Università di Roma Tor Vergata
Corso di Laurea triennale in Informatica

Sistemi operativi e reti

A.A. 2016-17

Pietro Frasca

Lezione 19

Martedì 20-12-2016

Terminazione di processi

- Un processo può terminare **volontariamente** o **involontariamente** (per via di vari tipi di eccezioni o per via di segnali di interruzione).
- La chiamata di sistema **`exit`** consente di terminare un processo.

  ```
  void exit (int stato);
  ```

  Il parametro **`stato`** consente al processo figlio che chiama la **`exit`** di comunicare al processo padre, un valore di tipo intero che ne indica lo stato di uscita.
- Per rilevare la notifica del processo figlio, il padre deve sincronizzarsi con il processo figlio e attendere la sua terminazione.

Per sincronizzarsi può utilizzare la chiamata **wait** o **waitpid**.

```
int wait (int *stato);
int waitpid (int pid, int *stato, int opz);
```

- La differenza tra le due chiamate sta nel fatto che **wait** ritorna il **PID** di un qualsiasi figlio che è terminato, mentre **waitpid** permette, tramite il primo parametro **pid** di specificare il particolare figlio da attendere.
- L'argomento **stato**, in ambedue le funzioni, è un riferimento ad una variabile che conterrà lo stato del processo figlio quando termina. Più precisamente, nel caso di terminazione volontaria, la variabile stato conterrà nel suo byte più significativo il valore che il processo figlio ha passato alla chiamata exit, mentre conterrà l'id del segnale che ha causato la terminazione nel caso di terminazione forzata.

- Il significato del valore stato è stabilito dal programmatore.
- Il terzo parametro in **`waitpid`** stabilisce se il processo chiamante si blocca in attesa della terminazione del figlio o invece continui l'esecuzione. Se il valore del terzo parametro **`opz`** è uguale a **`0`** il processo chiamante si blocca altrimenti se **`opz`** ha un valore diverso da 0 il processo chiamante continua la sua esecuzione.
- La **`wait`** è bloccante e ritorna il **`pid`** del processo figlio che ha risvegliato il padre.

```
main(){
   int pid,stato;
   pid=fork();
   if (pid==0){
        // codice del figlio
        printf("sono il figlio pid: %d \n",getpid());
        sleep(10); // sospensione per 10 secondi
       exit(2);//valore che il padre leggerà in stato
   } else if (pid > 0){
        //codice del padre
       pid=wait(&stato);
        printf("processo figlio pid: %d terminato\n",
          pid);
       if (stato<256)
          printf("terminaz. forzata: segnale n = %d",
                    stato);
       else
          printf("terminaz. volontaria stato: %d \n",
                  stato>>8);
  } else
        printf("fork fallita");
}
```

Sostituzione del codice

- Generalmente la fork si usa insieme a una delle chiamate di sistema della famiglia **exec.**
- La **exec** permette ad un processo di eseguire un altro processo, senza che tra di essi esistano relazioni gerarchiche, sostituendolo con l'immagine del processo che si vuole eseguire.
- Quindi dopo la **fork** nel sistema è presente un processo in più, mentre dopo la **exec** il numero di processi non cambia ed il codice del processo chiamante non esiste più.
- La exec ha varie forme, tra le quali, due molto usate sono la **execl** e la **execv.**

```
execl(char *path, char *arg1, char *arg2,… char *argN, (char *)0)
```

```
execv(char *path, char *argv[])
```

