Processi & Thread

Obiettivi

- Spiegare il modello concorrente e il concetto di processo
- Descrivere la struttura di un processo in memoria
- Descrivere gli stati e il diagramma degli stati di un processo
- Descrivere le strutture dati usate per la gestione dei processi
- Descrivere le tipologie di scheduling dei processi
- Descrivere creazione, terminazione e comunicazione tra processi
- Spiegare il concetto di thread e confrontarlo con il concetto di processo
- Descrivere vantaggi e problematiche della progettazione di processi multithread
- Descrivere le principali caratteristiche della programmazione multicore
- Descrivere i modelli di multithreading

Processi & Thread

- Modello concorrente e processi
- Scheduling dei processi
- Operazioni sui processi
- Comunicazione tra processi (IPC)
- Comunicazione nei sistemi client-server
- Processi e thread
- Concetto di thread
- Programmazione multicore
- Modelli di multithreading
- Librerie di thread (cenni)

Processi & Thread

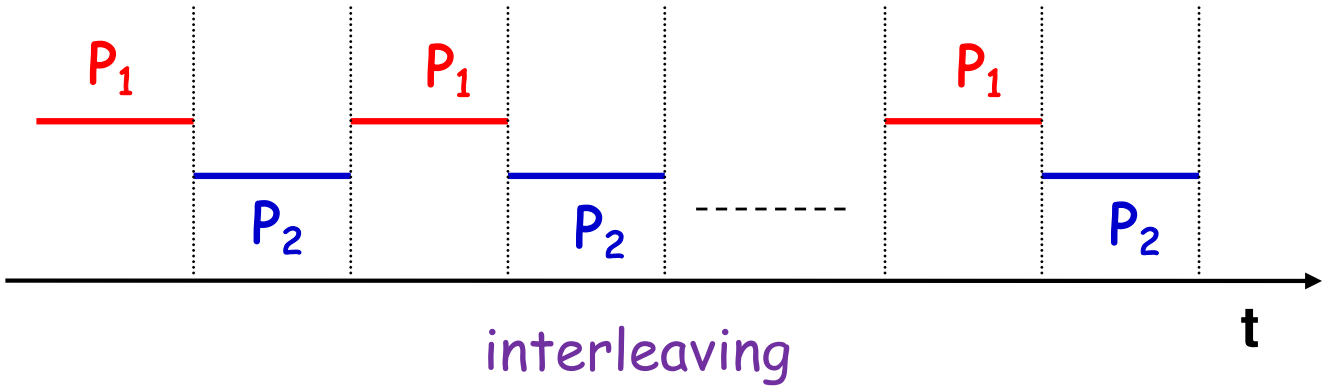
- Modello concorrente e processi
- Scheduling dei processi
- Operazioni sui processi
- Comunicazione tra processi (IPC)
- Comunicazione nei sistemi client-server
- Processi e thread
- Concetto di thread
- Programmazione multicore
- Modelli di multithreading
- Librerie di thread (cenni)

Modello concorrente

- Un sistema di elaborazione esegue, in genere, un grande numero di attività derivanti dall'esecuzione più o meno contemporanea da parte della CPU e dei dispositivi di I/O sia di programmi utente che di sistema
- L'esecuzione di tali attività si sovrappone nel tempo, nel senso che alcune possono avere inizio prima della terminazione di altre
  - Interleaving: l'esecuzione avviene sulla stessa CPU
  - Overlapping: l'esecuzione avviene su CPU diverse
- Senza un modello adeguato, la coesistenza delle diverse attività sarebbe difficile da descrivere ed analizzare
- Il modello che è stato appositamente sviluppato a tale scopo prende il nome di modello concorrente e si basa sul concetto astratto di processo

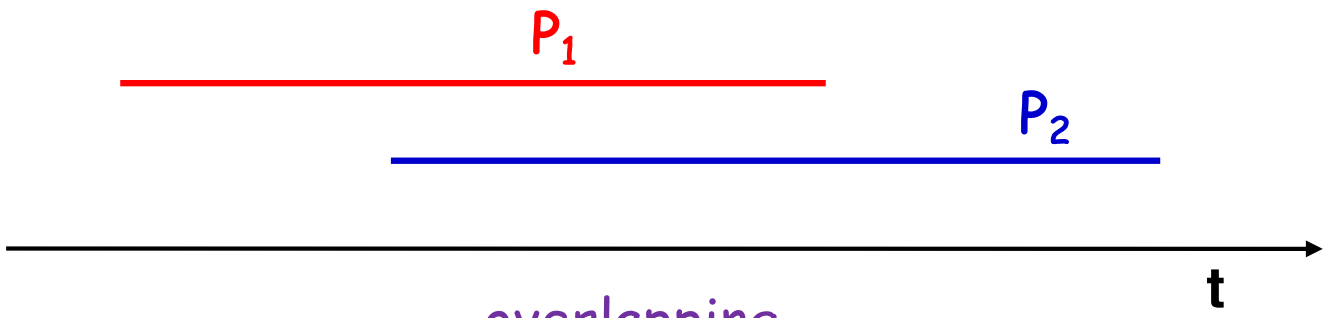
Interleaving / overlapping

Sistema con una singola CPU: esecuzione concorrente



interleaving

Sistema con più CPU: esecuzione parallela



overlapping

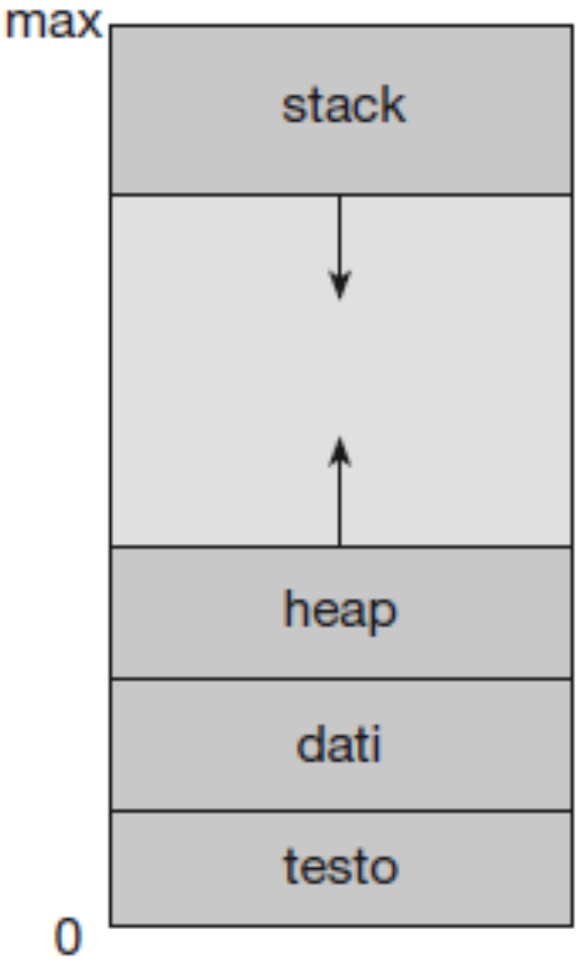
Concetto di processo

- Un programma descrive un algoritmo ed è un'entità passiva/statica, es. un file contenente un elenco di istruzioni memorizzato sul disco (spesso chiamato file eseguibile)
- Un processo è un'entità attiva/dinamica che identifica l'attività del calcolatore relativa all'esecuzione di un programma
  - Lo stato dell'attività corrente di un processo è rappresentato dal valore del registro Program Counter e dal contenuto degli altri registri della CPU
- Un programma diventa un processo quando il corrispondente file eseguibile viene caricato in memoria principale
- I processi forniscono la possibilità di far eseguire ad un calcolatore più programmi "contemporaneamente" (multiprogrammazione e multitasking) tramite condivisione della (o delle) CPU (o core)
- Più processi possono essere associati allo stesso programma (istanze): ciascuno rappresenta l'esecuzione dello stesso codice (es. con dati di ingresso differenti)
  - Es. diversi utenti possono eseguire concorrentemente copie differenti di uno stesso programma per la gestione della posta elettronica

Struttura di un processo in memoria

Lo spazio di indirizzamento di un processo, cioè l'insieme degli indirizzi che il processo può generare, è tipicamente suddiviso in più sezioni

- Testo: contiene il codice eseguibile
- Dati: contiene le variabili globali
- Heap: è un'area di memoria allocata dinamicamente durante l'esecuzione del programma
- Stack: è un'area di memoria temporanea utilizzata durante le chiamate di funzioni (es. per parametri, indirizzo di ritorno, variabili locali)



Struttura di un processo in memoria

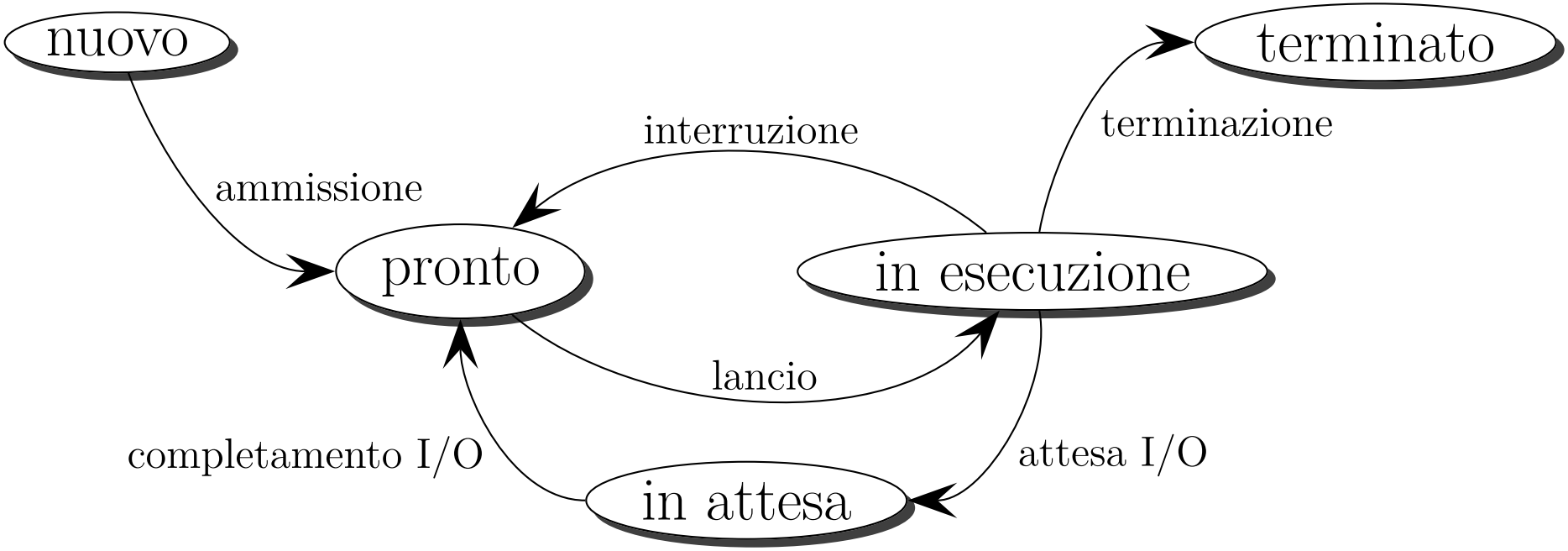
- Le dimensioni delle sezioni testo e dati sono fisse, poiché queste non cambiano durante l'esecuzione del programma
- Invece, le sezioni stack e heap possono ridursi e crescere dinamicamente durante l'esecuzione del programma
- Ogni volta che viene chiamata una funzione, un record di attivazione (*stack frame*) contenente parametri di funzione, variabili locali e indirizzo di ritorno viene inserito sullo stack; quando la funzione chiamata restituisce il controllo, il record di attivazione viene rimosso dallo stack
- Allo stesso modo, l'heap cresce man mano che la memoria viene allocata in modo dinamico e si riduce quando la memoria viene restituita al sistema
- Poiché le sezioni stack e heap crescono l'una verso l'altra, il SO deve assicurarsi che non si sovrappongano

Stati di un processo

Durante il suo ciclo di vita, un processo cambia *stato*

- *nuovo* (new): il processo viene creato
- in esecuzione (running): le istruzioni del processo vengono eseguite
- in attesa (waiting): il processo è in attesa che si verifichi un qualche evento
- pronto (ready): il processo è in attesa di essere assegnato a un processore
- terminato (terminated): l'esecuzione del processo è terminata

Diagramma degli stati di un processo



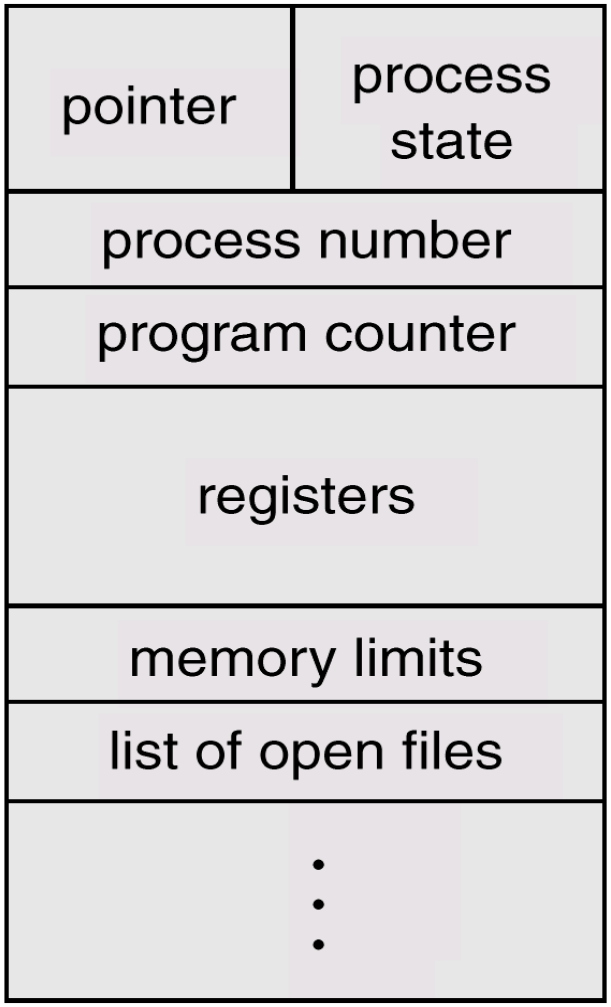
- In un dato momento, su una qualsiasi CPU (o core) può essere eseguito un solo processo
- Tuttavia, molti processi possono essere pronti o in attesa

Descrittore di processo
Process Control Block (PCB)

- Struttura dati del kernel associata ad ogni processo
- Le informazioni contenute nel PCB variano a seconda del SO
- Il PCB contiene 2 tipi di informazioni
  - quelle necessarie durante l'esecuzione del processo (es. privilegi, priorità d'esecuzione, risorse assegnate)
  - quelle necessarie quando il processo non è in esecuzione (es. contenuto dei registri al momento della sospensione)
- Quando un processo in esecuzione va in stato di *pronto* o *attesa,* le informazioni relative al processo vengono salvate nel PCB
- Quando un processo pronto va in stato di *esecuzione,* le informazioni relative al processo vengono recuperate dal PCB
- I PCB di tutti i processi presenti in un sistema sono organizzati nella tabella dei processi, una tabella memorizzata in un'area di memoria principale accessibile solo al kernel del SO

Descrittore di processo
Process Control Block (PCB)

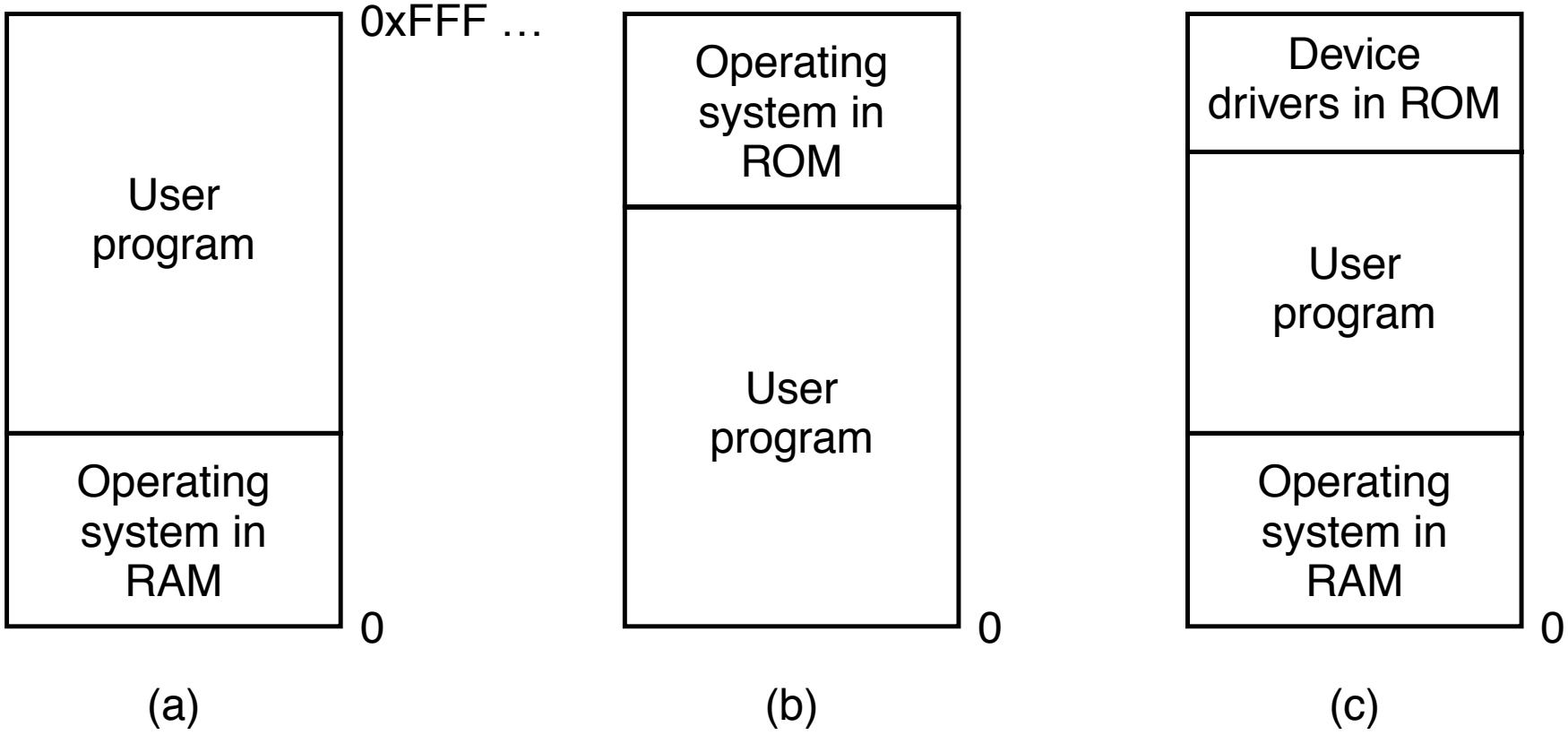
- Pointer, puntatore al successivo PCB in coda, per formare una lista di PCB
- Process State, stato corrente del processo
- Process Number, numero identificativo univoco del processo
- Contesto del processo, cioè program counter e altri registri della CPU (es., stack pointer, program status word)
- Info su gestione della memoria (es. registri base e limite, tabella delle pagine, tabella dei segmenti)
- Info su utilizzo delle risorse (es. file aperti, dispositivi di I/O assegnati, tempo uso CPU, PID genitore, PID figli)
- Info su modalità di servizio (es. FIFO, priorità fissa/variabile, quanto di tempo, deadline)
- Info sull'evento atteso (per un processo bloccato)



