UMAG
Universidad de Magallanes

Departamento de Ingeniería en Computación

Comunicación entre procesos:

Método de Transferencia de mensajes

Práctica No° 3

Estudiante: Milton Hernández - milherna@umag.cl

Fecha y hora: 6 de Noviembre del 2024 - 8:00 AM

Profesor: Eduardo Peña J.

Asignatura: Introducción a Sistemas Operativos

Status del documento

Práctica N°3 **Comunicación entre procesos:** **Método de Transferencia de mensajes**			
Número de referencia del documento			3
Versión	**Revisión**	**Fecha**	**Razones del cambio**
1	1	6 de Noviembre del 2024	Primera revisión del documento

Registro de cambios del documento		**RCD N°:**	1
		Fecha:	6 de Noviembre del 2024
		Otiginado por:	Milton Hernández M.
		Aprobado por:	Milton Hernández M.
Titulo del documento:		Práctica N°3 Comunicación entre procesos: Método de Transferencia de mensajes	
Número de referencia del documento:			003
Versión del documento	1	**Número de revisión:**	1
Página	**Párrafo**	**Razones del cambio**	

Índice

Resumen

Un proceso que se ejecute aislado de los demás en un sistema operativo puede ser útil en diversas ocasiones donde no se precise de información proveniente "del exterior"; sin embargo, en muchas ocasiones es necesario que el proceso tenga acceso a información proveniente de otros procesos o sistemas. Para esto se utiliza la **comunicación entre procesos**, que consiste en la transferencia de mensajes entre procesos. En el presente informe abordaremos una de las opciones de comunicación entre procesos, las colas de comunicación, lo anterior mediante el lenguaje de programación C y sus bibliotecas `sys/ipc.h` y `sys/msg.h`.

1. Introducción

Habiendo comprendido ya la jerarquía entre procesos mediante el uso de funciones como `fork()`, el siguiente paso para el manejo correcto de procesos dentro de un sistema operativo consiste en comprender la comunicación entre dos procesos.

Como se demostró en el anterior experimento de laboratorio los procesos creados mediante la función `fork()` no comparten memoria con el padre que los creó; lo anterior dificulta sobremanera el uso de este método para crear procesos que aprovechen la capacidad de computo del sistema operativo.

Entonces, ¿Existe un método para permitir que dos procesos compartan información entre sí de manera efectiva? Lo cierto es que existen muchas opciones para comunicar procesos entre sí, pero en el presente informe veremos una de ellas: la **Transferencia de mensajes**: "*Este método de comunicación entre procesos utiliza dos primitivas SEND y RECEIVE (...) son llamadas al sistema y no construcciones del lenguaje. Como tales, es fácil colocarlas en procedimientos de biblioteca, como* `send(destino, &mensaje);` *y* `receive(origen, &mensaje);`*. La primera llamada envía un mensaje a un destino dado, y la segunda recibe un mensaje de un origen dado (o de cualquiera [ANY] si al receptor no le importa). Si no hay un mensaje disponible, el receptor podría bloquearse hasta que uno llegue. Como alternativa, podría regresar de inmediato con un código de error.*" (Tanenbaum & Woodhull, 1997)

2. Objetivo Principal

Comprender el funcionamiento de la **Transferencia de mensajes** como método de comunicación entre procesos y su implementación en `C`.

3. Objetivos secundarios

1. Investigar la manera correcta de crear un enlace de comunicación entre dos procesos mediante el uso de colas de mensajes en `C`.
2. Analizar el funcionamiento de la Transferencia de mensajes entre más de dos procesos, generando grupos de productores de mensajes y consumidores de mensajes.

4. Marco teórico

Para comprender los procedimientos realizados en el presente laboratorio es necesario comprender conceptos teóricos y prácticos previos.

4.1. Cola de mensajes

Una cola de mensajes es una estructura de datos que almacena un conjunto de mensajes. Cada cola presente en un sistema operativo (Linux para el presente informe) tiene un identificador único, que se utiliza para acceder a la cola, así como diferentes permisos para el usuario que la creó, el grupo al que pertenece y demás usuarios.

En Linux es posible ver las colas que se encuentran activas con el comando `ipcs` y eliminar una cola con el comando `ipcrm -q <id>`.

Listing 1: colas de mensajes

```
------ Message Queues --------
key        msqid      owner      perms      used-bytes   messages
0x00003039 30         milton     666        0            0

------ Shared Memory Segments --------
key        shmid      owner      perms      bytes      nattch     status

------ Semaphore Arrays --------
key        semid      owner      perms      nsems
```

4.2. Uso de las bibliotecas `sys/ipc.h` y `sys/msg.h`

Las bibliotecas `sys/ipc.h` y `sys/msg.h` contienen las funciones necesarias para crear y manipular colas de mensajes, enviar y recibir mensajes.

Todo parte con una `key`, una especie de código que representa a una cola dentro de un sistema operativo (diferente al `ID` que explicaremos después). Como usuarios podemos definir una key como un numero cualquiera, por ejemplo 12345, pero es recomendable que la key sea un número único para cada cola de mensajes, lo que puede lograrse mediante la función `ftok()` la cuál "*Genera una clave de comunicación de interproceso estándar*"(«Subrutina ftok», 2023) en base a un archivo y un identificador, con lo que podríamos escribir algo como: `key_t key = ftok("main.c", 1);` para obtener una llave válida.

