Capitolo 3 : i Processi

Table of contents

- Capitolo 3 : i Processi
  - Memoria usata da un programma
  - Esecuzione di un programma
  - Process control block
  - Threads
  - Scheduling dei processi
  - Context Switch
- Operazioni sui processi
  - Creazione di un processo
  - Albero dei processi
  - Creazione di un processo figlio
  - Terminazione di un processo
- Comunicazione tra processi
  - Programmi cooperanti
  - Il comando grep
  - Motivazione per i processi cooperanti
  - Come avvengono le comunicazioni tra processi?
  - Riassumendo
  - Come comunicano due processi - punto di vista logico
  - Implementazione del canale di comunicazione (link)
  - Proprietà del canale di comunicazione (link)
  - Comunicazione indiretta
- Sincronizzazione
  - Buffering
- Sockets (veloce)

Un sistema operativo esegue una varietà di programmi, i programmi **in esecuzione** vengono detti **processi**. C'è una distinzione tra programma e processo:

- **Programma** : entità STATICA
- **Processo** : entità DINAMICA

Memoria usata da un programma

Quando il programma va in esecuzione, viene eseguito **sequenzialmente** istruzione per istruzione, ed ha uno spazio indirizzi a disposizione, suddiviso in:

- Text Section, ovvero il codice del programma
- Program Counter
- Stack, che contiene data temporanea
- Data section, che contiene le variabili globali
- Heap, che contiene memoria allocata dinamicamente durante il tempo di esecuzione.

Esecuzione di un programma

Stato NEW

Lo stato "new" è presente in sistemi che hanno capacità di **batching**, quindi non è presente in sistemi come windows. In questo caso, il SO prende atto della presenza di un programma che deve essere eseguito, situato in una **coda**. Il programma ci rimane finché non viene **ammesso**, e quindi si sposta allo stato ready.

Stato READY

Quando mandiamo in esecuzione un programma, questo riceve dal SO **spazio in memoria**, viene inserito nella lista dei programmi che **utilizzano** la CPU. Questo stato viene chiamato **READY**, proprio perché il programma è pronto per essere eseguito. Questo vuol dire che è presente in memoria, ma non ha ancora ricevuto la CPU per andare in esecuzione a tutti gli effetti.

OPERAZIONE DI DISPATCH

In un sistema tradizionale, dotato di di un'unica CPU, la gran parte dei processi sono nello stato Ready, mentre è presente un unico processo in esecuzione.
Prima o poi, il SO darà la possibilità ad un altro processo (nello stato ready) di essere eseguito.
Questa operazione viene chiamata **dispatch**, quindi lo **scheduler** è quella parte del SO che decide qual è il prossimo processo da mandare in esecuzione.

Stato RUNNING

In questo stato il processo in quell'istante è in **esecuzione**. Esso resta in esecuzione finché non avviene una delle seguenti cose:

1. Il processo termina volontariamente, ovvero esegue una syscall Exit, quindi si sposta nello stato **exit**.
2. Arriva **un'interrupt**, che potrebbe provenire dalle periferiche, come ad esempio la pressione di un tasto da parte dell'utente, oppure un Interrupt di timer, ovvero quando il processo è stato troppo tempo in esecuzione e quindi ne viene scelto un altro per l'esecuzione.\
   Un'altra possibilità di Interrupt potrebbe essere quella dell'attesa di input da parte dell'utente, quindi il processo **perde la CPU**, e va nello stato di **attesa**. Il processo puo uscire dallo stato di attesa in due casi:
3. Viene completata l'operazione di I/O
4. L'eventuale timer che lo ha fatto spostare nello stato di attesa finisce.

Process control block

Quando un processo è in esecuzione il SO deve prenderne atto, e quindi gestire delle informazioni relative a quel processo. Quindi, siccome il SO ha a disposizione delle informazioni per ogni processo ammesso, si dice, in maniera astratta, che è presente un **blocco di controllo dei processi**.

