**LABORATORIO DI PROGRAMMAZIONE 2**
**Corso di laurea in matematica**

# Sistemi Operativi : processi e thread

Marco Lapegna
Dipartimento di Matematica e Applicazioni
Universita' degli Studi di Napoli Federico II

wpage.unina.it/lapegna



## Cosa e' un processo?

- Un sistema operativo esegue programmi di varia natura:
  - Compilatori, word processor, programmi utente, programmi di sistema,…

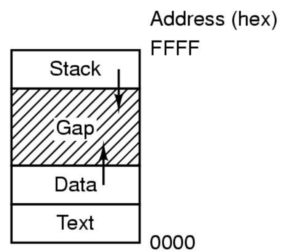
***Processo*** = un programma in esecuzione;

- l'esecuzione di un processo avviene in modo sequenziale.
- Ad un programma possono corrispondere piu' processi
- Termini sinonimi: task, job



## processi

- Un processo e' una entita' dinamica (il suo stato varia nel tempo)
- Ad ogni processo il sistema operativo assegna un'area di memoria




Address (hex)
FFFF
0000

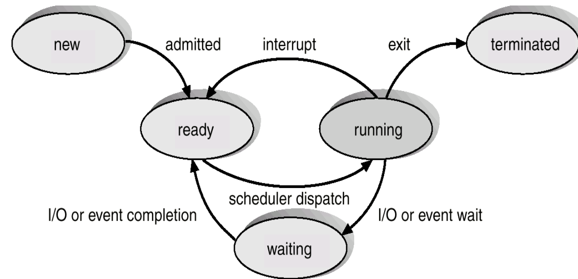
- area testo: contiene il codice eseguibile
- area dati: contiene le variabili globali
- area stack: contiene le variabili locali, e gli indirizzi di rientro delle subroutine



## Stato del processo

- In un sistema multiprogrammato, mentre viene eseguito un processo cambia **stato:**
  - **New** (nuovo): Il processo viene creato.
  - **Running** (in esecuzione): Le istruzioni vengono eseguite.
  - **Waiting** (in attesa): Il processo è in attesa di un evento.
  - **Ready** (pronto): Il processo è in attesa di essere assegnato ad un processore.
  - **Terminated** (terminato): Il processo ha terminato la propria esecuzione.

## stati di un processo




- sempre un solo processo running
- molti processi ready o waiting
- la cpu e' utilizzata a turno dai vari processi (context switch)



## Come il S.O. riconosce e gestisce i processi?

Per poter effettuare correttamente il context switch, il sistema operativo deve salvare una serie di informazioni in un opportuna struttura dati conservata nel kernel: Process Control Block

| pointer | process state |
|---|---|
| process number | |
| program counter | |
| registers | |
| memory limits | |
| list of open files | |
| ⋮ | |

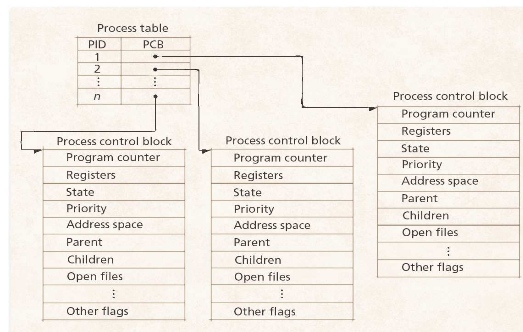
- Stato del processo
- process id
- Program counter
- Registri della CPU
  (accumulatori, indice, stack pointer)
- Informazioni sullo scheduling della CPU
  (priorità, puntatori alle code di scheduling)
- Informazioni sulla gestione della memoria
  (registri base e limite, tabella pagine/segmenti)
- Informazioni di contabilizzazione delle risorse
  (numero job/account, tempo di CPU)
- Informazioni sullo stato di I/O
  (lista dispositivi/file aperti)



## Process table

Il sistema operativo mantiene un puntatore ad ogni PCB in opportune Process table (per utente o generale)

Quando un processo e' terminato, il sistema operativo rimuove il processo dalla Process Table e libera le risorse





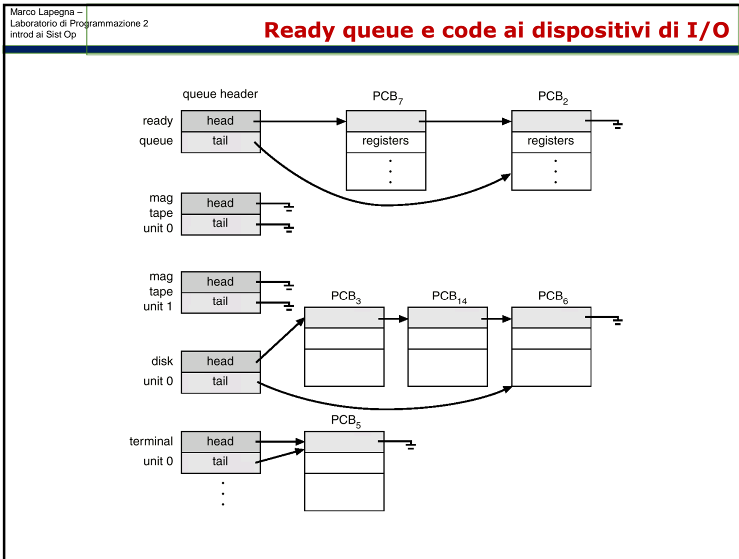

## Code per lo scheduling di processi

In un sistema operativo (monoprocessore) c'e' sempre un solo processo in esecuzione

Gli altri processi sono conservati in opportune code

- **Ready queue** (Coda dei processi pronti) — Insieme di tutti i processi pronti ed in attesa di esecuzione, che risiedono i memoria centrale.
- **I/O queue** (Coda dei dispositivi) — Insieme di processi in attesa per un dispositivo di I/O.

