INGENIERÍA | DEPARTAMENTO
UNIVERSIDAD DE LOS ANDES | DE COMPUTACIÓN
MÉRIDA VENEZUELA | ESCUELA DE SISTEMAS

Laboratorio de Sistemas Operativos
Semestre A-2018
Práctica Tres
Laboratorio

Prof. Rodolfo Sumoza
Prep. Alvaro Araujo
Prep. Luis Sanchez

2 de abril de 2018

1. Procesos.

Se puede definir de manera breve a un proceso como un programa que se encuentra en ejecución. Vale la pena resaltar que no es lo mismo un programa que un proceso; un programa se puede ver como una entidad pasiva, un archivo que contiene una serie de instrucciones que se deben llevar a cabo, mientras que un proceso involucra mucho mas; se puede ver a un proceso como una entidad activa que incluye un conjunto de elementos, entre ellos el código fuente(denominada sección de texto), también incluye una pila y una sección de datos las cuales utiliza el proceso para almacenar la información que maneja, un proceso también incluye la actividad actual, es decir, el valor del contador del programa.

Es posible la existencia de dos procesos asociados a un mismo programa, sin embargo, estos dos programas son dos secuencias de ejecución separadas (No guardan mas relación que su sección de texto la cual es la misma en ambos procesos). Para concluir nuestra diferencia entre un proceso y un programa, se puede pensar en ejecutar el mismo programa en dos maquinas diferentes, se creará un proceso diferente en cada maquina que aunque tienen la misma sección de texto, cada uno de estos procesos tendrá un estado, una pila y un heap diferente.

Para identificar un proceso se puede consultar su PID (process identification), este es un numero único que se le asigna a cada proceso. Ademas de su PID un proceso tiene otros datos asociados a el, para explorar un poco estos datos ejecutemos el siguiente comando

```
user@pc:/$ ps aux
```

1.1. Creación de nuevos procesos.

Por lo general en LINUX los procesos se crean a través de una "clonación" de los procesos antiguos, este procedimiento se puede resumir en tres pasos:

- El proceso padre realiza una llamada al sistema (**fork**) indicando que se desea "bifurcar", para generar un nuevo proceso.
- El sistema operativo crea una copia exacta del proceso padre (tanto código como datos), esta copia se crea como un nuevo proceso independiente, a este nuevo proceso se le denomina proceso hijo.
- Los dos procesos (padre e hijo) continúan su ejecución.

A continuación se presenta un ejemplo de la creación de nuevos procesos a partir de un proceso padre:

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <unistd.h>
4  #include <sys/types.h>
5
6  typedef struct
7  {
8    unsigned char dia;
9    unsigned char mes;
10   unsigned short int anio;
11 }t_fecha;
12
13 void funcion_generica(t_fecha *fecha);
14 float numero;
15
16 int main() {
17
18   t_fecha fecha;
19   pid_t pid;
20   numero = 3.141593;
21   funcion_generica(&fecha);
22
```

```
23    pid = fork();
24
25    if (pid == -1)
26    {
27      perror("Error al crear nuevo Hijo.\n");
28      return(1);
29    }
30
31    if (pid)
32    {
33      printf("\n\t -- Mi PID es %d, Soy el padre de %d\n
          ", getpid(), pid);
34      printf("\t -- Mi Padre es %d.\n ", getppid());
35      printf("\t -- Numero: %.4f\n", numero);
36      printf("\t -- Fecha: %d/%d/%d.\n ", fecha.dia,
          fecha.mes,fecha.anio);
37      fflush(stdout);
38    }
39    else
40    {
41      printf("\n\t ~~ Mi PID es %d, Soy el hijo de %d\n"
          , getpid(), getppid());
42      printf("\t ~~ Numero: %.4f\n", numero);
43      printf("\t ~~ Fecha: %d/%d/%d.\n\n", fecha.dia,
          fecha.mes,fecha.anio);
44      fflush(stdout);
45      fecha.dia++;
46      fecha.mes++;
47      fecha.anio++;
48    }
49
50    printf("\t --> Fecha: %d/%d/%d.\n\n", fecha.dia,
        fecha.mes,fecha.anio);
51    sleep(7);
52    return(0);
53  }
54
55  void funcion_generica(t_fecha *fecha)
56  {
57    fecha->dia = 28;
58    fecha->mes = 07;
59    fecha->anio = 2016;
60  }
```

1.2. Remplazar el código de un proceso.

Luego de ejecutar una llamada al sistema **fork** se puede modificar la ejecución del proceso hijo realizando una de las llamadas al sistema de la familia **exec**. En este caso utilizaremos la llamada **execvp**, que está definida de la siguiente manera:

```
int execvp(const char *file, char *const argv[]);
```

En el siguiente ejemplo se creara un proceso a partir del proceso original y este nuevo proceso ejecutará el código del ejemplo 1. A continuación se muestra el programa:

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <unistd.h>
4  #include <sys/types.h>
5  #include <sys/wait.h>
6
7  int main()
8  {
9    pid_t pid;
10   int estado;
11   char *argumentos[] = {"./fork01"};
12
13   pid = fork();
14
15   if(pid == -1)
16   {
17     perror("Error en la ejecucion del fork");
18     return 0;
19   }
20
21   switch(pid)
22   {
23     case 0:
24         execvp(argumentos[0], argumentos);
25         perror("Error en la ejecucion del exec");
26         return 0;
27
28     default:
29         wait(&estado);
30         if(!estado)
31           printf("\n\tCulmino la ejecucion del proceso
                hijo \n");
32         else
```

```
33            printf("\n\tError en la ejecucion del
                  proceso hijo \n");
34          break;
35    }
36    return 0;
37  }
```

1.3. Espera por culminación de procesos hijos.

Cuando se ejecuta un programa desde la consola se pueden devolver valores al sistema operativo mediante el valor de retorno de la función principal[1], cuando el programa termine su ejecución. Para realizar esta espera contamos principalmente con dos funciones **wait** y **waitpid**. A continuación se muestra un ejemplo de su funcionamiento [2]:

```
1  #include <sys/wait.h>
2  #include <stdlib.h>
3  #include <unistd.h>
4  #include <stdio.h>
5
6  int main(int argc, char *argv[])
7  {
8    pid_t cpid, w;
9    int wstatus;
10   cpid = fork();
11
12   if (cpid == -1)
13   {
14     perror("fork");
15     exit(EXIT_FAILURE);
16   }
17
18   if (cpid == 0)
19   {
20     /* Code executed by child */
21     printf("Child PID is %ld\n", (long) getpid());
22     if (argc == 1)
23       pause();                    /* Wait for signals
             */
24     _exit(atoi(argv[1]));
25   }
26   else
27   {
```

[1]Ejemplo footnote.
[2]Ejemplo tomado de *Linux Programmer's Manual*, Thomas Koenig y Michael Kerrisk.

```
28      /* Code executed by parent */
29      do
30      {
31        w = waitpid(cpid, &wstatus, WUNTRACED |
              WCONTINUED);
32        if (w == -1)
33        {
34          perror("waitpid");
35          exit(EXIT_FAILURE);
36        }
37        if (WIFEXITED(wstatus))
38        {
39          printf("exited, status=%d\n", WEXITSTATUS(
                wstatus));
40        }
41        else if (WIFSIGNALED(wstatus))
42        {
43          printf("killed by signal %d\n", WTERMSIG(
                wstatus));
44        }
45        else if (WIFSTOPPED(wstatus))
46        {
47          printf("stopped by signal %d\n", WSTOPSIG(
                wstatus));
48        }
49        else if (WIFCONTINUED(wstatus))
50        {
51          printf("continued\n");
52        }
53      } while (!WIFEXITED(wstatus) && !WIFSIGNALED(
            wstatus));
54      exit(EXIT_SUCCESS);
55    }
56  }
```

2. Hilos de ejecución

Un hilo de ejecución es una unidad básica de utilización de la CPU; posee un ID de hilo, un contador de programa, un conjunto de registros y una pila. Los hilos que conforman un determinado proceso comparten la sección de texto o código, la sección de datos y otros recursos del sistema operativo.

