Instituto Tecnológico de Buenos Aires

TRABAJO PRÁCTICO NÚMERO 1: INTER PROCESS COMMUNICATION

72.11 - Sistemas Operativos - Grupo 5

Autores:

Bernasconi, Ian - 62867

Feferovich, Jeremías - 62701

Gutierrez, Agustin - 62595

Índice

1. Introducción

En el presente informe se muestra el desarrollo del Trabajo Práctico número 1 de la materia Sistemas Operativos (72.11). El mismo se enfoca en Inter Process Communication (IPC) e implementa un sistema de distribución de cómputo para la generación del hash md5 de múltiples archivos entre varios procesos. Los IPCs utilizados en el trabajo son *pipes* y *shared memory*.

2. Decisiones tomadas

Para comunicarse con los esclavos, se utilizaron un par de pipes para cada uno: uno para enviar los nombres de los archivos y otro para leer los resultados. Definimos una longitud máxima de nombre de archivo de 1024 caracteres, y lo enviamos por el pipe de escritura al esclavo. El esclavo lo lee hasta esa cantidad de caracteres usando *getline*, lo cual puede bloquear al esclavo, el cual espera a que se le envíe un nuevo archivo para procesar. Para sincronizar del lado de la aplicación, usamos la función *select*, que espera que haya información para leer de algún esclavo, y en ese momento lee los datos del archivo procesado. Si todavía no se procesaron todos los archivos, le envía otro para procesar.

Para la comunicación entre proceso aplicación y vista, utilizamos semáforos como mecanismo de IPC para el uso de memoria compartida. Ambos procesos utilizan una librería de funciones *shm_buffer* para conectarse a la memoria compartida. Esta librería utiliza internamente una estructura llamada *shmData*, definida en *shm_buffer.c*. Esta estructura contiene dos punteros a un buffer ubicado en unica posición de memoria, uno de los cuales es mapeado a la memoria virtual del proceso la cual lo usa para escribir los datos procesados y el otro es mapeado en la memoria del proceso vista para leer los datos. Además, posee un semáforo llamado *readyFiles* que indica la cantidad de archivos ya procesados, lo que le permite al proceso vista imprimir los datos de un archivo. Por último, posee variables sobre la misma memoria compartida, lo que permite que el proceso vista se una solo con el PID de aplicación.

Para facilitar el desarrollo de código, se definieron longitudes máximas para el nombre del archivo y el PID de los archivos.

3. Limitaciones

Una de las limitaciones del código es que no se pueden ejecutar al mismo tiempo 2 procesos vista para un mismo proceso aplicación, porque hay un único puntero al buffer de lectura en textitshm_buffer.
Por otra parte, ningún archivo puede tener un nombre de más de 1024 caracteres, ya que el buffer utilizado para la memoria compartida tiene un espacio de este tamaño para el filename.

4. Diagrama

A continuación se muestra un diagrama de la conexión de los distintos procesos participantes:

