Trabajo de Diseño y Administración de Sistemas Operativos

Alumno: Michael Laudrup Luis gonzalez

Segunda PED

Introducción

En está práctica tenemos

Implementación

Inicialmente hay que señalar que la mayoría de la función de invocación y creación de mecanismos IPCs se han aglutinado en el archivo de cabecera “IPCsUtilidades.h” y su correspondiente implementación en “IPCsUtilidades.c”.

Tal y como se hizo en el trabajo práctico 1 de Diseño y administración de sistemas operativos, dado que el código fuente está bien comentado y los nombres de las variables, funciones...etc. son muy explícitos por lo que me limitaré a exponer lo más importante.

Utilización de barrera: tal y como se dijo en el anterior apartado la tubería sin nombre “barrera” tiene como función informar a los procesos hijos cuando empieza una contienda enviándoles un mensaje. Lo más importante de esto es:

- para que un único proceso hijo no lea todos los mensajes de la tubería, sino que solo lea un único mensaje se ha puesto un if (read(barrera) en lugar de un while (read(barrera)), con esto se consigue que solo lea un mensaje, ¿pero en la siguiente contienda como leerá un nuevo mensaje? Esto se hace por que el if se ha envuelto con un while el cual solo parara cuando se reciba por la barrera como mensaje la palabra “FIN”. En resumen, el propio mensaje indica cuando se debe dejar de esperar en la barrera.
- Además, el mensaje cuando no es FIN, es un número aleatorio proporcionado por el padre, dado que la función rand() genera el mismo numero aleatorio para todos los hermanos si se utilizará en el archivo hijo.c, sin embargo, si se utiliza rand() en el archivo padre.c si que se generan números aleatorios por cada ronda.
- Finalmente, cuando cada bifurcación generada por la función fork() llega a la función execv, envía como parámetro los descriptores de la tubería barrera para que el hijo sepa donde leer.

```
do{
    if((read(barrera[0], mensaje, n) > 0)){
        if(strcmp(mensaje,"FIN") == -1){ //Si el resultado de comparar el mensaje con la palabra fin es -1 indica que le mensaje recibido no es fin sino un numero aleatorio
            int aleatorio = atoi(mensaje); //El numero aleatorio le llega por la barrera a traves del padre en cada ronda
            abs(aleatorio);
            strcpy(estado, "OK"); //Hay procesos que no sufren ninguna ataque y por lo tanto su estado no puede ser la cadena vacia
            if(aleatorio%2 == 0) {
                    printf("El proceso con pid: %d esta en modo ataque. \n", pid);
                    signal(SIGUSR1, indefenso);
                    usleep(100000);
                    fflush(stdout);
                    int pidBajoAtaque;
                    int aux = 0;
                    bool pidEncontrado =false;
                    do{
                       int indice = ((aleatorio)+aux)%n;
                       wait_sem(semid);
                       pidBajoAtaque = arrayPidsHijos[indice];
                       signal_sem(semid);
                       if( (pidBajoAtaque != 0) && (pidBajoAtaque != pid) ){
                            pidEncontrado = true;
                       }else{
                            aux = aux + 1;
                       }
                    }while(!pidEncontrado);
                printf("El proceso con id: %d va a atacar al proceso con id: %d\n", pid, pidBajoAtaque);
                kill(pidBajoAtaque, SIGUSR1);
                usleep(100000);
                enviarMensaje(getpid(),estado,msqid);
            }else{
                printf("El proceso con pid: %d esta en modo defensa. \n",pid);
                fflush(stdout);
                signal(SIGUSR1, defensa);
                usleep(200000);
                enviarMensaje(getpid(),estado,msqid);
            }
        }else {
            finRondas = true;
        }
    }else{
        perror("Error al leer la tuberia: ");
        exit(0);
    }
    contadorVueltas++; //El contador de vueltas permite por cada vuelta darle mas aleatoriedad a los numeros
  }while(!finRondas); //Fin del bucle que inicia rondas
```

Mientras no se lea el mensaje “FIN” el proceso seguirá esperando a leer “un único mensaje”.

