

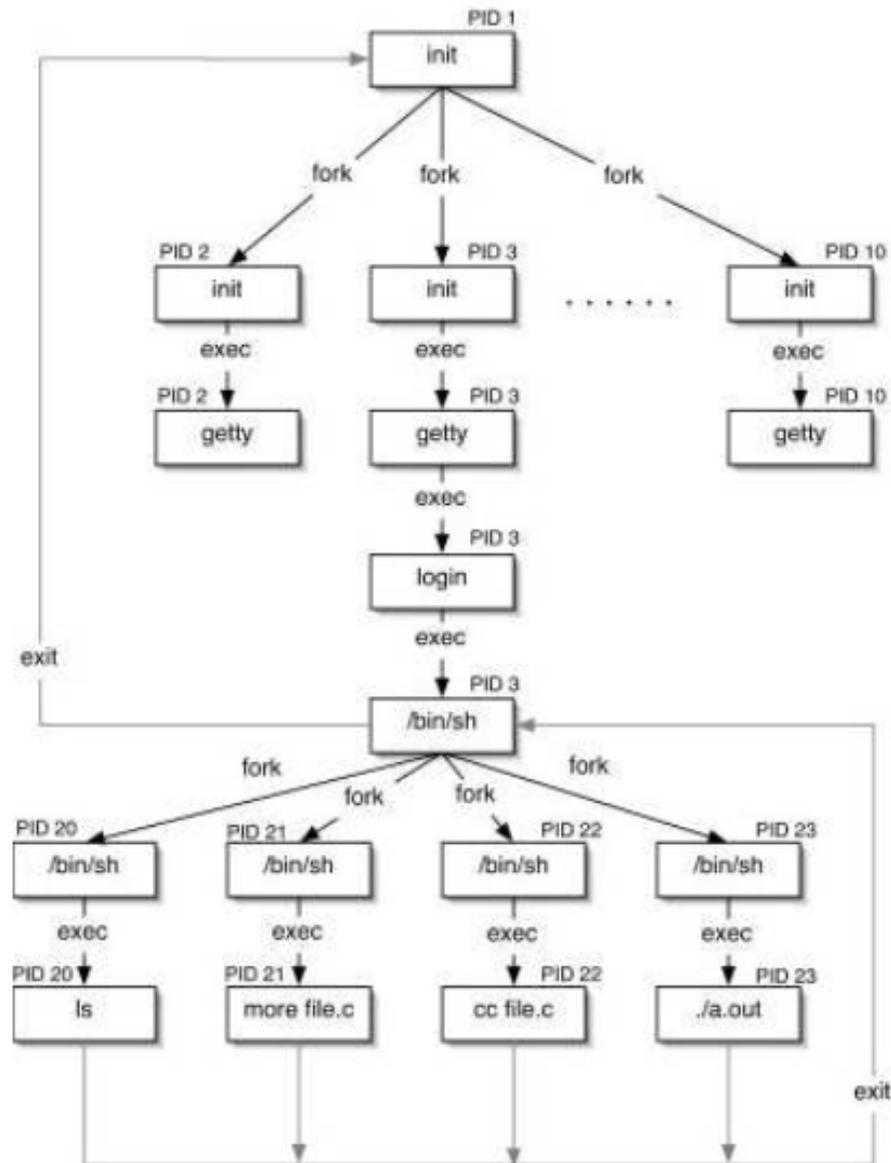
# Processi - II

Franco Maria Nardini

# Processi

- Programmi in esecuzione in memoria sono chiamati *processi*.
  - Caricati in memoria da una delle sei funzioni `exec(3)`.
  - Ogni processo ha un *identificatore univoco* (PID) non negativo.
  - Nuovi processi vengono creati con la system call `fork(2)`.
- Il controllo dei processi è fatto principalmente con: `fork(2)`, `exec(3)`, e `waitpid(2)`.

