Programación de Sistemas CCPG1008

Federico Domínguez, PhD.

Unidad 6 – Sesión 1: Procesos y señales

Contenido

- Procesos
- Creación de procesos
- UNIX Signals

¿Qué es un **proceso**?

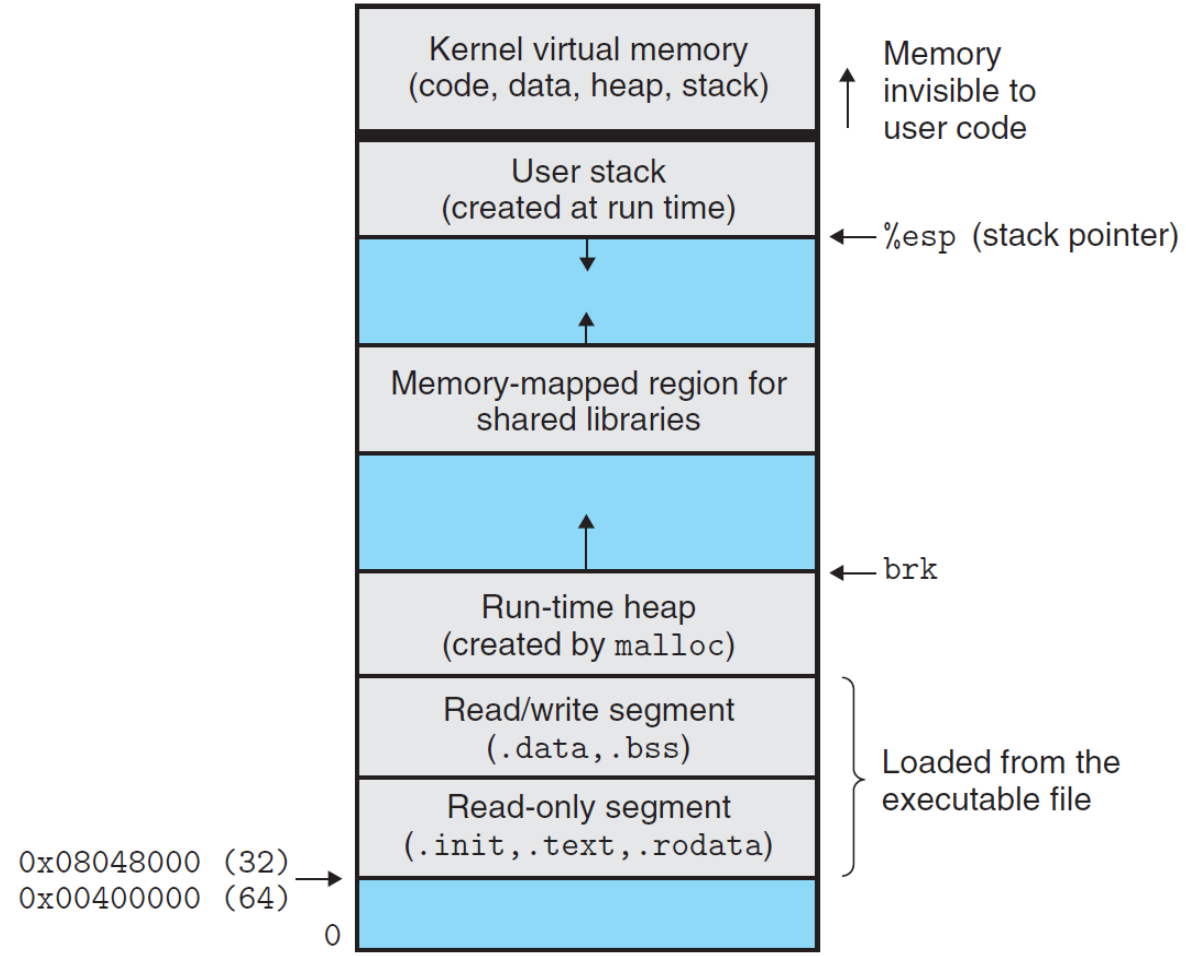
Un proceso es una instancia de un programa en ejecución.