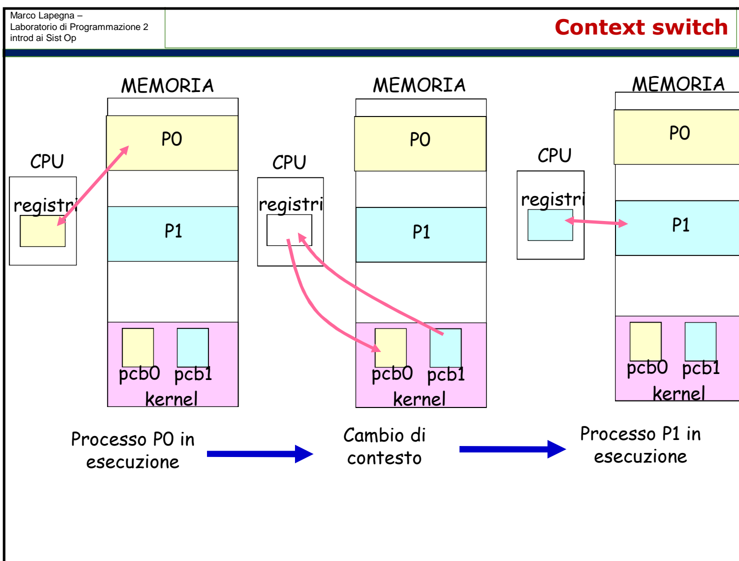
Le PCB dei processi si "spostano" fra le varie code.






## Context Switch

- Quando la CPU passa da un processo all’altro, il sistema deve salvare lo stato del vecchio processo e caricare lo stato precedentemente salvato per il nuovo processo.
- il sistema non lavora utilmente mentre cambia contesto.
- Il tempo di context–switch e’ un sovraccarico per il sistema e dipende dal supporto hardware (velocità di accesso alla memoria, numero di registri da copiare, istruzioni speciali, gruppi di registri multipli).






## Chi decide quale processo deve usare la cpu?

Un opportuno programma decide quale
tra i processi nella ready queue deve utilizzare la CPU

SCHEDULER

- Scheduler a **breve termine** (o scheduler della CPU) :
  - seleziona quale processo debba essere eseguito successivamente, ed alloca la CPU.
  - Deve essere chiamato molto spesso (~ 100 msec)
  - Deve essere veloce (~ 1 msec)
  - E’ responsabile dei tempi di attesa

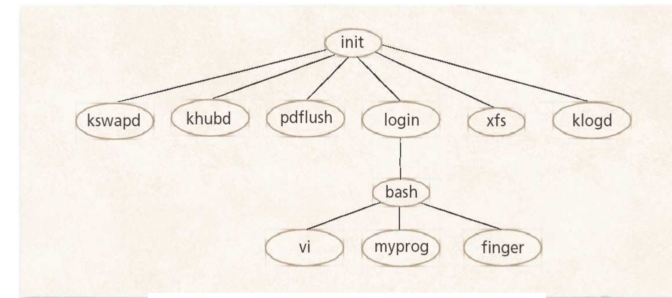
## Operazioni sui processi

- Tutti i sistemi operativi forniscono i servizi fondamentali per la gestione dei processi, tra cui:
  - Creazione processi
  - Distruzione processi
  - Sospensione processi (rimozione dalla memoria)
  - Ripresa processi
  - Cambio priorita' dei processi
  - Blocco processi (rimozione dalla cpu)
  - Sblocco processi
  - Attivazione processi
  - Comunicazione tra processi



## Come vengono creati i processi?

un processo (padre) puo' creare numerosi processsi (figli), che, a loro volta, possono creare altri processi, formando un albero (genealogico) di processi.




Albero dei processi in un sistema UNIX



## Creazione dei processi:

Risorse :
- Il padre e il figlio condividono tutte le risorse.
- I figli condividono un sottoinsieme delle risorse del padre.
- Il padre e il figlio non condividono risorse.

Minor carico nel sistema
↕
Maggior carico nel sistema

Spazio degli indirizzi
- Il processo figlio è un duplicato del processo padre (UNIX).
- Nel processo figlio è caricato subito un diverso programma (VMS).

Esecuzione
- Il padre e i figli vengono eseguiti concorrentemente.
- Il padre attende la terminazione dei processi figli.



## Un processo termina quando

- esegue l'ultima istruzione e chiede al sistema operativo di essere cancellato per mezzo di una specifica chiamata di sistema (**exit** in UNIX)
  - Può restituire dati al processo padre
  - Le risorse del processo vengono deallocate dal SO.
- Viene terminato dal padre quando, ad esempio:
  - Il figlio ha ecceduto nell'uso di alcune risorse.
  - Il padre termina (in alcuni sistemi)
    - *terminazione a cascata*
- Viene terminato da un altro processo (eventualmente il padre) per mezzo di una specifica chiamata di sistema (**abort** in UNIX)

## Esempio: UNIX

- la funzione fork crea un nuovo processo
  - il figlio viene creato copiando tutto il PCB del padre
  - ritorna 0 (zero) nel figlio
  - ritorna il pid del figlio nel padre
- la funzione execlp carica nel nuovo processo un programma
  - vengono sostituite le aree testo, data e stack
- la funzione exit fa terminare un processo
  - eventualmente comunica al padre lo stato di uscita
- la funzione wait fa attendere al padre la terminazione del figlio
  - eventualmente riceve dal figlio lo stato di uscita



## Esempio: UNIX

```
int main () {
    int pid;
    …
    pid = fork( );
    if( pid == 0 )   {
        execlp("/bin/ls","ls", NULL);      } Processo figlio

    }else{
        wait(NULL);                        } Processo padre

    }
    …
    exit(0);
}
```

A chi viene restituito 0
(chi e' il padre del main) ?



