**Esame OS parte 1**

Definizione di sistema operativo

*È un software che controlla l’esecuzione dei software ed interfaccia il software con l’hardware*

Definizione di Processo

*Un processo è un programma in esecuzione e le risorse che utilizza e il contesto vengono usate,*

*la memoria usata è detto spazio d’indirizzamento, è l’insieme delle celle di memoria usate dal processo.*

Il contesto di esecuzione comprende:

* Il contesto hardware (program counter, stack pointer …)
* Spazio d’indirizzamento
* Sistema di controllo delle informazioni
* Credenziali

In Unix il processo è un oggetto, una struct.

Programma è diverso da processo

Il programma è il software su disco, entità passiva

Il processo è il programma in esecuzione, entità attiva

Un sistema multi-programmato ha più di un processo in memoria allo stesso tempo

*La differenza tra processo e thread è che il thread ha solo le informazioni che li servono.*

Se un processo ha più threads di esecuzione si ha un multi-threaded process altrimenti è un single-threaded process

I thread possono essere implementati a livello kernel oppure a livello utente.

**Kernel level**

A livello kernel, il kernel sa che i thread sono dei thread e il loro scheduling è gestito dal kernel.

Se un thread si blocca gli altri thread continuano l’esecuzione

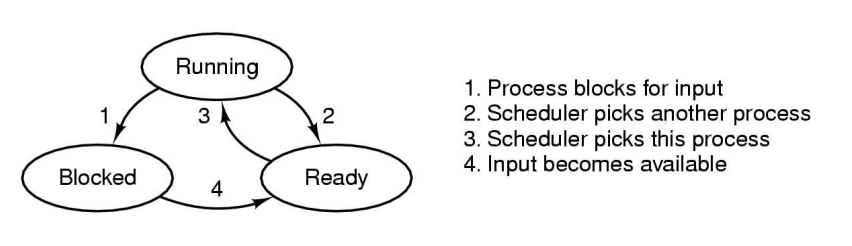
**User level**

Il kernel non sa niente dei thread, lo scheduling è gestito dal processo, quando il processo viene schedulato dal kernel si da una chance ai thread di essere eseguiti.

A differenza del kernel level se un thread si blocca, tutto il processo e gli altri thread di esso vengono bloccati.

Un processo può essere in 3 stati:

* Ready
* Running
* Blocked



**Creazione dei processi**

Ogni processo è identificato da un identificativo **PID**

Tramite il fork si crea un processo figlio, ad esso viene assegnato un nuovo PID, crea un nuovo contesto e copia al suo interno lo spazio di indirizzamento del padre nel figlio.

Il fork restituisse:

* 0 se è un figlio
* Il PID se è il padre
* -1 se il fork fallisce

Root è il processo init che ha PID=1 ed è creato in fase di boot

Si può ottenere l’id del padre con getppid

**Il fork consente di eseguire la duplicazione dello stesso processo**

Viene creato un nuovo contesto di esecuzione

Per poter avviare altri processi si usa execl, questa funzione sostituisce il codice dell’attuale istanza con il codice caricato, di conseguenza imposta stack e heap, per il nuovo processo.

Non viene effettivamente creato un nuovo processo.

**Terminazione dei processi**

I processi terminano 2 casi:

* Tramite return
* Tramite exit(int)

Il valore di ritorno può essere letto dal processo padre.

**Attendere un processo**

Un processo può attendere un processo figlio con wait e waitpid

La chiamata wait è bloccante

**Processi Zombie**

A meno che il processo padre non attenda i processi figli, quest’ultimi vanno in uno stato zombie, e non possono essere rimossi dalla lista dei processi.

Un processo zombie non può essere più schedulato dal sistema.

Se il padre termina il figlio è ereditato da init (PID=1), questo comportamento è chiamato re-parenting ed è usato per creare unix deamons.