Processi & Thread

- Modello concorrente e processi
- Scheduling dei processi
- Operazioni sui processi
- Comunicazione tra processi (IPC)
- Comunicazione nei sistemi client-server
- Processi e thread
- Concetto di thread
- Programmazione multicore
- Modelli di multithreading
- Librerie di thread (cenni)

Monoprogrammazione



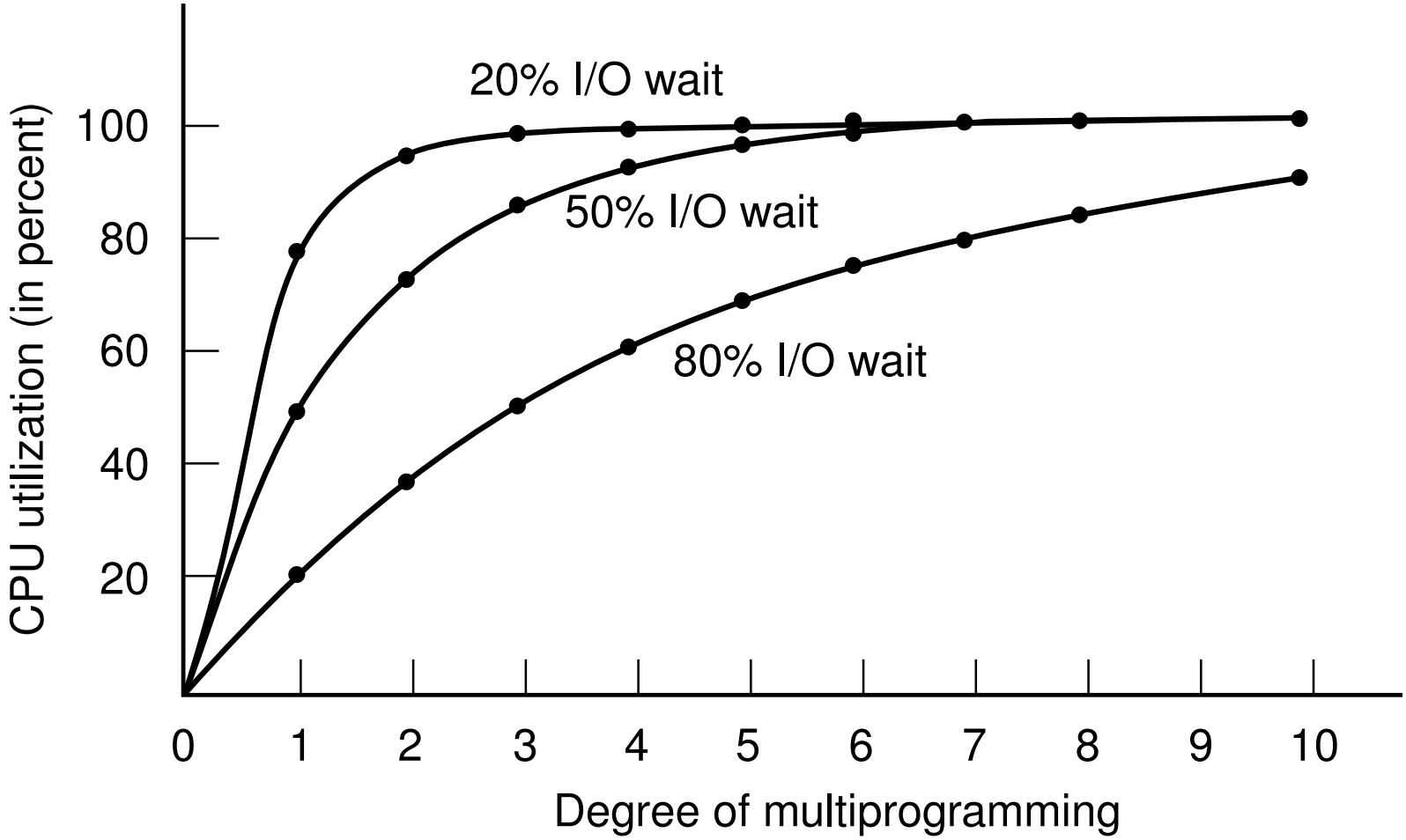
Primi sistemi di elaborazione
(a) e (b) un solo programma per volta in memoria, a parte il SO
(c) è il caso del DOS

Multiprogrammazione

- La monoprogrammazione non sfrutta appieno la CPU
- Idea: se un processo usa la CPU al 20%, 5 processi la usano al 100%
- Sia p x 100% la percentuale di tempo che un processo passa in attesa del completamento di un I/O. Con n processi (indipendenti), la probabilità che tutti siano in attesa di I/O è p^n, quindi abbiamo

  probabilità utilizzo CPU = $1 - p^n$
- Maggiore è il grado di multiprogrammazione (*numero di processi in memoria centrale*), maggiore è l'utilizzo della CPU
- Il modello è impreciso (i processi non sono indipendenti); basandosi sulla teoria delle code si potrebbe ottenere un modello più accurato
- Può però essere utile per stimare la necessità di upgrade della memoria
  - Se un processo passa l'80% del suo tempo in attesa del completamento di un I/O, abbiamo quindi p x 100% = 80%, cioè p = 0,8
  - Supponiamo che un sistema destini 2GB al SO e altrettanti ad ogni programma utente; al variare della memoria varia l'utilizzo della CPU:
    - Memoria = 8GB: grado = 3, utilizzo CPU = 49% (= $(1 - 0{,}8^3) \times 100\%$)
    - Memoria = 16GB: grado = 7, utilizzo CPU = 79% (guadagno 30%)
    - Memoria = 24GB: grado = 11, utilizzo CPU = 91% (guadagno 12%)

Multiprogrammazione



Più è bassa la percentuale di tempo che i processi attendono per il completamento di un I/O, più velocemente cresce la curva e quindi minore è il guadagno in percentuale dell'aumento del grado di multiprogrammazione

Scheduling dei processi

- Obiettivo della multiprogrammazione: utilizzare al meglio la CPU avendo sempre un processo in esecuzione
- Obiettivo del multitasking (o time-sharing): commutare la CPU tra i processi con una frequenza tale che gli utenti possano interagire con ciascun programma mentre è in esecuzione
- Per raggiungere questi obiettivi, il SO
  - mantiene contemporaneamente in memoria un insieme di processi
  - quando la CPU diventa disponibile lo scheduler seleziona un processo tra quelli pronti per l'esecuzione
- Bilanciare gli obiettivi della multiprogrammazione e del time-sharing richiede anche di tenere conto del comportamento generale dei processi, la gran parte dei quali può essere descritta come
  - Processo CPU-bound: effettua una gran quantità di computazioni e genera raramente richieste di I/O, usa la CPU per lunghi intervalli
  - Processo I/O-bound: effettua poche computazioni e genera una grande quantità di richieste I/O, usa la CPU per brevi intervalli
- Prestazioni migliori si ottengono con combinazioni equilibrate di processi CPU-bound e I/O-bound e, a volte, limitando il grado di multiprogrammazione

Strutture dati per lo scheduling dei processi

- La tabella dei processi, contenente i PCB di tutti i processi presenti nel sistema e memorizzata in un'area di memoria principale accessibile solo al kernel del SO
- Il PCB del processo in esecuzione, puntato da un registro della CPU gestito dal SO
- Un certo numero di 'code' di processi: durante il loro ciclo di vita i (PCB dei) processi passano da una coda ad un'altra
  - Coda dei processi pronti: coda formata dai (PCB dei) processi residenti in memoria principale e pronti per essere eseguiti
  - Coda di un dispositivo di I/O: coda formata dai (PCB dei) processi in attesa per l'I/O sul dispositivo

Code dei processi pronti e dei dispositivi di I/O

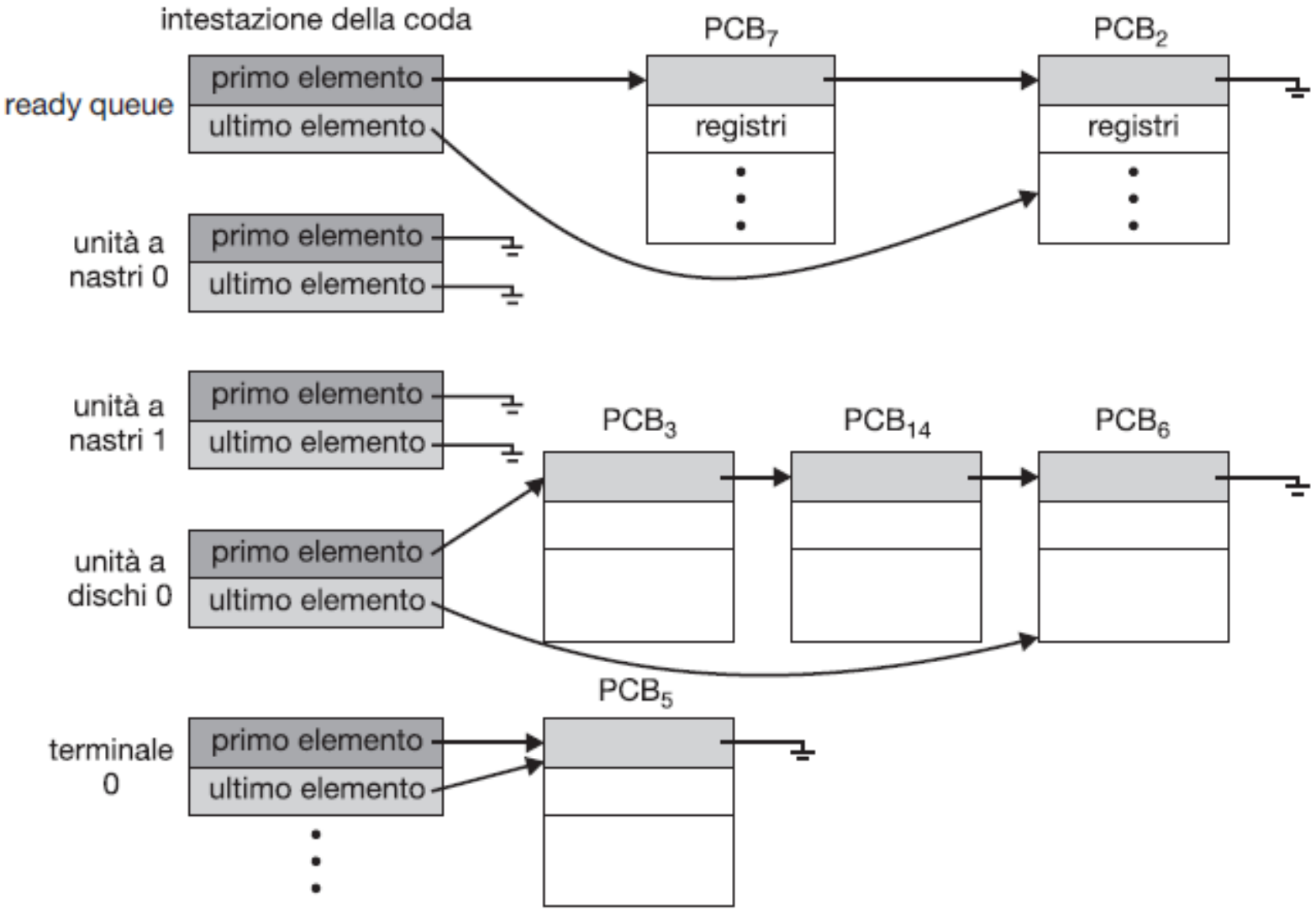
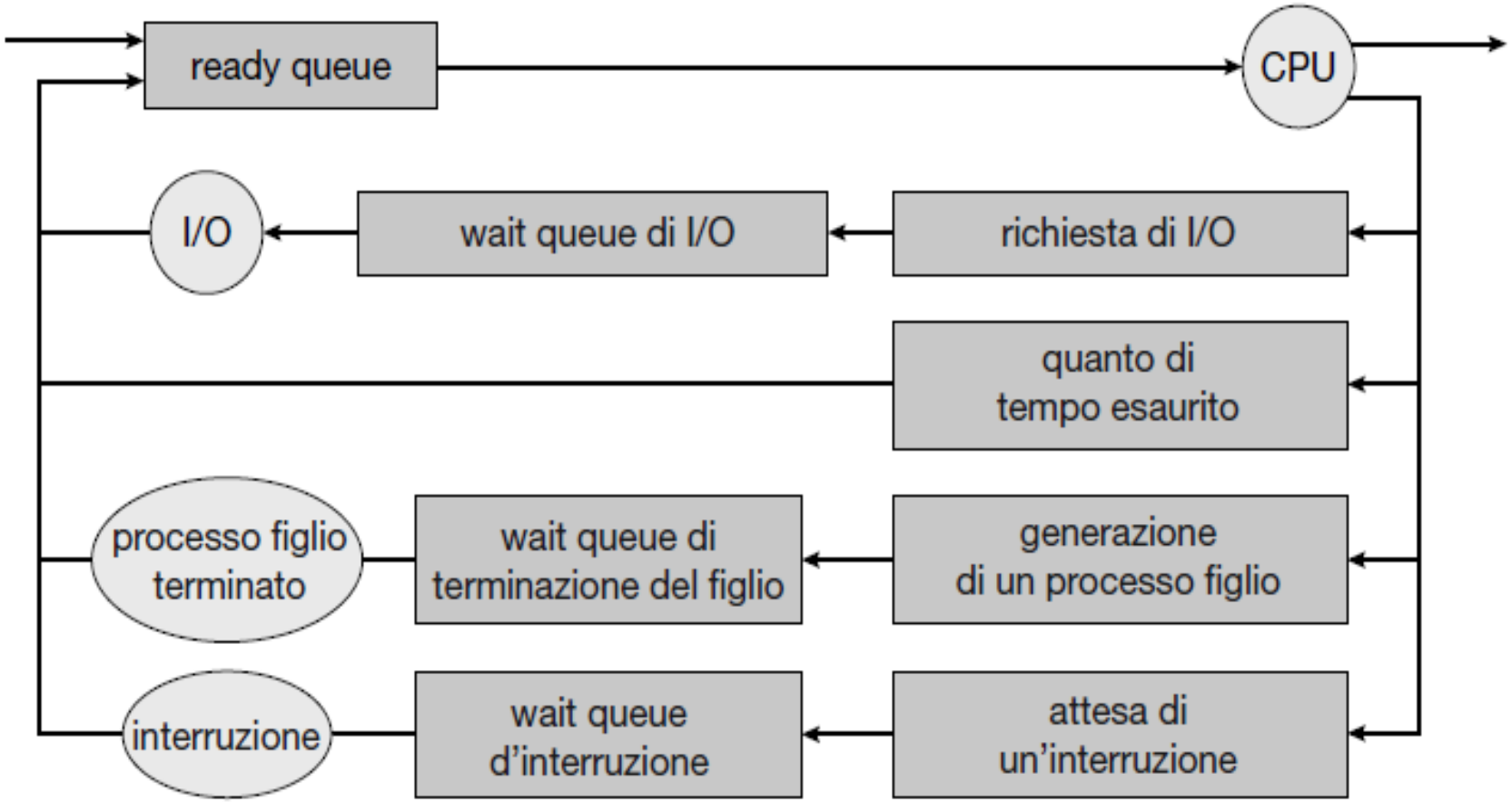


Diagramma di accodamento



- È una rappresentazione del fatto che durante il suo ciclo di vita (il PCB di) un processo passa da una coda ad un'altra
- Due tipi di code: la coda dei pronti e un insieme di code di attesa
- Gli ovali rappresentano le risorse/eventi e le frecce indicano il flusso dei processi nel sistema

Tipologie di scheduling dei processi

- Lo scheduling è l'attività mediante la quale il SO effettua delle scelte tra i processi riguardo a:
  - assegnazione della CPU
  - assegnazione della memoria principale
- 3 tipologie di scheduling:
  - Scheduling a breve termine (o della CPU)
    - Sceglie tra i processi pronti quello a cui assegnare la CPU
    - Interviene quando il processo in esecuzione perde il diritto di usare la CPU (con o senza prelazione)
  - Scheduling a medio termine (swapping)
    - Trasferisce temporaneamente processi parzialmente eseguiti in memoria secondaria
    - Interviene quando la memoria principale è inferiore alla necessità complessiva dei vari processi
  - Scheduling a lungo termine
    - Sceglie tra i programmi in memoria secondaria quali caricare in memoria principale