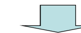
## Processi cooperanti

- Un processo è **indipendente** se non può influire su altri processi nel sistema o subirne l'influsso.
- Processi **cooperanti** possono influire su altri processi o esserne influenzati.
- La presenza o meno di dati condivisi determina univocamente la natura del processo.
- Vantaggi della cooperazione fra processi
  - Condivisione di informazioni
  - Accelerazione del calcolo (in sistemi multiprocessore)
  - Modularità
  - Convenienza



## Comunicazione tra processi (IPC)

Un sistema operativo e' composto da numerosi moduli che devono interagire tra loro



Necessita' di un meccanismo di comunicazione tra processi

- segnali
- Memoria condivisa
- scambio di messaggi

## segnali

- Interruzioni software per la comunicazione asincrona tra processi
  - Non permettono ai processi di scambiarsi dati
  - Il processo che riceve il segnale non e' in un particolare stato di attesa (evento asincrono)
  - I processi possono catturare, ignorare o mascherare un segnale
    - Catturare un segnale significa far eseguire al sistema operativo una specifica routine al momento della ricezione del segnale o una azione di default associata al segnale
    - Ignorare un segnale significa far eseguire al sistema operativo delle operazioni di default associate al segnale
    - Mascherare un segnale significa istruire il sistema operativo a non consegnare il segnale fino a nuovo ordine



## Scambio di messaggi / memoria condivisa



Mediante scambio di messaggi

Mediante memoria condivisa



## Esempio: produttore–consumatore

- È un paradigma classico per processi cooperanti. Il processo **produttore** produce informazioni che vengono consumate da un processo **consumatore**.
  - *Buffer illimitato:* non vengono posti limiti pratici alla dimensione del buffer.
  - *Buffer limitato:* si assume che la dimensione del buffer sia fissata.
- Esempio: Un programma di stampa produce caratteri che verranno consumati dal driver della stampante.



## Soluzione con buffer limitato e memoria condivisa

- Dati condivisi

```
#define BUFFER_SIZE 8
typedef struct {
    . . .
} item;
item buffer[BUFFER_SIZE];
int in = 0;
int out = 0;
```



- in    indice della prima posizione libera
- out   indice della prima posizione occupata

## osservazione

se il buffer e’ pieno il produttore si deve arrestare

se il buffer e’ vuoto il consumatore si deve arrestare

PROBLEMA

buffer pieno : non e’ possibile definire in
buffer vuoto : non e’ possibile definire out

E’ possibile utilizzare (BUFFER_SIZE-1) elementi
in maniera da poter definire sempre in e out



## Soluzione con buffer limitato e memoria condivisa

```
item nextProduced;
while (1) {
        while ( ((in + 1) % BUFFER_SIZE) == out ) ;
                /* buffer pieno: non fare niente */
        buffer[in] = nextProduced;
        in = (in + 1) % BUFFER_SIZE;
}
```

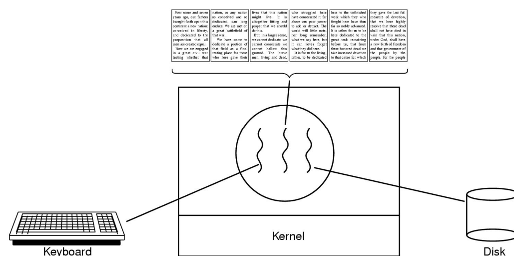
Processo Produttore

```
item nextConsumed;
while (1) {
        while (in == out) ;
                /* buffer vuoto: non fare niente */
        nextConsumed = buffer[out];
        out = (out + 1) % BUFFER_SIZE;
}
```

Processo Consumatore



## Problema: come e’ fatto un word processor



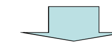

Molte azioni contemporanee e indipendenti (input, tabulazione, correzione ortografica, memorizzazione,..) sullo stesso insieme di dati !!



## Soluzioni

1: piu’ processi che comunicano attraverso una memoria condivisa
- ovrhead per la creazione di numerosi processi
- overhead per la gestione della memoria comune
- Condividere molti dati

2: un solo processo con “stringhe di esecuzioni” indipendenti che operano sullo stesso spazio di indirizzamento



