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

Processi

```
$ ps
  PID  TT  STAT      TIME COMMAND
  407  p3  S      0:00.04 -bash
  416  p3  R+     4:09.95 ruby script/server
15429  p8  S+     0:00.37 -bash
16106  p8  S      0:11.74 xpdf slides01.pdf
16332  pb  S      0:00.02 -bash

$ ps ax
  PID  TT  STAT      TIME COMMAND
    1  ??  S<s    0:06.02 /sbin/launchd
   26  ??  Ss     0:00.61 /sbin/dynamic_pager -F /private/var/vm
   30  ??  Ss     0:09.90 kextd
   67  ??  Ss    11:36.16 /usr/sbin/configd
   68  ??  Ss     0:09.67 /usr/sbin/coreaudiod
   69  ??  Ss     0:05.14 /usr/sbin/diskarbitrationd
   70  ??  Ss     0:00.19 /usr/sbin/memberd -x
   71  ??  Ss     0:03.15 /usr/sbin/securityd
  ...
```

Un processo

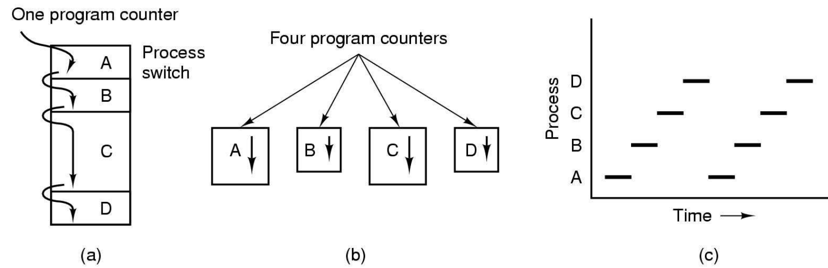
Processo — un programma in esecuzione

Un processo contiene:

- program counter
- stack
- data section

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);`

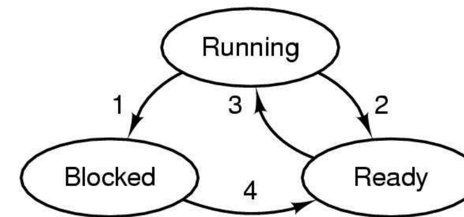
The wait function suspends execution of the current process until a child has exited — manuale di wait(2)

```
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

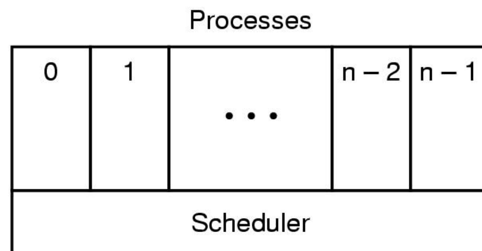
```
while (1) {
  type_prompt();
  read_command(command, parameters);
  pid_t pid = fork();
  if (-1 == pid) { /* errore! */ }
  if (0 == pid) {
    execv(command, parameters);
  } else {
    int status;
    wait(&status);
  }
}
```

Stati di un processo



1. Process blocks for input
2. Scheduler picks another proc
3. Scheduler picks this process
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 Minix; dal file kernel/proc.h I

```
struct proc {
  struct stackframe_s p_reg;  /* process' registers saved in stack fram

  proc_nr_t p_nr;             /* number of this process (for fast acces
  struct priv *p_priv;        /* system privileges structure */
  char p_rts_flags;           /* SENDING, RECEIVING, etc. */

  char p_misc_flags;          /* Flags that do suspend the process */

  char p_priority;            /* current scheduling priority */
  char p_max_priority;        /* maximum scheduling priority */
  char p_ticks_left;          /* number of scheduling ticks left */
  char p_quantum_size;        /* quantum size in ticks */

  struct mem_map p_memmap[NR_LOCAL_SEGS];   /* memory map (T, D, S) */

  clock_t p_user_time;        /* user time in ticks */
  clock_t p_sys_time;         /* sys time in ticks */
```

PCB di Minix; dal file kernel/proc.h II

```
  struct proc *p_nextready;   /* pointer to next ready process */
  struct proc *p_caller_q;    /* head of list of procs wishing to send
  struct proc *p_q_link;      /* link to next proc wishing to send */
  message *p_messbuf;         /* pointer to passed message buffer */
  proc_nr_t p_getfrom;        /* from whom does process want to receive
  proc_nr_t p_sendto;         /* to whom does process want to send? */

  sigset_t p_pending;         /* bit map for pending kernel signals */

  char p_name[P_NAME_LEN];    /* name of the process, including null */
};
```

Flag associati a un processo in Minix

```
/* Bits for the runtime flags. A process is runnable iff p_rts_f
#define SLOT_FREE 0x01    /* process slot is free */
#define NO_MAP    0x02    /* keeps unmapped forked child from ru
#define SENDING   0x04    /* process blocked trying to SEND */
#define RECEIVING 0x08    /* process blocked trying to RECEIVE *
#define SIGNALED  0x10    /* set when new kernel signal arrives
#define SIG_PENDING 0x20  /* unready while signal being processe
#define P_STOP    0x40    /* set when process is being traced */
#define NO_PRIV   0x80    /* keep forked system process from run
```

La tabella dei processi in Minix

