GUÍA DE LABORATORIO 01
SISTEMAS OPERATIVOS

Docente: Pablo Calcina Ccori

15 de setiembre de 2020

1 COMPETENCIA DEL CURSO

La comprensión intelectual y la capacidad de aplicar las bases matemáticas y la teoría de la informática.

2 COMPETENCIA DEL LABORATORIO

Identifica y aplica los principios de creación y finalización de procesos, memoria compartida y comunicación entre procesos.

3 CONCEPTOS BÁSICOS

La función `fork()` es usada para crear un nuevo proceso en sistemas tipo Unix. Al crear un proceso con `fork()`, el nuevo proceso creado es el *hijo* del proceso *padre* que invocó `fork()`. El proceso *hijo* será idéntico al proceso padre, inclusive tendrá las mismas variables, registros, descriptores de archivos, etc. Es decir el proceso hijo es una copia *casi igual* al padre. Las únicas diferencias serán algunas informaciones del control, presentes en el bloque de control (PCB) del proceso hijo, por ejemplo el PID o el PPID (parent PID).

La función `kill()` recibe el PID de un proceso y le envía una señal para matarlo. Por ejemplo, la instrucción `kill(4384, SIGKILL);` le envía la señal SIGKILL al proceso 4384 para interrumpir su ejecución.

En el ejemplo proporcionado con la práctica, se usa la variable `ptr` para almacenar contenido del tipo string. Sin embargo, es posible usar estructuras de memoria más complejas, dando el tamaño correcto a la función `mmap`, como en el ejemplo a continuación[1]:

```
#define MAX_LEN 10000
struct region {         /* Defines "structure" of shared memory */
    int len;
    char buf[MAX_LEN];
};
struct region *rptr;
int fd;

/* Create shared memory object and set its size */
fd = shm_open("/myregion", O_CREAT | O_RDWR, S_IRUSR | S_IWUSR);
if (fd == -1)
    /* Handle error */;

if (ftruncate(fd, sizeof(struct region)) == -1)
    /* Handle error */;

/* Map shared memory object */
rptr = mmap(NULL, sizeof(struct region),
       PROT_READ | PROT_WRITE, MAP_SHARED, fd, 0);
if (rptr == MAP_FAILED)
    /* Handle error */;

/* Now we can refer to mapped region using fields of rptr, ex. rptr->len */
```

[1] Tomado de: https://pubs.opengroup.org/onlinepubs/009695399/functions/shm_open.html

4 EQUIPOS Y MATERIALES

- Computadora con Linux instalado.
- Material del curso.

5 EJERCICIOS

A partir del código fuente del problema de productor-consumidor[2], basados en la Figura 3.16 del libro guía [1], implementaremos un ciclo infinito de ejecución para los procesos productor y consumidor, siguiendo la plantilla propuesta en el libro (Figuras 3.12 y 3.13) y reproducidas en esta práctica en las secciones 6.1 y 6.2.

En cada ciclo del bucle infinito, el proceso **productor** deberá:

1. Producir (crear) un nuevo proceso hijo, usando `fork()`.
2. Escribir en una estructura compartida (`struct`) el PID del proceso recién creado y la hora a la que fue creado, incluyendo milisegundos.
3. Actualizar las variables compartidas que `in/out`, según la plantilla.
4. Imprimir (`printf`) su propia información (P de productor y su propio PID) y la información del último proceso creado (ver ejemplo en 6.3).
5. Dormir `sleep()` un número aleatorio de segundos entre 1 y 5.

En cada ciclo del bucle infinito, el proceso **consumidor** deberá:

1. Consumir (leer) la cola (buffer) de procesos creados, leyendo el elemento más antiguo.
2. Actualizar las variables compartidas que `in/out`, según la plantilla.
3. Matar al proceso leído `kill`.
4. Imprimir (`printf`) su propia informacion (C de consumidor y su propio PID) y la información del último proceso creado (ver ejemplo en 6.3).
5. Dormir `sleep()` un número aleatorio de segundos entre 1 y 5.

Observaciones

Para este ejercicio no utilizaremos herramientas de sincronización avanzadas como semáforos y mutex. La solución al problema debe hacerse con variables compartidas simples, como las propuestas en la plantilla: *in*, *out*. Pueden crearse variables adicionales, y su funcionamiento deberá ser explicado en el reporte (README.md).

6 PLANTILLAS Y EJEMPLOS

6.1 Productor

```
item next produced;

while (true) {
    /* produce an item in next produced */

    while (((in + 1) % BUFFER SIZE) == out)
        ; /* do nothing */

    buffer[in] = next produced;
    in = (in + 1) % BUFFER SIZE;
}
```

[2] Disponible en: https://classroom.google.com/c/MTU5NDU0OTY0NzAz/a/MTY5NDU2MTc2ODE1/details

6.2 Consumidor

```
item next consumed;

while (true) {
    while (in == out)
        ; /* do nothing */

    next consumed = buffer[out];
    out = (out + 1) % BUFFER SIZE;
    /* consume the item in next consumed */
}
```

6.3 Ejemplo de ejecución

```
[12:32:03.221] P(4231): Creando Proceso 4232 (12:32:03.221)
[12:32:08.724] C(4233): Matando Proceso 4232 (vivió 05:503)
[12:32:13.221] P(4231): Creando Proceso 4234 (12:32:03.221)
```

7 ENTREGABLES

El principal entregable de esta práctica de laboratorio será el código fuente del programa, empaquetado en un sólo archivo (tar.gz, tar.bz2, zip, etc.) conteniendo los siguientes archivos:

- `producer.c`
- `consumer.c`
- `Makefile` — Script para compilar y ejecutar los dos programas.
- `README.md` — Una explicación sucinta de cómo fue resuelto el problema.

Nombrar el archivo siguiendo la nomenclatura `Lab-SO-Nombre-Apellidos.(tar.gz|zip|tar.bz2)`

8 RÚBRICA DE EVALUACIÓN

Criterios	Muy Bueno	Bueno	Regular	Malo
Programación	Resuelve el ejercicio sin errores mostrando cada uno de los puntos solicitados y siguiendo las restricciones dadas. **Puntaje: 16 puntos**	Resuelve el problema con pocos errores, mostrando casi o todos los puntos solicitados y siguiendo las restricciones dadas. **Puntaje: 14 puntos**	Resuelve el problema con varios errores, mostrando todos o pocos los puntos solicitados y siguiendo las restricciones dadas. **Puntaje: 8 puntos**	No resuelve todos los ejercicios, no entrega el laboratorio o no sigue las restricciones dadas. **Puntaje: 0 puntos**
Explicación de la estrategia	Explica de forma clara y concisa la implementación del programa presentado, incluyendo la manipulación de las variables compartidas. **Puntaje: 4 puntos**	Explica vagamente la implementación del programa desarrollado, incluyendo parcialmente la manipulación de las variables compartidas. **Puntaje: 2 puntos**	La presentación no es entendible y/o no explica la manipulación de las variables compartidas. **Puntaje: 1 punto**	No explica la implementación de su programa o no presenta el trabajo.. **Puntaje: 0 puntos**

- **IMPORTANTE** En caso de copia o plagio o similares todos los alumnos implicados tendrán sanción en toda la evaluación del curso.

BIBLIOGRAFÍA

[1] SILBERSCHATZ, A.; GALVIN, P. B.; GAGNE, G. **Operating System Concepts**. Hoboken, NJ: Wiley, 2018. OCLC: 1192966278. ISBN 9781119439257 9781119456339.