernel strutture) oppure due processi condividono dati, uno può essere interrotto durante l'aggiornamento di strutture dati condivise. Ci sono però possibili **soluzioni**, si può fare in modo che il processo attenda finché la chiamata di sistema non è stata completata o bloccata prima di concedere la preemption (problematico per i sistemi in tempo reale, poiché la risposta in tempo reale non può più essere garantita) oppure disabilitare gli interrupt prima di entrare nella sezione del codice critico e riabilitarli subito dopo (dovrebbe essere fatto solo in rare situazioni e solo su parti di codice molto brevi che finiranno rapidamente).

## 2.9 Il Dispatcher

Il dispatcher è il modulo che dà il controllo della CPU al processo selezionato dallo scheduler. Esso legge le richieste di lavoro in arrivo e dopo aver esaminato la richiesta, sceglie un **thread worker inattivo** (cioè bloccato) e gli fa gestire la richiesta scrivendo eventualmente un puntatore al messaggio all'interno di una parola sociale associata a ogni thread. Il dispatcher risveglia così il worker dormiente facendolo passare dallo stato di "bloccato" a quello di "ready".

Ha il compito di:

- Cambio di contesto
- Passaggio alla modalità utente
- Saltare alla posizione corretta nel programma appena caricato
- Il dispatcher viene eseguito su ogni cambio di contesto quindi il tempo che esso consuma (latenza di invio) deve essere il più breve possibile

## 2.10 Definizioni utili

10

- Orario di arrivo: ora in cui il processo arriva nella coda pronta
- Tempo di completamento (Arrival Time): momento in cui il processo completa la sua esecuzione
- Burst Time: tempo richiesto da un processo per l'esecuzione della CPU

<sup>9</sup>Nota: uno scheduler preemptive viene comunque attivato dalle condizioni 1 e 4, poiché in tali casi deve comunque essere selezionato un processo.

<sup>10</sup>NOTA: Il tempo di attesa I/O non è considerato qui!

- TurnaroundTime: differenza di tempo tra il completamento e l'orario di arrivo
- Waiting Time: differenza di tempo tra il tempo di turnaround e il burst time

$$T^{arrival} = \text{arrival time}$$

$$T^{completion} = \text{completion time}$$

$$T^{burst} = \text{burst time}$$

$$T^{turnaround} = \text{turnaround time} = T^{completion} - T^{arrival}$$

$$T^{waiting} = \text{waiting time} = T^{turnaround} - T^{burst}$$

Figura 14

## 2.11 Criteri/metriche di scheduling

Esistono diversi criteri da considerare quando si tenta di selezionare l'algoritmo di scheduling "migliore", e deve prendere in considerazione:

- Utilizzo della CPU, ossia la percentuale di tempo in cui la CPU è occupata. Idealmente la CPU sarebbe occupata il 100% del tempo, in modo da sprecare 0 cicli CPU. Su un sistema reale, l'utilizzo della CPU dovrebbe variare dal 40% (con carico leggero) al 90% (con carico pesante).
- Throughput, ossia il numero di processi completati in un'unità di tempo.
- Turnaround time, ossia il tempo necessario per il completamento di un particolare processo, dall'invio al completamento. Include tutto il tempo di attesa.
- Waiting time, ossia il tempo necessario ai processi trascorrono nella coda pronta aspettando il proprio turno per salire sulla CPU. I processi nello stato di attesa non sono sotto il controllo dello scheduler della CPU (semplicemente non sono pronti per essere eseguiti).<sup>11</sup>
- Response time, ossia il tempo impiegato dall'emissione di un comando all'inizio di una risposta a tale comando. Si usa principalmente, per processi interattivi.

Idealmente, si sceglie uno scheduler della CPU che ottimizzi tutte le metriche contemporaneamente infatti quanto è stato detto sopra è impossibile ed è necessario un compromesso. L'obiettivo era di scegliere un algoritmo di pianificazione in base alla sua capacità di soddisfare una data politica.

L'obiettivo dello scheduling è di ridurre al minimo il tempo medio di risposta<sup>12</sup>:

- Minimizzare il **tempo medio** di risposta
- Fornire l'output all'utente il più rapidamente possibile
- Ridurre al minimo il tempo di risposta massimo
- Limite nel caso peggiore
- Ridurre al **minimo** la varianza del tempo di risposta
- Gli utenti accettano di più un sistema coerente e prevedibile piuttosto che uno incoerente

---

<sup>11</sup>Da non confondere con lo waiting state

<sup>12</sup>Tipico dei sistemi interattivi

Massimizzare il throughput significa<sup>13</sup>:

- Riduzione al minimo dell'overhead (cambio di contesto del sistema operativo)
- Utilizzo efficiente delle risorse di sistema (CPU e dispositivi I/O)
- Ridurre al minimo i tempi di attesa
- Dare a ogni processo la stessa quantità di tempo sulla CPU
- Potrebbe aumentare il tempo medio di risposta

Per le politiche di scheduling assumiamo che:

- Vi è un singolo processo per utente
- I processi sono indipendenti tra di loro, dunque non vi è comunicazione
- Ogni processo è costituito da un singolo thread

## 2.12 Algoritmi di Scheduling

### 2.12.1 First-Come-First-Serve (FCFS)

Il FCFS ha solo una coda FIFO, lo scheduler esegue i lavori fino al completamento nell'ordine di arrivo ed entra in azione solo quando il lavoro attualmente in esecuzione lo richiede un'operazione di I/O (o termina la sua esecuzione). Un lavoro può continuare a utilizzare la CPU a tempo indeterminato (ovvero fino a quando non si blocca) per questo è di tipo Non-preemptive.

---

<sup>13</sup>Tipico dei sistemi batch. I **sistemi batch** (o "batch processing systems" in inglese) sono un tipo di sistema informatico che esegue una serie di lavori o processi in una sequenza predefinita, senza interazione diretta con l'utente o l'operatore del sistema. In un sistema batch, i lavori vengono raccolti in un'area di elaborazione chiamata "coda di lavoro" o "batch queue". Il sistema elabora poi i lavori uno per uno, seguendo un ordine predefinito, e può eseguirli in background, senza richiedere l'input dell'utente durante l'esecuzione. Fonte: Wikipedia

### 2.12.2 Esempi

- Consideriamo la seguente coda di processi:

New 

Order	Job	CPU burst (time units)
1	A	5
2	B	2
3	C	3

Figura 15

- I processi vengono creati allo stesso istante, dunque l'arrival time per tutti i processi è l'istante 0. Successivamente, tutti e tre i processi vengono messi nella ready queue

New 

Ready 

Waiting

Running

Order	Job	CPU burst (time units)
1	A	5
2	B	2
3	C	3

Figura 16

- Di seguito, lo scheduling FCFS seleziona il primo processo della ready queue, ossia il processo A, spostandolo in stato di ready e completando la sua esecuzione

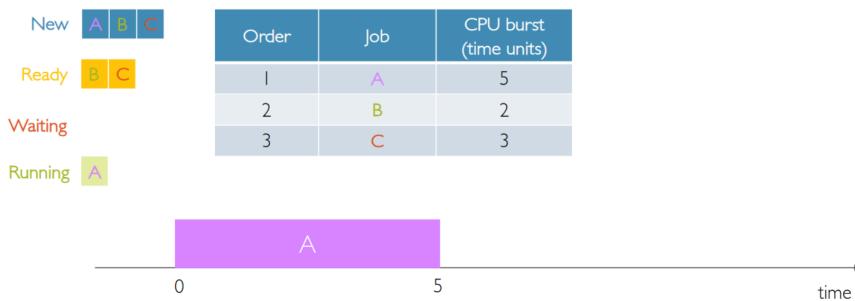


Figura 17

- Una volta terminato il processo A, lo scheduler ripeterà le stesse operazioni fino a che tutti i processi non saranno terminati

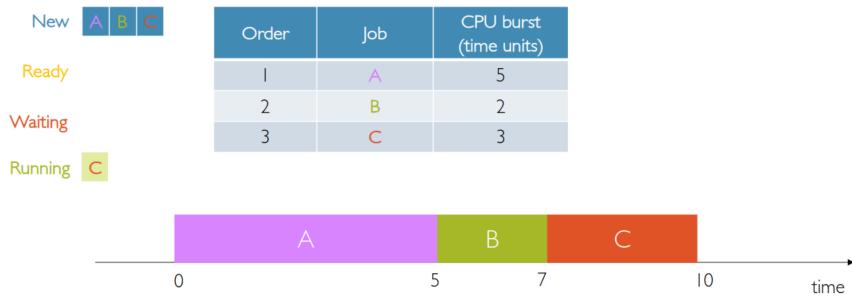


Figura 18

- Una volta terminati tutti i processi, calcoliamo il waiting time medio:
  - Per il processo A si ha:
$$T_{waiting} = T_{turnaround} - T_{burst} = T_{completion} - T_{arrival} - T_{burst} = 5 - 0 - 5 = 0$$
- Per il processo B si ha:
$$T_{waiting} = T_{turnaround} - T_{burst} = T_{completion} - T_{arrival} - T_{burst} = 7 - 0 - 2 = 5$$
- Per il processo C si ha:
$$T_{waiting} = T_{turnaround} - T_{burst} = T_{completion} - T_{arrival} - T_{burst} = 10 - 0 - 3 = 7$$
- Il waiting time medio, quindi, sarà:

$$\bar{T}_{waiting} = \frac{1}{n} \sum_{i=0}^n T_i^{waiting} = \frac{0+5+7}{3} = \frac{12}{3} = 4$$

2. Nel seguente scenario, utilizzando sempre uno scheduling FCFS, si ha:

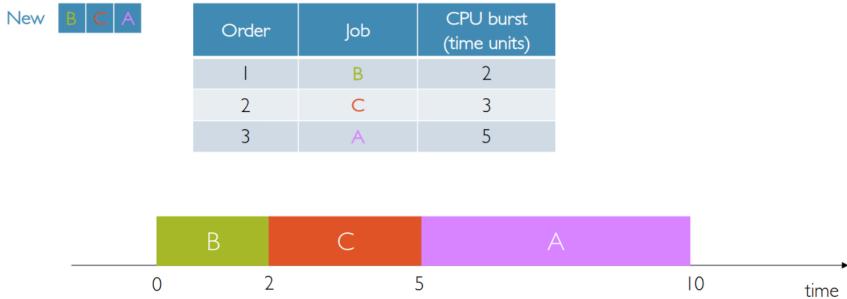


Figura 19

- Il waiting time medio, quindi, sarà:

$$\bar{T}_{waiting} = \frac{1}{n} \sum_{i=0}^n T_i^{waiting} = \frac{0+5+2}{3} = \frac{7}{3} \simeq 2.3$$

3. Consideriamo il seguente scenario:

New	A   B   C	Order	Job	CPU burst (time units)
Ready	A   B   C	1	A	5
		2	B	2
		3	C	3

Figura 20

- Supponiamo che dopo 2 unità temporali il processo A esegua una richiesta I/O, entrando quindi in stato di waiting, la quale verrà completata dopo un istante. Attenzione: poiché il tempo in attesa I/O non viene calcolato nel waiting time, sarà necessario sottrarre un istante dal waiting time del processo A.