- Alla **execl** è possibile passare un numero variabile di parametri. L'ultimo parametro è il carattere nullo, che indica la fine della lista di parametri. Il primo parametro **path** della funzione è il nome del file da eseguire. **Arg1**, **arg2**..**argN** sono i parametri da passare al programma specificato con il parametro path.
- Dopo l'esecuzione di execl, al processo chiamante è assegnata l'immagine del nuovo programma e quindi avrà **codice, dati e stack nuovi**. Del programma chiamante resta la **process structure**, della quale viene modificato solo il riferimento alla **text structure** (puntatore indiretto del segmento del codice); tutti gli altri campi come ad esempio il **PID** e il **PPID** restano uguali.
- Anche la **user structure** cambia leggermente, sono modificati solo i campi che dipendono dal nuovo codice come ad esempio il valore del PC (program counter). Le eventuali **risorse allocate** o **file aperti sono accessibili** al nuovo processo.

```
#include <stdio.h>
#include <stdlib.h>
main(){
  int pid,stato;
  pid=fork();
  printf("pid=%d \n",getpid());// stampa il pid del padre
                                  e del figlio
  if (pid==0) {
    //figlio
    execl("./pro1","Cari","saluti"," a tutti",(char *)0);
    printf("exec fallita");
    exit(1);
  } else if (pid > 0){
    printf("sono il padre con pid=%d",getpid());
    pid=wait(&stato);
  } else
       printf ("Errore fork");
}
```

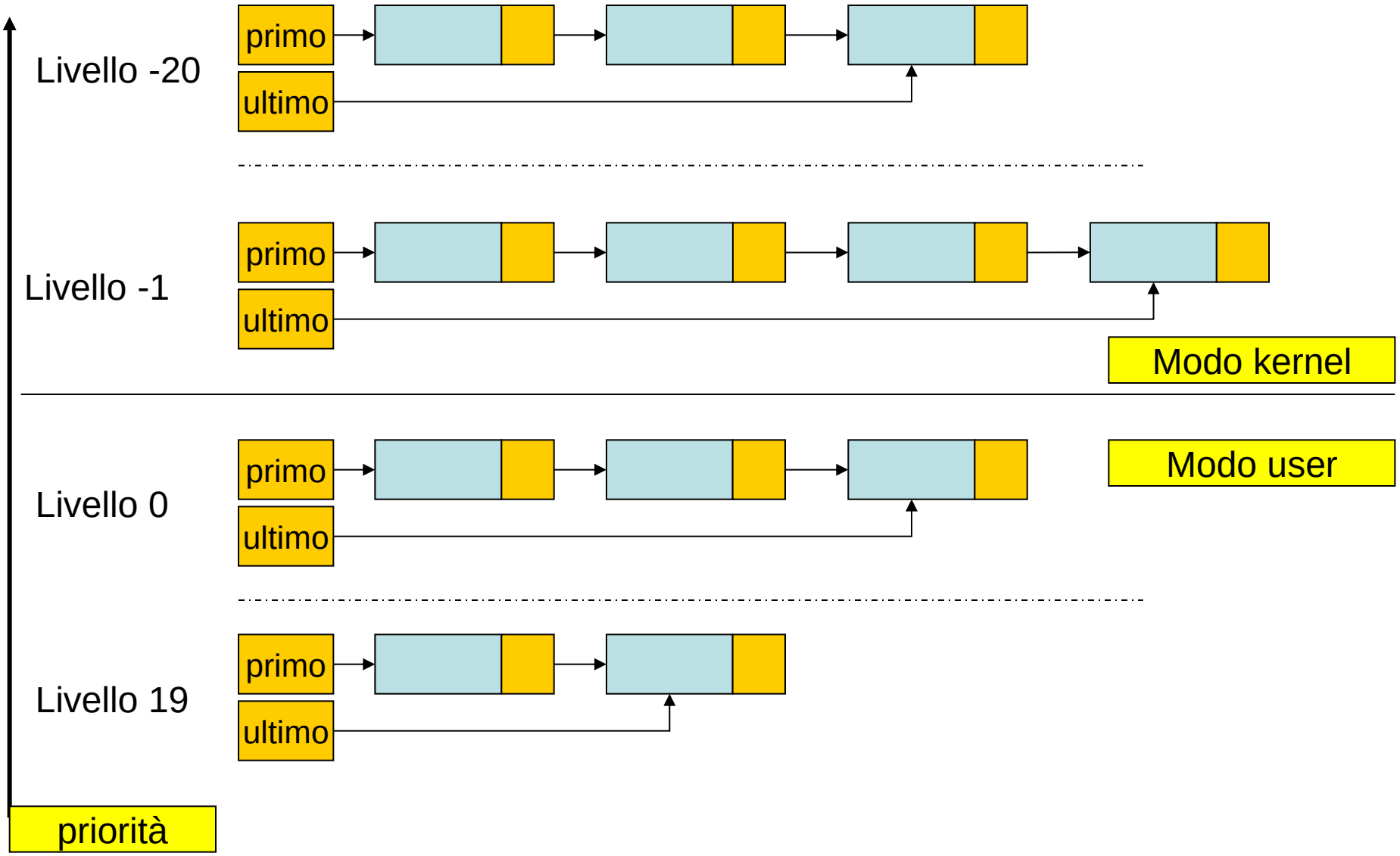
File pro1.c

```
#include <stdio.h>
#include <stdlib.h>
main(int argc, char *argv[]){
  int i;
  printf("Sono stato chiamato con execl il mio pid è %d
   \n",getpid());
  for (i=0;i<argc;i++)
    printf ("%s ",argv[i]); // visualizza i parametri
  printf("\n");
}
```

