Capitolo 4: i processi

- Il concetto di processo.
- Schedulazione dei processi.
- Operazioni sui processi.
- Processi cooperanti.
- Comunicazione tra processi.
- La comunicazione nei sistemi client-server (cenni).

Il concetto di processo

- CONVENZIONE: **`job = processo`**
- In maniera informale un processo è definibile come programma in esecuzione. Si tratta dell'unità di lavoro nei moderni time-sharing.
- L'esecuzione di un processo deve progredire in modo sequenziale.
- Il processo comprende una **sezione di codice** ma anche la cosiddetta **attività corrente** rappresentata da:
  - program counter
  - registri di CPU
  - stack (contiene i dati temporanei)
  - sezione dati (contiene le variabili globali)
  - heap (memoria allocata dinamicamente in fase di esecuzione).
- Due o più processi possono essere associati al medesimo programma (sezione di codice) ma saranno considerati come due differenze istanze di esso (differente attività corrente).

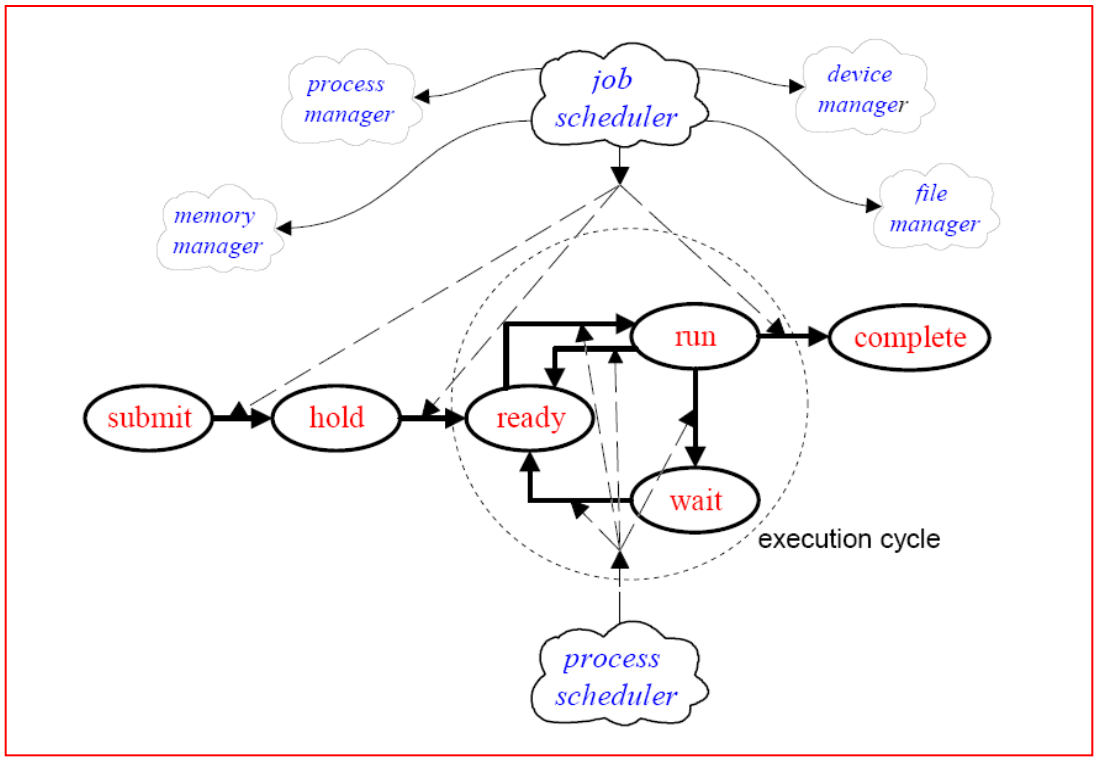
Stato dei processi

- Un processo in esecuzione evolve assumendo *stati* diversi. Esso può trovarsi in uno dei seguenti:
  - **nuovo** *(new):* il processo è stato creato.
  - **in esecuzione** *(running):* le istruzioni vengono eseguite.
  - **in attesa** *(waiting):* il processo sta aspettando il verificarsi di qualche evento:
    - completamento di un'operazione di I/O;
    - attesa di un segnale;
    - assegnazione di dispositivi.
  - **pronto all'esecuzione** *(ready):* il processo è in attesa di essere assegnato ad un processore.
    - tutte le risorse necessarie allo svolgimento dell'attività del processo (eccetto la CPU) risultano essere assegnate ad esso.
  - **terminato** *(terminated):* il processo ha terminato l'esecuzione.

Stato dei processi

- Il sistema operativo produce l'avanzamento dell'esecuzione dei programmi.
- Le richieste sottoposte vengono raggruppate, costituendo la coda relativa allo stato di **submit**.
- Dopo l'analisi delle risorse necessarie a soddisfare le richieste, queste vengono ordinate in relazione ai criteri di priorità (macroscheduling) dal **Job Scheduler**, costituendo la coda relativa allo stato di **hold**.