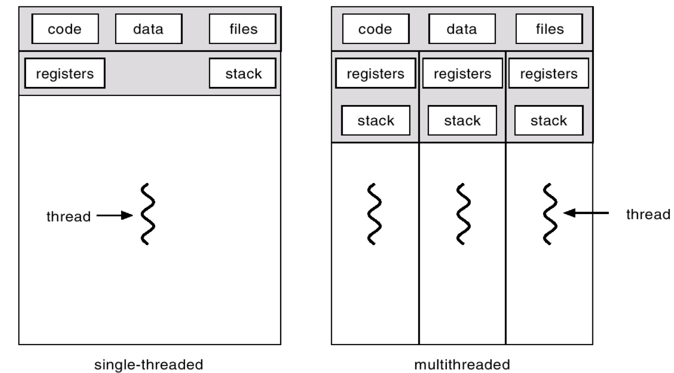
Stringhe di esecuzione
=
threads

## Definizione di Threads

- Nei moderni S.O. un ***thread*** (o *lightweight process,* **LWP**) è spesso l'unità di base di utilizzo della CPU e consiste di:
  - Program counter
  - Insieme dei registri
  - Spazio dello stack
- Un thread condivide con i thread ad esso associati:
  - Spazio di indirizzamento (non c'e' protezione !!)
  - Dati globali
  - File aperti
  - Gestori dei segnali
- L'insieme dei thread e dell'ambiente da essi condiviso è chiamato *task*.
- Un processo tradizionale, o *heavyweight,* corrisponde ad un task con un solo thread.



## Processi a thread singolo e multithread






## Vantaggi dei threads

- In un task multithread, mentre un thread è blocccato in attesa, un secondo thread nello stesso task può essere in esecuzione.
  - La cooperazione di più thread nello stesso job fornisce un maggior throughput.
  - Applicazioni che richiedono la condivisione di un buffer (es. produttore–consumatore) traggono beneficio dall'impiego di thread.

- Possono essere gestiti dal sistema operativo o da una applicazione utente
- E' un modo per condividere risorse
- Realizzano una forma di parallelismo
- Hanno un overhead molto inferiore a quelli dei processi



## Ciclo di vita di un thread

- Analogamente ad un processo i threads sono schedulati da uno scheduler e attraversano varie fasi:
  - *nuovo*
  - *pronto*
  - *esecuzione*
  - *terminato*
  - *bloccato*
  - *In attesa*
  - *dormiente*

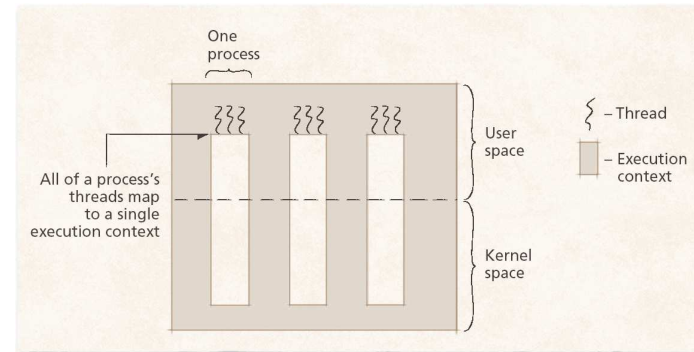
*… ma come sempre dipende dall'implementazione*

## Modelli di implementazioni

- Quasi tutti i sistemi operativi implementano i threads in uno dei seguenti modi
  - User-level threads (modello "molti a uno")
    - i threads sono creati e gestiti nello spazio utenti da specifiche librerie che non possono eseguire istruzioni privilegiate o accedere alle primitive del kernel direttamente
  - Kernel-level threads (modello "uno a uno")
    - i threads sono creati e gestiti direttamente dal kernel
    - In questo caso i thread rappresentano l' "unita' di esecuzione" della CPU
  - Combinazione ibrida delle due modalita' precedenti
    - Modello "molti a molti"

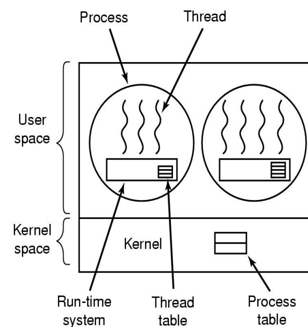


## User-level Threads






## Implementazione del modello ULT




Il kernel mantiene la propria process table e i relativi PCB

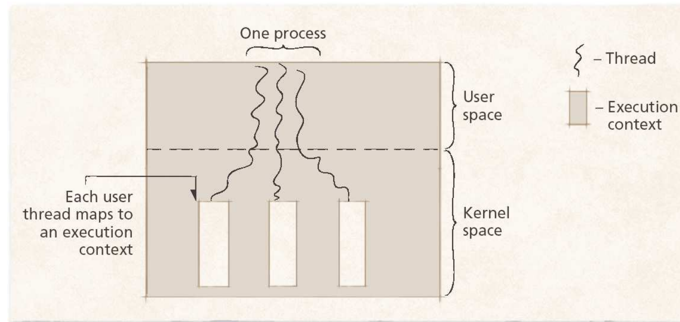
Ogni processo ha una propria thread table con relative strutture dati per la descrizione dei thread analoga (ma di dimensioni ridotte) ai PCB



## Thread a livello utente (ULT)

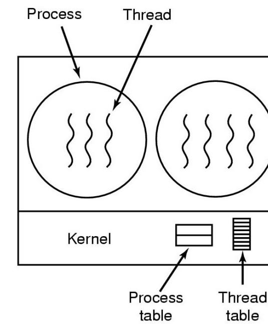
- Vantaggi:
  - Il cambio di contesto fra thread non richiede privilegi in modalità kernel (risparmia il sovraccarico del doppio cambiamento di modalità).
  - Lo scheduling può essere diverso per applicazioni diverse.
  - Gli **ULT** (*User Level Thread* ) possono essere eseguiti su qualunque SO senza cambiare il kernel sottostante. La libreria dei thread è un insieme di utilità a livello di applicazione.
- Svantaggi:
  - In caso di system call bloccanti, quando un thread esegue una chiamata di sistema, viene bloccato tutto il processo.
  - Un'applicazione multithread non può sfruttare il multiprocessing: in un dato istante un solo thread per processo è in esecuzione.