Scheduling in UNIX

- Poiché UNIX è un sistema multiutente e multitasking, l'algoritmo di scheduling della CPU è stato progettato per fornire buoni tempi di risposta ai **processi interattivi**.
- I thread sono generalmente a livello di kernel, pertanto lo scheduler si basa sui thread e non sui processi.
- E' un algoritmo a **due livelli** di scheduler.
- Lo **scheduler a breve termine** sceglie dalla coda dei processi pronti il prossimo processo/thread da eseguire.
- Lo **scheduler a medio termine (swapper)** sposta pagine di processi tra la memoria e il disco (area swap o file di paging) in modo che tutti i processi abbiano la possibilità di essere eseguiti.
- Ogni versione di UNIX ha uno scheduler a breve termine leggermente diverso, ma tutti seguono uno schema di funzionamento basato su code di priorità.

- In Unix le priorità dei processi eseguiti in **modalità utente** sono espresse con valori interi positivi mentre le priorità dei processi eseguiti in **modalità kernel** (che eseguono le chiamate di sistema) sono espresse con valori interi negativi.
- I valori **negativi** rappresentano **priorità maggiori**, rispetto ai valori positivi che hanno priorità minore.
- Lo scheduler a breve termine sceglie un processo dalla coda con priorità più alta. Le code sono gestite in modalità **RR**. Il quanto di tempo assegnato al processo per l'esecuzione dura generalmente **20**-**100 ms**.
- Un processo è posto in fondo alla coda, quando termina il suo quanto di tempo.



scheduling in unix

- I valori delle **priorità sono dinamici** e vengono ricalcolati **ogni secondo** in base ad una relazione che dipende dai seguenti parametri:

  - **valore iniziale (base)**
  - **uso_cpu**
  - **nice**

  priorità = f(base, uso_cpu, nice)

- Ogni processo viene poi inserito in una coda, come mostrato nella figura precedente, in base alla nuova priorità.
- **uso_cpu** rappresenta, l'uso medio della cpu da parte del processo durante gli ultimi secondi precedenti. Questo parametro è un campo del descrittore del processo.

- L'incremento del valore del parametro **uso_cpu** provoca lo spostamento del processo in una coda a priorità più bassa.
- Il valore di **uso_cpu** varia nel tempo in base a varie strategie usate nelle varie versioni di UNIX.
- Ogni processo ha, inoltre, un valore del parametro **nice** associato. Il valore di base è 0 e l'intervallo di valori possibili è compreso tra –20 e +19. Ad esempio, con il comando **nice** (che utilizza l'omonima chiamata di sistema **nice**), un utente può assegnare ad un proprio processo un valore nice compreso tra 0 e 19. Soltanto il **superuser** (root) può assegnare i valori di nice compresi tra –1 e –20 ad un processo.
- Esempio:

```
nice –n 10 mio_calcolo &
```

esegue il programma **mio_calcolo** in background assegnando al processo un valore nice pari a 10.