# Boot



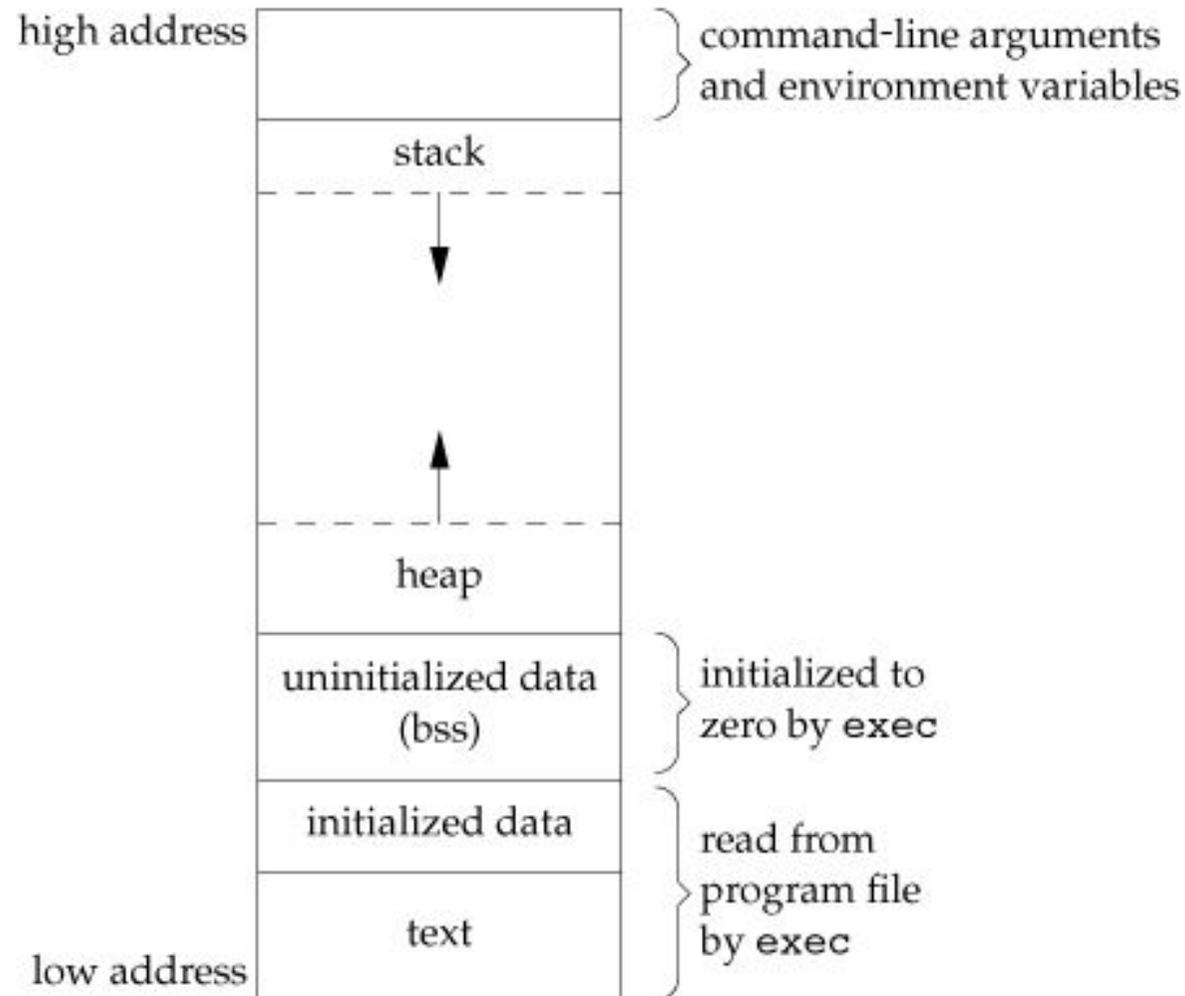
# Memory Layout

- un programma C è composto dalle seguenti parti:
  - text segments: sono le istruzioni che la CPU esegue. è condiviso in memoria (una sola copia). è read-only.
  - (initialized) data segments: contiene variabili che sono specificamente inizializzate nel programma. La dichiarazione `int count = 0;` fuori da funzioni, causa la memorizzazione in questa sezione.