Diagramma degli stati dei processi



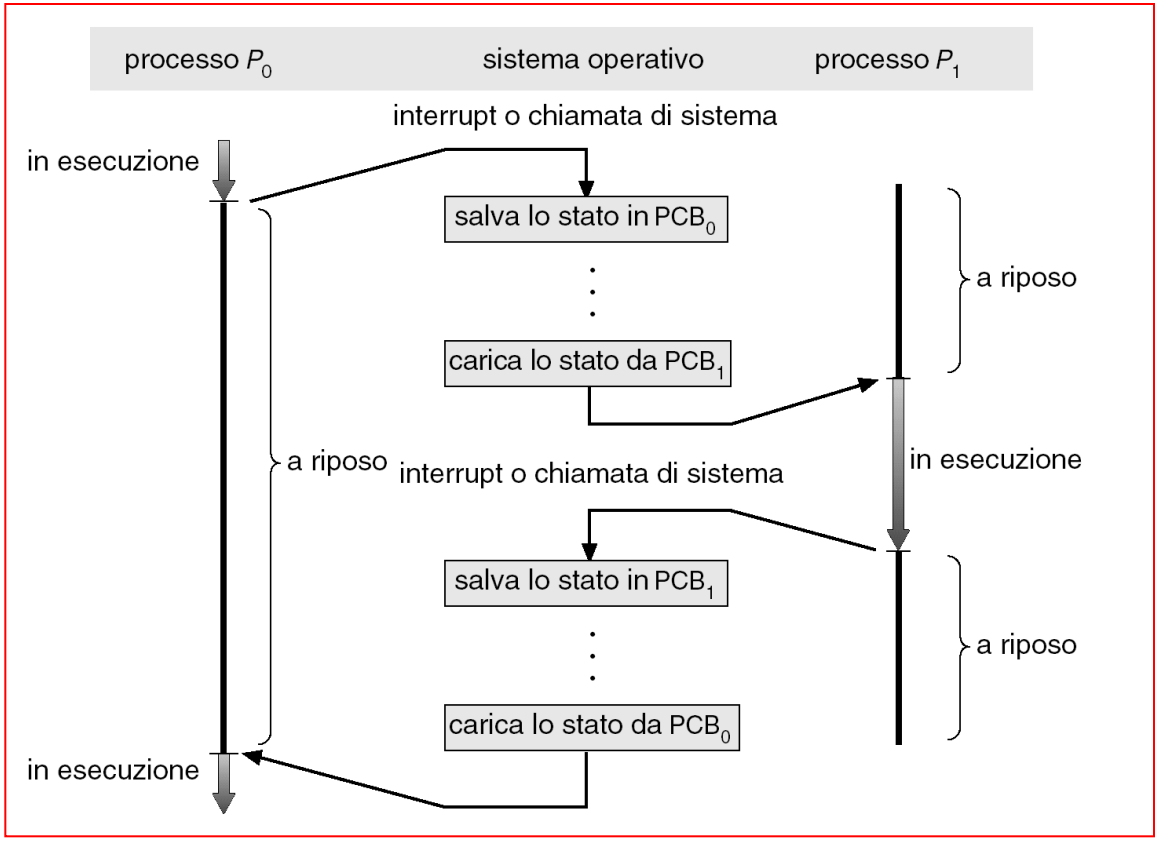
A partire dalla prima richiesta di esecuzione presente nella coda di hold, il Job Scheduler, richiamando i gestori delle varie risorse, verifica la disponibilità delle risorse necessarie e, dopo averne avuto conferma, attiva il ***Process Scheduler***. Questo trasforma la richiesta di esecuzione in processo e costruisce un blocco di informazioni (*Process* o *Task Control Block*), che, progressivamente aggiornate, gli permetteranno di gestire il programma nel *ciclo di esecuzione*.
Inizialmente il task viene immesso nella coda relativa allo **stato di ready**. La coda in questione è organizzata in relazione ai criteri di priorità (microscheduling) con cui il Process Scheduler ordina i vari task che abbisognano della sola CPU per essere eseguiti.
Non appena la CPU si libera (o il rispetto delle priorità lo richiede), viene operato il cambio del contesto computazionale (*context switching*) della CPU e un processo passa nello **stato di run** (ed uno passa in ready).
L'attesa di specifici eventi può costringere il S.O. a far transitare il processo nella coda corrispondente allo **stato di wait**.
Dopo aver ciclato tra gli stati del ciclo di esecuzione, il processo arriva a conclusione e transita nella coda dello **stato di complete**.

Process Control Block (PCB)

Il PCB è una struttura dati del kernel che si occupa di mantenere le informazioni relative ad ogni processo.

- Stato del processo (new, ready, running, waiting, stopped…).
- Program counter.
- Registri della CPU (accumulatori, registri indice, stack pointer, registri general purpose).
- Informazioni per la schedulazione della CPU (priorità del processo, puntatori alle code di schedulazione).
- Informazioni per la gestione della memoria centrale (valore dei registri base e limite, tabelle delle pagine o dei segmenti).
- Informazioni per l'accounting (quantità di CPU utilizzata, limiti di tempo, numero di processo).
- Informazioni sullo stato dell'I/O (lista dei dispositivi di I/O allocati al processo).

