



Università degli Studi di Parma

Dipartimento di Ingegneria e Architettura

Sistemi operativi e in tempo reale - a.a. 2023/24

Gli elementi della *concorrenza*: Processi, thread e spazi di indirizzamento

Alcune idee centrali sui Sistemi Operativi



- ❑ Alcune idee centrali sui sistemi operativi:
 - Meccanismi vs. politiche
 - Monoprogrammazione vs. multiprogrammazione
 - Multiprogrammazione vs. time sharing
 - Time sharing vs. elaborazione batch

- ❑ Tutte queste idee hanno a che fare con il concetto di processo e l'idea cardine della *multiprogrammazione*

Processi



- ❑ Processo: programma in esecuzione
- ❑ Un sistema multiprogrammato è costituito da un insieme di processi: processi utente (eseguono codice delle applicazioni utente), processi di sistema (eseguono codice del SO)
- ❑ Un processo è controllato da un programma e necessita di un processore per la sua esecuzione
- ❑ Può disporre di un processore dedicato o dividerne uno con altri
- ❑ E' caratterizzato da uno *stato*: in esecuzione, pronto, in attesa (principali)



Dai processi ai thread

- ❑ C'è stata una evoluzione storica nell'uso del termine *processo*, oggi affiancato da quello di *thread*
- ❑ Quali relazioni tra thread e processi?
- ❑ Tutte le caratteristiche precedenti si riferiscono sia ai processi che ai thread!



Concorrenza

- ❑ Elementi di base attivi: thread e processi
- ❑ Elementi di base passivi: spazi di indirizzamento
- ❑ Thread: *trama* di esecuzione
 - Ciclo indipendente di Fetch/Decode/Execute
 - Opera in uno spazio di indirizzamento
- ❑ Processi: qualcosa in più di un thread!
 - Uno o più thread e uno spazio di indirizzamento in cui essi operano

Monoprogrammazione e multiprogrammazione



- ❑ Monoprogrammazione: *un solo thread in esecuzione in ciascun istante*
 - Es.: MS/DOS, i primi Macintosh, elaborazione Batch
 - Semplifica il compito del SO
 - Elimina la *concorrenza*
 - Plausibile, un tempo, per semplici elaboratori personali, ora incompatibile con i nostri stili di utilizzo

- ❑ Multiprogrammazione: *più di un thread in esecuzione*
 - Es.: Multics, UNIX/Linux, OS/2, Windows NT/2000/XP, Vista, Windows 7-11, Mac OS X, iOS, Android ...
 - Talvolta chiamata anche “multitasking”
 - Determina una situazione di *concorrenza*

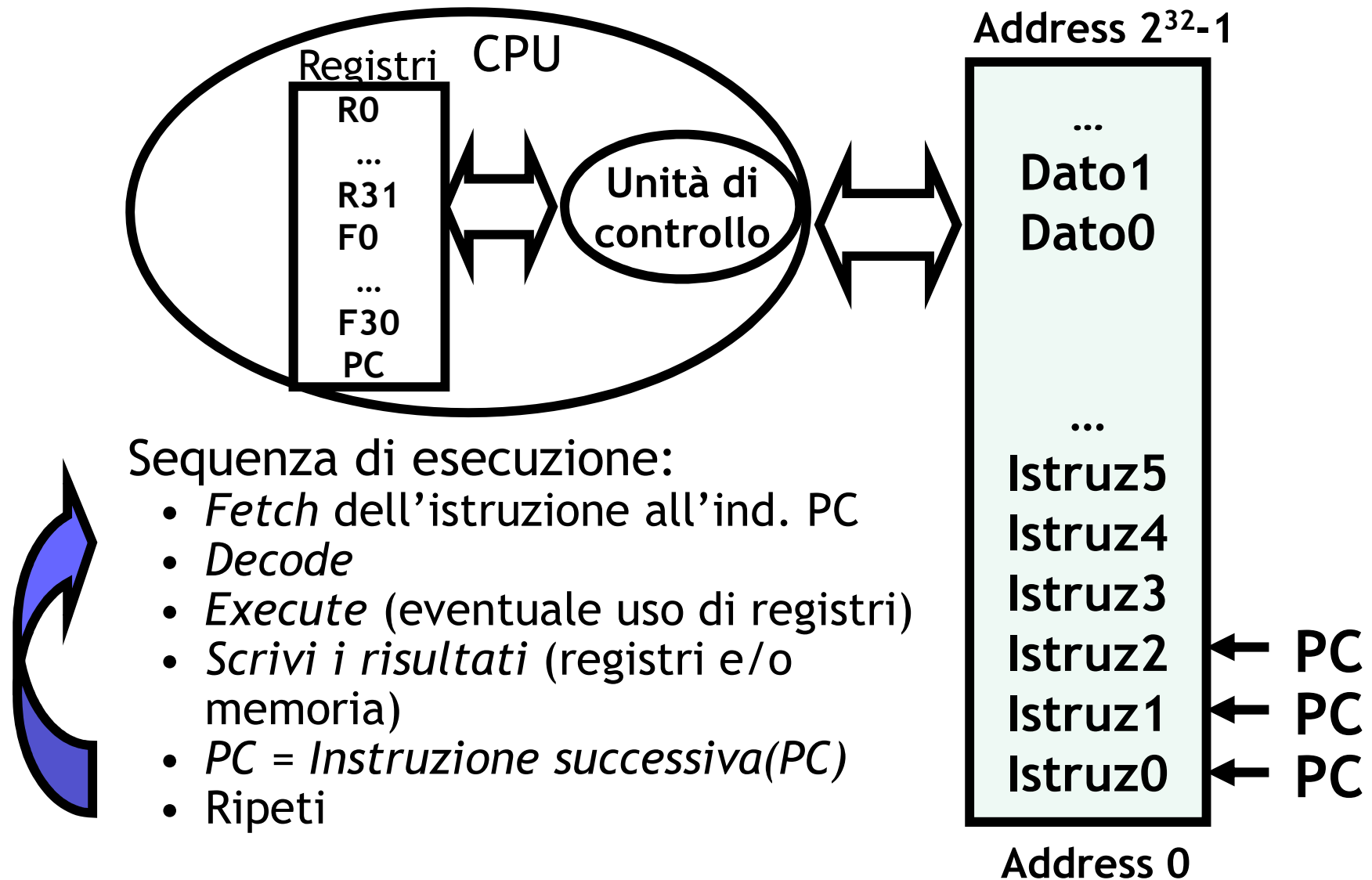


Multiprogrammazione e risorse

- ❑ Il problema di base della presenza di più thread in esecuzione riguarda le *risorse*:
 - L'hardware fornisce risorse limitate (CPU, memoria e dispositivi di I/O)
 - Le API di (multi)programmazione forniscono invece una visione di accesso esclusivo alla macchina
- ❑ Il SO deve coordinare tutte le attività:
 - Più utenti, interrupt di I/O, ...
 - Come gestirle in modo corretto?
- ❑ Idea base: uso dell'astrazione *Macchina Virtuale*
 - Ogni thread esegue in una macchina virtuale dedicata
 - --> Scomposizione in problemi più semplici
 - Astrae la nozione di programma in esecuzione
 - Richiede il multiplexing delle macchine astratte