Además de la llave se necesita un espacio donde guardar el mensaje, para lo que se suele usar una estructura `struct message` que contiene un tipo y un cuerpo del mensaje.

```
#define MAX_MSG_SIZE 100
```

```
typedef struct message {
    long type;
    char body[MAX_MSG_SIZE];
} Message;
```

El tipo del mensaje es un `long` que puede ser cualquier número; mediante este tipo de mensaje podríamos decidir que un receptor de mensajes solamente reciba mensajes del tipo 1, por ejemplo.

Por otro lado en el ejemplo el cuerpo del mensaje es un arreglo de caracteres, sin embargo la estructura del mensaje podría contener cualquier tipo de información, por ejemplo un `int` o un `char` o un puntero a una estructura. Sin embargo, es importante recordar que la cola de mensajes no almacena punteros a memoria del proceso (por ejemplo, un puntero a una estructura dinámica), ya que los datos deben ser autocontenidos y no depender de direcciones de memoria específicas del proceso emisor.

Teniendo esta llave, podemos crear una cola de mensajes con la función `msgget()` que devuelve un identificador de cola de mensajes, si la cola no existe se creará una nueva, si ya existe se devolverá el identificador de la cola existente. una manera de crear esta cola sería: `int msg_id = msgget(key, 0666 | IPC_CREAT);`[1], este proceso se hará en todos los procesos usando siempre la mima llave para que cada proceso se conecte a la misma cola.

Una vez teniendo la conexión establecida con la cola un proceso puede enviar un mensaje a otro proceso mediante la función `msgsnd()` que recibe como parámetros:

- El identificador de la cola de mensajes.
- Un puntero a un `struct message` que contiene el mensaje a enviar.
- El tamaño del mensaje (solamente el cuerpo, sin contar el tamaño del tipo).
- Una bandera que puede variar el comportamiento del envío del mensaje.

El mensaje se envía a través de la cola de mensajes, y el receptor puede recibirlo mediante la función `msgrcv()` que recibe como parámetros:

- El identificador de la cola de mensajes.
- Un puntero a un `struct message` que contiene el mensaje recibido.
- El tamaño del mensaje (solamente el cuerpo, sin contar el tamaño del tipo).
- EL tipo de proceso que se recibirá (0 para cualquier tipo).
- Una bandera que puede variar el comportamiento del receptor del mensaje.

[1] El números `0666` corresponde a los permisos de la cola (ver referencia: «Permisos del sistema de archivos en GNU/Linux»), en este caso se permite a cualquier usuario leer y escribir en la cola, mientras que `IPC_CREAT` indica que la cola debe ser creada si no existe.

Finalmente, luego de haber realizado las operaciones necesarias es menester eliminar la cola de mensajes, puesto que esta es independiente de los procesos que la usan, por lo que de no eliminarla quedará presente aún habiendo terminado los procesos; para su eliminación usamos la función `msgctl()` que recibe como parámetros:

- El identificador de la cola de mensajes.
- Un identificador con la operación que se desea realizar sobre la cola (`IPC_RMID` para eliminar la cola).
- Un puntero a una estructura `msqid_ds`. Esta estructura almacena información sobre la cola de mensajes, como permisos, tiempo de última modificación, número de mensajes en la cola, y límites de tamaño de los mensajes, entre otros.

5. Procedimiento experimental

Una vez comprendidos los conceptos teóricos y prácticos necesarios para trabajar con la Transferencia de mensajes, se pueden empezar a realizar algunos experimentos para una mejor comprensión del comportamiento del sistema operativo frente a este método de comunicación.

El Procedimiento experimental que se realizará a continuación consiste en:

1. Inicialmente crear un proceso que se encargue de generar mensajes de manera constante (con tiempos de espera *aleatorios* entre mensajes) y enviarlos a una cola de mensajes.
2. Construir otro proceso que se encargue de recibir mensajes de la cola de mensajes y imprimirlos en pantalla.
3. Realizar esto con una cantidad de mensajes considerable y analizar los resultados.

Además se buscará dar respuesta a las siguientes preguntas:

- ¿Qué ocurre cuando se envían mensajes a una cola de mensajes que ya está creada?
- ¿Cómo es el comportamiento de una cola cuando se tienen varios procesos que la usan? (productores y consumidores en diferentes cantidades).
- ¿Cómo enviar a más de un proceso consumidor el mismo mensaje?
- ¿De qué manera se podría recibir en un solo Proceso Consumidor los mensajes de más de un Proceso Productor?

6. Datos obtenidos

Lo primero que se llevó a cabo fue la creación de un programa que genere un proceso productor (que puede encnotrar en el anexo4) y un consumidor (que puede encnotrar en el anexo5) que se encargaran de enviar mensajes a una cola de mensajes, y recibirlos.