## Kernel-level Threads






## Implementazione del modello KLT




Il kernel mantiene
- la process table
  - i relativi PCB
- una thread table
- le relative strutture dati



## Thread a livello kernel (KLT)

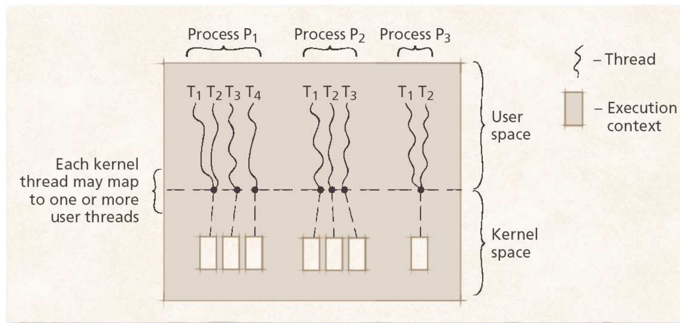
- Vantaggi:
  - Può schedulare simultaneamente più thread (predisposizione al parallelismo su multiprocessori)
  - Miglioramento della scalabilita' e dell'interattivita'
  - Se un thread di un processo è bloccato il kernel può schedulare un altro thread dello stesso processo.
- Svantaggi:
  - Il trasferimento del controllo fra thread dello stesso processo richiede il passaggio in modalità kernel: l'aumento di prestazioni è meno rilevante rispetto all'approccio ULT.
  - Sovraccarico del kernel che potrebbe gestire migliaia di threads (e delle relative strutture dati)



## Combinazione ibrida

- Cerca di combinare i vantaggi dei due precedenti approcci
- Numero dei threads utente e threads kernel non uguale
- Puo' ridurre l'overhread dell'approccio uno-a-uno
- Realizzazione:
  - Un insieme di thread permanenti (worker threads) vengono creati dal kernel e formano il "threads pool"
  - Ogni nuovo thread utente e' eseguito da un worker thread
- Ottimizzazione
  - possibilita' di specificare il numero dei worker threads in base al carico del sistema
  - Schedulazione dei thread direttamente nel kernel
- Svantaggi
  - Complicazione per il sistema operativo
  - Difficolta' a determinare il numero di worker thread

## Combinazione ibrida






## Threads POSIX (Pthreads)

- Uno standard POSIX (IEEE 1003.1c) per la creazione e sincronizzazione dei threads
- Definizione delle API
- Threads conformi allo standard POSIX sono chiamati Pthreads
- Lo standard POSIX stabilisce che registri dei processori, stack e signal mask sono individuali per ogni thread
- Lo standard specifica come il sistema operativo dovrebbe gestire i segnali ai Pthtrads i specifica differenti metodi di cancellazione (asincrona, ritardata, …)
- Permette di definire politiche discheduling e priorita’
- Alla base di numerose librerie di supporto per vari sistemi



## Esercizi

PROCESSI E THREADS

- Chiamate di sistema LINUX per la gestione dei processi
  - Libreria pthread (threads Posix)

## Per incominciare….

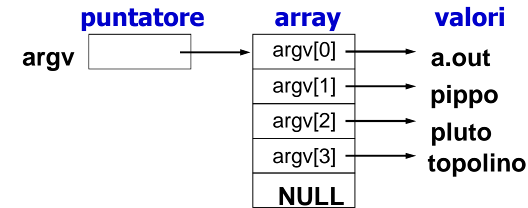
```
#include <stdio.h>

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

   for (i = 0; i < argc; i++)
                    printf("argv[%d]: %s\n", i, argv[i] );

   exit(0);
}
```

## Esempio (cont.)

```
[lapegna%] cc program.c
[lapegna%] a.out  pippo  pluto  topolino
argv[0]: a.out
argv[1]: pippo
argv[2]: pluto
argv[3]: topolino
[lapegna%]
```

Marco Lapegna –
Laboratorio di Programmazione 2
introd ai Sist Op

## argc e argv




argc  e' un intero (numero di argomenti sulla linea di comando)

argv e'  un array di puntatori a carattere (gli argomenti sulla linea di comando)

Marco Lapegna –
Laboratorio di Programmazione 2
introd ai Sist Op

## Identificativi di processo

Ogni processo e' identificato univocamente da un intero

```
[lapegna%] ps
   PID  TTY         TIME   CMD
11949  pts/0     00:00:00   bash
12031  pts/0     00:00:00   ps
[lapegna%]
```

Marco Lapegna –
Laboratorio di Programmazione 2
introd ai Sist Op

## Primitive  per la gestione  dei processi

Unix fornisce le primitive per la creazione e terminazione di  processi, e per l'esecuzione di programmi:

- **fork** crea nuovi processi;
- **exec** attiva nuovi programmi;
- **exit** termina  un processo.
- **wait** attende la terminazione di un processo.

## Funzione fork

```
# include <unistd.h>
pid_t   fork(void);
```

L'unico modo per istruire il kernel a creare un nuovo processo è di chiamare la funzione **fork** da un processo esistente.

Il processo creato viene detto figlio. Il processo che chiama **fork** viene detto genitore

Il figlio è una copia del genitore (data space, heap e stack), cioè essi non condividono parti di memoria.

