LABORATORIO DI SISTEMI OPERATIVI

Corso di Laurea in Ingegneria Informatica

A.A. 2019/2020

Ing. Guglielmo Cola

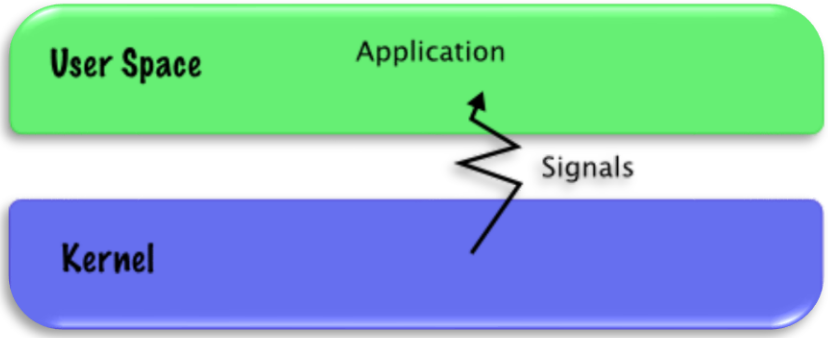

g.cola@iet.unipi.it (specificare "sistemi operativi" nell'oggetto)


www.iet.unipi.it/g.cola/sistemioperativi

ESERCITAZIONE 6

Processi in Unix/Linux (parte II)

SINCRONIZZAZIONE BASATA SU SEGNALI



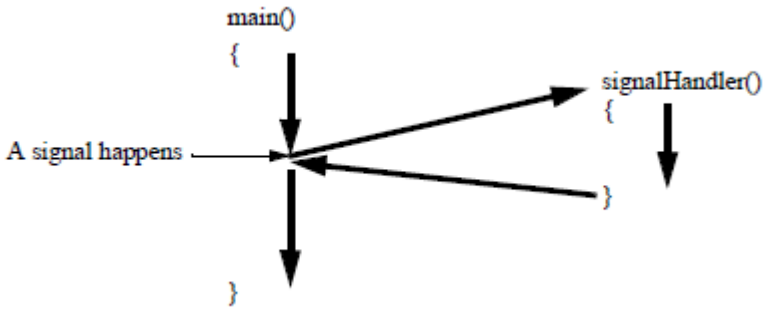
Interazione tra processi

- I processi Unix aderiscono al modello ad ambiente locale
  - Spazio di indirizzamento privato
  - Non c'è condivisione di variabili
- L'unica forma di interazione tra processi è la cooperazione
  - Sincronizzazione (imposizione di vincoli temporali)
  - Comunicazione (scambio di messaggi)
- Queste interazioni si basano su astrazioni realizzate dal kernel
  - I processi possono interagire mediante chiamate di sistema operativo (system calls)

Sincronizzazione mediante segnali

- I segnali sono il meccanismo messo a disposizione dai sistemi Unix/Linux per la sincronizzazione di processi
  - Permettono la notifica di eventi asincroni da parte di un processo a uno o più processi
  - Possono essere utilizzati dal sistema operativo per notificare il verificarsi di eccezioni a un processo utente

- I segnali sono "interrupt software"



Sincronizzazione mediante segnali

- La ricezione di un segnale ha tre possibili effetti sul processo:
  1. Viene eseguita una funzione di gestione (handler) definita dal programmatore
  2. Viene eseguita un'azione predefinita dal sistema operativo (default handler)
  3. Il segnale viene ignorato

- Nei casi 1 e 2 il processo si comporta in modo asincrono rispetto al segnale:
  - L'esecuzione viene interrotta per eseguire l'handler
  - Dopo, se non è terminato, il processo riprende dall'istruzione successiva all'ultima eseguita prima dell'interruzione

Sincronizzazione mediante segnali

- Versioni differenti di Unix possono definire segnali diversi
  - In Linux sono definiti 32 segnali
- La lista dei segnali è definita nel file di sistema `signal.h`
- Ciascun segnale è identificato da un intero e da un nome simbolico

