

Università di Roma Tor Vergata
Corso di Laurea triennale in Informatica
Sistemi operativi e reti
A.A. 2016-17

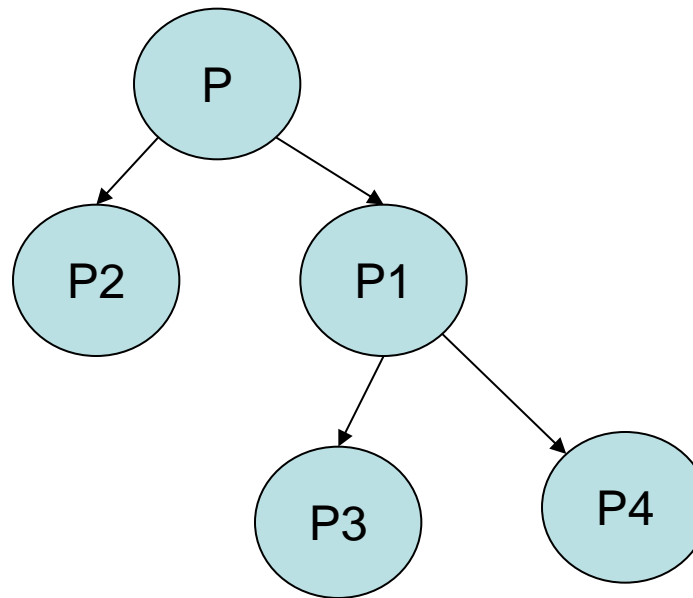
Pietro Frasca

Lezione 6

Giovedì 27-10-2016

Creazione e terminazione dei processi

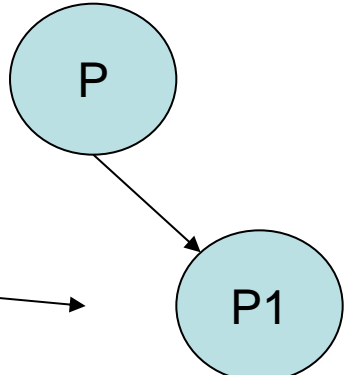
- In generale, durante la sua esecuzione un processo può creare altri processi utilizzando opportune chiamate di sistema fornite dal kernel. Il processo genitore prende il nome di processo **padre** ed il processo creato il nome di processo **figlio**. Ciascuno di questi nuovi processi può creare a sua volta altri processi, formando così un albero di processi.



- Tuttavia, si possono avere applicazioni nelle quali il numero dei processi è definito inizialmente e non viene più modificato durante il tempo di vita dell'applicazione. Questa gestione dei processi può essere implementata in applicazioni in tempo reale, ad esempio per il controllo di impianti fisici, in cui tutti i processi vengono creati all'avvio dell'applicazione (**creazione statica**).
- La condivisione di dati e risorse tra processi padri e figli e della loro sincronizzazione variano da sistema a sistema. Anche nel caso di terminazione di un processo, questa può avvenire secondo diverse politiche di segnalazione al processo padre.
- Generalmente, il kernel offre SC per la **creazione** e **terminazione** dei processi. La SC di creazione dovrà inizializzare il descrittore del processo da creare ed inserirlo nella coda dei processi pronti. Analogamente, la funzione di terminazione provocherà l'eliminazione del descrittore dalla tabella dei descrittori di processo e la notifica che l'area di memoria può essere recuperata dal sistema operativo.