Para ello se usó inicialmente la `key` 12345 para la cola de mensajes. Ya con la cola iniciada el proceso productor entraba en un bucle infinito que genera mensajes con un retardo dado por la función `delay()`, el mensaje es creado de manera secuencial y va acompañado del **identificador del proceso** que lo envió, para ello se utiliza la función `getpid()` vista en informes anteriores. El proceso generará inicialmente 14 mensajes, y el 15-ésimo será el mensaje "Terminado", que finalizará el programa (evidentemente se puede cambiar el numero de mensajes que se envían, pero para el análisis se optó por este valor).

El proceso consumidor similarmente funciona mediante un bucle infinito que recibe mensajes y los imprime por pantalla; cuando lee el mensaje "Terminado" lo imprime y finaliza el programa **Borrando la cola de mensajes** mediante la función `msgctl()` con la operación `IPC_RMID`.

Se realizaron entonces 4 experimentos diferentes para analizar el comportamiento del sistema operativo frente a la Transferencia de mensajes:

6.1. Un productor y un consumidor

Inicialmente se probó generar 15 mensajes entre un productor y un consumidor.

Listing 2: Un consumidor y un productor

```
$ ./send.out& ./recieve.out&
34619 enviado: Mensaje 34619-1
34620 recibido: Mensaje 34619-1
34620 recibido: Mensaje 34619-2
34619 enviado: Mensaje 34619-2
34619 enviado: Mensaje 34619-3
34619 enviado: Mensaje 34619-4
34619 enviado: Mensaje 34619-5
34619 enviado: Mensaje 34619-6
34619 enviado: Mensaje 34619-7
34619 enviado: Mensaje 34619-8
34619 enviado: Mensaje 34619-9
34619 enviado: Mensaje 34619-10
34619 enviado: Mensaje 34619-11
34619 enviado: Mensaje 34619-12
34619 enviado: Mensaje 34619-13
34619 enviado: Mensaje 34619-14
```

```
34620 recibido: Mensaje 34619-3
34619 enviado: Terminado
34620 recibido: Mensaje 34619-4
34620 recibido: Mensaje 34619-5
34620 recibido: Mensaje 34619-6
34620 recibido: Mensaje 34619-7
34620 recibido: Mensaje 34619-8
34620 recibido: Mensaje 34619-9
34620 recibido: Mensaje 34619-10
34620 recibido: Mensaje 34619-11
34620 recibido: Mensaje 34619-12
34620 recibido: Mensaje 34619-13
34620 recibido: Mensaje 34619-14
34620 recibido: Terminado, se termina el programa

[1]-  Done                    ./send.out
[2]+  Done                    ./recieve.out
```

Al hacer este experimento con el `delay()` activado se veía un patrón de envío y recibido perfecto, sin embargo al eliminar el `delay()` se pudo observar el comportamiento retratado en el anexo 2 donde se aprecia que en ocasiones Se recibe el mensaje antes de enviarlo o el consumidor termina su ejecución mucho después de que el productor.

El por qué de estos comportamientos será analizado en profundidad más adelante.

6.2. Tres productores y un consumidor

Para este experimento se probó con tres productores (cada uno de ellos generando 15 mensajes) y un consumidor, esto manteniendo el `delay()` activado. Los resultados del experimento se consignan en el anexo 6.

Lo que se pudo observar en este experimento fue que el consumidor pudo recibir los mensajes de sus trs productores, deteniéndose cuando uno de ellos envió el mensaje `Terminado`, momento en el cuál aparecen dos errores `Error al enviar el mensaje: Invalid argument` provocados porque los otros productores no encuentran la cola de mensajes, momento en el cuál termina su ejecución con un error que podemos ver al final de la ejecución del experimento.

6.3. Tres consumidores para un productor único

En este caso se mantuvo el `delay()` pero de eliminó la salida por pantalla del productor, obteniendo por resultado la salida mostrada en el anexo 7.

Podemos notar de este resultado un par de asuntos llamativos:

- Aparece un error en dos de los consumidores luego de que el primero reciba un mensaje de `terminado` el cuál

indica que la cola de mensajes ha sido eliminada, por lo que no se puede leer ningún mensaje.

- Las impresiones por pantalla sugieren que los procesos consumidores 'se repartieron en orden' los mensajes de la cola de mensajes ya que se repite el patrón 9550, 9551, 9552 en los PIDs de los consumidores.

6.4. Diez productores y un consumidor

Para este experimento se probó con diez productores (cada uno de ellos generando 15 mensajes) y un consumidor, esto manteniendo el `delay()` activado. Los resultados del experimento se consignan en el anexo 8.

Podemos notar los mismos comportamientos analizados en los casoa anteriores: tenemos, como era de esperar 9 menasjes de error por parte de los productores que no encuentran acceso a la cola de mensajes.

Cabe destacar que la cola de mensajes fué procesada sin mayor inconveniente por el proceso consumidor.

7. Análisis y discusión de los resultados

Una vez realizados los experimentos podemos realizar el siguiente análisis y discusión de los resultados obtenidos:

7.1. El paralelismo entre procesos

Se puede apreciar que los procesos productores y consumidores son atendidos por el sistema operativo "al mismo tiempo" lo cuál provoca que sea posible que un proceso productor envíe un mensaje a la cola de mensajes y antes de que este pueda ser impreso por el productor el consumidor lo reciba y procese. Esto genera la ilusión de que 'El mensaje fue recibido antes de ser enviado', pero es causa unicamente del comportamiento del sistema operativo que hemos analizado en informes anteriores.

