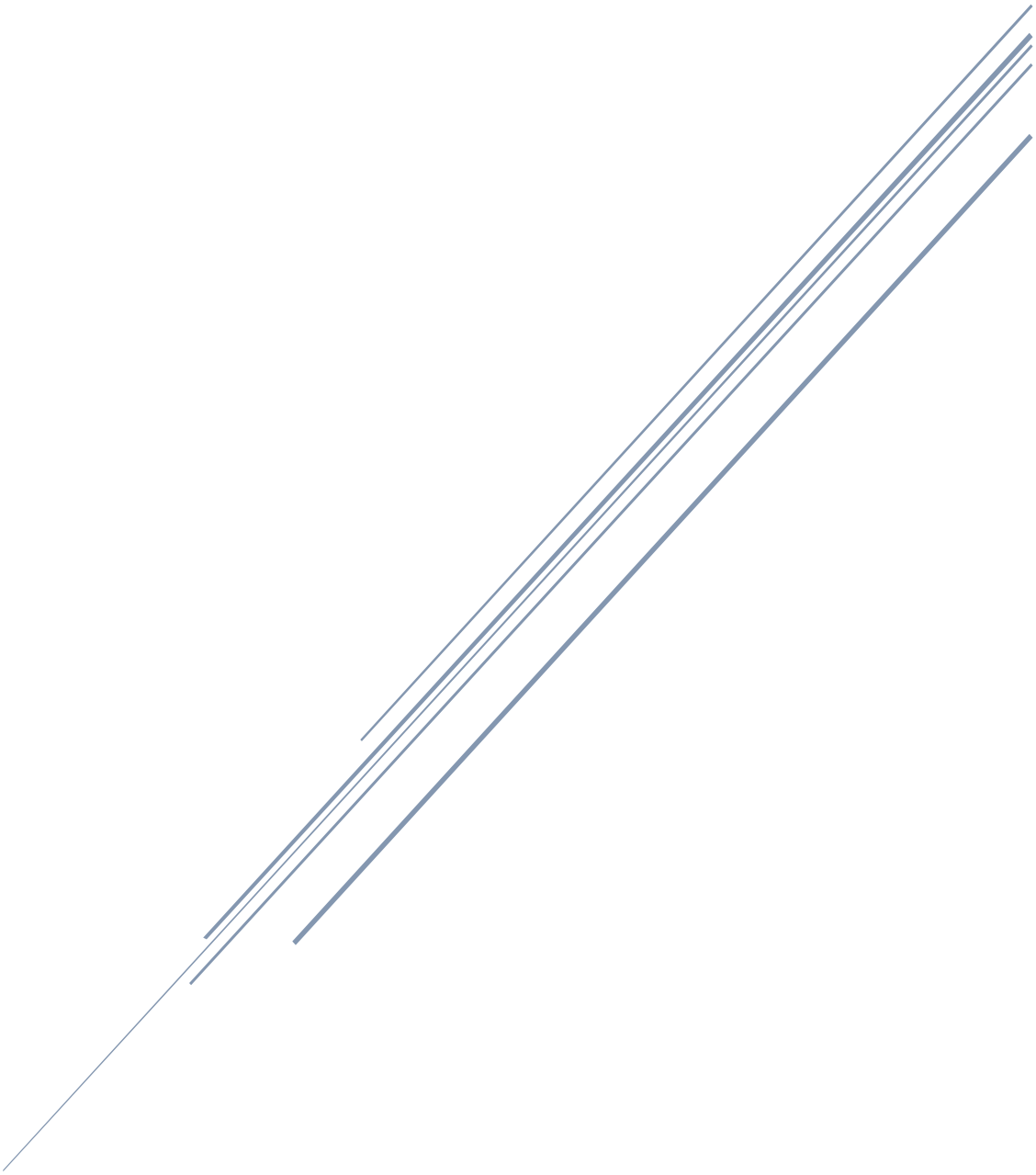
BATRACIOS

Práctica UNIX

GRADO INGENIERIA INFORMÁTICA EN SISTEMAS DE INFORMACIÓN

Sistemas Operativos II

2020/2021



David Barrios Portales
Víctor Vacas Amigo

Contenido

1 – Recursos IPC usados

1.1 - Semáforos

Vamos a usar un conjunto de semáforos, que se va a almacenar su ID en la variable global *id_semaforo*, en concreto estará formado por 10 semáforos.

Cuatro de ellos serán para el control de las ranas madre, otros tres restantes serán para llevar el conteo de las ranas nacidas, salvadas y perdidas, otro semáforo para las posiciones de cada ranita hija y por último un semáforo para controlar que haya un número máximo de 35 procesos, 1 proceso "general", 4 ranas madre y 30 ranitas. Este último número es el que vamos a controlar, 30, que está almacenado en la constante MAX_RANAS_HIJAS.