SIGCHLD è un segnale che viene inviato al padre quando il figlio termina->(SIGHLD child stopped or terminated), consente la prosecuzione con il wait

Il processo figlio è chiamato processo orfano, se il figlio è uno zombie, il padre è terminato

Se il figlio era in uno stato zombie, l’init provvederà a terminarlo facendo wait

Lo stato zombie è uno stato da evitare

**Interazione processi-sistema**

il kernel espone le funzionalità che implementa tramite un API delle system calls

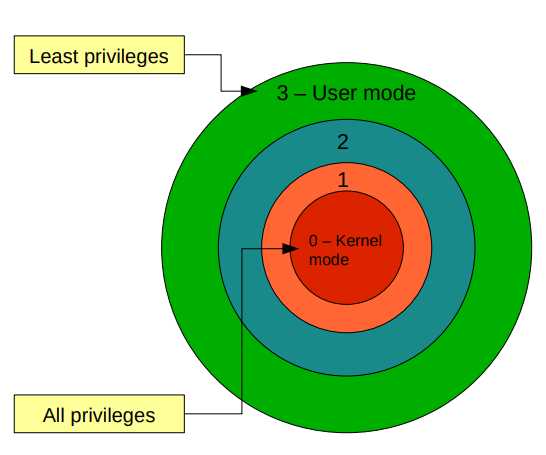
le API seguono lo standard POSIX, in fase di compilazione.

Un sistema operativo deve usare un sistema robusto per proteggere i vari servizi (spazi di indirizzamento, autenticazione, controllo accessi…)

Alcune istruzioni del kernel, come quella per disabilitare gli interrupt non devono essere disponibili all’utente.

Esistono due tipi di esecuzione:

* **Kernel mode o privileged**: si ha il completo controllo dell’hardware e un’operazione errata rischia il crash dell’intero sistema
* **User mode o unprivileged:** l’accesso all’hardware avviene tramite system call e gli errori non influiscono sugli altri processi

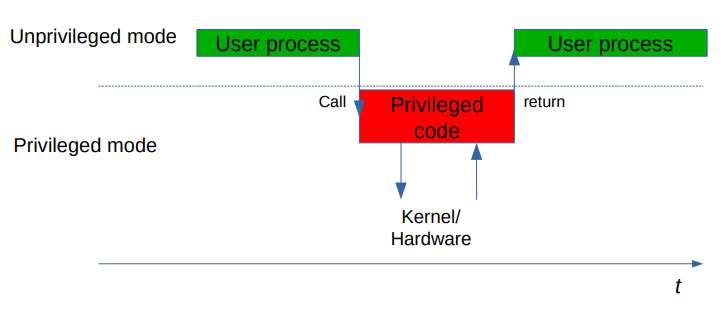
Alle aree di memoria possono essere assegnati dei livelli di priorità, se l’area di memoria è privilegiata e un processo non privilegiato tenta di accedere l’accesso genera un errore.

Su x86 il grado di privilegio si ottiene leggendo i registri della cpu.

In fase di boot la memoria è privilegiata per ad esempio caricare i driver.

Per utente root il kernel esegue il codice in modalità non privilegiata, per eseguire.

Per creare funzionalità kernel bisogna creare dei moduli kernel.



Per ragioni di sicurezza non si può saltare da un’area privilegiata a una non privilegiata e viceversa.

Si può fare tramite dei gate ossia degli indirizzi specifici che consentono di spostare i dati tra le zone.