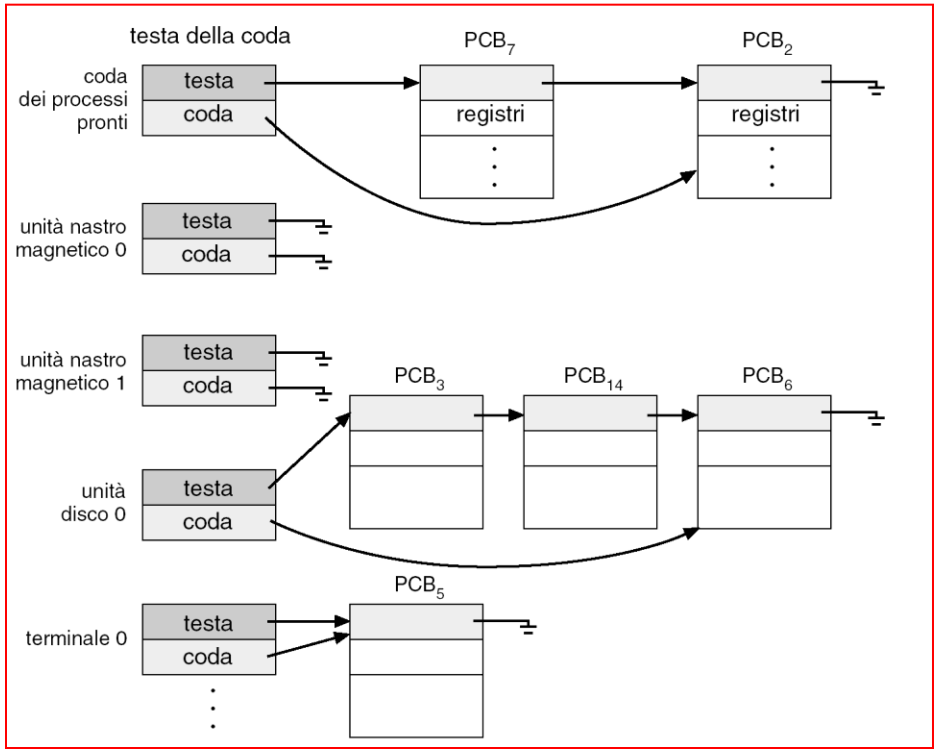
Context switching



- Far passare la CPU da un processo ad un altro richiede il salvataggio dello stato di esecuzione della computazione del vecchio processo ed il caricamento dello stato di esecuzione di quello nuovo.
- Il tempo per il cambio di contesto è puro tempo di gestione del sistema, poichè durante il cambio non vengono compiute operazioni utili per la computazione.
- I tempi per i cambi di contesto dipendono sensibilmente dal supporto hardware.

Code di schedulazione

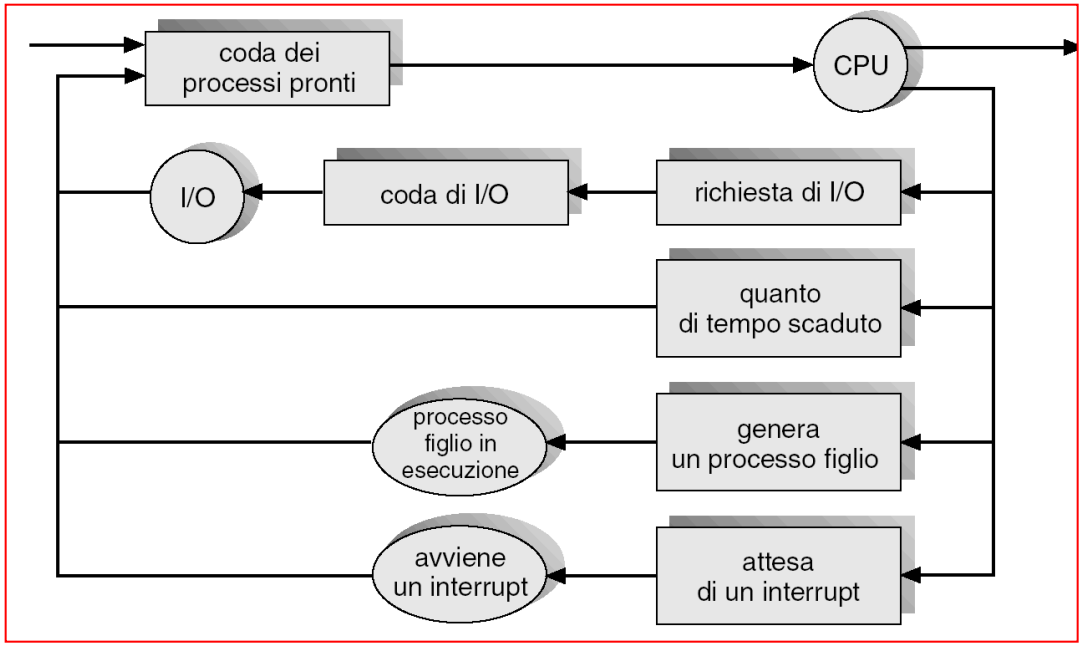
- **Coda di submit *(submit queue)*** – contiene tutti i processi nel sistema.
- **Coda di hold *(hold queue)*** – contiene tutti i processi di cui sia stata effettuata l'analisi delle risorse richieste.
- **Coda dei processi pronti *(ready queue)*** – contiene tutti i processi che risiedono nella memoria centrale, pronti e in attesa di esecuzione.
- **Code delle periferiche di I/O *(device queues)*** – contiene i processi in attesa di una particolare periferica di I/O.
- Il processo si muove fra le varie code.