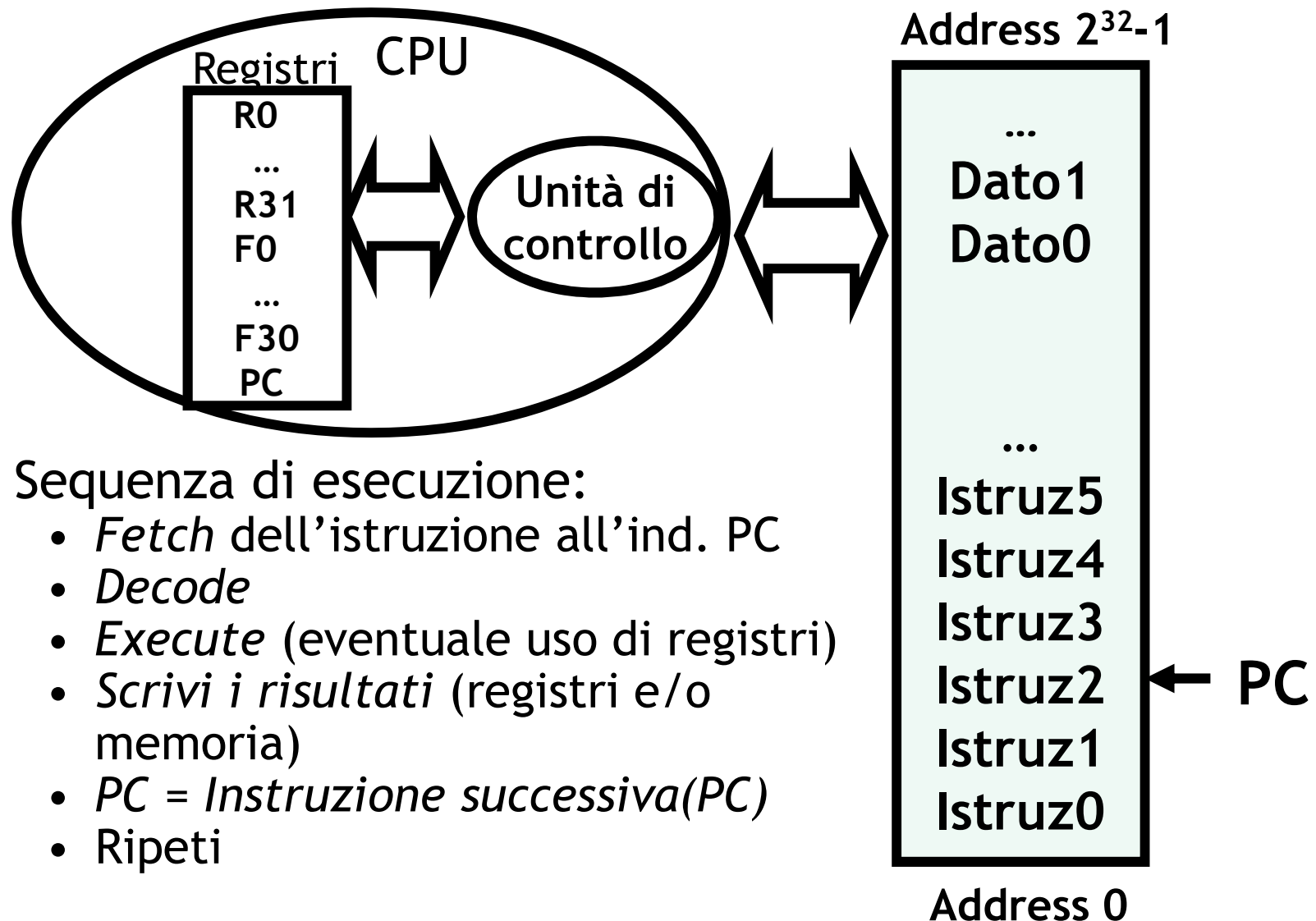


Esecuzione di un thread



Esecuzione di un thread

Come consentire
l'esecuzione di un altro
thread?





Esecuzione di più thread

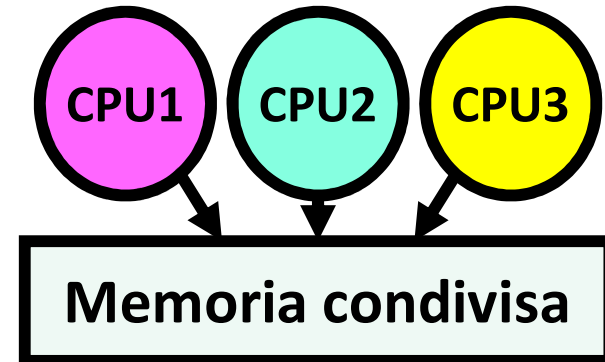
- ❑ Thread: uno specifico e univoco contesto di esecuzione
 - Program Counter, registri, flag di esecuzione, stack, memoria
- ❑ Un thread è in esecuzione su un processore o core quando *risiede nei suoi registri*
- ❑ Il thread utilizza *tutti i registri* del processore e ha a disposizione *l'intero spazio* di indirizzamento
 - PC punta ad una istruzione del thread in memoria,
 - i registri ospitano valori del thread o puntano a locazioni di memoria con variabili del thread,
 - SP punta allo stack del thread in memoria, etc.
- ❑ La multiprogrammazione, per definizione, è la *presenza in memoria principale di più thread!*
- ❑ Come risolvere questo conflitto?

Virtualizzazione della CPU



Obiettivo: virtualizzazione CPU

- ❑ Ogni thread esegue il ciclo di fetch-execute in una propria *CPU virtuale*
- ❑ Ogni CPU virtuale accede allo *spazio di memoria condiviso*

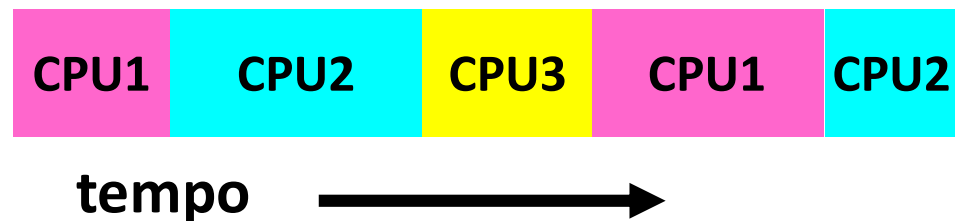


Problema:

- ❑ Come fornire a ciascun thread una CPU (virtuale) e altre risorse private ?

