



SIMULADOR NAT

Comunicación entre procesos



DICIEMBRE 15, 2023

EQUIPO1

Marla Jasel Aguilar Aguilar | Ariana Corona González

Introducción:

En el presente trabajo, exploraremos la interacción entre procesos mediante el uso de semáforos y memoria compartida en el contexto de sistemas operativos. Estos conceptos, fundamentales en la programación concurrente, se aplicarán para facilitar la comunicación y coordinación entre diferentes programas, contribuyendo así a la comprensión de los principios clave de la sincronización y el intercambio de datos en ambientes multitarea.

En un entorno de sistemas operativos, la programación concurrente se vuelve esencial para aprovechar al máximo los recursos del sistema. La comunicación interprocesos (IPC) se convierte en un desafío que requiere soluciones eficientes y seguras. Aquí, semáforos y memoria compartida emergen como herramientas valiosas para facilitar la coordinación y el intercambio de información entre procesos.

El propósito principal de esta implementación es abordar la sincronización y la comunicación entre procesos de manera eficiente y simular un proceso NAT. Se aplicarán semáforos para controlar el acceso concurrente a recursos compartidos y la memoria compartida para intercambiar datos entre procesos.

En nuestro enfoque, se han diseñado tres programas independientes: el servidor, el cliente principal y el modificador del mensaje. Estos programas reflejan la naturaleza colaborativa de sistemas operativos multitarea, donde la coordinación y el intercambio de datos son fundamentales.

¿Cómo funcionan?

1. Servidor:

- Inicia una región de memoria compartida y establece semáforos para la sincronización.

```
#define NOMBRE_SEMAFORO "semaforo"  
#define NOMBRE_SEMAFORO2 "semaforo2"  
#define NOMBRE_SEMAFORO3 "semaforo3"  
#define NOMBRE_SEMAFORO4 "semaforo4"  
#define PERMISOS (S_IRUSR | S_IWUSR | S_IRGRP | S_IWGRP)  
#define FLAGS (O_CREAT | O_EXCL)
```

Se definen los semáforos

```

sem_unlink(name: NOMBRE_SEMAFORO);
sem_unlink(name: NOMBRE_SEMAFORO2);
sem_unlink(name: NOMBRE_SEMAFORO3);
sem_unlink(name: NOMBRE_SEMAFORO4);
sem_t *semaforo = sem_open(NOMBRE_SEMAFORO, FLAGS, PERMISOS, 0);
assert(semaforo != SEM_FAILED);
sem_t *semaforo2 = sem_open(NOMBRE_SEMAFORO2, FLAGS, PERMISOS, 0);
assert(semaforo2 != SEM_FAILED);
sem_t *semaforo3 = sem_open(NOMBRE_SEMAFORO3, FLAGS, PERMISOS, 0);
assert(semaforo3 != SEM_FAILED);
sem_t *semaforo4 = sem_open(NOMBRE_SEMAFORO4, FLAGS, PERMISOS, 0);
assert(semaforo4 != SEM_FAILED);

```

Se crean los semáforos, pero antes de crearlos los elimina para asegurarse de que los recursos compartidos estén limpios y no generen ningún problema.

```

int shm_fd = shm_open(nombre, O_CREAT | O_RDWR, 0600);
if(shm_fd == -1){
    perror(ErrMsg: "Error al crear la memoria compartida");
    return -1;
}
if(ftruncate(shm_fd, tam) == -1){
    perror(ErrMsg: "Error al vaciar la memoria");
    return -1;
}

zona_memoria = mmap(NULL, tam, PROT_READ | PROT_WRITE, MAP_SHARED, shm_fd, 0);
if(zona_memoria == MAP_FAILED){
    perror(ErrMsg: "Error al mapear la memoria compartida");
    return -1;
}

```

Utiliza shm_open para crear la memoria compartida, luego con ftruncate establece su tamaño y por ultimo con mmap la mapea en el espacio de direcciones.