NOTA IMPORTANTE: Más adelante me di cuenta de que como el proceso puede terminar los procesos hijos mediante la señal SIGTERM, y la llamada kill, por ello cambie la condicio del while a 1, es decir siempre estará dando vueltas hasta que el padre le mande a terminar. Se deja para ilustrar el proceso de aprendizaje en el desarrollo de la PEC

```
bool mensajeEsFin;
if(num_procVivos < 2){
     strcpy(mensaje,"FIN");
     mensajeEsFin = true;
}
for (int i = 0; i < num_procVivos; i++){
    if(!mensajeEsFin){
        int numAleatorio;
        numAleatorio = rand()%1000; //Se intenta que nada numero aleatorio cambien en cada vuelta para que no se repitan patrones
        char aleatorioEnTexto[num_procVivos]; //El num aleatorio tendra maximo 3 cifras
        sprintf(aleatorioEnTexto,"%d",numAleatorio);
        strcpy(mensaje, aleatorioEnTexto);
    }
    write(barrera[1], mensaje, strlen(mensaje)); //Se escribe mensaje en tuberia
    usleep(1); //El tiempo entre mensaje y mensaje es de 5 ns
}
int mensajesPendientes = num_procVivos;
```

En el código del padre, si el número de procesos vivos es menor que 2, se envía la palabra “FIN” como mensaje para que finalice ese proceso. Sucede los mismo, solo se manda como mensaje números aleatorios, ya no es necesaria la palabra FIN

```c
do{
    printf("El hijo con pid %d ha llegado a la barrera y esta esperando a recibir un mensaje\n", pidHijoActual);
    if((read(barreraLectura, mensajeBarrera, n) > 0)){
        int aleatorio = atoi(mensajeBarrera); //El numero aleatorio le llega por la barrera a traves del padre en cada
        abs(aleatorio);
        strcpy(estado, "OK"); //Hay procesos que no sufren ninguna ataque y por lo tanto su estado no puede ser la cade
        if(aleatorio%2 == 0) {
            printf("El proceso con pid: %d esta en modo ataque. \n", pidHijoActual);
            fflush(stdout);
            signal(SIGUSR1, indefenso);
            usleep(100000);
            int pidBajoAtaque;
            int aux = 0;  //rodamos cero posiciones inicialmente a partir del aleatorio obtenido
            bool pidEncontrado =false; //Se parte de la premisa de que el pid no se ha encontrado
            do{
                int indice = ((aleatorio)+aux)%n; //Obtenemos indice del vector, rodando una posicion en cada vuelta en
                wait_sem(semid); //se espera a disponibilidad de semaforo
                pidBajoAtaque = arrayPidsHijos[indice]; //Se asigna como pid a atacar el numero aleatorio recibido por
                signal_sem(semid); //Se libera semaforo
                if( (pidBajoAtaque != 0) && (pidBajoAtaque != pidHijoActual) ){ //Si el pid actual es diferente de cero
                    pidEncontrado = true;
                }else{
                    aux = aux + 1;
                }
            }while(!pidEncontrado);
            printf("El proceso con id: %d va a atacar al proceso con id: %d\n", pidHijoActual, pidBajoAtaque);
            fflush(stdout);
            kill(pidBajoAtaque, SIGUSR1);
            usleep(100000);
            enviarMensaje(pidHijoActual,estado,msqid);
        }else{
            printf("El proceso con pid: %d esta en modo defensa. \n",pidHijoActual);
            fflush(stdout);
            signal(SIGUSR1, defensa);
            usleep(200000);
            enviarMensaje(pidHijoActual,estado,msqid);
        }
    }else{
        perror("Error al leer la tuberia: ");
        exit(0);
    }
}while(1); //Lo va a estar haciendo infinitas veces hasta que el proceso padre le mande una SIGTERM
```

En este caso se muestra como el bucle while es infinito, porque sabemos de antemano que el padre tarde o temprano finalizará a la fuerza la ejecución del proceso hijo

```c
char mensaje[num_procVivos];
for (int i = 0; i < num_procVivos; i++){
    int numAleatorio;
    numAleatorio = rand()%1000; //Se intenta que nada numero aleat
    sprintf(mensaje,"%d",numAleatorio);
    write(barrera[1], mensaje, strlen(mensaje)); //Se escribe mens
    usleep(1); //El tiempo entre mensaje y mensaje es de 1 ns
}
int mensajesPendientes = num_procVivos;
usleep(400000); //Se esperan 4 milisegundos a que acabe la contien
while(mensajesPendientes > 0  ) { //COMIENZO DEL PROCESAMIENTO DE
```

En este caso se ha optado por enviar únicamente como mensajes números aleatorios para evitar que en un mismo instante de tiempo se generen números aleatorios iguales. Además, entre mensaje y mensaje se duerme un microsegundo para que le dé tiempo a algún proceso hijo a leerlo.

Semáforos: Se utiliza un único semáforo para controlar acceso a región compartida de la región de memoria apuntada por el puntero lista, el cual es un vector de números enteros. Cada vez que se desea consultar información de esta lista de pids o modificarla se utiliza el semáforo, se muestran dos secciones de código donde se usa el semáforo.