Soluzione:

- ❑ Realizzando l'illusione di processori multipli mediante multiplexing nel tempo



Virtualizzazione della CPU



- ❑ Ogni CPU virtuale richiede una struttura (*blocco di stato* all'interno del *descrittore* o PCB/TCB) per:
 - Program Counter (PC), Stack Pointer (SP), Registri
- ❑ Per commutare tra le CPU, il SO esegue:
 - Salvataggio di PC, SP, e registri nel descrittore corrente
 - Caricamento di PC, SP, e registri dal nuovo descrittore
- ❑ La commutazione è determinata da:
 - Timer, precedenza volontarie (yield), I/O, + altro
- ❑ La sola commutazione della CPU realizza una *semplice forma di multiprogrammazione* caratterizzata da:
 - *condivisione* della CPU *mediante commutazione* e TCB, *condivisione diretta* di tutte le altre risorse

Multiprogrammazione con risorse condivise



- ❑ Proprietà di questa tecnica di multiprogrammazione:
- ❑ Tutte le CPU virtuali *condividono* le altre risorse hw:
 - Dispositivi di I/O, Memoria
- ❑ Conseguenze della condivisione:
 - (1) Ogni thread può accedere ai dati degli altri thread
 - (2) I thread possono condividere le istruzioni
 - (1) e (2) utili per la condivisione efficiente, negative per la protezione
 - I thread potrebbero sovrascrivere funzioni del SO!
 - ➔ *multiprogrammazione senza protezione*
- ❑ Questo *modello non protetto* è utilizzato in:
 - Applicazioni embedded
 - Windows 3.1/Macintosh (commutazione solo con yield)
 - Windows 95/98/ME? (commutazione con yield e timer)



Multiprogrammazione con protezione dei thread



- ❑ In genere, nei sistemi di elaborazione è indispensabile prevedere la protezione mutua tra i thread e del SO → *Multiprogrammazione con protezione*
- ❑ Richiede di inserire nel SO alcune caratteristiche:
 1. Protezione della memoria:
 - i thread non devono avere accesso a tutta la memoria
 - partizionamento della memoria, controllo dell'accesso
 2. Protezione dei dispositivi di I/O:
 - i thread non devono avere accesso a tutti i dispositivi
 3. Protezione del processore → Garanzia della commutazione tra i thread mediante *preemption*:
 - uso di timer
 - gestione del timer (ad es. per disabilitarlo) non consentita al codice dei thread utente



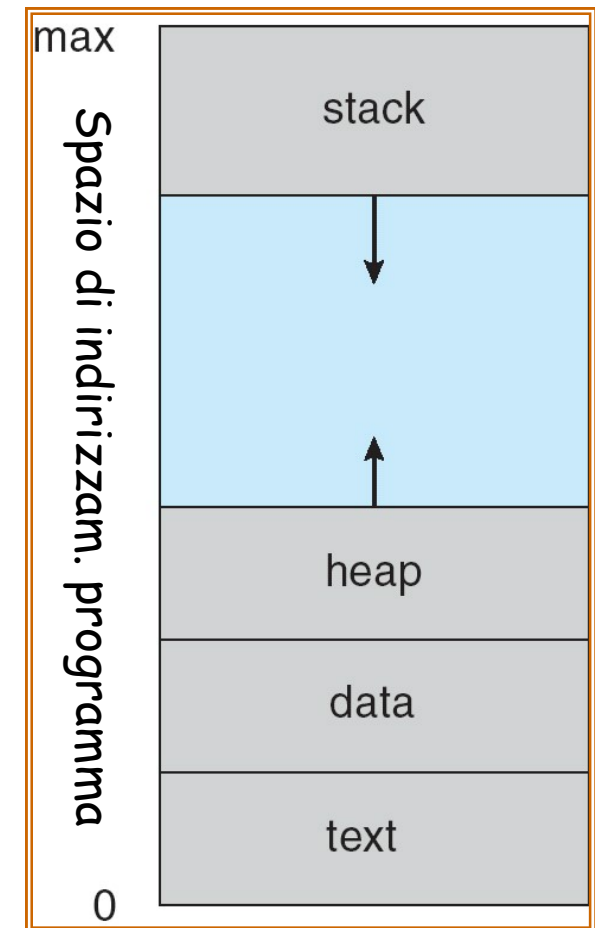
Processi

- ❑ Un *processo* definisce un ambiente di esecuzione vincolato e protetto
 - E' costituito da uno *spazio di indirizzamento* in cui operano *uno o più thread*
 - Possiede descrittori di file e accessi a risorse di I/O
 - Incapsula *uno o più thread* che *condividono le risorse del processo*
- ❑ Beneficio: si ottiene protezione reciproca dei processi e protezione del S.O.
- ❑ Compromesso tra protezione ed efficienza:
 - i processi garantiscono la protezione della memoria
 - i thread efficienza nell'elaborazione
- ❑ Un'*applicazione* consiste di uno o più processi