Tipologie di scheduling dei processi

- Lo scheduler a breve termine viene eseguito molto frequentemente (millisecondi) ed è quindi importante che sia molto veloce
- Lo scheduler a lungo termine viene eseguito con una frequenza che dipende dal *grado di multiprogrammazione* del sistema
  - Permette di controllare il grado di multiprogrammazione
- Lo scheduler a medio termine è presente in alcuni sistemi, quali ad esempio i sistemi time-sharing
  - La rimozione (swap-out) dalla memoria principale (e quindi dalla contesa per la CPU) di processi parzialmente eseguiti e la loro successiva reintroduzione (swap-in) può portare a dei vantaggi grazie a
    - riduzione del grado di multiprogrammazione
    - migliore bilanciamento delle tipologie di processi

Diagramma degli stati di un processo

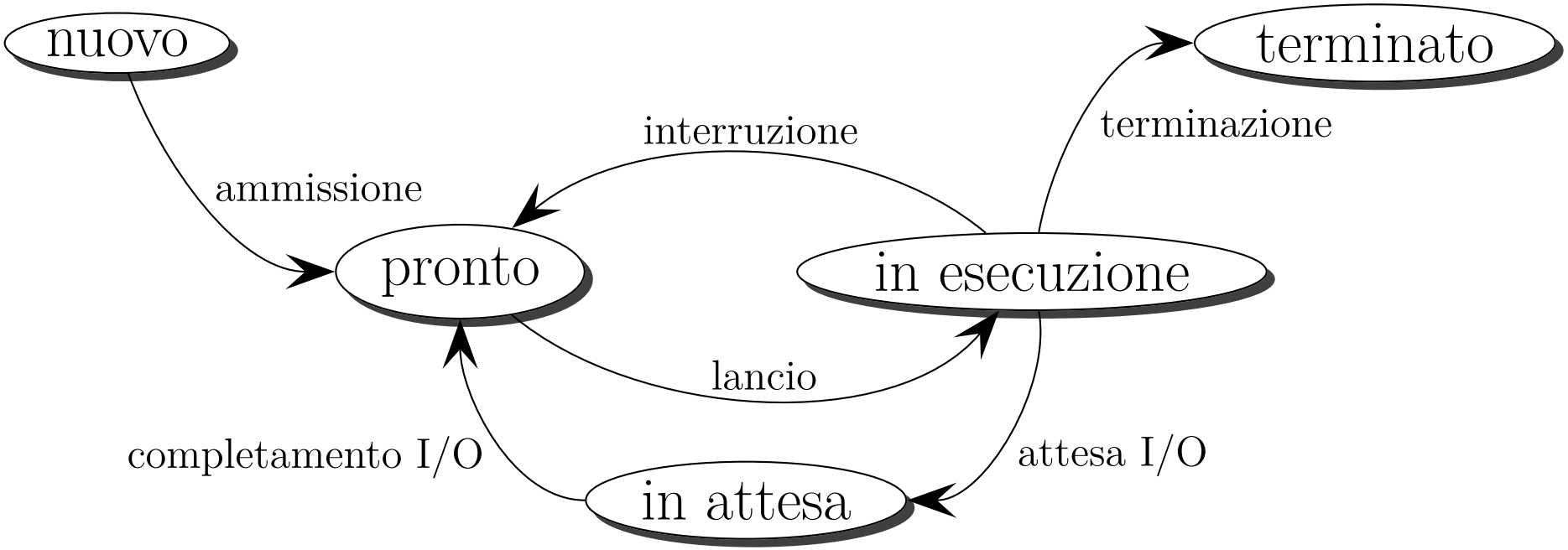
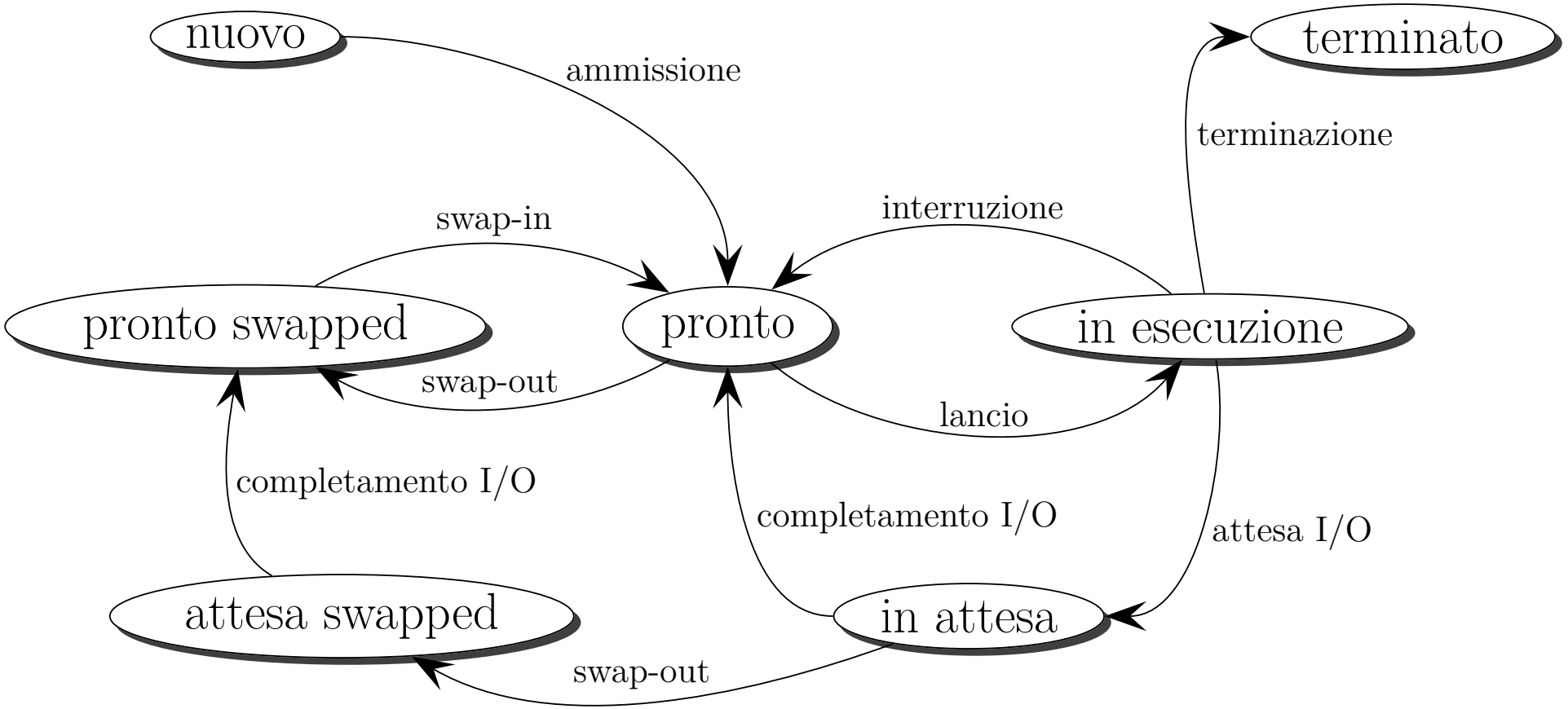


Diagramma degli stati di un processo con swapping



Si aggiungono 2 stati corrispondenti a processi pronti o in attesa il cui spazio di indirizzamento si trova nella swap area del disco

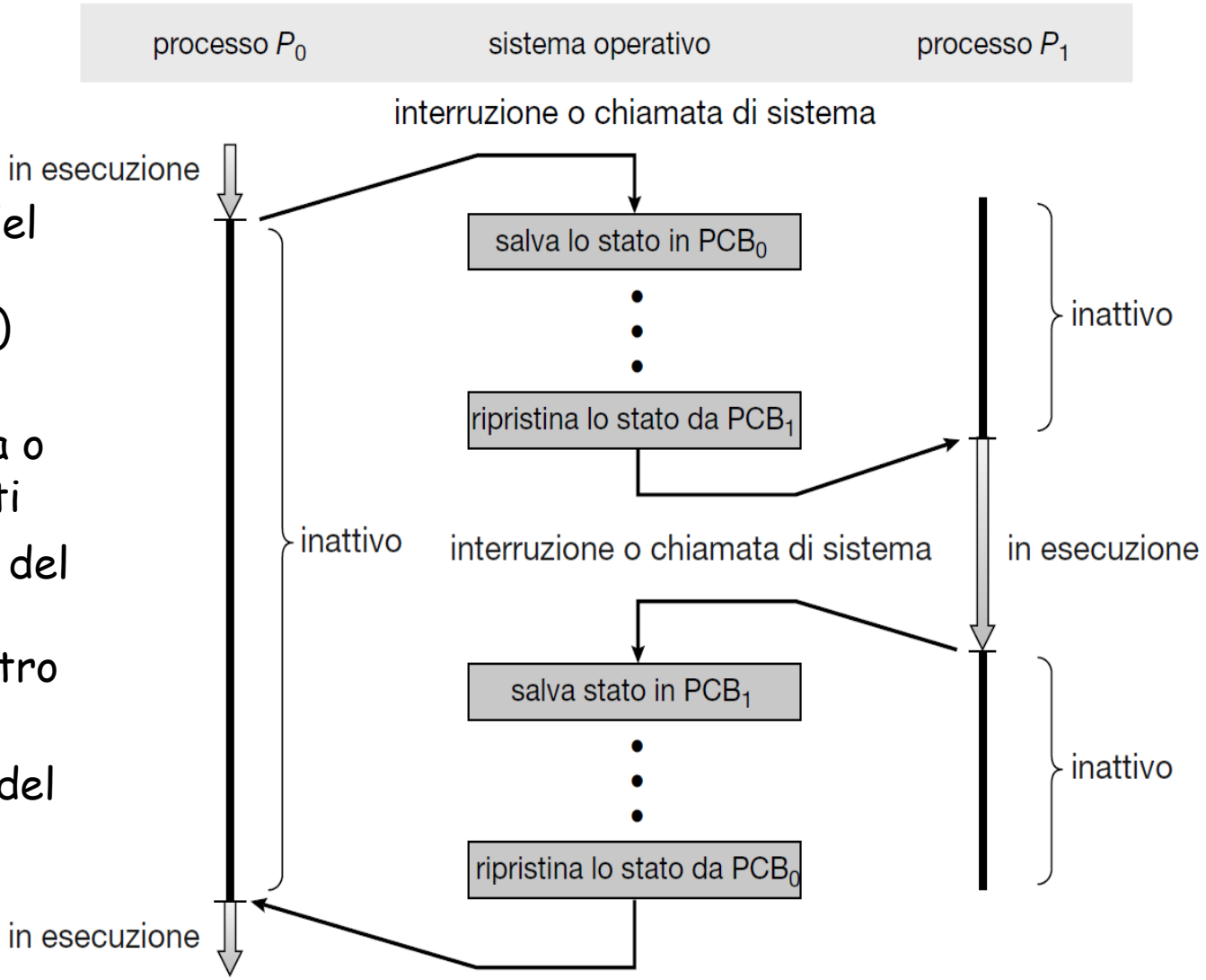
Context switch (commutazione di contesto)

- È la procedura effettuata quando l'utilizzo della CPU passa da un processo ad un altro
- Consiste nel salvare il contesto d'esecuzione del processo corrente e ripristinare il contesto, eventualmente salvato in precedenza, del processo da eseguire

Utilizzo del PCB per il context switch

Operazioni effettuate

1. Salvataggio del contesto del processo in esecuzione nel suo PCB (salvataggio stato)
2. Inserimento del PCB nelle code dei processi in attesa o in quella dei processi pronti
3. Caricamento dell'indirizzo del PCB del processo a cui assegnare la CPU nel registro 'processo in esecuzione'
4. Caricamento del contesto del processo da eseguire nei registri del processore (ripristino stato)



Context switch (commutazione di contesto)

- Il tempo di context-switch è puro tempo di gestione del sistema, quindi è un overhead: *il sistema non fa un lavoro utile mentre modifica il contesto d'esecuzione*
- Il tempo impiegato tipicamente richiede parecchi microsecondi e dipende anche dal supporto hardware
  - Ad esempio, alcuni processori forniscono più set di registri: in tal caso, un cambio di contesto richiede semplicemente di cambiare il puntatore al set di registri corrente
  - Se però ci sono più processi attivi che set di registri, il sistema deve ricorrere alla copia dei dati di un set di registri da e verso la memoria (come nel caso di un unico set)
- La quantità di lavoro che deve essere eseguita durante un cambio di contesto aumenta al crescere della complessità del SO
  - Ad esempio, tecniche avanzate di gestione della memoria possono richiedere trasferimenti onerosi da/verso la memoria secondaria, per allocare e deallocare gli spazi di indirizzamento dei processi

Processi & Thread

- Modello concorrente e processi
- Scheduling dei processi
- Operazioni sui processi
- Comunicazione tra processi (IPC)
- Comunicazione nei sistemi client-server
- Processi e thread
- Concetto di thread
- Programmazione multicore
- Modelli di multithreading
- Librerie di thread (cenni)

Operazioni sui processi

- Meccanismi forniti dal kernel per la gestione dei processi
  - Creazione
  - Terminazione
  - Interazione tra processi (InterProcess Communication)
- Richiamabili tramite system call
- Ciascuno di questi meccanismi può assumere varie forme

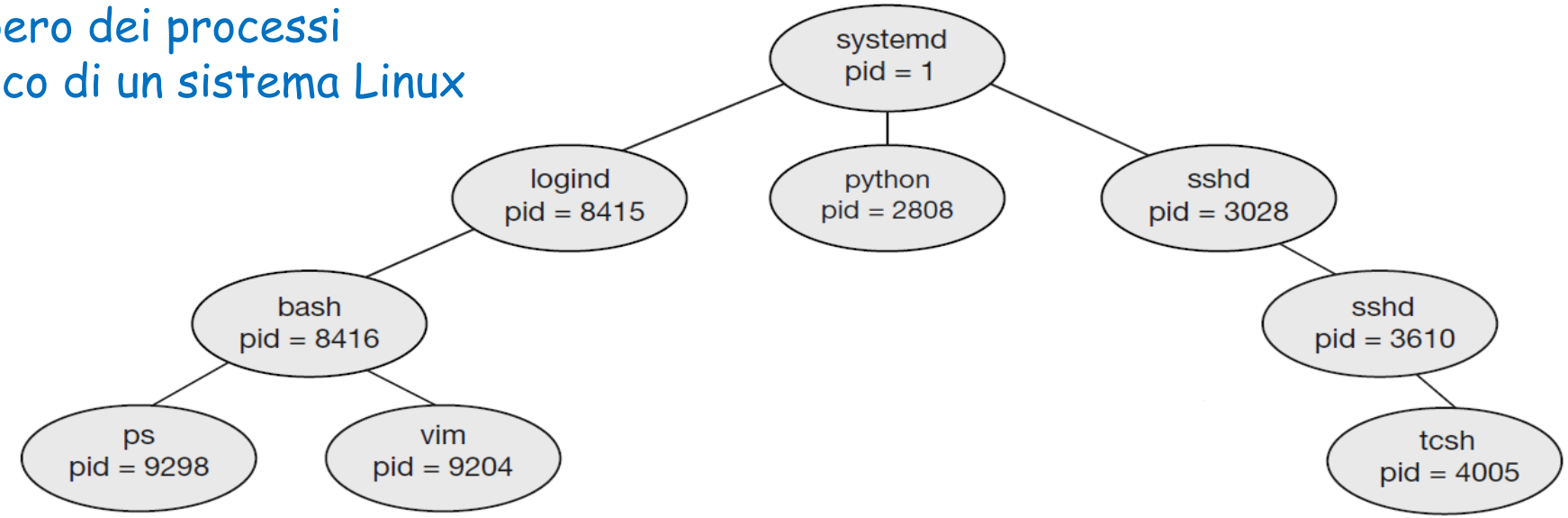
Creazione di processi

- Statica
  - Inizializzazione del sistema (es. controllo impianti industriali)
- Dinamica
  - Invocazione di una system call, che si occupa di creare un processo, da parte di un processo già in esecuzione
  - Richiesta da parte di un utente di creare un nuovo processo
  - Lancio di un programma
- Compiti del SO
  - Assegnare un identificatore unico al processo (pid)
  - Allocare memoria principale (per codice, dati, stack, heap)
  - Allocare altre risorse (es. tempo di CPU, dispositivi di I/O, file)
  - Inizializzare e collegare il PCB con le altre strutture del SO

Albero dei processi

Un processo (*genitore*) può creare dei processi (*figli*) che, a loro volta, possono creare altri processi, formando così un *albero di processi*

Albero dei processi
tipico di un sistema Linux

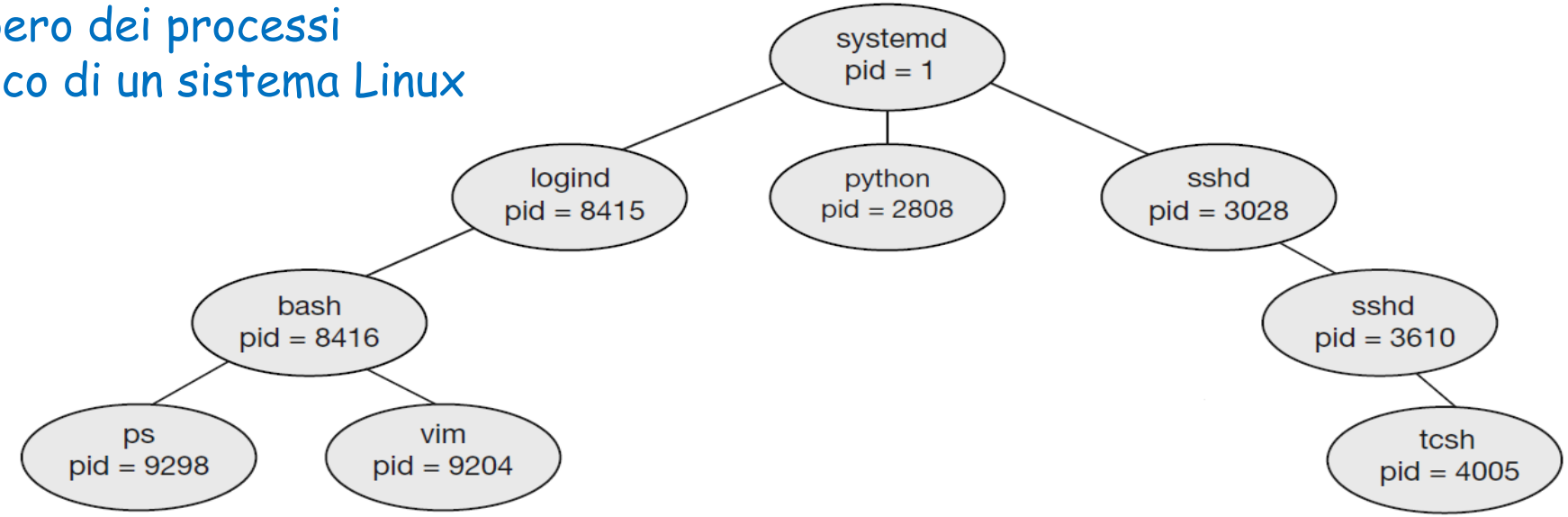


- Il processo `systemd` (che ha sempre pid 1) funge da processo genitore per tutti i processi utente ed è il primo processo creato all'avvio del sistema
- Una volta avviato il sistema, il processo `systemd` crea processi figli che forniscono servizi aggiuntivi, come ad esempio un server stampa, un server web, un server ssh e simili