La declaración de los semáforos es la siguiente en la función main:

```
id_semaforo = semget(IPC_PRIVATE, 10, IPC_CREAT | 0600);
```

En esta línea hemos declarado un conjunto de semáforos, más en concreto 10 semáforos, hemos almacenado el id del conjunto en la variable id_semaforo.

Más adelante establecemos el máximo de "accesos" que permite cada semáforo. Un ejemplo

```
semctl(id_semaforo, RANA_MADRE_1, SETVAL, 1);
semctl(id_semaforo, RANA_MADRE_2, SETVAL, 1);
semctl(id_semaforo, RANA_MADRE_3, SETVAL, 1);
semctl(id_semaforo, RANA_MADRE_4, SETVAL, 1);
```

con los semáforos que controla cada rana madre:

Las definiciones de cada constante son:

```
#define RANA_MADRE_1 2          // De la primera rana madre
#define RANA_MADRE_2 3          // De la segunda rana madre
#define RANA_MADRE_3 4          // De la tercera rana madre
#define RANA_MADRE_4 5          // De la cuarta rana madre
```

Funcionamiento del semáforo de MAX_RANAS_HIJAS:

1.2 Memoria compartida

Vamos a crear los 2048xint necesarios para el funcionamiento correcto de la biblioteca proporcionada. El id de esta memoria se guardará en id_memoria.

También vamos a usar 33 posiciones de una estructura que hemos llamado "posicion_struct" (modificando el nombre respecto al incluido en la biblioteca, ya que el nombre era más confuso).

La declaración de la estructura es la siguiente:

```
struct posicion_struct {int x,y;};
```

Posicionamiento de los datos:

- 0-29: Almacena las posiciones X e Y de cada ranita hija.
- 30-32: Almacena un contador de las ranas nacidas, salvadas y perdidas, respectivamente.

Una vez hecho esto enlazamos la memoria compartida con las variables posiciones y memoria, así:

```
memoria = (char *)shmat( id_memoria, NULL, 0);
posiciones =(struct posicion_struct *) shmat(id_posiciones, NULL, 0);
```

Inicializamos todas las posiciones de memoria a -2, ambas componentes.

Como última tarea antes de comenzar, ponemos a 0 tanto el contador para ranas nacidas, el de ranas salvadas y ranas perdidas de la siguiente forma:

```
//La memoria para las ranitas nacidas
semaforo_wait(id_semaforo,SEMAF_RANITAS_NACIDAS);
//Lo ponemos a 0 al principio
posiciones[30].x=0;
semaforo_signal(id_semaforo,SEMAF_RANITAS_NACIDAS);
```

Vemos si podemos acceder a la memoria compartida de las posiciones, si el semáforo nos los permite entonces ponemos a 0 dicho contador y cuando acabamos damos un signal al semáforo.

Así con el resto de contadores.

1.3 Señales

En la ejecución se enmascara la señal “SIGINT”, para que cuando se mande esta señal usando “CTRL^C”, la ejecución del programa acabe.

2 - Variables globales

Las variables globales que hemos usado son: *id_semaforo*, un puntero a finalizar y por último una estructura para almacenar las posiciones.

- La variable *id_semaforo* es usada para, como su nombre indica, almacena el ID del semáforo usado en la práctica.
- En la variable *finalizar* vamos a almacenar el estado, para en los “bucles infinitos” poder salir
- La variable *id_memoria* contiene el ID de la memoria compartida.

3 – Pseudocódigo

Función RANITA

Función CODIGO_RANA_MADRE

Función MAIN (Simplificada)

4 - Otras funciones relevantes

Tenemos las funciones *presentacio()* y *error_parametros(),* las cuales nos muestran una pequeña cabecera al inicio del programa y nos muestra lo que hacer si no hemos introducido los parámetros correctamente, respectivamente.

Al inicio, la práctica se pausa 6 segundos para que podamos ver por pantalla todos los datos que se nos muestran. Esto se hace creando un hijo y esperando a que finalice.

La función GENERA_ALEATORIO recibe un puntero y un entero, genera un array aleatorio a ese puntero de la dimensión que le hayamos pasado como segundo parámetro.

```
// ------------------------------------------------------------------------
// Funcion genera_aleatorio
// Genera los elementos de un string de forma aleatoria
// ---------------------------------
void genera_aleatorio(int *vector,int num){
    int i;

    if(num>1){
        //Recorremos todo el vector
        for(i=0; i<num;i++){
            //E introducimos los nums aleatorios
            vector[i] = rand() % (RANDOM_MAX + 1 - RANDOM_MIN) + RANDOM_MIN;
        }
        //RANDOM_MAX
        //RANDOM_MIN
        //rand() % (max_number + 1 - minimum_number) + minimum_number
    }
}
// ------------------------------------------------------------------------
```

Se ha intentado reutilizar el máximo código de las prácticas, adaptando ciertas cosas.

Como último aporte, el código se encuentra en github.com, en el siguiente enlace: github.com/Biohazard86/batracios_SO2