

Un processo è l'esecuzione di un programma in esecuzione su una macchina, se eseguire lo stesso programma in momenti diversi sareanno processi differenti.

Lo spazio di indirizzamento del processo si compone di:

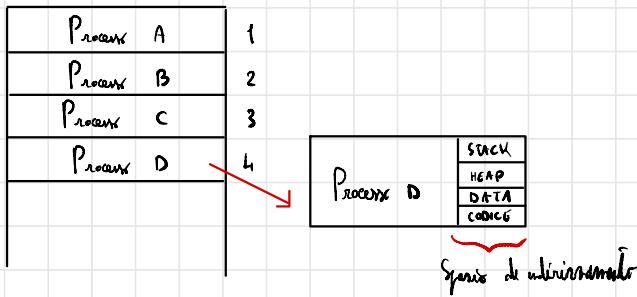
- codice
- dati statici (data)
- dati dinamici (Heap)
- pila dei processi (Stack)

I registri della CPU fanno parte dello stato del processo.

Durante il context-switch si salvano i registri e verranno usati per riportarli allo stato del processo.

I processi sono entità indipendenti legati da un legame di "parentela" a partire dal processo init (pid 1) istanziato dal so, tutti gli altri vengono generati a partire da altri processi presenti.

Se si gestisce i processi attraverso la struttura della **process Table**, i cui record sono detti PCB (process control block). Un PCB è un processo attivo. Ogni processo è identificato univocamente dal suo PID (process id).



Il PCB di un processo contiene tutte le informazioni di un processo. La CPU virtualizzata permette la gestione dei processi, essi condividono l'area di memoria ed in un determinato istante vi è un solo processo in esecuzione, poiché la CPU-virtualizzata permette solo un c'processo-parallelismo.

Ci sono diversi approcci per la creazione di processi:

- alla Unix: una system call **fork()** crea un clone del processo in esecuzione (figlio diverso) con un proprio spazio di indirizzamento avendo anche lo stesso codice e gli stessi dati.  
La fork restituisce valori di riferimento differenti al processo padre e al figlio per rendere distinguibile il contesto.

La fork si combina poi con la chiamata **exec()** che permette di sovrascrivere lo spazio di indirizzi del processo e quindi l'effetto è quello di ottenere un processo nuovo che potrà avere riempito col codice del nuovo processo da eseguire.

- alla Windows: una system call **createProcess()** crea un nuovo processo a partire da uno preesistente senza

fare da passaggi intermedi come per esempio  
la clonazione del process

Un processo può terminare per diversi motivi

- **uscita normale (volontaria)**: interrarsi nel codice un'istruzione  
di terminazione del processo e rilasciare le risorse che erano  
dedicate precedentemente al processo.

unix => exit()

WIN32 => ExitProcess()

- **uscita su errore (volontaria)**: il processo si trova davanti  
ad una situazione anomala e la gestisce terminando il  
processo (exit(1), exit(...)) . Ritorna un valore diverso  
da 0

- **errore critico (imvolontario)**: c'è stato un problema  
che non è risolvibile o trattabile. Si ha

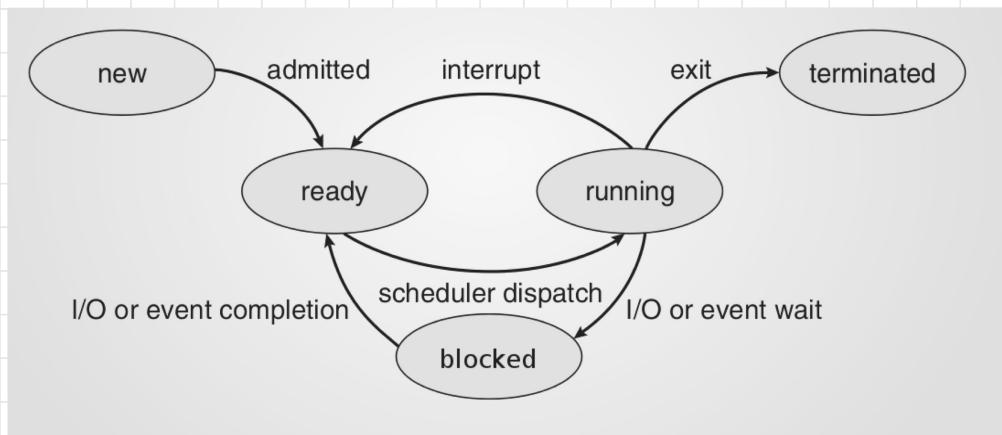
un'anomalia critica, per esempio l'esecuzione di un'istruzione non valida da parte della CPU e quindi non si può continuare l'esecuzione del processo (division per zero, tentativo di accesso a locazioni di memoria non formesse).