Esempio di creazione di processo in POSIX

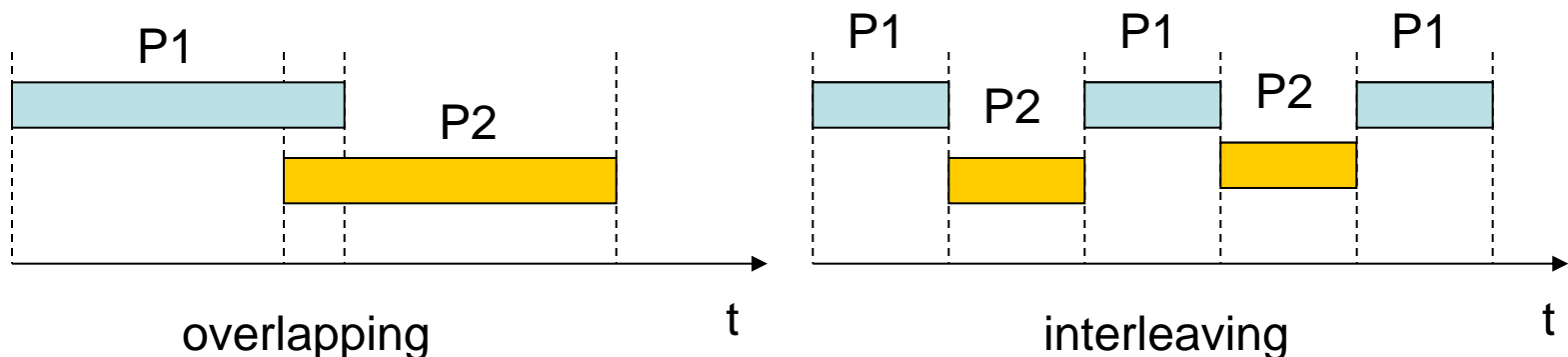
```
#include <stdio.h>
#include <stdlib.h>
int main() {
    int pid;
    pid=fork();
    /* fork ritorna il pid del figlio al padre,
       il valore 0 al figlio e -1 in caso di errore.
       In base a questo è possibile separare il codice del
       padre da quello del figlio.
       */
    if (pid==-1) {
        printf("Errore nella fork \n");
        exit(0);
    } else if (pid==0) {
        // codice del figlio
        printf("figlio con pid=%d \n",getpid());
    } else {
        // codice del padre
        printf("padre con pid=%d \n",getpid());
    }
    // codice eseguito dal padre e dal figlio
    printf("Questa istruzione printf è eseguita da pid=%d \n", getpid());
}
```



The diagram illustrates the process creation. A light blue circle labeled 'P' (parent) has an arrow pointing to another light blue circle labeled 'P1' (child). A line from the code 'pid=fork();' points to the arrow connecting P and P1, indicating that the fork system call creates the child process.

Interazione tra i processi

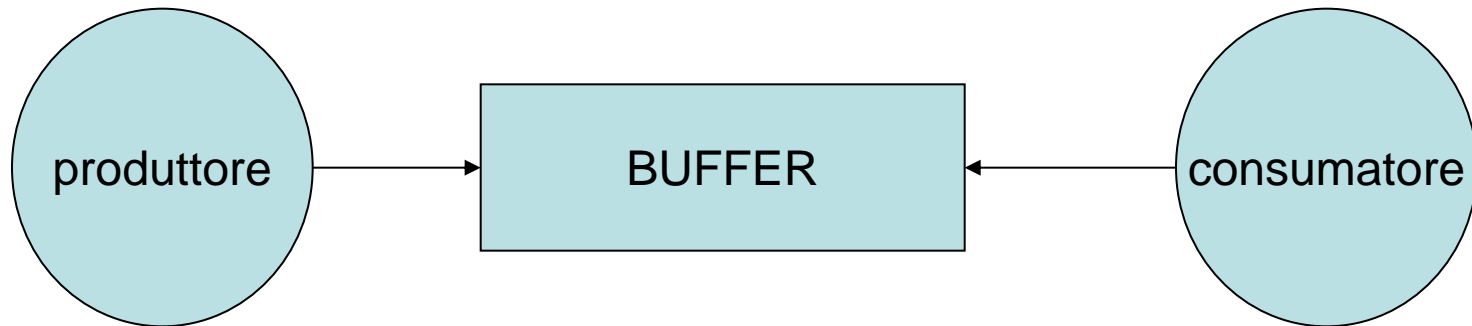
- Il termine ***processi concorrenti*** indica un insieme di processi la cui esecuzione si sovrappone nel tempo. La figura mostra il caso in cui ogni processo viene eseguito su una diversa CPU (***overlapping***) ed il caso in cui più processi condividono la stessa CPU (***interleaving***).
- I processi concorrenti possono comportarsi come ***indipendenti*** o come ***interagenti***.



