Prácticas de Sistemas operativos

David Arroyo Guardeño

Escuela Politécnica Superior de la Universidad Autónoma de Madrid

Segunda Semana: Procesos, Procesos Padre y Procesos Hijo, familia exec()

1. *Entregas*
2. *Procesos*
   - *Fork*
   - *Estados de un proceso*
   - *Terminación de procesos*
   - *Wait*
   - *Exec*

Entregas

- ✓ Ejercicios 2, 3, 4, 5,6, 7 y 8
- ☠ Ejercicios 4, 5, 6 y 8

Definición de proceso

- ✗ Instancia de un programa en ejecución
  - ✓ Programa = 1 o más procesos
  - ✓ Proceso $\rightarrow$ espacio en memoria + datos: conjunto de estructuras de datos
- ✗ Elemento central en un SO
- ✗ SO $\leftrightarrow$ gestión de procesos

Programa Colección de instrucciones y de datos almacenados en un fichero (a.out)

- ✓ Conjunto de cabeceras $\rightarrow$ atributos del fichero
- ✓ Texto del programa $\rightarrow$ intrucciones lenguaje máquina
- ✓ bss: *block started by symbol* $\rightarrow$ datos a inicializar al arrancar
- ✓ Otras secciones

Proceso Programa leído por el núcleo y cargado en memoria para ejecutarse

- ✓ Segmento de texto → instrucciones CPU
- ✓ Segmento de datos → variables estáticas y globales
- ✓ Segmento de pila → marcos de pila

Marcos de pila

- ✓ Ejecución función $\rightarrow$ marco de pila
- ✓ Información necesaria para restaurar el marco de pila anterior a la llamada
- ✓ Contador de programa y puntero de pila anteriores a la llamada a la función

Modos de ejecución

Modo usuario Pila: argumentos, variables locales, datos funciones en modo usuario

Modo supervisor Marcos de pila de las llamada a sistema

Estructuras de control del SO

- ✗ Tabla de memoria
- ✗ Tabla de dispositivos de E/S
- ✗ Tabla de ficheros
- ✗ Tabla de procesos

Estructuras de control del SO

- Tabla de memoria
  - Gestión de la memoria principal y secundaria
  - Asignación de memoria a procesos, atributos de protección de memoria, información para gestionar memoria
- Tabla de dispositivos de E/S
- Tabla de ficheros
- Tabla de procesos

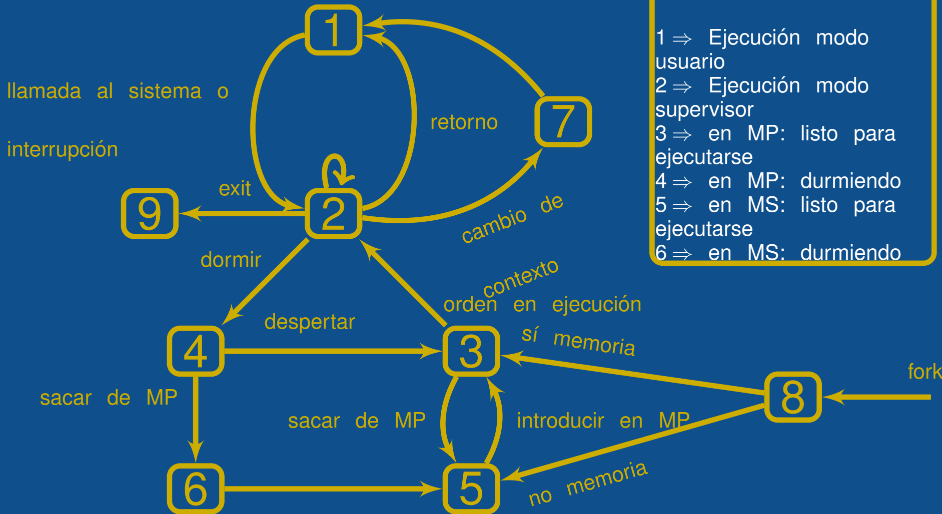
Estructuras de control del SO

- Tabla de memoria
- Tabla de dispositivos de E/S
  - Administración de dispositivos y canales de E/S
  - Almacena si dispositivo asignado a proceso
  - Estado del dispositivo: información en curso
- Tabla de ficheros
- Tabla de procesos

fork → *man fork*

- ✗ Proceso en UNIX: entidad tras llamada a fork. . . salvo el proceso número 0
- ✗ Proceso padre $\xrightarrow{\textit{fork}}$ proceso hijo
- ✗ proceso → PID
- ✗ Proceso 0: tras arrancar el sistema $\xrightarrow{\textit{fork}}$ init (proceso 1 $\xrightarrow{\textit{arrancar}}$ /etc/rc*): proceso 0, intercambiador | gestión de la memoria virtual