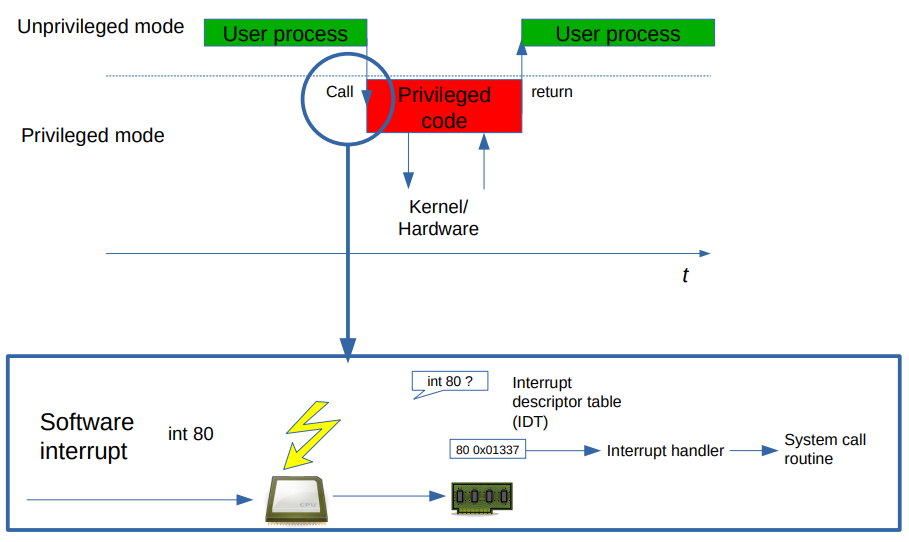
**Interrupt**

Gli interrupt interrompono l’esecuzione di un processo dando il controllo al OS e salta a una handling routing predefinita, anche codice non privilegiato può generare degli interrupt, ma la handling routine è determinata dal kernel.

Esistono interrupt:

* Hardware
* Software

Le chiamate di sistema sono degli interrupt software.



Gli interrupt sono operazioni lente, poiché devono salvare e ripristinare il contesto.

Principio del privilegio minimo: il programma deve avere solo i privilegi che necessita, usando solo le risorse necessarie

Ci sono 3 tipi di architetture di kernel:

* Monolitico: i software unico che viene eseguito in modalità privilegiata
* Microkernel: solo una minima parte è gestita in modalità privilegiata
* Hybrid Kernel: alcune parti sono a livello kernel altre no

**Scheduling**

Nei sistemi multi-programmati è presente uno scheduler che ha il compito di determinare l’ordine in cui i processi/threads ottengono le risorse (solitamente la cpu), l’ordine dipende dalla policy/algorithm di scheduling.

Uno scheduler può essere:

* **Cooperative multitasking**: ogni processo coopera con altri rilasciando la cpu dopo un tempo predefinito
* **Preemptive multitasking:** il sistema gestisce il rilascio della cpu per ogni processo

Lo scheduler viene chiamato in diverse situazioni:

* Quando un processo viene creato
* Quando un processo termina
* Quando un processo attende una risorsa
* Quando una risorsa diventa disponibile

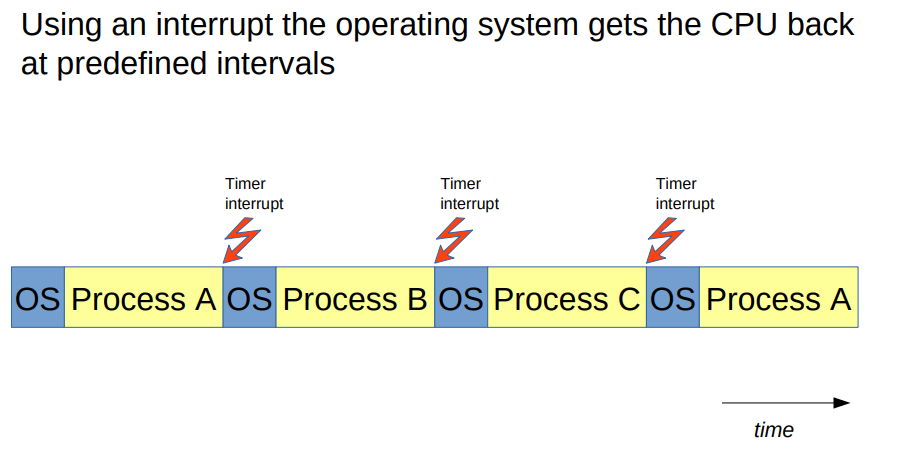
Il comportamento dei processi influenza lo scheduler:

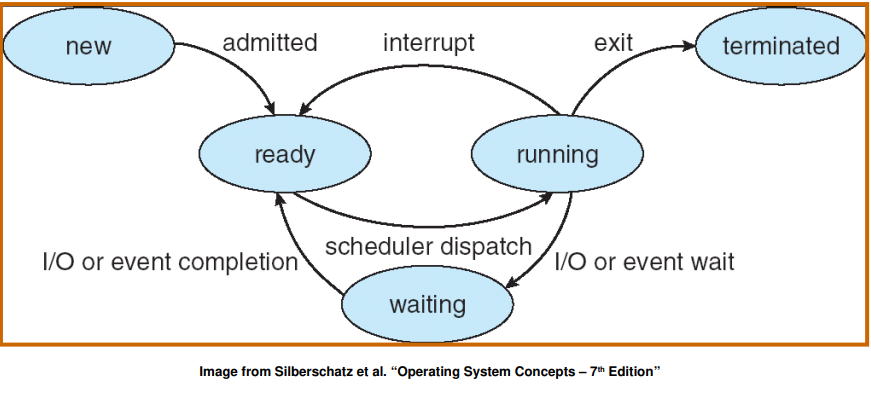
* **Processi I/O Bound**: sono processi con frequenti input e output, quindi numero interruzioni numerose (programmi GUI)
* **Processi CPU Bound**: sono processi dove richiedono la cpu in modo intenso, lo schedulere deve bloccare i processi dopo x tempo per garantire l’equità