```
EXTERN struct proc proc[NR_TASKS + NR_PROCS]; /* process tal
```

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

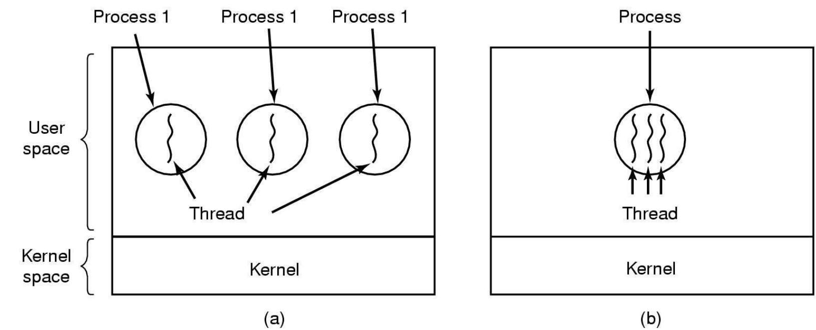
Handle numero che identifica un *Win32 kernel object*

(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



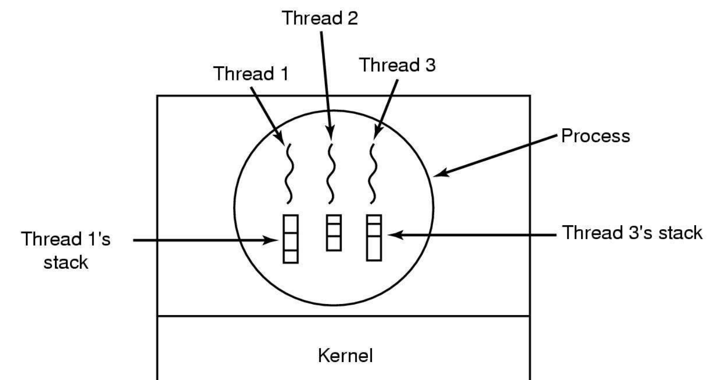
(a) tre processi con un thread ciascuno
(b) un processo con tre thread

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

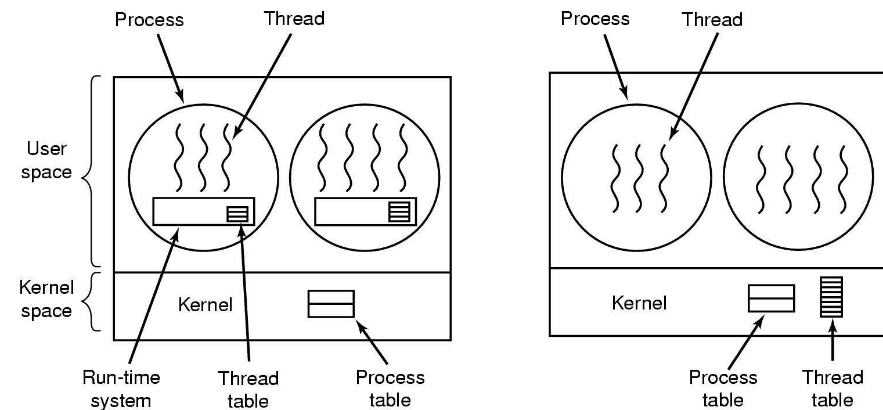
Due maniere di implementare i thread

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

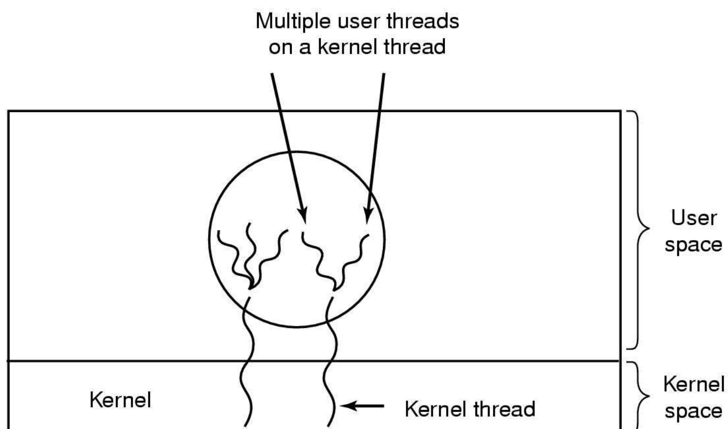
implementati in spazio utente (Java Virtual Machine, Ada, Posix Threads,...)

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 (???)

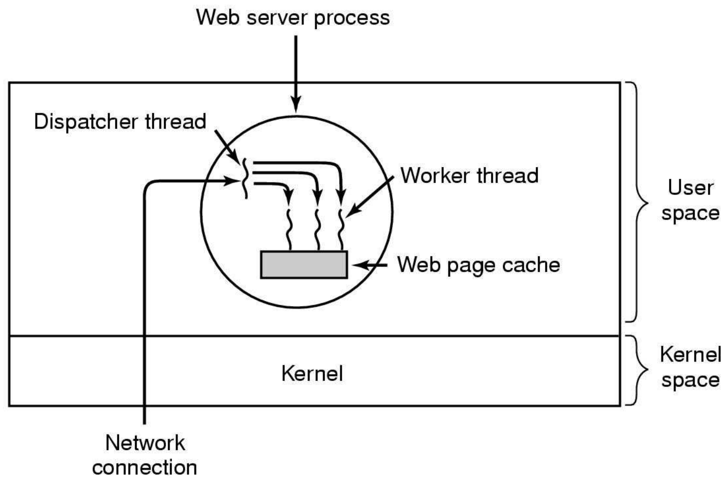
- esempio: graphical user interface
- esempio: server di rete

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



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

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
}
```