Anche se queste informazioni non risiedono tutte nello stesso punto, sono **sempre** presenti:

- **Stato del processo** : running - waiting - ecc
- **Program Counter** : posizione della prossima istruzione da eseguire
- **Registri CPU** : contenuti di tutti i registri del processo
- **Informazioni di scheduling CPU** : priorità, puntatori a code di scheduling
- **memoria** : informazioni sulla memoria allocata dal processo
- **Informazioni di accounting** : CPU utilizzata, tempo passato dall'esecuzione, utilizzato nei sistemi a pagamento per tenere traccia del tempo di CPU usato per poi calcolare un pagamento.
- **Informazioni sullo stato di I/O** : lista di files aperti ecc.

Threads

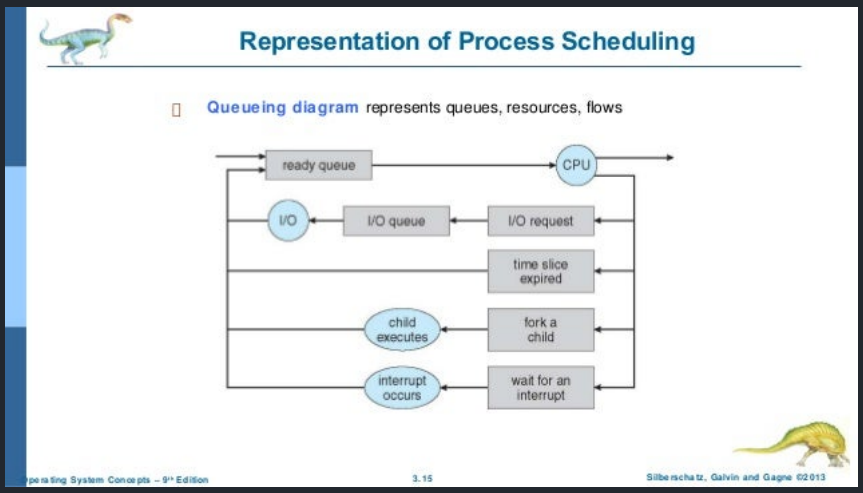
Un processo, ha un program counter, e quindi esegue codice in maniera sequenziale. Negli ultimi anni si è diffuso sempre di più l'utilizzo di tecniche di **threading.** Questo significa che esiste la possibilità di più esecuzioni di codice, **contemporaneamente.**

🏁 1:35 05-06

Scheduling dei processi

Il SO deve Schedulare i processi ed alternarli, in modo tale che la CPU sia quanto più utilizzata possibile. La parte del SO che si occupa di queste operazioni, è proprio il **Process Scheduler**, che seleziona tra i processi in attesa di esecuzione tra una coda.

Questi processi sono posti in apposite strutture dati molto efficienti, in modo da avere sempre a disposizione il prossimo processo da eseguire.



Context Switch

Le operazioni che fermano un processo e ne attivano un altro, sono dette **context switch**; nel momento in cui il SO decide di fermare un processo per attivare il prossimo della coda, salva lo **stato** di **program counter e registri** del processo precedente, e ne manda un altro in esecuzione.

Siccome questo switch richiede del tempo di CPU, deve essere il quanto più veloce ed ottimizzato possibile. Questo tempo dipende sopratutto dall'hardware, perchè più registri ci sono da salvare, maggiore sarà il tempo dell'operazione.

Inoltre, questa parte di codice (del context switch) è scritta in **assembly**, proprio perché è l'unico codice che ci permette di maneggiare i registri, oltre al fatto che l'assembly ci permette di scrivere un codice molto ottimizzato e veloce.

🏁 fine lezione 4

Operazioni sui processi

I processi nascono perchè sono "figli" di un altro processo. Questo vuol dire che ogni processo ha un processo **padre** che lo lancia in esecuzione.

Se prendiamo come esempio un programma a **linea di comando**, sarà lo **shell** che manda in esecuzione il programma da noi scelto.

Creazione di un processo

Come abbiamo detto, il processo **padre** crea un processo **figlio**, in modo da creare un **albero** di processi. Ogni processo ha un **identificativo univoco**, chiamato **pid**.

