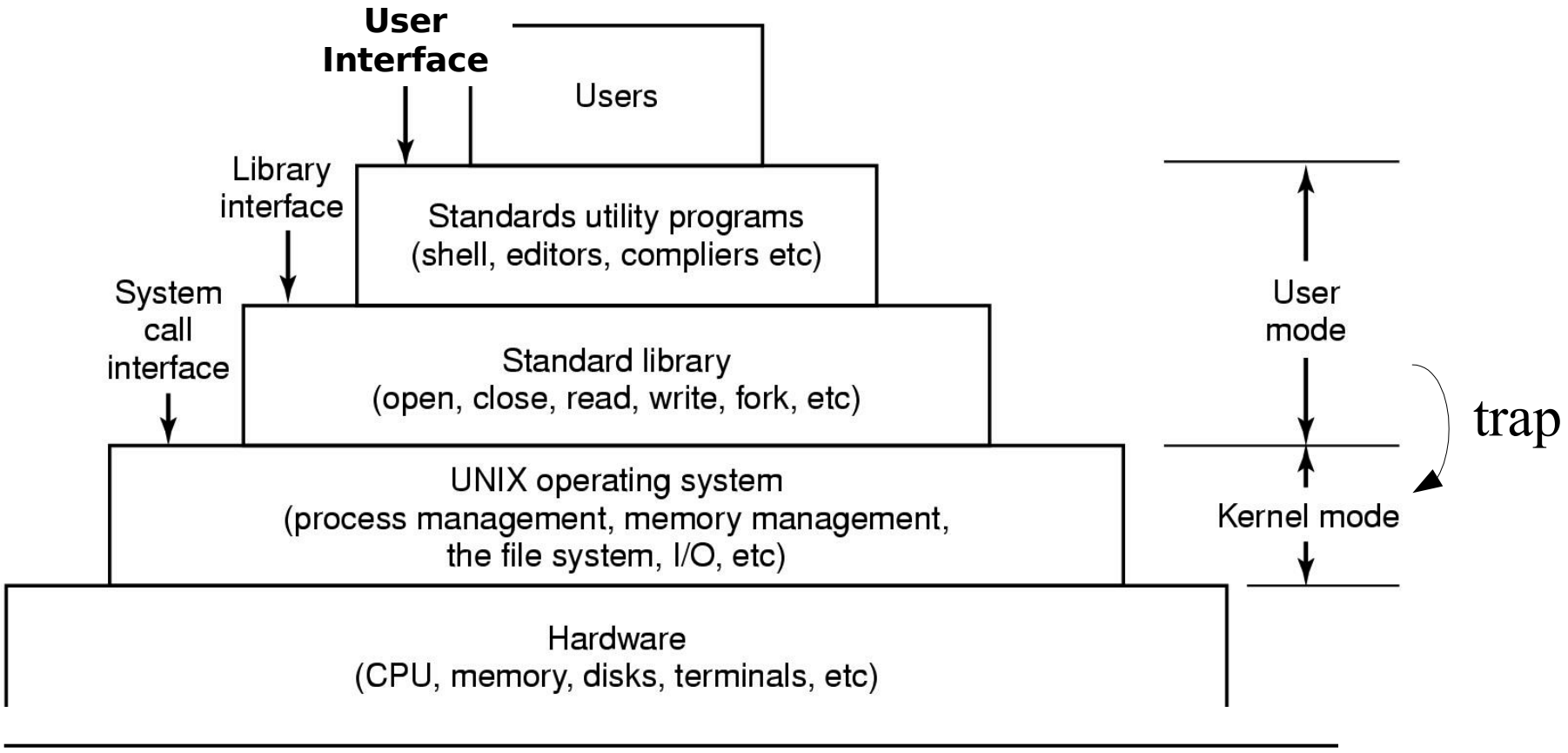
Processi e Thread

- Processi
- Thread
- Meccanismi di comunicazione fra processi (IPC)
- Problemi classici di IPC
- Scheduling
- Processi e thread in Unix
- Processi e thread in Windows

UNIX: struttura generale



UNIX/Linux (1)

- Molte versioni di UNIX (trattiamo le caratteristiche più comuni)
- Ci riferiamo allo standard POSIX
- Trattiamo a parte le caratteristiche di Linux che si discostano maggiormente

UNIX/Linux (2)

(Bovet-Cesati, pag 11)

- La maggior parte dei kernel Unix, incluso quello di Linux, e' monolitico.
- Ogni kernel layer e' integrato all'interno del programma kernel e gira in modo kernel a supporto del processo corrente.