Albero dei processi

Un processo (*genitore*) può creare dei processi (*figli*) che, a loro volta, possono creare altri processi, formando così un *albero di processi*

Albero dei processi
tipico di un sistema Linux

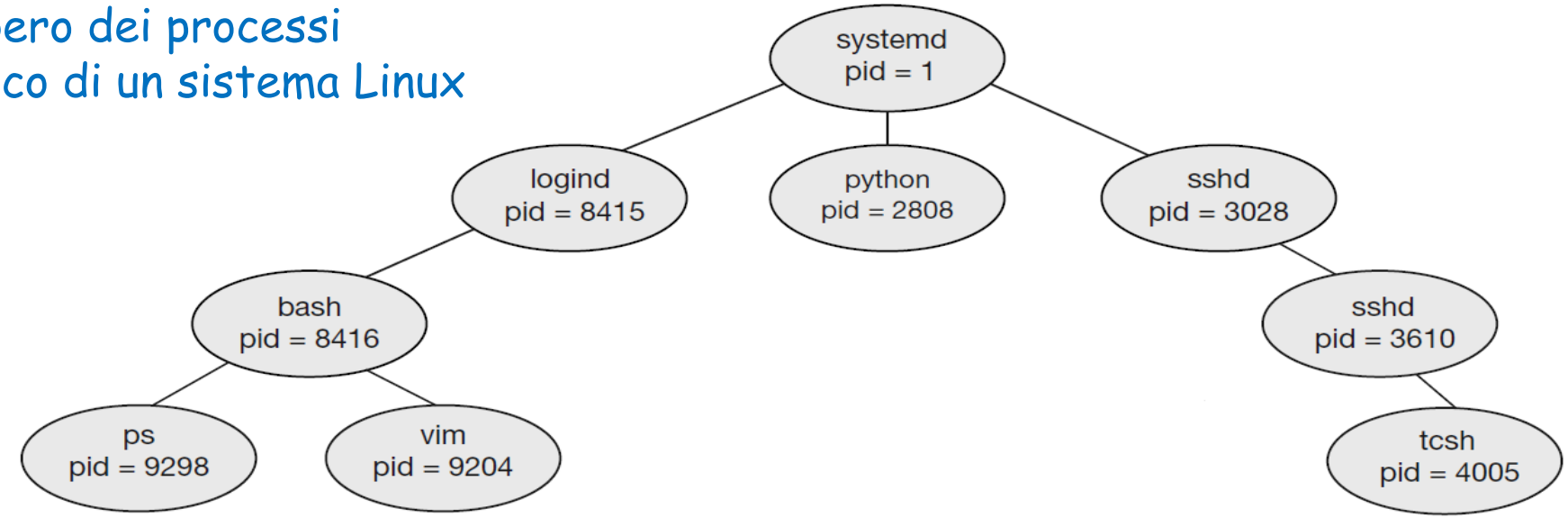


- **`logind`** è responsabile della gestione degli utenti che accedono direttamente al sistema
- In questo esempio, un utente ha effettuato l'accesso e utilizza la shell **`bash`**, a cui è stato assegnato pid 8416
- Utilizzando l'interfaccia di bash a riga di comando, l'utente ha creato il processo **`ps`** e avviato l'editor **`vim`**

Albero dei processi

Un processo (*genitore*) può creare dei processi (*figli*) che, a loro volta, possono creare altri processi, formando così un *albero di processi*

Albero dei processi
tipico di un sistema Linux



- Il processo **`sshd`** è responsabile della gestione degli utenti che si connettono al sistema da remoto utilizzando **`ssh`** (secure shell)

Creazione di processi

- Il kernel deve fornire meccanismi per la realizzazione di politiche per
  - uso delle risorse
    - genitore e figlio condividono tutte le risorse
    - il figlio condivide un sottinsieme di risorse del genitore
    - genitore e figlio non condividono risorse
  - esecuzione
    - genitore e figlio possono essere eseguiti concorrentemente
    - il genitore attende che il figlio termini
  - uso dello spazio di indirizzamento
    - il figlio è un clone del genitore (stesso programma e stessi dati)
    - nel figlio è caricato un programma differente
- Esempi tratti da UNIX
  - la system call **`fork`** crea nuovi processi
  - la (famiglia di) system call **`exec`** può essere usata dopo una **`fork`** per sostituire il programma eseguito da un processo con un altro

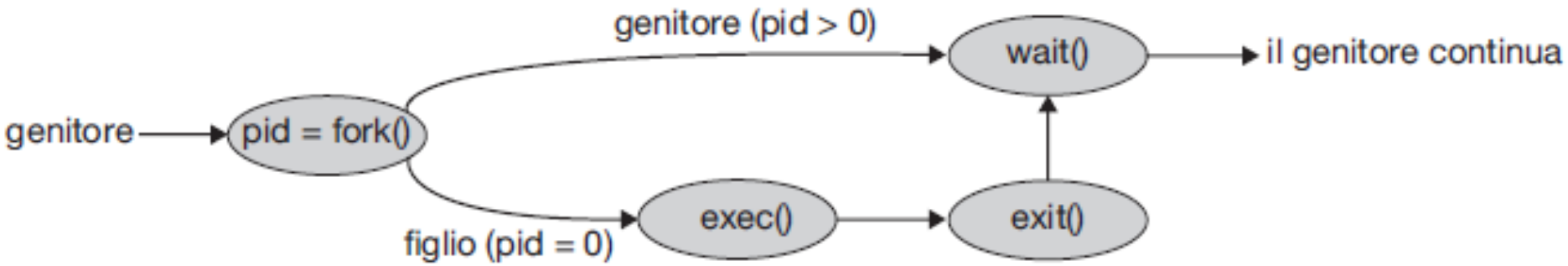
Terminazione di processi

- Può avvenire con o senza il consenso del processo
- Tipiche situazioni di terminazione
  - Uscita normale o per errore (volontaria)
  - Fatal error (involontaria)
  - Terminazione forzata da un altro processo (involontaria)
- Terminazione volontaria: il processo esegue l'ultima istruzione del programma e invoca una **`exit`** (direttamente o indirettamente tramite un'istruzione **`return`**)
  - Le risorse assegnate al processo (memoria fisica e virtuale, file aperti, buffer di I/O) sono deallocate
  - Tramite una **`wait`**, il genitore può ricevere il codice di terminazione inviato dal figlio che esegue una **`exit`**
    - A quel punto, anche il PCB del processo figlio sarà deallocato

Terminazione di processi

- Terminazione forzata da un altro processo: un genitore può forzare la terminazione dell'esecuzione di un figlio (`abort`) perché
  - il figlio ha ecceduto nell'uso delle risorse allocate
  - il task assegnato al figlio non serve più
  - il genitore sta per terminare e il SO non permette ai figli di continuare se il genitore termina (terminazione a cascata)
- Esempio UNIX (Linux ha un comportamento simile):
  - processi zombie: processi terminati ma in attesa che il genitore accetti il loro codice di terminazione (solo allora il PCB del figlio sarà rimosso dalla tabella dei processi)
  - processi orfani: processi il cui genitore è già terminato senza attenderli (vengono adottati dal processo `init` che invoca `wait` periodicamente per accettare i codici di terminazione dei processi rimasti orfani e rilasciare così i loro PCB)

UNIX/Linux: operazioni sui processi



Progamma C che opera su processi

```
#include <sys/types.h>
#include <stdio.h>
#include <unistd.h>

int main()
{
pid_t  pid;
   pid = fork();                          /* fork another process */
   if (pid < 0) {                         /* error occurred */
        fprintf(stderr, "Fork Failed");
        exit(-1);
   }
   else if (pid == 0) {                            /* child process */
        execlp("/bin/ls", "ls", NULL);             /* child executes ls */
   }
   else {                                 /* parent process */
        wait (NULL);                       /* parent waits for the child */
                                           /* to complete */
        printf ("Child Completed");
        exit(0);
   }
}
```

Processi & Thread

- Modello concorrente e processi
- Scheduling dei processi
- Operazioni sui processi
- Comunicazione tra processi (IPC)
- Comunicazione nei sistemi client-server
- Processi e thread
- Concetto di thread
- Programmazione multicore
- Modelli di multithreading
- Librerie di thread (cenni)

Tipologie di Processi

- Alcuni processi sono indipendenti, altri sono interagenti
  - Indipendente: il processo non può condizionare o essere condizionato dall'esecuzione di altri processi
    - Non condivide dati e non scambia informazioni
    - Comportamento deterministico (il risultato dipende solo dagli input) e riproducibile
  - Interagente (o cooperante): il processo può condizionare o essere condizionato dall'esecuzione di altri processi
    - Comportamento nondeterministico: dipende dalle velocità relative dei processi in esecuzione e non può essere previsto
    - Il comportamento può non essere riproducibile
- Le funzioni stesse del SO sono realizzate da processi interagenti

Processi interagenti

Esistono diversi motivi per fornire un ambiente che consenta l'interazione tra processi

- Condivisione di informazioni: applicazioni differenti potrebbero essere interessate agli stessi dati (es. file) cosicché l'ambiente deve consentire l'accesso simultaneo a tali dati
- Prestazioni: per eseguire più velocemente una specifica attività, possiamo suddividerla in attività secondarie, ognuna delle quali verrà eseguita in parallelo alle altre (servono più CPU o CPU multicore)
- Modularità: potremmo voler organizzare una applicazione in modo modulare, suddividendone le funzionalità in processi (o thread) separati

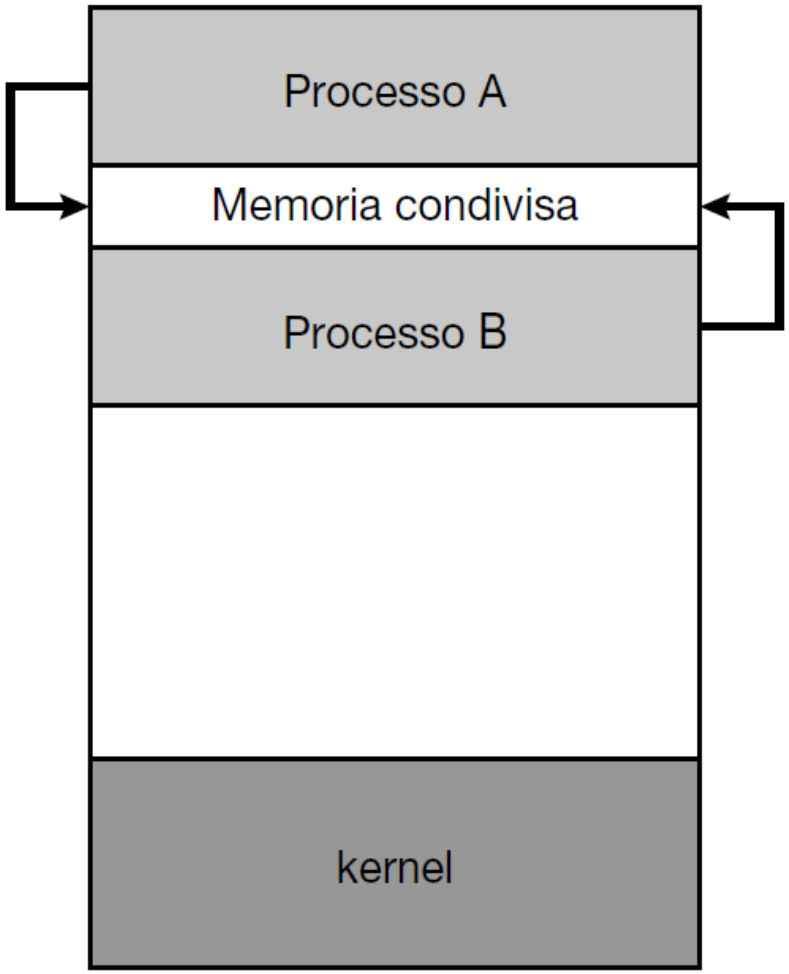
Comunicazione tra processi

- L'interazione richiede un qualche meccanismo di comunicazione tra processi (InterProcess Communication, IPC) che consenta ai processi di scambiarsi informazioni
- I due modelli fondamentali di IPC sono
  - ambiente globale (con memoria condivisa)
  - ambiente locale (con scambio di messaggi)
- Nei SO sono diffusi entrambi i modelli, spesso coesistono in un unico sistema

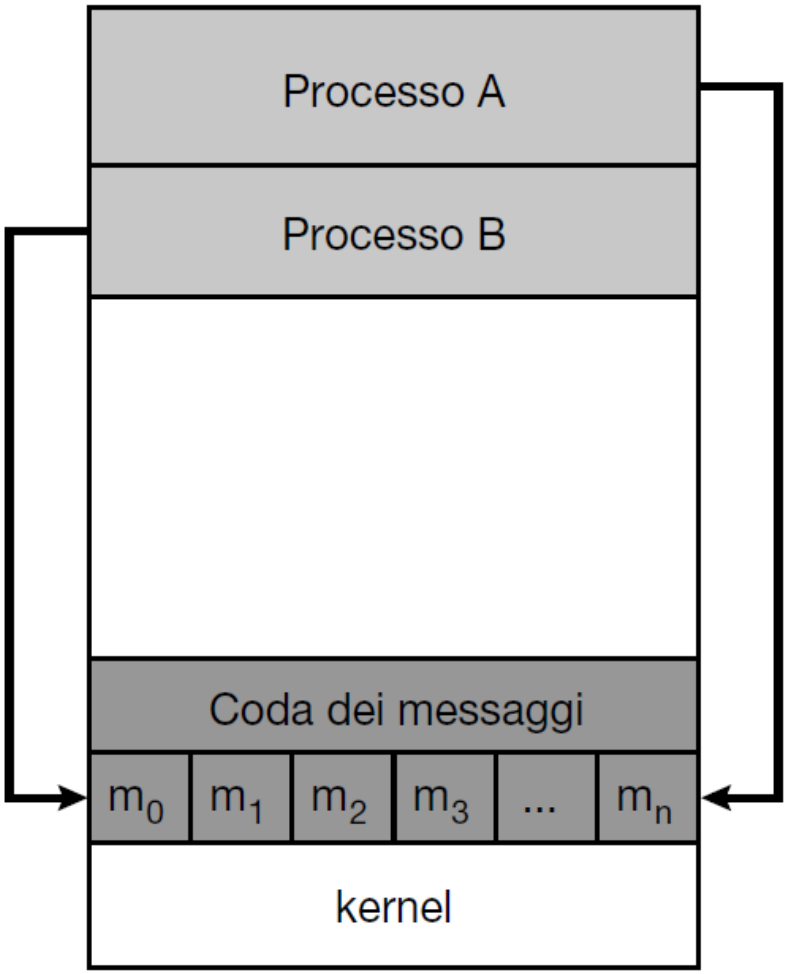
Modelli di IPC

- Ad ambiente globale, cioè con memoria condivisa
  - richiede condivisione (parziale) degli spazi di indirizzamento
  - richiede mutua esclusione sui dati condivisi (per evitare non-determinismo nel comportamento)
  - *strumenti di interazione messi a disposizione dal SO:* semafori e primitive wait e signal, monitor, …
- Ad ambiente locale, cioè senza memoria condivisa
  - gli spazi di indirizzamento dei processi sono separati
  - *strumenti di interazione messi a disposizione dal SO:* messaggi e operazioni send e receive

Modelli di IPC



globale



locale

Modelli di IPC: considerazioni

- L'IPC tramite scambio dei messaggi
  - è più semplice usare per scambiare piccole quantità di dati, poiché non ci sono conflitti da evitare
  - è più facile da implementare in un sistema distribuito
- L'IPC tramite memoria condivisa può essere più veloce
  - Infatti lo scambio di messaggi è generalmente implementato utilizzando le system call e quindi consuma più tempo poiché richiede l'intervento del kernel
  - Invece, nei sistemi a memoria condivisa, le system call sono necessarie solo per stabilire aree di memoria condivisa
    - Dopodiché, tutti gli accessi vengono trattati come accessi ordinari alla memoria e non è richiesta assistenza dal kernel

IPC in sistemi con memoria condivisa

- Normalmente il SO cerca di impedire che due o più processi possano accedere ad una stessa regione di memoria
- Quindi la condivisione della memoria richiede che due o più processi si accordino a rimuovere questa restrizione e stabiliscano invece una regione di memoria condivisa cosicché possano scambiarsi informazioni leggendo e scrivendo dati nell'area condivisa
  - Tipicamente tale regione risiede nello spazio di indirizzamento del processo che la crea
  - Gli altri processi che desiderano comunicare col processo in questione devono associare quella regione al loro spazio di indirizzamento
- I processi interagenti sono responsabili sia del tipo di dati scambiati, sia di sincronizzare le loro operazioni evitando di scrivere simultaneamente nella stessa locazione di memoria

IPC in sistemi con scambio di messaggi

- Il meccanismo di IPC fornisce almeno due operazioni:
  - `send`(*message*)
  - `receive`(*message*)
- Se *P* e *Q* vogliono comunicare, hanno bisogno di:
  - stabilire un *canale (link) di comunicazione* tra loro
  - scambiare messaggi tramite send/receive
- I messaggi scambiati possono avere dimensione fissa o variabile
  - Messaggi di dimensione fissa semplificano l'implementazione a livello di sistema ma complicano l'attività di programmazione
  - Al contrario, messaggi di dimensioni variabili richiedono un'implementazione a livello di sistema più complessa, ma semplificano l'attività di programmazione
- Realizzazione di un canale di comunicazione
  - fisica (es., memoria condivisa, bus hardware, rete)
  - logica (es., proprietà logiche)

Canali di comunicazione

Esamineremo alcuni metodi per implementare logicamente un canale di comunicazione e le operazioni `send/receive`

Problemi realizzativi

- Come sono stabiliti i canali di comunicazione?
- Può un canale essere associato a più di due processi?
- Quanti canali possono esistere tra ogni coppia di processi comunicanti?
- Qual è la capacità di un canale?
- La forma dei messaggi che possono essere scambiati su un canale è fissa o variabile?
- I canali sono unidirezionali o bidirezionali?