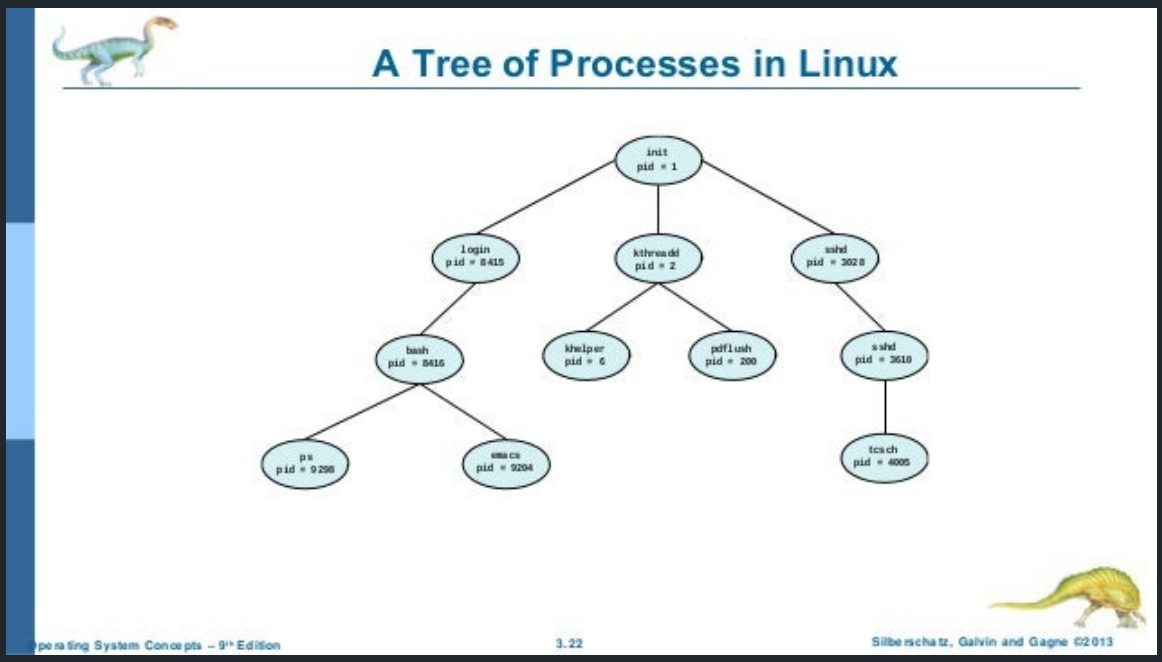
Il fattore "memoria" tra processo padre e figlio può essere strutturato in tre modi:

- Padre e figlio condividono tutte le risorse
- Il figlio condivide un sottoinsieme delle risorse del padre (soluzione più comune)
- Padre e figlio **non** condividono risorse.

Opzioni di esecuzione:

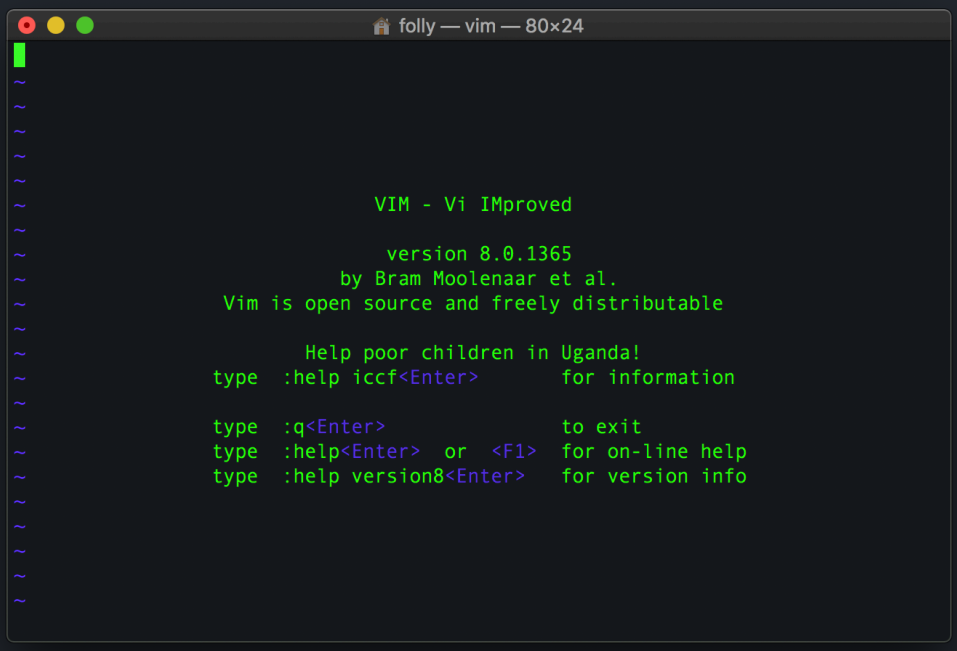
- Il padre va in stop: Una possibilità di esecuzione, quando il padre crea un processo figlio, è quella dove il processo padre si ferma durante l'esecuzione del processo figlio.
  Questo tipo di esecuzione potrebbe dare problemi, perchè il processo padre non può lanciare altri processi figlio, finchè il p
- I due processi vengono eseguiti in modo concorrente.

Albero dei processi



In questa figura si può osservare un albero dei processi di **linux**. Storicamente in **UNIX** viene lanciato al **boot**, dopo il lancio del **kernel**, un processo avente **pid = 1**, perchè a catena fa partire tutto il resto dei processi:

- Tutti i processi che finiscono con " **d** " sono i cosiddetti **deamons** , ovvero un processo eseguito in background. Ad esempio, **sshd** è la versione sicura (criptata) di **shd** , ovvero un programma che serve a connettersi ad un sistema **via rete** (remota), e quindi attivare uno shell sul sistema a cui ci connettiamo.



🏁 00:25 05-07

Creazione di un processo figlio

In un sistema **UNIX**, se un processo(padre) volesse lanciare in esecuzione un altro processo (figlio), tipicamente diverso da se stesso, ha bisogno di compiere due passi:

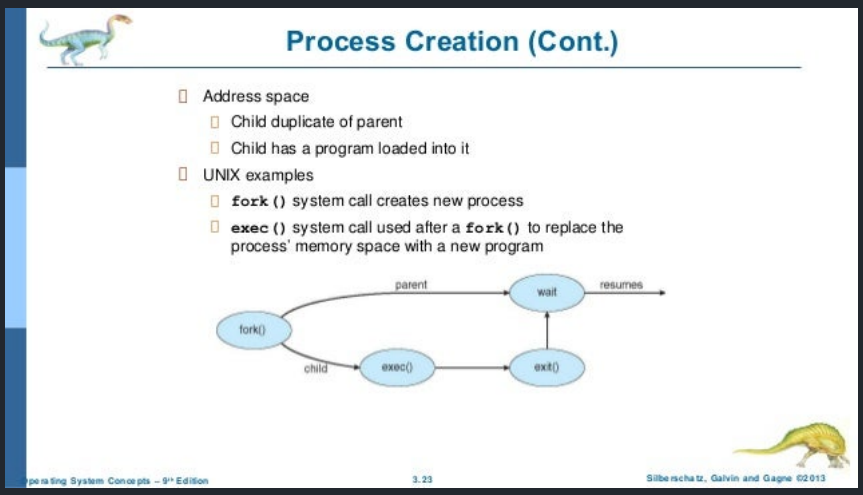
1. Viene creato un processo figlio che è una copia esatta del padre, con un **pid** diverso, ma con la memoria esattamente uguale a quella del padre. La syscall responsabile di questa operazione è la **fork()**.
   Questa funzione è particolare, perchè viene **invocata** una volta, ma "ritorna" due volte. Questo perchè ritorna un valore di ritorno diverso per il padre ed il figlio.