- Espera a que el cliente principal escriba un mensaje.

```
sem_wait(&sem: semaforo);
//fgets(mensaje, 200, stdin);
strncpy(mensaje, zona_memoria, 200);
printf(format: "El mensaje recibido es %s", mensaje);
printf(format: "Enviando el mensaje al convertidor");
sprintf(zona_memoria, "%s", mensaje);
sem_post(&sem: semaforo2);
```

Con el semáforo se espera que el usuario ingrese un mensaje para que el servidor pueda leerlo

- Notifica al cliente principal sobre la disponibilidad del mensaje procesado

En la misma imagen se puede observar que manda un mensaje informando al cliente de lo que está haciendo.

```
printf(format: "El mensaje recibido es %s", mensaje);
printf(format: "Enviando el mensaje al convertidor");
sprintf(zona_memoria, "%s", mensaje);
sem_post(&sem: semaforo2);
```

2. Cliente Principal:

- Accede a la memoria compartida y utiliza semáforos para coordinar la escritura y lectura.

```
sem_t *semaforo = sem_open(NOMBRE_SEMAFORO, FLAGS, PERMISOS, 0);
assert(semaforo != SEM_FAILED);
sem_t *semaforo4 = sem_open(NOMBRE_SEMAFORO4, FLAGS, PERMISOS, 0);
assert(semaforo4 != SEM_FAILED);

int shm_fd = shm_open(nombre, O_RDWR, 0600);
if(shm_fd == -1){
    perror(ErrMsg: "Error al abrir la memoria compartida");
    return -1;
}
```

Utiliza los semáforos, no tiene permiso para crearlos solo para utilizarlos.

```
zona_memoria = mmap(NULL, tam, PROT_READ | PROT_WRITE, MAP_SHARED, shm_fd, 0);
if(zona_memoria == MAP_FAILED) {
    perror(ErrMsg: "Error al mapear la memoria compartida");
    return -1;
}
```

Accede al espacio de memoria compartida con permisos de lectura y escritura.

- Introduce un mensaje en la memoria compartida.

```
printf( format: "Mensaje a enviar: ");  
fgets( Buf: mensaje, MaxCount: 200, File: stdin);  
  
sprintf(zona_memoria, "%s", mensaje);  
sem_post( sem: semaforo);
```

Recibe el mensaje ingresado por el usuario y lo escribe en la memoria con sprintf.

- Notifica al servidor sobre la disponibilidad del mensaje y espera que el servidor lo comunique con el transformador.

Con sem_post le avisa al servidor que el usuario ya ingresó un mensaje, y así el servidor puede leerlo y “mandarlo” a el transformador.

- Confirma la recepción del mensaje modificado por el traductor.

```
sem_wait( sem: semaforo4);  
strncpy(mensaje, zona_memoria, 200);  
printf( format: "El mensaje recibido es: %s", mensaje);
```

Recibe el mensaje ya transformado y lo muestra en pantalla.

3. Transformador:

- Procesa el mensaje, lo transforma y lo almacena en la memoria compartida.

```
sem_wait(semaforo2);  
strncpy(mensaje, zona_memoria, 200);  
convertirAMayusculas( cadena: mensaje);  
printf("mensaje recibido, convirtiendo mensaje");  
  
// Copiar la cadena convertida de vuelta a la memoria compartida  
strncpy(zona_memoria, mensaje, 200);
```

En esta parte del código el transformador recibe el mensaje y lo convierte a mayúsculas

- Utiliza semáforos para sincronizar con el servidor y acceder a la memoria compartida.

```
int shm_fd = shm_open(nombre, O_RDWR, 0600);
if(shm_fd == -1){
    perror("Error al abrir la memoria compartida");
    return -1;
}

zona_memoria = mmap(NULL, tam, PROT_READ | PROT_WRITE, MAP_SHARED, shm_fd, 0);
if(zona_memoria == MAP_FAILED) {
    perror("Error al mapear la memoria compartida");
    return -1;
}
```

Accede a la memoria compartida con permisos de lectura y escritura.

```
sem_wait(semaforo2);          sem_post(semaforo3);
strcpy(mensaje, zona_memoria + 2);
```

Se utilizan semáforos para sincronizarlo con otros procesos.

- Espera a que el servidor notifique la disponibilidad de un nuevo mensaje.

```
sem_wait(semaforo2);
strcpy(mensaje, zona_memoria + 2);
```

Este semáforo es utilizado para que el transformador este en espera en lo que recibe un mensaje del servidor.

- Modifica el mensaje, en este caso, convirtiéndolo a mayúsculas.

```
convertirAMayusculas( cadena: mensaje);
printf("mensaje recibido, convirtiendo mensaje");
```

Se convierte el mensaje y el transformador le avisa al usuario que lo está cambiando a mayúsculas.

- Libera el semáforo para notificar al servidor sobre la finalización del procesamiento.

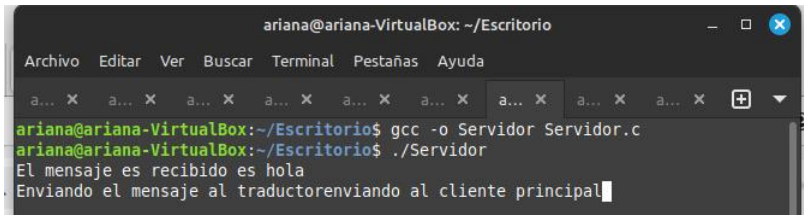
```
sem_post(semaforo3);
```

Se libera el semáforo para que el servidor pueda leer el mensaje convertido y mandarlo al cliente.

