http://matteo.vaccari.name/so/

Processo

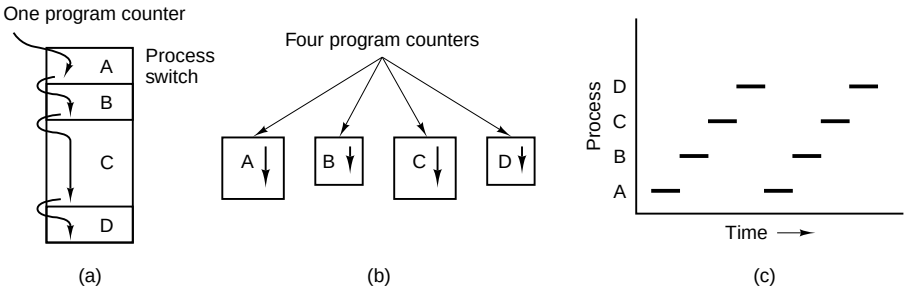
Definizione: un *processo* è un programma in esecuzione

Un programma è una lista di istruzioni (statica)

Un processo è *dinamico*: il suo stato cambia nel tempo

Pseudo parallelismo

Una sola CPU viene suddivisa rapidamente fra n processi



Context-switch

Avviene un *context-switch* quando il SO interrompe l'esecuzione di un processo e riprende l'esecuzione di un altro processo

Operazione *invisibile* per il processo

Realizzata in assembler

0. salva i registri del processo A
1. carica i registri del processo B
2. per ultimo, carica il registro PC del processo B

Diversi tipi di SO

Nei SO convenzionali:
un processo riceve la CPU dopo un tempo non predicibile
$\Rightarrow$ i programmi *non possono fare assunzioni sul tempo*

Nei SO *real-time*:
un processo riceve la CPU entro un tempo ben definito
(ordine di grandezza: da millisecondi a centinaia di ms)

Compito del SO

fornire i mezzi per

- creare e distruggere processi
- sospendere l'esecuzione di un processo e riprenderla
- sincronizzare processi
- mettere in comunicazione i processi

In UNIX

Ciascun processo ha un process identifier (pid)

Un pid è un intero positivo di 16 bit

```
pid_t getpid(void);
```

Anche in Win32 esistono i process id

Creazione di processi

- quando il sistema è inizializzato
- su richiesta di un processo esistente
- su richiesta di un utente
- batch jobs

in ogni caso si arriva alla *esecuzione di una system call* per creare il nuovo processo

In Unix: fork(2) + exec(2)

`pid_t fork(void);` crea una copia identica del processo corrente

unica differenza fra i due processi:
- hanno pid diverso
- il valore restituito da fork è diverso

```
pid_t pid = fork();
if (-1 == pid) {
  // errore!
}
if (0 == pid) {
  // processo figlio; usa getpid(2) per avere il pid
} else {
  // processo genitore; ‘‘pid’’ contiene il pid del figlio
}
```


`int execv(const char *path, char *const argv[]);`

The exec family of functions replaces the current process image with a new process image — manuale di exec(2)

```
pid_t pid = fork();
if (-1 == pid) { /* errore! */ }
if (0 == pid) {
   // processo figlio
   char *argv[] = { "/bin/ls", "-l", NULL };
   execv("/bin/ls", argv);
   // se siamo qui la execv ha fallito!
} else {
   // processo genitore
}
```


Terminazione di processi

- terminazione normale (volontaria)
- terminazione per errore (volontaria)
- fatal error (non volontario)
- uccisione da parte di un altro processo (non volontaria)

Unix:
`void _exit(int status);`
system call; termina il processo corrente

`void exit(int status);`
libreria C; chiama `_exit(2)`

`int kill(pid_t pid, int sig);`
system call; manda un “segnale” a un processo

Terminazione

normale:

```
main () {
   return 0;
}
```

oppure:

```
void f() {
    _exit(0);
}
main () {
   f();
}
```

per errore: (qualsiasi numero != 0)

```
main () {
   return 1;
}
```

oppure:

```
void f() {
    _exit(1);
}
main () {
   f();
}
```

Questo programma termina con uno stato non determinato

```
main () { }
```


Attendere la terminazione del processo figlio

```
pid_t wait(int *status);
```

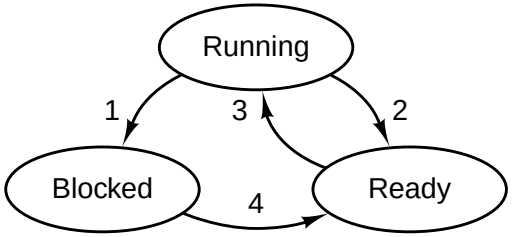
The wait function suspends execution of the current process until a child has exited — manuale di wait(2)

```
pid_t pid = fork();
if (-1 == pid) { /* errore! */ }
if (0 == pid) {
   char *argv[] = { "/bin/ls", "-l", NULL };
   execv("/bin/ls", argv);
} else {
   int status;
   pid_t p = wait(&status);
   printf("child %d terminated with status %d\n", p, status);
}
```