Forme di comunicazione

Esamineremo alcuni metodi per implementare logicamente un canale di comunicazione e le operazioni `send`/`receive`

Aspetti

- Comunicazione diretta o indiretta
- Comunicazione sincrona o asincrona
- Gestione del buffer associato al canale

Comunicazione diretta simmetrica

- I processi comunicanti devono nominarsi l'uno con l'altro esplicitamente
  - `send`(*P, message*)
    invia il messaggio *message* al processo *P*
  - `receive`(*Q, message*)
    riceve, nella variabile *message,* un messaggio dal processo *Q*
- Proprietà dei canali di comunicazione
  - I canali sono stabiliti/rimossi automaticamente
  - Un canale è associato ad una sola coppia di processi comunicanti
  - Tra ogni coppia di processi comunicanti esiste esattamente un canale
  - Il canale è solitamente bidirezionale, ma può anche essere unidirezionale

Comunicazione diretta asimmetrica

- Variante con asimmetria nell'indirizzamento: solo il mittente deve nominare il ricevente
  - `send`(*P, message*)

    invia il messaggio *message* al processo *P*
  - `receive`(*id, message*)

    riceve, nella variabile *message,* un messaggio inviato da un qualsiasi processo e registra il nome del processo mittente nella variabile *id*
- Svantaggi (di entrambi gli schemi simm./asimm.): limitata modularità nella definizione dei processi (se si cambia il nome di un processo …)

Comunicazione indiretta

- I messaggi sono inviati/ricevuti a/da porte (o caselle postali o mailbox)
  - Ogni *porta* ha un identificatore (nome) unico
  - Due processi possono comunicare solo se sono entrambi in grado di accedere una stessa porta
- Le primitive di comunicazione sono definite così
  - `send`(*A, message*)

    invia il messaggio *message* alla porta *A*
  - `receive`(*A, message*)

    riceve, nella variabile *message,* un messaggio dalla porta *A*
- Proprietà del canale di comunicazione
  - Un canale è stabilito solo se i processi condividono una porta
  - Un canale può essere associato a molti processi
  - Ogni coppia di processi può condividere diversi canali di comunicazione
  - Un canale può essere unidirezionale o bidirezionale

Comunicazione indiretta

- Condivisione delle porte
  - Supponiamo che P_1, P_2, e P_3 condividano la porta A
  - P_1 invia, P_2 e P_3 ricevono
  - Chi ottiene il messaggio?
- Soluzioni per evitare il nondeterminismo in ricezione
  - Permettere che una porta sia associata soltanto ad al più due processi (un mittente e un destinatario)
  - Permettere ad un solo processo per porta di eseguire operazioni di ricezione
  - Permettere al sistema di selezionare il ricevente in maniera arbitraria (es. 'a turno')
    - Al mittente potrebbe essere comunicato il nome del processo che ha effettivamente ricevuto il messaggio

Comunicazione indiretta

In generale è preferibile avere un unico ricevente per porta: a chi appartiene una porta?

- Ad un processo, cioè la porta fa parte dello spazio di indirizzamento del processo
  - il processo proprietario è l'unico processo ricevente
  - altri processi utente possono solo spedire
  - la porta scompare quando il proprietario termina
- Al SO, il quale offre ai processi alcune operazioni sulle porte
  - creare/rimuovere una porta
  - spedire/ricevere messaggi a/da una porta
  - il processo creante è il proprietario: può passare ad altri processi il diritto di proprietà o di ricezione sulla porta

Comunicazione sincrona & asincrona

Le operazioni `send` e `receive` possono essere bloccanti o non-bloccanti

- Bloccante (o sincrono)
  - Invio bloccante: il processo che invia viene bloccato finchè il messaggio non viene ricevuto dal ricevente o dalla porta
  - Ricezione bloccante: il ricevente si blocca sino a quando un messaggio non è disponibile
- Non bloccante (o asincrono)
  - Invio non bloccante: il processo che invia manda il messaggio e riprende l'attività
  - Ricezione non bloccante: il ricevente acquisisce o un messaggio valido o uno nullo
- Sono possibili varie combinazioni
  - Es. se `send` e `receive` sono entrambe bloccanti si parla di rendezvous tra mittente e ricevente

Gestione del buffer associato al canale

- I messaggi scambiati dai processi che comunicano risiedono in una coda, in altri termini code di messaggi sono associate ai canali di comunicazione
- Ci sono tre modi per implementare tali code:
  1. Capacità 0, cioè la coda non può contenere messaggi
     Il mittente deve bloccarsi finchè il destinatario non riceve il messaggio (rendezvous)
  2. Capacità limitata, cioè la coda ha lunghezza finita
     Il mittente deve bloccarsi se la coda è piena
  3. Capacità illimitata, cioè la coda può avere lunghezza infinita
     Il mittente non si blocca mai
- Nel primo caso si parla di sistema di messaggistica senza buffering, negli altri casi di sistema con buffering automatico

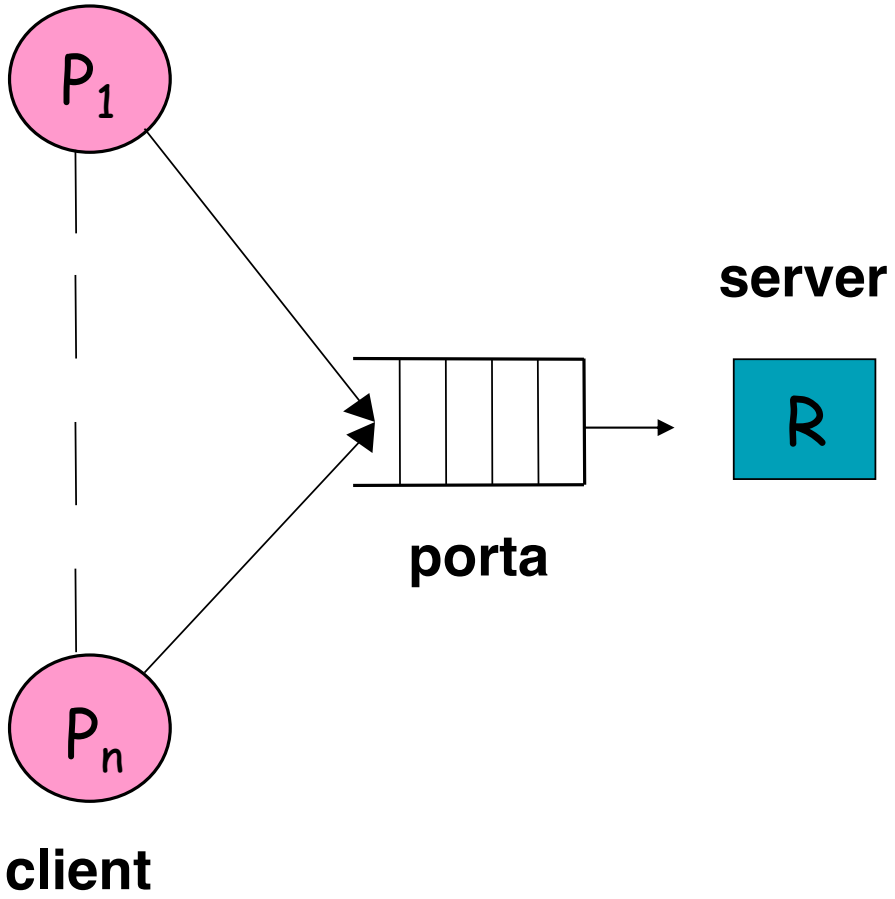
Esempi di sistemi IPC

- POSIX fornisce un'API per la gestione della memoria condivisa
- Il SO Mach utilizza lo scambio di messaggi come forma principale di comunicazione tra processi
- Windows fornisce anche un tipo di scambio di messaggi realizzato usando memoria condivisa
- Le pipe, canali di comunicazione tra processi, sono uno dei primi meccanismi IPC nei sistemi UNIX (sono disponibili anche in Windows con una semantica leggermente diversa)

Processi & Thread

- Modello concorrente e processi
- Scheduling dei processi
- Operazioni sui processi
- Comunicazione tra processi (IPC)
- **Comunicazione nei sistemi client-server**
- Processi e thread
- Concetto di thread
- Programmazione multicore
- Modelli di multithreading
- Librerie di thread (cenni)

Paradigma client-server



Comunicazione nei sistemi client/server

I sistemi client-server, in aggiunta a

- Memoria condivisa
- Scambio di messaggi

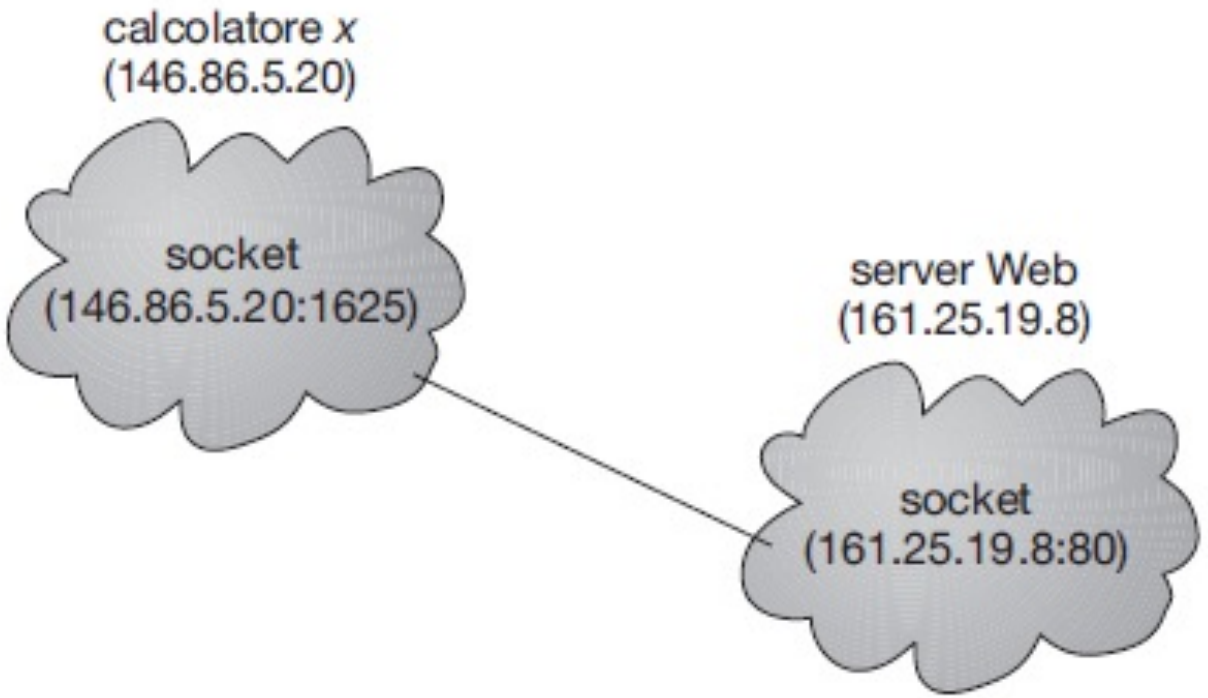
possono utilizzare altri due meccanismi di comunicazione

- Socket
- Remote Procedure Call (RPC)

Socket

- È l'estremità di un canale di comunicazione
- È identificato da un indirizzo IP e da un numero di porta (es. in UNIX)
  - Il socket **161.25.19.8:1625** fa riferimento alla porta **1625** dell'host **161.25.19.8**
  - Server che forniscono servizi specifici stanno in ascolto su porte prestabilite: http 80, ftp 21, telnet 23, …
- Una connessione di rete (canale di comunicazione) viene creata tra una coppia di socket
  - Ogni connessione consiste di un'unica coppia di socket
  - Client connessi con lo stesso server usano socket (lato client) differenti

Comunicazione tra socket



Socket vs. RPC

- La comunicazione tramite socket, sebbene comune ed efficiente, è considerata una forma di comunicazione di basso livello
- La ragione è che i socket consentono ai processi comunicanti di scambiarsi solo uno flusso non strutturato di byte
- È responsabilità del client o dell'applicazione server imporre una struttura sui dati
- Un metodo di comunicazione di livello più alto è fornito dalla chiamata di procedura remota (RPC)
  - Astrae il meccanismo di chiamata di procedura per usarlo fra sistemi collegati tramite una connessione di rete

Remote Procedure Call (RPC)

- Quando il client effettua una RPC, il sistema delle RPC invoca lo stub corrispondente passandogli i parametri di invocazione
  - Stub: segmento di codice risiedente sul client che interagisce con l'effettiva procedura risiedente sul server
- Lo stub del client
  - individua la porta per comunicare con lo stub del server relativo alla procedura,
  - effettua il *marshalling* (strutturazione per la trasmissione in rete) dei parametri da passare alla procedura, e
  - effettua l'*unmarshalling* degli eventuali risultati
- Lo stub del server
  - riceve il messaggio,
  - effettua l'*unmarshalling* dei parametri,
  - invoca la procedura sul server, e
  - effettua il *marshalling* dei risultati e li inoltra al client

Remote Procedure Call (RPC)

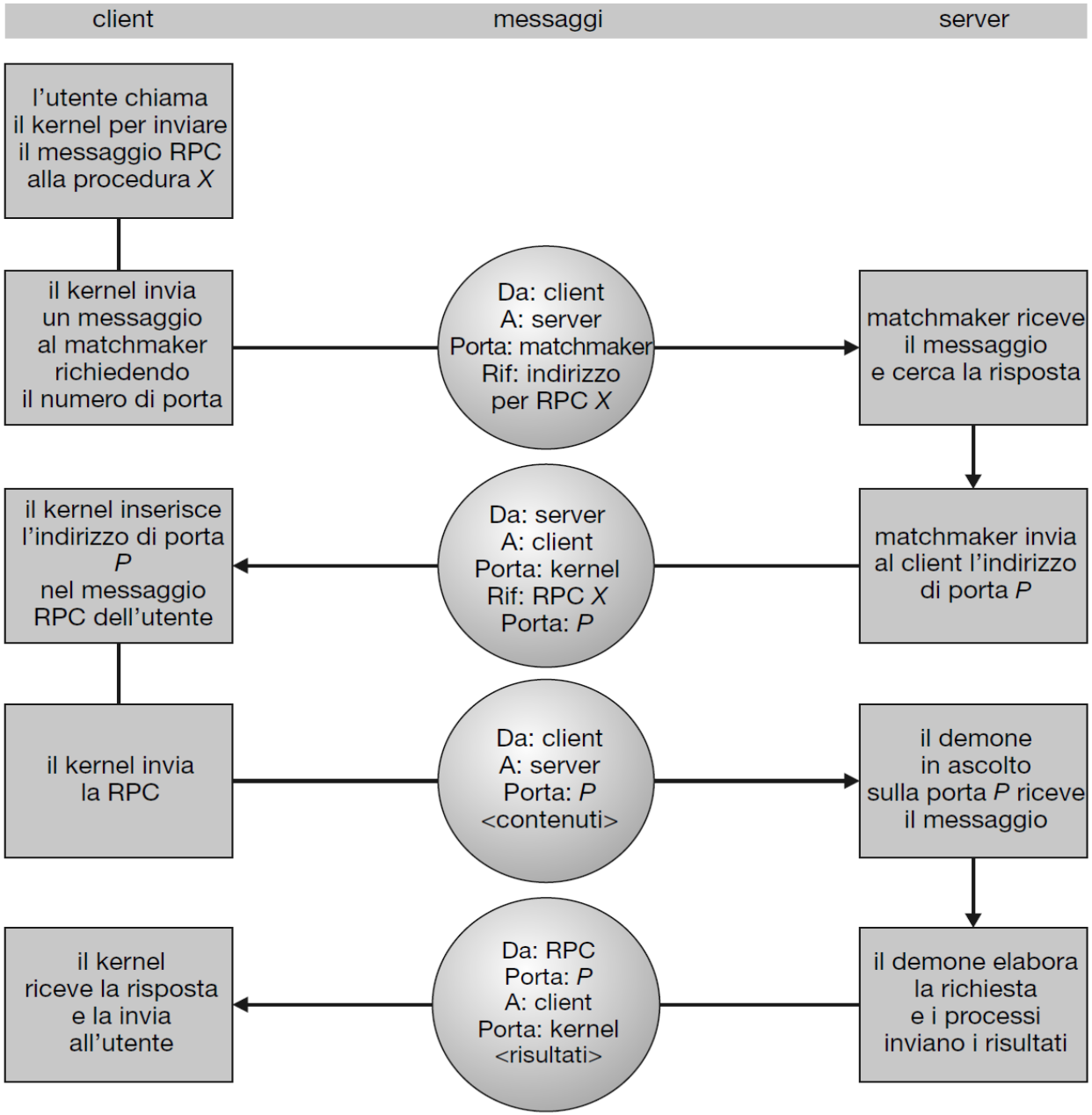
Problema: il server risponde a RPC di tipo diverso, ognuna su una porta differente

- Come fa il client a determinare la porta del server per la RPC che vuole invocare?