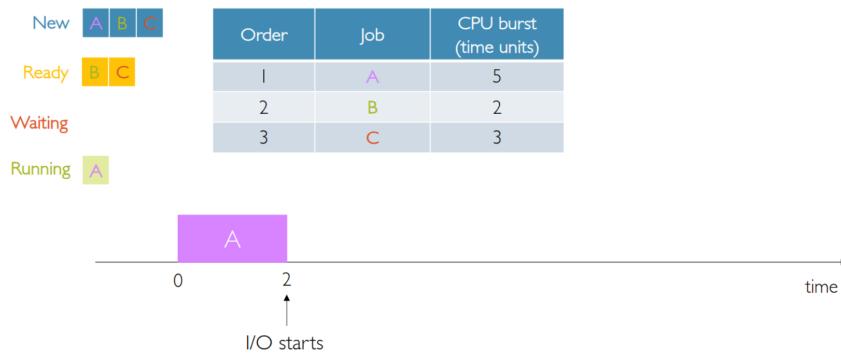


Figura 21

- Una volta entrato in stato di waiting, lo scheduler selezionerà il processo seguente nella coda, ossia il processo B, portandolo a termine

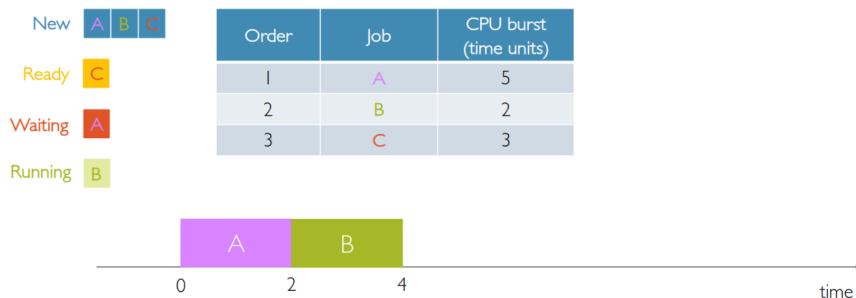


Figura 22

- Una volta terminata l'esecuzione del processo B, il processo A verrà selezionato per riprendere l'esecuzione. Attenzione: viene selezionato il processo A e non il processo C poiché l'algoritmo FCFS utilizza una queue basata sull'arrival time dei processi

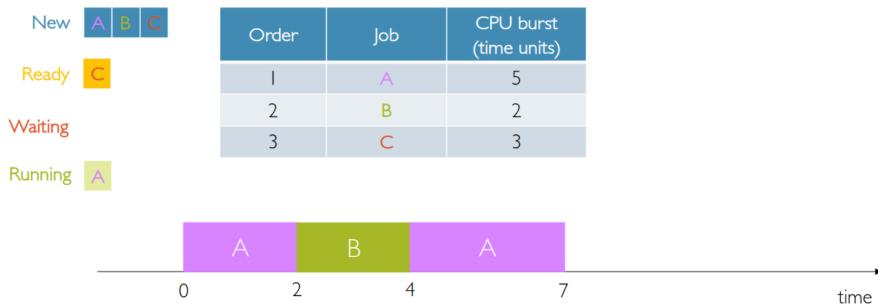


Figura 23

- Infine, il waiting time medio sarà:

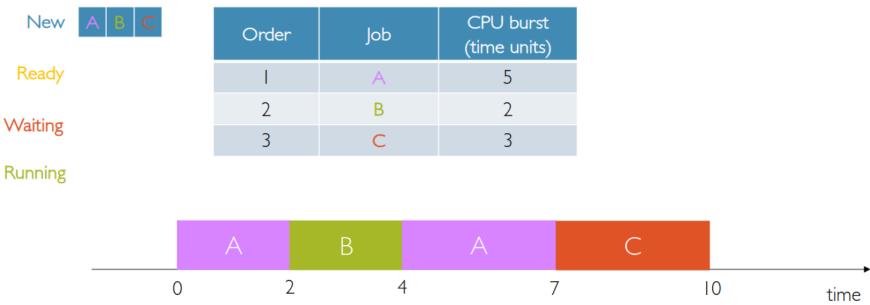


Figura 24

$$\bar{T}_{waiting} = \frac{1}{n} \sum_{i=0}^n T_i^{waiting} = \frac{1+2+7}{3} = \frac{10}{3} \simeq 3.3$$

## Pro e Contro

Uno dei **vantaggi** è la semplicità di implementazione.

Tra gli **svantaggi**:

- Il tempo di attesa (medio) è molto variabile in quanto i lavori di breve durata della CPU possono rimanere indietro quelli molto lunghi.
- effetto convoglio: scarsa sovrapposizione tra CPU e I/O poiché i lavori legati alla CPU costringeranno i lavori legati all'I/O ad attendere.

### 2.12.3 Round - Robin (RR)

Il Round Robin è un algoritmo di scheduling **preemptive** dove viene selezionato il primo processo presente in ready queue dove i burst della CPU sono assegnati con limiti chiamati **time quantum** o **(time slice)**.

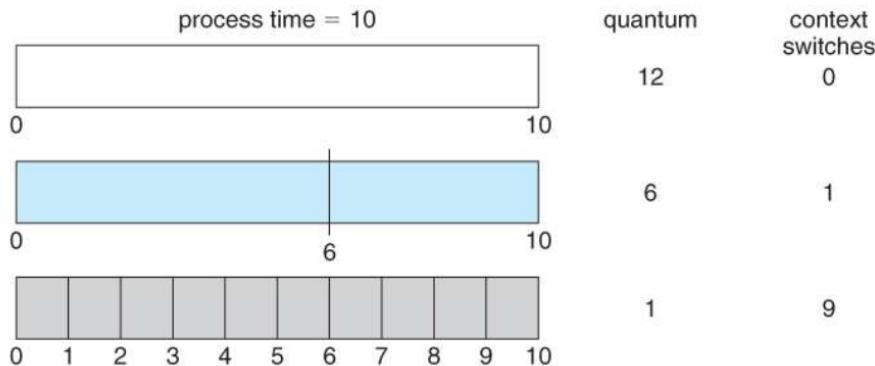


Figura 25

Quando un lavoro viene assegnato alla CPU, viene impostato un timer per un determinato valore, se il lavoro termina prima della scadenza del quanto temporale, viene sostituito da CPU proprio come il normale algoritmo FCFS. Se il timer si spegne per primo, il lavoro viene scambiato fuori dalla CPU e spostato nel back-end della coda dei pronti.

Viene utilizzato in molti sistemi di time-sharing in combinazione con gli interrupt del timer.

La queue pronta viene mantenuta come queue circolare. Quando tutti i lavori hanno avuto un turno, lo scheduler ne assegna un altro al primo lavoro girare e così via. L'algoritmo RR viene considerato equo in quanto condivide la CPU equamente tra tutti i lavori. Il tempo medio di attesa può essere più lungo rispetto ad altri algoritmi di scheduling.

Il limite superiore per lo start time di un processo in una coda di n processi utilizzando uno scheduler RR corrisponde a  $\sup\{T_i^{start}\} = \delta \cdot (i - 1)$ , dove  $i \in [1, n]$  e dove  $\delta$  è il time slice.

### 2.12.4 Esempi

- Consideriamo la seguente coda dei processi utilizzando un algoritmo RR con un time slice di 2 e un context switch trascurabile:

	Order	Job	CPU burst (time units)
New	A	A	5
Ready	B	B	2
Waiting	C	C	3
Running			

Figura 26

- Il primo processo a prendere il controllo della CPU è il processo A, venendo bloccato ed aggiunto alla ready queue dopo 2 unità temporali per via del time slice

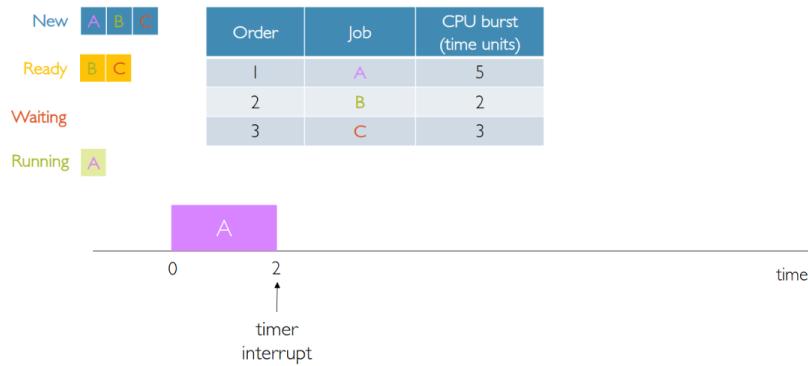


Figura 27

- Successivamente, il processo B prende controllo della CPU, venendo anch'esso bloccato dopo 2 unità temporali. In questo caso, tuttavia, il processo B non verrà aggiunto alla fine della ready queue poiché la sua esecuzione è terminata

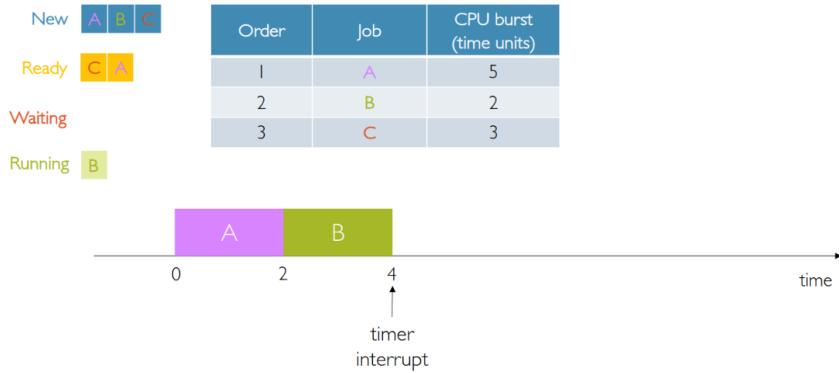


Figura 28

- Di seguito, il processo C prenderà il controllo, venendo bloccato dopo 2 tempi
- Infine, vengono eseguiti i processi A e C finché essi non verranno completati
- Il waiting time medio, quindi, sarà:

$$\bar{T}_{waiting} = \frac{1}{n} \sum_{i=0}^n T_i^{waiting} = \frac{5+2+6}{3} = \frac{13}{3} \simeq 4.3$$

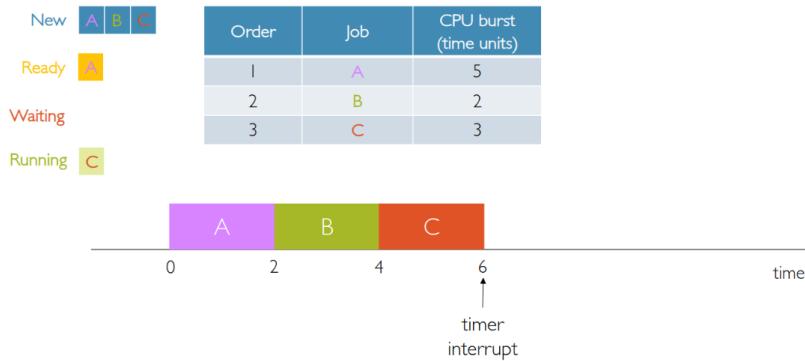


Figura 29

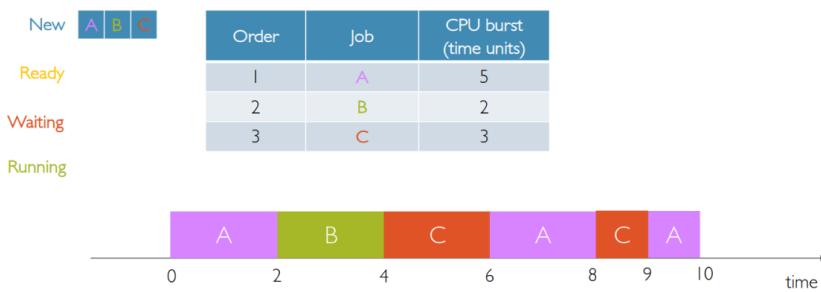


Figura 30

## FCFS vs RR

Confrontando il turnaround time e il waiting time medio tra uno scheduler FCFS e uno scheduler RR, il primo sembra essere a primo occhio più performante del secondo.

Tuttavia, considerando la varianza tra i waiting time di ogni processo, notiamo come il RR risulta essere più equo, fornendo ad ogni processo la stessa quantità di tempo di utilizzo della CPU.

Job	CPU burst	turnaround time		waiting time	
		FCFS	RR	FCFS	RR
A	50	50	150	0	100
B	40	90	140	50	100
C	30	120	120	90	90
D	20	140	90	120	70
E	10	150	50	140	40
Avg.		110	110	80	80

Figura 31

### 2.12.5 Shortest-Job-First (SJF)

Il Shortest-Job-First è un altro algoritmo di scheduling di tipo **non preemptive** che parte dal presupposto che i tempi di esecuzione siano conosciuti in anticipo. Lo scheduler preleva per primo il lavoro più breve (da qui il nome **shortest job first**).

Una versione **preemptive** dell'algoritmo shortest job first è l'algoritmo shortest remaining time first dove ogni volta che un nuovo job arriva nella ready queue e il suo CPU burst stimato è minore di quello rimanente del job attualmente in esecuzione, tale job prende controllo della CPU.

Lo SJF presenta dei vantaggi infatti risulta ottimale quando l'obiettivo è minimizzare il tempo medio di attesa. Funziona sia con schedulatori preventivi che non preventivi.

Tra gli svantaggi possiamo affermare che risulta quasi impossibile prevedere la quantità di tempo CPU di un lavoro. Anche i lavori legati alla CPU a esecuzione prolungata possono morire di *starve* (come hanno fatto implicitamente quelli legati all'I/O priorità maggiore rispetto a loro).