7.2. Límites de la cola de mensajes

Respecto a los límites de la cola y el comportamiento de los programas se encontró que, de acuerdo con Linux (2024a) existen dos archivos donde se almacena la información del límite de mensajes de una cola y el límite de tamaño de la misma, estos son: '/proc/sys/fs/mqueue/msg_max' y '/proc/sys/fs/mqueue/msgsize_max'. (En mi sistema operativo las colas tienen un límite de 10 mensajes y un tamaño de 8192 bytes).

Entonces ¿Cómo se manejan los mensajes cuando estos límites se alcanzan?

Pues esta es una de las facilidades que nos entrega la función `msgsnd()` es la de bloquear el proceso hasta que la cola tenga un espacio disponible; este es el comportamiento por defecto, pero también se puede cambiar mediante la bandera `IPC_NOWAIT` que indica que no se debe bloquear el proceso, en su lugar la función devolverá -1 indicando que el mensaje no pudo ser enviado; además de esto alterará el valor de `errno` a `EAGAIN` indicando que el el recurso está temportalmente no disponible: "*Resource temporarily unavailable*".

De acuerdo con IBM (2023b): En caso de no activar la bandera `IPC_NOWAIT` la función `msgsnd()` bloqueará el proceso hasta que se cumpla una de las siguientes condiciones:

- La condición responsable de la suspensión ya no existe, en cuyo caso se envía el mensaje.
- El parámetro `MessageQueueID` se elimina del sistema. Cuando esto ocurre, la variable global de `errno` se establece igual al código de error de `EIDRM` y se devuelve un valor de -1.
- El proceso de llamada recibe una señal que se va a atrapar. En este caso, el mensaje no se envía y el proceso de llamada reanuda la ejecución en la forma prescrita en la subrutina sigaction.

Sin embargo `msgrcv()` también tiene ciertas consideraciones a la hora de tratar con las colas, en particular cuando la cola está vacía. Por defecto si se con `msgrcv()` a una cola vacía el proceso se bloqueará hasta que se reviba un mensaje,

pero también se puede cambiar mediante la bandera `IPC_NOWAIT` que indica que no se debe bloquear el proceso, en su lugar la función devolverá -1 indicando que el mensaje no pudo ser recibido; además de esto alterará el valor de `errno` a `ENOMSG` indicando que no hay mensajes (o no del tipo deseado) en la cola: "*Error NO MeSaGe*".

De acuerdo con IBM (2023a): En caso de no activar la bandera `IPC_NOWAIT` la función `msgrcv()` bloqueará el proceso hasta que se cumpla una de las siguientes condiciones:

- Se coloca un mensaje del tipo deseado en la cola.
- El identificador de cola de mensajes especificado por el parámetro `MessageQueueID` se elimina del sistema. Cuando esto ocurre, la `errno` se establece en el código de error `EIDRM` y se devuelve un valor de -1.
- El proceso de llamada recibe una señal que se va a capturar. En este caso, no se recibe un mensaje y el proceso de llamada se reanuda de la forma descrita en la "subrutina sigaction".

La variable `errno`

`erno` es una variable global presente en la librería estándar de `C` que contiene el código del último error que ocurrió en la ejecución de un programa. Esta variable está definida dentro de la cabecera `errno.h` que define esta como un valor entero que diferentes funciones pueden alterar.

Es posible conocer todos los posibles valores de `errno` mediante el comando `errno -l` que se encuentra dentro del paquete `moreutils` de `GNU/Linux`.

```
$errno -l
EPERM 1 Operation not permitted
ENOENT 2 No such file or directory
ESRCH 3 No such process
EINTR 4 Interrupted system call
EIO 5 Input/output error
ENXIO 6 No such device or address
E2BIG 7 Argument list too long
ENOEXEC 8 Exec format error
EBADF 9 Bad file descriptor
ECHILD 10 No child processes
EAGAIN 11 Resource temporarily unavailable
ENOMEM 12 Cannot allocate memory
EACCES 13 Permission denied
...
```

Incluso es posible consultar el valor de `errno` mediante la función `strerror()` que recibe como parámetro el código de error y devuelve una cadena con el mensaje correspondiente, así podríamos usar esta función para obtener un mensaje de error de forma más clara.

Listing 3: Ejemplo de uso de errno y strerror

```c
#include <stdio.h>
#include <errno.h>
#include <string.h>

int main() {
    FILE *file;

    // Intentar abrir un archivo que no existe
    file = fopen("archivo_inexistente.txt", "r");

    // Comprobar si fopen fallo
    if (file == NULL) {
        // Imprimir un mensaje de error especifico
        printf("Error al abrir el archivo: %s\n", strerror(errno));
    } else {
        // Si fopen fue exitoso, realizar operaciones con el archivo
        printf("Archivo abierto correctamente.\n");
        // Cerrar el archivo al finalizar
        fclose(file);
    }

    return 0;
}
```

¿Cómo afectan estos limites a los experimentos realizados?