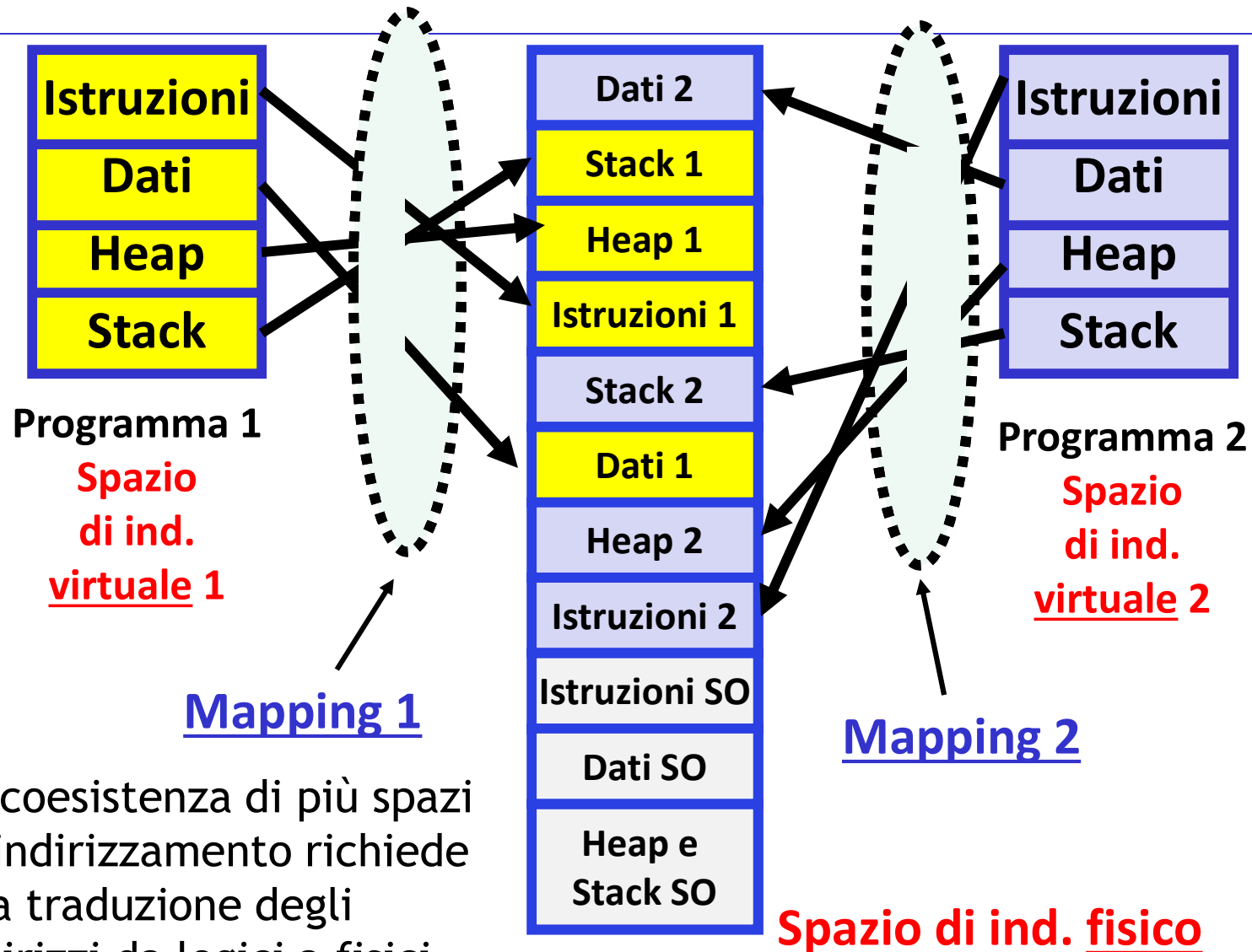


Lo spazio di indirizzamento di un programma

- *Address space* \Rightarrow insieme degli indirizzi accessibili + lo stato associato ad essi:
 - Per un processore a 32 bit: $2^{32} \approx 4$ miliardi di indirizzi
 - Spazio di indirizzamento logico
- Effettuando una lettura o scrittura ad un indirizzo si ottiene uno dei seguenti risultati:
 - Non accade nulla
 - Si comporta come memoria normale
 - Ignora i comandi di scrittura
 - Si determina un'operazione di I/O (\rightarrow memory-mapped I/O)
 - Si genera un'eccezione (errore)



Realizzazione dello spazio di indirizzamento



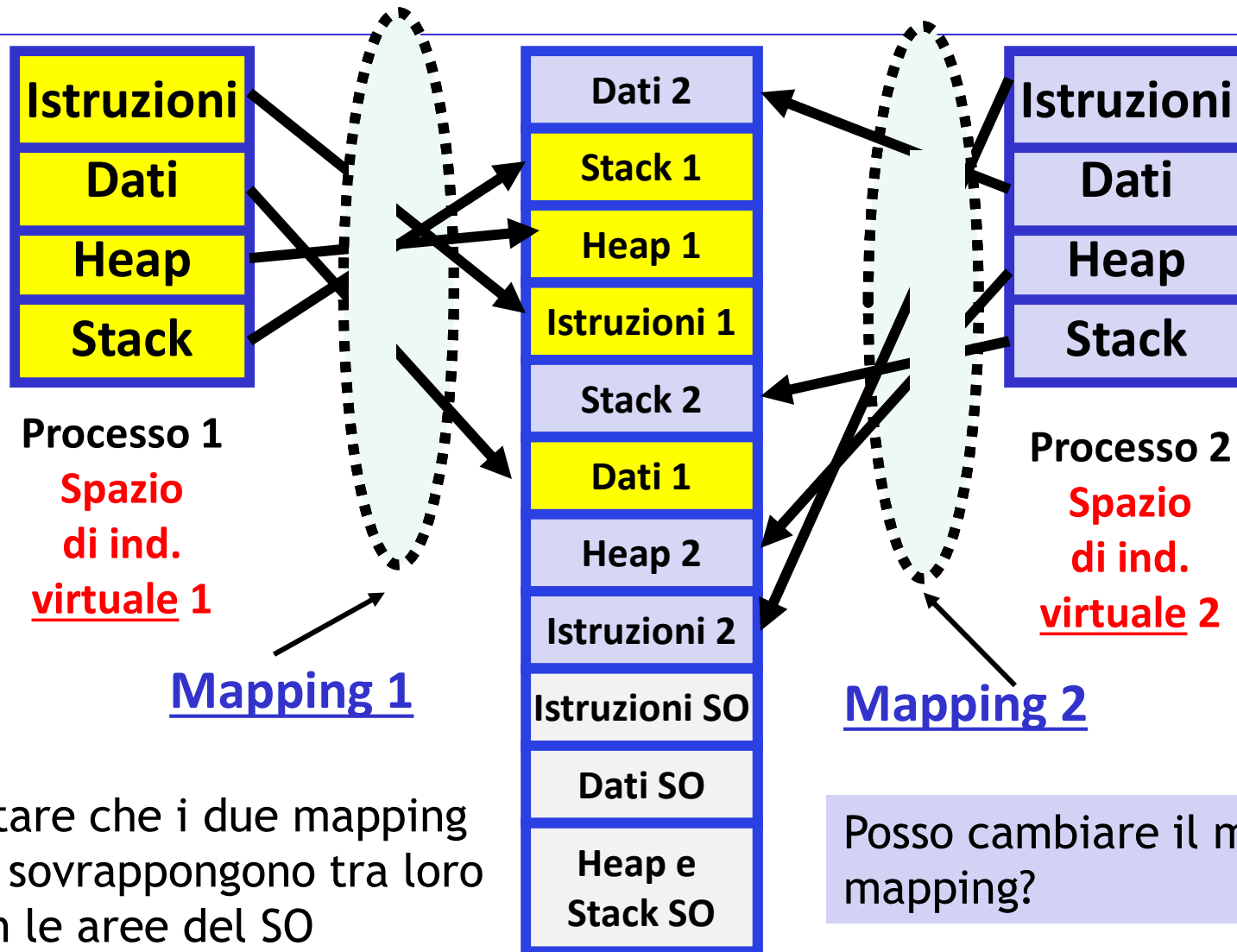
- La coesistenza di più spazi di indirizzamento richiede una traduzione degli indirizzi da logici a fisici



Cambio dello spazio di indirizzamento

- ❑ Il thread in esecuzione referencia dati e istruzioni nel proprio spazio di indirizzamento logico
- ❑ Per fornire ai processi l'illusione di spazi di indirizzamento separati è necessario caricare *la nuova mappa* per la traduzione degli indirizzi da logici a fisici all'atto della commutazione dello spazio di indirizzamento
- ❑ → Commutazione di *processo*
- ❑ Quando parliamo di processi:
 - Per gli aspetti di *concorrenza*, si fa riferimento ai thread del un processo
 - Per gli aspetti di *protezione*, si fa riferimento allo spazio di indirizzamento del processo

Realizzazione dello spazio di indirizzamento



Da notare che i due mapping non si sovrappongono tra loro né con le aree del SO

Posso cambiare il mio mapping?