L’hardware ha un sistema di segnali che abilitano le periferiche a interrompere la cpu in modo asincrono.

**Interrupt handler:** funzione richiamata da uno specifico interrupt

Il sistema operativo tramite un interrupt riprende il controllo della CPU a intervalli predefiniti.





Ci sono 3 tipi di scheduling:

* Long term scheduling
* Medium term scheduling (Swap memory)
* Short term scheduling

**Ambienti di computazione**

* **Batch**: lunghi tempi di computazione, no interazione utente
* **Interattivo**: più utenti allo stesso tempo
* **Realtime**: scheduler deve garantire un certo tempo di risposta

**Parametri di valutazione dello scheduler**

* **Throuput:** numero di lavori per unità di tempo
* **Turnaround time:** tempo medio richiesto per completare un lavoro
* **Response time:** tempo tra ricezione di un comando e ottenimento del risultato
* **Predicibilità e regolarità:** il tempo di risposta deve essere garantito

**Obbiettivi comuni degli scheduler**

* **Equità**: tutti i processi devono poter essere eseguiti
* **Policy enforcement**: le decisioni di scheduling devono essere rispettate
* **Bilanciamento**: massimizzare l’uso delle risorse

**Algoritmi di scheduling**

**Sistemi Batch**

**FCFS (First-come First-served)**

Ogni processo usa la cpu per quanto ne necessita, i nuovi lavori vengono messi in coda

Se un processo si blocca, viene interrotto ed eseguito il successivo.

Quando un processo è nello stato di ready viene messo alla fine della coda.

**SJF (Shortest Job First)**

Vengono prioritizzati i lavori brevi (vanno stimati, tempo aggiuntivo)

Questo tipo di algoritmo massimizza il throughput e il turnaround, ma richiede tempo di calcolo aggiuntivo per le stime, e può finire in una situazione di starvation per i processi più lunghi.

**SRTN (Shortest remaining time next)**

Da la priorità ai processi che hanno tempo di esecuzione rimanente minore (variante del SJF)

**Sistemi interattivi**

**Round Robin**

I processi sono organizzati in liste circolari.

Lo scheduler assegna ad ogni processo uno slot di tempo detto quantum.

Quando il quantum finisce la CPU passa a un altro processo in stato di ready.

Se il processo cede la CPU, essa viene assegnata ad un altro processo.

La preemption è l’interruzione momentanea di un task che verrà ripreso successivamente, esso avviene dopo x tempo per poter garantire l’esecuzione di tutti i processi. Lo gestisce lo scheduler.

Durante la preemption avvengono i context switch, la lunghezza del quantum deve tenere conto di impiegare troppo tempo, si può ridurre il quantum ma si deve determinare anche il tempo di context switch.

**Priorità dei processi**

Il round-robin di base non tiene conto dell’importanza dei vari processi.

La soluzione è introdurre le priorità tramite delle code multiple di priorità.

Ogni livello di priorità ha una coda dedicata.

Le code multiple per evitare la starvation dei processi con bassa priorità vengono adattate in modo dinamico, nonostante ciò, si rischia la starvation se ci sono tanti processi con priorità alta.

**Code circolari**

Le code di priorità sono circolari ed i processi con priorità più alta avranno un quantum più alto, mentre quelli con priorità minore avranno un peggior tempo di risposta.

**MLFQ (Multilevel Feedback Queue)**

Ogni livello di priorità è linkato in una coda round-robin, fin quando ci sono processi nella coda con priorità più alta eseguili, altrimenti passa a una coda con priorità inferiore.