Code dei processi pronti e delle periferiche

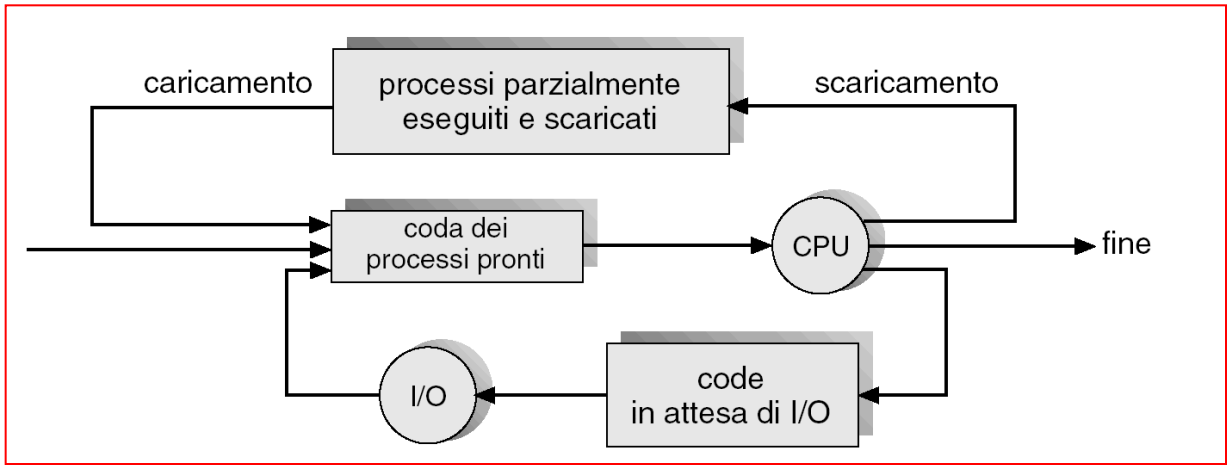
Diagramma delle code

- Si tratta di una rappresentazione comune per lo studio della schedulazione dei processi.
- Si possono individuare:
  - coda dei processi pronti (ready queue)
  - coda delle periferiche di I/O (device queue)
- Un processo selezionato dalla coda di ready per l'utilizzo della CPU **(dispatched)** proseguirà nell'esecuzione fino a che:
  - effettua una richiesta di I/O
  - genera un child process ed attende per la sua terminazione
  - viene forzatamente rimosso a seguito di:
    - conclusione dell'intervallo di tempo (time slice) a sua disposizione
    - arrivo di un interrupt