Per stimare la lunghezza del prossimo CPU burst, viene utilizzato l'exponential smoothing:

- Sia  $x_t$  la lunghezza effettiva del  $t$ -esimo CPU burst
- Sia  $S_{t+1}$  la lunghezza stimata del  $(t + 1)$ -esimo CPU burst
- Sia  $\alpha \in R$  dove  $0 \leq \alpha \leq 1$
- Si ha che:

$$\begin{aligned} s_1 &= x_0 \\ s_{t+1} &= \alpha x_t + (1 - \alpha)s_t \end{aligned}$$

### 2.12.6 Esempi

1. Consideriamo la seguente coda di processi gestita da uno scheduler SJF non preemptive. In tal caso, si ha che:

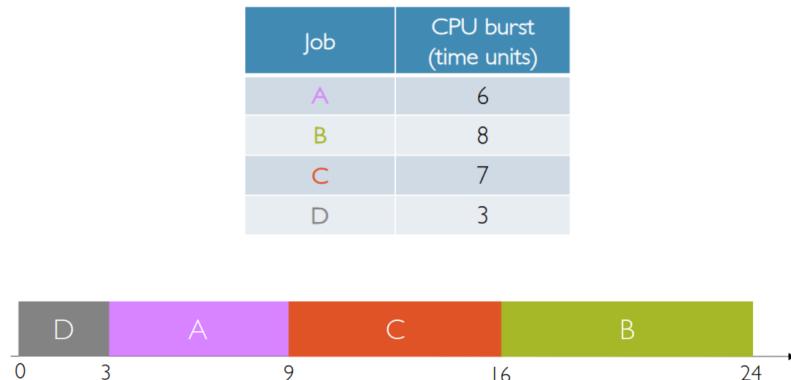


Figura 32

$$\bar{T}_{ave.waiting} = \frac{3+16+9+0}{4} = \frac{28}{4} = 7$$

2. Consideriamo la seguente coda di processi gestita da uno scheduler SJF preemptive, ossia uno scheduler SRTF.

Job	Arrival time	CPU burst (time units)
A	0	8
B	1	4
C	2	9
D	3	5



Figura 33

- Il primo job ad essere eseguito è il processo A, poiché il primo ad essere creato ed inserito nella ready queue.

Job	Arrival time	CPU burst (time units)
A	0	8
B	1	4
C	2	9
D	3	5



Figura 34

- Una volta che il processo B viene creato, il suo CPU burst stimato, ossia 4, viene comparato con il CPU burst rimanente del processo A, ossia 7. Poiché  $4 < 7$ , allora il processo B prende controllo della CPU. Analogamente, nell'istante in cui il processo C viene creato, il suo CPU burst stimato, ossia 9, viene comparato con quello rimanente del processo A, ossia 6, e quello del processo B, ossia 3. Siccome  $3 < 7 < 9$ , allora il processo B mantiene il controllo della CPU. Lo stesso ragionamento viene effettuato anche dopo la creazione del processo D, dove il processo B mantiene il controllo ( $2 < 7 < 5 < 9$ ).

Di conseguenza, il processo B verrà eseguito fino al suo completamento.

Job	Arrival time	CPU burst (time units)
A	0	8
B	1	4
C	2	9
D	3	5



Figura 35

- Una volta completato il processo B, verrà eseguito il processo avente il CPU burst stimato minore:
  - Il processo A ha un CPU burst rimanente pari a 7
  - Il processo C ha un CPU burst rimanente pari a 9
  - Il processo D ha un CPU burst rimanente pari a 5 Di conseguenza, verrà eseguito il processo D fino al suo completamento

Job	Arrival time	CPU burst (time units)
A	0	8
B	1	4
C	2	9
D	3	5



Figura 36

- Analogamente al processo D, verranno eseguiti il processo A e il processo C, ognuno di essi fino al loro completamento (poiché, nel frattempo, nessun altro processo viene inserito nella ready queue, dunque non viene mai attivato l'algoritmo di scheduling).

Job	Arrival time	CPU burst (time units)
A	0	8
B	1	4
C	2	9
D	3	5



Figura 37

$$\bar{T}_{ave.waiting} = \frac{(17-0-8)+(5-1-4)+(26-2-9)+(10-3-5)}{4} = 6.5$$

### Confronto tra FCFS, RR e SJF

Job	CPU burst	turnaround time			waiting time		
		FCFS	RR	SJF	FCFS	RR	SJF
A	50	50	150	150	0	100	100
B	40	90	140	100	50	100	60
C	30	120	120	60	90	90	30
D	20	140	90	30	120	70	10
E	10	150	50	10	140	40	0
Avg.		110	110	70	80	80	40

Figura 38

### 2.12.7 Priority Scheduling

Nel **Priority Scheduling** a ciascun processo è assegnata una priorità e il processo eseguibile con la priorità più alta è quello cui è consentita l'esecuzione. Per impedire che i processi ad alta priorità siano eseguiti **indefinitamente**, lo scheduler può abbassare la priorità del processo attualmente in esecuzione a ogni scatto del clock (cioè a ogni interrupt). Se questa azione fa sì che la sua **priorità** vada al di sotto di quella del processo successivo, avviene uno scambio di processo. In alternativa a ciascun processo può essere assegnato uno slice massimo in cui può essere eseguito.

Le priorità possono essere assegnate in due modi:

- Internamente, ossia assegnate dall'OS in base a determinati criteri (ad esempio il CPU burst medio, il rateo di attività tra CPU e I/O, risorse utilizzate, ...)
- Esternamente, ossia assegnate dall'utente in base all'importanza del job o di altri criteri.

Un semplice algoritmo che offre un buon servizio ai processi I/O bound è quello di impostare la priorità a  $1/f$ , dove  $f$  è la frazione dell'ultimo quanto usato dal processo. Un processo che ha usato uno solo dei suoi 50 ms di slice avrà una priorità 50, mentre un processo che sia stato eseguito per 25 ms prima di bloccarsi avrà priorità 2; un processo che abbia usato tutto lo slice avrà priorità 1.

Il priority scheduling può essere sia **non-preemptive** sia **preemptive**.

### Problemi

Il Priority Scheduling può soffrire di *starvation*, dove un job di bassa priorità rimane in attesa per un tempo indefinito, poiché altri job hanno sempre una priorità maggiore. Tali job potrebbero essere eseguiti eventualmente quando il carico del sistema è minore o dopo che il sistema stesso vada in crash, venga spento o venga riavviato. Come contromisura alla *starvation* viene utilizzato l'*aging*, ossia l'incremento della priorità di un job in base al suo tempo in attesa, finché essi non verranno eventualmente schedulati.

### 2.12.8 Multilevel Queue (MLQ e MLFQ)

L'algoritmo **Multilevel Queue (MLQ)** è un algoritmo di scheduling basato sull'utilizzo di queue multiple separate tra loro, ognuna per ogni categoria di job, dove ogni queue utilizza l'algoritmo di scheduling più appropriato per una determinata categoria di job. Una volta inserito all'interno di una queue, nessun job può essere spostato in un'altra queue.

Le due opzioni più comuni per l'implementazione di tale algoritmo sono:

- **String priority**, dove nessun job all'interno di una queue di priorità più bassa viene eseguito finché esiste almeno un job delle queue di priorità più alta
- **Round robin**, dove ogni queue utilizza un proprio time slice, il quale aumenta esponenzialmente al diminuire della priorità della coda

L'algoritmo **Multilevel Feedback Queue (MLFQ)** segue la stessa idea dell'algoritmo MLQ, con l'aggiunta della possibilità per ogni job di essere spostato da una queue all'altra.

Lo spostamento di un job può rivelarsi necessario quando:

- Un job passa dall'utilizzare molto la CPU all'utilizzare molto l'I/O e viceversa
- Un job è in stato di starving, venendo spostato per breve tempo in una queue a priorità più alta

Per gestire gli spostamenti, l'algoritmo utilizza le seguenti regole:

- Inizialmente ad ogni job viene assegnata la priorità più alta
- Se il time slice di un job scade, allora quest'ultimo viene spostato nella queue con un livello di priorità inferiore rispetto a quella precedente
- Se il time slice di un job non scade, allora quest'ultimo viene spostato nella queue con un livello di priorità superiore rispetto a quella precedente
- La priorità di processi strettamente legati alla CPU viene diminuita rapidamente
- La priorità di processi strettamente legati all'I/O rimarrà alta

### 2.12.9 Esempio di suddivisione tramite MLQ e MLFQ:

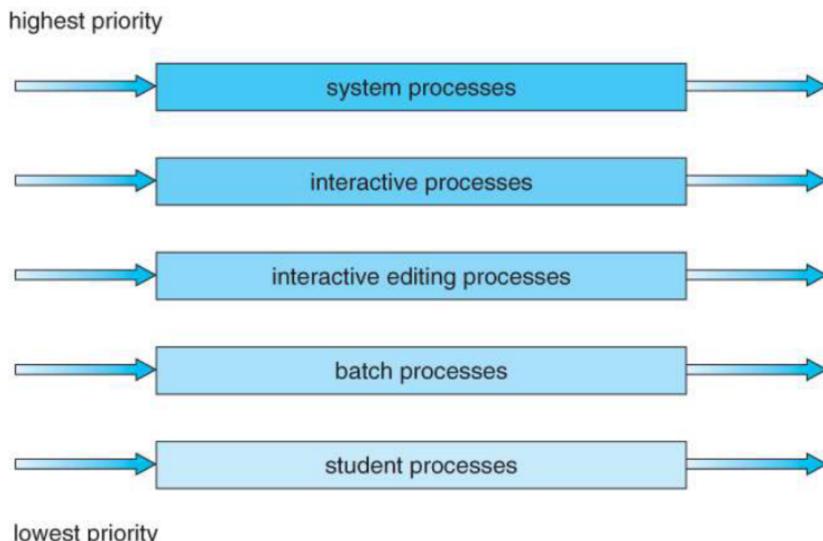


Figura 39

### 2.12.10 Esempio

- Consideriamo la seguente coda di processi gestita da un algoritmo MLFQ con 3 queue e strict priority.

- Supponiamo:

New [A|B|C]

Order	Job	CPU burst (time units)
1	A	30
2	B	20
3	C	10

Queue	Time Slice (time units)	Jobs
1	1	
2	2	
3	4	

Figura 40

- Per indicare l'avanzamento di ogni processo all'avanzare del tempo, utilizziamo la notazione:

$$JOB_{total\_elapsed\_time}^{execution\_time}$$

New [A|B|C]

Order	Job	CPU burst (time units)
1	A	30
2	B	20
3	C	10

Queue	Time Slice (time units)	Jobs
1	1	A <sub>1</sub> , B <sub>1</sub> , C <sub>1</sub>
2	2	
3	4	

Figura 41

- Siccome il time slice per tutti e tre i processi è scaduto, ognuno di essi viene spostato nella queue a priorità inferiore

New [A|B|C]

Order	Job	CPU burst (time units)
1	A	30
2	B	20
3	C	10

Queue	Time Slice (time units)	Jobs
1	1	A <sub>1</sub> , B <sub>1</sub> , C <sub>1</sub>
2	2	A <sub>2</sub> , B <sub>2</sub> , C <sub>2</sub>
3	4	

Figura 42

- Analogamente, a prima, anche in questo caso il time slice per tutti e tre scade, dunque vengono spostati alla queue inferiore

New	A	B	C	Order	Job	CPU burst (time units)
				1	A	30
				2	B	20
				3	C	10
Queue				Time Slice (time units)	Jobs	
1				1	$A^1_1, B^1_2, C^1_3$	
2				2	$A^3_5, B^3_7, C^3_9$	
3				4	$A^7_{13}, B^7_{17}, C^7_{21}$	

Figura 43

- Una volta raggiunta la queue a priorità più bassa, il time slice di tutti e tre i processi scadrà sempre fino al loro completamento

New	A	B	C	Order	Job	CPU burst (time units)
				1	A	30
				2	B	20
				3	C	10
Queue				Time Slice (time units)	Jobs	
1				1	$A^1_1, B^1_2, C^1_3$	
2				2	$A^3_5, B^3_7, C^3_9$	
3				4	$A^7_{13}, B^7_{17}, C^7_{21}$ $A^{11}_{25}, B^{11}_{29}, C^{10}_{32}$	

Figura 44

### 2.12.11 Extra: Lottery scheduling

L'algoritmo **lottery scheduling** è un algoritmo di scheduling basato sulla casualità:

- Ad ogni job vengono assegnati un determinato numero di biglietti. I job attivi da poco riceveranno più biglietti, mentre quelli attivi da molto ne riceveranno di meno. Per evitare la starvation, ad ogni job viene assegnato almeno un biglietto
- Ogni volta che un time slice scade viene determinato un biglietto vincitore. Il processo che detiene tale biglietto prenderà il controllo della CPU.
- Ai processi più importanti vengono assegnati biglietti extra
- Successivamente, il procedimento viene ripetuto

Per via della legge dei grandi numeri, all'aumentare del numero di estrazioni effettuate il tempo di utilizzo della CPU di ogni processo tenderà a raggiungere la media.