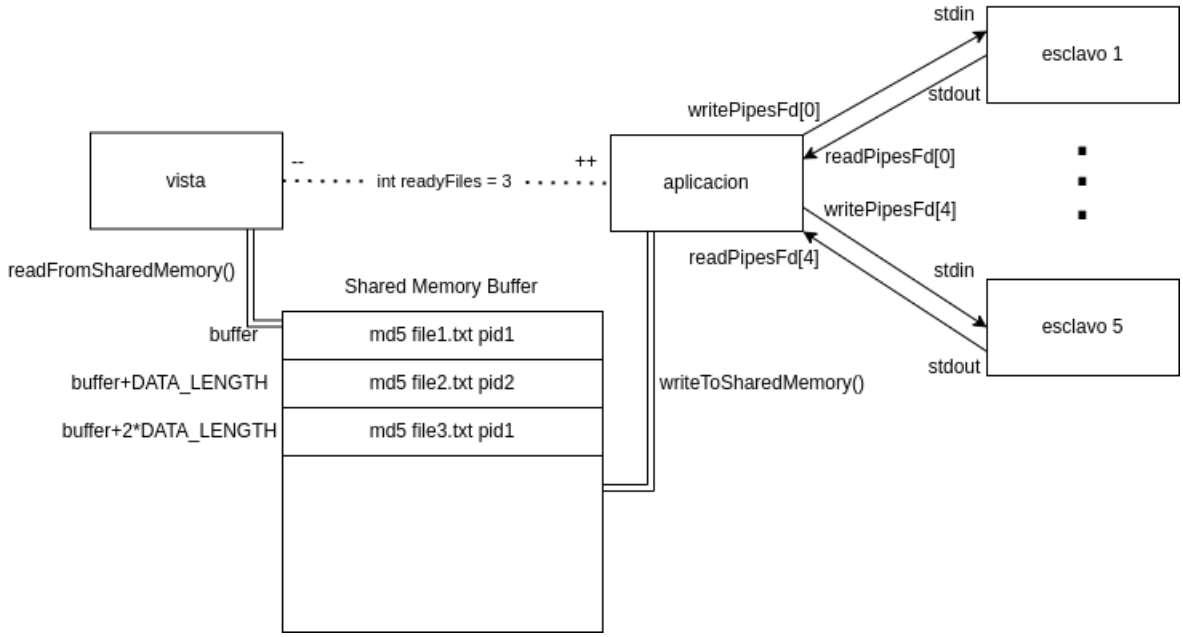

Diagrama de conexión de procesos

5. Instrucciones de compilación y ejecución

Para la compilación, se debe realizar el comando

```
$ make all
```

Para ejecutarlo hay diferentes opciones: si no se quiere ver la salida por la consola en tiempo real, se puede ejecutar solamente:

```
$ ./aplicacion.out <archivos>
```

y ver el resultado en el archivo *results.txt*.

Para ver la salida en tiempo real hay 2 opciones, usar un pipe entre el proceso aplicación y el proceso vista, de la siguiente forma:

```
$ ./aplicacion.out <archivos> | ./vista.out
```

Por último, se puede ejecutar

```
$ ./aplicacion.out <archivos> &
```

y luego, antes de los 2 segundos, ejecutar, copiando el valor del PID de aplicación que se imprime por consola.

```
$ ./vista.out <PID aplicacion>
```

6. Problemas encontrados

Un problema que tuvimos fue que al tener demasiados archivos iniciales el buffer se llenaba y el programa no funcionaba correctamente, por lo que se tuvo que limitar la cantidad de archivos iniciales, como se puede ver en la función *calculateInitialFilesPerSlave*.

Nos costó delimitar el fin del nombre de archivo enviado desde aplicación a esclavo con un \n, ya que la lectura desde el lado del esclavo presentaba problemas. Esto fue solucionado utilizando la función *getline* en vez de *read*.

El análisis con PVS-Studio arroja el siguiente error:

```
/root/aplicacion.c 154 warn V755 A copy from unsafe
data source to a buffer of fixed size. Buffer overflow is possible.
```

el cual se refiere a la linea de código:

```
int stringLen = sprintf(string, "md5: %s || ID: %d ||
                filename: %s%c", md5result, pidSlaves[slaveNumber],
                argv[fileIndexReadFromSlave], '\0');
```

Pero mientras las longitudes del PID y de los nombres de archivo estén dentro de las limitaciones mencionadas anteriormente, esta función no debería generar problemas.

Otro problema encontrado fue que cuando la cantidad de archivos era menor a la cantidad de esclavos, el programa no funcionaba correctamente. Esto fue solucionado chequeando cuando se envían los archivos iniciales que no se hayan enviado todos los archivos y modificando el calculo de la cantidad de archivos iniciales para que no diera 0.

7. Citas de fragmentos de código

Para el manejo de Shared Memory, utilizamos como base el ejemplo disponible en *man shm_open*. El mismo se muestra a continuación:
archivo *pshm_ucase.h*:

```
#include <sys/mman.h>
#include <fcntl.h>
#include <semaphore.h>
#include <sys/stat.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

#define errExit(msg)    do { perror(msg); exit(EXIT_FAILURE); \
                        } while (0)

#define BUF_SIZE 1024   /* Maximum size for exchanged string */

/* Define a structure that will be imposed on the shared
   memory object */

struct shmbuf {
    sem_t  sem1;            /* POSIX unnamed semaphore */
    sem_t  sem2;            /* POSIX unnamed semaphore */
    size_t cnt;             /* Number of bytes used in 'buf' */
    char   buf[BUF_SIZE];   /* Data being transferred */
};
```