- **Terminare da un altro processo**: una richiesta di terminazione da parte di un altro processo termina l'esecuzione del processo (chiusura di una finestra)

unix => Kill

win 32 => TerminateProcess

## Stati di un processo



**Stato di pronto (ready)**  $\Rightarrow$  indica un processo pronto per essere eseguito (in attesa della disponibilità della CPU). Una struttura dinamica gestisce questo insieme ed è la **coda dei processi pronti**. L'elemento generico è un PCB (process control block). Un **scheduler** è una routine del SO (un software quindi) che preleva un processo dalla coda dei processi pronti e lo fa passare in stato di running.

**NB** Su un processo, in un determinato istante di tempo, solo un processo può essere in stato di running

**Stato di blocco (blocked)  $\Rightarrow$**  un processo viene bloccato da una chiamata lenta (solitamente di I/O) ed entra in uno stato logico di bloccato. Non potendo essere eseguito si mette in stato di "bloccato" in quanto non puo' essere inserito nella coda di ready.

Quando l'evento di I/O termina, il processo non torna subito in running ma viene inserito nella coda dei processi pronti.

**Stato di esecuzione (running)  $\Rightarrow$**  un processo è in esecuzione. A seguito di un interrupt regolare, un processo si mette sottratta la CPU e ritorna nello stato dei ready. Delle cause potrebbero essere **prelazione** oppure **pirata** più alte da parte di altri processi. Un esempio di prelazione è il termine del ciclo di clock che gli era stato assegnato per **multiplexing temporale** dal SO.

Un' interrupt deve essere seguita in modo che il context switch avvenga senza compromettere il processo:

### Gestione degli **interrupt** per il passaggio di processo:

- salvataggio nello stack del PC e del PSW nello stack attuale;
- caricamento dal vettore degli interrupt l'indirizzo della procedura associata;
- salvataggio registri e impostazione di un nuovo stack;
- esecuzione procedura di servizio per l'interrupt;
- interrogazione dello scheduler per sapere con quale processo proseguire;
- ripristino dal PCB dello stato di tale processo (registri, mappa memoria);
- ripresa nel processo corrente.

La relazione emula che un processo monopolizzi la CPU.  
Pur non essendo necessaria, se non è implementata (embedded oppure real-time) si necessita di collaborazione da parte dei processi. La relazione introduce overhead.  
Quando un processo avrà figli termina, essi negano collettivamente dal processo padre (+) in quanto sono rimasti infine  
Un processo "padre" ottiene in Totem che gli consenta alcune

zioni sul process figlio.

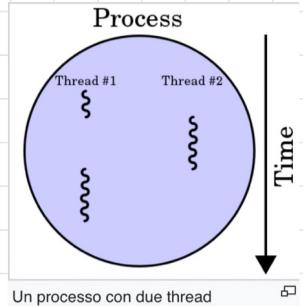
Un "rendezvous" colloca un process nella coda dei process bloccati, una struttura dinamica in cui si collocano i processi bloccati. Quando arriva il rendez, il primo process che si era messo in coda varia il punto a partire.

## Programmazione multi-Thread

Un process è un flusso di istruzioni in esecuzione su una macchina. Si può pensare di suddividere uno stesso process in più flussi di esecuzione, detti Thread. I Thread sono all'interno del contenitore process, significa che potranno accedere facilmente alle risorse di elaborazione del process. L'utilizzo dei Thread rende molto più rapida l'esecuzione delle istruzioni a fronte di una programmazione più complessa. L'utilizzo dei Thread introduce multithread. Grazie all'utilizzo dei Thread si ha un pseudoparallelismo che ottimizza il tempo di esecuzione dell'intero process.