UNIX/Linux (3)

- In Linux un *modulo* e' un oggetto il cui codice puo' essere collegato/scollegato al kernel al runtime.
- Un modulo puo' esssere un driver o un insieme di funzioni che implementa il file system
- Vantaggi dei moduli:
  - approccio modulare
  - indipendenza dalla paittaforma#
  - uso frugale della RAM
  - no penalita' di prestazioni

Processi in UNIX

- Adottano il modello a processi sequenziali
- Ogni processo nasce con un solo thread
- Alcune chiamate di sistema per la gestione dei processi sono:

Process management

Call	Description
pid = fork( )	Create a child process identical to the parent
pid = waitpid(pid, &statloc, options)	Wait for a child to terminate
s = execve(name, argv, environp)	Replace a process' core image
exit(status)	Terminate process execution and return status

Creazione di Processi

- Avviene in due passi
- fork()
  - crea una copia esatta del processo invocante
  - restituisce 0 al figlio ed il PID del figlio al padre
- execve()
  - differenzia un processo rimpiazzando il suo spazio di indirizzamento con quello dell'eseguibile passato come parametro

Una *shell* (interprete di comandi)

cp file1 file 2

```
while (TRUE) {                                  /* repeat forever /*/
        type_prompt( );                         /* display prompt on the screen */
        read_command(command, params);          /* read input line from keyboard */

        pid = fork( );                          /* fork off a child process */
        if (pid < 0) {
                printf("Unable to fork0);       /* error condition */
                continue;                       /* repeat the loop */
        }

        if (pid != 0) {
                waitpid (–1, &status, 0);       /* parent waits for child */
        } else {
                execve(command, params, 0);     /* child does the work */
        }
}
```

Una shell molto semplificata

Terminazione di processi (1)

- pid=waitpid(pid,&status,opt)
  - attende la terminazione di un processo figlio
  - dopo l'esecuzione di waitpid, status contiene l'esito della computazione del processo figlio
  - status = 0 terminazione normale, != 0 terminazione in presenza di errore
- exit(status)
  - termina il processo e restituisce il valore di status al padre (nella variabile status restituita da waitpid)

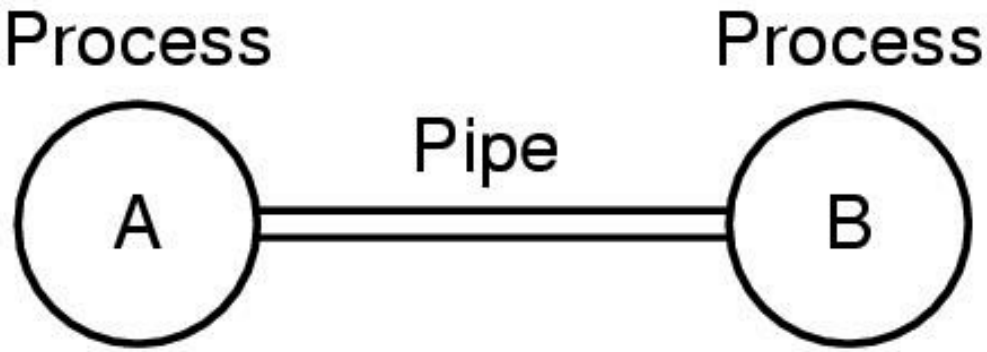
Terminazione di processi (2)

- Processi *zombie*
  - processi terminati il cui padre non ha (ancora) eseguito la waitpid()
  - attendono di restituire il codice di terminazione e svanire

Meccanismi IPC di Unix (1)

sort < f | read

- Pipe : file speciali utilizzati per connettere due processi con un canale unidirezionale di comunicazione



- Se B cerca di leggere da una pipe vuota si blocca
- Quando la pipe è piena A viene automaticamente sospeso
- L'ampiezza della pipe dipende dal sistema

Meccanismi IPC di Unix (2)

- Segnali (interruzioni software)
  - comunicano al processo il verificarsi di una certo evento
  - possono essere inviati solo ai membri del proprio gruppo (antenati, discendenti)
  - generalmente possono essere ignorati, catturati o possono terminare il processo (default per molti segnali)
  - per i segnali catturabili si può specificare un *signal handler* che viene mandato in esecuzione appena il segnale viene rilevato

Meccanismi IPC di Unix (3)

- Segnali (cont.)
  - particolari combinazioni di tasti inviano dei segnali al processo in foregroud
    - Control-C corrisponde a SIGINT
    - Control-Z corresponde a SIGTSTP
  - i segnali servono anche al SO per a comunicare al processo il verificarsi di particolari eventi (es. SIGFPE, errore floating-point)