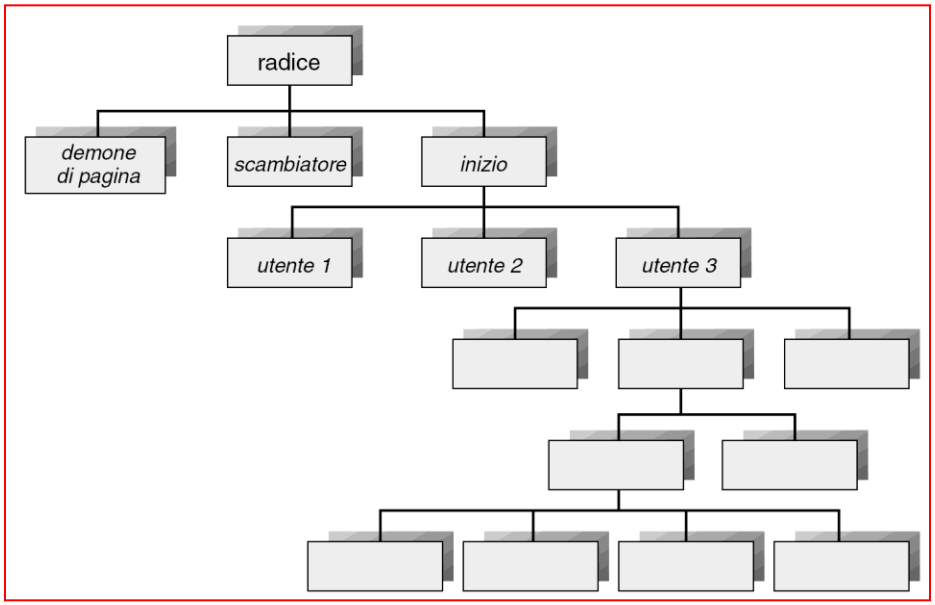
Gli schedulatori

- *Schedulatore a lungo termine (long-term scheduler)* – seleziona quale processo deve essere inserito nella coda dei processi pronti (in memoria).
  - Viene eseguito con una frequenza dell'ordine dei minuti
  - Controlla il livello di multiprogrammazione (numero di processi in memoria centrale)
  - Deve garantire una certa stabilità del grado di multiprogrammazione
  - Bilanciamento dei processi **I/O bound** (molti e brevi utilizzi di CPU) e **CPU bound** (pochi e lunghi utilizzi di CPU)
- *Schedulatore a breve termine (short-term scheduler)* – seleziona quale processo deve essere eseguito e alloca la CPU a uno di essi.
  - Viene eseguito almeno una volta ogni 100 msec
- *Schedulatore a medio termine* (riduce temporaneamente il grado di multiprogrammazione)
  - Rimuove processi dalla memoria centrale (e quindi dalla contesa attiva per la CPU) per poi reintrodurli
  - Utilizza lo **swapping** tra memoria centrale e memoria di massa



Creazione dei processi (1/2)

- Molti O.S. offrono la possibilità di creare e terminare dinamicamente I processi.
- Durante la sua esecuzione, un *processo padre* crea *processi figli*, i quali a loro volta creano altri processi formando un *albero* di discendenze.
- Condivisione delle risorse:
  - padre e figli condividono tutte le risorse;
  - i figli condividono un sottoinsieme delle risorse del padre;
  - padre e figli non condividono risorse (l'O.S. alloca nuove risorse per ciascun processo figlio).
- Esecuzione:
  - il padre continua l'esecuzione in modo concorrente ai figli;
  - il padre attende finchè alcuni o tutti i suoi figli sono terminati.



Albero di processi su un sistema UNIX

Creazione dei processi (2/2)

```
#include <stdio.h>
#include <unistd.h>

int main(int argc, char *argv[])
{
int pid;

   /* genera un altro processo */
   pid = fork();

   if (pid < 0) { /* si è verificato un errore */
      fprintf(stderr, "Fork fallita");
      exit(-1);
   }
   else if (pid == 0) { /*  processo figlio */
      execlp("/bin/ls","ls",NULL);
   }
   else { /* processo padre */
      /* il processo padre attenderà il completamento del figlio */
      wait(NULL);
      printf("Figlio terminato");
      exit(0);
   }
}
```

- Spazio di indirizzamento:
  - il figlio è un duplicato del padre;
  - il figlio ha un nuovo programma caricato nel proprio spazio di indirizzamento.
- Esempi in UNIX
  - la chiamata di sistema *fork* crea un nuovo processo;
  - la chiamata di sistema *exec*, usata dopo la fork, carica un nuovo programma nello spazio di memoria del processo che la esegue.

Terminazione dei processi

- Il processo esegue la sua ultima istruzione e chiede al sistema operativo di rimuoverlo dal sistema tramite la chiamata *exit*.
  - Il figlio restituisce un valore di stato al padre (che attende la terminazione tramite la chiamata *wait*).
  - Le risorse del processo sono deallocate dal sistema operativo.
- Il padre può terminare l'esecuzione di uno dei suoi figli per varie ragioni:
  - Il figlio ha ecceduto nell'uso di una risorsa a lui allocata.
  - Il compito assegnato al figlio non è più necessario.
  - Se il padre sta terminando:
    - alcuni sistemi operativi non permettono ad un figlio di proseguire.
      - Tutti i figli terminano – terminazione a cascata (*cascading termination).*
    - alcuni altri sistemi operativi consentono l'"adozione" dei processi figli da parte di *init.*

Processi cooperanti

- Un *processo* **indipendente** non può influenzare o essere influenzato dagli altri processi in esecuzione nel sistema.
- Un *processo* **cooperante** può influenzare o essere influenzato da altri processi in esecuzione nel sistema.
- Vantaggi della cooperazione:
  - Condivisione delle informazioni.
    - Più utenti possono essere interessati alle stesse porzioni di informazioni
  - Velocizzazione della computazione.
    - Suddivisione di un job in sotto-attività per una esecuzione in parallelo (in caso di multiplazione di una o più risorse).
  - Modularità.
  - Convenienza.
    - Un singolo utente può lavorare su più attività contemporaneamente

Produttore-consumatore

- Si tratta di un paradigma molto comune nel caso di processi cooperanti: un processo *produttore* genera informazioni che sono utilizzate da un processo *consumatore*.
  - Es.: compilatore → assemblatore → loader
  - La concorrenza implica la presenza di memorie tampone riempite dal produttore e svuotate dal consumatore
    - *Buffer illimitato (unbounded-buffer):* non c'è un limite teorico alla dimensione del buffer.
      - Il consumer può dover attendere nuovi oggetti
    - *Buffer limitato (bounded-buffer):* la dimensione del buffer è fissata.
      - Il consumer deve attendere se il buffer è vuoto, mentre il producer deve attendere se il buffer è pieno
  - Il paradigma producer-consumer implica l'adozione di strategie di sincronizzazione tra i processi coinvolti:
    - evita il consumo di oggetti non ancora prodotti

Inter Process Communication

- Meccanismo per consentire ai processi di comunicare e sincronizzare le loro azioni.
- Scambio di messaggi – un processo comunica con un altro senza ricorrere a dati condivisi (non si condivide lo spazio di indirizzi).
- La IPC fornisce due operazioni:
  - ***send*** (messaggio) – dimensione del messaggio fissa o variabile.
  - ***receive*** (messaggio).
- Se i processi *P* e *Q* vogliono comunicare, devono:
  - stabilire un canale di comunicazione tra di loro;
  - scambiare messaggi mediante *send/receive*.
- Implementazione di un canale di comunicazione:
  - fisica (come memoria condivisa, hardware bus, rete),
  - logica.