A dipendenza del comportamento dei vari processi, essi vengono spostati nelle varie code.

Se un lavoro esaurisce il quantum, viene abbassata la priorità quando viene rischedulato (fa molti yield della cpu)

Per evitare la starvation dei processi a bassa priorità, periodicamente vengono spostati nelle code di priorità maggiore (boost)

**Lottery scheduling**

Si assegna ad ogni processo un numero di tickets > 1 proporzionale alla priorità del processo stesso.

Periodicamente effettua l’estrazione e il processo vincente viene eseguito.

Vantaggi:

* priorità dinamica
* i processi possono cedere ticket ad altri

**Fair-Share Scheduling**

È una policy che divide il tempo di CPU tra gli utenti, assegnare una priorità agli utenti indipendentemente dal numero di processi.

È possibile combinare algoritmi di Fair-share con tutti gli altri

Gli algoritmi di scheduling usano policy che guardano ai processi e non gli utenti.

In questo modo si ha un’equa distribuzione dell’uso della CPU tra gli utenti.

**Sistemi multicore**

I sistemi multicore hanno una memoria condivisa che fornisce risorse ai vari thread e processi, ciò introduce il problema della località.

La località può essere:

* **Temporale:** quando un dato è accessibile, sarà probabilmente riutilizzato in un futuro vicino
* **Locale:** un dato all’indirizzo di memoria x probabilmente verranno usati anche i dati vicini ad esso.

Quando un thread è eseguito per un istante di tempo la cache corrispondente conterrà più dati di quelli effettivamente necessari.

La cache lavora secondo il principio d località:

infatti, riduce il numero di dati da trasferire dalla RAM (nasconde la latenza della memoria e la banda utilizzata dalla memoria).

Ci sono 2 tipi di architetture di accesso alla memoria principale:

* **UMA (Uniform memory access architetture**): accede ai dati che non sono nella cache locale, l’accesso alla memoria principale è uguale per tutti i processori
* **NUMA(Nonuniform memory access architetture):** accede ai dati che non sono nella memoria privata, l’accesso ad alcune aree della memoria principale è più veloce per alcuni processori rispetto ad altri parti della memoria

Nei sistemi multicore l’accesso ai dati che non sono nelle cache l1 e l2 è molto più lento

**Policy della località degli scheduler**

* **Soft affinity:** lo scheduler prova sempre ad assegnare un thread allo stesso core o a un set di CPU/core specifici
* **Hard affinity:** lo scheduler forza l’assegnamento di un thread a un core specifico or a un set di CPU/core specifici

**Scheduling con code singole**

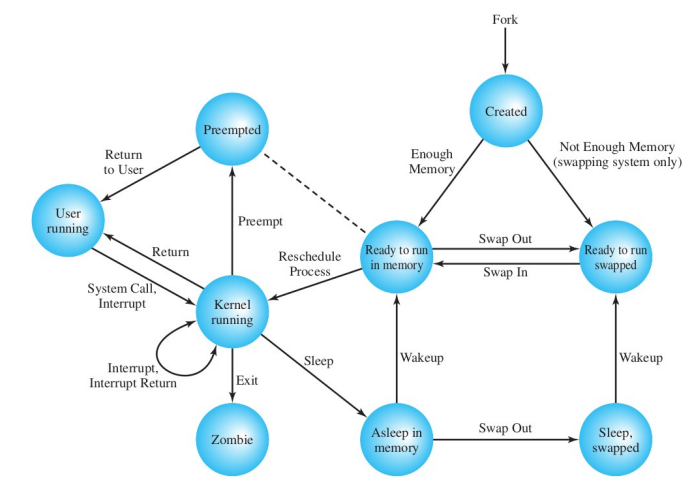
Tutti i processi sono gestiti da una singola coda.

