Processi e Threads

> Un processo è **propietario di uno spazio virtuale** ed è visto come **entita indipendente** in un os.

Il **multithreading** è l'abilità di un OS di gestire più "thread" (percorsi) all'interno di un processo.

I **thread**, a differenza dei processi, **condividono tutti l'area di memoria del processo di cui fanno parte**.

Vantaggi dei thread *rispetto ai processi*:

- Minor tempo di creazione e minor tempo di switching.
- Possono comunicare senza l'intervento del kernel.

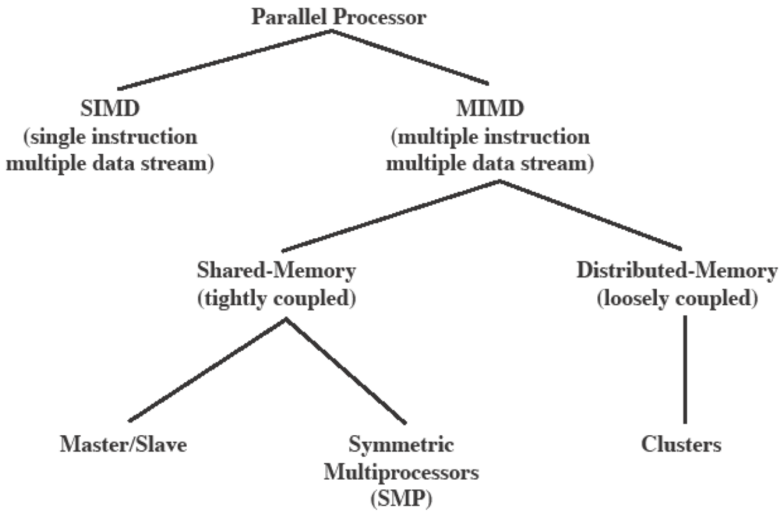
Ci sono due tipi di thread:

1. *User level thread*: sono gestiti completamente dall'applicazione, il kernel gestisce solo il processo che li crea. **Il *vantaggio* è la possibilità di switching senza l'intervento del kernel e che sono OS indipendenti**. **Lo *svantaggio* è che una system call bloccante di un thread li bloccherebbe tutti, e inoltre gli UTL non sfruttano i vantaggi delle architetture multicore.**
2. *Kernel level thread:* il kernel gestisce anche lo scheduling. **I vantaggi e gli svantaggi sono opposti a quelli degli ULT**.

> Sono possibili degli approcci combinati che sfruttano entrambi i tipi.

Categorie di ComputerSystems

- *Single instruction Single data:* un singolo processore esegue una singola istruzione, in una memoria singola.
- *Single instruction Multiple data*: molti processori eseguono una singola istruzione, ognuno su un diverso set di dati.
- *Multiple Instruction Single data*: molti processori eseguono differenti istruzioni, tramite una sequenza di dati che gli viene trasmessa.
- *Multiple instruction Multiple data*: molti processori eseguono differenti istruzioni, su sequenze di dati differenti.



Concorrenza

Table 5.2 Process Interaction

Degree of Awareness	Relationship	Influence that One Process Has on the Other	Potential Control Problems
Processes unaware of each other	Competition	• Results of one process independent of the action of others • Timing of process may be affected	• Mutual exclusion • Deadlock (renewable resource) • Starvation
Processes indirectly aware of each other (e.g., shared object)	Cooperation by sharing	• Results of one process may depend on information obtained from others • Timing of process may be affected	• Mutual exclusion • Deadlock (renewable resource) • Starvation • Data coherence
Processes directly aware of each other (have communication primitives available to them)	Cooperation by communication	• Results of one process may depend on information obtained from others • Timing of process may be affected	• Deadlock (consumable resource) • Starvation

Mutual Exclusion

La **mutual exclusion** deve essere *forzata*, e riguarda la propietà che ci deve essere affinchè un processo entri in maniera sicura in una **sezione critica.**

> Solo un porcesso alla volta deve poter accedere ad una **sezione critica** per effettuare operazioni o modificare dati.

- Nei sistemi con un solo processore si possono **eliminare le interrupts**, ma è una soluzione che porta a vari problemi.
- *Soluzioni Hardware:*
    - **Compare&Swap e Exchange:** la prima compara un valore di memoria con un valore di test, se i valori sono uguali, viene effetttuato uno swap con un nuovo valore. La seconda cambia il valore di un registro con quello di una posizione di memoria.

      > Hanno vari vantaggi, ma presentano un **busy waiting**, ed è possibile sial la starvation che il deadlock.