- Un processo si dice **indipendente** quando non influenza l'esecuzione di altri processi. Processi che non condividono dati e che non si scambiano informazioni sono processi *indipendenti*.
- Al contrario, **i processi interagenti**, durante l'esecuzione, possono influenzarsi tra di loro. Questo può avvenire in:
 - **Cooperazione**: se i processi si **scambiano messaggi** o **segnali**;
 - **Competizione**: se *concorrono* per l'uso di una stessa risorsa.
- Gli effetti delle interazioni dipendono dalle **velocità relative di avanzamento di processi**. Poiché in generale tali velocità non sono prevedibili, perché sono funzione di eventi (come ad esempio le interruzioni) non dipendenti dal programma, il comportamento dei processi potrebbe cambiare, non essere riproducibile.

Esempio di processi cooperanti: modello produttore consumatore.

Un processo produttore genera periodicamente un messaggio e lo scrive in un buffer in grado di poter accettare un solo messaggio. Il processo consumatore periodicamente legge il messaggio dal buffer e lo *consuma*.

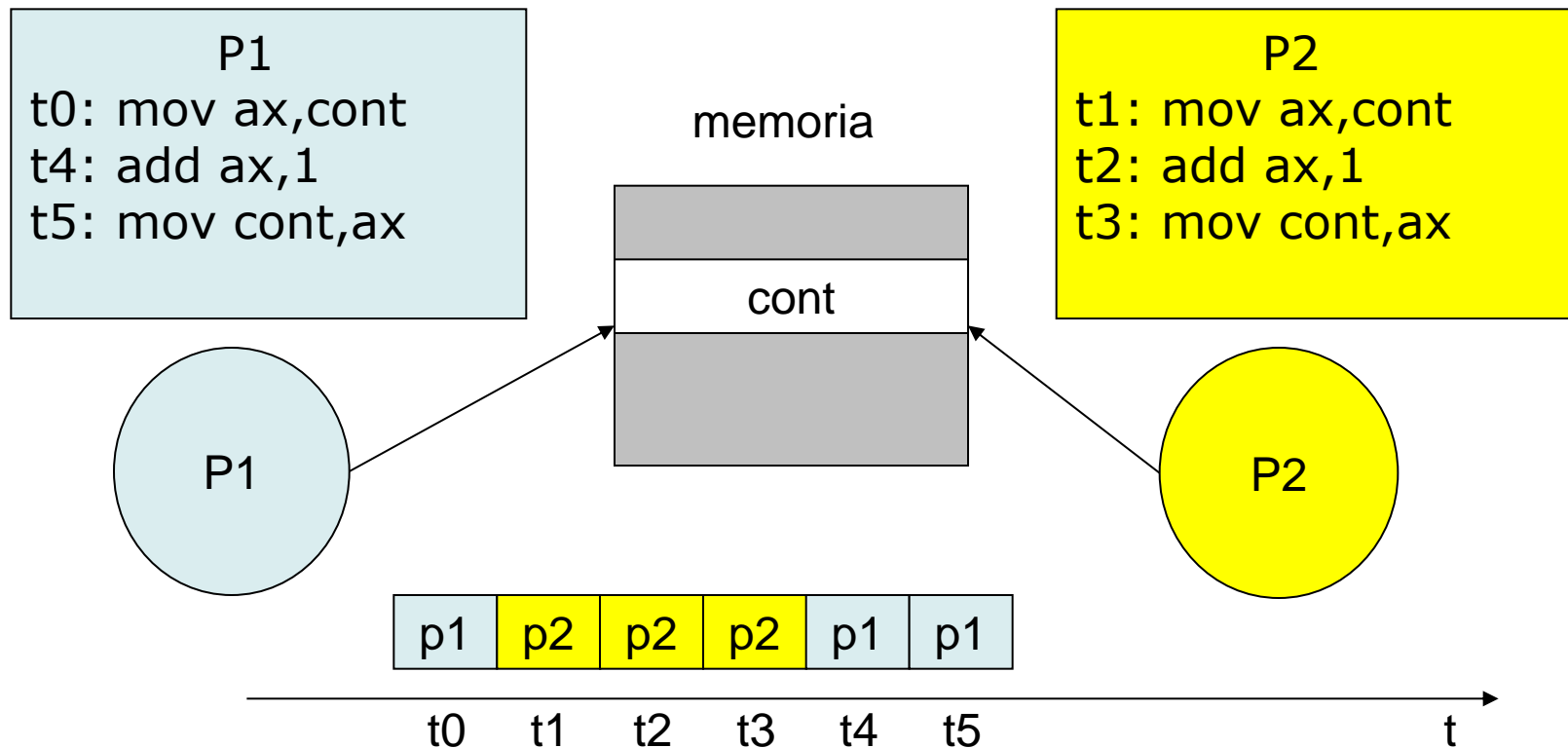


- Il corretto funzionamento è dato dalla sequenza: inserimento - prelievo – inserimento - prelievo...
- Altre sequenze come ad esempio inserimento – inserimento -prelievo oppure inserimento – prelievo - prelievo porterebbero ad un funzionamento errato.