   In questo modo, il padre riceve il **pid** del processo figlio (pid>0), mentre il figlio riceve un pid = 0. Questo ci permette di effettuare un **if** sul **pid restituito dalla fork**, in modo da capire se ci troviamo nel processo padre o in quello figlio.

   In modo analogo, anche la **exit()** ha una semantica particolare, perchè viene chiamata una volta e non ritorna mai.

2. Dopo aver ricevuto un pid = 0, il processo figlio chiama la funzione **exec():**
   Il processo figlio effettua una system call **exec()** con cui chiede il caricamento di altri dati nella memoria.
   In poche parole, questa funzione **distrugge** lo spazio degli indirizi, e carica un **nuovo programma**, all'interno di quel numero di processo (il pid non cambia).

3. Nel frattempo il padre può attendere che il figlio abbia completato la sua esecuzione, e quindi chiama **wait().** Questo avviene con la **shell**, che riprende la sua esecuzione quando il programma da noi scelto termina.
   Il padre può anche scegliere di continuare la sua esecuzione, non chiamando wait().



Se volessimo lanciare un programma, e non attendere la sua terminazione (tramite shell) basta scrivere il comando e terminare con una E commerciale (&):

```
/bin/ls &
```

> Input

```
MBP-di-Giuliano:ArchitetturaDeiCalcolatori folly$ /bin/ls &
[1] 39275
MBP-di-Giuliano:ArchitetturaDeiCalcolatori folly$ README.md Resources

[1]+  Done                    /bin/ls
MBP-di-Giuliano:ArchitetturaDeiCalcolatori folly$
```

> output

Terminazione di un processo

Un processo termina eseguendo una system call di *fine* che tipicamente viene chiamata **exit().**

Un processo padre potrebbe **volontariamente** terminare un processo figlio con una syscall chiamata **abort()**. Questo potrebbe essere necessario per diverse motivazioni. Di conseguenza deduciamo che i processi padre hanno un "potere decisionale" sui processi figli.

Cosa succede se un processo padre termina prima del figlio?

Una possibilità è che si abbia una **cascading termination**, ovvero una terminazione a cascata; se il primo processo termina, tutti i processi figli vengono terminati a cascata. Questa **non è** la soluzione adottata dai sistemi **UNIX**; infatti, se un processo padre viene terminato, i processi figli possono continuare la loro normale esecuzione.

Cosa succede quando un processo figlio esegue una exit()?

L'exit del figlio fornisce un'informazione al padre attraverso la **wait()**, che sta attendendo l'esecuzione del figlio. Quando il padre riceve questa informazione, il processo figlio viene completamente espulso dal sistema con la deallocazione della sua memoria.

Se il processo padre invece è terminato senza invocare la **wait()**, il processo figlio viene detto **orfano.**

Se il processo padre invece non è in attesa, quindi anche in questo caso non ha invocato la **wait()**, il processo figlio viene detto **zombie**.

Questo vuol dire che bisogna trovare un modo per far si che l'informazione comunicata con la **exit()** dal figlio sia recapitata a qualche componente del sistema, in modo da far terminare il processo figlio. Finchè questo valore di ritorno della **exit()** non viene ricevuto da qualche componente, il processo figlio è teoricamente terminato, ma la sua memoria è ancora allocata (**zombie**).

Soluzione nei sistemi UNIX

Il valore di **exit()** viene preso da un processo che deve **per forza** essere in esecuzione (per tutta l'accensione del sistema) che è il processo con **pid = 1**, ovvero **init**. Questo vuol dire che se il **padre** termina prima del **figlio**, la **wait** viene eseguita dal processo **init**, in modo da espellere lo **zombie** dal sistema.

```
ps -l
```

```
UID   PID  PPID       F CPU PRI NI      SZ    RSS WCHAN     S             ADDR
501 38391 38390    4006   0  31  0 4317024   1236 -       S                0
```