En los experimentos realizados no se nota el efecto de los límites de lad colas de mensajes puesto que no se usó la bandera `IPC_NOWAIT`, por lo que los procesos se bloqueaban y desbloqueaban según correspondía.

7.3. Resolviendo preguntas planteadas

Como mencionado en la sección 5 es menester responder a algunas preguntas relacionadas con la Transferencia de mensajes:

- **¿Qué ocurre cuando se envían mensajes a una cola de mensajes que ya está creada?**: Ahora sabemos que lo que pasará será que simplemente será utilizada, y en caso de que no se encuentre la cola será creada al usar la función `msgget()`. Sin embaego si se intenta enviar o recibir información a través de una cola que no está creada, se producirá un error que será detectado por las funciones `msgsnd()` y `msgrcv()`.
- **¿Cómo es el comportamiento de una cola cuando se tienen varios procesos que la usan? (productores y**

consumidores en diferentes cantidades): La cola almacenará los mensajes en el orden en que lleguen y cuando la cola alcance su tamaño máximo los procesos que intenten enviar mensajes a ella se bloquearán o enviarán un error según corresponda. Similarmente si hay procesos en busca de leer una cola que se encuentra vacía también se bloquearán o se enviará un error.

- **¿Cómo enviar a más de un proceso consumidor el mismo mensaje?**: A pesar de que no se hizo un experimento al respecto es sencillo intuir que se pueden crear dos colas (con keys diferentes) y enviar un determinado mensaje a una y posteriormente a otra, luego cada proceso productor leerá una determinada cola de procesos y leerá el mensaje que le corresponda. Otra alternativa podría ser usar una única cola con la presencia de mensajes de dos tipos diferentes, luego desde un proceso consumidor leer solo los mensajes de tipo *A* y desde el otro los mensajes de tipo *B*.
- **¿De qué manera se podría recibir en un solo Proceso Consumidor los mensajes de más de un Proceso Productor?**: Este ejemplo sí se realizó en uno de los experimentos; lo que se debe hacer es que los procesos productores envíen los mensajes a una cola que será leída por el proceso consumidor.

Luego de haber respondido estas preguntas podemos darnos cuenta de la versatilidad que nos permiten las colas de mensajes: Estas son independientes de los procesos que las usan, por lo que podemos tener una cola donde llegan mensajes de *m* productores que son leídos por *n* consumidores y nuestro programa podrá funcionar de manera adecuada. Además podemos enviar datos de diversos tipos (a pesar de que los ejemplos fueron solo con cadenas de texto), lo cuál puede ser realmente útil en situaciones reales.

7.4. Desventajas de la Transferencia de mensajes

El método de **Transferencia de mensajes** analizado en el presente informe es muy versátil, pero tiene algunas desventajas que deben ser consideradas a la hora de trabajar con él:

- **Limitaciones de espaico y cantidad**: Como se analizó anteriormentem las colas de mensajes tienen un límite de mensajes y tamaño de los mensajes, que, a pesar de que puede ser controlado mediante bloqueos de los procesos pueden evitar una comunicación rápida y eficiente.
- **Eficiencia**: El método de Transferencia de mensajes no es eficiente en comparación con otros métodos de comunicación, puesto que para enviar un mensaje a una cola es necesario copiar su información en la misma lo que puede ser muy costoso en comparación con otros métodos de comunicación.

Para superar este tipo de dificultades se puede utilizar otros métodos de comunicación, como por ejemplo los *Semáforos* o los *Monitores* los cuales permiten manejar información mediante **memoria compartida** y **sincronización** de la misma, pero que serán posiblemente analizados en futuras experiencias de laboratorio.

8. Conclusiones y recomendaciones

Finalmente a raíz de lo realizado en este experimento se puede concluir lo siguiente:

1. Existen diferentes métodos de comunicación entre procesos, cada uno de los cuales trae sus propias ventajas y desventajas; entre ellos se encuentra el método de **Transferencia de mensajes**, que permite a varios procesos leer y escribir en una cola de mensajes facilitando el procesamiento de información por parte de varios productores y consumidores.

2. El lenguaje de programación `C` trae incorporadas diferentes librerías y funciones que facilitan el manejo de colas de mensajes, permitiendo a los programadores realizar operaciones de comunicación entre procesos de manera sencilla mediante el uso de llamadas al sistema como `SEND` y `RECEIVE` a través de las funciones `msgsnd()` y `msgrcv()`.

3. Como dato adicional encontrado gracias a este laboratorio se destaca la existencia de la variable `errno` que contiene el código del último error que ocurrió en la ejecución de un programa, lo que permite el fácil reconocimiento de los errores ocurridos mediante las diferentes llamadas al sistema.