## 2.13 Thread e Multi-threading

Un **thread** è un'unità base dell'utilizzo della CPU e consiste in un contatore di programma, uno stack, e un insieme di registri oltre che un proprio ID. Ogni processo definisce le risorse "globali" (ossia lo spazio d'indirizzamento, le istruzioni da eseguire, i dati, le risorse, ...) mentre ogni suo thread definisce un singolo stream di esecuzione all'interno del processo stesso. Poiché lo spazio d'indirizzamento del processo è condiviso tra tutti i suoi thread, nessuna syscall è richiesta per far cooperare i thread tra di loro, risultando in una comunicazione più semplice rispetto allo scambio di messaggi e alla memoria condivisa.

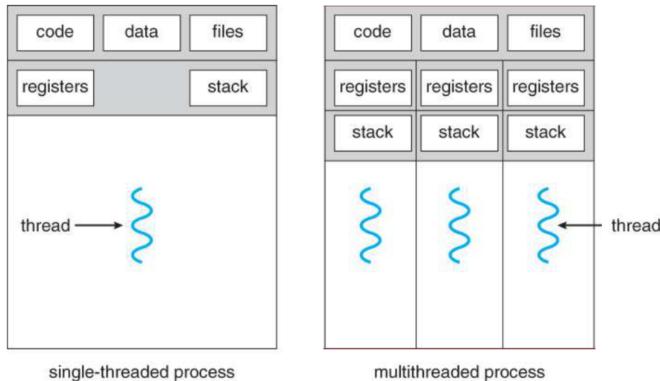


Figura 45

Possiamo notare infatti che:

- Ciascun thread possiede un insieme di registrasti indipendenti e un proprio stato
- Tutti i thread di processi condividono lo stesso codice e le risorse globali
- Da quando tutti i thread sono attivi nello stesso spazio di indirizzamento la comunicazione tra loro risulta più semplice della comunicazione tra i processi

I thread sono molto utili nella programmazione moderna ogni volta che un processo ha più attività per eseguire indipendentemente dagli altri. Ciò è particolarmente vero quando una delle attività può bloccare, ed è desiderato consentire alle altre attività di procedere senza bloccare (E.g: elaboratore di testi). Un thread può verificare l'ortografia e la grammatica mentre un altro thread gestisce l'input dell'utente (tasti) e un terzo fa backup periodici del file che viene modificato.

Teoricamente, ogni attività secondaria di un'applicazione potrebbe essere implementata come un nuovo processo a thread singolo piuttosto che come un processo a più thread.

Ci sono almeno 2 motivi per cui questa non è la scelta migliore:

- La **comunicazione** tra thread è significativamente più veloce di quella tra processi
- Il **context switch** tra thread è molto più veloce che tra processi

### 2.13.1 Vantaggi

Possiamo scrivere di quattro importanti vantaggi:

- **Reattività**, poiché un processo potrebbe fornire una risposta in modo rapido mentre tutti gli altri thread sono bloccati o rallentati a causa dell'intenso uso della CPU
- **Condivisione** delle risorse "globali" come codice, dati e spazio di indirizzamento
- **Economicità**, poiché creare e gestire thread è più veloce rispetto alla creazione e gestione dei processi
- **Scalabilità**, poiché un processo single-thread può essere eseguito su un singolo core, mentre ogni thread di un processo multithread può essere suddiviso tra tutti i core disponibili della CPU (solo per architetture multi-core)

### 2.13.2 Programmazione Multi-core

Una tendenza recente nell'architettura dei computer è quella di produrre chip con più core, o CPU su un singolo chip. Un'applicazione multi-thread in esecuzione su un tradizionale chip single-core dovrebbe intercalare i thread. Su un chip multi-core, tuttavia, i thread potrebbero essere distribuiti tra i core disponibili, consentendo una vera **elaborazione parallela**.<sup>14</sup>

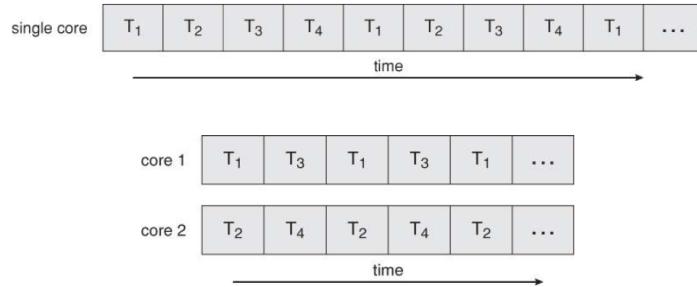


Figura 46: I chip multi-core richiedono nuovi algoritmi di pianificazione del sistema operativo per utilizzare al meglio i più core disponibili

### 2.13.3 Tipi di parallelismo

In teoria, ci sono due modi per parallelizzare il carico di lavoro:

- Parallelismo dei **dati**: suddivide i dati tra più core (thread) e esegue la stessa attività su ciascun blocco di dati
- Parallelismo dei **compiti**: suddivide i diversi compiti da eseguire tra i diversi core e li esegue contemporaneamente
- In pratica, nessun programma è mai suddiviso unicamente dall'uno o dall'altro di questi, ma piuttosto da una sorta di combinazione **ibrida**

## Classificazione degli OSs

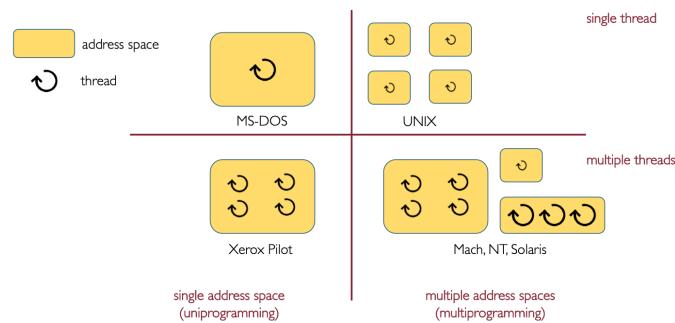


Figura 47

### 2.13.4 Kernel threads vs User threads

Il supporto per (più) thread può essere fornito in due modi ossia a livello di **kernel** (thread del kernel) e a livello **utente** (thread utente).

I thread del kernel viene gestito direttamente dal kernel del sistema operativo stesso, mentre i thread utente vengono gestiti in spazio utente da una libreria di thread a livello utente, senza intervento del sistema operativo.

<sup>14</sup>Le CPU sono state sviluppate per supportare più thread simultanei per core nell'hardware (ad esempio, l'hyper-threading di Intel). Ogni core fisico appare come due processori al sistema operativo, consentendo la pianificazione simultanea di due processi per core.

I **thread del kernel** hanno la più piccola unità di esecuzione che può essere pianificata dal sistema operativo. Il sistema operativo è responsabile del supporto e della gestione di tutti i thread. Possiedono un Process Control Block (**PCB**) per ogni processo, un Thread Control Block (**TCB**) per ogni thread. Il sistema operativo di solito fornisce chiamate di sistema per creare e gestire i thread dallo spazio utente.

Tra i **vantaggi** del Kernel Thread:

- Il kernel ha piena conoscenza di tutti i thread
- Lo Scheduler può decidere di concedere più tempo di CPU a un processo con un gran numero di thread
- Buono per le applicazioni che si bloccano frequentemente
- Il passaggio da un thread all'altro è più rapido del passaggio da un processo all'altro

Tra gli **svantaggi**:

- Overhead significativo e aumento del kernel complessità
- Lento e inefficiente (necessita di invocazioni del kernel)
- Il Context switching, sebbene più leggero, è gestito dal kernel

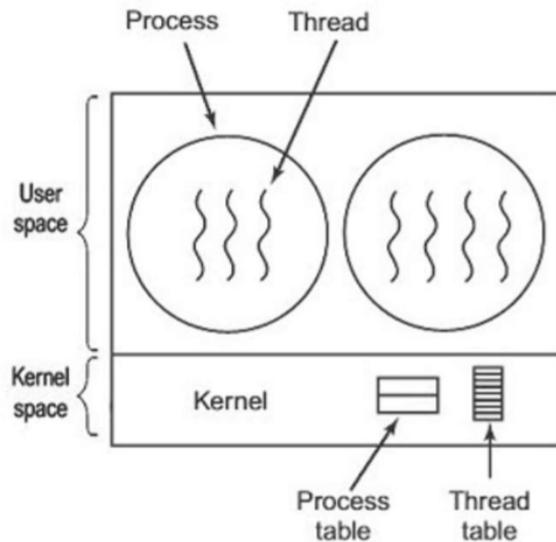


Figura 48

Gli **User Thread** invece vengono gestiti interamente dal sistema run-time (libreria thread a livello utente). Il kernel del sistema operativo non sa nulla dei thread a livello utente e gestisce i thread a livello utente come se fossero processi a thread singolo. Idealmente, le operazioni sui thread dovrebbero essere veloci quanto una chiamata di funzione.

Tra i **vantaggi**:

- Veramente veloce e leggero
- Le politiche di programmazione sono più flessibili
- Può essere implementato in sistemi operativi che non supportano il threading
- Nessuna chiamata di sistema coinvolta, solo chiamate di funzione in spazio utente
- Nessun Context Switch effettivo

Tra gli svantaggi:

- Nessuna vera concorrenza di multi-thread processi
- Decisioni di pianificazione sbagliate
- Mancanza di coordinamento tra kernel e thread
- Un processo con 100 thread compete per un intervallo di tempo con un processo con solo 1 thread
- Richiede chiamate di sistema non bloccanti, altrimenti tutti i thread all'interno di un processo devono attendere

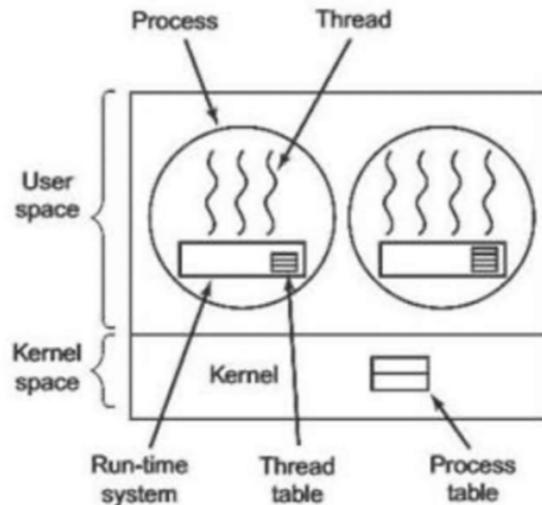


Figura 49

Un **lightweight process (LWP)** è un processore virtuale attivo nella user space contenente un singolo kernel thread, mentre multipli user thread gestiti tramite una libreria per i thread vengono posti su uno o più LWP, i quali assumono un ruolo di "tramite" tra le due tipologie di thread.