Notiamo che se digitiamo ps -l, ci vengono mostrati i processi. Il processo **38391** (ovvero la shell) è il figlio del processo **38390**. Se effettuiamo la **kill** del padre, nel campo **PPID** apparirà il numero **1**, ovvero **init()**.

```
MBP-di-Giuliano:ArchitetturaDeiCalcolatori folly$ ps ax
  PID   TT  STAT      TIME COMMAND
    1   ??  Ss    13:21.22 /sbin/launchd
   39   ??  Ss     0:27.96 /usr/sbin/syslogd
   40   ??  Ss     0:43.05 /usr/libexec/UserEventAgent (System)
  ...
```

Invece, digitando `ps ax` otteniamo la lista di tutti i processi, e su **MacOS** il processo numero 1, al posto di **init** è **launchd (launch deamon)**.

Comunicazione tra processi

`recap:` In un sistema operativo lancio dei programmi in esecuzione; questi programmi vengono eseguiti tramite una gerarchia padre-figli, e questi programmi vengono eseguiti "contemporaneamente" contendendosi la CPU. I sistemi moderni sono **Multiprogrammati**, ovvero possono, appunto, eseguire più programmi per volta; l'unico sistema non multiprogrammato era **MS-DOS**, dove poteva essere eseguito un unico programma per volta.

Dei programmi semplici (come un hello world) sono detti **indipendenti**, siccome non comunicano con l'esterno. Questo **non è** il modo migliore per programmare.

Programmi cooperanti

Un programma può cooperare con un altro programma. Infatti, a partire dai primi sistemi **UNIX** si è capito come far comunicare diversi software, ed un esempio è il seguente:

```
ps ax |more
```

Questo comando `ps ax` ci permette di listare **tutti** i processi attivi nel sistema. Il risultato di questo comando è una lunga lista di processi. Se aggiungiamo `|more` possiamo vedere il risultato a pagine.

Questo significa che l'output di `ps ax` va ad un altro programma chiamato `more` che ci permette di visualizzare il risultato a pagine.

Con questo modo di programmare, abbiamo che **l'output** di un programma, diventa **l'input** di un altro, a catena.

Il comando grep

```
ps ax | grep bash
```

grep permette di cercare una parola all'interno di un file, stream, o all'interno di più files. È quindi una funzione di ricerca avanzata da linea di comando, che ci permette di trovare una stringa di caratteri.

> output:

```
MBP-di-Giuliano:ArchitetturaDeiCalcolatori folly$ ps ax | grep bash
38391 s000  S      0:00.14 -bash
39430 s000  S+     0:00.00 grep bash
```

Motivazione per i processi cooperanti

- Condivisione delle informazioni: durante l'esecuzione ho delle informazioni che sono condivise da più programmi e questi si scambiano delle informazioni.
- Speedup della computazione: voglio creare un programma **parallelo** , ovvero ho diversi "pezzi" del mio programma che effettuano operazioni diverse, questi possono girare su core diversi ed ottenere quindi una computazione più veloce.

In tutti questi casi ho bisogno di far comunicare i processi; questo funzionamento è "antico" e si trova nei sistemi operativi praticamente da sempre.

La parte dei sistemi operativi che si occupa della comunicazione tra processi, viene detta in gergo **IPC - Interprocess Communication**.

Come avvengono le comunicazioni tra processi?

Memoria condivisa - Shared Memory

Il modo più comodo e performante per far condividere ai processi delle informazioni, in un sistema tradizionale, è utilizzare una fetta di **memoria** che deve essere resa accessibile a più processi. La memoria deve essere resa accessibile perché solitamente i SO **restringono l'accesso** allo spazio degli indirizzi per quel processo, impedendo a quest'ultimo di accedere alla memoria di altri processi.

Questa tecnica è preferita nel momento in cui è presente della **memoria fisica**.

Scambio di messaggi - Message Passing

Piuttosto di scrivere in un'area di memoria dove poi un altro processo andrà a leggere, vengono effettivamente trasportati dei messaggi dal SO e ricevuti dal secondo processo. Questa tecnica **richiede l'intervento del sistema operativo.**

Questa tecnica è preferita nel momento in cui i due processi che devono comunicare sono su due sistemi diversi, ovvero quando non è presente della memoria fisica.