I segnali previsti da POSIX

Signal	Cause
SIGABRT	Sent to abort a process and force a core dump
SIGALRM	The alarm clock has gone off
SIGFPE	A floating-point error has occurred (e.g., division by 0)
SIGHUP	The phone line the process was using has been hung up
SIGILL	The user has hit the DEL key to interrupt the process
SIGQUIT	The user has hit the key requesting a core dump
SIGKILL	Sent to kill a process (cannot be caught or ignored)
SIGPIPE	The process has written to a pipe which has no readers
SIGSEGV	The process has referenced an invalid memory address
SIGTERM	Used to request that a process terminate gracefully
SIGUSR1	Available for application-defined purposes
SIGUSR2	Available for application-defined purposes

corrispondono a interruzioni sw

Chiamate di sistema relative ai processi

System call	Description
pid = fork( )	Create a child process identical to the parent
pid = waitpid(pid, &statloc, opts)	Wait for a child to terminate
s = execve(name, argv, envp)	Replace a process' core image
exit(status)	Terminate process execution and return status
s = sigaction(sig, &act, &oldact)	Define action to take on signals
s = sigreturn(&context)	Return from a signal
s = sigprocmask(how, &set, &old)	Examine or change the signal mask
s = sigpending(set)	Get the set of blocked signals
s = sigsuspend(sigmask)	Replace the signal mask and suspend the process
s = kill(pid, sig)	Send a signal to a process
residual = alarm(seconds)	Set the alarm clock
s = pause( )	Suspend the caller until the next signal

s è un codice di errore

pid è il codice di identificazione di un processo

residual è il tempo rimanente dal precedente settaggio di alarm()

Le chiamate di sistema relative ai Thread POSIX

err = pthread_create (&tid, attr, funtion, arg);

	Thread call	Description
fork →	pthread_create	Create a new thread in the caller's address space
exit →	pthread_exit	Terminate the calling thread
waitpid →	pthread_join	Wait for a thread to terminate
mutex	pthread_mutex_init	Create a new mutex (semaforo binario)
	pthread_mutex_destroy	Destroy a mutex
down →	pthread_mutex_lock	Lock a mutex
up →	pthread_mutex_unlock	Unlock a mutex
	pthread_cond_init	Create a condition variable
	pthread_cond_destroy	Destroy a condition variable
wait →	pthread_cond_wait	Wait on a condition variable
signal →	pthread_cond_signal	Release one thread waiting on a condition variable

mutex: mutua escusione su variabile condivisa

condition variabile: attesa a lungo termine per sincronizzazione (piu' lenta)

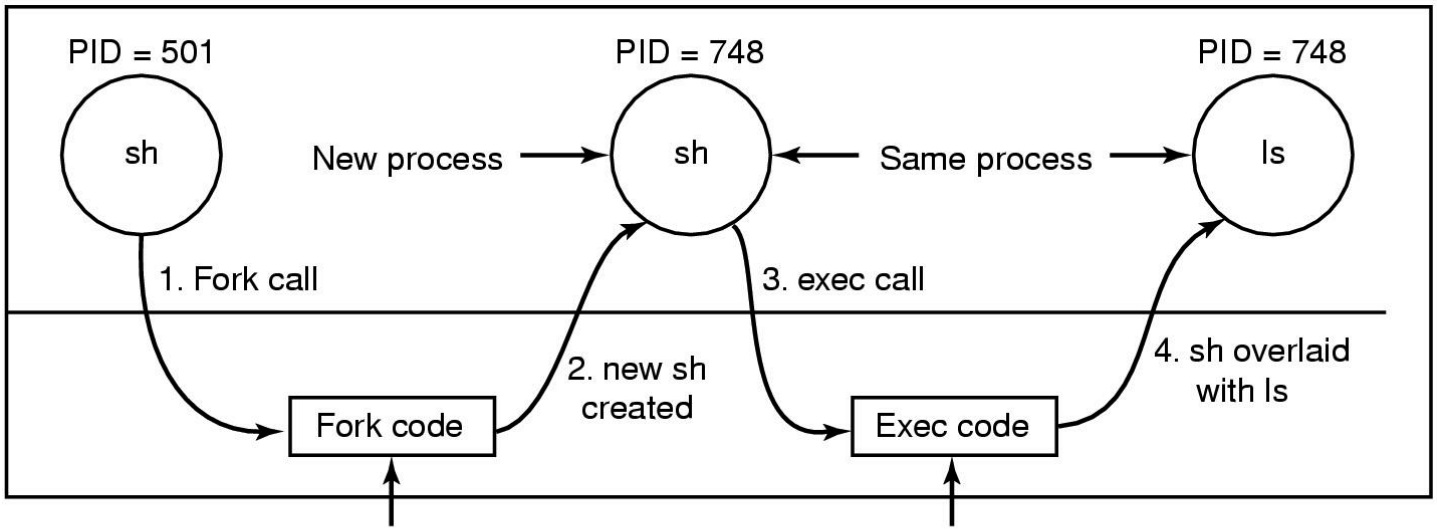
Implementazione di processi (1)