- Pagina del manuale sui segnali:

```
man 7 signal
```

Segnali – esempi

Nome segnale	# segnale	Descrizione segnale
SIGHUP	1	Hang up (il terminale è stato chiuso)
SIGINT	2	Interruzione del processo. CTRL+C da terminale.
SIGQUIT	3	Interruzione del processo e core dump. CTRL+\ da terminale.
SIGKILL	9	Interruzione immediata. Questo segnale non può essere ignorato ed il processo che lo riceve non può eseguire operazioni di chiusura
SIGTERM	15	Terminazione del programma.
SIGUSR1	10	Definito dall'utente. Default: termina processo.
SIGUSR2	12	Definito dall'utente. Default: termina processo.
SIGSEGV	11	Errore di segmentazione

Segnali – esempi

Nome segnale	# segnale	Descrizione segnale
SIGALRM	14	Il timer è scaduto
SIGCHLD	17	Processo figlio terminato, fermato, o risvegliato. Ignorato di default.
SIGSTOP	19	Ferma temporaneamente l'esecuzione del processo: questo segnale non può essere ignorato
SIGTSTP	20	Sospende l'esecuzione del processo. CTRL+Z da terminale.
SIGCONT	18	Il processo può continuare, se era stato fermato da SIGSTOP o SIGTSTP.

System call per i segnali

`signal`
- Permette di definire la funzione che dovrà gestire il segnale

`kill`
- Invio di segnali

`alarm` e `sleep`
- Invio implicito di segnali

System call per i segnali – signal

```
typedef void (*sighandler_t)(int);
sighandler_t signal(int sig, sighandler_t handler)
```

- Permette di definire la funzione ("handler") che dovrà gestire il segnale "sig"
  - La funzione handler deve prevedere un parametro intero, che al momento della ricezione del segnale conterrà il codice del segnale
  - handler può valere anche `SIG_IGN` (ignora il segnale) o `SIG_DFL` (ripristina azione di default)
- Restituisce un puntatore al precedente handler del segnale, `SIG_ERR` in caso di errore
- Pagina del manuale:
  `man 2 signal`

System call per i segnali – signal

- Il figlio eredita dal padre le informazioni relative alla gestione dei segnali
- Eventuali signal eseguite dal figlio non hanno effetto sul padre
- Le syscall exec non mantengono le associazioni segnale-handler
  - I segnali ignorati, però, continuano ad essere ignorati

System call per i segnali – kill

```
int kill(pid_t pid, int sig)
```

- Invia il segnale "sig" al processo "pid"
  - `pid > 0` → il segnale viene inviato a pid
  - `pid == 0` → il segnale viene inviato a tutti i processi nello stesso process group del chiamante
  - `pid == -1` → il segnale viene inviato a tutti i processi a cui il chiamante può inviare segnali
  - `pid < -1` → il segnale viene inviato ai processi il cui process group è –pid
- Ritorna zero in caso di successo
- Pagina del manuale:
  `man 2 kill`

System call per i segnali – sleep

```
unsigned int sleep(unsigned int seconds)
```

- Il processo chiamante va nello stato sleep fino a che
  - Sono passati "seconds" secondi
  - Arriva un segnale che non viene ignorato
- Quando è passato il tempo indicato, il processo viene svegliato dal segnale `SIGALARM`
- Ritorna zero se è passato il tempo previsto ("seconds"), altrimenti il tempo rimasto dopo l'arrivo di un segnale
- Pagina del manuale:
  `man 3 sleep`

System call per i segnali – alarm

`unsigned int alarm(unsigned int seconds)`

- Provoca la ricezione di un segnale `SIGALARM` dopo "seconds" secondi
  - Un eventuale "allarme" invocato precedentemente viene cancellato
  - Se "seconds" è zero, viene eliminato un eventuale "allarme" invocato precedentemente
  - Ritorna zero se non c'era un allarme programmato, altrimenti ritorna il numero di secondi mancanti all'ultimo allarme programmato

