**Livello di kernel (es SO)**

* Il kernel sa quali thread ha
* thread scheduling è fatto da il kernel
* Se un thread blocca, altri thread all'interno dello stesso processo possono continuare a essere eseguiti
* I thread a livello kernel sono in esecuzione in user mode (non privilegiata)

**Livello Utente (es Programmazione)**

* Il kernel non conosce niente dei threads
* La schedulazione dei thread viene eseguita dal processo
  + Quanto viene pianificato il processo, i thread possono essere eseguiti
* Se un thread blocca, l’intero processo è bloccato

**Race condition**

Quando si lavora con risorse condivise, esiste il problema di accesso simultaneo alle risorse

**Come evitarli?**

* Più processi/thread non devono eseguirsi simultaneamente dentro regioni critiche (mutual exclusion)
* Il programma deve funzionare a prescindere dalla velocità, nr.processi e timing
* Quanto fuori da una regione critica un therad/processo non può bloccare un altro thread
* Nessun thread deve aspettare indefinitivamente prima di entrare in una zona critica (starvation)

**Atomicità (per assicurare la mutual exclusion)**

* Un operazione è atomica se è completata in un singolo passo (cioè non può essere interrotto)
* Nessun altro thread può osservare un incoerenza di stato (modifiche parziali)

**Soluzioni valide**

* L’esecuzione sequenziale permette di non avere due thread allo stesso tempo in una sezione critica
* Disabilitando gli interrupt previene il preemption e quindi forza esecuzione sequenziale (non funziona su sistemi multi-core!)

**Busy Waiting**

Le soluzioni strict aternation, Perterson e spinlock basate su TSL o XCHG impiegano tempo per entrare nelle sezioni critiche, questo porta ad un continuo loop di attesa. Se il ciclo è breve o ci sono più CPU o gli inconvenienti dei core sono trascurabili, altrimenti può causare un inversione di priorità causando deadlock (siccome potrebbe non uscire mai dalla regione critica)

**Sincronizzazione Primitive**

Per evitare race conditions, esistono dei synchronization primitives che permettono di non avere busy waiting. (Mutex, Semaphores, Barriers, Condition Variables)

**Mutex**

Per entrare in una sezione critica un thread tenta di acquisire il mutex corrispondente (blocca il mutex)

* Solo un thread alla volta può acquisirlo ( se è in stato unlocked)
* Da quel momento solo lui può sbloccarlo
* Se il mutex è già bloccato, gli altri thread dovranno aspettare

**Semafori**

Un semaforo rappresenta un valore intero, che può essere incrementato o decrementato (il programmatore decide il valore iniziale)

Quando un thread decrementa il semaforo, se il risultato è negativo, i thread si bloccano e devono aspettare fino a quando un altro thread aumenta il semaforo

**Condition Variables (alternativa ai semafori)**

* Wait (per un evento che si verifica)
* Signal (che qualche evento è successo, risvegli almeno uno)
* Brodcast (uguale a signal ma risveglia tutti)

Purtroppo non gestisce la mutual exclusion, frequentemente l'evento di una variabile di condizione è correlato ad una risorsa condivisa (che dovrebbe essere protetta con un mutex).

se non ci sono thread in attesa signal non fa nulla, se più di un thread è in attesa sulla variabile condition e c'è un mutex, quei thread cercheranno di riacquistarlo ma solo uno avrà successo (gli altri aspetteranno)

**Barrier**

La barriera è un meccanismo di sincronizzazione che è utile per controllare l'esecuzione di un algoritmo parallelo a più fasi, dove ogni passo dipende dai risultati calcolati al passo precedente

