Comunicazione tra processi

In un sistema operativo ci sono un numero molto elevato di programmi in esecuzione (processi). Idealmente tali processi dovrebbero "comportarsi bene" evitando di interferire uno con l'altro. Nella pratica, difficilmente il comportamento di un processo è indipendente da quello degli altri processi in esecuzione.

Competizione

I processi competono innanzitutto per le risorse comuni. Ad esempio:

- Apertura dello stesso file;
- Utilizzo della stessa stampante;
- Condivisione della CPU.

La competizione per le risorse crea *interferenze* tra processi. Vediamo alcuni esempi:

- **Starvation**: un processo è bloccato indefinitamente a causa di altri processi che monopolizzano una o più risorse. Abbiamo visto un esempio estremo di programma "fork-bomb" che se non limitato opportunamente può arrivare a sottrarre tutte le risorse agli altri processi:

```
main() {
        while(1)
                if (fork() < 0)
                        perror ("errore fork");
}
```

- **I/O**: L'accesso a una risorsa di input/output può variare notevolmente di prestazioni a seconda di quanti processi la stanno utilizzando.

Il sistema operativo deve **gestire la competizione** su risorse comuni in modo da ridurre il più possibile le interferenze e da garantire correttezza. Ad esempio:

- Le stampe su una stampante devono essere accodate e non "mischiate" in modo incontrollato;
- L'accesso al disco deve garantire che diversi processi non scrivano inavvertitamente sulla stesso blocco;
- L'accesso alla CPU deve essere regolato da un timer che la assegna in modo sufficientemente equo ai diversi processi in esecuzione.

A tale scopo, il sistema operativo **virtualizza** l'hardware in modo da dare l'impressione che ogni processo abbia una istanza dedicata della risorsa. Quando scriviamo sul disco, a parte la velocità di accesso, non ci accorgiamo se altri processi stanno leggendo o scrivendo su altri file. Allo stesso modo, non ci accorgiamo di altri processi in esecuzione sulla CPU, fintantoché questo non rallenta visibilmente l'esecuzione dei nostri programmi.

Cooperazione

Ci sono molti casi in cui l'interazione è però voluta. Vediamo alcuni esempi:

- **Condivisione**: quando si vuole condividere informazione è necessario interagire. Ad esempio in un progetto software o un wiki. Ci sono molti file nel sistema che sono condivisi da diverse applicazioni;
- **Prestazioni**: con le architetture multi-core si possono utilizzare algoritmi paralleli per aumentare le prestazioni. Se un programma è scritto in modo sequenziale non andrà a sfruttare la presenza di più core;
- **Modularità**: un'applicazione complessa viene spesso suddivisa in attività distinte più semplici, ognuna delle quali viene eseguita da un programma distinto (processo o thread). Un correttore ortografico in un editor di testo o in un browser è un tipico esempio: la ricerca di errori avviene parallelamente all'attività principale ma chiaramente i dati (il testo) sono condivisi ed è necessaria una interazione.
  I comandi Unix sono un altro tipico esempio di comportamento modulare:

```
$ ls -al | grep pippo
-rw-rw-r--   1 focardi focardi       0 Jul 12 12:48 pippo.html
-rw-r--r--   1 focardi focardi    2014 Feb 13  2012 pippo.txt
$
```

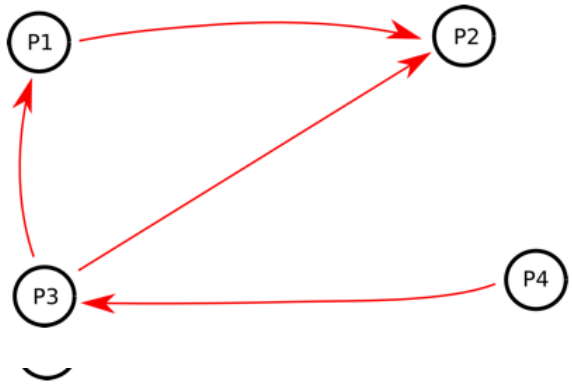
Il comando `ls -al` mostra il contenuto del folder. Il suo output viene dato in input a un secondo comando `grep pippo` che stampa solo le righe contenenti la parola `pippo`. Il simbolo "|" (pipe) indica appunto che l'output del primo comando deve essere dato in input al secondo. In questo modo otteniamo un comportamento utile combinando due comandi più semplici;

- **Convenienza**: può risultare comodo eseguire un'attività mentre si continua con un'altra. Vogliamo ad esempio lanciare una stampa e continuare a lavorare sul documento mentre il file viene stampato;
- **Replicazione**: quando è necessaria l'esecuzione simultanea di diverse istanze di un'attività diventa comodo replicarla su più processi o thread. Un esempio tipico è un server che si replica per servire diversi utenti contemporaneamente. Ogni processo servente dovrà tipicamente coordinarsi con gli altri nell'accesso alle risorse comuni.