`Esempio:` Quando accediamo ad un server web, il processo locale (browser) manda un messaggio al server, dove è presente un altro processo, che riceve il messaggio e restituisce la pagina web.

Riassumendo

- **Processi Indipendenti:** Il processo non può influenzare o essere influenzato dall'esecuzione di un altro processo.
- **Processi cooperanti:** I processi possono influenzare o essere influenzati dall'esecuzione di altri processi.
- **Vantaggi della cooperazione dei processi:**

- Condivisione delle informazioni
- Speed-up della computazione
- Modularità
- Convenienza

🏁 1:20 05-07

Come comunicano due processi - punto di vista logico

Il concetto fondamentale utilizzato per far comunicare i processi, è nella maggioranza dei casi un paradigma di tipo **produttore - consumatore**. Ciò significa che il **produttore** produce dei dati tramite un processo, che poi viene **consumata** da un consumatore.

Memoria condivisa

Se questa interazione avviene tramite **memoria condivisa**, come fa il produttore a far arrivare i dati al consumatore? Questo avviene tramite **un'area buffer** che sia appositamente organizzata per l'operazione. Questo buffer può essere immaginato come uno spazio **illimitato**, anche se non esiste memoria illimitata. Il miglior modo per immaginare il buffer è immaginando un buffer **limitato**, ovvero uno spazio di memoria con dei limiti.

Scambio di messaggi

Il SO, ed in particolare L'interprocess Communication, deve fornire delle syscall generiche del tipo:

- **send(message)**
- **receive(message)**

Questo significa che un processo effettua una **send()**, che conterrà dati binari, testo ecc, mentre l'altro processo effettua una **receive()**.

Ogni IPC utilizza una tecnica tendenzialmente dagli altri, ma alla base è sempre presente il medesimo ragionamento.

Se i processi P e Q vogliono comunicare, essi devono:

- Stabilire un **link di comunicazione** tra di loro
- Scambiarsi messaggi tramite **send e receive** .

Problemi di implementazione

- Come si fa a stabilire un link tra due processi?
- Un link unisce solo due processi oppure può essere pensato come un canale di comunicazione che può far comunicare più processi?
- Qual è la capacità di un link (size)?
  Supponiamo ci siano due processi, con un link di comunicazione tra di loro. Supponiamo che uno dei due invii dei messaggi ma il secondo ancora non li abbia ricevuti. Questo vuol dire che il sistema deve avere un **buffer** dei messaggi inviati ma non ancora ricevuti.
- La grandezza dei messaggi che un link può supportare sono di grandezza fissa o variabile?
- Un link è **unidirezionale o bidirezionale?**

Ogni implementazione fa le sue scelte, e sceglierà delle soluzioni.

Implementazione del canale di comunicazione (link)

Livello Fisico

Il supporto per la comunicazione può essere sicuramente la **shared memory**, anche se in questo caso, se la memoria condivisa è presente, conviene utilizzare direttamente la memoria stessa, e non lo scambio di messaggi. Potrebbe inoltre fare utilizzo di un **bus hardware**, oppure della **rete** stessa.

Livello Logico

- **Diretto o indiretto** : ovvero una comunicazione diretta tra due processi, oppure tra i due può esserci un qualcosa che funziona come una casella postale.
- **Sincrono o asincrono**
- **Buffer Automatico o esplicito**

Proprietà del canale di comunicazione (link)

Per stabilire un **link di comunicazione** i processi devono utilizzare il cosiddetto **naming esplicito**, ovvero ogni processo **deve** nominare esattamente l'altro processo con cui vuole comunicare. Questo naming è un vero e proprio parametro da inviare con la **send() e receive():**

- **send(P , Message)**
- **receive(Q, Message)**

Proprietà generali:

- I link sono stabiliti automaticamente
- Un link è associato ad esattamente una sola **coppia di processi** .
- Tra ogni coppia esiste un solo link.
- Il link potrebbe essere unidirezionale, ma è solitamente **bidirezionale.**

Comunicazione indiretta

Nel caso di comunicazione indiretta, significa che tra i due processi viene vista **un'entità logica**, chiamata **mailbox**. Significa che il processo non comunica l'altro processo con cui vuole parlare, ma il contenitore gestito dal SO, dove andrà a finire il messaggio inviato dal processo. Il **link** viene stabilito mediante la condivisione di una mailbox comune ai due processi.