- Cuando el padre va creando procesos hijos va insertando cada PID en la región de memoria apuntada por lista con un desplazamiento de [x (int size)], es decir, si se pone lista[1] se desplazaría desde la posición inicial del puntero hasta 1 veces el tamaño de un entero en C.

```
int pidResultanteFork;
ushort valorSemaforo;
for(int i = 0; i < n; i++){
   pidResultanteFork = fork();
   if(pidResultanteFork == 0){
       char llaveEnTexto[15];
       char tuberia1[] = {'0'+barrera[0],'\0'};
       char tuberia2[] = {'0'+barrera[1],'\0'};
       char index[] = {'0'+i,'\0'};
       sprintf(llaveEnTexto,"%d",llave);
       char *const argumentos[] = {argv[1], llaveEnTexto, tuberia1, tuberia2,index, NULL};
       execv("./Trabajo2/hijo", argumentos);
       i = n;
   }else if(pidResultanteFork == -1){
       perror("Error en la creacion de un proceso hijo ");
       exit(1);
   }else{ //PADRE
       wait_sem(semid);//Espera a que el semaforo este disponible
       lista[i] = pidResultanteFork; //insertar pid de nuevo hijo #region critica
       signal_sem(semid);//Liberar el semaforo
   }
}
```

- Cuando el proceso hijo quiere consultar si un pid está puesto a cero, o si un pid es el mismo pid que si mismo, con el fin de encontrar un pid al que atacar:

```
if(aleatorio%2 == 0) {
        printf("El proceso con pid: %d esta en modo ataque. \n", pid);
        signal(SIGUSR1, indefenso);
        usleep(100000);
        fflush(stdout);
        int pidBajoAtaque;
        int aux = 0;  //rodamos cero posiciones inicialmente a partir del aleatorio obtenido
        bool pidEncontrado =false; //Se parte de la premisa de que el pid no se ha encontrado
        do{
           int indice = ((aleatorio)+aux)%n; //Obtenemos indice del vector, rodando una posicion en cada vuelta en cas
           wait_sem(semid); //se espera a disponibilidad de semaforo
           pidBajoAtaque = arrayPidsHijos[indice]; //Se asigna como pid a atacar el numero aleatorio recibido por mens
           signal_sem(semid); //Se libera semaforo
           if( (pidBajoAtaque != 0) && (pidBajoAtaque != pid) ){ //Si el pid actual es diferente de cero y del pid que
                pidEncontrado = true;
           }else{
                aux = aux + 1;
           }
        }while(!pidEncontrado);
```

- Cuando se desea alterar el valor de un determinado pid en la lista, según los mensajes recibidos en la cola de mensajes.

```
while(mensajesPendientes > 0) { //COMIENZO DEL PROCESAMIENTO DE LA COLA DE MENSAJES DESPUES DE LA CONTIENDA
    if(msgrcv(msqid, &mensajeCola, longitud, 2,0) == -1){
        perror("Error en la lectura de un mensaje: ");
    }else{
        if((mensajeCola.cadena[0])== 'K' ){   //Esta muy controlado el formato de mensaje recibido,tal que los dos pri
            char pid_hijo_texto[6];
            for(int i = 2; i <= 7; i++){ //Se cogen los ultimos 6 digitos del mensaje recibido
                pid_hijo_texto[i-2] = mensajeCola.cadena[i];
            }
            int pid_hijo_num = atoi(pid_hijo_texto);
            printf("Asesinato del proceso con pid %d\n", pid_hijo_num);
            if(kill(pid_hijo_num, SIGTERM) == -1){
                printf("Error en la terminacion del proceso con pid %d", pid_hijo_num);
                perror("SIGTERM");
                exit(0);
            }
            waitpid(pid_hijo_num, &status,0);
            for(int i = 0; i < n ; i++){
                wait_sem(semid);
                if(lista[i] == pid_hijo_num) {
                    lista[i] = 0;
                }
                signal_sem(semid);
            }
            num_procVivos--;
            proc_muertos++;
        }//FIN DE TRATAMIENTO EN CASO DE QUE UN PROCESO HAYA ESTADO KO
    }
    mensajesPendientes--;
} //FIN DEL PROCESAMIENTO DE LA COLA DE MENSAJES
```

Región de memoria compartida: ya en el anterior caso donde se explicaban los semáforos se vio el uso de la región compartida, como información complementaria se expone la inicialización de la memoria compartida y su compromiso con la región de código del padre y del hijo:

- Obtención del id de la memoria compartida y compromiso con el puntero lista.

```
int shmid = obtenerIDMemoriaCompartida(llave,n); //Este metodo esta en ipcsUtilidades.h
printf("Memoria compartida id:  %d \n", shmid);
int *lista;
lista = shmat(shmid,0,0);
```

- En el caso del hijo, la llave recibida como parámetro permite recuperar la región de memoria

compartida y vincular a el puntero de “arrayPidsHijos”.

```
int *arrayPidsHijos;
int shmid = shmget(llave,0, 0);
arrayPidsHijos = shmat(shmid,0,0);
```

Finalmente nos queda la cola de mensajes: cada vez que se invoca la función “defensa” o “indefenso”, al finalizar su procesamiento envía un mensaje a la cola de mensajes con mismo id que la cola de mensajes generada por el padre a través de llave se procede a mostrar el código relacionado con la cola de mensajes.

- Tras una ronda cada proceso envía el estado y el pid a través de un mensaje a la cola de mensajes, con un formato “OKXXXXX” o “KOXXXXX” donde cada X es un número natural, agrupadas las x se indica el pid del proceso que envía el mensaje, destacando que KO señala que el proceso debe ser puesto a cero por el padre y OK, el proceso padre no debe hacer nada con ese proceso. Se muestra el envió tras ronda de ataques:

```
if(aleatorio%2 == 0) {
        printf("El proceso con pid: %d esta en modo ataque. \n", pid);
        signal(SIGUSR1, indefenso);
        usleep(100000);
        fflush(stdout);
        int pidBajoAtaque;
        int aux = 0;  //rodamos cero posiciones inicialmente a partir del aleatorio obtenido
        bool pidEncontrado =false; //Se parte de la premisa de que el pid no se ha encontrado
        do{
           int indice = ((aleatorio)+aux)%n; //Obtenemos indice del vector, rodando una posicion en cada vuelta en cas
           wait_sem(semid); //se espera a disponibilidad de semaforo
           pidBajoAtaque = arrayPidsHijos[indice]; //Se asigna como pid a atacar el numero aleatorio recibido por mens
           signal_sem(semid); //Se libera semaforo
           if( (pidBajoAtaque != 0) && (pidBajoAtaque != pid) ){ //Si el pid actual es diferente de cero y del pid que
                pidEncontrado = true;
           }else{
                aux = aux + 1;
           }
        }while(!pidEncontrado);
    printf("El proceso con id: %d va a atacar al proceso con id: %d\n", pid, pidBajoAtaque);
    kill(pidBajoAtaque, SIGUSR1);
    usleep(100000);
    enviarMensaje(getpid(),estado,msqid);
}else{
    printf("El proceso con pid: %d esta en modo defensa. \n",pid);
    fflush(stdout);
    signal(SIGUSR1, defensa);
    usleep(200000);
    enviarMensaje(getpid(),estado,msqid);
```

Vemos que llaman al método enviarMensaje enviándoles como argumentos el pid del proceso hijo en ejecución, su estado (“KO” u “OK”) y el identificador de la cola de mensajes, el método implementado es el expuesto en la siguiente página:

```
void enviarMensaje(int pid, char estadoo[],int msqid){
    char pidTexto[6]; //Normalmente un pid tiene maximo 5 digitos, es raro ver un pid
    sprintf(pidTexto, "%d", pid);
    tMensajeCola mensajeCola;
    int longitud = sizeof(mensajeCola) -sizeof(mensajeCola.tipo);
    mensajeCola.cadena[0] = estado[0];
    mensajeCola.cadena[1] = estado[1];
    for(int i = 2; i <= 7; i ++) {
        mensajeCola.cadena[i] = pidTexto[i-2];
    }
    mensajeCola.cadena[8] = '\0'; //Marca de fin de cadena
    mensajeCola.tipo = 2;
    if(msgsnd(msqid, &mensajeCola, longitud, 0) == -1){
        perror("Ha habido un error durante el envio de un mensaje: ");
        exit(2);
    }
```