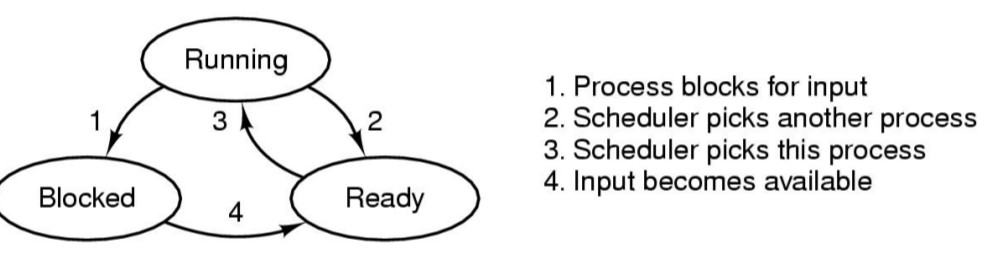
**Processi**

Un processo rappresenta un'istanza di un programma che è in esecuzione. I processi richiedono risorse (tempo di CPU, memoria, accesso al filesystem e i dispositivi I/O).Inoltre comprende il contesto di esecuzione del calcolo.

* Contesto hardware (contatore del programma, puntatore dello stack, parola di stato del processore, registri, traduzione degli indirizzi tavolo)
* Spazio indirizzo (regioni della memoria)
* Informazioni di controllo
* Credenziali

Il processo è un entità attiva, mentre un programma è un entità passiva (codice macchina in memoria). In un sistema multi programmato più di un processo può risiedere nella memoria

Gli stati di un processo si rappresentano cosi. Il kernel assegna un id chiamato process identifier (PID) ad ogni processo.



Creare un nuovo processo viene chiamato Fork:

* Crea una struttura dati di processo per il processo figlio
* Crea un nuovo PID per il figlio
* Copia lo spazio di indirizzamento del genitore in quello del figlio
* I valori di ritorno sono:
  + Al genitore: il PID del figlio
  + Al figlio: 0
  + Se fork fallisce: -1

Un processo viene terminato quando viene invocato dal main return e quando viene chiamato exit(int)

**Processo Zombie**

* Quando un processo figlio termina, il genitore deve attendere con wait oppure waitpid
* Finché il processo genitore non aspetta un figlio, quest'ultimo rimane in uno stato di zombi (o defunto) e non può essere rimosso dalla lista dei processi
* Se il genitore termina prima del figlio, il figlio eredita il processo init (PID 1) questo comportamento viene chiamato re-parenting viene usato per creare daemons (double fork)
  + Il segnale SIGCHLD viene inviato al genitore quando il bambino esce
  + Il processo figlio è chiamato processo orfano
  + Se il figlio era in uno stato zombie, il processo di init si prenderà cura di esso

**Inter-Process Communication**

I segnali POSIX sono semplici messaggi (Asincroni) che possono essere scambiati tra processi o inviati dal sistema operativo a un processo. I Signal Handlers sono Async. Minore è il valore del segnale maggiore è la priorità.

Ogni processo ha un signal mask, il quale può aggiungere segnali alla maschera e impedire l'esecuzione del gestore (segnali bloccati) o rimuovere i segnali dalla maschera per sbloccarli.

SIGKILL e SIGSTOP non possono essere intercettati (ignorati o gestiti) ne bloccati.

**Stati dei segnali**

* Pending 🡪 tra generazione e consegna
* Blocked 🡪 La consegna è impedita dalla maschera, il segnale bloccato è ancora in attesa
* Delivered 🡪 Il processo esegue la procedura di gestione del segnale

**Tipi di segnali**

* Normali
  + Valori numerici semplici
  + Non possono essere salvati
  + L’ordine di ricezione non garantita (tutti la stessa priorità, più alta dei segnali realtime)
  + Non ha nessun payload

Quando arriva un segnale, imposta un flag sulla maschera: viene registrato un solo evento per ciascun tipo (in sospeso o bloccato) se è lo stesso tipo prima della gestione viene considerata solo la prima occorrenza.

* Realtime
  + SIGRTMIN a SIGRTMax
  + Priorità inferiore ai segnali normali
  + Può essere messa in coda (ordinata)
  + Può avere un piccolo payload

Se il processo riceve più segnali dello stesso tipo, vengono accodati e gestiti singolarmente. (SIGRTMIN è la massima priorità in tempo reale). Se il processo sta gestendo un segnale, solo un altro segnale di priorità più alta può interromperlo.