- Si basa su due strutture per ogni processo:
  - *process table* e *user structure*
- Process table : risiede sempre in RAM
  - parametri per lo scheduling
  - immagine di memoria
  - informazioni sui segnali
  - stato
  - PID, PID del padre
  - user e group id.

Implementazione di processi (2)

- User structure/area : risiede su disco se il processo è *swapped*
  - registri
  - tabella dei descrittori di file
  - stato della system call corrente
  - kernel stack
  - informazioni di *accounting* (tempo CPU, etc.)

Il comando *ls*



Passi effettuati durante l'esecuzione del comando *ls* da parte della shell

Implementazione di thread (1)

- Può essere *user-level* o *kernel-level*
- Problema : come mantenere la semantica tradizionale di UNIX?
  - *fork :* tutti i (kernel) thread del padre devono essere creati nel figlio?
  - *I/O* : cosa accade se due thread agiscono sullo stesso file in modo incontrollato?
  - *segnali* : devono essere diretti a un thread in particolare o a tutto il processo?

Implementazione di thread (2)

- I thread di Linux
  - kernel level
  - tipica attivazione di un nuovo thread :

    pid=clone(function, stack_ptr, sharing_flags, arg)
  - function : funzione da cui iniziare l'esecuzione
  - stack_ptr : puntatore alla pila privata del thread
  - arg : argomenti con cui viene attivata function
  - sharing_flags : bitmap di condivisione fra thread padre e thread figlio

I flag per la clone()

Flag	Meaning when set	Meaning when cleared
CLONE_VM	Create a new thread	Create a new process
CLONE_FS	Share umask, root, and working dirs	Do not share them
CLONE_FILES	Share the file descriptors	Copy the file descriptors
CLONE_SIGHAND	Share the signal handler table	Copy the table
CLONE_PID	New thread gets old PID	New thread gets own PID