Esempio di processi in competizione

- Due processi **P1** e **P2** condividono una variabile intera **cont**, inizializzata a zero, che incrementano ogni volta che eseguono un'operazione:

cont = cont + 1



- I due esempi evidenziano che i processi interagenti richiedono **operazioni di sincronizzazione**.
- Nel caso di processi cooperanti, le operazioni devono essere svolte seguendo un ordine preciso.
- Per i processi in competizione occorre garantire che solo un processo alla volta abbia accesso alla risorsa condivisa.

Il Kernel di un sistema a processi

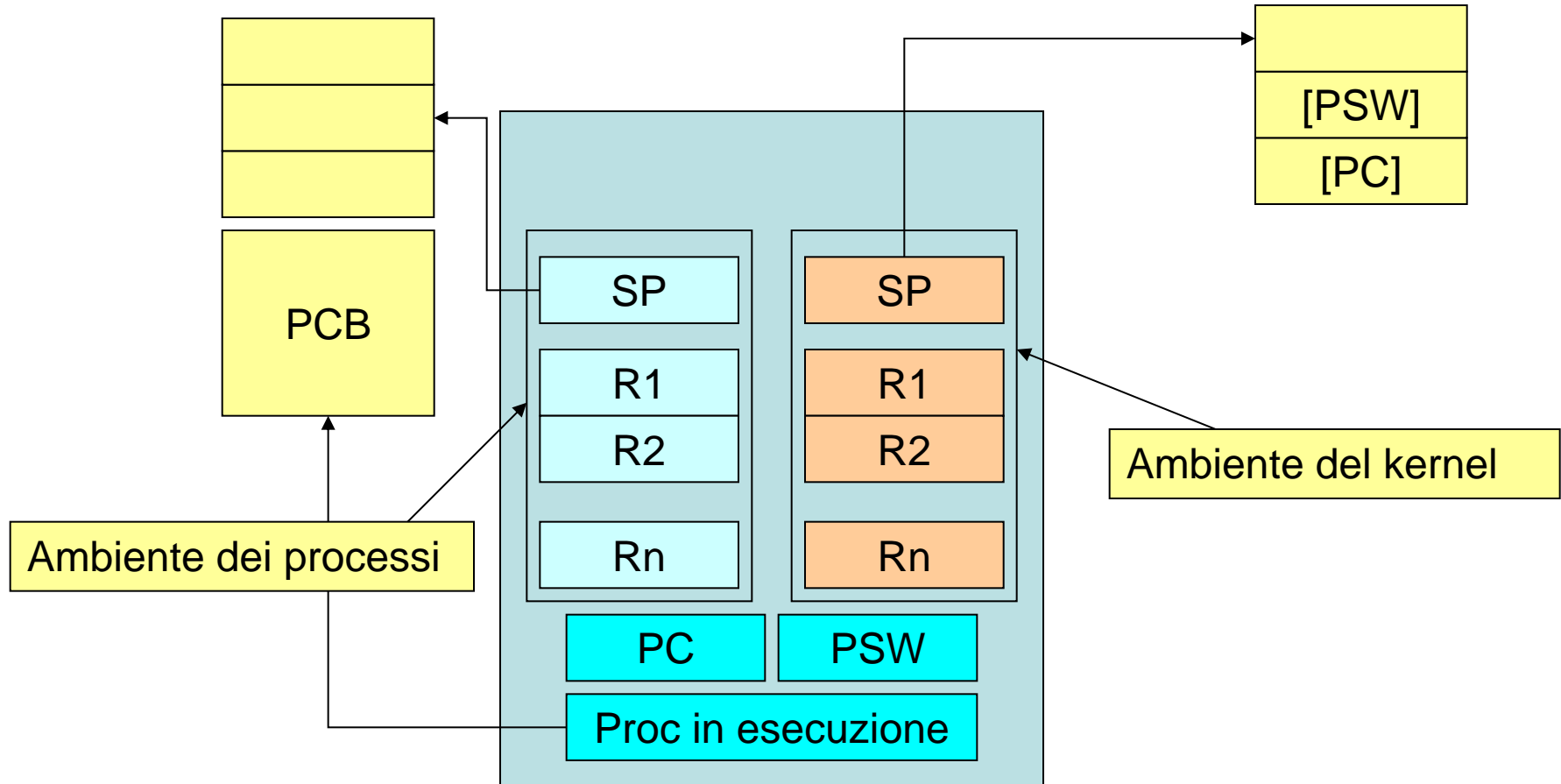
- Le strutture dati e le funzioni fin'ora descritte tra le quali il descrittore e le code dei processi, il cambio di contesto, lo scheduler, la sincronizzazione dei processi e molto altro ancora fanno parte del kernel.
- Le funzioni del kernel devono essere realizzate in modo che la loro esecuzione risulti molto efficace in termini di velocità. Pertanto, spesso tali funzioni sono realizzate in assembly e anche in hardware.
- Il passaggio dall'ambiente dei processi all'ambiente kernel è basato sull'uso delle interruzioni.
- Le funzioni del Kernel possono essere attivate da *interruzioni esterne (hardware)*, generate da dispositivi (periferiche) oppure da processi mediante opportune chiamate che generano *interruzioni interne (software)*, analoghe a quelle esterne.