Cada programa en el sistema es ejecutado en el contexto de un proceso.

El contexto de un proceso es:

- Memoria estática y código en memoria
- Stack y Heap
- Contenidos de los registros
- Program Counter (PC)
- Variables de ambiente
- Descriptores de archivos abiertos

Cada proceso tiene un espacio **privado** de memoria, dando la impresión que tiene toda la memoria para si mismo.

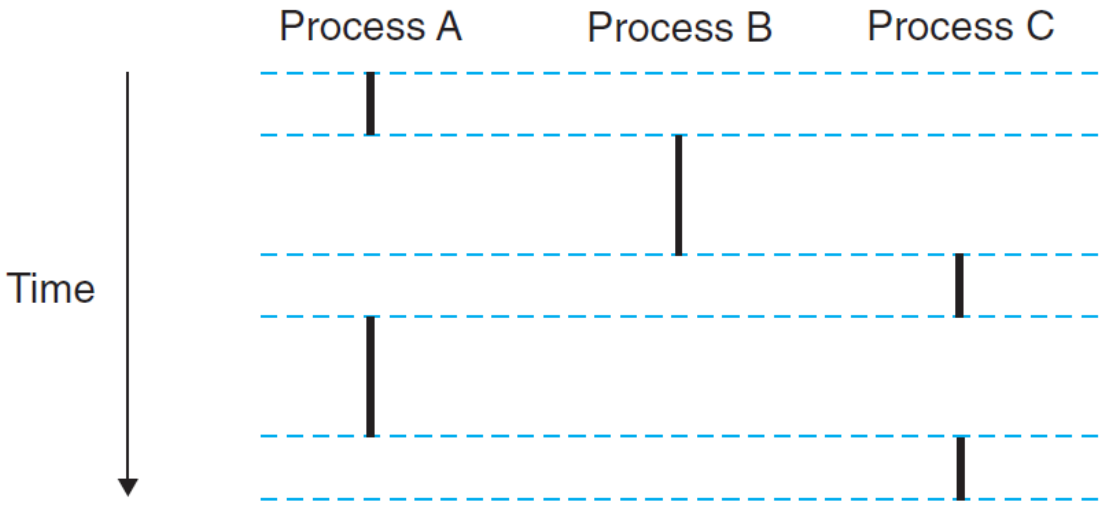


El *kernel* mantiene la ilusión de multitarea mediante **cambio de contexto**.

El *kernel* mantiene un contexto para cada proceso.

La ilusión de que en un solo procesador se ejecutan varios procesos de manera simultánea se denomina concurrencia y se obtiene interrumpiendo un proceso para permitir que otro se ejecute.

Al interrumpir un proceso, el *kernel* cambia al contexto de un proceso a otro.



Todo proceso tiene un identificador o *pid*.

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

pid_t getpid(void);
pid_t getppid(void);
```

Returns: PID of either the caller or the parent

Desde la perspectiva de un programador, un proceso puede estar en tres estados.

Running: Está siendo ejecutado o esperando a ser ejecutado.

Stopped: El proceso ha sido detenido al recibir una señal como SIGSTOP, no será ejecutado (volver al estado *Running*) a menos que reciba la señal SIGCONT.

Terminated: El proceso ha sido detenido permanentemente. Esto puede suceder al recibir una señal como SIGKILL, retornar de la función *main* o llamar a la función *exit().*

```
#include <stdlib.h>

void exit(int status);
```

This function does not return

Un proceso crea un proceso hijo mediante la función *fork().*

La función *fork()* crea un proceso casi idéntico al proceso padre, su contexto es una copia exacta del padre. La diferencia principal es tener un diferente ***pid***.

La función *fork()* es llamada una vez pero retorna dos veces, en el proceso padre retorna el ***pid*** del hijo y en el proceso hijo retorna 0.

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

pid_t fork(void);
```