9. Anexos

Listing 4: send.c

```c
#include <stdio.h>
#include <sys/ipc.h>
#include <sys/msg.h>
#include <unistd.h>
#include <string.h>
#include <time.h>

#include "delay.h"

#define MAX_MSG_SIZE 100
#define MSG_KEY 12345

typedef struct message {
    long type;
    char body[MAX_MSG_SIZE];
} Message;

int main() {
    srand(getpid());
    Message m;
    int msg_id, msg_count = 1;

    // Crear la cola de mensajes
    msg_id = msgget(MSG_KEY, 0666 | IPC_CREAT);
    if (msg_id == -1) {
        perror("Error al crear la cola de mensajes");
        return 1;
    }

    while (1) {
        delay(1,3);
        m.type = 1;  // Tipo de mensaje
        sprintf(m.body, "Mensaje %d-%d" , getpid(), msg_count);

        if(msg_count == 15) {
            sprintf(m.body, "Terminado");
        }

        // Enviar el mensaje
        if (msgsnd(msg_id, &m, sizeof(m.body), 0) == -1) {
            perror("Error al enviar el mensaje");
            return 1;
```

```
        }

        printf("%d enviado: %s\n",getpid(), m.body);

        if(strcmp(m.body, "Terminado") == 0) {
            break;
        }

        msg_count++;
    }

    return 0;
}
```

Listing 5: recieve.c

```
#include <stdio.h>
#include <sys/ipc.h>
#include <sys/msg.h>
#include <unistd.h>
#include <string.h>

#define MAX_MSG_SIZE 100
#define MSG_KEY 12345

typedef struct message {
    long type;
    char body[MAX_MSG_SIZE];
} Message;

int main() {
    Message m;
    int msg_id;

    // Obtener el ID de la cola de mensajes
    msg_id = msgget(MSG_KEY, 0666 | IPC_CREAT);
    if (msg_id == -1) {
        perror("Error al obtener la cola de mensajes");
        return 1;
    }

    while (1) {
        // Recibir el mensaje
        if (msgrcv(msg_id, &m, sizeof(m.body), 0, 0) == -1) {
            perror("Error al recibir el mensaje");
            return 1;
```

```
        }

        if(strcmp(m.body, "Terminado") == 0) {
            break;
        }
        printf("%d recibido: %s\n",getpid(), m.body);
    }

    printf("%d recibido: %s, se termina el programa\n", getpid(), m.body);
    msgctl(msg_id, IPC_RMID, NULL);

    return 0;
}
```

Listing 6: Tres productores y un consumidor

```
$ ./send.out& ./send.out& ./send.out& ./recieve.out&
[1] 36254
[2] 36255
[3] 36256
[4] 36257
36257 recibido: Mensaje 36255-1
36255 enviado: Mensaje 36255-1
36254 enviado: Mensaje 36254-1
36257 recibido: Mensaje 36254-1
36256 enviado: Mensaje 36256-1
36257 recibido: Mensaje 36256-1
36255 enviado: Mensaje 36255-2
36257 recibido: Mensaje 36255-2
36254 enviado: Mensaje 36254-2
36257 recibido: Mensaje 36254-2
36256 enviado: Mensaje 36256-2
36257 recibido: Mensaje 36256-2
36255 enviado: Mensaje 36255-3
36257 recibido: Mensaje 36255-3
36254 enviado: Mensaje 36254-3
36257 recibido: Mensaje 36254-3
36256 enviado: Mensaje 36256-3
36257 recibido: Mensaje 36256-3
36254 enviado: Mensaje 36254-4
36257 recibido: Mensaje 36254-4
36256 enviado: Mensaje 36256-4
36257 recibido: Mensaje 36256-4
36255 enviado: Mensaje 36255-4
36257 recibido: Mensaje 36255-4
36254 enviado: Mensaje 36254-5
```

```
36257 recibido: Mensaje 36254-5
36254 enviado: Mensaje 36254-6
36257 recibido: Mensaje 36254-6
36255 enviado: Mensaje 36255-5
36257 recibido: Mensaje 36255-5
36256 enviado: Mensaje 36256-5
36257 recibido: Mensaje 36256-5
36254 enviado: Mensaje 36254-7
36257 recibido: Mensaje 36254-7
36254 enviado: Mensaje 36254-8
36257 recibido: Mensaje 36254-8
36255 enviado: Mensaje 36255-6
36257 recibido: Mensaje 36255-6
36255 enviado: Mensaje 36255-7
36257 recibido: Mensaje 36255-7
36256 enviado: Mensaje 36256-6
36257 recibido: Mensaje 36256-6
36254 enviado: Mensaje 36254-9
36257 recibido: Mensaje 36254-9
36256 enviado: Mensaje 36256-7
36257 recibido: Mensaje 36256-7
36257 recibido: Mensaje 36255-8
36255 enviado: Mensaje 36255-8
36255 enviado: Mensaje 36255-9
36257 recibido: Mensaje 36255-9
36257 recibido: Mensaje 36254-10
36254 enviado: Mensaje 36254-10
36255 enviado: Mensaje 36255-10
36257 recibido: Mensaje 36255-10
36254 enviado: Mensaje 36254-11
36257 recibido: Mensaje 36254-11
36257 recibido: Mensaje 36256-8
36256 enviado: Mensaje 36256-8
36254 enviado: Mensaje 36254-12
36257 recibido: Mensaje 36254-12
36254 enviado: Mensaje 36254-13
36257 recibido: Mensaje 36254-13
36256 enviado: Mensaje 36256-9
36257 recibido: Mensaje 36256-9
36257 recibido: Mensaje 36255-11
36255 enviado: Mensaje 36255-11
36254 enviado: Mensaje 36254-14
36257 recibido: Mensaje 36254-14
36256 enviado: Mensaje 36256-10
36257 recibido: Mensaje 36256-10
36256 enviado: Mensaje 36256-11
```

```
36257 recibido: Mensaje 36256-11
36255 enviado: Mensaje 36255-12
36257 recibido: Mensaje 36255-12
36254 enviado: Terminado
36257 recibido: Terminado, se termina el programa
Error al enviar el mensaje: Invalid argument
Error al enviar el mensaje: Invalid argument

[1]   Done                    ./send.out
[2]   Exit 1                  ./send.out
[3]-  Exit 1                  ./send.out
[4]+  Done                    ./recieve.out
```