- Per quanto riguarda lo scheduling per le estensioni real-time, lo standard P1003.4 di UNIX, cui aderisce anche Linux, prevedono le seguenti classi di thread:
  - **Real-time FIFO;**
  - **Real time round-roubin;**
  - **Timesharing**
- I thread real-time FIFO hanno la priorità massima. A questi thread può essere revocata la cpu solo da thread della stessa classe con più alta priorità.
- I thread real-time RR sono simili ai real-time FIFO, ma viene loro revocata la CPU allo scadere del proprio quanto di tempo, il cui valore dipende dalla priorità.
- Le due classi di thread sono **soft real-time**.
- I thread real-time hanno livelli di priorità da 0 a 99, dove 0 è il livello di priorità più alto.

- I thread standard, non real-time, hanno livelli di priorità compresi tra 100 e 139. In totale, quindi si hanno 140 livelli di priorità.
- IL quanto di tempo è misurato in numero di scatti di clock. Lo scatto di clock è detto **jiffy** e dura 1 ms.

Interazione tra processi

- I processi possono cooperare tra loro o competere per l'uso di risorse comuni.
- I processi unix seguono il modello ad ambiente locale, un processo ha un proprio spazio di indirizzamento privato e pertanto non può condividere dati con altri processi.
- In Unix, la sincronizzazione può avvenire attraverso lo scambio di ***segnali***, mentre la comunicazione può realizzarsi mediante l'uso di memoria condivisa e/o lo scambio di messaggi oppure utilizzando ***pipe*** *e/o* ***socket***.

Sincronizzazione: i segnali

- in unix la sincronizzazione avviene mediante i segnali, meccanismi realizzati a livello di kernel che consentono la notifica di eventi asincroni tra processi.

- Il segnale è un evento che un processo mittente invia ad uno o più processi destinatari. Il segnale genera nel processo destinatario un'interruzione del flusso di esecuzione.
- In particolare, quando un processo riceve un segnale, può comportarsi in uno dei seguenti modi:
  - **Eseguire un'azione predefinita dal sistema operativo**
  - **Ignorare il segnale**
  - **Gestire il segnale con una funzione (handler) definita dal programmatore**
- Per ogni versione di unix esistono diversi segnali. Ogni segnale è identificato da un **intero** e da un **nome simbolico** definiti nel file header di sistema **signal.h**.
- Con la shell, si può visualizzare l'elenco dei segnali mediante il comando **kill –l**.

- Sono disponibili 2 segnali **SIGUSR1** e **SIGUSR2** a cui non è associata nessuna azione di default. Questi segnali possono essere usati dai processi utente per realizzare specifiche politiche di sincronizzazione.
- Alcuni segnali non sono intercettabili mediante handler (ad esempio **SIGKILL** che provoca la terminazione del processo)

System Call per l'uso dei segnali

- Un processo che riceve un segnale può gestire l'azione di risposta alla ricezione di tale evento utilizzando la system call signal:

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

  - **sig** è un intero (o la costante simbolica) che specifica il segnale da gestire;
  - **handler** è il puntatore alla funzione che implementa il codice da eseguire quando il processo riceve il segnale. Il parametro handler può specificare la funzione di gestione dell'interruzione *(handler),* oppure assumere il valore
    - **SIG_IGN** nel caso in cui il segnale debba essere ignorato;
    - **SIG_DFL** nel caso in cui debba essere eseguita l'azione di default.

- la funzione **handler** ha un parametro di tipo intero che, al momento della sua attivazione, assumerà il valore dell'identificativo del segnale che ha ricevuto.
- La chiamata **sigaction** appartenente allo standard POSIX è da preferirsi alla signal.
- L'esempio seguente mostra l'uso della system call signal.

```
#inc1ude <signal.h>
void gestore (int signum){
  printf("Ricevuto il segnale %d \n", signum);
  /* In alcune versioni di unix l'associazione
   segnale/gestore non è persistente. In questo caso è
   necessario rieseguire la signal.
  */
  // signal(SIGUSR1, gestore);
}
main () (
  signal (SIGUSRl, gestore) ;
  /* da qui in poi il processo eseguirà la funzione gestore
   quando riceverà il segnale SIGUSRl */
  ..............
  signal (SIGUSRl, SIG_IGN) ;
  / * SIGUSRl è da qui ignorato: il processo
      non eseguirà più la funzione gestore in risposta a
       SIGUSRl */
}
```