- Significato dei bit nella bitmap sharing_flags

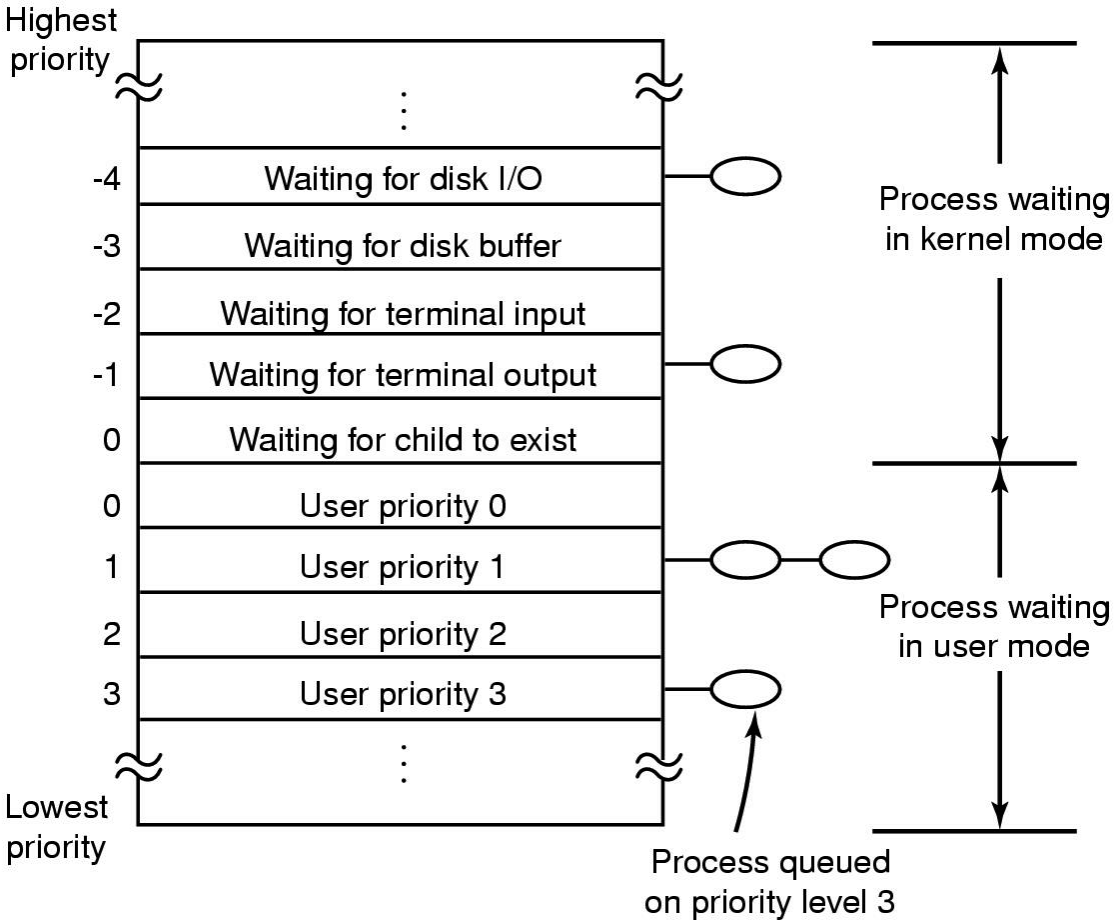
Scheduling in UNIX

Scheduling a due livelli :

- *scheduler a basso livello (low-level)*: sceglie il prossimo processo da mandare in esecuzione fra quelli in RAM
- *scheduler ad alto livello (high-level)*: sposta i processi fra RAM e disco in modo da dare a tutti la possibilità di ottenere l'accesso alla CPU

Nel seguito descriveremo lo scheduler a basso livello

Lo scheduler di UNIX (1)



Lo scheduling a basso livello è basato su una coda a più livelli di priorità

1 quanto = 100 ms=.1 s

Lo scheduler di UNIX (2)

- Si esegue il primo processo della prima coda non vuota per massimo 1 *quanto* (tipicamente 100ms)
- Scheduling round robin fra processi con la stessa priorità
- Una volta al secondo tutte le priorità vengono ricalcolate:

$$priorità = cpu\_usage + nice + base$$

cpu _usage : numero di clock tick per secondo che il processo ha avuto negli ultimi secondi

nice : valore intero nell'intervallo [-20, +20]

base : valore intero che dipende da cosa sta facendo il processo

- ha il valore della priorità precedente se il processo sta eseguendo elaborazione normale in user mode
- ha un valore negativo molto basso se sta effettuando I/O da disco o da terminale

Lo scheduler di UNIX (3)

Meccanismo di *aging (invecchiamento)* usato per il calcolo di *cpu _usage* :

- Fissiamo un intervallo di decadimento Δt
- I tick ricevuti mentre il processo P è in esecuzione vengono accumulati in una variabile temporanea *tick*
- Ogni Δt

  cpu _usage = cpu _usage / 2 + tick

  tick = 0

- Il peso dei tick utilizzati decresce col tempo
- La penalizzazione dei processi che hanno utilizzato molta CPU diminuisce nel tempo

Lo scheduler di Linux (1)

- Vengono schedulati i thread, non i processi
- Tre classi di thread : real-time FIFO, real-time Round Robin, Timesharing
- Ogni thread ha
  - una *priorità* nell'intervallo [0, +40], generalmente all'inizio la priorità di default è 20 (+ nice con system call nice(.))
  - un *quanto* (misurato in *jiffy* = 10ms) jiff = clock tick
- Lo scheduler calcola la *goodness* (*gdn*) di ogni thread come

```
if (class == real-time) gdn = 1000 + priority
if (class == timeshar && quantum > 0) gdn = quantum + priority
if (class == timeshar && quantum == 0) gdn = 0
```

Lo scheduler di Linux (2)

Algoritmo di scheduling :

- Ogni volta viene selezionato il thread con goodness maggiore
- Ogni volta che arriva un tick il quanto del processo in esecuzione viene decrementato
- Un thread viene de-schedulato se si verifica una delle seguenti condizioni
  - il quanto diventa 0
  - il thread si blocca (*i/o, semaforo, ecc*)
  - diventa ready un thread con una goodness maggiore

Lo scheduler di Linux (3)

Algoritmo di scheduling (contd.):

- Quando tutti i quanti dei thread ready sono andati a 0 , lo scheduler ricalcola il quanto di ogni thread (anche se *blocked*) come segue :

quantum = quantum / *2* + priority

favorisce
thread i/o bound

nice