# Memory Layout

- un programma C è composto dalle seguenti parti:
  - uninitialized data segments (bss): tutte le variabili inizializzate dal kernel a zero o NULL prima dell'esecuzione. `long sum[1000]` fuori da funzioni è memorizzato qui.
  - stack: dove la parte automatica è salvata insieme a tutte informazioni salvate ogni volta che una funzione è chiamata. ogni chiamata corrisponde ad un nuovo stack frame
  - heap: contiene l'allocazione dinamica.

# Memory Layout



# Memory Layout

```
$ cc hw.c
$ file a.out
a.out: ELF 32-bit LSB executable, Intel 80386, version 1 (SYSV),
dynamically linked (uses shared libs), for GNU/Linux 2.6.15, not stripped
$ ldd a.out
linux-gate.so.1 => (0x00c66000)
libc.so.6 => /lib/tls/i686/cmov/libc.so.6 (0x006b4000)
/lib/ld-linux.so.2 (0x005fe000)
$ size a.out
   text    data     bss     dec     hex filename
   918     264         8    1190    4a6 a.out
$ objdump -d a.out >obj
$ wc -l obj
225 obj
$
```

# Allocazione di Memoria

- ISO C specifica tre funzioni per l'allocazione di memoria
  - `malloc`: alloca uno specifico numero di byte
  - `calloc`: alloca la memoria per N oggetti di M bytes
  - `realloc`: incrementa o diminuisce la memoria precedentemente allocata:
    - se aumentata: l'area di memoria può essere spostata
    - copie del contenuto (problemi con puntatori!)
    - ritorno della size totale

# Allocazione di Memoria

```
#include <stdlib.h>
```

```
void malloc(size_t size);
```

```
void calloc(size_t nobj, size_t size);
```

```
void realloc(void ptr, size_t newsize);
```

All three return: non-null pointer if OK, NULL on error

# Allocazione di Memoria

- implementate con `sbrk(2)`:
  - espande o contrae la memoria di un processo
  - `malloc` e `free` non diminuiscono la memoria:
    - non ritornata al kernel e tenuta nella `malloc pool` per allocazioni successive

# Segnali

- Sono il mezzo con cui i processi sono notificati di eventi asincroni:
  - un timer settato inizia a suonare (SIGALRM)
  - I/O richiesto è pronto (SIGIO)
  - utente che ridimensiona una finestra (SIGWINCH)
  - utente disconnesso dal sistema (SIGHUP)

# Segnali

- Altri modi di generare un evento:
  - segnali generati dal terminale (utente preme una combinazione di tasti che genera segnale)
  - eccezioni hardware (div by 0, reference invalida, ecc).
  - `kill(1)` e (`kill(2)`) abilitano l'utente all'invio di segnali a processi (se owner o superuser)
  - condizioni da software: dati da network file description pronti, ecc.

# Segnali

```
#include <sys/types.h>
#include <signal.h>

int kill(pid_t pid, int signo);
int raise(int signo);
```

- `pid > 0`, il segnale è mandato al processo identificato dal PID
- `pid == 0`, il segnale è mandato a tutti i processi aventi PGID uguale a quello del mandante
- `pid == -1`
  - POSIX.1 non definisce, BSD si (`kill(2)`)

# Segnali

- Dopo il ricevimento del segnale, si può fare una di queste cose:
  - Ignorarlo. (ci sono segnali che non si **può** o non si **deve** ignorare)
  - Catturarlo, attraverso una funzione definita da noi che il kernel chiama e che tratta lo specifico caso.
  - Accettare l'azione di default. Il kernel chiama la funzione che implementa l'azione di default per il segnale specifico

KILL(1)

BSD General Commands Manual

KILL(1)

**NAME**`kill -- terminate or signal a process`**SYNOPSIS**

```
kill [-s signal_name] pid ...
kill -l [exit_status]
kill -signal_name pid ...
kill -signal_number pid ...
```

**DESCRIPTION**

The `kill` utility sends a signal to the processes specified by the `pid` operand(s).

Only the super-user may send signals to other users' processes.

The options are as follows:

**-s signal\_name**

A symbolic signal name specifying the signal to be sent instead of the default TERM.

**-l [exit\_status]**

If no operand is given, list the signal names; otherwise, write the signal name corresponding to `exit_status`.