Spazio di ind. fisico



Protezione dei processi

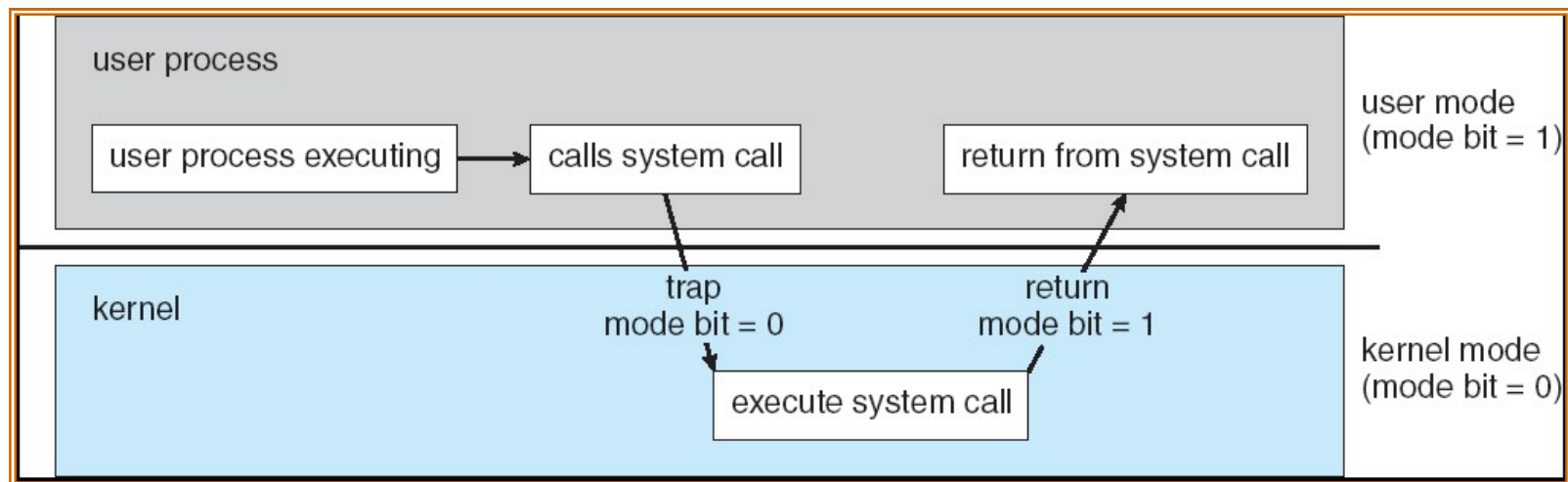
- Per una piena protezione dei processi e del SO, è indispensabile che il processore fornisca almeno due *modi di funzionamento*:
 - modo *utente* (accesso non consentito ad I/O, timer, hardware, registri di memoria, istruzioni riservate)
 - modo *supervisore* o *kernel* (accesso completo ad I/O e hardware in generale, intero set di istruzioni, registri di memoria, etc.)

- La protezione viene realizzata restringendo gli accessi alle risorse private del processo:
 - Il *mapping della memoria* isola i processi tra loro
 - La presenza di *modalità di funzionamento* riservate del processore isola e protegge l'I/O e le altre risorse



Protezione: Modi di funzionamento

- ❑ L'hardware deve fornire almeno due modi di funzionamento: *kernel* (o *supervisor*) e *utente*
- ❑ Alcune istruzioni sono proibite in modo utente e generano un'eccezione (es. modifica tabella pagine)
- ❑ La transizione da modo utente a modo kernel avviene solo con system call, interruzioni, eccezioni





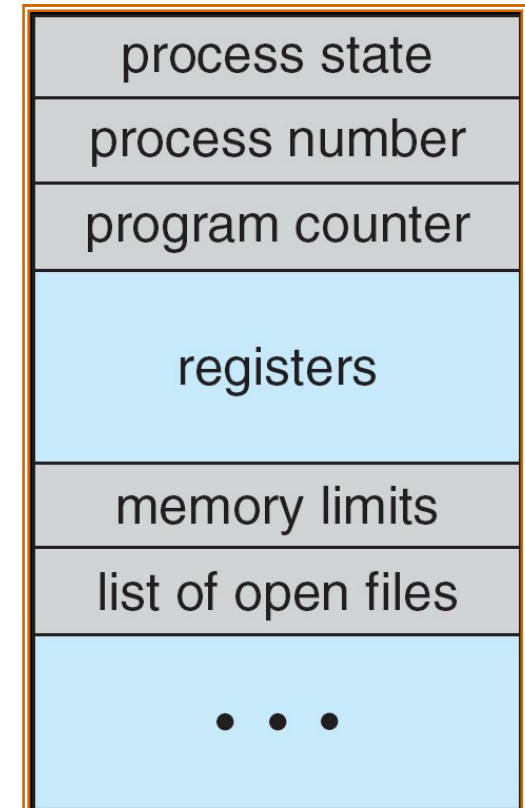
Il processo tradizionale in stile Unix

- ❑ E' l'astrazione utilizzata dal SO per rappresentare *ciò che serve per eseguire un programma*
- ❑ Un singolo flusso di esecuzione nel proprio spazio di indirizzamento
 - talvolta denominato processo *a grana grossa*
- ❑ Consiste di due parti:
 1. Un *unico flusso di esecuzione* sequenziale, a cui è associato anche lo stato di CPU e registri --> *un thread*
 2. Un insieme di *risorse protette*:
 - stato della memoria principale: contenuti dello spazio di indirizzamento,
 - stato dell'I/O: descrittori dei file
- ❑ Che tipo di multiprogrammazione viene resa possibile dai processi tradizionali UNIX? Quali implicazioni?



Per commutare tra i processi

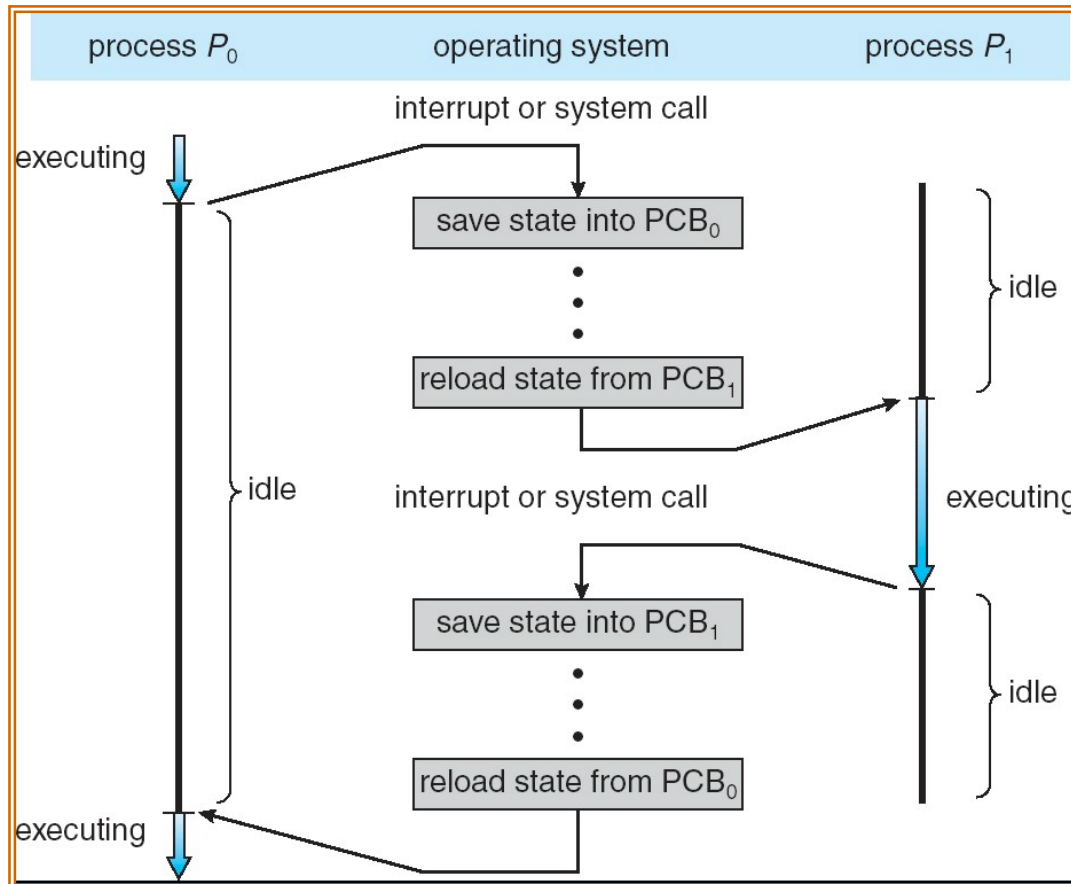
- ❑ Lo stato del processo è mantenuto in un *descrittore* (o *PCB - Process Control Block*)
- ❑ Il tempo di CPU è ripartito tra diversi processi
- ❑ Le risorse sono protette attraverso un accesso controllato
 - *Mappa di memoria*: ogni processo ha uno spazio privato
 - *Modo di funzionamento*
utente/supervisor: il multiplexing dell'I/O avviene attraverso le system call