Trasferimento tra l'ambiente dei processi e l'ambiente de kernel

- Per illustrare il trasferimento tra l'ambiente dei processi all'ambiente del kernel e viceversa, faremo riferimento ad una tipica architettura di calcolatore. Assumiamo inoltre che il modello di memoria per i processi comprenda un segmento (area di memoria) di stack, gestita dallo SP (Stack Pointer) , un segmento per il codice e un segmento di dati indirizzate dal PC (Program Counter).
- Lo stack costituisce l'area di lavoro del processo e contiene i record di attivazione delle funzioni o delle funzioni chiamate e delle loro variabili locali.
- Supponiamo che il processore possieda due copie di registri di uso generale R1, R2...RN e due registri SP relativamente ai due ambienti processi e kernel, anche se molti processori moderni hanno più di due set di registri.

Stack del processo

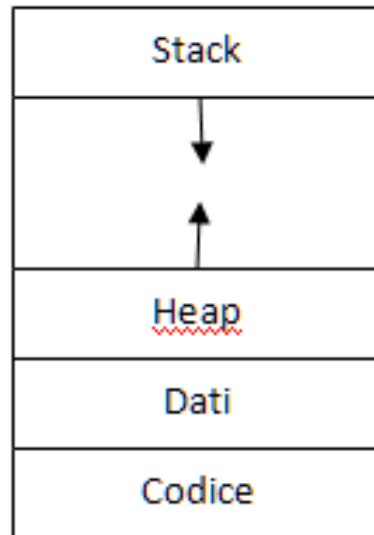
Stack del kernel



- Quando il processo esegue una system call (SC) si genera un'interruzione e il kernel esegue le seguenti operazioni:
 - salvataggio dei registri PC e PSW nello stack del kernel;
 - il caricamento in PC e in PSW dell'indirizzo della routine di gestione dell'interruzione e del registro di stato (PSW) del kernel.
 - **La routine di gestione dell'interruzione viene eseguita ora usando l'insieme dei registri dell'ambiente del kernel** e, in base ai parametri passati, provvede a richiamare la funzione richiesta.
- La SC può far passare il processo chiamante nello stato di bloccato oppure no. Se la SC **non è bloccante**, il kernel riporta in esecuzione il processo P mediante l'istruzione **IRET** che ripristina nei registri PC e PSW i valori contenuti nel suo stack (del kernel).