La shell di Unix *molto* semplificata

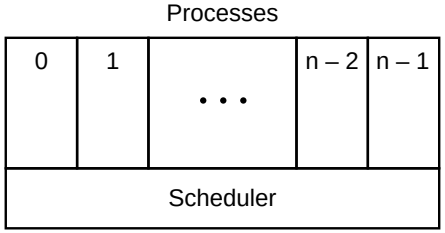
```
while (1) {
  type_prompt();
  read_command(command, parameters);
  pid_t pid = fork();
  if (-1 == pid) { /* errore! */ }
  if (0 == pid) {
    execv(command, parameters);
  } else {
    int status;
    wait(&status);
  }
}
```


Stati di un processo



1. Process blocks for input
2. Scheduler picks another p
3. Scheduler picks this proc
4. Input becomes available

Lo scheduler



è lo strato più "basso" del SO

gestisce interrupt, scheduling

sopra questo strato i processi sono sequenziali

Implementazione dei processi

per ogni processo c'è un *process control block* (PCB)

- stato
- pid, ppid, ...
- uid, gid, ...
- cwd
- file descriptors
- contatori
- registri
- ...

al centro del SO c'è la *process table*
(array o lista di PCB)

PCB di Linux versione 0.01; dal file include/linux/sched.h

```
struct task_struct {
        long state;     /* -1 unrunnable, 0 runnable, >0 stopped */
        long counter;
        long priority;
        long signal;
        fn_ptr sig_restorer;
        fn_ptr sig_fn[32];
/* various fields */
        int exit_code;
        unsigned long end_code,end_data,brk,start_stack;
        long pid,father,pgrp,session,leader;
        unsigned short uid,euid,suid;
        unsigned short gid,egid,sgid;
        long alarm;
        long utime,stime,cutime,cstime,start_time;
        unsigned short used_math;
        ...
```


PCB di Linux versione 0.01, cont.

```
        ...
/* file system info */
        int tty;                /* -1 if no tty, so it must be signed */
        unsigned short umask;
        struct m_inode * pwd;
        struct m_inode * root;
        unsigned long close_on_exec;
        struct file * filp[NR_OPEN];
/* ldt for this task 0 - zero 1 - cs 2 - ds&ss */
        struct desc_struct ldt[3];
/* tss for this task */
        struct tss_struct tss;
};
```

27 righe commenti inclusi; sono 134 in Linux 2.4.18

Altra definizione di processo

Un processo è
- un *filo di esecuzione* (thread)
- e una collezione di risorse
  - uno spazio di indirizzamento (memoria)
  - file aperti
  - ...

Gerarchie di processi

Un processo crea un altro processo che crea altri processi...

formano una gerarchia

in molti SO esiste un legame fra processo genitore e figlio

in Unix il figlio eredita tutti i *file descriptor* del genitore

Gerarchie di processi in UNIX, cont.

`pid_t getppid(void)`
restituisce il pid del processo genitore

I processi sono raggruppati in *process groups*

`pid_t getpgrp(void)`
restituisce il process group id

`int setpgid(pid_t pid, pid_t pgid)` cambia il process group del processo pid

Lo scopo dei process groups

Process groups are used for distribution of signals, [...] These calls are thus used by programs such as csh(1) to create process groups in implementing job control.
— manuale di getpgrp(2)

in poche parole: servono per mandare un segnale a un insieme di processi

sono usati dalla shell per controllare una pipeline

Gerarchie di processi in Win32

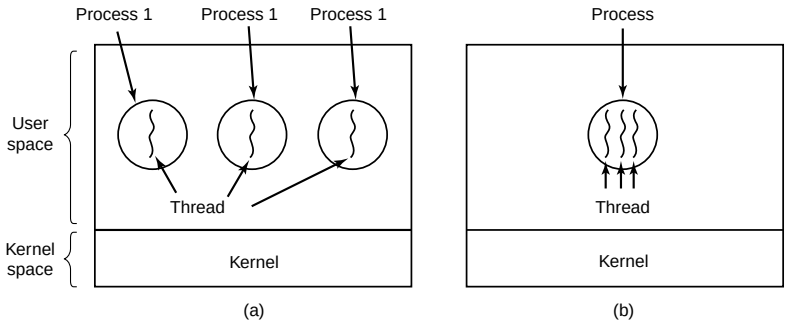
Handle numero che identifica un *Win32 kernel object*

(Esempi di kernel objects: file aperti, pipe, socket, semafori...)