Returns: 0 to child, PID of child to parent, −1 on error

Los procesos padres e hijo se ejecutan de manera concurrente y su orden es no determinístico.

code/ecf/fork.c

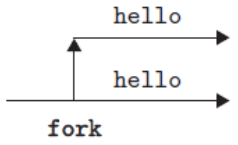
```
1   #include "csapp.h"
2
3   int main()
4   {
5       pid_t pid;
6       int x = 1;
7
8       pid = Fork();
9       if (pid == 0) {  /* Child */
10          printf("child : x=%d\n", ++x);
11          exit(0);
12      }
13
14      /* Parent */
15      printf("parent: x=%d\n", --x);
16      exit(0);
17  }
```

code/ecf/fork.c

```
unix> ./fork
parent: x=0
child : x=2
```

Múltiples llamadas a *fork()* pueden ser visualizadas usando grafos.

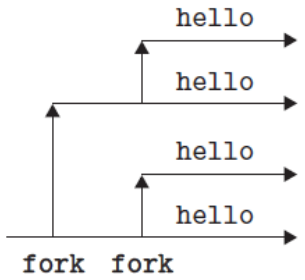
```
1   #include "csapp.h"
2
3   int main()
4   {
5       Fork();
6       printf("hello\n");
7       exit(0);
8   }
```



(c) Calls `fork` twice

```
1   #include "csapp.h"
2
3   int main()
4   {
5       Fork();
6       Fork();
7       printf("hello\n");
8       exit(0);
9   }
```

(d) Prints four output lines



Un proceso padre debe de **esperar** y "*recoger*" a sus hijos.

Cuando un proceso termina, por cualquier razón, su contexto no es eliminado inmediatamente por el kernel.

El proceso es eliminado completamente en el momento que el proceso padre lo recoge explícitamente.

Un proceso que ha terminado y aún no ha sido recogido por su padre es un proceso "***zombie***".

Un proceso existente en el sistema y cuyo padre ha terminado es un proceso **huérfano**. El proceso huérfano es recogido eventualmente por el proceso *init*.

Un proceso puede esperar y recoger a sus hijos usando las funciones *waitpid* y *wait*.

La función *waitpid* bloquea hasta que el proceso especificado (con el parámetro *pid*) termine; retorna el *pid* del proceso terminado.

Si el parámetro *pid* es -1, la función espera por el primer proceso hijo que termine.

El parámetro *status* representa el estatus de salida del proceso hijo.

El parámetro *options* permite configurar el comportamiento de *waitpid*, por ejemplo con WNOHANG la función retorna inmediatamente si no existe ningún hijo *zombie*.

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

pid_t waitpid(pid_t pid, int *status, int options);
```

Returns: PID of child if OK, 0 (if WNOHANG) or −1 on error

La función *wait* es una versión simplificada de *waitpid*.

Equivale a:

```
waitpid(-1, &status, 0)
```

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

pid_t wait(int *status);
```

Returns: PID of child if OK or −1 on error

code/ecf/waitpid1.c

```c
#include "csapp.h"
#define N 2

int main()
{
    int status, i;
    pid_t pid;

    /* Parent creates N children */
    for (i = 0; i < N; i++)
        if ((pid = Fork()) == 0)  /* Child */
            exit(100+i);

    /* Parent reaps N children in no particular order */
    while ((pid = waitpid(-1, &status, 0)) > 0) {
        if (WIFEXITED(status))
            printf("child %d terminated normally with exit status=%d\n",
                   pid, WEXITSTATUS(status));
        else
            printf("child %d terminated abnormally\n", pid);
    }

    /* The only normal termination is if there are no more children */
    if (errno != ECHILD)
        unix_error("waitpid error");

    exit(0);
}
```

code/ecf/waitpid1.c

```
unix> ./waitpid1
child 22966 terminated normally with exit status=100
child 22967 terminated normally with exit status=101
```

La función *execve* carga y ejecuta un programa desde el contexto de un proceso.

Al ejecutar *execve*, se carga el archivo ejecutable *filename* y se lo ejecuta en el contexto del proceso actual.