Estados de un proceso



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

int main (int argc,char *argv [],char *env [])
{
  int i, a=1;

  for (i=0; i<10; i++){
    switch(fork()){
      case 0:
        a = 3;
        printf("a=%d\n",a); break;
    }
    printf(``a=%d\n",a);
  }
  exit(0);
}
```

Llamada a fork I

1. Verificar si espacio en tabla de procesos
2. Asignar memoria al proceso hijo
3. Imagen proceso padre $\rightarrow$ memoria proceso hijo
4. Hallar una ranura de proceso libro y copiar la ranura del proceso padre en él
5. Asociar mapa de memoria a la tabla del proceso
6. Elegir un pid para el proceso hijo

Llamada a fork II

7. Notificar la existencia del proceso hijo al kernel y al sistema de archivo
8. Informar del mapa de la memoria del proceso hijo al kernel
9. Enviar mensajes de notificación al proceso padre y al proceso hijo

¿Por qué puede fallar fork?

- ✓ Hay que controlar la respuesta de fork

¿Por qué puede fallar fork?

- ✓ Hay que controlar la respuesta de fork
- ✓ No hay recuros suficientes

¿Por qué puede fallar fork?

- Hay que controlar la respuesta de fork
- No hay recuros suficientes
- Número máximo de procesos en ejecución

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

int main(int argc,char *argv[],char *env)
{
  int i = 0;

  switch (fork()){
  case -1:
    perror("Error al crear procesos");
    exit (-1);
    break;

 case 0:
   while (i<10) {
     sleep(1);
     printf("\t \t Soy el proceso hijo: %d\n", i++);
   }
   break;
 default:
   while (i<10){
     printf("Soy el proceso padre: %d\n", i++);
     sleep(2);
   }
  };
}
```

Terminación de procesos

```
#include <stdlib.h>
void exit (int status);
...
#include <sys/wait.h>
pid_t wait (int * stat_loc);
...

pid_t pid;
int estado;
...
pid = wait (&estado);

pid = wait(NULL);
```

```c
#include <sys/wait.h>
#include <stdlib.h>
#include <stdio.h>

#define NUM_PROCESOS 5

int I = 0;
void codigo_del_proceso(int id)
{
  int i;
  for(i=0;i<50;i++){
    printf("Proceso %d: i = %d, I = %d\n", id, i, I++);
  }
  exit(id);
}
```

```
main()
{
  int p,pid,salida;
  int id[NUM_PROCESOS] = {1,2,3,4,5};
  for (p=0;p<NUM_PROCESOS;p++){
    pid = fork();
    if(pid == -1){
      perror("Error al crear un proceso: ");
      exit(-1);
    } else if (pid == 0)
      codigo_del_proceso(id[p]);
  }

  for (p=0; p < NUM_PROCESOS; p++){
    pid = wait(&salida);
    printf("Proceso %d con id = %x terminado\n",pid,salida>>8);
  }
}
```

```c
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>

int main(){
  pid_t pid, ppid;

  //obtener el pid del proceso
  pid = getpid();

  //obtener el pid del padre del proceso
  ppid = getppid();


  printf("Mi pid e: %d\n",pid);
  printf("El pid de mi padre es %d\n", ppid);

  return 0;
}
```

Familia exec

- No se debe realizar ningún tratamiento tras llamada a exec
- Sólo tratamiento de error: exec no vuelve a la función que la llaman a menos que se produzca un error

```c
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>

int main(int argc, char * argv[]){

  //Para ejecutar ls, los argmuentos de entrada son: ls -l /bin
  char * ls_args[4] = { "/bin/ls", "-l", "/bin", NULL} ;

  //ejecuta ls
  execv( ls_args[0], ls_args);

  //aqui solo se llega si se ha producido un error
  perror("execv ha fallado");

  return 2; //se devuelve codigo de error
}
}
```

```c
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>

int main(int argc, char * argv[]){


  char * ls_args[4] = { "ls", "-l", "/bin", NULL} ;



  execvp( ls_args[0], ls_args);


  perror("execvp ha fallado");

  return 2;
}
```

```c
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>
#include <sys/types.h>
#include <sys/wait.h>

int main(int argc, char * argv[]){
  char * ls_args[4] = { "ls", "-l", "/bin", NULL} ;
  pid_t pid_0, pid;
  int status;
  pid_0 = fork();
  if (pid_0 == 0){
    /* HIJO */
    printf("Hijo: ejecutando ls\n");
    execvp( ls_args[0], ls_args);
    perror("execvp ha fallado");
  }else if (pid_0 > 0){
    /* PADRE */
    if( (pid = wait(&status)) < 0){
      perror("wait ha fallado");
      _exit(1);
    }
    printf("Padre: he terminado\n");

  }else{
    perror("Ha fallado el fork");
    _exit(1);
  }
  return 0; //retorno con exito
}
```

Referencias

⇒ Francisco M. Márquez. Unix, Programación Avanzada. Editorial: Ra-Ma. 3ª Edición. ISBN: 84-7897-603-5