2.1. Creacion de un Hilo (thread)

Para crear un nuevo hilo de ejecución se puede utilizar la función **pthread_create**. El prototipo de esta función es el siguiente:

```
pthread_create(pthread_t *thread, pthread_attr_t *attr
    , void * (*start_routine)(void *), void *arg);
```

2.2. Esperar por la culminación de un Hilo

En ocasiones para controlar la concurrencia, es necesario detener la ejecución de un hilo hasta que otro termine su ejecución, para lograr este objetivo se utiliza la función **pthread_join**, otra bondad que ofrece esta función es que permite obtener un valor de retorno en caso de que el hilo al que se espera lo retorne. El prototipo de esta función es el siguiente:

```
int pthread_join(pthread_t th, void **thread_return);
```

2.3. No guardar el resultado de ejecución de un Hilo

Por defecto el sistema guarda la información que genera un hilo al ejecutarse; para que luego con la función **pthread_join** recuperar esta información. En caso de que no interese conocer esta información y en pro de liberar el espacio ocupado y aumentar la capacidad de generar nuevos hilos, se puede indicar al sistema que no guarde dicha información; para ello se utiliza la función **pthread_detach**. El prototipo de esta función es el siguiente:

```
int pthread_detach(pthread_t th);
```

2.4. Terminar la ejecución de un Hilo

Para terminar la ejecución de un hilo basta con invocar la función **pthread_exit** que ademas de terminar la ejecución de un hilo permite retornar un valor que luego puede ser recuperado con **pthread_join**. Cabe destacar que la función donde se define el funcionamiento del hilo no retorna ningún dato; por lo tanto, una alternativa para retornar un valor sin utilizar una variable global es esta función. El prototipo de la función es el siguiente:

```
void pthread_exit(void *retval);
```

A continuación se muestra un ejemplo para ilustrar todo lo tratado hasta este punto:

```
1  #include <pthread.h>
2  #include <stdio.h>
3  #include <stdlib.h>
4  #include <string.h>
5
6  #define NRO_THR 5
7
8  pthread_t info_hilos[NRO_THR];
9
10 typedef struct
11 {
12   unsigned char dia;
13   unsigned char mes;
14   unsigned short int anio;
15 }t_fecha;
16
17 typedef struct
18 {
19   unsigned int id;
20   t_fecha fecha;
21 }t_hilo;
22
23 t_hilo parametros_hilos[NRO_THR];
24
25 void * manejador_hilos(t_hilo *pto)
26 {
27   printf("\n\tHILO ID: %d\n", pto->id);
28   printf("\tFecha: %d/%d/%d\n",pto->fecha.dia,pto->
       fecha.mes, pto->fecha.anio);
29   fflush(stdout);
30   pthread_exit((void *)(&(pto->id)));
31 }
32
33 int main()
34 {
35   int i;
36   t_hilo *retorno;
37
38   for (i = 0; i < NRO_THR; i++)
39   {
40     parametros_hilos[i].id = i;
41     parametros_hilos[i].fecha.dia = 10+i;
42     parametros_hilos[i].fecha.mes = 4+i;
```

```
43      parametros_hilos[i].fecha.anio = 2016+i;
44      pthread_create(&info_hilos[i],NULL,(void *)&
            manejador_hilos, (void *)&parametros_hilos[i]);
45    }
46
47    printf("\n\t*** Hilos de Ejecucion Creados! \n\n");
48
49    printf("\n");
50    for (i = 0; i < NRO_THR; i++)
51    {
52      pthread_join(info_hilos[i], (void *)&retorno);
53      printf("\t --> El Hilo Nro:%d, Retorno el valor: %
            d\n", i, retorno->id);
54      fflush(stdout);
55    }
56    printf("\n\t\a*** Todos los Hilos han terminado su
         ejecucion!\n\n");
57    return 0;
58  }
```

3. Ejercicios

3.1. -

Dibuja la estructura del árbol de procesos que se obtendría al ejecutar el siguiente fragmento de código:

```
for (num= 0; num< 2; num++)
{
  nuevo= fork(); /* 1 */
  if (nuevo== 0)
    break;
}
nuevo= fork();  /* 2 */
nuevo= fork();  /* 3 */
printf("\n\tSoy el proceso: %d y mi padre es: %d\n",
     getpid(), getppid());
```

3.2. -

Realice un programa que al ser ejecutado cree 2 procesos que emulen un juego de ping pong, luego de recibir la pelota uno de los procesos deberá esperar 2 segundos para devolver la pelota. Cabe destacar que alguno de los dos procesos debe realizar el saque, es decir comenzar la partida. El partido debe culminar luego de 7 toques de cada proceso.

3.3. -

Realice un programa que al ser ejecutado cree 2 hilos que emulen un juego de ping pong; el juego debe cumplir las mismas características que el ejercicio anterior.