El padre procesa solo aquellos mensajes que empiezan por la letra “K” omitiendo el resto dado que no surten efecto en el funcionamiento del programa normal (no hay que poner ningún pid a cero). La recepción y procesamiento de los mensajes por parte del padre queda como sigue:

```
while(mensajesPendientes > 0) { //COMIENZO DEL PROCESAMIENTO DE LA COLA DE MENSAJES DESPUES DE LA CONTIENDA
    if(msgrcv(msqid, &mensajeCola, longitud, 2,0) == -1){
        perror("Error en la lectura de un mensaje: ");
    }else{
        if((mensajeCola.cadena[0])== 'K' ){   //Esta muy controlado el formato de mensaje recibido,tal que lo
            char pid_hijo_texto[6];
            for(int i = 2; i <= 7; i++){ //Se cogen los ultimos 6 digitos del mensaje recibido
                pid_hijo_texto[i-2] = mensajeCola.cadena[i];
            }
            int pid_hijo_num = atoi(pid_hijo_texto);
            printf("Asesinato del proceso con pid %d\n", pid_hijo_num);
            if(kill(pid_hijo_num, SIGTERM) == -1){
                printf("Error en la terminacion del proceso con pid %d", pid_hijo_num);
                perror("SIGTERM");
                exit(0);
            }
            waitpid(pid_hijo_num, &status,0);
            for(int i = 0; i < n ; i++){
                wait_sem(semid);
                if(lista[i] == pid_hijo_num) {
                    lista[i] = 0;
                }
                signal_sem(semid);
            }
            num_procVivos--;
            proc_muertos++;
        }//FIN DE TRATAMIENTO EN CASO DE QUE UN PROCESO HAYA ESTADO KO
    }
    mensajesPendientes--;
} //FIN DEL PROCESAMIENTO DE LA COLA DE MENSAJES
```

Finalmente, el ultimo mecanismo IPC es el archivo FIFO, el código como sigue:

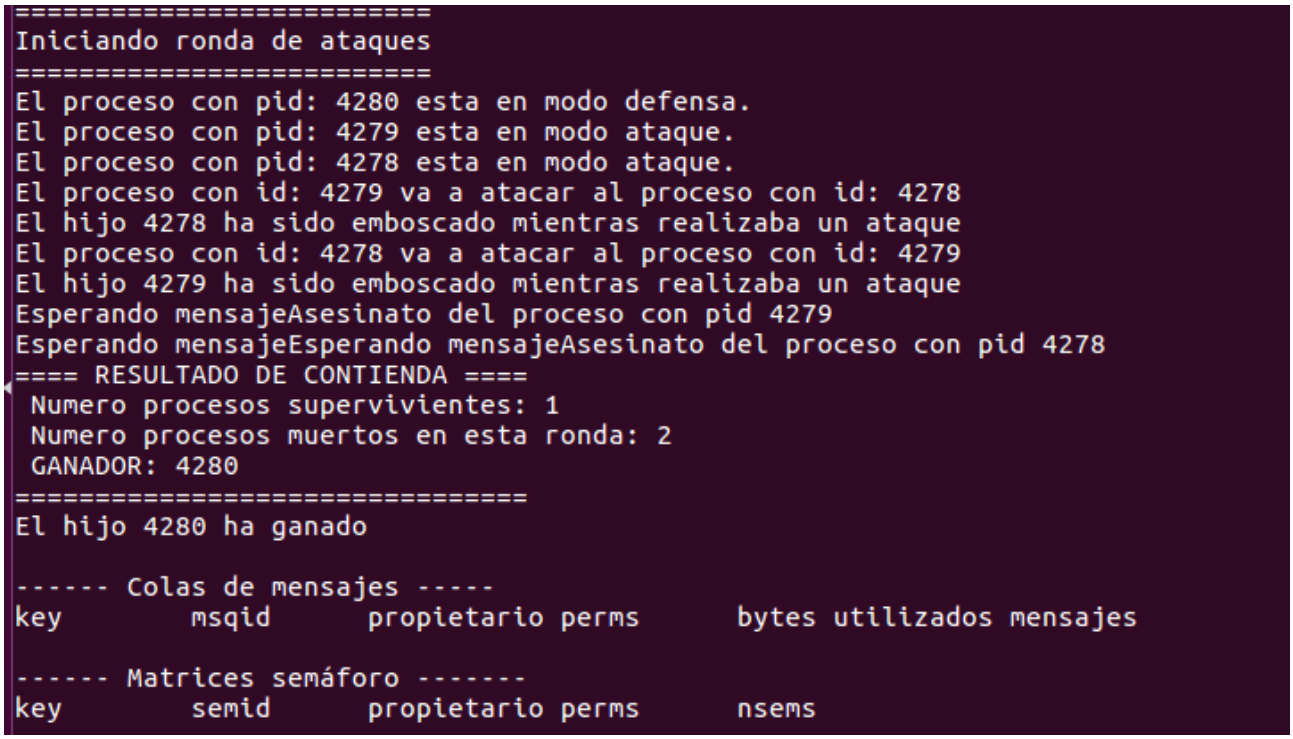
```
//printf("Mensaje que se debe almacena en la tuberia FIFO: %s", mensaje1); //Depurando el mensaje esta bien // SOLO PAR
char *direccion = "./resultado";
int descriptor_tuberia;

if( (descriptor_tuberia = open("./resultado", O_WRONLY)) ==-1){ perror("Error en la apertura de la tuberia: ");}
if((write(descriptor_tuberia, mensaje1, strlen(mensaje1)+1))==-1){perror("Error en la escritura sobre la tuberia: ");}
if( close(descriptor_tuberia) == -1){perror("Error en el ciere de la tuberia: ");}
```