- Se la SC è **bloccante** per il processo, viene eseguito il salvataggio dei registri dell'ambiente processi e dei valori di PC e di PSW (comuni ai due ambienti) nel descrittore del processo P. Al termine della SC richiesta, viene eseguita la funzione per ripristinare lo stato che consiste nel caricare nei registri dell'ambiente processi i corrispondenti valori contenuti nel PCB ed i valori dei registri PC e PSW del nuovo processo a cui sarà assegnata la CPU.

- Come già descritto, un processo è composto da un descrittore, dal codice del programma che esegue, da un'area di memoria contenente i dati inizializzati e le variabili globali e dallo stack, un'area di memoria di lavoro necessaria per salvare i record di attivazione delle funzioni e le variabili locali. L'insieme di queste componenti è detta ***immagine del processo***.



- Al processo, inoltre, possono essere associate varie risorse, come processi figli, dispositivi di I/O, file, handler di segnali, ecc.
- L'immagine del processo e le risorse ad esso associate è detto il suo **spazio di indirizzamento**.
- Ogni processo ha uno spazio di indirizzamento distinto da quello di altri processi.
- La gestione dello spazio di indirizzamento dipende dalla tecnica di gestione della memoria che il SO utilizza. Come vedremo, lo spazio di indirizzamento potrà essere caricato completamente o solo parzialmente in memoria, in modo contiguo o in blocchi non contigui, di dimensioni fisse (*pagine*) o variabili (*segmenti*).
- La comunicazione tra processi può richiedere un alto tempo di esecuzione (*overhead*) dato che è necessario effettuare il salvataggio e il ripristino, anche se parziale, del loro spazio di indirizzamento.

- Anche la creazione e la terminazione di un processo risultano costose in termini di overhead.
- La *separazione* degli spazi di indirizzamento dei processi, consente da una parte la protezione dei dati dei singoli processi, ma dall'altra rende complesso l'accesso a strutture dati comuni, che si realizza attraverso **memoria comune** o mediante **scambio di messaggi**.
- Molte applicazioni possono essere sviluppate in moduli che girano in parallelo. Generalmente, ciascun modulo condivide dati e risorse comuni. Un esempio è dato da applicazioni in tempo reale per il controllo di impianti fisici in cui si possono individuare attività come il controllo di dispositivi di I/O dedicati a prelevare dati dall'ambiente fisico o a inviare comandi verso di esso. Ogni attività è implementata da un modulo che deve poter accedere a strutture dati comuni che rappresentano lo stato complessivo del sistema da controllare.
- Un altro esempio di applicazione che prevede l'esecuzione in parallelo di varie attività è l'editor di testi.

- Per ottenere una soluzione efficiente per le applicazioni che presentano un tipico grado di parallelismo, sono stati introdotti, nei moderni SO, **i thread (processi leggeri)**.

Thread

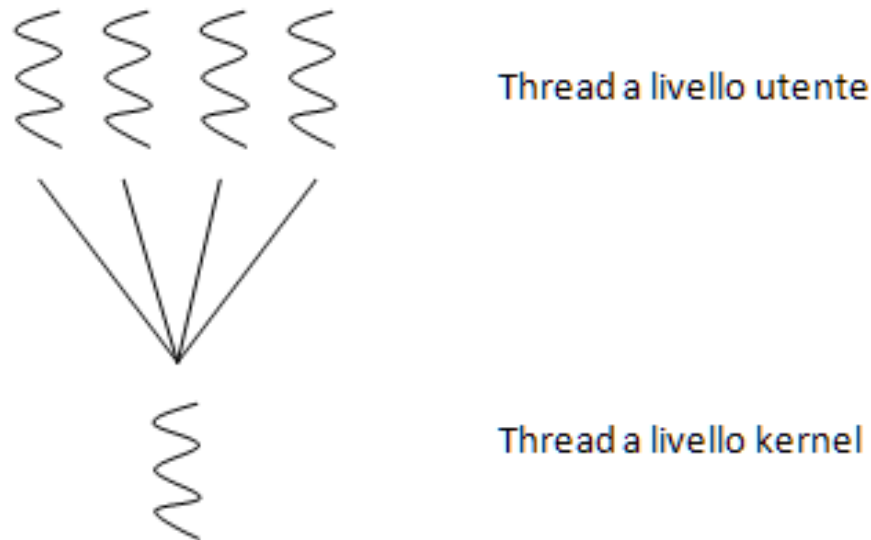
- Un thread è un **flusso di esecuzione** all'interno di uno stesso processo.
- All'interno di un processo è possibile definire più thread, ognuno dei quali condivide le risorse del processo, appartiene allo stesso spazio di indirizzamento e accede agli stessi dati, definiti con visibilità globale.
- Non possedendo risorse i thread possono essere creati e distrutti più facilmente rispetto ai processi; il cambio di contesto è più efficiente.
- Il termine **multithreading** significa che un processo possiede più thread.