Ciascun processo possiede un insieme di handle

Certe handle sono ereditabili dai processi figli
⇒ posso restringere l'accesso a una risorsa ai soli figli del processo che l'ha creata

Threads



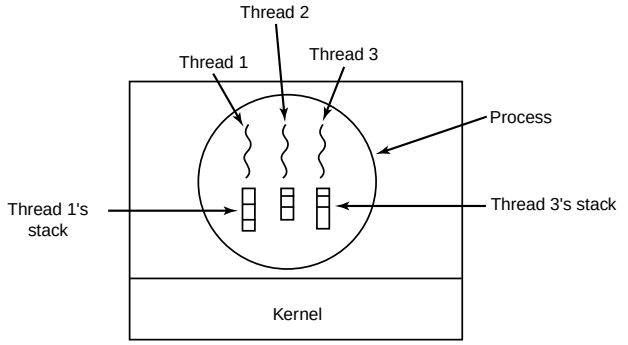
(a) tre processi con un thread ciascuno
(b) un processo con tre thread

Differenza fra thread e processo

Due thread all'interno dello stesso processo
condividono quasi tutte le risorse del processo

risorse condivise	**risorse private del thread**
spazio di indirizzamento	program counter
var globali	registri CPU
file aperti	stack
processi figli	stato (running, blocked, ready)
informazioni di accounting	
... tutto il resto	

Threads



Ciascun thread ha il suo stack

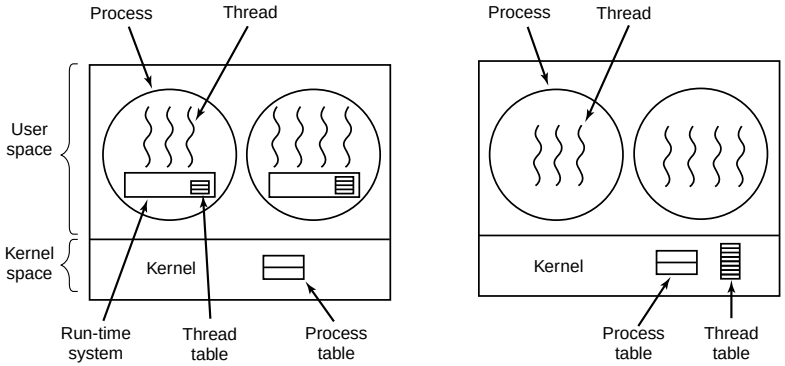
Due maniere di implementare i thread

supportati dal kernel (Windows NT/95/98, Mach, Linux)

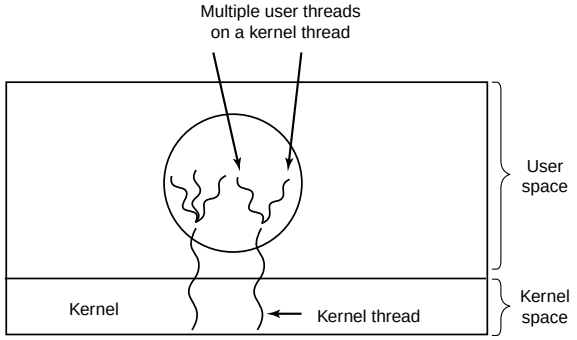
implementati in spazio utente (Java Virtual Machine, Ada, Posix Threads,...)

ibrido: thread sia nel kernel che in spazio utente (Solaris)

Implementare i thread in spazio utente vs. nel kernel



Implementazione ibrida



Perché scrivere applicazioni multithread?

Decomporre più attività concorrenti in più thread sequenziali
⇒ più facili da programmare (???)

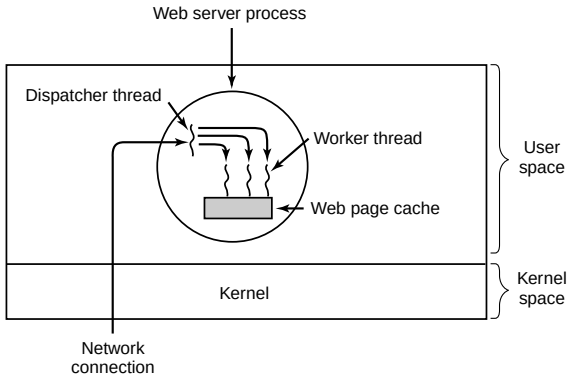
- esempio: graphical user interface
- esempio: server di rete

È più veloce creare e distruggere un thread che un processo

Maggiore throughput: un thread può fare system call bloccanti ma gli altri continuano a eseguire

Sfruttare più di una CPU

Multi-threaded web server



Multi-threaded web server

Dispatcher thread

```
while (1) {
  get_next_request(&buf);
  create_thread(do_work, buf)
}
```

Worker thread

```
do_work(char * buf) {
  // ... restituisci una pagina
  // all'utente
}
```


Maniere di costruire un server

Processo single-thread	nessun parallelismo, chiamate bloccanti; facile da implementare; pessimo throughput, pessima risposta interattiva
Threads	parallelismo, chiamate bloccanti
Macchina a stati	parallelismo, chiamate non bloccanti; molto difficile da implementare, potenzialmente massimo throughput
Pre-forking	parallelismo, chiamate bloccanti (su più processi) es. Apache Http Server 1.x