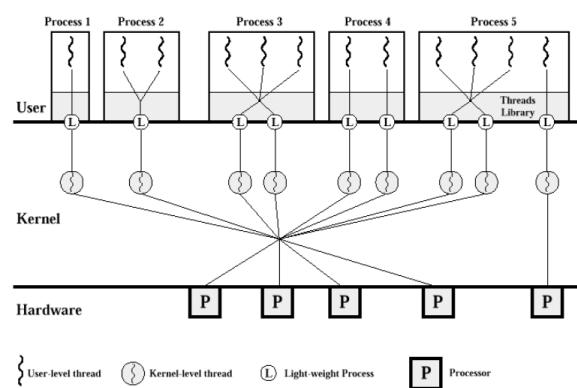


Figura 50: Hybrid Management: Lightweight Processes

### 2.13.5 Modelli di Multi-threading

In un'implementazione specifica, i thread utente devono essere mappati ai thread del kernel e ne esistono di quattro tipi:<sup>15</sup>

- Molti a uno (puro livello utente)
  - Molti thread utente sono tutti mappati su un singolo thread del kernel
  - Il processo può eseguire un solo thread utente alla volta perché c'è solo un thread del kernel ad esso associato
  - Poiché un singolo thread del kernel può operare su una singola CPU, i processi multi-user-thread non possono essere suddivisi su più CPU
  - Se viene effettuata una chiamata di sistema bloccante, l'intero processo si blocca, anche se altri thread utente sarebbero in grado di continuare
- Uno a uno (puro livello kernel)
  - Un thread del kernel separato per gestire ciascun thread utente
  - Supera i limiti del blocco delle chiamate di sistema e suddivisione dei processi su più CPU
  - Il sovraccarico della gestione del modello uno a uno è più significativo e può rallentare il sistema
  - La maggior parte delle implementazioni di questo modello pone un limite al numero di thread che possono essere creati
- Molti a molti
  - Esegue il multiplexing di un numero qualsiasi di thread utente su un numero uguale o inferiore di thread del kernel
  - Gli utenti non hanno restrizioni sul numero di thread creati
  - I processi possono essere suddivisi su più processori
  - Il blocco delle chiamate di sistema del kernel non blocca l'intero processo
- Due livelli
  - Una variante del modello molti-a-molti
  - Combina molti a molti con uno a uno
  - Aumenta la flessibilità delle politiche di programmazione

---

<sup>15</sup>Un thread del kernel è l'unità di esecuzione pianificata dal sistema operativo per l'esecuzione sulla CPU (simile al processo a thread singolo)

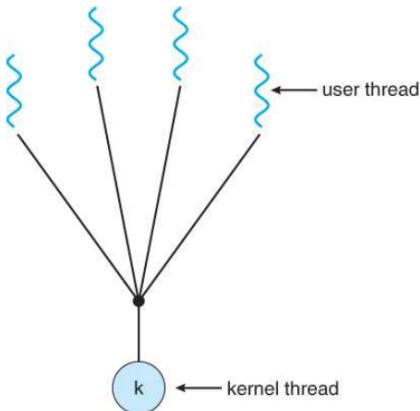


Figura 51: Molti a uno

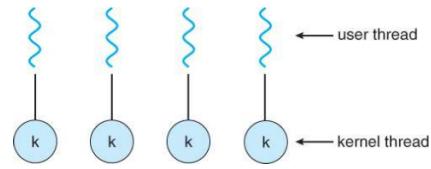


Figura 52: Uno a uno

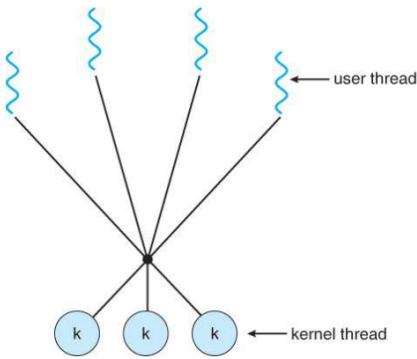


Figura 53: Molti a molti

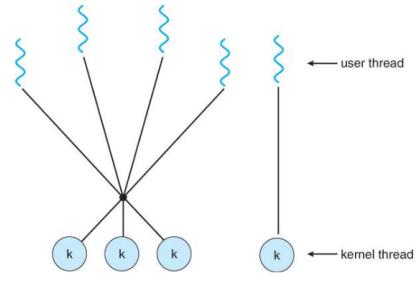


Figura 54: Due livelli

Quando un thread effettua una syscall `fork()` o `exec()` in un processo, si presenta la questione di come gestire la creazione di un nuovo processo. In particolare, ci sono due opzioni:

1. Copiare l'intero processo, comprensivo di tutti i thread presenti, al fine di generare un nuovo processo con una copia esatta dell'originale;
2. Generare un nuovo processo single-thread contenente una copia del singolo thread che ha effettuato la syscall.

La scelta tra queste due opzioni dipende dalla progettazione dell'OS, che deve garantire il corretto funzionamento del sistema. Tuttavia, in molti casi, è possibile adottare una **soluzione ibrida**: se il nuovo processo viene eseguito subito dopo la syscall, allora non c'è necessità di copiare anche gli altri thread del processo originale. Invece, se il nuovo processo viene eseguito in un momento successivo, l'intero processo dovrebbe essere copiato.

Nel caso in cui un processo multi-thread riceva un segnale, la gestione di quale thread è il destinatario del segnale è un'importante problematica da affrontare. Esistono diversi approcci per gestire questa problematica, tra cui:

1. Invio del segnale al thread specifico che ne ha bisogno.
2. Invio del segnale a tutti i thread del processo.
3. Invio del segnale solo ad alcuni thread specifici del processo.
4. Scelta di un thread specifico che gestisca tutti i segnali ricevuti.

Perché un thread possa essere schedulato ed eseguito dalla CPU, deve competere con gli altri thread che cercano di accedere alla stessa risorsa. La gestione dei thread contendenti avviene tramite il contention scope, che può essere di due tipologie: **Process Contention Scope (PCS)** e **System Contention Scope (SCS)**.

Nel **PCS**, la competizione avviene tra i thread appartenenti allo stesso processo. Per implementare questa tipologia di contention scope, il sistema deve utilizzare un modello Molti a Molti o Molti a Uno.

Nel primo caso, i thread sono mappati su un insieme di kernel threads e il sistema operativo si occupa di distribuire i kernel threads sui processori disponibili. Nel secondo caso, i thread sono mappati su un singolo kernel thread e la distribuzione sui processori è gestita dal sistema operativo.

Nello **SCS**, la competizione avviene tra tutti i thread di ogni processo e viene implementato in sistemi basati sul modello One-to-One, dove ogni thread è mappato su un kernel thread dedicato.

Per comunicare con le librerie per i thread nel momento in cui un evento si verifica, il kernel utilizza un lightweight process (**LWP**). Le up-call, ovvero le chiamate dal kernel alla libreria, vengono gestite da un upcall handler presente nella libreria stessa. Ogni upcall fornisce un nuovo LWP tramite cui l'upcall handler verrà eseguito, garantendo una gestione efficiente e ottimizzata dei thread.

### 3 Sincronizzazione tra Processi/Thread

Abbiamo già accennato al fatto che i processi/thread possono cooperare tra loro per raggiungere un compito comune. Tuttavia, la cooperazione può richiedere la **sincronizzazione**<sup>16</sup> tra i thread a causa della presenza delle cosiddette sezioni critiche (o regioni critiche). Le **primitive** di sincronizzazione sono necessarie per garantire che solo un thread alla volta esegua una sezione critica.

Le **race condition** sono situazioni dove due o più processi stanno scrivendo o leggendo i medesimi dati condivisi e il risultato finale dipende da chi viene prescelto per l'esecuzione.<sup>17</sup>

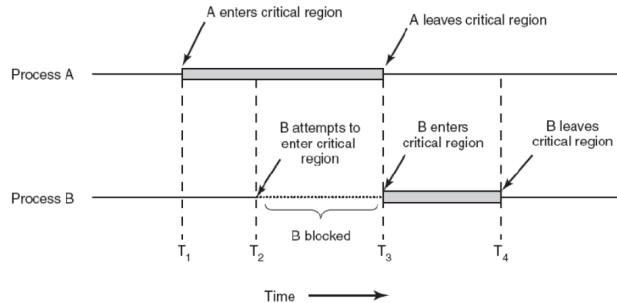


Figura 55: The Critical Section Problem

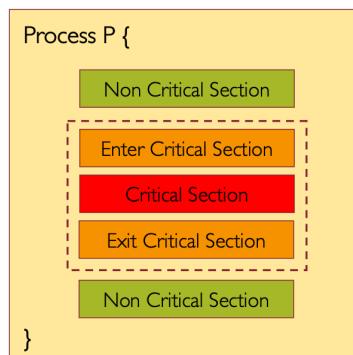


Figura 56: The Critical Section Problem

Qualsiasi soluzione di sincronizzazione al problema della sezione critica deve soddisfare tre proprietà:

- **Mutua esclusione**, ossia solo un processo/thread alla volta può trovarsi nella sua sezione critica
- **Liveness**, ossia se nessun processo è nella sua sezione critica, e uno o più vogliono eseguirlo, allora ognuno di questi deve essere in grado di entrare nella sua sezione critica
- **Bounded Waiting**, ossia un processo che richiede l'ingresso nella sua sezione critica avrà un turno alla fine, e c'è un limite su quanti altri possono iniziare per primi.

Nell'esempio del latte:

- Garantire l'esclusione reciproca significa non più latte di quanto sarà necessario comprare (cioè, solo uno tra *Bob* e *Carla* comprerà il latte se necessario)
- Garantire la liveness significa che qualcuno dovrebbe comprare del latte (cioè, l'opzione dove sia *Bob* che *Carla* non fanno nulla è sicuramente sicuro ma indesiderabile)
- Garantire l'attesa del limite significa che alla fine *Bob* e *Carla* entreranno nella loro sezione critica

<sup>16</sup>La sincronizzazione come soluzione al problema della sezione critica

<sup>17</sup>La **mutua esclusione** si occupa di bloccare un processo che deve leggere dei dati che vengono già letti da un altro processo

Time	Bob	Carlo
5:00pm	Arrive home	
5:05pm	Look in the fridge → No milk!	
5:10pm	Leave home for the grocery	
5:20pm		Arrive home
5:25pm	Arrive at the grocery	Look in the fridge → No milk!
5:30pm	Buy milk	Leave home for the grocery
5:45pm	Arrive home, put the milk in the fridge	Arrive at the grocery
5:50pm		Buy milk
6:05pm		Arrive home, put the milk in the fridge
6:05pm	Oh f%#k!	Oh f%#k!

Figura 57: The Critical Section Problem

Per poter sincronizzare i processi/thread tra di loro, i linguaggi di programmazione forniscono delle primitive atomiche basate su una delle seguenti tre soluzioni:

- La sincronizzazione avviene tramite un **lock**, dove prima di accedere ad un settore critico un processo acquisisce tale lock, per poi rilasciarlo una volta uscito dal settore critico
- La sincronizzazione avviene tramite un **semaforo**, una generalizzazione del lock
- La sincronizzazione avviene tramite un **monitor**, il quale connette dei dati condivisi alle primitive di sincronizzazione

## 3.1 I Lock

Fornire l'esclusione reciproca ai dati condivisi utilizzando due primitive atomiche:

- `Lock.acquire()` à attendere che il blocco sia libero, quindi acquisirlo
- `Lock.release()` per sbloccare e riattivare qualsiasi thread di attesa in `acquire()`

Esistono delle regole per l'utilizzo di un lock:

- **Acquisire** sempre il blocco prima di accedere ai dati condivisi
- **Rilasciare** sempre il blocco dopo aver terminato con i dati condivisi
- Il blocco deve essere inizialmente **libero**
- Solo un processo/thread può acquisire il lock, gli altri **aspetteranno**

Poiché lo scheduler della CPU prende il controllo solo a seguito di eventi interni, ossia quando il thread in esecuzione lascia il controllo della CPU, oppure a seguito di eventi esterni, per evitare problemi all'interno dei settori critici è necessario impedire lo scheduling durante il rilascio o l'acquisizione di un lock. Per implementare le **primitive** ad alto livello relative alla sincronizzazione, quindi, è necessario del supporto hardware di basso livello che possa eseguire tali **operazioni atomiche**<sup>18</sup>, in particolare tramite la **disabilitazione degli interrupt**<sup>19</sup> o l'uso di istruzioni atomiche.

Sui sistemi a CPU singola, possiamo impedire che lo scheduler possa prendere il sopravvento:

- eventi **interni** imponendo ai thread di non richiedere alcuna operazione di I/O durante un sezione critica
- evento **esterno** per disabilitare gli interrupt (ovvero, dire all'HW di ritardare la gestione di qualsiasi evento esterno fino a quando il thread corrente non è terminato con la sezione critica)

Copriamo tutti i possibili casi in cui il thread corrente potrebbe perdere il controllo della CPU, volontariamente (a causa di eventi interni) o involontariamente (a causa di eventi esterni).

In sintesi supponiamo di avere una sola variabile condivisa (lock), inizialmente a 0. Quando un processo vuole entrare nella sua regione critica, prima controlla il lock. Se è 0, il processo lo imposta a 1 ed entra nella regione critica. Se è già a 1, il processo aspetta finché non vale 0. In questo modo 0 significa che nessun processo è nella sua regione critica, 1 che un qualche processo è nella sua regione critica.

<sup>18</sup>Un'operazione viene detta atomica se essa non può essere interrotta in alcun modo, impedendo quindi che possa avvenire un context switch durante la sua esecuzione.