Comunicazione diretta

- I processi devono conoscere esplicitamente il nome del destinatario o del mittente della comunicazione:
  - INDIRIZZAMENTO SIMMETRICO
    - **send** (*P, messaggio*) – manda un messaggio al processo P;
    - **receive** (*Q, messaggio*) – riceve un messaggio dal processo Q.
  - INDIRIZZAMENTO ASIMMETRICO
    - **send** (*P, messaggio*) – manda un messaggio al processo P;
    - **receive** (*id, messaggio*) – riceve un messaggio da un processo qualunque (la variabile *id* conterrà di volta in volta il nome del processo con cui è stata instaurata una comunicazione).
- Proprietà di un canale di comunicazione:
  - Le connessioni sono stabilite automaticamente.
  - Una connessione è associata esattamente a due processi.
  - Fra ogni coppia di processi esiste esattamente una connessione.
  - La connessione può essere unidirezionale, ma di norma è bidirezionale.
- Limitata modularità: l'esplicitazione degli identificatori dei processi è meno flessibile rispetto al caso in cui è presente indirezione.

Comunicazione indiretta

- I messaggi sono mandati e ricevuti attraverso mailbox o porte.
  - Ciascuna mailbox ha un identificatore univoco.
  - I processi possono comunicare solo se hanno una mailbox condivisa.
- Il canale di comunicazione ha le seguenti proprietà:
  - viene stabilita una connessione fra due processi solo se entrambi hanno una mailbox condivisa.
  - una connessione può essere associata a più di due processi.
  - fra ogni coppia di processi comunicanti possono esserci più connessioni. Ciascuna di esse corrisponde ad una mailbox.
  - la connessione può essere unidirezionale o bidirezionale.
- Azioni:
  - creare una mailbox;
  - mandare e ricevere messaggi per mezzo della mailbox;
  - cancellare una mailbox.
- Le primitive sono definite come:
  - **send** (*A, messaggio*): manda un messaggio alla mailbox A;
  - **receive** (*A, messaggio*): riceve un messaggio dalla mailbox A.

Condivisione di una mailbox

- P_1, P_2, e P_3 condividono una mailbox A.
  - P_1 invia un messaggio ad A; P_2 e P_3 eseguono una receive da A.
  - Quale processo riceverà il messaggio spedito da P_1?
  - La risposta dipende dallo schema che si sceglie:
    - permettere che una connessione sia associata con al più due processori.
    - Permettere ad un solo processo alla volta di eseguire un'operazione di receive.
    - Permettere al sistema di decidere arbitrariamente quale processo riceverà il messaggio. Il sistema può anche notificare il ricevente al mittente.
- Una mailbox può appartenere:
  - al sistema operativo
    - ha una esistenza indipendente non correlata ad alcun processo
  - ad un processo
    - è parte dello spazio di indirizzi di un processo
    - si distingue tra **proprietario** (può solo riceverne messaggi) e **utente** (può solo inviarvi messaggi)
    - una mailbox ha come UNICO proprietario il processo che la crea
      - non esiste possibilità di confusione tra i destinatari di un messaggio
    - alla terminazione di un processo viene meno la mailbox ad esso associata
      - i processi che inviano messaggi ad una mailbox non più esistente ricevono una notifica di errore dal sistema operativo.

Sincronizzazione

- Il passaggio di messaggi può essere bloccante oppure non bloccante.
- **Invio/ricezione bloccanti:** sono considerati **sincroni.**
  - **Invio bloccante:** il processo che invia viene bloccato finchè il messaggio viene ricevuto dal processo che riceve o dalla mailbox.
  - **Ricezione bloccante:** il ricevente si blocca sin quando un messaggio non è disponibile.
- **Invio/ricezione non bloccanti:** sono considerati **asincroni.**
  - **Invio non bloccante:** il processo che invia manda il messaggio e riprende l'attività.
  - **Ricezione non bloccante:** il ricevente acquisisce un messaggio o valido o nullo.
- In caso di send() e receive() bloccanti si parla di **rendezvous** fra mittente e destinatario.