**Scheduling**

In un sistema multiprogrammato, lo scheduler determina l'ordine in cui processi / thread possono ottenere risorse di sistema (tipicamente la CPU) in base a un criterio / algoritmo di sheduling

**Cooperative Multitasking**

Ogni processo coopera con gli altri e restituisce volontariamente la CPU (con uno YIELD)

**Pre-Emptive Multitasking**

Il SO deve impiegare una tecnica per riprendere la CPU (dopo qualche tempo) dal processo di esecuzione

L'hardware implementa un meccanismo di segnalazione che consente ai dispositivi di interrompere la CPU(e quindi il flusso del programma) per portarlo ai livelli predefiniti in modo asincrono \*. Esistono anche sincroni.

* **Throughput** (batch systems) Numero di lavori che il sistema può completare per unità di tempo
* **Turnaround Time** (batch systems) Tempo medio necessario per completare un lavoro
* **Response Time** (interactive systems) Tempo tra l'emissione di un comando e il risultato
* **Predictability and regularity** (realtime systems) Il tempo di risposta deve essere garantito

**Non-preemptive scheduling**

* Un processo viene eseguito fino a quando non blocca (attesa I /O) o rinuncia volontariamente alla CPU e chiama lo scheduler (yielding)
* Un processo errato (o un bug) potrebbe bloccare l'intero sistema

**Preemptive scheduling**

* Un processo viene eseguito per un tempo massimo fisso (quantum) prima di essere interrotto da un interrupt di clock che restituisce il controllo allo scheduler (preemption)

**I/O Bound processes**

Attese I / O frequenti (I/O burst) 🡪frequenti Yield

**CPU Bound**

Lunga computazione (CPU burst) 🡪 richiede preemption

**Batch Scheduling**

* First-come First-served (FCFS)
  + Processi seguono l’ordine di arrivo
    - Ogni processo lavora finche vuole
    - Se bloccato passa al successive (viene messo in coda quando pronto)
* Shortest Job First (SJF)
  + priorità al processo con il tempo di esecuzione stimato più breve
    - Massimizza la velocità di trasmissione (molte attività brevi, completate rapidamente)
    - Riduce al minimo i tempi di consegna per processi brevi
    - Richiede un tempo di esecuzione stimato
    - Può portare alla starvation di processi lunghi (posticipati indefinitamente)
* Shortest Remaining Time Next (SRTN)
  + Versione Preemptive di SJF, considera il tempo di esecuzione rimanente del processo

**Interactive scheduling**

* Round Robin
  + Processi definiti in un circular buffer
    - Viene assegnata una fascia oraria (quantistica) ad ogni processo
    - Quando il Quantum scade, la CPU viene data ad un altro processo
    - Se viene fatto uno Yield viene assegnata la CPU ad un altro processo
    - la lunghezza di un quantum deve essere attentamente determinata per evitare di spendere troppo tempo per "context-switch"
    - non tiene conto delle differenze (alcuni processi possono essere più importanti di altri)
* Priority Scheduling
  + sceglie il processo di priorità più alta
  + Per evitare la starvation di processi a bassa priorità, bisogna avere priorità dinamiche
* Multilevel Feedback Queue (MLFQ)
  + Risolve i problemi di short term scheduling

**Lottery Scheduling**

* Ogni processo ha un ticket con valore proporzionale alla sua priorità
* Periodicamente il processo con il biglietto vincente verrà eseguito
* Vantaggi
  + Facile da implementare
  + Ticket trasferibili per velocizzare alcuni lavori

**Fair-Share Scheduling**

* Implementazione di politiche che dividono il tempo della CPU tra gli utenti o assegnano priorità agli utenti, indipendentemente dal numero dei loro processi.