La función *execve* nunca retorna, simplemente pasa el control del proceso al código contenido en *filename*.

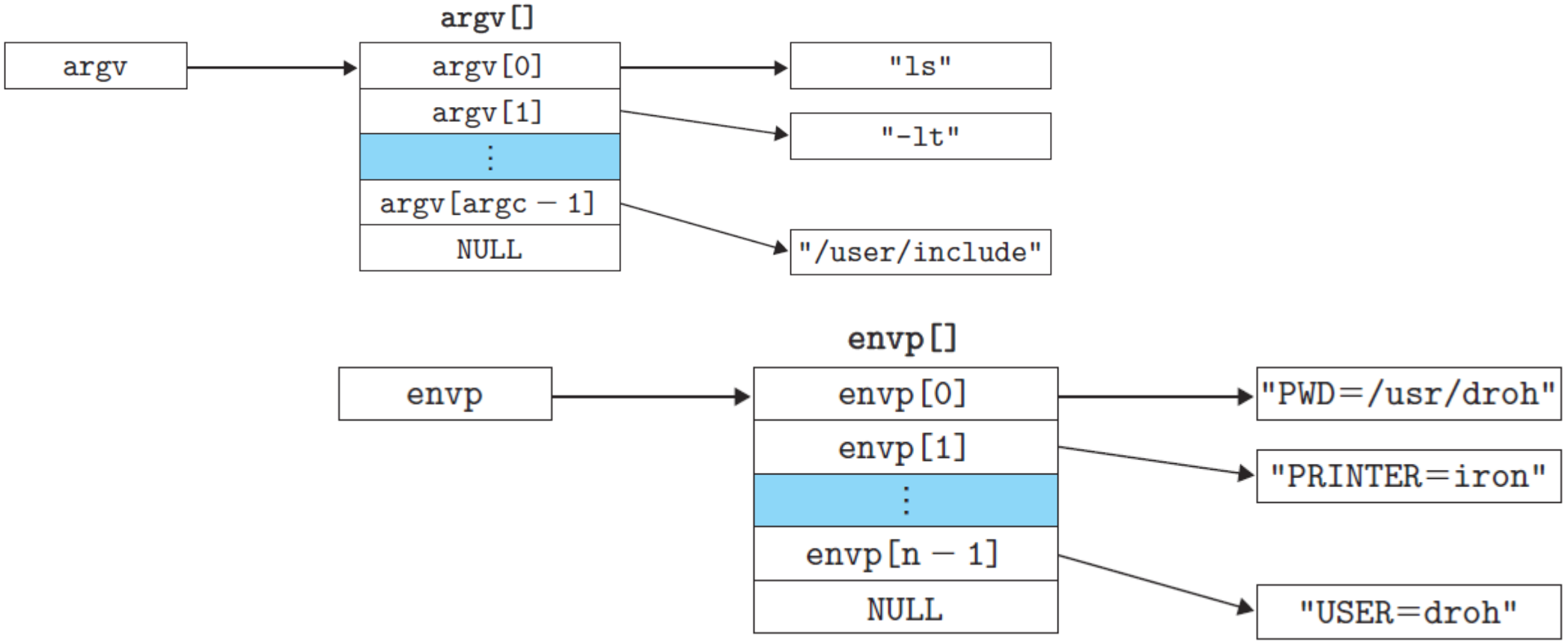
Es una forma de cambiar el programa en un proceso.

```
#include <unistd.h>

int execve(const char *filename, const char *argv[],
           const char *envp[]);
```

Does not return if OK, returns −1 on error

Los parámetros *arg* y *envp* contienen los argumentos y variables de ambiente que se enviarán al programa a ejecutar.



Combinando *fork* y *execve*, podemos hacer que un proceso ejecute archivos ejecutables.

```
if ((pid = Fork()) == 0) {    /* Child runs user job */
    if (execve(argv[0], argv, environ) < 0) {
        printf("%s: Command not found.\n", argv[0]);
        exit(0);
    }
}
```

Demostración *fork()*

UNIX Signals

UNIX Signal: Un mecanismo de control de flujo de programas.