archivo *pshm_ucase.c*:

```
 #include <ctype.h>
#include "pshm_ucase.h"

int
main(int argc, char *argv[])
{
    if (argc != 2) {
        fprintf(stderr, "Usage: %s /shm-path\n", argv[0]);
        exit(EXIT_FAILURE);
```

```
10          }
11
12          char *shmpath = argv[1];
13
14          /* Create shared memory object and set its size to the size
15             of our structure. */
16
17          int fd = shm_open(shmpath, O_CREAT | O_EXCL | O_RDWR,
18                            S_IRUSR | S_IWUSR);
19          if (fd == -1)
20              errExit("shm_open");
21
22          if (ftruncate(fd, sizeof(struct shmbuf)) == -1)
23              errExit("ftruncate");
24
25          /* Map the object into the caller's address space. */
26
27          struct shmbuf *shmp = mmap(NULL, sizeof(*shmp),
28                                     PROT_READ | PROT_WRITE,
29                                     MAP_SHARED, fd, 0);
30           if (shmp == MAP_FAILED)
31                  errExit("mmap");
32
33          /* Initialize semaphores as process-shared, with value 0. */
34
35          if (sem_init(&shmp->sem1, 1, 0) == -1)
36              errExit("sem_init-sem1");
37          if (sem_init(&shmp->sem2, 1, 0) == -1)
38              errExit("sem_init-sem2");
39
40          /* Wait for 'sem1' to be posted by peer before touching
41             shared memory. */
42
```

```
43          if (sem_wait(&shmp->sem1) == -1)
44              errExit("sem_wait");
45
46          /* Convert data in shared memory into upper case. */
47
48          for (int j = 0; j < shmp->cnt; j++)
49              shmp->buf[j] = toupper((unsigned char) shmp->buf[j]);
50
51          /* Post 'sem2' to tell the peer that it can now
52             access the modified data in shared memory. */
53
54          if (sem_post(&shmp->sem2) == -1)
55              errExit("sem_post");
56
57          /* Unlink the shared memory object. Even if the peer process
58             is still using the object, this is okay. The object will
59             be removed only after all open references are closed. */
60
61          shm_unlink(shmpath);
62
63          exit(EXIT_SUCCESS);
64      }
```

archivo *pshm_ucase_send.c*:

```
1       #include <string.h>
2       #include "pshm_ucase.h"
3
4       int
5       main(int argc, char *argv[])
6       {
7           if (argc != 3) {
8               fprintf(stderr, "Usage: %s /shm-path string\n", argv[0]);
9               exit(EXIT_FAILURE);
10          }
```

```
11
12          char *shmpath = argv[1];
13          char *string = argv[2];
14          size_t len = strlen(string);
15
16          if (len > BUF_SIZE) {
17              fprintf(stderr, "String is too long\n");
18              exit(EXIT_FAILURE);
19          }
20
21          /* Open the existing shared memory object and map it
22             into the caller's address space. */
23
24          int fd = shm_open(shmpath, O_RDWR, 0);
25          if (fd == -1)
26              errExit("shm_open");
27
28          struct shmbuf *shmp = mmap(NULL, sizeof(*shmp),
29                                     PROT_READ | PROT_WRITE,
30                                     MAP_SHARED, fd, 0);
31
32           /* Copy data into the shared memory object. */
33
34          shmp->cnt = len;
35          memcpy(&shmp->buf, string, len);
36
37          /* Tell peer that it can now access shared memory. */
38
39          if (sem_post(&shmp->sem1) == -1)
40              errExit("sem_post");
41
42          /* Wait until peer says that it has finished accessing
43             the shared memory. */
```

```
44
45              if (sem_wait(&shmp->sem2) == -1)
46                  errExit("sem_wait");
47
48          /* Write modified data in shared memory to standard output.*/
49
50              write(STDOUT_FILENO, &shmp->buf, len);
51              write(STDOUT_FILENO, "\n", 1);
52
53              exit(EXIT_SUCCESS);
54          }
```