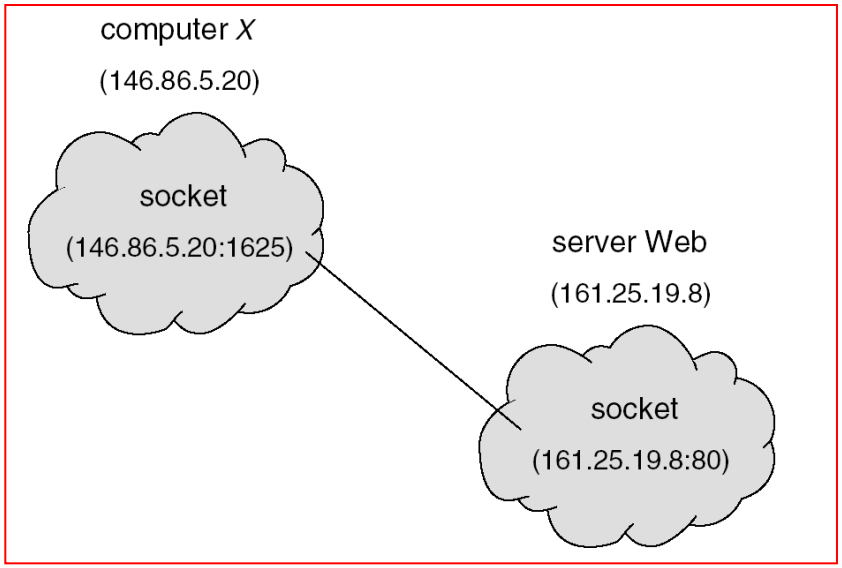
Bufferizzazione

- I messaggi scambiati dai processi in fase di comunicazione risiedono in una coda temporanea
- Esistono tre possibilità per implementare il buffer dei messaggi:
  1. Capacità zero – lunghezza massima della coda: 0 messaggi.
     - La connessione non potrà avere nessun messaggio in attesa nella coda.
     - Il mittente deve bloccarsi finchè il destinatario riceve il messaggio (rendezvous).
  2. Capacità limitata – coda a lunghezza finita di *n* messaggi.
     - Se la coda non è piena, un nuovo messaggio viene inserito nella coda.
       - Il messaggio viene copiato o ne viene mantenuto un puntatore.
     - Il mittente deve bloccarsi solo se la coda di comunicazione è piena, altrimenti può continuare l'esecuzione senza attendere.
  3. Capacità illimitata – coda di lunghezza potenzialmente infinita.
     - Un qualunque numero di messaggi può essere accodato.
     - Il mittente non si blocca mai.

Sistemi client-server

- *La comunicazione nei sistemi client-server può essere implementata secondo gli schemi canonici oppure mediante ulteriori strategie:*
  - Socket.
  - Chiamata di procedure remote (RPC).
  - Invocazione di metodi remoti (RMI).

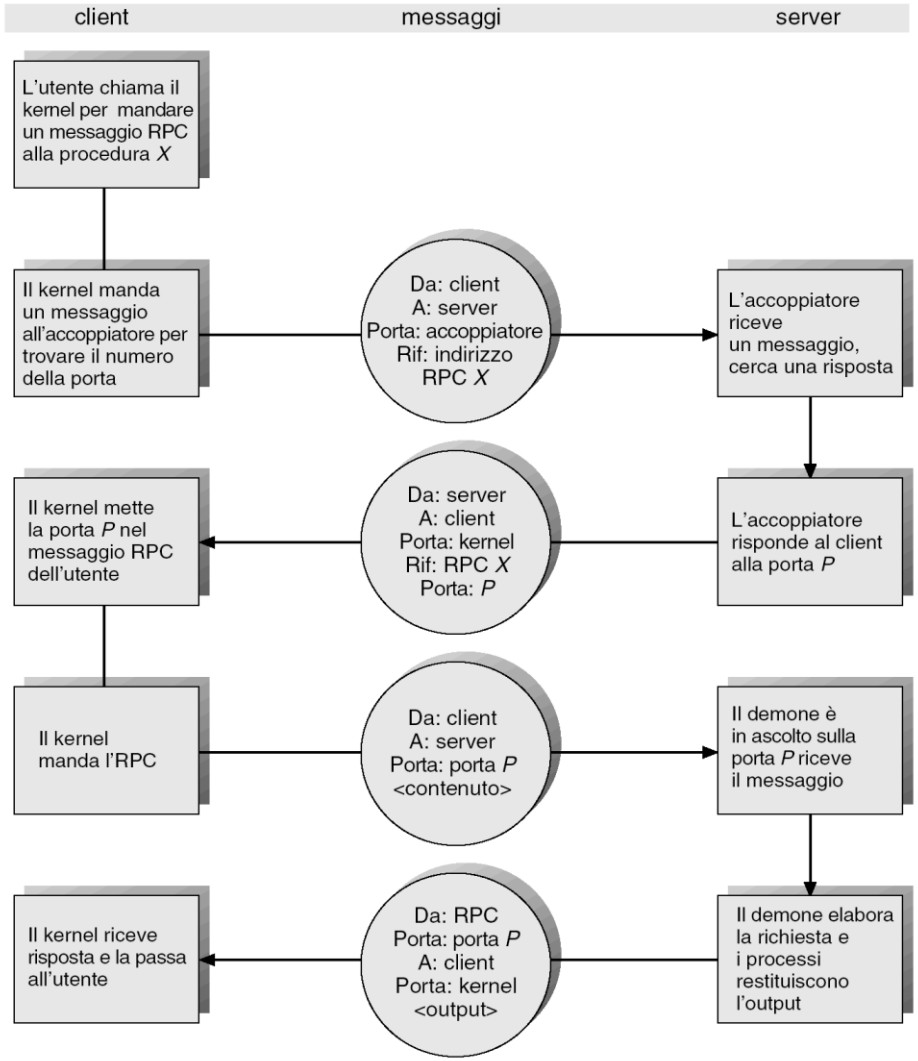
- Un **socket** è definito come *un estremo di un canale di comunicazione.*
  - È identificato dalla concatenazione di un indirizzo IP e un numero di porta.
  - Un server resta in ascolto su di una data porta per l'arrivo di richieste da un client.
    - I server che implementano servizi specifici (telnet, ftp, http) ascoltano sulle cosiddette *well-known ports* (port < 1024).