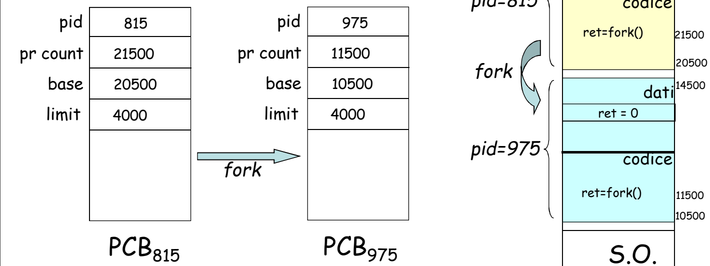


## Funzione fork

La funzione ritorna :

- l'identificativo del figlio nel processo genitore,
- 0 nel processo figlio

Genitore e figlio riprendono l'esecuzione dall'istruzione successiva al fork






## Come differenziare il padre dal figlio?

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

int main () {
  int pid;

  if( (pid=fork( ))==0)     /* sono il figlio */

            printf("Ciao, sono il figlio\n");

  else                      /* sono il padre */

            printf("Ciao, sono il padre\n");

  exit(0);
}
```



## Esempio (esecuzione)

```
[lapegna%] cc  –o  eseg   program.c
[lapegna%] eseg
Ciao sono il padre
Ciao sono il figlio
[lapegna%]
```

## Identificativi dei processi

```
# include <unistd.h>
# include <sys/types.h>
pid_t  getpid(void);        identificativo di processo
pid_t  getppid(void);       identificativo del padre
```

Queste funzioni ritornano
gli identificativi di processo.



## esempio

```
# include <stdio.h>
# include <unistd.h>
# include <sys/types.h>


int main ( ) {
  pid_t pid;

  if( (pid=fork( )) ==0)    /* sono il figlio */

    printf(“F: miopid = %d , pidpadre = %d \n“, getpid( ), getppid( ) );

  else                      /* sono il padre */

    printf(“P: miopid = %d , pidfiglio = %d \n“, getpid( ), pid );

  exit(0);
}
```



## Esempio (esecuzione)

```
[lapegna%] cc –o  eseg   program.c
[lapegna%] eseg
F: miopid = 428 , pidpadre = 426
P: miopid = 426 , pidfiglio = 428
[lapegna%]
```



## Esercizio:

```
# include <stdio.h>
# include <unistd.h>
# include <sys/types.h>

int main ( ) {
pid_t   pid;
int  i;

   for ( i=0 ; i<3 ; i++) {

          pid=fork( );

   }
   printf(“ processo %d terminato\n”, getpid() );

  exit(0);
}
```

Quanti processi vengono generati?

## Esecuzione

```
[lapegna%] cc -o eseg  program.c
[lapegna%] eseg
processo 2167 terminato
processo 2170 terminato
processo 2163 terminato
processo 2164 terminato
processo 2169 terminato
processo 2166 terminato
processo 2168 terminato
processo 2165 terminato
[lapegna%]
```

8 processi !!!



## infatti

$P_1$ fork → $P_2$; fork → $P_3$; fork → $P_4$
$P_3$ fork → $P_5$; fork → $P_6$; fork → $P_7$; fork → $P_8$

In generale $2^n$ processi !!!



## Funzione exit

**exit** termina un programma normalmente e libera le risorse

```
# include <stdio.h>

void exit(int status);
```

La funzione ha per argomento un intero che rappresenta il valore di ritorno del programma

Tale valore non è definito quando:
1. la funzione viene chiamata senza argomento,
2. se il programma termina prima del dovuto.



## Comunicazione padre - figlio

Un figlio che termina normalmente comunica al genitore il suo valore di uscita mediante l'argomento della funzione exit

Nel caso di terminazione non normale, il kernel genera uno stato di terminazione per indicare la ragione.

Se il genitore termina prima del figlio, il processo **init** eredita il figlio e il parent process ID di questo diventa 1.

Se il figlio termina prima che il genitore sia in grado di controllare la sua terminazione, il kernel conserva almeno il process ID e lo stato di terminazione.
Tali processi vengono detti zombie.

## esempio

```
# include <stdio.h>
# include <unistd.h>
# include <sys/types.h>


int main ( )  {
   pid_t pid;

   if( (pid=fork( )) ==0) {    /* sono il figlio */

     sleep(1);
     printf("F: miopid = %d , pidpadre = %d \n", getpid( ), getppid( ) );

   }else {                     /* sono il padre */

      printf("P: miopid = %d , pidfiglio = %d \n", getpid( ), pid );

  }
   exit(0);
}
```



## esempio

```
[lapegna%] cc  –o  eseg   program.c
[lapegna%] eseg
P: miopid = 723 , pidfiglio = 724
F: miopid = 724 , pidpadre = 1
[lapegna%]
```



## Funzione wait

Il genitore è in grado di ottenere lo stato di terminazione di un figlio mediante la funzione **wait**