<sup>19</sup>La prima soluzione a supporto hardware per poter implementare la sincronizzazione, corrisponde alla disabilitazione degli interrupt mentre un thread acquisisce o rilascia un lock

```

Class Lock {
    public void acquire(Thread t);
    public void release();
    private int value; // 0=FREE, 1=BUSY
    private Queue<Thread> q;

    Lock() {
        // lock is initially FREE
        this.value = 0;
        this.q = null;
    }
}

public void acquire(Thread t) {
    disable_interrupts();
    if(this.value) { // lock is held by someone
        q.push(t); // add t to waiting queue
        t.sleep(); // put t to sleep
    }
    else {
        this.value = 1;
    }
    enable_interrupts();
}

public void release() {
    disable_interrupts();
    if(!q.isEmpty()) {
        t = q.pop(); // extract a waiting thread from q
        push_onto_ready_queue(t); // put t on ready queue
    }
    else {
        this.value = 0;
    }
    enable_interrupts();
}

```

Figura 58: The Critical Section Problem

Sfortunatamente questa idea presenta esattamente lo stesso **difetto** constatato nella directory di pool di stampa. Supponiamo che un processo legga il lock e veda che è 0. Prima che riesca a metterlo a 1, viene eseguito un altro processo schedulato e imposta il lock a 1.

Quando riparte ancora il primo processo, anch'esso lo imposterà a uno e i due processi sarebbero nella loro regione critica nello stesso momento.

## Istruzione atomica

Un'istruzione atomica di lettura-modifica-scrittura legge un valore dalla memoria in un registro e scrive un nuovo valore in un colpo solo. Su un uniprocessore sarà semplice implementare l'aggiunta di una nuova istruzione mentre su un multiprocessore, anche il processore che impartisce l'istruzione deve essere in grado di farlo invalidare qualsiasi copia del valore che altri processi potrebbero avere nella loro cache.

Prendiamo in esempio il **test&set** (la maggior parte delle architetture) legge un valore, riscrive in memoria oppure uno **scambio** (x86) scambia valori tra registro e memoria.

```

Class Lock {
    public void acquire();
    public void release();
    private int value;

    Lock() {
        // lock is initially free
        this.value = 0;
    }
}

public void acquire() {
    while(test&set(this.value) == 1) {
        // while busy do nothing
    }
}

public void release() {
    this.value = 0;
}

```

Figura 59: The Critical Section Problem

L'implementazione del lock tramite **disabilitazione degli interrupt** presenta alcune **problematiche**, infatti ha la necessità dell'invocazione del kernel che rende il tutto più complesso. Inoltre risulta inutilizzabile in sistemi multi-core.

Esistono due **problemi** principali con le istruzioni atomiche: il **busy waiting**<sup>20</sup> (che andrebbe generalmente evitato dato che consuma tempo di CPU)<sup>21</sup>.

Infine abbiamo la **unfairness** in quanto non esiste una coda in cui i thread attendono il rilascio del blocco.

<sup>20</sup>L'espressione busy waiting o busy wait (letteralmente "attesa impegnata", più spesso tradotto come "attesa attiva") indica una tecnica di sincronizzazione per cui un processo o un thread che debba attendere il verificarsi di una certa condizione (per esempio la disponibilità di input dalla tastiera o di un messaggio proveniente da un altro processo) lo faccia verificando ripetutamente (ciclicamente) tale condizione. Questo approccio è alternativo all'uso di una sospensione del processo e del suo successivo risveglio tramite un segnale specifico (per esempio un interrupt nel caso dell'input da tastiera). Fonte: Wikipedia

<sup>21</sup>un lock che utilizza il busy waiting si chiama **spin lock**

Nonostante il tempo in attesa della CPU sia un problema irrisolvibile nel caso delle istruzioni atomiche, esso può essere minimizzato. Inoltre, l'aggiunta di una queue permette di gestire in modo deterministico quale thread vada ad ottenere il lock a seguito del suo rilascio:

```

Class Lock {
    public void acquire(Thread t);
    public void release();
    private int value;
    private int guard;
    private Queue q;

    Lock() {
        // lock is initially free
        this.value = 0;
    }
}

public void acquire(Thread t) {
    while(test&set(this.guard) == 1) {
        // while busy do nothing
    }
    if(this.value) {
        q.push(t);
        t.sleep_and_reset_guard_to_0();
    }
    else {
        this.value = 1;
        this.guard = 0;
    }
}

public void release() {
    while(test&set(this.guard) == 1) {
        // while busy do nothing
    }
    if(!q.is_empty()) {
        t = q.pop();
        push_onto_ready_queue(t);
    }
    else {
        this.value = 0;
    }
    this.guard = 0;
}

```

Figura 60: The Critical Section Problem

Esiste un miglioramento di **test&set** per ridurre l'attesa occupata rendendola indipendente da quanto tempo è delimitata la sezione critica da acquisizione e rilascio:

```

Class Lock {
    public void acquire(Thread t);
    public void release();
    private int value;
    private int guard;
    private Queue q;

    Lock() {
        // lock is initially free
        this.value = 0;
    }
}

public void acquire(Thread t) {
    while(test&set(this.guard) == 1) {
        // while busy do nothing
    }
    if(this.value) {
        q.push(t);
        t.sleep_and_reset_guard_to_0();
    }
    else {
        this.value = 1;
        this.guard = 0;
    }
}

public void release() {
    while(test&set(this.guard) == 1) {
        // while busy do nothing
    }
    if(!q.is_empty()) {
        t = q.pop();
        push_onto_ready_queue(t);
    }
    else {
        this.value = 0;
    }
    this.guard = 0;
}

```

Figura 61: The Critical Section Problem

## 3.2 I Semafori

I semafori sono un'altra struttura dati che fornisce mutua esclusione alle sezioni critiche. Può inoltre svolgere il ruolo di contatore atomico.

Tipo speciale di variabile (intera) che supporta 2 operazioni atomiche

- **wait()** (anche **P()**): decremento, blocco fino all'apertura del semaforo
- **signal()** (anche **V()**): incremento, consente l'ingresso di un altro thread

A ciascun semaforo è associata una queue di processi/thread in attesa. Quando **wait()** viene chiamato da un thread, se il semaforo è aperto il thread continua, altrimenti il thread si blocca in coda. Successivamente **signal()** apre il semaforo. Se un thread è in attesa in coda il thread viene sbloccato, mentre se non ci sono thread in attesa in coda, il segnale viene ricordato per il thread successivo. In altre parole, **signal()** è stateful e ha una "history".

Esistono due categorie di semafori:

- **Binary Semaphore** detti Mutex<sup>22</sup> (lo stesso di un Lock), hanno il compito di garantire l'accesso mutuamente esclusivo a una risorsa (ad esempio, solo un processo/thread esegue in una sezione critica). Ad essi vengono assegnati la variabile intera associata può assumere solo 2 valori: 0 o 1. Viene inizializzato per aprire (ad esempio, valore = 1).
- **Conteggio Semaforo**, utile per gestire più risorse condivise. Il valore iniziale del semaforo è solitamente il numero di risorse. Un processo può accedere a una risorsa purché ne sia disponibile almeno una.

```
// Semaphore S
S.wait(); // wait until S is available
<critical section>
S.signal(); notify other processes that S is open
```

Figura 62: Ogni semaforo supporta una coda di processi in attesa di accedere alla sezione critica (es. per acquistare latte)

Se un processo esegue **S.wait()** e il semaforo S è aperto (diverso da zero), continua l'esecuzione, altrimenti il sistema operativo mette il processo in coda di attesa. Un **S.signal()** sblocca un processo sulla coda di attesa del semaforo S.

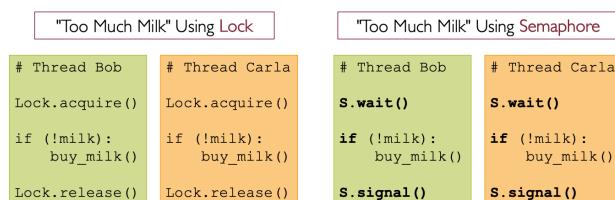


Figura 63: Esempio semaforo binario

<sup>22</sup>In sostanza i Mutex sono utili solo per gestire la mutua esclusione di alcune risorse condivise o di pezzi di codice

Vediamo ora l'implementazione del semaforo:

```

Class Semaphore {
    public void wait(Thread t);
    public void signal();
    private int value;
    private int guard;
    private Queue q;

    Semaphore(int val) {
        // initialize semaphore
        // with val and empty queue
        this.value = val;
        this.q = null;
    }
}

public void wait(Thread t) {
    while(testiset(this.guard) == 1) {
        // while busy do nothing
    }
    this.value -= 1;
    if(this.value < 0) {
        q.push(t);
        t.sleep_and_reset_guard_to_0();
    } else {
        this.guard = 0;
    }
}

public void signal() {
    while(testiset(this.guard) == 1) {
        // while busy do nothing
    }
    this.value += 1;
    if(!q.isEmpty()) // this.value >= 0
        t = q.pop();
    push_onto_ready_queue(t);
    this.guard = 0;
}

```

Figura 64: `wait()` e `signal()` sono istruzioni atomiche

### 3.2.1 Esempi

- Considerando i seguenti due processi A e B, un possibile sviluppo dell'esecuzione è il seguente:

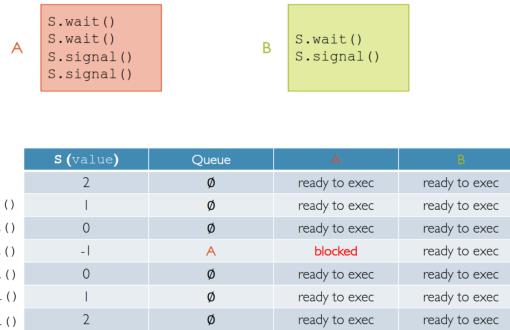


Figura 65

- Consideriamo i due seguenti processi Producer e Consumer:

```

Producer Process:
while (true)
{
    /* produce an item in nextProduced */
    while (counter == BUFFER_SIZE)
        ; /* do nothing */
    buffer[in] = nextProduced;
    in = (in + 1) % BUFFER_SIZE;
    counter++;
}

Consumer Process:
while (true)
{
    while (counter == 0)
        ; /* do nothing */
    nextConsumed = buffer[out];
    out = (out + 1) % BUFFER_SIZE;
    counter--;
    /* consume the item in nextConsumed */
}

```

Figura 66

- I due processi condividono un buffer comune. La variabile counter tiene traccia del numero di elementi attualmente nel buffer.
- In tale caso, potrebbe crearsi una race condition: in assenza di sincronizzazione, i due processi potrebbero lavorare contemporaneamente sulla variabile
- Ad esempio, ipotizziamo che inizialmente si abbia  $counter = 5$ . In tal caso, una possibile esecuzione del programma potrebbe essere la seguente:

**Producer:**

```
register1 = counter
register1 = register1 + 1
counter = register1
```

**Consumer:**

```
register2 = counter
register2 = register2 - 1
counter = register2
```

**Interleaving:**

$T_0$ :	producer	execute	$register1 = counter$	$\{register1 = 5\}$
$T_1$ :	producer	execute	$register1 = register1 + 1$	$\{register1 = 6\}$
$T_2$ :	consumer	execute	$register2 = counter$	$\{register2 = 5\}$
$T_3$ :	consumer	execute	$register2 = register2 - 1$	$\{register2 = 4\}$
$T_4$ :	producer	execute	$counter = register1$	$\{counter = 6\}$
$T_5$ :	consumer	execute	$counter = register2$	$\{counter = 4\}$

Figura 67

**3.2.2 Problemi semafori**

Non è facile ottenere il significato di attesa/segnalazione su un semaforo, infatti si tratta essenzialmente di variabili globali condivise. Non esiste inoltre una connessione diretta tra il semaforo e i dati ai quali il semaforo controlla l'accesso. Servono a molteplici scopi (ad esempio, mutex, vincoli di pianificazione, ecc...).

La loro correttezza dipende dall'abilità del programmatore.

I semafori possono creare deadlock, ossia una situazione in cui due o più processi o azioni si bloccano a vicenda, aspettando che uno esegua una certa azione (es. rilasciare il controllo su una risorsa come un file, una porta input/output ecc.) che serve all'altro e viceversa.