Soluzione: sul server, un servizio del SO (matchmaker) risponde su una porta prestabilita e restituisce la porta del server a cui inviare l'effettiva richiesta di RPC

- Quindi, per ogni RPC, sono effettuate due comunicazioni:
  - la prima con il matchmaker,
  - la seconda con l'effettiva RPC

Esecuzione di una RPC



Processi & Thread

- Modello concorrente e processi
- Scheduling dei processi
- Operazioni sui processi
- Comunicazione tra processi (IPC)
- Comunicazione nei sistemi client-server
- Processi e thread
- Concetto di thread
- Programmazione multicore
- Modelli di multithreading
- Librerie di thread (cenni)

Processi e thread

- Processo: è un elemento che possiede

  risorse: sezione di codice, sezione di dati, descrittori dei file aperti, gestori dei segnali, ...

  +

  un flusso di controllo dell'esecuzione: stato della CPU
- Thread (o processo leggero): elemento che rappresenta un flusso di controllo dell'esecuzione
  - Processo *tradizionale* (o processo pesante): elemento che possiede anche risorse
- La separazione e la gestione indipendente dei due aspetti fa sì che un processo possa contenere al suo interno più thread che condividono le risorse assegnate al processo (*multithreading*)

Processi e thread

- L'uso dei thread di fatto divide lo stato del processo in due parti:
  - lo stato delle risorse: è unico e associato al processo
  - lo stato dell'esecuzione, cioè lo stato della CPU: è replicato per ogni thread
- Avere molteplici thread eseguiti in parallelo in un processo è come avere molteplici processi in esecuzione su uno stesso computer
  - Nel primo caso, i thread condividono lo spazio di indirizzamento e le risorse assegnate al processo dal SO
  - Nel secondo, i processi condividono memoria fisica, dischi, stampanti e le altre risorse messe a disposizione dal computer

Processi e thread

- La realizzazione della concorrenza a livello di thread anzicché di processo migliora le prestazioni del sistema
  - L'overhead dovuto al context switch tra processi ha due componenti:
    - overhead relativo all'uso delle risorse
    - overhead relativo al flusso dell'esecuzione
  - La commutazione tra thread di uno stesso processo richiede semplicemente di modificare lo stato dell'esecuzione, cioè lo stato della CPU (che è associato ad ogni thread)
    - Lo stato delle risorse (che è associato al processo) non è modificato
  - Lo stesso dicasi per le operazioni di creazione e terminazione
- Alcune CPU hanno anche un supporto HW diretto per il multithreading e consentono che il context switch tra thread avvenga in tempi dell'ordine dei nanosecondi

Processi e thread

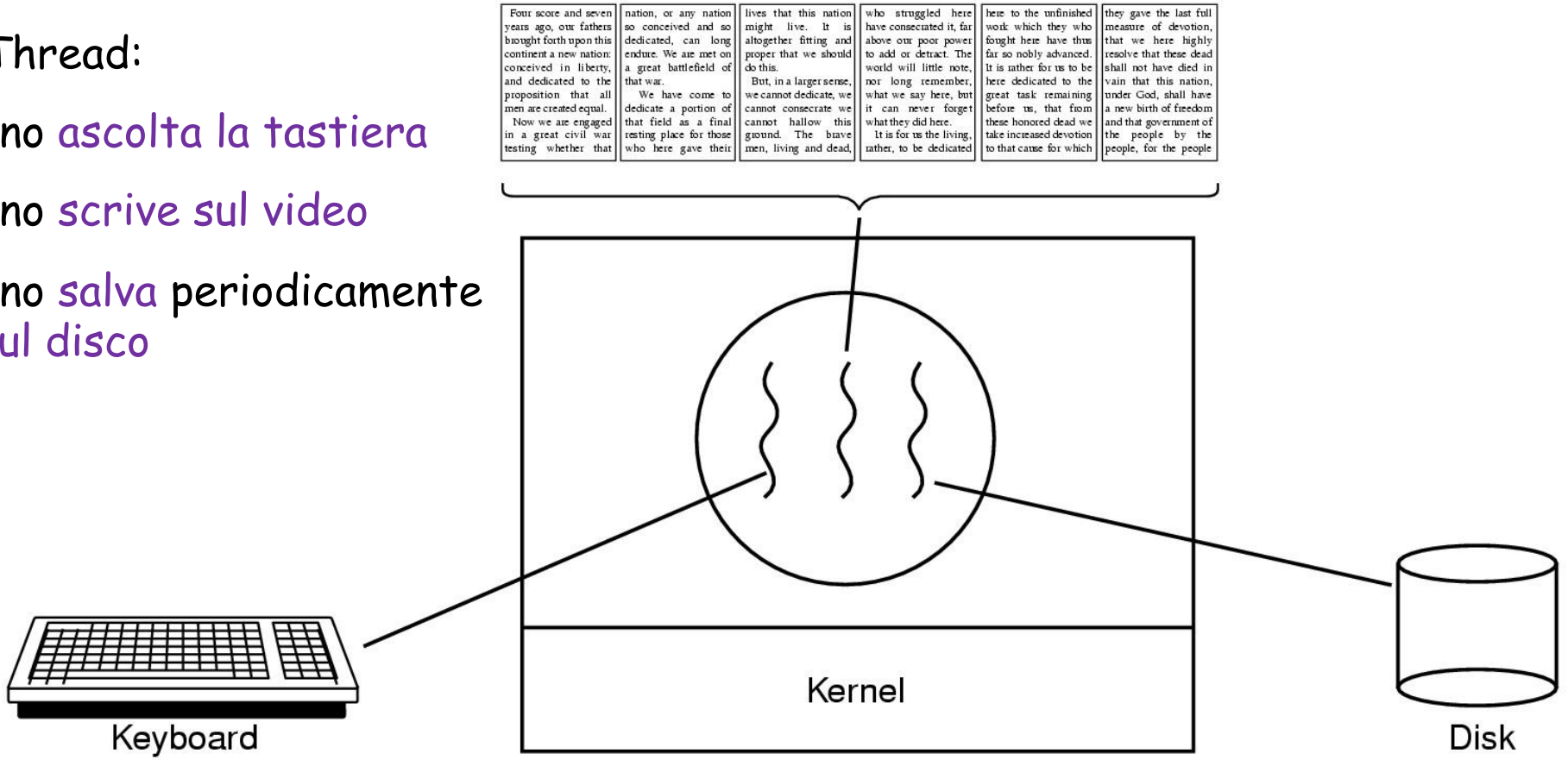
Inoltre, la separazione degli spazi di indirizzamento, rende complesso l'uso dei processi stessi nel caso di interazioni basate su accesso a strutture comuni

- Es. applicazioni in tempo reale per il controllo di impianti fisici
  - Strutture dati comuni a varie attività rappresentano lo stato complessivo dell'impianto da controllare
- Es. un programma di elaborazione di testi contiene varie attività concorrenti che operano sugli stessi dati
  - Lettura dei dati immessi da tastiera, scrittura su video, salvataggio periodico su disco, correzione ortografica e grammaticale, …

Un word processor multithread

3 Thread:

- uno ascolta la tastiera
- uno scrive sul video
- uno salva periodicamente sul disco



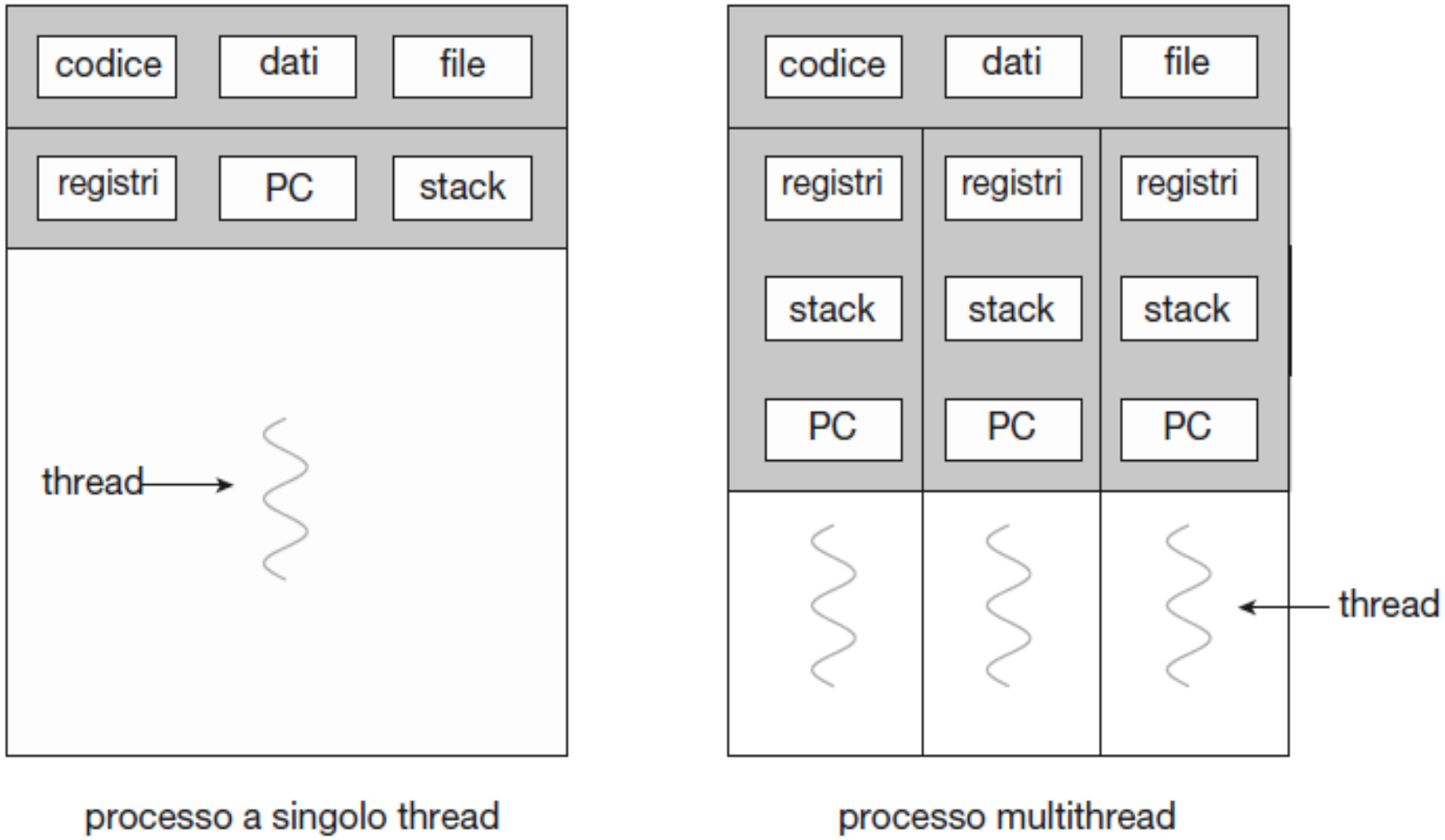
Processi & Thread

- Modello concorrente e processi
- Scheduling dei processi
- Operazioni sui processi
- Comunicazione tra processi (IPC)
- Comunicazione nei sistemi client-server
- Processi e thread
- Concetto di thread
- Programmazione multicore
- Modelli di multithreading
- Librerie di thread (cenni)

Concetto di thread

- Un thread rappresenta un flusso di esecuzione all'interno di un processo
- I thread di uno stesso processo risiedono nello stesso spazio di indirizzamento e condividono
  - sezione di codice
  - sezione di dati
  - risorse del SO (es. file aperti, gestori di segnali)
- Es. se un thread apre/crea un file con determinati diritti di accesso, tutti gli altri thread del processo avranno gli stessi diritti sul file
- Un thread è caratterizzato da
  - un identificatore
  - uno stato di esecuzione (pronto, in attesa, in esecuzione)
  - uno spazio di memoria per le variabili locali
  - un contesto, rappresentato dai valori del Program Counter e dei registri della CPU utilizzati dal thread
  - uno stack
- Ad ogni thread è associato un Thread Control Block (TCB), una struttura dati del kernel (se i thread sono gestiti dal SO)

Processi single-thread e multithread



processo a singolo thread

processo multithread

Processi e thread

Per-process items	Per-thread items
Address space	Program counter
Global variables	Registers
Open files	Stack
Child processes	State
Pending alarms	
Signals and signal handlers	
Accounting information	

- Gli elementi nella colonna di sinistra sono condivisi da tutti i thread di uno stesso processo
- Quelli della colonna di destra sono replicati per ogni thread

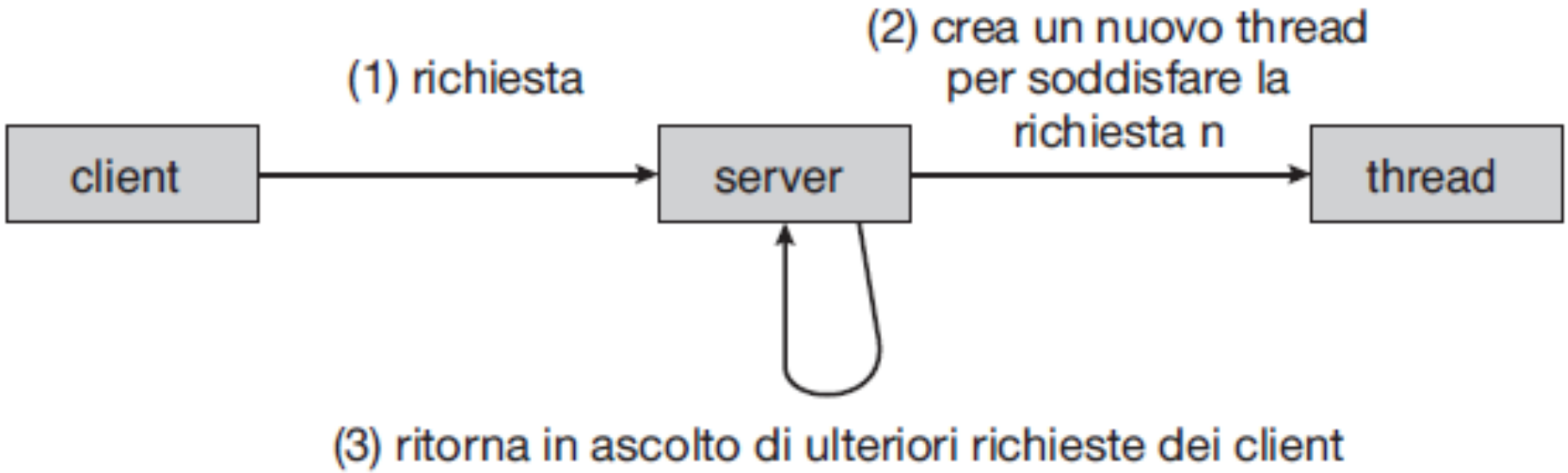
Processi single-thread e multithread

Possono convivere in uno stesso sistema

- Processi single-thread
  - Operano indipendentemente con un proprio program counter, stack pointer, spazio di indirizzamento ed insieme di file aperti
  - Organizzazione conveniente per svolgere task non correlati
- Processi multithread
  - Grazie alla condivisione, impiegano meno risorse (memoria, file aperti, scheduling della CPU)
  - Organizzazione conveniente per svolgere task correlati

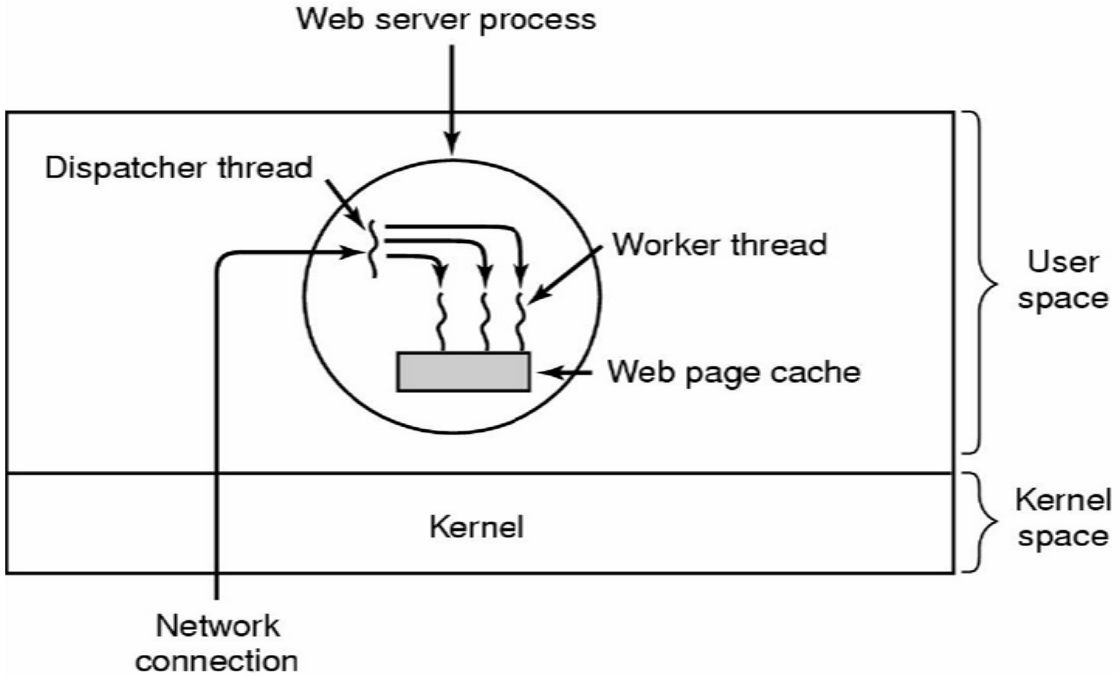
Architettura di un server multithread

La maggior parte delle applicazioni per i moderni computer è multithread



Esempio: server web multithread

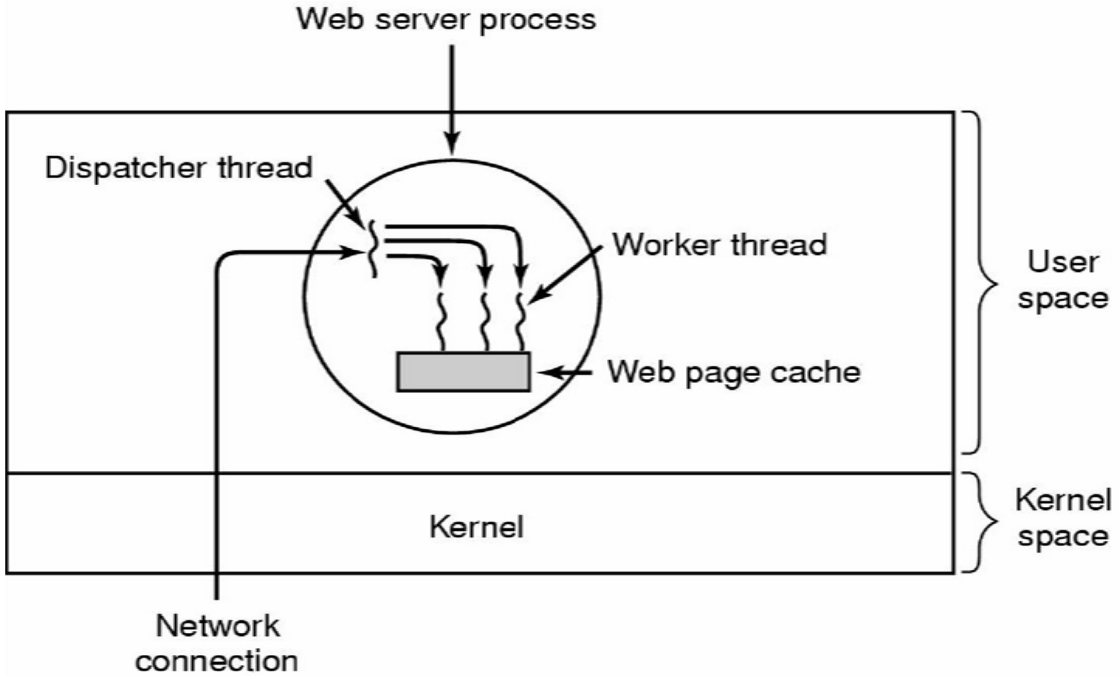
- Obiettivo: servire richieste multiple di pagine dai client
- Il server può essere decomposto in due tipi di thread:
  - dispatcher
  - worker (in numero fissato)
- Il *dispatcher* riceve le richieste e le assegna a thread worker liberi, i quali vengono svegliati affinché svolgano il lavoro