Una señal permite que el *kernel* u otros procesos interrumpan la ejecución de un proceso.

Una señal es un mensaje asincrónico que notifica al proceso que un evento ha ocurrido.

Las señales permiten a los procesos de usuarios responder a eventos de bajo nivel (por ejemplo hardware) que son usualmente gestionados por el *kernel*.

Existen 30 diferentes tipos de señales que un sistema UNIX puede generar.

Number	Name	Default action	Corresponding event
1	SIGHUP	Terminate	Terminal line hangup
2	SIGINT	Terminate	Interrupt from keyboard
3	SIGQUIT	Terminate	Quit from keyboard
4	SIGILL	Terminate	Illegal instruction
5	SIGTRAP	Terminate and dump core (1)	Trace trap
6	SIGABRT	Terminate and dump core (1)	Abort signal from `abort` function
7	SIGBUS	Terminate	Bus error
8	SIGFPE	Terminate and dump core (1)	Floating point exception
9	SIGKILL	Terminate (2)	Kill program
10	SIGUSR1	Terminate	User-defined signal 1
11	SIGSEGV	Terminate and dump core (1)	Invalid memory reference (seg fault)
12	SIGUSR2	Terminate	User-defined signal 2
13	SIGPIPE	Terminate	Wrote to a pipe with no reader
14	SIGALRM	Terminate	Timer signal from `alarm` function
15	SIGTERM	Terminate	Software termination signal
16	SIGSTKFLT	Terminate	Stack fault on coprocessor
17	SIGCHLD	Ignore	A child process has stopped or terminated
18	SIGCONT	Ignore	Continue process if stopped
19	SIGSTOP	Stop until next SIGCONT (2)	Stop signal not from terminal

...

Es posible enviar una señal a un proceso usando *kill*.

```
unix> /bin/kill -9 15213
```

Este comando envía la señal SIGKILL al proceso 15213.

```
unix> /bin/kill -2 15213
```

Este comando envía la señal SIGINT al proceso 15213.

La función de sistema *kill()* permite enviar señales a otros procesos de manera programática.

Si *pid* es positivo, la señal es enviada al proceso, de lo contrario al grupo.

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

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

Returns: 0 if OK, −1 on error

La función *alarm()* permite a un proceso enviar una señal SIGALRM a si mismo.

La función *alarm()* puede ser usada para la implementación de un *timer*.

```
#include <unistd.h>

unsigned int alarm(unsigned int secs);
                    Returns: remaining secs of previous alarm, or 0 if no previous alarm
```

Un proceso puede registrar un *signal handler* con el *kernel* para gestionar la recepción de señales.

```
#include <signal.h>
typedef void (*sighandler_t)(int);

sighandler_t signal(int signum, sighandler_t handler);
```

Returns: ptr to previous handler if OK, SIG_ERR on error (does not set errno)

Ejemplo de uso de un *signal handler* para implementar una cuenta regresiva...

```
unix> ./alarm
BEEP
BEEP
BEEP
BEEP
BEEP
BOOM!
```

code/ecf/alarm.c

```
1   #include "csapp.h"
2
3   void handler(int sig)
4   {
5       static int beeps = 0;
6
7       printf("BEEP\n");
8       if (++beeps < 5)
9           Alarm(1); /* Next SIGALRM will be delivered in 1 second */
10      else {
11          printf("BOOM!\n");
12          exit(0);
13      }
14  }
15
16  int main()
17  {
18      Signal(SIGALRM, handler); /* Install SIGALRM handler */
19      Alarm(1); /* Next SIGALRM will be delivered in 1s */
20
21      while (1) {
22          ;  /* Signal handler returns control here each time */
23      }
24      exit(0);
25  }
```

code/ecf/alarm.c

Demostración *signal()*

Referencias

Libro texto *Computer Systems: A programmers perspective*. Secciones 8.2-5