### 3.3 I Monitor

Un **monitor** è un costrutto del linguaggio di programmazione che controlla l'accesso ai dati condivisi. E' simile a una classe (Java/C++) che incorpora tutto insieme: dati, operazioni e sincronizzazione. Il codice di sincronizzazione viene aggiunto dal compilatore, applicato in fase di esecuzione. Diversamente dalle classi, i monitor garantiscono la mutua esclusione, cioè solo un thread alla volta può eseguire il metodo di un monitor, inoltre richiedono che tutti i dati siano privati.

I monitor definiscono un blocco e zero o più variabili di condizione per la gestione dell'accesso simultaneo ai dati condivisi. Utilizza il blocco per garantire che all'interno del monitor sia attivo un solo thread alla volta. Il blocco prevede, come anticipato prima, l'esclusione reciproca per i dati condivisi.

È semplice trasformare una classe Java in un monitor semplicemente, basta rendere privati tutti i dati, sincronizzare tutti i metodi (o quelli non privati).

```
class Queue {
    ...
    private ArrayList<Item> data;
    ...

    public void synchronized add(Item i) {
        data.add(i);
    }

    public Item synchronized remove() {
        if (!data.isEmpty()) {
            Item i = data.remove(0);
            return i;
        }
    }
}
```

Figura 68: La parola chiave *synchronized* indica che il metodo è soggetto a mutua esclusione.

Nell'esempio precedente, il metodo `remove()` deve attendere finché qualcosa non è disponibile nella coda. Intuitivamente, il thread dovrebbe dormire all'interno della sezione critica. Ma se il thread dorme mentre mantiene ancora un blocco, nessun altro thread può accedervi alla coda, aggiungi un elemento ad essa e alla fine riattiva il thread dormiente. Le variabili condizionali sono delle soluzioni che risolvono questo problema. Consentono a un thread di dormire all'interno di una sezione critica. Qualsiasi blocco mantenuto dal thread viene rilasciato atomicamente prima di andare a dormire.

Ogni variabile<sup>23</sup> di condizione supporta tre operazioni:

- `wait` release lock e vai a dormire atomicamente (coda di camerieri)
- `signal` sveglia un thread in attesa se ne esiste uno, altrimenti non fa nulla
- `broadcast` risveglia tutti i thread in attesa

Il thread deve mantenere il blocco durante l'esecuzione di operazioni sulle variabili di condizione.

Le variabili di condizione hanno le stesse operazioni dei semafori ma semantica completamente diversa:

- L'accesso al monitor è controllato da una serratura
- `wait()` blocca il thread chiamante e rinuncia al blocco
- per chiamare `wait()`, il thread deve essere nel monitor (quindi, ha il blocco)
- su un semaforo, `wait()` blocca solo il thread sulla coda
- `signal()` provoca l'attivazione di un thread in attesa
- Se non c'è un thread in attesa, il segnale viene comunque perso!
- su un semaforo, il segnale aumenta il contatore, consentendo l'ingresso futuro anche se nessun thread è attualmente in attesa

<sup>23</sup>Nota: le variabili di condizione non sono oggetti booleani

Comunemente, i monitor vengono implementati secondo due stili:

- Lo stile Mesa, dove il thread segnalante inserisce un thread in attesa nella ready queue mentre il primo continua l'esecuzione all'interno del monitor, e dove la condizione, la quale deve essere verificata continuamente, non deve essere necessariamente verificata quando il thread in attesa viene nuovamente eseguito

```
class Queue {
    ...
    private ArrayList<Item> data;
    ...

    public void synchronized add(Item i) {
        data.add(i);
        notify();
    }

    public Item synchronized remove() {
        while (data.isEmpty()) {
            wait(); // give up the lock and sleep
        }
        Item i = data.remove(0);
        return i;
    }
}
```

Figura 69

- Lo stile Hoare, dove il thread segnalante viene trasformato immediatamente in un thread in attesa e dove la condizione anticipata dal thread in attesa è garantita quando esso viene eseguito

```
class Queue {
    ...
    private ArrayList<Item> data;
    ...

    public void synchronized add(Item i) {
        data.add(i);
        notify();
    }

    public Item synchronized remove() {
        if (data.isEmpty()) {
            wait(); // give up the lock and sleep
        }
        Item i = data.remove(0);
        return i;
    }
}
```

Figura 70

### 3.3.1 Problemi di Lettura e Scrittura

Un oggetto è condiviso tra più thread, ciascuno appartenente a una delle due classi:

- Reader: legge i dati, non li modifica mai
- Scrittori: leggono i dati e li modificano

La soluzione più semplice suggerisce di utilizzare un singolo blocco sull'oggetto dati per ciascuna operazione, ma potrebbe risultare troppo restrittivo.

Esistono due varianti del problema a seconda che la priorità sia sui lettori o sugli scrittori:

- primo problema di lettori-scrittori (priorità ai lettori), se un lettore desidera accedere ai dati e non vi è già un utente che vi accede, allora l'accesso è consentito al lettore. Esiste anche una possibile "fame" degli scrittori, in quanto potrebbero esserci sempre più lettori in arrivo per accedere ai dati
- secondo problema lettori-scrittori (priorità agli scrittori), si verifiva quando un writer vuole accedere ai dati, salta in testa alla coda. Esiste anche una possibile "fame" dei lettori, in quanto sono tutti bloccati finché ci sono scrittori.

### 3.3.2 (First) Readers-Writers Problem: Soluzione I

E' possibile utilizzare un contatore e due semafori binari:

- `numReaders`, utilizzato dai processi reader per contare il numero di lettori attualmente accedendo ai dati
- semaforo `mutex` binario, utilizzato solo dai lettori per l'accesso controllato `numReader`
- `rw_mutex`, binario semaforo utilizzato per bloccare e rilasciare i writer

Il primo lettore che arriva bloccherà su `rw_mutex` se c'è attualmente un utente che sta accedendo ai dati. Tutti i lettori successivi si bloccheranno solo sul `mutex` per il loro turno di incremento `numReader`.

Il primo lettore si blocca se è presente un writer; qualsiasi altro lettore che tenta di entrare viene bloccato su `mutex`. Solo l'ultimo lettore ad uscire segnala uno scrittore in attesa. Quando uno scrittore esce, se c'è sia un lettore che uno scrittore in attesa, dipende da chi va dopo sul programmatore.

Se uno scrittore esce e un lettore va dopo, allora tutti i lettori che sono in attesa falliranno (almeno uno è in attesa su `rw_mutex` e zero o più possono essere in attesa su `mutex`). In alternativa, lascia che uno scrittore entri prima nella sua sezione critica (priorità per gli scrittori).

### 3.3.3 Soluzione II: usare i Monitor

## 3.4 Deadlock

Consideriamo il famoso problema dei filosofi a cena:

- Ci sono 5 filosofi seduti ad un tavolo rotondo
- Ogni filosofo ha una bacchetta di legno alla propria sinistra, per un totale di 5 bacchette
- Per poter mangiare, ogni filosofo necessita di due bacchette
- L'obiettivo è far sì che ogni filosofo mangi

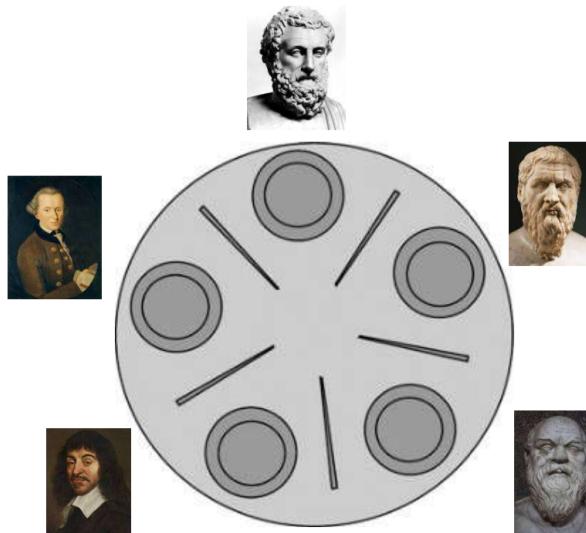


Figura 71

- Esiste una prima soluzione banale al problema: possiamo utilizzare un lock globale che permette ad un singolo filosofo per volta di poter prendere in mano due bacchette. Tale soluzione, risulta essere inefficiente per via dell'assenza di concorrenza tra i filosofi affinché essi possano mangiare contemporaneamente
- Una seconda soluzione prevede l'uso di un semaforo:

```
Semaphore chopsticks[5];

while(True) {
    chopsticks[i].wait();           // wait on the left chopstick
    chopsticks[(i+1)%5].wait();    // wait on the right chopstick

    eat();

    chopsticks[i].signal();        // signal on the left chopstick
    chopsticks[(i+1)%5].signal(); // signal on the right chopstick

    think();
}
```

Figura 72

- Tale soluzione presenta alcune problematiche: nel caso in cui ogni filosofo prenda in mano la bacchetta sinistra in contemporaneamente agli altri, si andrebbe a creare un **deadlock**, dove ogni filosofo rimarrebbe in attesa che la bacchetta destra sia libera.
- Per far sì che non ci crei un deadlock e che vi sia concorrenza tra i filosofi, una terza soluzione prevede l'uso dei **monitor**: prima di prendere in mano una delle due bacchette, è necessario assicurarsi che la seconda sia libera, altrimenti sarà necessario aspettare che entrambe siano libere. Dunque, sarà necessario controllare se il filosofo alla propria destra e il filosofo alla propria sinistra stiano mangiano oppure no (variabili condizionate), portando nessun filosofo a prender in mano una singola bacchetta.

I problemi che abbiamo visto finora sono interessanti perché identificano alcuni modelli che sono molto comuni nella pratica: **Produttore-Consumatore**, **Lettore audio/video** incorporato in un browser Web che contiene un buffer dati condiviso + rete e rendering, Lettore-Scrittore (E.g sistema bancario: read vs. update account balances), oppure i **Filosofi a tavola** che bloccano più risorse: (ad esempio, prenotazioni di viaggi (hotel, compagnie aeree, database di autonoleggio)).

### 3.4.1 Cos'è un deadlock

Intuitivamente, una condizione in cui due o più thread sono in attesa di un evento che può essere generato solo dagli stessi thread. I deadlock possono verificarsi sia su risorse hardware sia su risorse software. Si può verificare quando più thread competono per un numero finito di risorse:

- Rilevamento deadlock: trova istanze di deadlock e tenta di ripristinarle
- Prevenzione deadlock (offline): impone restrizioni/regole su come scrivere programmi senza deadlock
- Evitamento dei deadlock (online): il supporto runtime controlla le richieste di risorse effettuate dai thread per evitare i deadlock

Il **deadlock** è diverso dalla **starvation** i termini correlati ma ognuno si riferisce a una situazione specifica. La starvation si verifica quando un thread attende a tempo indeterminato una risorsa, ma altri thread stanno effettivamente facendo progressi utilizzando quella risorsa. La **differenza principale** con deadlock è che il sistema non è completamente bloccato.

Il deadlock può verificarsi se tutte e quattro le condizioni sottostanti sono soddisfatte:

- **Mutua esclusione**: almeno un thread deve contenere una risorsa non condivisibile (solo un thread contiene la risorsa)
- **Hold and Wait** almeno un thread detiene una risorsa non condivisibile ed è in attesa che altre risorse diventino disponibili (un altro thread detiene le risorse)
- **No Preemption** un thread può rilasciare una risorsa solo volontariamente; né un altro thread né il sistema operativo possono forzarlo a rilasciare la risorsa
- **Circular Wait** un insieme di thread in attesa  $t_1, \dots, t_n$  dove  $t_i$  è in attesa su  $t_{(i+1)\%n}$

### 3.4.2 Resource Allocation Graph (RAG)

Definiamo un **grafo orientato**  $G = (V, E)$  dove:

- $V$  è l'insieme dei vertici che rappresentano sia le risorse  $r_1, \dots, r_m$  che i thread  $t_1, \dots, t_n$
- $E$  è l'insieme degli archi tra risorse e thread

I bordi possono essere di 2 tipi:

- **Archi di richiesta** un edge diretto  $(t_i, r_j)$  indica che  $t_i$  ha richiesto  $r_j$ , ma non ancora acquisita
- **Archi di assegnamento** un edge diretto  $(r_j, t_i)$  indica che il sistema operativo ha assegnato  $r_j$  a  $t_i$

### 3.4.3 Modellazione dei deadlock

Holt (1972) mostrò come modellare queste quattro condizioni usando i grafi orientati i grafi hanno due tipi di nodi:

- i **processi**, mostrati come **cerchi**
- le **risorse**, mostrate come **quadrati**

Un **arco**, con una **freccia** a indicarne la direzione. Da un nodo risorsa (un quadrato) a un nodo processo (un cerchio) significa che la risorsa è stata in precedenza richiesta, assegnata e attualmente posseduta da quel processo.