- Soluzioni comuni:
    - **Semafori**.
    - Mutex : *semafori che vengono bloccati/sbloccati dallo stesso processo*
    - Monitor: *equivalenti ai semafori, ma più facili da controllare*
    - Messaggi
    - Event flags

Quando un processo interagisce con un altro...

Sono fondamentali la **SINCRONIZZAZIONE** e la **COMUNICAZIONE**.

Questo porta al:

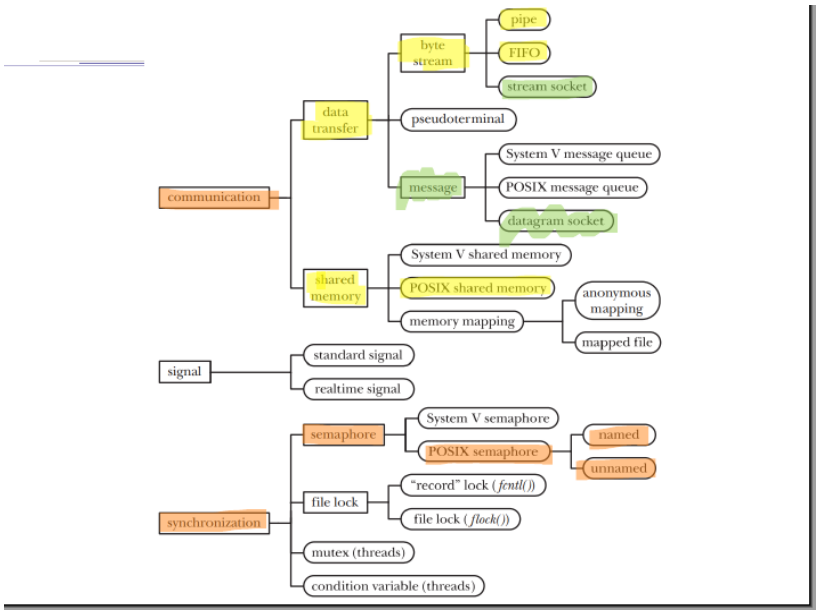
Message Passing

Comunicazione Inter Processo

> **Punti base** :
>
> - Un processo può accedere solo alla sua area di memoria
> - Ogni processo ha la sua
> - Il kernel può accedere a tutto

I processi in un sistema possono essere indipendenti o cooperanti, **indipendenti** quando non influiscono su altri processi, cooperanti altrimenti.

Ci sono vari vantaggi per **rendere i processi cooperanti**.

> Meccanismi per i processi per comunicare e sincronizzarsi
>
> 
>

Per la comunicazione tra processi ci sono 2 modelli basici:

- *Shared memory*: una regione di memoria condivisa è stabilita tra 2 o più processi, questi possono leggere e scrivere in questa memoria.
- *Data transfer:* i processi comunicano tramite due primitive di base `send(dest, msg)` e `recive(src, mesg)`. Per questo si usano le Pipe e le Fifo.
  - Pipe: permettono a più processi di comunicare attraverso un "tubo", una volta lette le informazioni non possono più ripresentarsi, i processi che usano le pipe devono essere relazionati (`fork()`).

    > Per evitare che non si verifichino *deadlock* è necessario che tutti i processi chiudano i descrittori delle Pipe che non gli servono, usando `close()`.

- FIFO: hanno le stesse funzioni delle pipe, ma permettono la comunicazione anche tra processi non relazionati.

Nella gestione si differenziano solo nella fase di apertura e chiusura.

Esempio di una scrittura tramite Pipe

```
#include <stdio.h>
#define Errore_(x) { puts(x); exit(1); }

int main(int argc, char *argv[]) {
    char messaggio[30];
    int pid, status, fd[2];

    ret = pipe(fd); /* crea una PIPE */
    if ( ret == -1 )
        Errore_("Errore nella creazione pipe");
    pid = fork(); /* crea un processo figlio */
    if ( pid == -1 ) Errore_("Errore nella fork");

    if ( pid == 0 ) { /* processo figlio: lettore */
        close(fd[1]); /* il lettore chiude fd[1] */
        while( read(fd[0], messaggio, 30) > 0 )
            printf("letto messaggio: %s", messaggio);
    close(fd[0]);
    _exit();
    }
    /* processo padre: scrittore */
    else {
        close(fd[0]);
        puts("digitare testo da trasferire (quit per terminare):");
    do {
        fgets(messaggio,30,stdin);
        write(fd[1], messaggio, 30);
        printf("scritto messaggio: %s", messaggio);
    } while( strcmp(messaggio,"quit\n") != 0 );
    close(fd[1]);
    wait(&status);
    }
}
```