Proprietà generali della comunicazione indiretta:

- Il link è stabilito solo se i processi condividono una **mailbox comune.**
- Un link può essere associato con diversi processi
- Ogni coppia di processi potrebbe condividere diversi link di comunicazione

- i link possono essere di tipo unidirezionale o bidirezionale.

Abbiamo delle operazioni basilari per la mailbox:

- **Creare una nuova mailbox (port)**
- **Inviare** e ricevere messaggi attraverso una mailbox
- **Distruggere** una mailbox

In questo caso la **send() e receive()** diventano:

- **send(A, message)** inviare un messaggio alla **mailbox A** .
- **receive(A, message)** ricevere un messaggio dalla **mailbox A** .

Piccolo problema

Un problema abbastanza evidente è il seguente: se diversi processi inviano messaggi ad una mailbox, e diversi processi eseguono la **receive()** il messaggio a chi deve essere recapitato?

Soluzione:

- In alcuni casi i SO chiedono che un dato link di comunicazione venga associato a **solo due processi.**
- In altri casi solo un processo per volta può effettuare la **receive()** .
- In altri casi il receive viene selezionato arbitrariamente. In questo caso il processo che ha inviato viene notificato dell'identità del ricevente. Non una buona soluzione.

🏁 00:04 05-12

Sincronizzazione

Il **message passing** può essere di tipo **bloccante o non bloccante.**

Quando lo scambio di messaggi è **bloccante**, abbiamo che il tipo di comunicazione è **sincrono.**
Nel momento in cui eseguo una primitiva di comunicazione, perdo l'utilizzo della CPU finchè non viene eseguita l'operazione sull'altro processo.

- **Blocking send:** Il processo che invia è bloccato finchè il messaggio viene ricevuto.
- **Blocking receive:** Il processo ricevente è bloccato finchè un messaggio non è disponibile.

Quando lo scambio è **non blocking**, il tipo di comunicazione è **asincrono**. Questo tipo di comunicazione è comunemente utilizzato durante la **send()**.

- **Non-blocking send** , che è il metodo più comunemente utilizzato, permette al processo di depositare il messaggio, e continuare l'esecuzione del processo che ha inviato il messaggio.
- **Non-blocking receive** : è un'operazione "strana", perchè il processo attende il messaggio, ma nel frattempo il processo continua ad essere eseguito; questo vuol dire che dietro il codice del processo è presente un **loop** che viene eseguito finchè non viene ricevuto il messaggio. Questo metodo non è molto utilizzato.

Esiste infine un ultimo tipo, detto **Randezvous**, ovvero quando sia la send che la receive sono bloccanti, ovvero i due processi si attendono a vicenda.

Buffering

Quando abbiamo una **send asincrona** deve essere presente una coda di messaggi gestita dal sistema in modo da tenere traccia dei messaggi inviati. Se il **size del buffer** è 0, vuol dire che la comunicazione deve essere **sincrona**, in modo da attendere quindi il ricevimento di un messaggio alla volta.

Esistono altre due size possibili del buffer, **Limitato** ed **illimitato**. Quando il buffer è **limitato**, il processo che invia il messaggio deve attendere (diventa una sorta di comunicazione sincrona) solo nel momento in cui il buffer è pieno. Nel caso del buffer **illimitato**, il processo che invia non deve mai attendere (caso puramente teorico).

Sockets (veloce)

I sockets sono una forma di comunicazione tramite scambio di messaggi, che permettono di connettersi ad un sistema esterno. Questo è il tipo di comunicazione sulla quale si basano tutte le connessioni sulla rete.