PCB

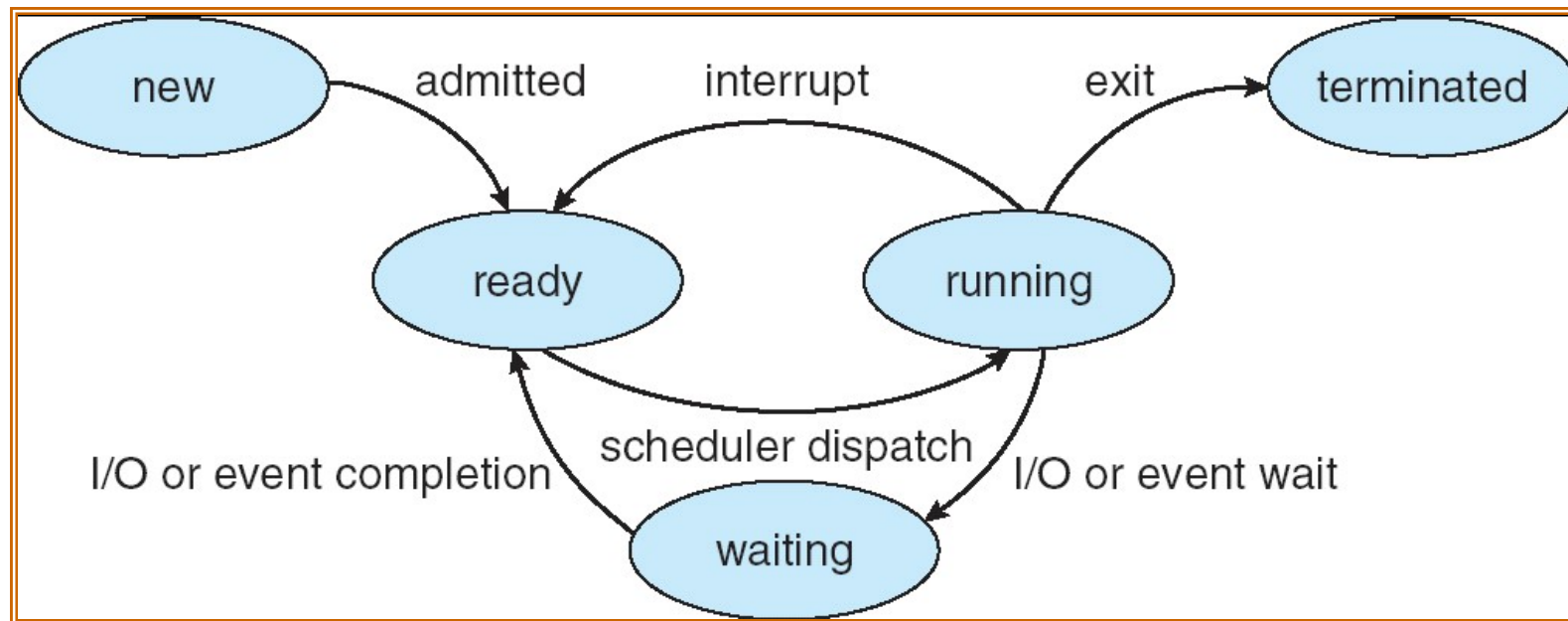


Commutazione della CPU tra processi



- E' denominata *context switch* (cambio di contesto)
- Il codice eseguito dal kernel è *overhead* e limita la frequenza massima di commutazione

Il quadro generale: un sistema con *processi e thread*



New: in corso di creazione

Ready: in attesa di eseguire

Running: in esecuzione

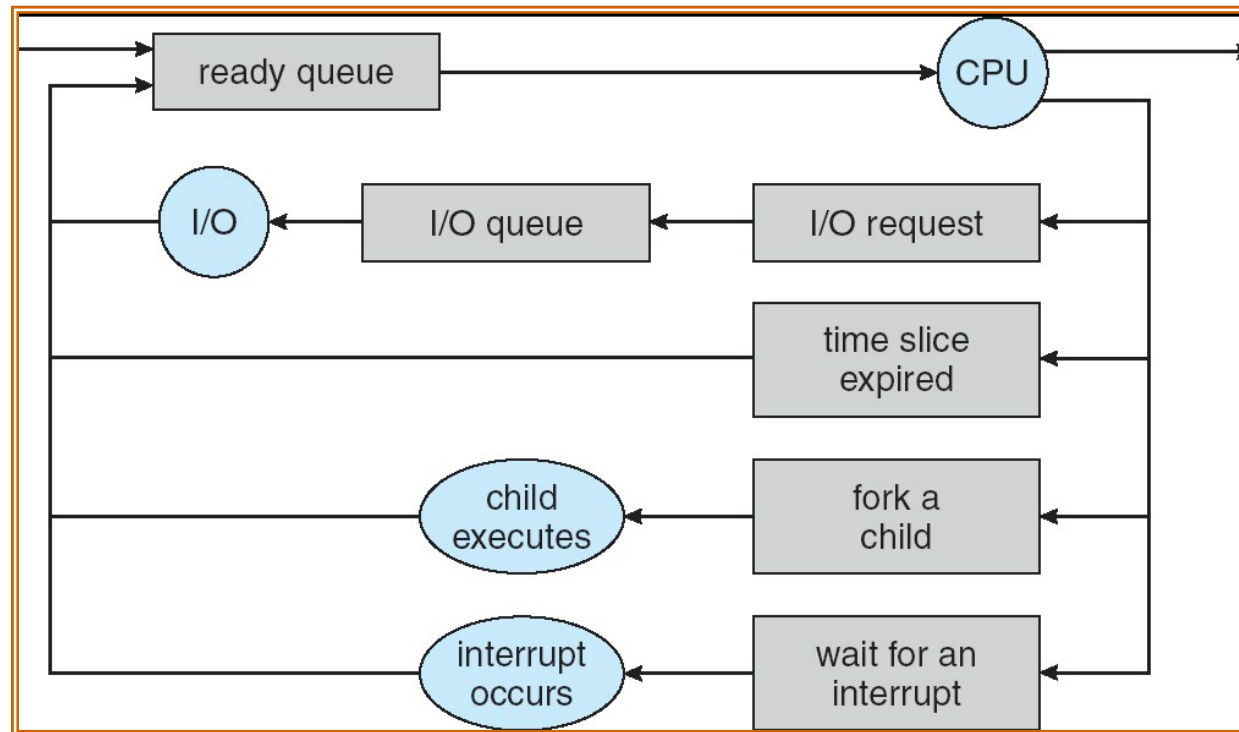
Waiting: in attesa di qualche evento

Terminated: completati, liberano le risorse

- ❑ Quali entità popolano gli stati di questo diagramma?
- ❑ Cosa accade quando cambiano di stato?