- Ad ogni thread sono associati un descrittore, uno stato: esecuzione, pronto e bloccato, uno spazio di memoria per le variabili locali, uno stack, un contesto rappresentato dai valori di registri del processore utilizzati dal thread.
- Ad un thread non appartengono le risorse, che invece appartengono al processo che lo contiene.
- Le informazioni relative ai thread sono ridotte rispetto a quelle dei processi, quindi le operazioni di cambio di contesto, di creazione e terminazione sono molto semplificate rispetto a quelle dei processi.
- La gestione dei thread può avvenire sia a **livello utente** che a **livello kernel**.

Thread a livello utente

Modello da molti a uno

- Nel caso di thread a livello utente si utilizzano librerie realizzate a livello utente che forniscono tutto il supporto per la gestione dei thread: creazione, terminazione, sincronizzazione nell'accesso di variabili globali del processo, per lo scheduling, etc.; tutte queste funzioni sono realizzate nello spazio utente, il SO non vede l'esistenza dei thread e considera solo il processo che li contiene.

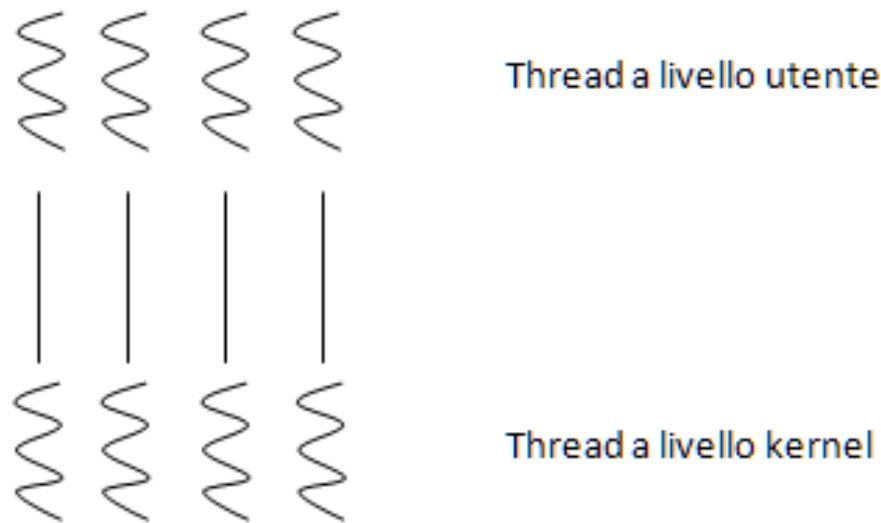


- I thread possono chiamare le system call, ad esempio per operazione di I/O; in questo caso interviene il SO che blocca il processo e di conseguenza tutti i thread in esso contenuti.
- I thread a livello utente hanno il vantaggio che possono essere utilizzati anche in SO che non supportano direttamente i thread; è possibile usare politiche di scheduling senza intervenire sul kernel del SO.
- Non è possibile, con i thread a livello utente, sfruttare il parallelismo delle architetture multiprocessore, dato che quando un processo è assegnato ad uno dei processori, tutti i suoi thread sono eseguiti, uno alla volta, su quel solo processore.