* Tutte le cpu possono essere sfruttate in modo completo
* Non sono scalabili
* Ha problemi di località, i processi vengono messi tra i vari core
* La cache affinity viene compromessa
* Sfruttamento massimo delle risorse

**Scheduling con code multiple**

Ogni processore/core ha una sua coda

* Scalabilità
* La cache affinity è assicurata
* Le code si svuotano a velocità diverse tra i vari core
* Se il core è in idle può essere preso da task di altre code



**Scheduling tradizionale di Unix**

Utilizza un sistema di feedback a più livelli, basati su Round-Robin

La preemption avviene se un processo non cede la cpu prima di un 1 secondo

CPUi(j)= utilizzo del processore da parte del processo j nel intervallo i

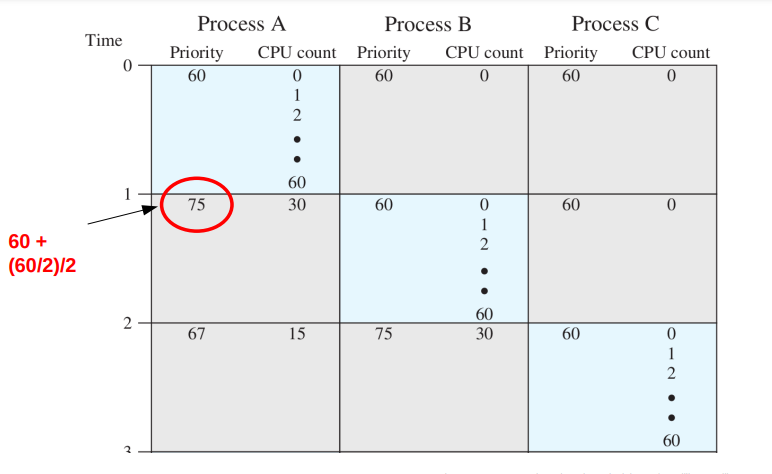
Pj(i)= priorità del processo j all’inizio del processo i, il valore più piccolo è pari alla parità più alta

Basej= priorità base del processo j

Nicej= fattore di aggiustamento

Per velocizzare le code usare i bitmaps (dispatch queues in UNIX)

***\*\*\*nell’esempio il cpu count è la cpu usage\*\*\****



**Unix CFS**

È un algoritmo che cerca di assicurare la fairness a tutti i processi.

Usa un AVL dove ogni nodo rappresenta un task ed il peso indica il tempo consumato usato dalla cpu, esso essendo un AVL viene aggiustato dalla priorità, complessità operazioni O (log n)

Per migliorare la responsività di alcuni thread viene effettuato un boost di priorità per i thread che ne necessita.

**Real time OS (RTOS)**

Un sistema real time è un sistema multi-tasking per che esegue applicazioni con requisiti real time.

In un’applicazione real time la correttezza non dipende solo dal risultato logico, ma anche dal tempo in cui viene dato il risultato.

Un sistema real-time deve essere predicibile, ovvero, bisogna soddisfare i limiti di tempo nello scheduling, se li soddisfa si dice che il sistema è **schedulabile**

Se il pool di processi è dinamico, deve verificare la validità dei requisisti temporali.

**Latenza**

Il tempo che passa tra la chiamata di un evento e la sua risposta è detta latenza.

La latenza può essere generata sia lato hardware come (interrapt, propagazione dei segnali elettrici)

e anche software (handling delle interrupt routine, scheduling, context switch,…)

Preemption significa la capacità del sistema operativo di anticipare o interrompere un'attività attualmente pianificata per eseguire un'attività con priorità più alta.

**Hard Real-Time**

I task **devono** essere completati entro una deadline prestabilita

il non rispetto della deadline non può essere tollerato: una risposta anticipata è inutile e può essere pericolosa.

Esempi: sistemi di controllo critici (autopilot, nuclear reactor control,..)

**Soft Real-Time**

I task devono essere completati entro una deadline, ma le eccezioni sono tollerate, il task può finire prima.

Il non rispetto della deadline non costituisce problemi seri o pericoli.

Esempi: DVD-player

**Firm Real-Time**

La risposta in ritardo è inutile

I task non possono continuare dopo la deadline

Esempio: videoconferenza (trasmissione audio e video)

**Schedulability**

La si raggiunge quando i task raggiungono la deadline, il suo ruolo è importante nella policy di scheduling.