Listing 7: Tres consumidores para un productor único

```
9550 recibido: Mensaje 9549-1
9551 recibido: Mensaje 9549-2
9552 recibido: Mensaje 9549-3
9550 recibido: Mensaje 9549-4
9551 recibido: Mensaje 9549-5
9552 recibido: Mensaje 9549-6
9550 recibido: Mensaje 9549-7
9551 recibido: Mensaje 9549-8
9552 recibido: Mensaje 9549-9
9550 recibido: Mensaje 9549-10
9551 recibido: Mensaje 9549-11
9552 recibido: Mensaje 9549-12
9550 recibido: Mensaje 9549-13
9551 recibido: Mensaje 9549-14
9552 recibido: Terminado, se termina el programa
Error al recibir el mensaje: Identifier removed
Error al recibir el mensaje: Identifier removed

[2]   Done                    ./send.out
[3]   Exit 1                  ./recieve.out
[4]   Exit 1                  ./recieve.out
[5]-  Done                    ./recieve.out
```

Listing 8: diez productores y un consumidor

```
10317 recibido: Mensaje 10314-1
10317 recibido: Mensaje 10312-1
10317 recibido: Mensaje 10316-1
10317 recibido: Mensaje 10311-1
10317 recibido: Mensaje 10309-1
10317 recibido: Mensaje 10314-2
10317 recibido: Mensaje 10313-1
```

```
10317 recibido: Mensaje 10307-1
10317 recibido: Mensaje 10312-2
10317 recibido: Mensaje 10308-1
10317 recibido: Mensaje 10315-1
10317 recibido: Mensaje 10313-2
10317 recibido: Mensaje 10309-2
10317 recibido: Mensaje 10312-3
10317 recibido: Mensaje 10310-1
10317 recibido: Mensaje 10311-2
10317 recibido: Mensaje 10316-2
10317 recibido: Mensaje 10314-3
10317 recibido: Mensaje 10312-4
10317 recibido: Mensaje 10309-3
10317 recibido: Mensaje 10308-2
10317 recibido: Mensaje 10312-5
10317 recibido: Mensaje 10310-2
10317 recibido: Mensaje 10307-2
10317 recibido: Mensaje 10315-2
10317 recibido: Mensaje 10311-3
10317 recibido: Mensaje 10313-3
10317 recibido: Mensaje 10313-4
10317 recibido: Mensaje 10316-3
10317 recibido: Mensaje 10310-3
10317 recibido: Mensaje 10312-6
10317 recibido: Mensaje 10308-3
10317 recibido: Mensaje 10314-4
10317 recibido: Mensaje 10309-4
10317 recibido: Mensaje 10307-3
10317 recibido: Mensaje 10315-3
10317 recibido: Mensaje 10311-4
10317 recibido: Mensaje 10307-4
10317 recibido: Mensaje 10312-7
10317 recibido: Mensaje 10310-4
10317 recibido: Mensaje 10313-5
10317 recibido: Mensaje 10310-5
10317 recibido: Mensaje 10311-5
10317 recibido: Mensaje 10315-4
10317 recibido: Mensaje 10308-4
10317 recibido: Mensaje 10314-5
10317 recibido: Mensaje 10312-8
10317 recibido: Mensaje 10309-5
10317 recibido: Mensaje 10316-4
10317 recibido: Mensaje 10307-5
10317 recibido: Mensaje 10316-5
10317 recibido: Mensaje 10309-6
10317 recibido: Mensaje 10313-6
```

```
10317 recibido: Mensaje 10311-6
10317 recibido: Mensaje 10316-6
10317 recibido: Mensaje 10307-6
10317 recibido: Mensaje 10310-6
10317 recibido: Mensaje 10312-9
10317 recibido: Mensaje 10310-7
10317 recibido: Mensaje 10308-5
10317 recibido: Mensaje 10313-7
10317 recibido: Mensaje 10315-5
10317 recibido: Mensaje 10315-6
10317 recibido: Mensaje 10310-8
10317 recibido: Mensaje 10314-6
10317 recibido: Mensaje 10311-7
10317 recibido: Mensaje 10307-7
10317 recibido: Mensaje 10309-7
10317 recibido: Mensaje 10316-7
10317 recibido: Mensaje 10312-10
10317 recibido: Mensaje 10313-8
10317 recibido: Mensaje 10311-8
10317 recibido: Mensaje 10308-6
10317 recibido: Mensaje 10311-9
10317 recibido: Mensaje 10314-7
10317 recibido: Mensaje 10310-9
10317 recibido: Mensaje 10307-8
10317 recibido: Mensaje 10308-7
10317 recibido: Mensaje 10316-8
10317 recibido: Mensaje 10310-10
10317 recibido: Mensaje 10309-8
10317 recibido: Mensaje 10312-11
10317 recibido: Mensaje 10314-8
10317 recibido: Mensaje 10316-9
10317 recibido: Mensaje 10313-9