Scheduling dei processi



- ❑ I descrittori sono trasferiti da una coda all'altra quando i processi *cambiano stato*
- ❑ Decisione di *Scheduling*: decisione sull'ordine di estrazione di un processo da una coda (-> vari algoritmi)



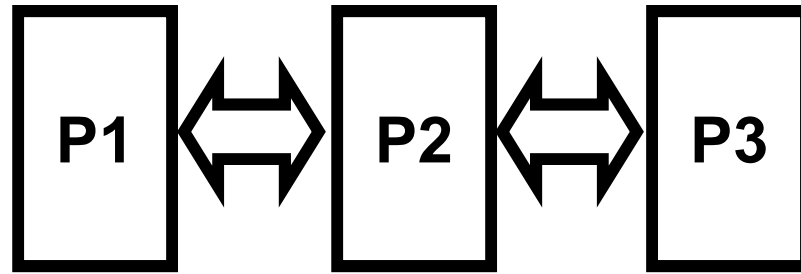
Per la creazione di un processo

- ❑ Il SO deve fornire un nuovo PCB
 - Poco costoso
- ❑ Il SO deve predisporre una nuova tabella delle pagine per lo spazio di indirizzamento
 - Più costoso
- ❑ Copia dei dati dal processo padre? (`fork()` in Unix)
 - La semantica della `fork()` Unix prevede che il processo figlio riceva una copia completa della memoria e dello stato di I/O del padre
 - In origine *molto* costoso
 - Meno costoso utilizzando “copy on write”
- ❑ Copia dello stato di I/O (file handles, etc)
 - Costo intermedio



Cooperazione tra processi

Se ci interessa la
cooperazione?



Problemi nello sviluppo di *applicazioni multi-processo*:

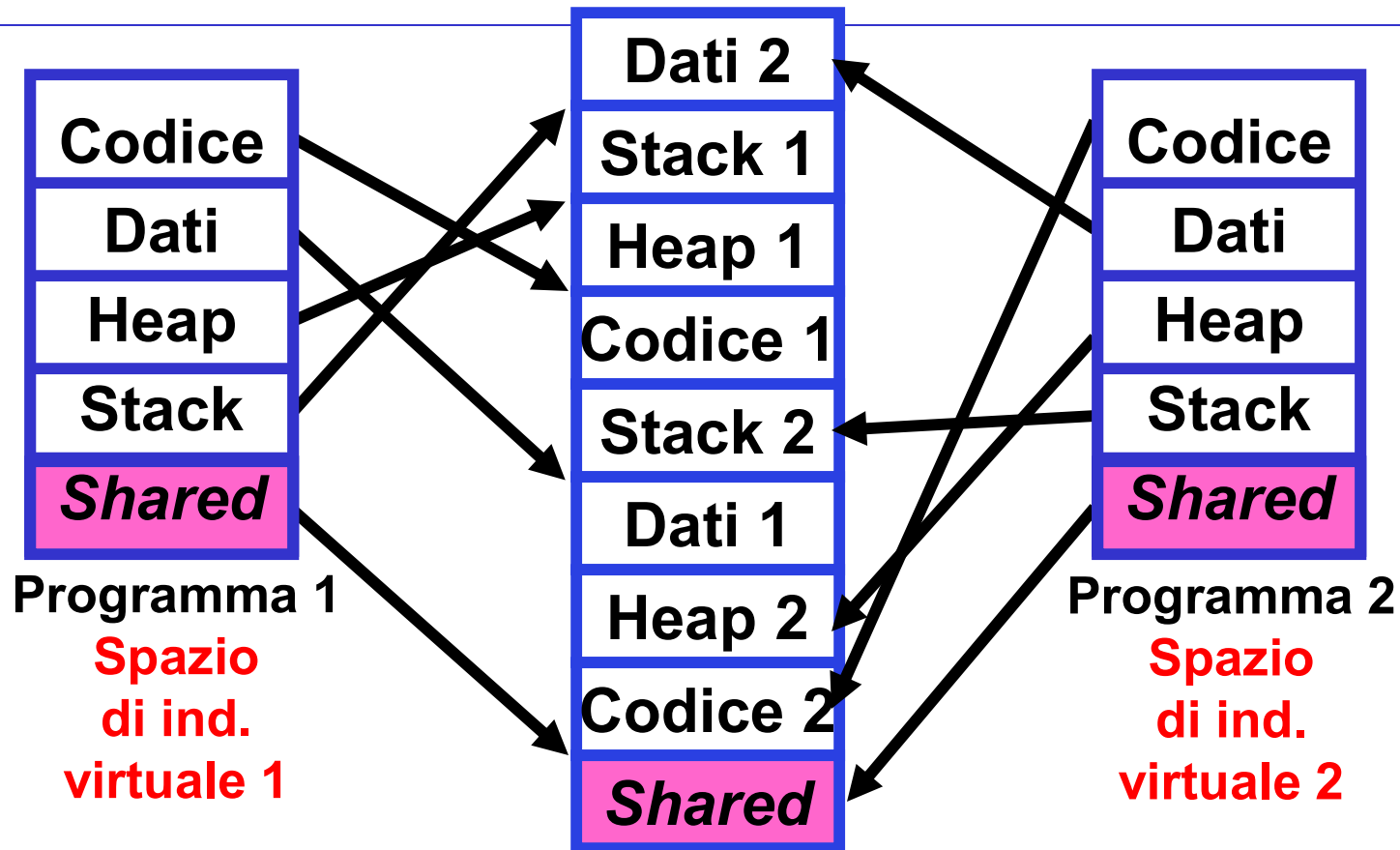
- ❑ Elevato overhead per creazione dei processi e allocazione della memoria
- ❑ Overhead di context switch
- ❑ Necessità di meccanismi di comunicazione:
 - Lo spazio di indirizzamento isola i processi
 - Ad es.: Memoria condivisa, da rendere accessibile a più processi mediante il meccanismo di mapping
 - Oppure: Scambio di messaggi



Comunicazione interprocesso

- ❑ Gli spazi di indirizzamento separati *isolano* i processi
- ❑ Protezione \neq isolamento ...
- ❑ Possibili meccanismi di comunicazione:
 - Mapping di memoria condivisa
 - Realizzato mappando indirizzi ad una porzione di DRAM comune
 - Lettura e scrittura attraverso la memoria
 - Message Passing
 - `send()` e `receive()` di messaggi
 - Si applica anche in sistemi distribuiti su rete

Comunicazione mediante memoria condivisa



- La comunicazione avviene leggendo e scrivendo ad una pagina il cui indirizzo è condiviso
 - overhead di comunicazione contenuto (dipende...)
 - problemi di sincronizzazione non banali



Comunicazione interprocesso (IPC)

- ❑ Meccanismi per comunicazione e sincronizzazione dei processi
- ❑ Sistema a scambio di messaggi: i processi comunicano tra loro senza utilizzare variabili condivise
- ❑ Il sistema IPC fornisce due operazioni base:
 - send**(*message*)
 - receive**(*message*)
- ❑ Due processi *P* e *Q* per comunicare devono:
 - stabilire tra loro un *canale di comunicazione*
 - scambiare messaggi mediante send/receive
- ❑ Realizzazione del canale di comunicazione
 - fisica (es. memoria condivisa, bus, trap)
 - logica (es. proprietà logiche)