El archivo fifo se crea en el archivo Ejercicio2.sh y luego se lanza un cat en segundo plano hasta que haya algo que leer, en el anterior codigo se rescata el archivo resultado creado por el bash script, se obtiene su descriptor con la clausula open, se escribe sobre el descriptor el resultado de la contienda y se cierra el descriptor.

Ejecución de ejemplo

Lo primero que hay que señalar es que la probabilidad de empate o de obtener un ganador, depende mucho de cómo se trate la aleatoriedad de asignación de procesos hijos en modo defensa o en modo ataque y del valor de “n”, para n=10 tiende a que acabe en empate, aunque si se repite la ejecución varias veces dará 1 de cada 7 veces un ganador, por ello, es recomendable probar diferentes valores de “n”, dado que hay otros valores de “n” que hace más probable que haya un ganador y no quede en empate. Una captura de un resultado para n = 10 es el que sigue en las siguientes páginas, pero antes me gustaría enfatizar que este problema podría haberse resuelto si la función rand() no fuera en función del tiempo sino en función de otro valor, dado que los N procesos actúan en el orden de nano segundos, y el tiempo del generador de aleatorios funciona en el orden de segundos, por lo que generaría mismos aleatorios para procesos hermanos, esto yo lo he solucionado enviando por la barrera los números aleatorios, pero también podría haberse solucionado usando la función srand() que establece en función de que cosa se generan aleatorios, pudiéndose haberse hecho en función del pid de cada proceso hijo.

N = 10 (Un ganador)



Por motivos de tamaño de memoria, en esta memoria se expondrán las salidas para los casos n = x, tal que x es un número comprendido entre 3 (con empate) y 4 (con un proceso ganadora), se ha optado por generar impresiones en pantalla que detallen que es lo que está sucediendo en cada momento para esclarecer el modo de funcionamiento del software final.