Thread a livello kernel

Modello da uno a uno

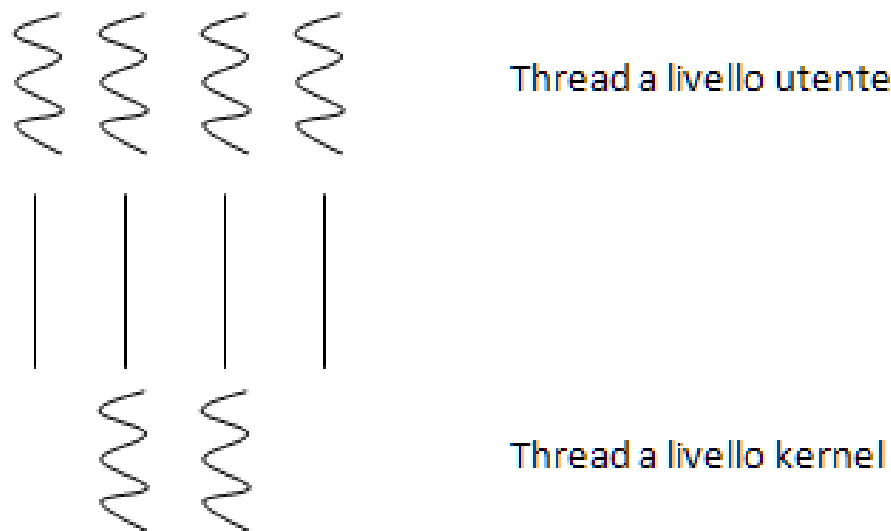
- La gestione dei thread avviene a livello kernel. In questo caso a ciascuna funzione di gestione dei thread, comprese la creazione, la terminazione, la sincronizzazione e lo scheduling, corrisponde a una chiamata di sistema e il kernel deve mantenere sia i descrittori dei processi sia i descrittori di tutti i thread.



- A differenza del modello precedente, ora se un thread esegue una chiamata di sistema bloccante, il thread si blocca ma gli altri thread possono continuare la loro esecuzione. Tuttavia, il carico dovuto alla creazione di un thread è più onerosa. Pertanto, per non degradare eccessivamente le prestazioni di un'applicazione multithread, generalmente si limita il numero di thread gestibili dal kernel.
- Praticamente tutti i sistemi operativi moderni, compresi Windows, Linux, Mac OS X, Tru64 UNIX, supportano questo tipo di modello. Inoltre, il modello uno a uno consente la reale esecuzione di thread in parallelo nei sistemi basati su architetture multiprocessore.

Modello da molti a molti

- Il modello da molti a molti mette in corrispondenza i thread a livello utente con un numero minore o uguale di thread a livello kernel. Questo modello presenta le caratteristiche vantaggiose dei precedenti modelli. È possibile creare tutti thread che sono necessari all'interno di un'applicazione e i corrispondenti thread a livello kernel possono essere eseguiti in parallelo nelle architetture multiprocessore. Inoltre, se un thread esegue una chiamata di sistema bloccante, il kernel manda in esecuzione un altro thread.



```

#include <pthread.h>
int a=10;          /* variabili globali condivise
char buffer[1024];    tra i thread */
void *codice_Th1 (void *arg){
    <CODICE DI TH1>
    pthread_exit(0);
}
void *codice_Th2 (void *arg){
    <CODICE DI TH2>
    pthread_exit(0);
}

int main(){
    pthread_t th1, th2;
    int ret;
    // creazione e attivazione del primo thread
    if (pthread_create(&th1, NULL, codice_Th1, "Lino")!=0){
        fprintf(stderr,"Errore di creazione thread 1 \n");
        exit(1);
    }
}

```

```
// creazione e attivazione del secondo thread
if (pthread_create(&th2,NULL,codice_Th2, "Eva")!=0){
    fprintf(stderr,"Errore di creazione thread 2 \n");
    exit(1);
}
// attesa della terminazione del primo thread
ret=pthread_join(th1,NULL);
if (ret !=0)
    fprintf(stderr,"join fallito %d \n",ret);
else
    printf("terminato il thread 1 \n");
    // attesa della terminazione del secondo thread
    ret=pthread_join(th2,NULL);
    if (ret !=0)
        fprintf(stderr,"join fallito %d \n",ret);
    else
        printf("terminato il thread 2 \n");
}
```