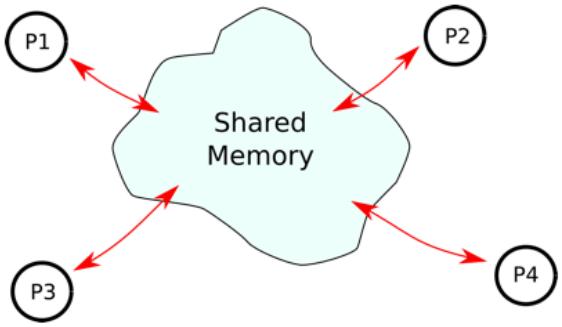
Modelli di comunicazione

Per cooperare è necessario comunicare. Esistono due modelli fondamentali di comunicazione tra processi e thread:

1. **Message passing** (scambio di messaggi): i processi o thread si scambiano informazioni tramite messaggi, un po' come avviene sulla rete;



2. **Shared memory** (memoria condivisa): i processi o thread condividono dati in memoria e accedono in lettura e scrittura a tali dati condivisi.



In questa prima parte del corso ci interessiamo allo scambio di messaggi in quanto costituisce il modello di riferimento per la comunicazione tra processi. Nella seconda parte del corso discuteremo approfonditamente il modello a memoria condivisa che è, invece, il modello di riferimento per la comunicazione tra thread. I thread, infatti, condividono tutte le risorse del processo, tra cui anche la memoria.

Scambio di messaggi

I processi dispongono di due primitive

- **send(m)**, invia il messaggio m;
- **receive(&m)**, riceve un messaggio e lo salva in m.

Vengono realizzate tramite opportune System Call dette InterProcess Communication (**IPC**).
Mittente e destinatario possono essere indicati direttamente o indirettamente.

Nominazione diretta

Mittente e destinatario sono nominati esplicitamente nelle primitive:

- send(**P**,m), invia il messaggio m a **P**;
- receive(**Q**,&m), riceve un messaggio da **Q** e lo salva in m.

È come se esistesse **un canale riservato** per ogni coppia di processi. Basta conoscere la reciproca identità. Esiste anche una variante asimmetrica in cui si riceve da qualsiasi utente

- receive(**&Q**,&m), riceve un messaggio m da un qualsiasi utente. Messaggio e mittente vengono salvati in m e **Q**.

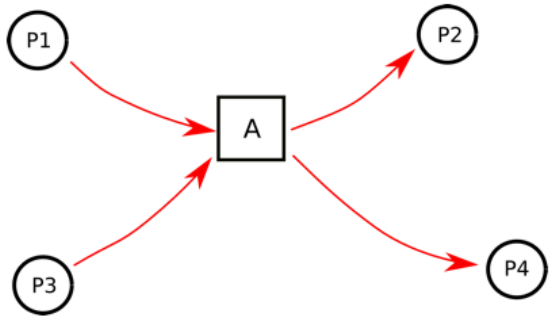
Vantaggi: la nominazione diretta è molto semplice e permette di comunicare in modo diretto tra coppie di processi;
Svantaggi: È però necessario un "accordo" sui nomi dei processi. Il PID infatti viene dato dinamicamente dal sistema e non possiamo prevederne il valore a priori. Nella pratica è difficile da implementare a meno che i processi non siano in relazione stretta di parentela genitore-figlio. In tale caso, infatti, il genitore conosce il PID dei vari figli e potrebbe comunicare direttamente con loro utilizzando tale identificativo. Vedremo, però, che nel caso di processi parenti non è nemmeno necessario utilizzare questo tipo di nominazione.

Nominazione indiretta

Per ovviare ai difetti della nominazione diretta si utilizza, in pratica, una nominazione indiretta tramite **porte** (o **mailbox**). Le porte sono gestite dal sistema operativo tramite opportune chiamate che ne permettono la creazione e distruzione.

- send(**A**,m), invia il messaggio m sulla **porta A**;
- receive(**A**,&m), riceve un messaggio dalla **porta A** e lo salva in m.

In questo modo non è necessario conoscere il nome dei processi ma solamente quello delle porte.



Competizione: cosa accade se due processi cercano di inviare o ricevere messaggi dalla stessa porta? In questo caso sarà il sistema operativo a gestire la competizione in modo opportuno, se possibile. Spesso si associa una porta a un processo (il creante e possessore della porta) che è l'unico a leggere da tale porta. Programmando in questo modo si evitano problemi legati alla competizione in ricezione. L'invio è meno problematico in quanto i messaggi possono essere accodati sulla porta.