CASO N = 3 (empate)

```
sistemas@DyASO:~/Descargas/Plantilla Trabajo DyASO/DyASO_PED2_LUIS_GONZALEZ_MICHAEL_
Para generar la llave se ha introducido ruta de fichero : ./Trabajo2/padre
Se ha introducido como valor de n : 3
====================================
Llave:  1476473958
Cola de mensajes id: 2523136
Memoria compartida id:  7700506
Semaforo id:  2523136
Tuberia descriptor1: 3 ,descriptor2: 4
====================================
El hijo con pid 7913 ha llegado a la barrera y esta esperando a recibir un mensaje
El hijo con pid 7912 ha llegado a la barrera y esta esperando a recibir un mensaje
El hijo con pid 7911 ha llegado a la barrera y esta esperando a recibir un mensaje
=========================
Iniciando ronda de ataques
=========================
El proceso con pid: 7913 esta en modo defensa.
El proceso con pid: 7912 esta en modo ataque.
El proceso con pid: 7911 esta en modo defensa.
El proceso con id: 7912 va a atacar al proceso con id: 7913
El hijo 7913 ha repelido un ataque
==== RESULTADO DE CONTIENDA ====
 Numero procesos supervivientes: 3
 Numero procesos muertos en esta ronda: 0
 GANADOR: Sin determinar
==============================
=========================
Iniciando ronda de ataques
=========================
El proceso con pid: 7913 esta en modo defensa.
El proceso con pid: 7912 esta en modo defensa.
El proceso con pid: 7911 esta en modo defensa.
==== RESULTADO DE CONTIENDA ====
 Numero procesos supervivientes: 3
 Numero procesos muertos en esta ronda: 0
 GANADOR: Sin determinar
==============================
=========================
Iniciando ronda de ataques
=========================
El proceso con pid: 7913 esta en modo ataque.
El proceso con pid: 7912 esta en modo ataque.
El proceso con pid: 7911 esta en modo defensa.
El proceso con id: 7913 va a atacar al proceso con id: 7911
El proceso con id: 7912 va a atacar al proceso con id: 7911
El hijo 7911 ha repelido un ataque
==== RESULTADO DE CONTIENDA ====
 Numero procesos supervivientes: 3
 Numero procesos muertos en esta ronda: 0
 GANADOR: Sin determinar
```

```
==========================
Iniciando ronda de ataques
==========================
El proceso con pid: 7913 esta en modo defensa.
El proceso con pid: 7912 esta en modo ataque.
El proceso con pid: 7911 esta en modo defensa.
El proceso con id: 7912 va a atacar al proceso con id: 7913
El hijo 7913 ha repelido un ataque
==== RESULTADO DE CONTIENDA ====
 Numero procesos supervivientes: 3
 Numero procesos muertos en esta ronda: 0
 GANADOR: Sin determinar
================================
==========================
Iniciando ronda de ataques
==========================
El proceso con pid: 7913 esta en modo ataque.
El proceso con pid: 7912 esta en modo ataque.
El proceso con pid: 7911 esta en modo defensa.
El proceso con id: 7913 va a atacar al proceso con id: 7911
El hijo 7911 ha repelido un ataque
El proceso con id: 7912 va a atacar al proceso con id: 7913
El hijo 7913 ha sido emboscado mientras realizaba un ataque
Asesinato del proceso con pid 7913
==== RESULTADO DE CONTIENDA ====
 Numero procesos supervivientes: 2
 Numero procesos muertos en esta ronda: 1
 GANADOR: Sin determinar
================================
==========================
Iniciando ronda de ataques
==========================
El proceso con pid: 7912 esta en modo ataque.
El proceso con pid: 7911 esta en modo ataque.
El proceso con id: 7912 va a atacar al proceso con id: 7911
El hijo 7911 ha sido emboscado mientras realizaba un ataque
El proceso con id: 7911 va a atacar al proceso con id: 7912
El hijo 7912 ha sido emboscado mientras realizaba un ataque
Asesinato del proceso con pid 7912
Asesinato del proceso con pid 7911
==== RESULTADO DE CONTIENDA ====
 Numero procesos supervivientes: 0
 Numero procesos muertos en esta ronda: 2
 GANADOR: empate
================================

------ Colas de mensajes -----
key        msqid      propietario perms      bytes utilizados mensajes

------ Matrices semáforo -------
key        semid      propietario perms      nsems
```