- Le associazioni tra segnali e azioni sono registrate nella ***User Structure*** del processo.
- Dato che la **fork** copia la *User Area* del padre nella *User Area* del figlio e che padre e figlio condividono lo stesso codice, il figlio eredita dal padre le informazioni relative alla gestione dei segnali e quindi:
  - **Le azioni di default dei segnali del figlio sono le stesse del padre;**
  - **ogni processo figlio ignora i segnali ignorati dal padre;**
  - **ogni processo figlio gestisce i segnali con le stesse funzioni usate dal padre;**
- Dato che padre e figlio hanno *User Structure* distinte, eventuali chiamate signal eseguite dal figlio sono indipendenti dalla gestione dei segnali del padre.
- Inoltre, un processo quando chiama una funzione della famiglia exec non mantiene l'associazione segnale/handler dato che una **exec** mantiene la *User Structure* del processo che la chiama, ma **non dati e codice** e quindi neanche le funzioni di gestione dei segnali.

Invio di segnali tra processi

- I processi possono inviare segnali ad altri processi con la system call **kill**:

```
#include <signal.h>
int kill (int pid, int sig);
```

  - **pid** è il pid del processo destinatario del segnale **sig**. Se **pid** vale **zero**, il segnale **sig** viene inviato a tutti i processi della gerarchia del processo mittente.
  - **sig** è il segnale da inviare, espresso come numero o come costante simbolica.

- L'esempio seguente mostra l'uso di signal e kill. Il programma, genera due processi (padre e figlio). Entrambi i processi gestiscono il segnale SIGUSR1 mediante la funzione gestore: il figlio, infatti, eredita l'impostazione della signal del padre chiamata prima della fork. Una volta attivi entrambi i processi, il padre invia continuamente il segnale SIGUSR1 al figlio.

```
#include <stdio.h>
#include <signal.h>
void gestore (int signum) {
   static cont=0;
   printf ("Pid %d; ricevuti n. %d segnali %d \n",
   getpid(), cont++, signum);
}
int  main () {
  int pid;
  signal(SIGUSR1, gestore);
  pid = fork ();
  if (pid==0) /* figlio */
    for (; ;) pause();
  else /* padre */
    for ( ; ;) {
      kill (pid, SIGUSR1);
      sleep(1);
     }
}
```

- Oltre alla system call kill, esistono altre chiamate di sistema che automaticamente inviano segnali. Ad esempio la funzione **alarm** causa l'invio del segnale **SIGALRM** al processo che la chiama dopo un intervallo di tempo specificato nell'argomento della funzione.

```
#include <unistd.h>
unsigned int alarm (unsigned int seconds)
```

- L'esempio seguente mostra l'uso di **alarm** e **pause**. Dopo **ns** secondi viene inviato un segnale di allarme (SIGALRM) e viene eseguita la function **azione** specificata in **signal**. Il tempo di allarme ns viene incrementato dopo ogni chiamata di alarm. La function system manda in esecuzione il programma specificato nell'argomento. Nell'esempio viene eseguita la funzione **system** che consente di mandare in esecuzione un programma specificato nell'argomento, in questo caso viene eseguito comando **date** che visualizza data e ora correnti.

```c
#include <stdio.h>
#include <stdlib.h>
#include <signal.h>
int ns=1; // periodo iniziale di allarme (1 secondo)
int nmax=10; // valore massimo dell'intervallo di allarme
void azione(){
  /* questa funzione viene eseguita ogni volta
     che il processo riceve il segnale SIGALRM,
  */
  printf("Segnale di allarme ricevuto...eseguo date \n");
  system("date"); // esegue il comando date

  /*
    riassegnamento del periodo di allarme
    che cancella il precedente periodo assegnato.
  */
  alarm(ns); // ns viene incrementato
}
```

```
int main() {
  int i;
  signal(SIGALRM,azione);
  alarm(ns);
  while(ns <= nmax) {
    printf("processo in pausa\n");
    pause();
    printf("fine pausa\n");
    ns++;// incremento del periodo di allarme
   }
   exit(0);
}
```