**-signal\_name**

A symbolic signal name specifying the signal to be sent instead of the default TERM.

**-signal\_number**

A non-negative decimal integer, specifying the signal to be sent instead of the default TERM.

The following pids have special meanings:

**-1** If superuser, broadcast the signal to all processes; otherwise broadcast to all processes belonging to the user.

Some of the more commonly used signals:

1	HUP (hang up)
2	INT (interrupt)
3	QUIT (quit)
6	ABRT (abort)
9	KILL (non-catchable, non-ignorable kill)
14	ALRM (alarm clock)
15	TERM (software termination signal)

Some shells may provide a builtin `kill` command which is similar or identical to this utility. Consult the `builtin(1)` manual page.

-- MOST: \*stdin\*

(1,1) 0%

Press `Q' to quit, `H' for help, and SPACE to scroll.

# Esempio

```
$ cc -Wall ../01-intro/simple-shell.c
```

```
$ ./a.out
```

```
$$ ^C
```

```
$ echo $?
```

```
130
```

```
$ cc -Wall ../01-intro/simple-shell2.c
```

```
$ ./a.out
```

```
$$ ^C
```

```
Caught SIGINT!
```

# signal ( 3 )

```
#include <signal.h>
```

```
void (*signal(int signo, void (*func)(int)))(int);
```

Returns: previous disposition of signal if OK, SIG\_ERR otherwise

- func può essere:
  - SIG\_IGN che consente di ignorare il segnale `signo`
  - SIG\_DFL che consente di accettare l'azione di default per il segnale `signo`
  - o l'indirizzo di una funzione che può catturare e gestire il segnale

# Esempio

```
$ cc -Wall siguser.c
$ ./a.out
^Z
$ bg
$ ps | grep a.ou[t]
11106 ttys002    0:00.00 ./a.out
$ kill -USR1 11106
received SIGUSR1
$ kill -USR2 11106
received SIGUSR2
$ kill -INT 11106
$
[2]-  Interrupt                ./a.out
$
```

# sigaction(2)

```
#include <signal.h>

int sigaction(int signo, const struct sigaction *act, struct sigaction *oact);
```

- consente di esaminare o modificare le azioni associate ad un segnale

```
struct sigaction {
    void (*sa_handler)();    /* addr of signal handler, or
                             SIG_IGN or SIG_DFL */
    sigset_t sa_mask;        /* additional signals to block */
    int sa_flags;            /* signal options */
};
```

- `signal(3)` è implementata attraverso `sigaction(2)`

# Startup

- Quando un programma è `exec`-guito: :)
  - lo status dei segnali è `default` o `ignore`
- Quando un programma chiama `fork(2)`
  - il figlio eredita la disposizione dei segnali del padre
- Limite di `signal(3)`:
  - si può determinare la disposizione di un segnale solo settandola espressamente

# System Call - Interruzioni

- Alcune system call possono bloccare per lungo tempo. Alcuni esempi:
  - `read(2)` da file che possono bloccare (pipes, networks, ecc.)
  - `write(2)` sui file sopra
  - `open(2)` su un device che aspetta una determinata condizione (es., un modem)
  - `pause(3)` che mette in sleep un processo finché un segnale non si verifica
- Catturare un segnale durante l'esecuzione di queste call porta ad abortirle
  - `errno == EINTR`
- Le implementazioni moderne fanno ripartire automaticamente alcune system calls.

# Esempio

```
$ cc -Wall eintr.c
$ ./a.out
^C
read call was interrupted
||
$ ./a.out
^\a

read call was restarted
|a|
$
```

# Homework

- pagine man dei contenuti visti
- studiare Stevens, cap. 7, 10

# Es. 5

- Scrivere un programma che simula `grep(1)`
  - ricerca stringhe

# Es. 6

- Scrivere un programma che simula `cut(1)`
  - selezione di sottocampi

# Es. 7

- Utilizzare la semplice implementazione di `cat(1)`
  - sul file “queries.txt” (pagina del corso)
  - per cercare la sottostringa “google”
  - e selezionare la terza colonna
- usando il pipelining