```
# include <sys/types.h>
# include <sys/wait.h>

pid_t wait(int *statloc)
```

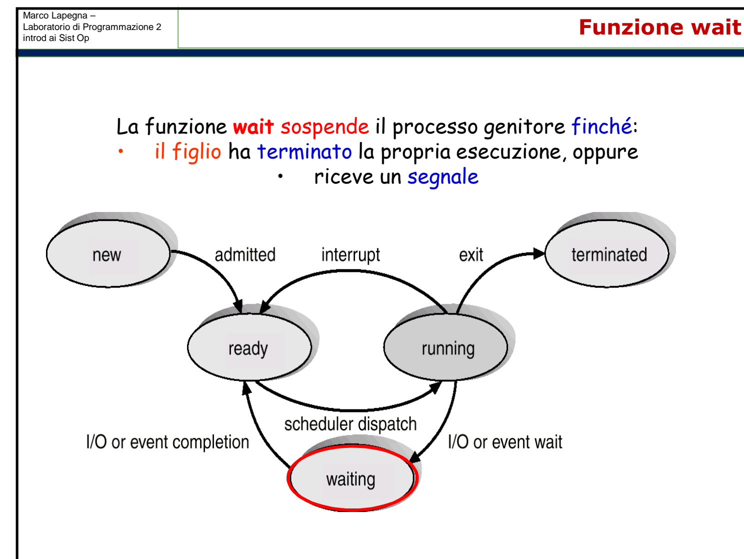
l'argomento *statloc* é un puntatore ad un intero.

In tale intero sono codificati
- lo stato di terminazione
- il valore di ritorno del processo figlio (comunicato da exit)



## Funzione wait

La funzione **wait** sospende il processo genitore finché:
- il figlio ha terminato la propria esecuzione, oppure
- riceve un segnale

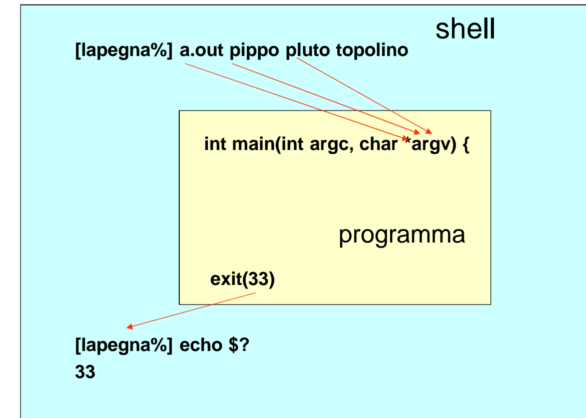


## Esempio: Shell e main

- un programma C e' un insieme di funzioni che si richiamano l'un l'altra
- una (e solo una) deve chiamarsi main, ed e' la funzione dalla quale inizia l'esecuzione
- la shell crea un nuovo processo per ogni comando che riceve
- la shell passa gli argomenti alla funzione main e da essa riceve i valori di ritorno



## Shell e processi






## Stato di terminazione

Le macro definite in <sys/wait.h> interrogano la variabile status

| Macro | Descrizione |
|---|---|
| WIFEXITED(*status*) | vero se il figlio è terminato normalmente<br>**WEXITSTATUS(*status*) ritorna lo stato** |
| WIFSIGNALED(*status*) | vero se il figlio è uscito per un segnale<br>**WTERMSIG(*status*) ritorna il segnale** |
| WIFSTOPPED(*status*) | vero se il figlio è fermato<br>**WSTOPSIG(*status*) ritorna il segnale** |



## esempio

```
# include <sys/types.h>
# include <sys/wait.h>
# include <stdio.h>
# include <unistd.h>

int main(void) {
    pid_t  pid;
    int    status, a, b, c;

    if ( (pid = fork()) == 0)    {                    /* figlio */

                printf("dammi a e b --> ");
                scanf("%d %d", &a, &b);
                c = a+b;
                exit(0);

        }else{                                        /* padre */

                wait(&status) ;     /* aspetta il figlio */
                if ( WIFEXITED(status))
                    printf(" ritorno dal figlio = %d \n", WEXITSTATUS(status));
                else
                    printf("figlio non terminato correttamente\n");

        }
}
```

## esempio

```
[lapegna%] cc  –o  eseg   program.c
[lapegna%] eseg
dammi a e b -->8 9
ritorno dal figlio = 0
[lapegna%]
```



## Funzione execl

La funzione **execl** sostituisce il codice e i dati del programma in esecuzione con il codice e i dati di un nuovo programma all'interno dello stesso processo

```
# include <unistd.h>
int execl   ( const char *path, const char *arg0, ... , NULL );
```

nuovo programma (path) — argomenti del programma (arg0, ...)

L'identificativo del nuovo processo è lo stesso di quello sostituito.



## Esempio

Programma "hello"

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

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

    printf("hello! Il mio pid = %d, getpid() );
    printf("argomenti = %s %s \n", argv[1], argv[2]);

  exit(0);
}
```



## Esempio (cont)

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

int main ()  {
  int pid;

  if( (pid=fork( ))==0)     /* sono il figlio */
    printf("Ciao, sono il figlio, pid = %d \n", getpid());
    execl("/home/lapegna/hello", "hello", "pippo", "pluto", NULL);

  else                       /* sono il padre */
    printf("Ciao, sono il padre\n");

  exit(0);
}
```

## Esempio (esecuzione)

```
[lapegna%] a.out
Ciao sono il figlio, pid = 11543
Ciao sono il padre
hello, il mio pid = 11543
argomenti = pippo pluto
[lapegna%]
```



## Threads POSIX (Pthreads)

- Uno standard POSIX (IEEE 1003.1c) per la creazione e sincronizzazione dei threads
- Definizione delle API
- Threads conformi allo standard POSIX sono chiamati Pthreads
- Lo standard POSIX stabilisce che registri dei processori, stack e signal mask sono individuali per ogni thread
- Lo standard specifica come il sistema operativo dovrebbe gestire i segnali ai Pthtrads i specifica differenti metodi di cancellazione (asincrona, ritardata, …)
- Permette di definire politiche discheduling e priorita'
- Alla base di numerose librerie di supporto per vari sistemi



## Creazione di un thread