**Locality**

* Temporal locality
  + quando si accede a un dato, è probabile che vi si possa accedere nuovamente nel prossimo futuro
* Spatial locality
  + quando si accede a un pezzo di dati all'indirizzo X, è probabile che si acceda anche a dati vicini

**Località in sistemi uniprocessore**

* la cache corrispondente al thread conterrà la maggior parte dei dati
* La cache funziona sul principio della località per:
* Ridurre la possibilità di arrestare la CPU
* ridurre la quantità di dati che devono essere trasferiti dalla memoria principale

**Locality in multi-processor systems**

* **UMA** (Uniform Memory Access) (l'accesso alla memoria principale avviene alla stessa velocità per tutti i processori)
* **NUMA** (Nonuniform Memory Access) (l'accesso ad alcune aree del mainmemory è più veloce per alcuni processori rispetto ad altre parti della memoria)
* Con **UMA** l'accesso ai dati che non si trova nella cache locale è più lento; lo stesso con **NUMA** (una CPU deve chiedere ad un'altra CPU)

**Locality-aware scheduling**

* **Soft Affinity**
  + Lo scheduler tenta di assegnare sempre un thread alla stessa CPU/core o a un set specifico di CPU / core
* **Hard Affinity**
  + Lo schedulatore è obbligato ad assegnare un thread a una CPU/core specifica o a un set specifico di CPU/core

**Single queue scheduling**

Tutte le CPU possono essere compl. sfruttate (bilanc. del carico)

* necessario sincronizzazione per accesso sicuro alla coda condiv., **non scalabile**
* Introdurre problemi locali:
  + processi / thread si spostano tra CPU / core
  + L'affinità della cache è compromessa

**Multiple queues scheduling**

* Scalabile, può assicurare cache affinity
* Può portare a caricare uno squilibrio, risolto con
  + migrazione di processi a CPU meno caricate
  + furto del lavoro (la CPU può eseguire attività da altre code quando è inattiva)

**Real-Time Operating System (RTOS)**

È un SO multitasking, per app RealTime (la correttezza dipende dal risultato e dal tempo)

Un sistema in tempo reale deve essere **predictable**:

* Deve soddisfare i limiti di tempo di programmazione (in tal caso si dice che il sistema è **schedulabile**)
* Se il pool di processi è dinamico, bisogna verificare che i vincoli siano sempre soddisfatti

**Design**

* Eventi driven:
  + Priority Scheduling e Task eseguiti in risposta da eventi
* Time Sharing:
  + Round Robin Scheduling e più deterministico

**Task**

* Event triggered
  + Attività che vengono eseguite in risposta a un evento (esterno) (compiti aperiodici)
* Time Triggered
  + Attività eseguite in un determinato momento in intervalli regolari (attività periodiche)

**Latenza**

* Tempo tra il verificarsi di un evento (stimolo) e la risposta del software

**Preemption**

In un SO MT potrebbe essere in esecuzione qualche altra attività quando si verifica un evento: è necessaria la Preemption (anticipare/interrompere per attività con priorità più alta)

**Requisiti**

* **Hard Real-Time**
  + Compiti devono essere completati entro una deadline (non tollerati ritardi e pericolosi)
  + Scheduling dinamico/statico
* **Soft Real-Time**
  + Uguale ad hard RT ma ritardi tollerati e non pericolosi
  + Scheduling prioritario
* **Firm Real-Time**
  + Non rispettare deadline non pericolosa (cestinato risultato)

**Schedulability**

La schedulability è raggiunta quando vengono rispettate le scadenze per ogni attività (important role played by the scheduling policy/algorithm).

**Rate Monotonic Scheduling (RMS)**

Se una serie di attività non può essere pianificata utilizzando l'algoritmo RMS, non può essere pianificata utilizzando qualsiasi altro algoritmo a priorità statica.

* Priorità statica
* RMS assegna priorità fisse a compiti periodici al fine di massimizzare la loro "schedulabilità"
* assegna la priorità di ogni attività in base al suo periodo, più breve è il periodo più alta la priorità
* task indipendenti e periodici e deadline stessa lunghezza del periodo schedulato
* tempo esecuzione costante e context switch trascurabile

**Scheduling Earliest Deadline First (EDF)**

* Priorità dinamica, Il limite di utilizzo è del 100%:
* L'attività con la prima scadenza ha la priorità più alta → le priorità cambiano durante l'esecuzione
* Quando il sistema è sovraccarico, la pianificazione diventa imprevedibile
* Funziona anche per attività non periodiche