- Pagina del manuale:
  `man alarm`

Comunicazione mediante scambio di messaggi – pipe

- I processi possono comunicare sfruttando il meccanismo delle pipe
  - Comunicazione indiretta, senza naming esplicito
  - Realizza il concetto di mailbox nella quale si possono accodare messaggi in modo FIFO
  - La pipe è un canale monodirezionale
    - Ci sono due estremi, uno per la lettura e uno per la scrittura
  - Astrazione realizzata in modo omogeneo rispetto alla gestione dei file:
    - A ciascun estremo è associato un file descriptor
    - I problemi di sincronizzazione sono risolti dalle primitive read/write
- I figli ereditano gli stessi file descriptor e possono utilizzarli per comunicare con il padre e gli altri figli
  - Per la comunicazione di processi che non si trovano nella stessa gerarchia si utilizzano i socket
- Pagina del manuale:

```
man pipe
```

GESTIONE DEI PROCESSI DA TERMINALE

Invio di segnali da terminale – kill

- Il comando `kill` permette l'invio di segnali a processi da terminale

```
kill [options] pid [pid2…]
```

  - Il segnale di default è `SIGTERM`
  - `kill -l` mostra l'elenco dei segnali disponibili
  - `kill -SEGNALE pid` invia il segnale `SEGNALE` al processo `pid`

- Un utente "normale" può inviare segnali solo ai processi di cui è proprietario
- Root può inviare segnali a tutti i processi

Visualizzazione dei processi – ps

- Il comando ps permette di visualizzare i processi in esecuzione (snapshot, informazione statica)
  - Opzioni principali in Linux
    - `-u utente` visualizza i processi dell'utente specificato
    - `u` formato output utile all'analisi dell'utilizzo delle risorse
    - `a` processi di tutti gli utenti
    - `x` anche processi che non sono stati generati da terminali
    - `o` mostra solo i campi specificati di seguito
    - `-O` mostra i campi specificati di seguito, oltre ad alcuni campi di default

Visualizzazione dei processi – ps

- Esempio:

```
studenti@studenti:~$ ps u
USER       PID %CPU %MEM    VSZ   RSS TTY      STAT START   TIME COMMAND
studenti  1511  0.0  0.6  24288  6168 pts/0    Ss+  21:58   0:00 bash
studenti  1783  2.0  0.5  24284  6056 pts/1    Ss   23:58   0:00 bash
studenti  1788  0.0  0.2  19104  2412 pts/1    R+   23:58   0:00 ps u
```

- Stati principali
  - S sleep
  - T bloccato
  - R running
  - Z zombie

ESERCIZI

Esercizio 1

- Realizzare un programma in C che stampa un messaggio dentro un ciclo infinito
- Eseguire il programma e, da terminale, lanciare il segnale SIGINT (CTRL+C) per terminarlo
- Modificare il programma in modo da gestire SIGINT
  - Quando riceve il segnale, il processo stampa un messaggio "Ricevuto segnale <codice segnale>".
  - Fare in modo che questo messaggio sia visibile per qualche secondo
- Adesso il processo non può più essere terminato con CTRL+C
  - Aprire un nuovo terminale e utilizzare ps per trovare il PID del processo e poi kill per terminarlo

Esercizio 2

- Partire dall'esercizio es2.c dell'esercitazione precedente (creazione di processi figli e system call wait)
- Modificare il codice come segue:
  - Il padre genera 3 processi figli
  - Il primo processo figlio creato stampa il solito messaggio e poi termina con exit(0).
  - Gli altri 2 processi figli entrano in un loop infinito prima di terminare
  - Il padre, dopo aver creato i processi figli
    - Si sospende per tre secondi
    - Utilizza la primitiva kill per inviare a tutti i figli il segnale SIGUSR1
  - Scrivere un gestore per il segnale SIGUSR1 che stampa il messaggio "Ho ricevuto il segnale <sig>, il mio PID è <pid>" e poi termina il processo con exit(1)
    - Fare in modo che solo i figli vengano terminati in questo modo (il padre ignora il gestore). Dopo aver inviato il segnale, il padre deve continuare regolarmente la sua esecuzione, effettuando le chiamate wait() previste