10317 recibido: Mensaje 10308-8
10317 recibido: Mensaje 10315-7
10317 recibido: Mensaje 10309-9
10317 recibido: Mensaje 10310-11
10317 recibido: Mensaje 10311-10
10317 recibido: Mensaje 10309-10
10317 recibido: Mensaje 10307-9
10317 recibido: Mensaje 10314-9
10317 recibido: Mensaje 10315-8
10317 recibido: Mensaje 10308-9
10317 recibido: Mensaje 10315-9
10317 recibido: Mensaje 10311-11
10317 recibido: Mensaje 10316-10
10317 recibido: Mensaje 10313-10
```

```
10317 recibido: Mensaje 10312-12
10317 recibido: Mensaje 10310-12
10317 recibido: Mensaje 10312-13
10317 recibido: Mensaje 10314-10
10317 recibido: Mensaje 10307-10
10317 recibido: Mensaje 10313-11
10317 recibido: Mensaje 10308-10
10317 recibido: Mensaje 10309-11
10317 recibido: Mensaje 10316-11
10317 recibido: Mensaje 10315-10
10317 recibido: Mensaje 10316-12
10317 recibido: Mensaje 10311-12
10317 recibido: Mensaje 10307-11
10317 recibido: Mensaje 10311-13
10317 recibido: Mensaje 10314-11
10317 recibido: Mensaje 10312-14
10317 recibido: Mensaje 10313-12
10317 recibido: Mensaje 10315-11
10317 recibido: Mensaje 10310-13
10317 recibido: Mensaje 10316-13
10317 recibido: Mensaje 10308-11
10317 recibido: Mensaje 10313-13
10317 recibido: Mensaje 10309-12
10317 recibido: Mensaje 10313-14
10317 recibido: Mensaje 10314-12
10317 recibido: Terminado, se termina el programa
Error al enviar el mensaje: Invalid argument
Error al enviar el mensaje: Invalid argument
Error al enviar el mensaje: Invalid argument
Error al enviar el mensaje: Invalid argument
Error al enviar el mensaje: Invalid argument
Error al enviar el mensaje: Invalid argument
Error al enviar el mensaje: Invalid argument
Error al enviar el mensaje: Invalid argument
Error al enviar el mensaje: Invalid argument

[2]   Exit 1                  ./send.out
[3]   Exit 1                  ./send.out
[4]   Exit 1                  ./send.out
[5]   Exit 1                  ./send.out
[6]   Exit 1                  ./send.out
[7]   Done                    ./send.out
[8]   Exit 1                  ./send.out
[9]   Exit 1                  ./send.out
[10]   Exit 1                  ./send.out
[11]   Exit 1                  ./send.out
```

```
[12]-  Done                    ./recieve.out
```

Referencias

Dueñas, J. B. (2016). *Permisos del sistema de archivos en GNU/Linux*. AlcanceLibre. Consultado el 1 de noviembre de 2024, desde https://blog.alcancelibre.org/staticpages/index.php/permisos-sistema-de-archivos

IBM. (2023a). *subrutina msgrcv*. Consultado el 1 de noviembre de 2024, desde https://www.ibm.com/docs/es/aix/7.3?topic=m-msgrcv-subroutine

IBM. (2023b). *Subrutina msgsnd*. Consultado el 1 de noviembre de 2024, desde https://www.ibm.com/docs/es/aix/7.3?topic=m-msgsnd-subroutine

ipc.h. (2024). IBM. Consultado el 1 de noviembre de 2024, desde https://www.ibm.com/docs/es/aix/7.2?topic=files-ipch-file

Linux. (s.f.). *errno(3) — Linux manual page*. Consultado el 1 de noviembre de 2024, desde https://man7.org/linux/man-pages/man3/errno.3.html

Linux. (2024a). *mq_overview(7) — Linux manual page*. Consultado el 1 de noviembre de 2024, desde https://man7.org/linux/man-pages/man7/mq_overview.7.html

Linux. (2024b). *msgctl(2) — Linux manual page*. Consultado el 1 de noviembre de 2024, desde https://man7.org/linux/man-pages/man2/msgctl.2.html

Subrutina ftok. (2023). IBM. Consultado el 1 de noviembre de 2024, desde https://www.ibm.com/docs/es/aix/7.3?topic=f-ftok-subroutine

Tanenbaum, A. S., & Woodhull, A. S. (1997). *Sistemas Operativos: Diseño e implementación* (R. Escalona, Trad.; Segunda edición). Prentice Hall.