Nel prossimo laboratorio vedremo come le pipe di Unix implementino un meccanismo di scambio di messaggi a nominazione indiretta. Vedremo inoltre che il riferimento alla pipe può avvenire tramite un descrittore (pipe senza nome) oppure tramite un nome conosciuto a livello di file-system (pipe con nome).

Comunicazione sincrona e asincrona

Invio e ricezione possono essere sincroni o asincroni.

- *send sincrona*: il messaggio viene inviato solo quando la corrispondente receive viene eseguita. La send è quindi bloccante fintantoché il messaggio non viene ricevuto. Un esempio può essere una telefonata: si inizia a parlare solo quando dall'altro lato c'è un interlocutore. Lo stesso avviene in una chat;
- *send asincrona*: il messaggio viene inviato indipendentemente dalla receive. Per realizzare una send asincrona è necessaria una **coda** (buffer) in cui depositare temporaneamente il messaggio. Un esempio può essere la posta che viene depositata nella buchetta e, analogamente, la email che viene inviata indipendentemente dalla presenza online del ricevente;
- *receive sincrona*: il messaggio viene ricevuto solo se presente. La receive è bloccante fintantoché non c'è un messaggio da leggere. Un server web in attesa di connessioni è un esempio di receive sincrona.

- *receive asincrona*: la receive ritorna un messaggo se presente o NULL se non ci sono messaggi da ricevere. Un client di email che periodicamente verifica la presenza di nuovi messaggi è un esempio di receive asincrona: se non ci sono nuovi messaggi il client prosegue con quelli presenti, altrimenti li aggiorna.

Produttore-consumatore

La comunicazione a scambio di messaggi è molto adatta nelle situazione in cui un processo "produce" un dato e un altro lo "consuma". Abbiamo visto l'esempio Unix:

```
$ ls -al | grep pippo
-rw-rw-r--   1 focardi focardi       0 Jul 12 12:48 pippo.html
-rw-r--r--   1 focardi focardi    2014 Feb 13  2012 pippo.txt
$
```

Il processo `ls -al` produce un output che viene dato in input a `grep pippo` che lo "consuma" generando un ulteriore output.

Questo esempio mostra anche una realizzazione della comunicazione a nominazione indiretta con send asincrona e receive sincrona. Vediamo in dettaglio:

1. la shell crea **due processi** figlio "`ls -al`" e "`grep pippo`";
2. la shell crea inoltre una **pipe** indicata dal simbolo "|". La creazione della pipe avviene prima delle fork. La pipe quindi è condivisa dai processi figli;
3. tale pipe diventa l'equivalente di una porta: i due processi figlio possono nominarla e utilizzarla per comunicare: "`ls -al`" manda tutto il suo output sulla pipe mentre "`grep pippo`" legge il proprio input dalla pipe;
4. la pipe ha un buffer che consente al primo processo di inviare in modo asincrono i dati.
   Per illustrare questo comportamento aggiungiamo una sleep prima di "grep" e mettiamo una stampa sullo standard error dopo "ls" (per deviare lo standard output sullo standard error è sufficiente usare la sintassi 1>&2 che ridirezione 1, stdout, su 2, stderr).

   ```
   $ (ls -al; echo "DONE ls" 1>&2) | (sleep 10;grep pippo)
   DONE ls
   -rw-rw-r--   1 focardi focardi       0 Jul 12 12:48 pippo.html
   -rw-r--r--   1 focardi focardi    2014 Feb 13  2012 pippo.txt
   $
   ```

   Notiamo che "DONE ls" viene stampato immediatamente mentre la stampa delle linee contenenti pippo avviene dopi 10 secondi. Significa quindi che "ls -al" ha terminato l'esecuzione immediatamente stampando "DONE ls" e il suo output (il contenuto del folder) è stato bufferizzato dalla pipe. Quando la sleep termina tale output viene letto dalla pipe e processato da "grep";

5. la ricezione dalla pipe è, di default, sincrona: il secondo processo attende fintantoché non ci sono dati da leggere.

   ```
   $ (sleep 2;ls -al) | grep pippo
   -rw-rw-r--   1 focardi focardi       0 Jul 12 12:48 pippo.html
   -rw-r--r--   1 focardi focardi    2014 Feb 13  2012 pippo.txt
   $
   ```

   In questo caso la sleep è prima di "ls" e quindi l'esecuzione è ritardata di 2 secondi. Il comando "grep" è in attesa sulla pipe e aspetta anch'esso l'esecuzione della sleep. La ricezione è infatti sincrona.