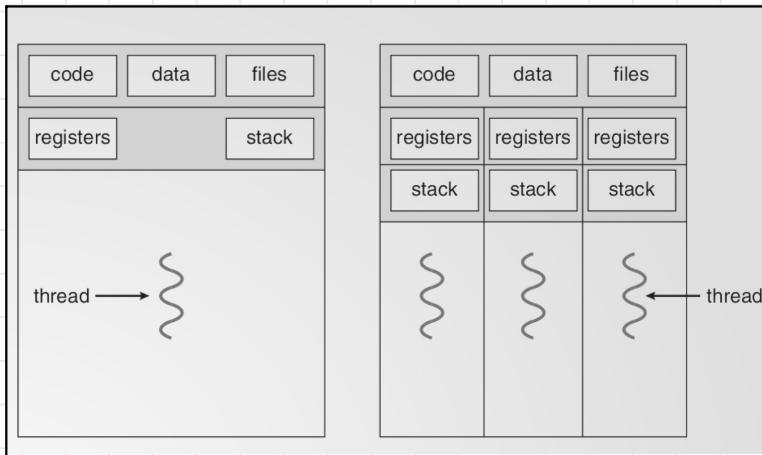
Se un thread si dovesse bloccare a causa di una risorsa che deve essere letta dalla memoria, viene eseguito un altro thread che invece può andare avanti con l'esecuzione (scambio). Risulta molto veloce scambiarsi con un altro thread della stessa applicazione grazie al fatto che il processo ricorda l'uso dei thread e ne tiene conto.

Divide l'esecuzione di un process in flussi che eseguono in **pseudo-parallelo** parte diverse del process

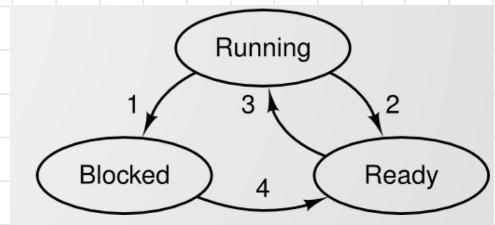


Un thread "dispatcher" si mette in ascolto su una porta e svolge la gestione di una richiesta dal resto degli altri thread detti "worker". Il vantaggio è ottenere dal fatto che si riesce a scalare l'entità di un process da eseguire su diverse unità di elaborazione indipendenti che fanno avanti l'esecuzione in **pseudo-parallelo**. In un istante di tempo, la CPU, è disegnata ad un solo thread. Un applicazione che non sfrutta il multi-thread viene vista come un unico thread che

gestisce l'intero processo vedendolo come unico thread di esecuzione.



Un Thread ha una propria CPU virtuale assegnata e quindi è caratterizzato da un proprio stack, PC, stato (wait ready o blocked) e i propri registri, mentre condivide tutto il resto, come memoria, file aperti, il codice ecc...  
Gli scheduler salvano direttamente un Thread e non fanno nulla della reale del processo e vedono come un unico Thread i processi che non ne fanno uso.



Quando si fa una chiamata all'esecuzione di un thread a quella di un altro si ha comunque un context-switch, quindi si salvano i registri che verranno riportati ma esiste un cambio di contesto limitato allo stesso processo non si necessita di riprogrammare l'MMU e non si cambia localizzazione di memoria, dunque tutto molto più velocemente rispetto ad un context-switch tra diversi processi.

Ecco perché i thread sono della processi leggieri e veloci.

Un processo manda con unico thread diversi ed è possibile eseguire le seguenti operazioni sul thread:

- Thread-create  $\Rightarrow$  ne crea uno
- Thread-exit  $\Rightarrow$  il thread chiama termin
- Thread-join  $\Rightarrow$  uscita per bloccare l'esecuzione del thread chiamante fino a quando il thread figlio non ha completato la sua esecuzione
- Thread-yield  $\Rightarrow$  il thread chiamante rilascia volontariamente la CPU