```
#include <pthread.h>
int pthread_create( pthread_t  *tid,
                    const pthread_attr_t  *attr,
                    void *(*func)(void*),
                    void *arg );
```

**tid**: puntatore all'identificativo del thread ritornato dalla funzione

**attr**: attributi del thread (priorita', dimensione stack, …) . A meno di esigenze particolari si usa NULL

**func**: funzione di tipo **void *** che costituisce il corpo del thread

**arg** : unico argomento di tipo **void *** passato a **func**

La funzione ritorna subito dopo creato il thread



## Attesa per la fine di un thread

```
#include <pthread.h>

int pthread_join( pthread_t  tid,
                  void ** status );
```

**tid**: identificativo del thread di cui attendere la fine

**status**: puntatore al valore di ritorno del thread. Di solito NULL

## Processi vs Pthread

| | Processi | Threads |
|---|---|---|
| creazione | **fork+exec** | **pthread_create** |
| attesa | **wait** | **pthread_join** |

Altre funzioni

```
int pthread_detach(pthread_t thread_id);
int pthread_exit(void *value_ptr);
pthread_t pthread_self();
```



## Esercizio: sia dato il seguente codice

5bis esercizi su processi e threads

```
int value =0;
Void *runner(void *param);

int main(int argc, char *argv[]){
int pid;   pthread_t tid;   pthread_attr_t attr;

pid=fork( );
if (pid == 0){
    ......
    pthread_create(&tid, &attr, runner, NULL);
    pthread_join(tid, NULL);
    printf("proc figlio value =%d\n", value);

}else{
    wait(NULL);
    printf("proc padre value =%d\n", value);
}
}


void *runner(void *param){
     value = 5;
     pthread_exit(0);
}
```

Quali sono i dati in output?



## soluzione

proc figlio value = 5
proc padre value = 0

Perche?

al momento della fork viene duplicato lospazio di indirizzamento per il figlio e il thread in esso contenuto modifica solo la sua copia.

nel padre resta value = 0



## Esempio

Realizzare un programma con
2 threads tale che

- thread scrivi incrementa una variabile condivisa *shareddata* da 1 a 5, aspettando 1 secondo tra un incremento e l'altro
- thread leggi legge e stampa il contenuto di *shareddata* solo quando il thread scrivi ha modificato la variabile condivisa
- Far terminare il thread leggi quando termina il thread scrivi

## Multithreading: main

```
#include<pthread.h>
#include<stdio.h>
#include<stdlib.h>
struct  dati{
int    shareddata  ;
int   finethread;
};    // dichiarazione tipo

int main( ){
void *scrivi(void *), *leggi(void *);
struct dati  datiglobali;
pthread_t    tid_scrivi,   tid_leggi;
datiglobali.shareddata=0;
datiglobali.finethread=0;
pthread_create(&tid_scrivi, NULL, scrivi, &datiglobali);
pthread_create(&tid_leggi, NULL, leggi, &datiglobali);
pthread_join(tid_scrivi, NULL);
pthread_join(tid_leggi, NULL);

}  //fine main
```



## Thread  scrivi

```
void *scrivi (void *arg){
int   i;
struct dati *arglocale;
arglocale = (struct dati *) arg;

printf(" partito  thread  scrivi\n");
for (i=0;  i<5  ; i++){

      sleep(1);
      arglocale->shareddata = (arglocale->shareddata)+1;
      printf("thread scrivi: shareddata = %d \n",arglocale->shareddata);

}
sleep(1);
arglocale->finethread = 1;
}
```



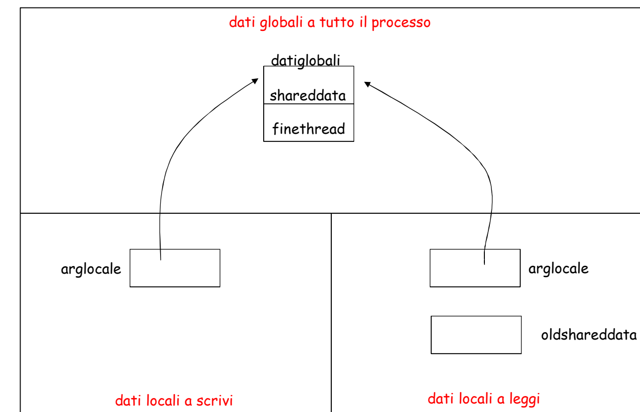
## thread leggi

```
void *leggi (void *arg){
struct dati *arglocale;
arglocale = (struct dati *) arg;
int oldshareddata=0;

printf(" partito  thread  leggi\n");
while  (arglocale->finethread==0        ){
      while(arglocale->shareddata == oldshareddata && arglocale->finethread==0);
      if( arglocale->finethread ==0 ){
            printf("thread leggi: shareddata = %d \n",arglocale->shareddata);
            oldshareddata = arglocale->shareddata;
      }
}
```

N.B. compilare il tutto linkando la libreria libpthread.a



## schema dell'utilizzo della memoria



## Esempio (esecuzione)

```
[lapegna%] a.out
 partito  thread  scrivi
 partito  thread  leggi
thread scrivi: shareddata = 1
thread leggi: shareddata = 1
thread scrivi: shareddata = 2
thread leggi: shareddata = 2
thread scrivi: shareddata = 3
thread leggi: shareddata = 3
thread scrivi: shareddata = 4
thread leggi: shareddata = 4
thread scrivi: shareddata = 5
thread leggi: shareddata = 5
[lapegna%]
```