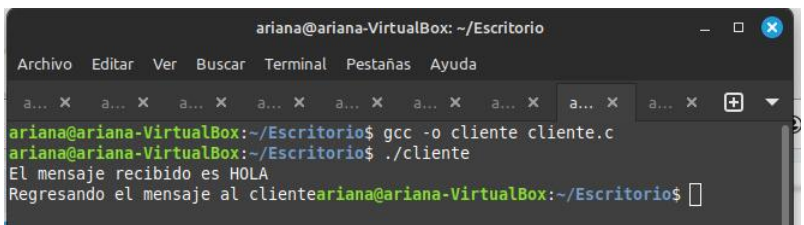
4. Ejecutar

Para que el archivo pueda ser ejecutable se utiliza el comando **gcc -o nombreArchivo nombreArchivo.extension** que en este caso estuvimos trabajando con el lenguaje de programación C

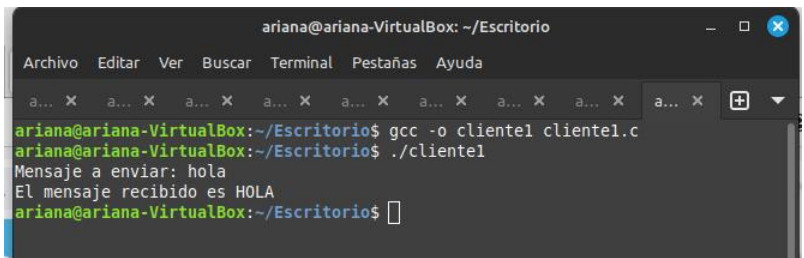
Seguido de hacer el archivo ejecutable se corre el programa con el comando **./nombreArchivo** se hace lo mismo para cada programa en una ventana diferente ejecutando en primer lugar al servidor, en segundo lugar, al cliente y en tercer lugar al cliente 1 que será desde donde se envíe el mensaje, gradualmente en cada ventana aparecerá el mensaje enviado y el mensaje recibido.



```
ariana@ariana-VirtualBox: ~/Escritorio
Archivo Editar Ver Buscar Terminal Pestañas Ayuda
a... X a... X a... X a... X a... X a... X a... X a... X a... X
ariana@ariana-VirtualBox:~/Escritorio$ gcc -o Servidor Servidor.c
ariana@ariana-VirtualBox:~/Escritorio$ ./Servidor
El mensaje es recibido es hola
Enviando el mensaje al traductorenviando al cliente principal
```



```
ariana@ariana-VirtualBox: ~/Escritorio
Archivo Editar Ver Buscar Terminal Pestañas Ayuda
a... X a... X a... X a... X a... X a... X a... X a... X a... X
ariana@ariana-VirtualBox:~/Escritorio$ gcc -o cliente cliente.c
ariana@ariana-VirtualBox:~/Escritorio$ ./cliente
El mensaje recibido es HOLA
Regresando el mensaje al cliente
```



```
ariana@ariana-VirtualBox: ~/Escritorio
Archivo Editar Ver Buscar Terminal Pestañas Ayuda
a... X a... X a... X a... X a... X a... X a... X a... X a... X
ariana@ariana-VirtualBox:~/Escritorio$ gcc -o cliente1 cliente1.c
ariana@ariana-VirtualBox:~/Escritorio$ ./cliente1
Mensaje a enviar: hola
El mensaje recibido es HOLA
```

Importancia:

Este trabajo subraya la esencial importancia de técnicas fundamentales como semáforos y memoria compartida en el ámbito de la programación concurrente, ofreciendo una perspectiva aplicada en el contexto específico de sistemas operativos. En entornos operativos modernos, la implementación efectiva de estas herramientas se revela como un componente crítico para la instauración de una comunicación segura y eficiente entre procesos. La capacidad de coordinar la ejecución de múltiples tareas de manera sincronizada mediante semáforos, y la facilitación del intercambio directo de datos entre procesos a través de la memoria compartida, se vuelven esenciales para maximizar la utilización de los recursos del sistema. En este escenario, la aplicación práctica de estas técnicas no solo optimiza la eficiencia en la ejecución de tareas concurrentes, sino que también contribuye de manera significativa a la robustez y fiabilidad de los sistemas operativos modernos.