Remote Procedure Call

- La chiamata di procedura remota (RPC) astrae il meccanismo di chiamata di procedura al caso di sistemi connessi in rete.
- La RPC è una IPC basata su scambio di messaggi ben strutturati (non semplici *data unit*).
- Ciascun messaggio è indirizzato verso un **demone RPC** che ascolta su una porta del sistema remoto. Esso contiene:
  - un identificativo della funzione da eseguire;
  - i parametri da passare.
- L'output della procedura viene restituito al client.
- **Stub** – terminale remoto sul lato client della procedura lato server.
  - Ogni procedura remota mantiene uno stub client.
  - Lo stub del lato client gestisce l'agreement della connessione attraverso la porta sul server e compie la traduzione (**marshalling**) dei parametri secondo un formato comprensibile dal destinatario.
    - Uso della eXternal Data Representation (XDR)
  - Lo stub dal lato server riceve il messaggio, estrae i parametri tradotti ed invoca la procedura sul server.

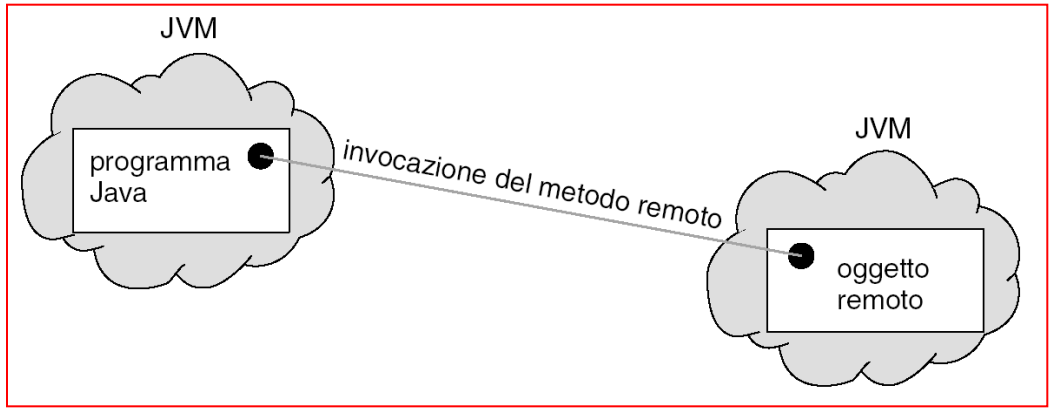
Esecuzione di una RPC



- Una RPC non deve mai comportare un'esecuzione duplicata o nulla.
  - Una marca temporale etichetta tutti I messaggi elaborati da un server che ne mantiene lo STORICO.
    - In tal modo i messaggi ripetuti possono essere riconosciuti e ignorati
  - Un ack viene rispedito al client come conferma di ricezione.
    - Trascorso un timeout senza che il client riceva l'ack di una trasmissione, si porcede ad una nuova RPC.
- Le informazioni di connessione possono essere:
  - predeterminate;
  - ricavate dinamicamente secondo un rendezvous realizzato mediante un apposito demone chiamato **matchmaker** o accoppiatore.

Remote Method Invocation (1/2)

- L'invocazione di metodo remoto (Remote Method Invocation – RMI) è una caratteristica di Java simile alla RPC.
- RMI permette ad un programma Java di invocare metodi relativi oggetti remoti.
- RMI vs. RPC
  - RPC supporta solo programmazione procedurale;
  - RMI rende possibile passare come parametri dei metodi remoti oggetti oltre che strutture dati ordinarie.
- La RMI implementa i metodi remoti usando:
  - **Stub** (componente lato client del metodo remoto, responsabile della creazione di un *parcel* contenente nome del metodo da invocare e parametri su cui è eseguito il marshalling)
  - **Skeleton** (componente lato server responsabile della traduzione dei parametri e dell'invocazione del metodo desiderato)



Remote Method Invocation (2/2)

- Una volta invocato il metodo remoto, lo skeleton restituisce un valore di ritorno (o un'eccezione) al client attraverso un parcel.
- Lo stub esegue la traduzione del valore di ritorno nel formato interno e lo passa al processo client.
- Passaggio dei parametri:
  - se i parametri su cui è stato effettuato il marshalling sono oggetti locali il passaggio avviene per valore;
    - la tecnica adoperata va sotto il nome di serializzazione degli oggetti;
  - in caso contrario esso avviene per riferimento.