Soft Real-Time

* Priority Scheduling

Hard Real-Time

* Dynamic Scheduling
* Static Scheduling

**Tipi di sistemi real-time**

* **Event driven design**
  + Priority scheduling
  + I task sono eseguiti in risposta a più eventi
  + I task con priorità maggiore prevalgono su quelli con priorità minore
* **Time sharing design**
  + Round Robin scheduling
  + Più deterministico
* **Event triggered**
  + I task sono eseguiti in risposta a un evento esterno (task non periodico)
  + Esempio: Bottone premuto
* **Time triggered**
  + I task sono eseguiti in un certo istante di tempo a intervalli regolari (task periodici)
  + Esempio: lettura sensore ogni t secondi

**Tipi di scheduling**

**Static Scheduling**

Il numero di task e l’ordine di schedulazione sono definiti e a runtime lo scheduling avviene in poco tempo, ma ciò paga in poca flessibilità

**Dynamic Scheduling**

I task possono essere aggiunti o rimossi in modo dinamico dalla coda di scheduling, l’ordine dello scheduling è definito a runtime.

Il sistema deve assicurare la validità della coda ad ogni cambiamento.

L’approccio è flessibile, ma è difficile assicurare la correttezza

**Priority Scheduling**

Ogni task è schedulato in base alla priorità, le priorità possono essere modificate a run-time

Fornisce best-effort performances, ma il risultato dello scheduling è unpredictable.

**Come valutare un algoritmo di scheduling in ambiente Real-Time**

**Slack time o laxity time:**

ti= task

X(ti)= tempo di esecuzione del task

D(ti)= deadline interval del task

Il task viene schedulato con periodo Pi

se D(ti)= Pi  allora è possible determinare se il set dei task è schedulabile su m processori

utilizzo accumulato u si ottiene da:

se u<=m allora il sistema ha la schedulability

**RMS (Rate Monotomic Scheduling)**

È uno scheduling di tipo statico

Assegna priorità fisse a tutti i task in ordine per massimizzare la schedulability

L’idea di base è: assegnare la priorità ad ogni task in base al periodo; quindi, il task con periodo minore ha priorità più alta.

Si assume che i task siano indipendenti e periodici

Di=Pi

Il tempo di esecuzione è noto e costante

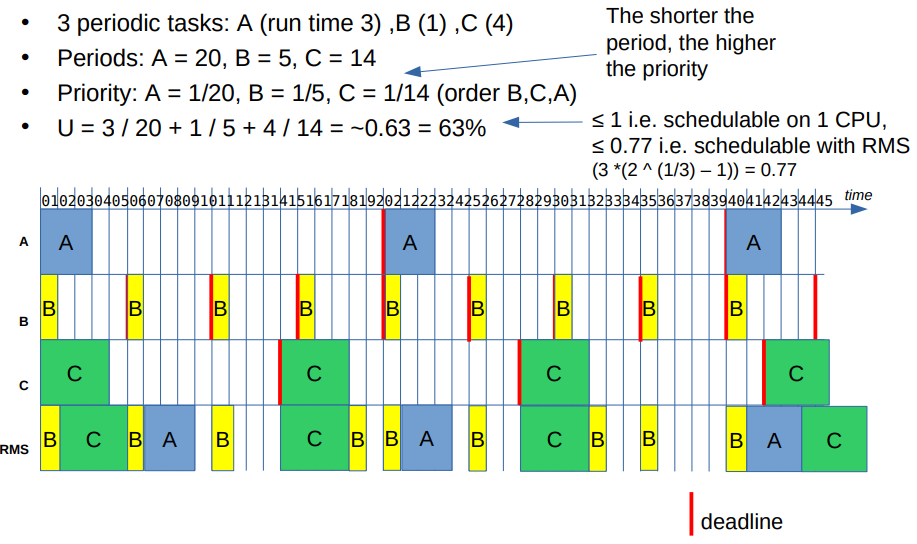
La latenza del context switch è trascurabile

Condizione di schedulabilità:

n= numero task

RMA è l'algoritmo ottimale con priorità statica: Se un insieme di attività non può essere pianificato utilizzando l'algoritmo RMA, non può essere pianificato utilizzando nessun altro algoritmo con priorità statica

Priorità si calcola 1/periodo



**EDF**

È un algoritmo di scheduling a priorità dinamica.

I task con deadline più breve hanno priorità maggiore.

La priorità cambia Runtime.

Il limite di utilizzo è il 100%, le deadlines sono garantite se la cpu lavora al 100%.

Funziona anche con task non periodici

Se non si rispetta

Usare EDF