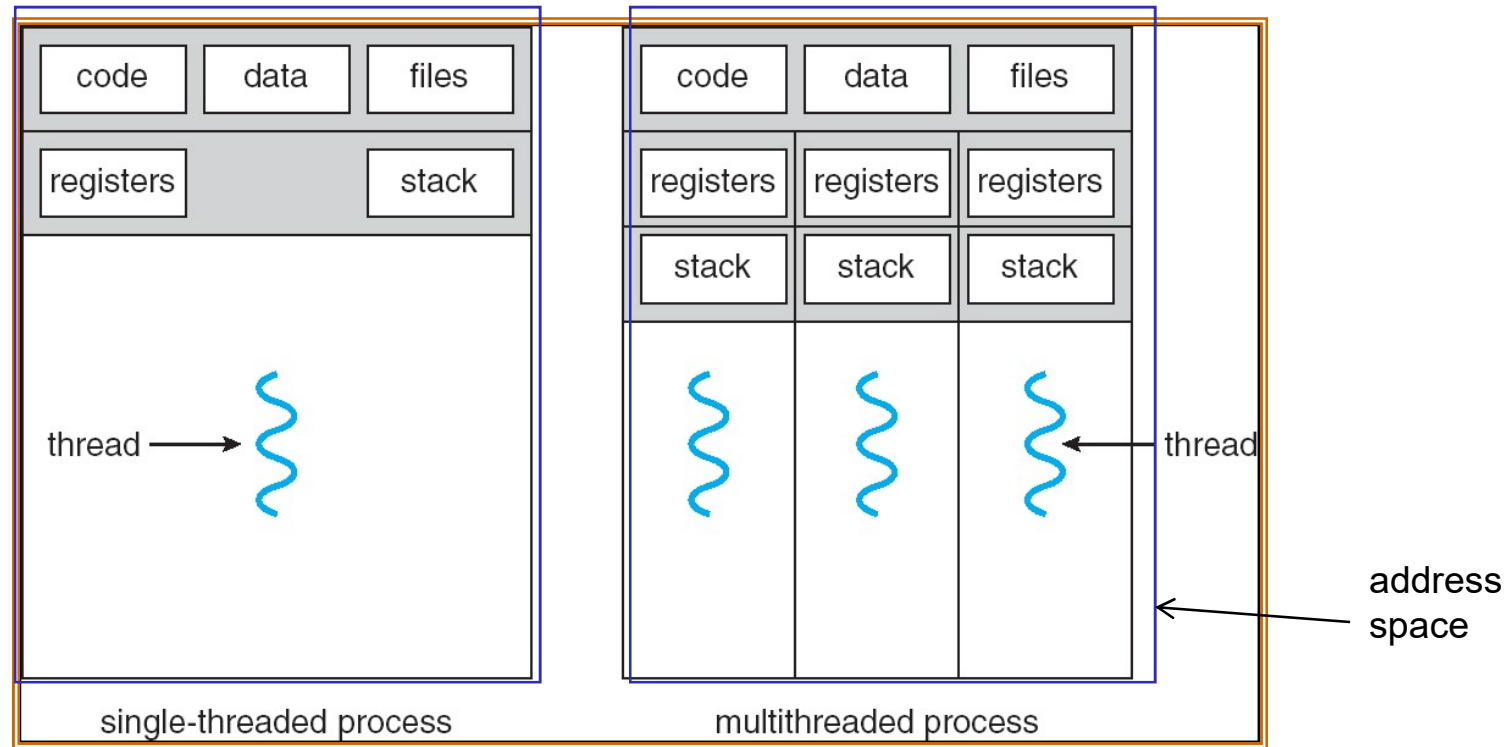


Processi con struttura *a thread*

- ❑ Thread: *flusso di esecuzione sequenziale all'interno di un processo* (denominazione alternativa: “processo leggero”)
 - Il processo è ancora caratterizzato da *un unico spazio di indirizzamento*
 - Nessuna protezione tra i thread del processo
- ❑ Multithreading: *un singolo programma costituito da più attività concorrenti* (anche: multitasking)
- ❑ Separazione del concetto di thread da quello di processo
 - La parte “thread” del processo ne esprime la concorrenza interna ed esterna
 - Lo spazio di indirizzamento esprime la protezione
 - Processo a grana grossa \equiv Processo con un solo thread (processo Unix tradizionale)



Processi con thread multipli



- ❑ Thread -> concorrenza, componente attiva
- ❑ Spazi di indirizzamento -> protezione, componente passiva
 - un programma malfunzionante non può bloccare l'intero sistema
- ❑ Perché più thread in uno stesso spazio di indirizzamento?



Esempi di programmi multithreaded

- ❑ Sistemi embedded
 - Ascensori, aeroplani, apparecchiature medicali, robot autonomi
 - Un solo programma, operazioni concorrenti
- ❑ Nucleo nei SO moderni
 - Concorrenza interna perchè il SO deve gestire richieste concorrenti da parte di più utenti
 - Motivata anche dai processori moderni multicore
 - Nessuna protezione necessaria entro il kernel
- ❑ Server dei Database
 - Accesso a dati condivisi da parte di più utenti contemporanei
 - Elaborazioni interne e di sistema da eseguire in background



Altri esempi di programmi multithreaded

- ❑ Server di rete
 - Richieste concorrenti dalla rete
 - Un solo programma, più operazioni concorrenti
 - File server, Web server, sistemi di prenotazione online
- ❑ Programmazione parallela (CPU multiple)
 - Il programma è scomposto in thread per sfruttare il parallelismo
 - Parallelismo possibile a diverse scale: processori multicore, multicomputer, cluster NoW, sistemi distribuiti, etc.



Stato dei thread

- ❑ Tutti i thread di un processo condividono:
 - I contenuti della memoria (variabili globali, heap)
 - Stato di I/O (file system, connessioni di rete, etc)
- ❑ Stato “privato” di ogni thread
 - Mantenuto nel TCB - Thread Control Block
 - Registri della CPU (tra cui il Program Counter)
 - Stack di esecuzione
- ❑ Stack di esecuzione
 - Contiene parametri e variabili temporanee
 - Memorizza i valori salvati del PC mentre sono in esecuzione i sottoprogrammi
 - Consente chiamate ricorsive





Classificazione dei SO

- Possibilità: uno o più spazi di indirizzamento, uno o più thread per ogni spazio di indirizzamento
 - Un solo spazio di indirizzamento, un solo thread: MS-DOS, primi Macintosh
 - Un solo spazio di indirizzamento con più thread: VxWorks, sistemi embedded, JavaOS, primi sistemi operativi per palmari
 - Più spazi di indirizzamento, un solo thread per ogni spazio di indirizzamento: processo Unix tradizionale
 - Più spazi, più thread per spazio di indirizzamento: Linux, Windows XP, Solaris, Win 7-11, Android, iOS, etc. → i moderni SO general purpose

Why threads are a bad idea (for most purposes)



- <http://www.stanford.edu/~ouster/cgi-bin/papers/threads.pdf>
 - (presentazione di J. Ousterhout a USENIX, 1996)
- Conclusions:
 - *Threads are fundamentally hard; avoid whenever possible*
 - ...

- Ma (anche le idee) ...



A volte ritornano



- ❑ Perché questa idea è ritornata?