- I *worker*, quando vengono svegliati:
  1. controllano se la pagina è in cache (condivisa tra i thread)
  2. se non è in cache la leggono dal disco
  3. la forniscono al client
- Se deve essere eseguita una lettura da disco, il worker si blocca e può essere messo in esecuzione il worker di un'altra richiesta

Esempio: server web multithread

- Obiettivo: servire richieste multiple di pagine dai client
- Il server può essere decomposto in due tipi di thread:
  - dispatcher
  - worker (in numero fissato)



Alcuni vantaggi

- Il servizio di una richiesta tramite un thread esistente è più rapido che tramite la creazione di un nuovo thread
- Riusare i thread liberi è più efficiente che far terminare thread non più in uso e ricrearne di nuovi all'occorrenza
- Esiste sempre un numero limitato di thread

Vantaggi dell'uso dei thread

- Prontezza di risposta
  - In un processo a thread multipli, mentre un thread è bloccato in attesa di un evento, un altro thread dello stesso processo può essere eseguito
- Condivisione di risorse
  - I thread condividono memoria e risorse del processo a cui appartengono, consentendogli di avere diversi thread di attività all'interno dello stesso spazio di indirizzamento
- Prestazioni (Economia)
  - L'alto grado di condivisione fa sì che il context switch tra thread sia più veloce di quello tra processi (perché non richiede operazioni di gestione della memoria), così come pure creazione e terminazione
- Scalabilità
  - Nelle architetture multi processore/core i thread si possono eseguire in parallelo su processori/core distinti

Processi & Thread

- Modello concorrente e processi
- Scheduling dei processi
- Operazioni sui processi
- Comunicazione tra processi (IPC)
- Comunicazione nei sistemi client-server
- Processi e thread
- Concetto di thread
- Programmazione multicore
- Modelli di multithreading
- Librerie di thread (cenni)

Architetture multicore

- L'architettura dei sistemi di elaborazione si è evoluta in risposta alla necessità di disporre di una maggiore potenza di calcolo
  - Inizialmente, i sistemi a CPU singola si sono evoluti in sistemi multi-CPU
  - Più di recente, sono apparsi sistemi in cui più unità di elaborazione (core) sono montati sullo stesso chip e ogni core appare al SO come una CPU separata
- Sia che i core appartengano allo stesso chip o a più chip, noi chiameremo questi sistemi multicore

Evoluzione del modello dei processi nei SO

- I primi SO supportavano solo un singolo processo (*single-tasking*)
- Nella seconda metà degli anni '70, molti SO erano già multiprogrammati e *multitasking*
  - Supportavano processi multipli, ma ancora single-threading
- Agli inizi degli anni '80, alcuni SO per personal computer usavano ancora single-tasking (es. MS-DOS)
- Negli anni '90, molti SO passarono al *multithreading*
- Oggi, quasi tutti i computer possono eseguire thread multipli simultaneamente, con anche un supporto HW da parte della CPU
  - Ogni processore tipicamente contiene più core
  - Ogni core è capace di eseguire thread
  - Ci sono più thread che core
  - In ogni istante, diversi thread sono in attesa di un qualche evento

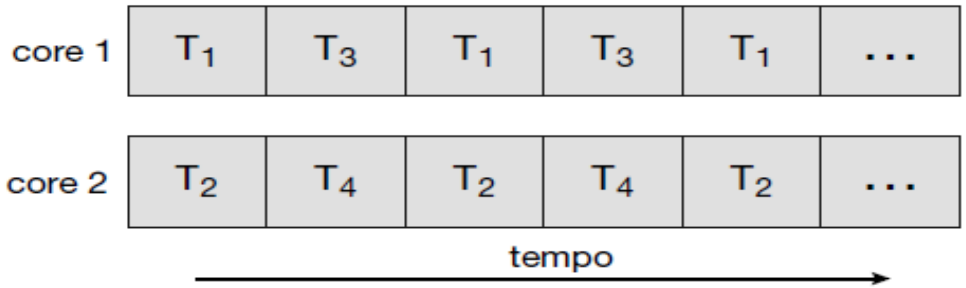
Programmazione multicore

- La programmazione multithread su sistemi multicore offre l'opportunità di un utilizzo più efficiente di questi core e di migliorare la concorrenza
- Su un sistema multicore, "esecuzione concorrente" significa che i thread possono essere eseguiti effettivamente in parallelo, dal momento che il sistema può assegnare thread diversi a ciascun core

Programmazione multicore

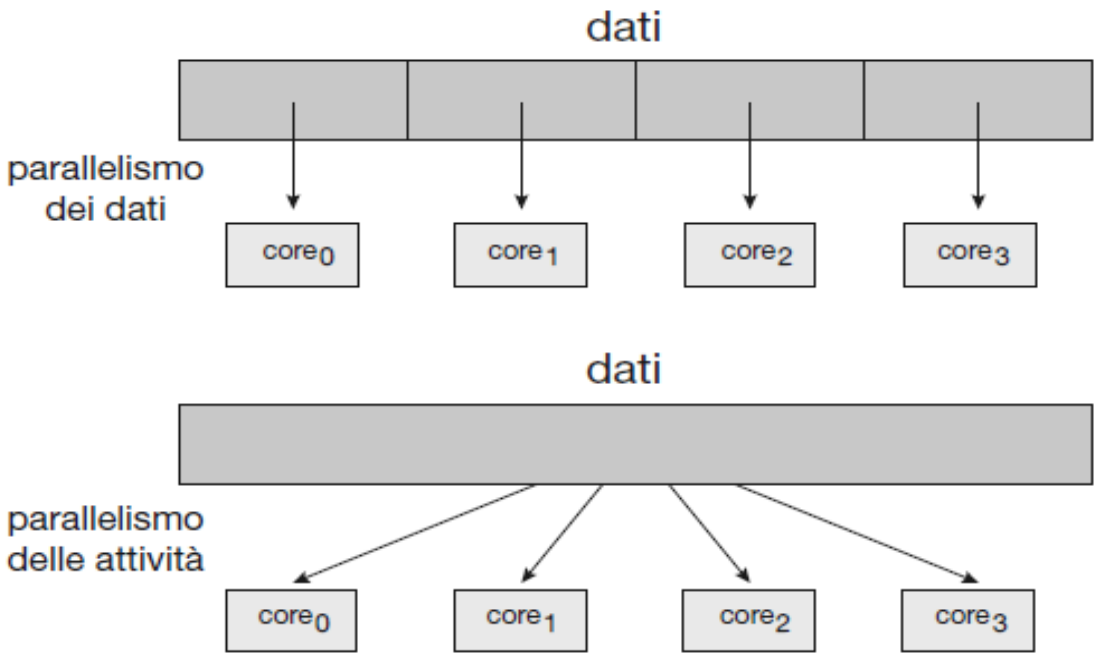
esecuzione concorrente
in un sistema con un singolo core → supporta più thread permettendo a ciascuno di progredire nell'esecuzione (*interleaving*)

esecuzione parallela
in un sistema multicore → può eseguire simultaneamente più di un thread (*overlapping*)



Tipi di parallelismo

- Parallelismo dei dati: distribuzione di sottoinsiemi dei dati su più core di elaborazione ed esecuzione della stessa attività (thread) su ogni core
  - Es. somma dei valori contenuti in un vettore
- Parallelismo delle attività: distribuzione su più core di attività (e non di dati), ciascuna esegue una operazione distinta sugli stessi dati o su dati diversi
  - Es. operazioni diverse sugli elementi dello stesso vettore



Sfide della programmazione multicore

- Identificazione dei task: suddividere le applicazioni in task concorrenti e, se possibile, eseguibili in parallelo su più core
- Bilanciamento: equilibrare, per quanto possibile, le attività eseguite dai vari task
- Suddivisione dei dati: suddividere i dati manipolati dai task da utilizzare su core distinti
- Individuazione delle dipendenze dei dati: sincronizzare l'esecuzione dei task in modo da rispettare le dipendenze tra task sulla base dei dati prodotti da uno ed usati da un altro
- Effettuazione di test e debugging: su flussi di esecuzione multipli sono più difficili che nel caso della programmazione a singolo thread

Processi & Thread

- Modello concorrente e processi
- Scheduling dei processi
- Operazioni sui processi
- Comunicazione tra processi (IPC)
- Comunicazione nei sistemi client-server
- Processi e thread
- Concetto di thread
- Programmazione multicore
- Modelli di multithreading
- Librerie di thread (cenni)

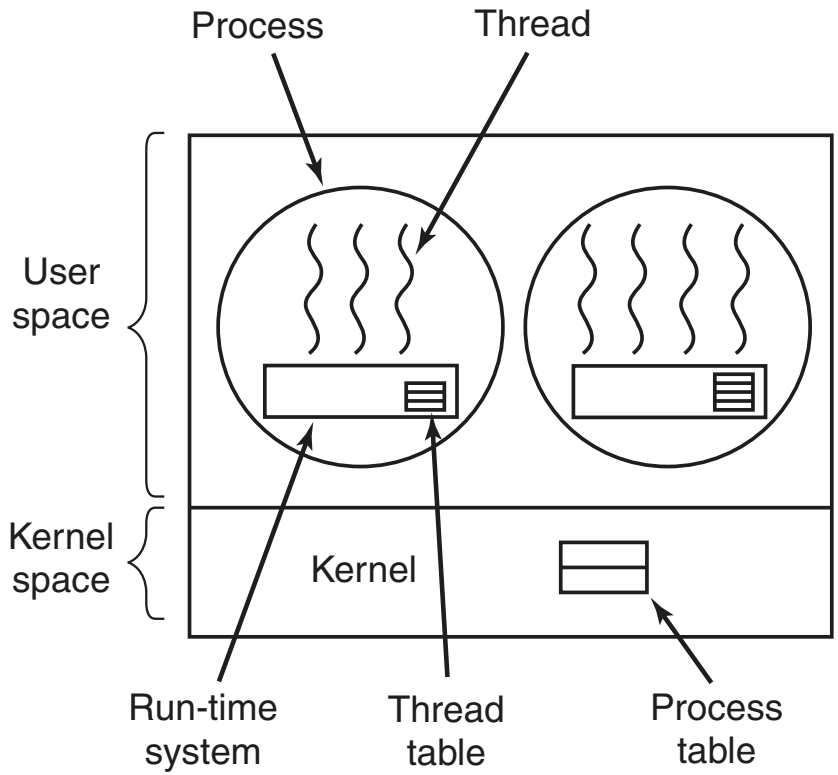
Modelli di multithreading

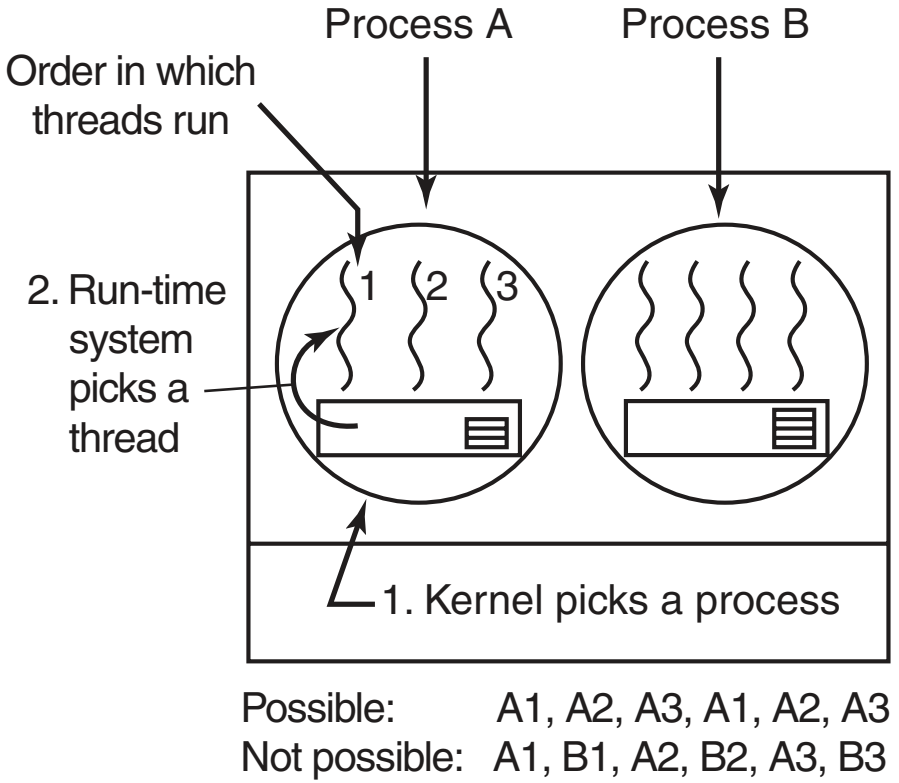
- La gestione dei thread può avvenire
  - a livello utente: i thread sono gestiti sopra il livello del kernel e senza il suo supporto
  - a livello kernel: i thread sono gestiti direttamente dal SO
- Questi approcci differiscono per il ruolo che hanno i processi utente ed il kernel del SO
- Tale differenza ha un impatto significativo su
  - overhead della commutazione di contesto dei thread
  - concorrenza e parallelismo all'interno dei processi

Thread a livello utente

- La gestione dei thread è effettuata tramite una libreria di funzioni a livello utente che è collegata al codice di ogni processo
  - Le funzioni supportano creazione, terminazione, sincronizzazione e scheduling dei thread e sono eseguite in modalità "utente"
- Per ogni processo c'è (nella memoria utente) una tabella dei thread che memorizza lo stato dei singoli thread (PC, SP, registri, etc.)
- Il kernel non è a conoscenza della presenza dei thread a livello utente all'interno di un processo: vede e schedula solo processi
- Quando il thread in esecuzione invoca una funzione di libreria che richiede il verificarsi di un qualche evento, la stessa funzione si occupa dello *scheduling* e seleziona per l'esecuzione un altro thread del processo
  - Se non riesce a trovare un thread del processo che sia *ready*, allora invoca una system call "block me"
  - A questo punto il kernel blocca il processo e ne seleziona un altro
- Il processo sarà poi sbloccato quando un qualche evento attiverà uno dei suoi thread e farà riprendere l'esecuzione della funzione di libreria (la quale eseguirà lo *scheduling* e invierà in esecuzione il thread appena attivato)