### 3.4.4 Esempio

1. Prendiamo il seguente esempio:

- Consideriamo il seguente RAG:

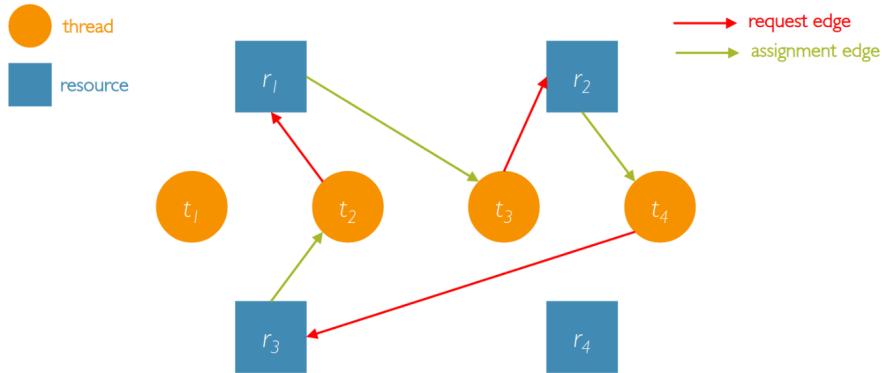


Figura 73

- E' facilmente individuabile la presenza di un deadlock:
  - Il thread  $t_2$ , possidente la risorsa  $r_3$ , richiede la risorsa  $r_1$ , rimanendo in attesa che essa sia liberata dal thread  $t_3$
  - A questo punto, il thread  $t_3$ , possidente la risorsa  $r_1$ , richiede la risorsa  $r_2$ , rimanendo in attesa che essa sia liberata dal thread  $t_4$
  - A sua volta, il thread  $t_4$ , possidente la risorsa  $r_2$ , richiede la risorsa  $r_3$ , rimanendo in attesa che essa sia liberata dal thread  $t_2$
  - In definitiva, si è in una situazione dove  $t_2$  aspetta  $t_3$ , il quale sta aspettando  $t_4$ , il quale a sua volta sta aspettando  $t_2$ , creando quindi un deadlock
- Se il thread  $t_4$  non richiedesse la risorsa  $r_3$ , le condizioni necessarie affinché possa verificarsi un deadlock non sarebbero soddisfatte, per via dell'assenza dell'attesa circolare

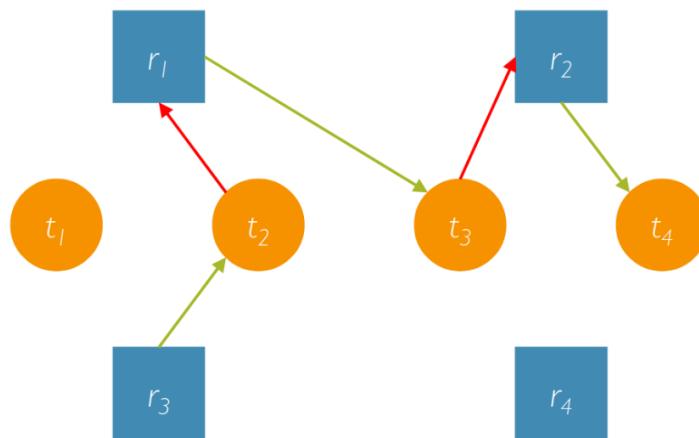


Figura 74

2. Prendiamo ora il seguente esempio:

- Consideriamo il caso in cui esistano due istanze della risorsa  $r_3$ , dove prima istanza è assegnata al thread  $t_2$  e la seconda è assegnata al thread  $t_3$ .

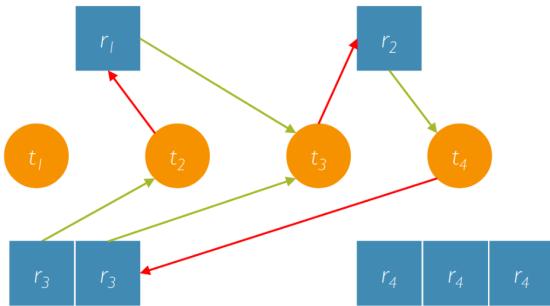


Figura 75

- Anche in tal caso, siamo in una situazione di deadlock:

- (a)  $t_4$  richiede un'istanza di  $r_3$ , rimanendo in attesa che  $t_2$  o  $t_3$  ne rilascino almeno una
- (b) Tuttavia,  $t_3$  è in attesa che  $t_4$  ceda  $r_2$ , mentre  $t_2$  è in attesa che  $t_3$  ceda  $r_1$

3. Invece, se ci fossero tre istanze di  $r_3$  e solo due di esse venissero utilizzate da  $t_2$  e  $t_3$ , non si andrebbe a creare un deadlock, poiché  $t_4$  non rimarrebbe in attesa, sbloccando quindi la catena

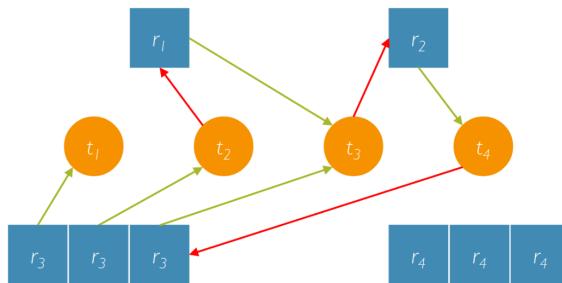


Figura 76

## 3.5 Prevenire ed evitare un deadlock

Bisogna scansionare il grafico di allocazione delle risorse (RAG) per i cicli e poi rompili.

Esistono diversi modi per farlo:

- Uccidere tutti i thread nel ciclo ()
- Uccidere tutti i thread uno alla volta, forzando ciascuno di essi a rilasciare risorse
- Anticipa le risorse una alla volta tornando a uno stato coerente (ad es. nelle transazioni di database)

Come visto nella sezione precedente, per prevenire un deadlock è sufficiente che una sola delle quattro condizioni necessarie non si verifichi.

Inoltre, è possibile studiare il comportamento di una sequenza di  $n$  thread per determinare se tale sequenza sia sicura o non:

- Sia  $m_i$  il numero massimo di risorse che il thread  $i$  possa richiedere
- Sia  $c_i$  il numero di risorse attualmente occupate dal thread  $i$
- Sia  $C = \sum_{i=1}^n c_i$  il numero totale di risorse attualmente allocate nel sistema
- Sia  $R$  il numero massimo di risorse disponibili

- Definiamo una sequenza di thread come sicura se per ogni thread si verifica che:

$$m_i - c_i \leq R - C + \sum_{j=1}^{i-1} c_j$$

- dove:
  - $m_i - c_i$  è il numero di risorse ancora richiedibili da  $t_i$
  - $R - C$  è il numero di risorse attualmente disponibili
  - $\sum_{j=1}^{i-1} c_j$  è il numero di risorse attualmente allocate fino al thread  $t_j$ , dove  $j < i$

### 3.5.1 Stato sicuro

L'idea alla base della policy di allocazione delle risorse è quella di garantire la sicurezza del sistema attraverso la definizione di uno **stato sicuro**. Uno stato sicuro si verifica quando esiste una sequenza di esecuzione dei thread che permette a tutti i thread di terminare la loro esecuzione senza causare uno stato di interblocco. Tuttavia, la presenza di uno **stato insicuro** non implica necessariamente la presenza di un deadlock, poiché alcuni thread potrebbero non richiedere simultaneamente il massimo numero di risorse necessarie dichiarato.

Per garantire la sicurezza del sistema, la policy prevede che una risorsa venga assegnata ad un thread solo se, dopo l'allocazione, il sistema risulta in uno stato sicuro. In caso contrario, il thread viene messo in attesa della risorsa fino a quando non diventa disponibile in modo sicuro. Questa politica di gestione delle risorse previene la creazione di attese circolari, in cui i thread attendono risorse che non sono disponibili causando uno stato di interblocco del sistema.

### 3.5.2 Archi di pretesa

Oltre alla tipologia di archi di assegnamento che rappresentano l'allocazione effettiva di una risorsa ad un thread, viene utilizzata una variante del RAG che prevede anche archi di pretesa. Gli **archi di pretesa** indicano che un thread  $t_i$  potrebbe richiedere in futuro la risorsa  $r_j$ , senza però richiederla immediatamente.

Nella gestione delle risorse, soddisfare una pretesa significa trasformare un arco di pretesa in un arco di assegnamento, cioè assegnare effettivamente la risorsa al thread che la richiede. La presenza di cicli nel grafo indica un possibile stato insicuro, in cui alcuni thread attendono risorse che altri thread detengono. Se l'assegnamento di una risorsa generasse uno stato insicuro, tale assegnamento non verrà effettuato anche nel caso in cui la risorsa sia effettivamente disponibile, in quanto potrebbe causare un interblocco del sistema. In questo modo, la gestione delle pretese aiuta a prevenire la creazione di situazioni di interblocco e a garantire la sicurezza del sistema.

### 3.5.3 Esempi

- Consideriamo il seguente RAG:

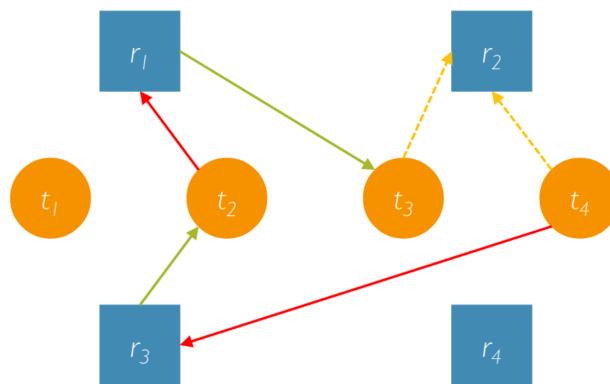


Figura 77

- Supponiamo che venga soddisfatta la pretesa del thread  $t_4$ . In tal caso, si andrebbe a creare uno stato insicuro poiché, nel caso in cui anche il thread  $t_3$  vada a richiedere la risorsa  $r_3$  (dunque trasformando l'arco di pretesa in un arco di richiesta) si andrebbe a creare un deadlock. Dunque tale pretesa non verrebbe soddisfatta.

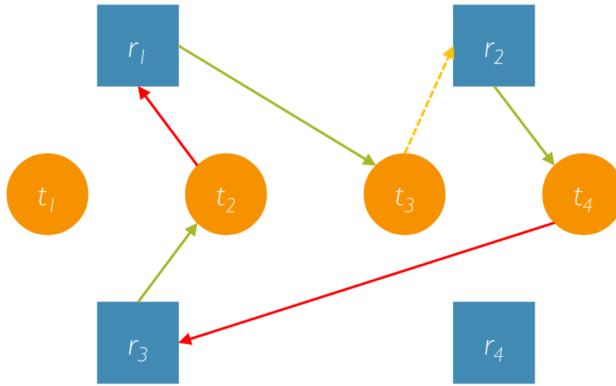


Figura 78

- Nel caso in cui invece venga soddisfatta la pretesa del thread  $t_2$ , si rimarrebbe in uno stato sicuro poiché, nel caso in cui il thread  $t_4$  andasse a richiedere la risorsa  $r_3$  esso rimarrebbe in attesa di  $t_3$ , il quale può continuare a lavorare poiché non è in attesa di nessuno. Dunque tale pretesa verrebbe soddisfatta.

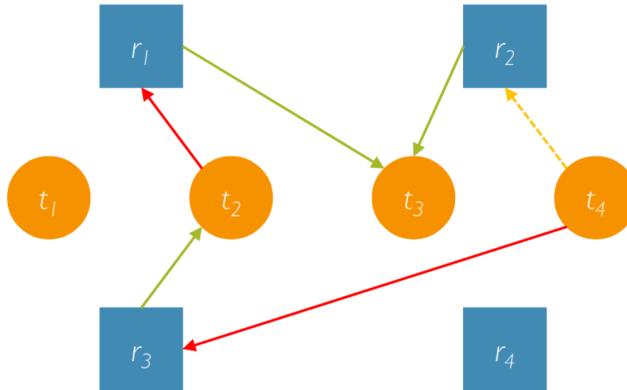


Figura 79

## 4 Gestione della Memoria

## 5 Gestione dei Sistemi di I/O

## 6 File System

## 7 Advanced Topics

## 8 Esercizi

## 9 Formulario