CASO N = 4 (Un ganador)

```
Para generar la llave se ha introducido ruta de fichero : ./Trabajo2/padre
Se ha introducido como valor de n : 4
====================================
Llave:  1476473958
Cola de mensajes id: 2752512
Memoria compartida id:  7929882
Semaforo id:  2752512
Tuberia descriptor1: 3 ,descriptor2: 4
====================================
El hijo con pid 8085 ha llegado a la barrera y esta esperando a recibir un mensaje
El hijo con pid 8084 ha llegado a la barrera y esta esperando a recibir un mensaje
El hijo con pid 8082 ha llegado a la barrera y esta esperando a recibir un mensaje
El hijo con pid 8083 ha llegado a la barrera y esta esperando a recibir un mensaje
========================
Iniciando ronda de ataques
========================
El proceso con pid: 8085 esta en modo defensa.
El proceso con pid: 8083 esta en modo ataque.
El proceso con pid: 8084 esta en modo defensa.
El proceso con pid: 8082 esta en modo defensa.
El proceso con id: 8083 va a atacar al proceso con id: 8084
El hijo 8084 ha repelido un ataque
==== RESULTADO DE CONTIENDA ====
 Numero procesos supervivientes: 4
 Numero procesos muertos en esta ronda: 0
 GANADOR: Sin determinar
==============================
```

```
========================
Iniciando ronda de ataques
========================
El proceso con pid: 8085 esta en modo defensa.
El proceso con pid: 8083 esta en modo defensa.
El proceso con pid: 8084 esta en modo defensa.
El proceso con pid: 8082 esta en modo ataque.
El proceso con id: 8082 va a atacar al proceso con id: 8083
El hijo 8083 ha repelido un ataque
==== RESULTADO DE CONTIENDA ====
 Numero procesos supervivientes: 4
 Numero procesos muertos en esta ronda: 0
 GANADOR: Sin determinar
==============================
========================
Iniciando ronda de ataques
========================
El proceso con pid: 8085 esta en modo defensa.
El proceso con pid: 8083 esta en modo defensa.
El proceso con pid: 8084 esta en modo ataque.
El proceso con pid: 8082 esta en modo ataque.
El proceso con id: 8084 va a atacar al proceso con id: 8085
El hijo 8085 ha repelido un ataque
El proceso con id: 8082 va a atacar al proceso con id: 8083
El hijo 8083 ha repelido un ataque
==== RESULTADO DE CONTIENDA ====
 Numero procesos supervivientes: 4
 Numero procesos muertos en esta ronda: 0
 GANADOR: Sin determinar
==============================
========================
Iniciando ronda de ataques
========================
El proceso con pid: 8085 esta en modo ataque.
El proceso con pid: 8083 esta en modo defensa.
El proceso con pid: 8084 esta en modo defensa.
El proceso con pid: 8082 esta en modo ataque.
El proceso con id: 8085 va a atacar al proceso con id: 8084
El hijo 8084 ha repelido un ataque
El proceso con id: 8082 va a atacar al proceso con id: 8084
El hijo 8084 ha repelido un ataque
==== RESULTADO DE CONTIENDA ====
 Numero procesos supervivientes: 4
 Numero procesos muertos en esta ronda: 0
 GANADOR: Sin determinar
==============================
```

```
Iniciando ronda de ataques
==========================
El proceso con pid: 8085 esta en modo ataque.
El proceso con pid: 8083 esta en modo defensa.
El proceso con pid: 8084 esta en modo ataque.
El proceso con pid: 8082 esta en modo ataque.
El proceso con id: 8085 va a atacar al proceso con id: 8082
El hijo 8082 ha sido emboscado mientras realizaba un ataque
El proceso con id: 8082 va a atacar al proceso con id: 8083
El hijo 8083 ha repelido un ataque
El proceso con id: 8084 va a atacar al proceso con id: 8082
El hijo 8082 ha sido emboscado mientras realizaba un ataque
Asesinato del proceso con pid 8082
==== RESULTADO DE CONTIENDA ====
 Numero procesos supervivientes: 3
 Numero procesos muertos en esta ronda: 1
 GANADOR: Sin determinar
================================
==========================
Iniciando ronda de ataques
==========================
El proceso con pid: 8085 esta en modo defensa.
El proceso con pid: 8083 esta en modo defensa.
El proceso con pid: 8084 esta en modo defensa.
==== RESULTADO DE CONTIENDA ====
 Numero procesos supervivientes: 3
 Numero procesos muertos en esta ronda: 0
 GANADOR: Sin determinar
================================
==========================
Iniciando ronda de ataques
==========================
El proceso con pid: 8085 esta en modo defensa.
El proceso con pid: 8083 esta en modo defensa.
El proceso con pid: 8084 esta en modo ataque.
El proceso con id: 8084 va a atacar al proceso con id: 8085
El hijo 8085 ha repelido un ataque
==== RESULTADO DE CONTIENDA ====
 Numero procesos supervivientes: 3
 Numero procesos muertos en esta ronda: 0
 GANADOR: Sin determinar
================================
```

```
==========================
Iniciando ronda de ataques
==========================
El proceso con pid: 8085 esta en modo ataque.
El proceso con pid: 8083 esta en modo defensa.
El proceso con pid: 8084 esta en modo ataque.
El proceso con id: 8085 va a atacar al proceso con id: 8084
El hijo 8084 ha sido emboscado mientras realizaba un ataque
El proceso con id: 8084 va a atacar al proceso con id: 8085
El hijo 8085 ha sido emboscado mientras realizaba un ataque
Asesinato del proceso con pid 8085
Asesinato del proceso con pid 8084
==== RESULTADO DE CONTIENDA ====
 Numero procesos supervivientes: 1
 Numero procesos muertos en esta ronda: 2
 GANADOR: 8083
================================

------ Colas de mensajes -----
key        msqid      propietario perms      bytes utilizados mensajes

------ Matrices semáforo -------
key        semid      propietario perms      nsems
```

Bibliografía.

- Pautas en el enunciado de PEC 2 de diseño y administración de sistemas operativos
- Fundamentos del Sistema Operativo Unix por José Manuel Díaz Martínez, Rocío Muñoz Mansilla y Dictino Chaos García.
- https://www.geeksforgeeks.org/named-pipe-fifo-example-c-program/
- https://stackoverflow.com/questions/4113986/example-of-using-named-pipes-in-linux-bash
- https://medium.com/@calmarianet/3-formas-de-ejecutar-comandos-en-segundo-plano-en-linux-9dedb779ca7d
- http://manpages.ubuntu.com/manpages/bionic/es/man2/wait.2.html
- http://manpages.ubuntu.com
-