Thread a livello utente: scheduling





Thread a livello utente: considerazioni

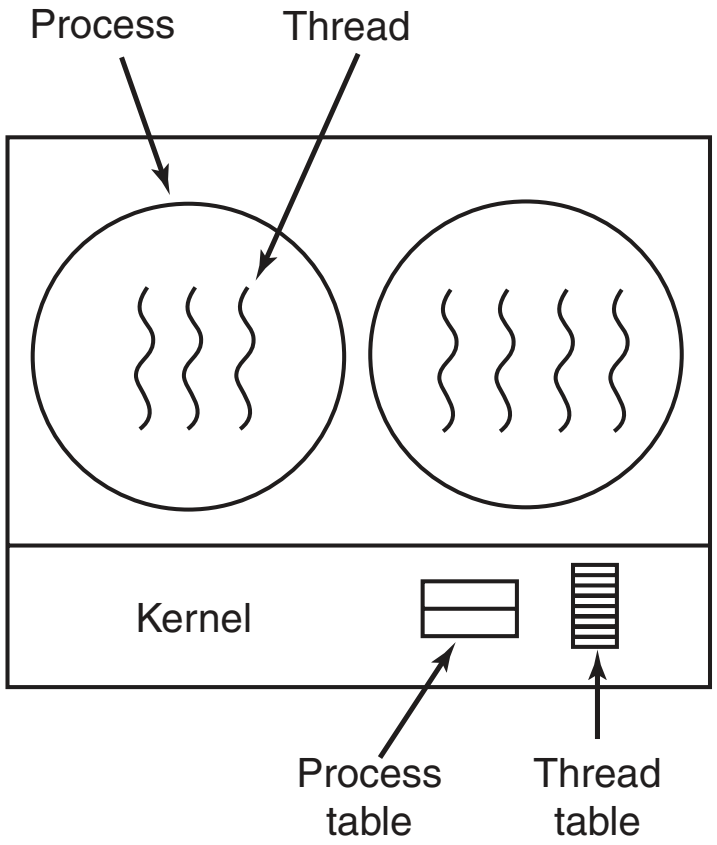
- Vantaggi
  - Efficienza: sincronizzazione e schedulazione sono implementate dalla libreria, ciò evita l'overhead della gestione delle system call per la sincronizzazione dei thread e l'overhead della commutazione potrebbe essere di un ordine di grandezza inferiore a quello tra thread a livello kernel
  - Flessibilità: il programmatore può usare la politica di scheduling dei thread che meglio si adatta alla natura di ciascun processo
  - Portabilità: anche su SO che non supportano direttamente i thread, dato che il SO gestisce processi e ignora la presenza dei thread
- Svantaggi
  - Non c'è scheduling automatico tra i thread quindi
    - Non c'è prelazione dei thread: se un thread non passa il controllo esplicitamente, monopolizza la CPU (all'interno del processo)
    - L'invocazione di una system call bloccante blocca tutti i thread del processo: deve essere sostituita con l'invocazione di una routine di libreria che, eventualmente, blocchi il solo thread invocante (*jacketing*)
  - L'accesso al kernel è sequenziale (per thread dello stesso processo) perché il kernel schedula un processo e la libreria schedula un thread all'interno del processo
    - Non sfrutta sistemi multiprocessore (perchè tutti i thread dello stesso processo devono risiedere sullo stesso processore)
  - Meno utile per processi I/O bound

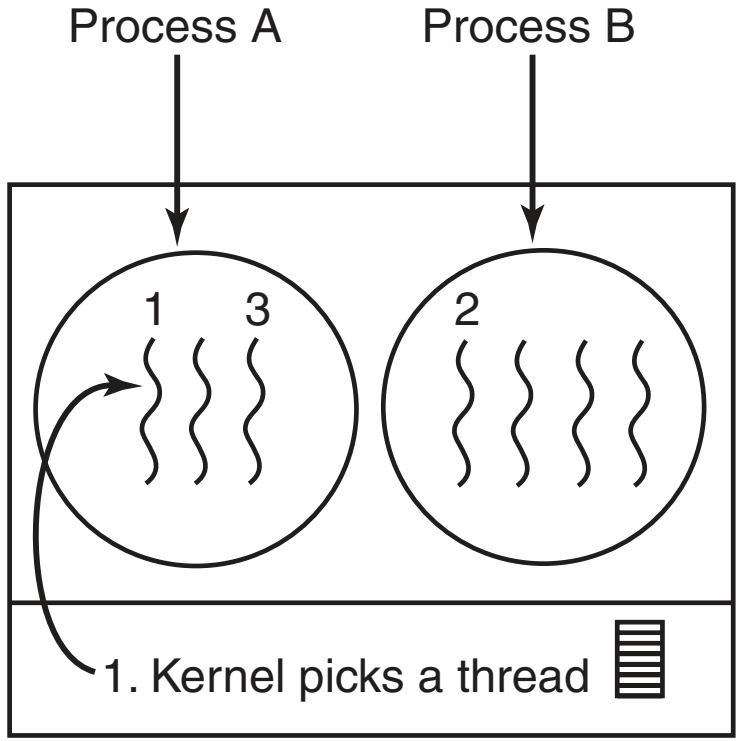
Thread a livello kernel

- La gestione dei thread è effettuata direttamente dal kernel del SO: creazione, terminazione e verifica dello stato di un thread sono effettuate tramite system call
  - È il kernel a supportare i thread con una propria tabella dei thread
- Quando si verifica un evento, il kernel salva lo stato corrente della CPU del thread interrotto nel suo Thread Control Block (TCB)
- Dopo la gestione dell'evento, lo *scheduler* considera i TCB di tutti i thread *ready* e ne seleziona uno per l'esecuzione
- Il *dispatcher* usa il puntatore al PCB contenuto nel TCB del thread selezionato per verificare se il thread selezionato appartiene ad un processo differente da quello a cui appartiene il thread interrotto
  - Nel caso, salva lo stato del processo a cui appartiene il thread interrotto e carica lo stato del processo a cui appartiene il thread selezionato

  Infine manda in esecuzione il thread selezionato
- La commutazione tra thread appartenenti ad uno stesso processo potrebbe essere anche di un ordine di grandezza più veloce della commutazione tra processi

Thread a livello kernel: scheduling





Possible: A1, A2, A3, A1, A2, A3
Also possible: A1, B1, A2, B2, A3, B3

Thread a livello kernel: considerazioni

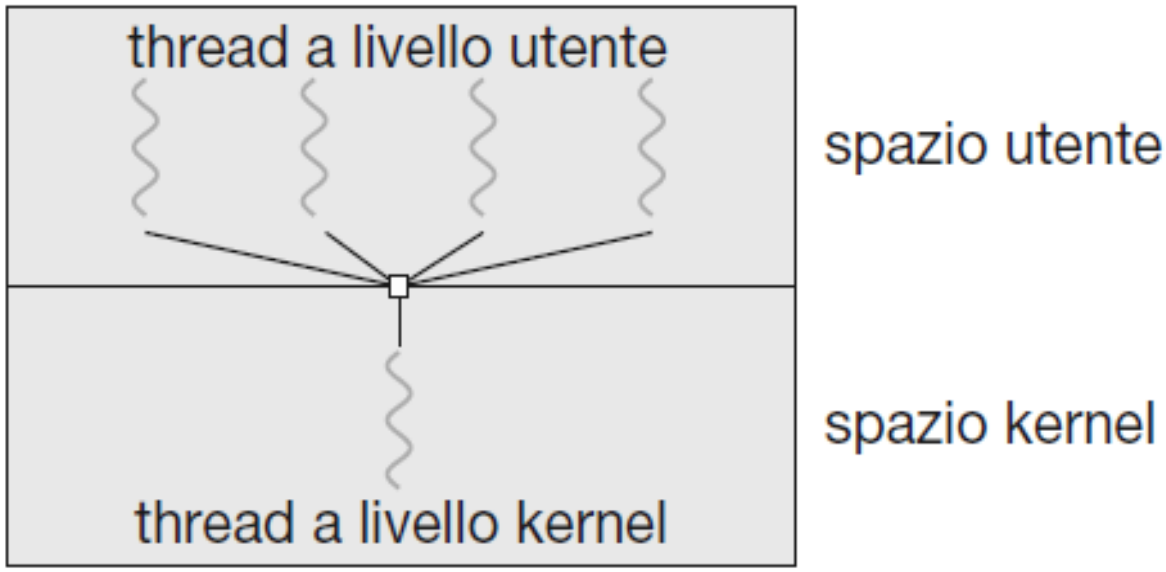
- Vantaggi
  - Scheduling della CPU per thread (non per processo): poiché è il kernel a gestire i thread, se un thread esegue una system call bloccante, il kernel può attivare l'esecuzione di un altro thread dell'applicazione
  - Conveniente per il programmatore: programmare per thread è simile a programmare per processi (un thread a livello kernel è infatti simile ad un processo ma con una quantità inferiore di informazioni di stato)
  - Consente il parallelismo all'interno di un processo: in un sistema multicore i thread di un processo possono essere eseguiti in overlapping
  - Utile per i processi I/O bound
- Svantaggi
  - Flessibilità: la politica di scheduling è stabilita dal kernel
  - Efficienza: bisogna eseguire una system call per ogni operazione sui thread ed ogni commutazione tra thread comporta la gestione di un evento (anche se i thread commutati appartengono allo stesso processo)
  - Portabilità: può richiedere l'aggiunta e/o la riscrittura di system call preesistenti

Modelli di corrispondenza

- Nel caso di presenza di thread ad entrambi i livelli, che è la situazione più commune, deve esistere una associazione tra i thread a livello utente e i thread a livello kernel, perché il kernel assegna la CPU (solo) a thread a livello kernel
- Associazioni differenti sono caratterizzate da caratteristiche diverse riguardo
  - flesibilità ed efficienza
  - concorrenza e parallelismo
- Modelli comuni di tali associazioni sono
  - Molti-a-uno
  - Uno-a-uno
  - Molti-a-molti
  - A due livelli

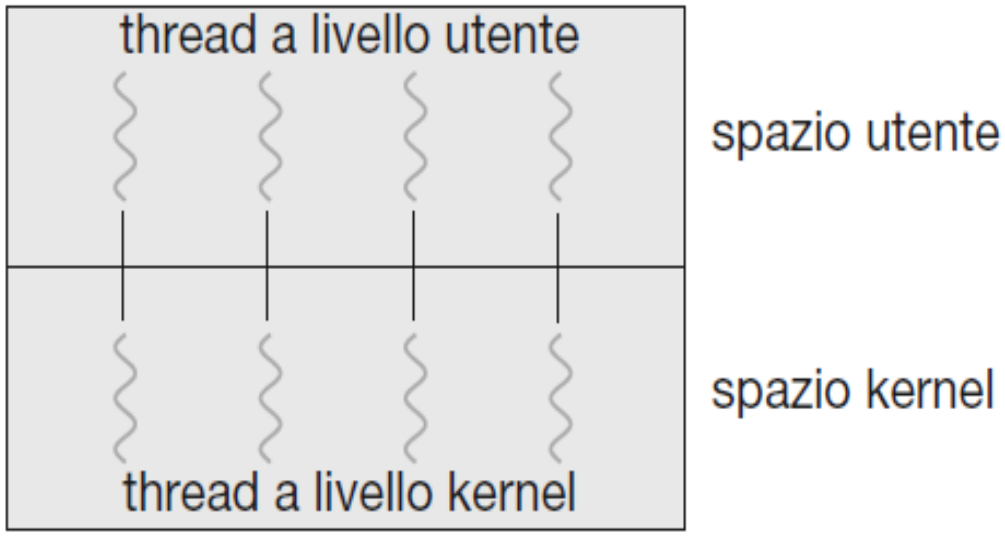
Modello molti-a-uno

- Molti thread a livello utente sono associati ad un singolo thread del kernel
- Modello usato in sistemi che non supportano il multithread a livello kernel
- Gestione efficiente perché effettuata da una libreria a livello utente,
  blocco del processo in caso di blocco di un thread su una system call,
  accesso sequenziale al kernel quindi non sfrutta i il parallelismo sui multicore
- Poco usato per via dell'incapacità di avvantaggiarsi dalla presenza di più core



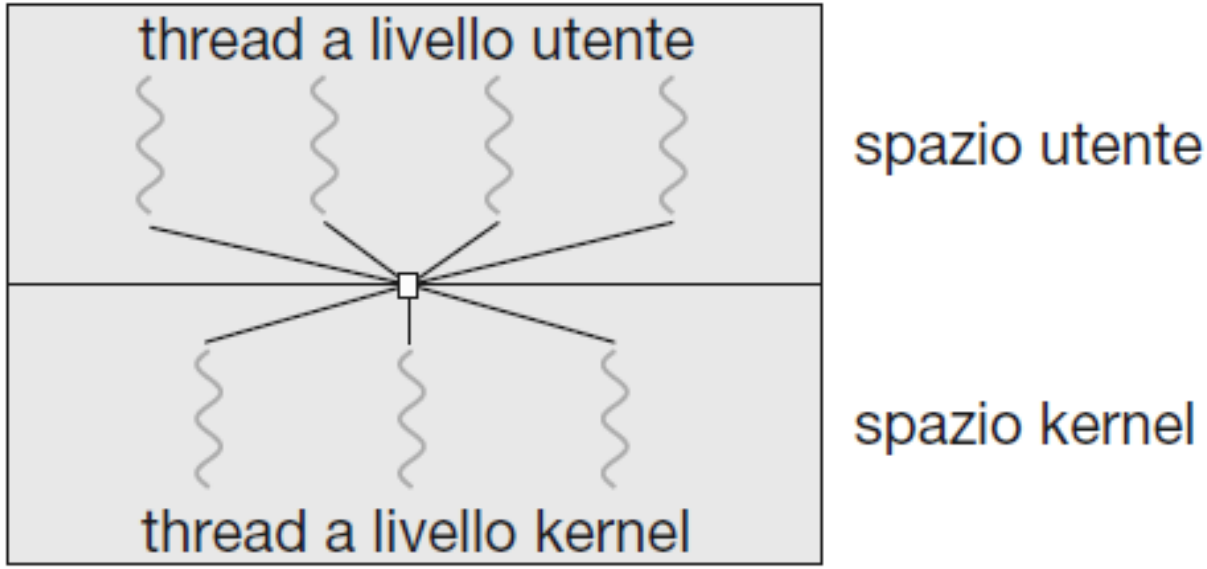
Modello uno-a-uno

- Ogni thread a livello utente è associato ad un thread del kernel
  - Effetto simile ai semplici thread a livello kernel
- Permette un maggiore grado di concorrenza tra thread di uno stesso processo
  - Se un thread si blocca su una system call è possible eseguirne un altro
  - Consente esecuzione dei thread in overlapping
- La creazione di un thread a livello utente implica la creazione del thread corrispondente a livello kernel
  - Un numero elevato di thread del kernel può compromettere le prestazioni di un sistema, perciò molti sistemi limitano il numero massimo di thread
- Utilizzato dalla maggior parte dei SO (tra cui Linux e Windows)



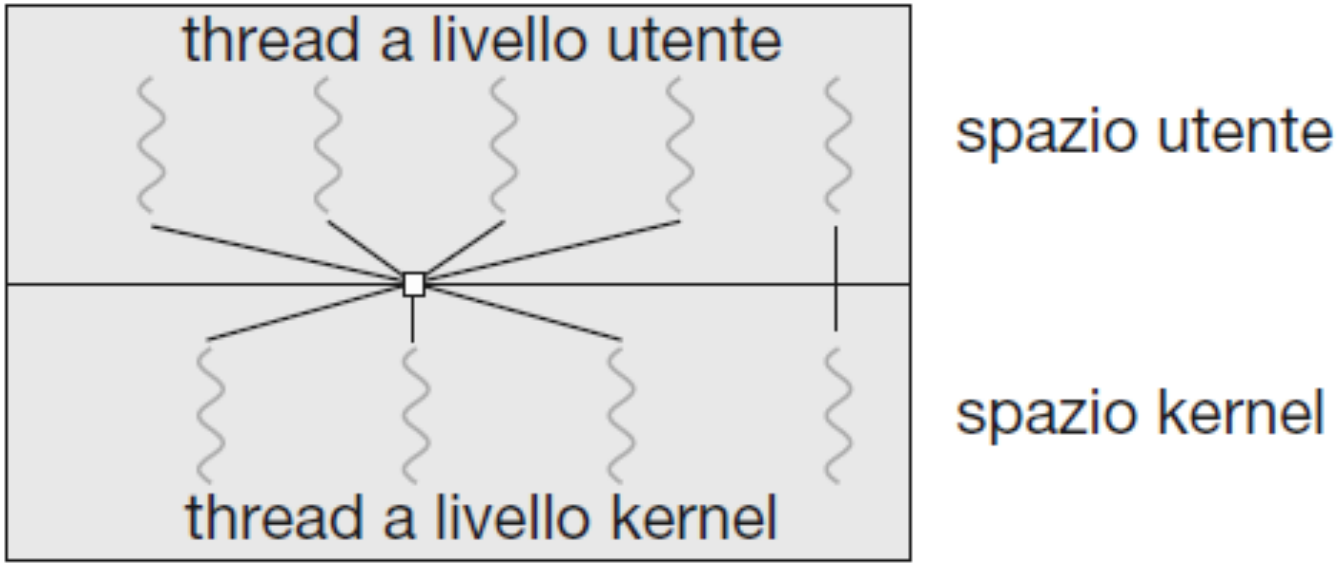
Modello molti-a-molti

- Molti thread a livello utente sono associati a molti thread del kernel
- Flessibile perché i programmatori possono creare liberamente i thread a livello utente che ritengono necessari senza che questo implichi la creazione di altrettanti thread a livello kernel
- Elevata concorrenza e parallelismo dei thread di livello kernel
  - Se un thread si blocca su una system call è possible eseguirne un altro
  - Consente esecuzione dei thread in overlapping
- Implementazione complessa



Modello a due livelli

- Variante del modello molti-a-molti che consente anche associazioni uno-a-uno



Processi & Thread

- Modello concorrente e processi
- Scheduling dei processi
- Operazioni sui processi
- Comunicazione tra processi (IPC)
- Comunicazione nei sistemi client-server
- Processi e thread
- Concetto di thread
- Programmazione multicore
- Modelli di multithreading
- Librerie di thread (cenni)

Librerie di thread

- Una libreria di thread fornisce una API per la creazione e la gestione di thread
- Due metodi principali per implementarla
  - Interamente nello spazio utente, senza supporto del kernel
    - Strutture dati e codice per la libreria sono nello spazio utente
    - Invocare una funzione nell'API della libreria si traduce in una chiamata di funzione locale nello spazio utente e non in una sytem call
  - Supportata direttamente dal kernel del SO
    - Strutture dati e codice per la libreria sono nel kernel
    - Invocare una funzione nell'API della libreria in genere comporta una system call al kernel

Librerie di thread

Le tre librerie di thread più comunemente usate

- Windows: libreria a livello kernel, disponibile solo per i sistemi Windows
- Pthreads: estensione dello standard POSIX, può essere realizzata sia come libreria a livello utente che a livello kernel, disponibile per sistemi compatibili con POSIX come UNIX, Linux e macOS
- threading Java: thread Java eseguiti su qualsiasi sistema che supporti la JVM, solitamente implementata per mezzo di una libreria di thread del sistema ospitante