72.11 Sistemas Operativos

Informe del Trabajo Práctico Nº1

Inter Process Communication

Link de GitHub
https://github.com/jimenalozano/SOtp1

Hash del Commit
53a2afeb29369f5eba1c47d612b85ac18fdc24d9

Integrantes del Grupo nº4:

- Emilio Basualdo Cibils 58172
- Jimena Lozano 58095
- Pedro Remigio Pingarilho 58451

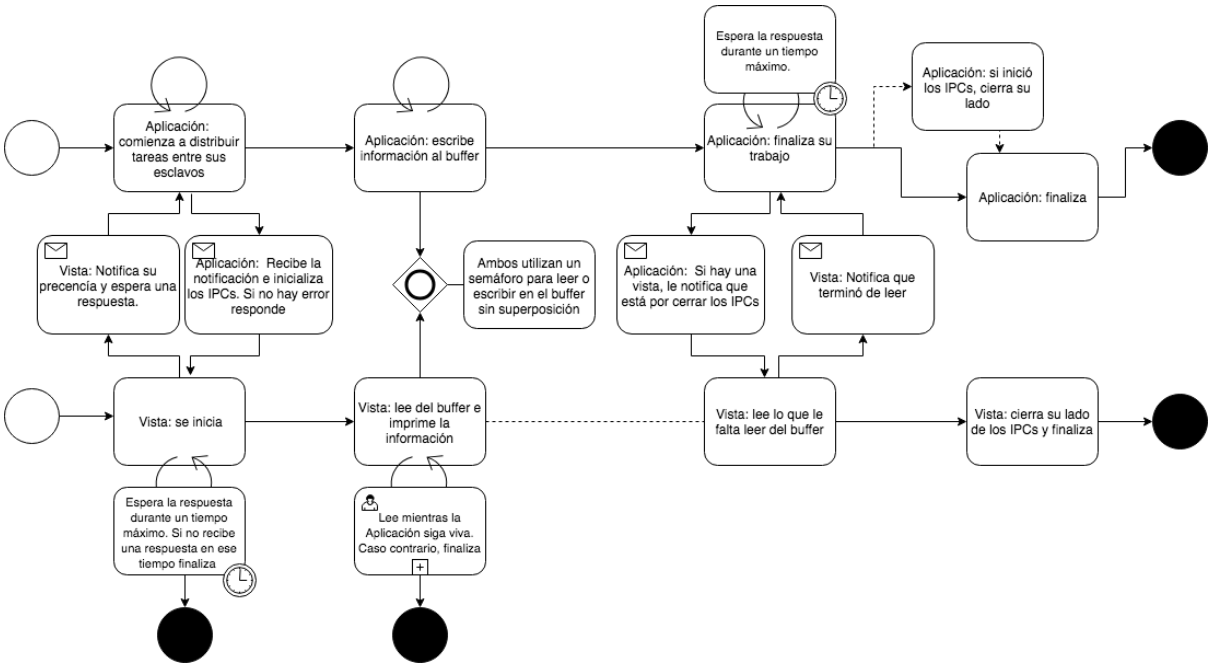
Decisiones

Comunicación Aplicación - Vista

Como lo especifica los requerimientos del TP, se utilizó semáforos y memoria compartida, ambos con nombres. Además, utilizamos la señal definida como *SIGUSR2* de la biblioteca <signal.h> para notificar la presencia de la Vista y el fin de la Aplicación.

Decidimos iniciar los IPCs (semáforo y memoria compartida) únicamente en el caso en que se corriera un proceso Vista. No parecía coherente iniciar estos, si no hay necesidad de una comunicación.

El siguiente diagrama describe la comunicación entre estas partes.



El protocolo de comunicación entre estas partes está definido en el archivo fuente "view*Protocol.h*".

Señales

El proceso Vista recibe como argumento o leyendo de un archivo el *pid* del proceso Aplicación y utiliza la función *kill(app_pid, SIGUSR2)* para enviar una señal notificando su intención de leer del buffer.

Al iniciar el proceso Aplicación, creamos un *handler* para la señal *SIGUSR2*. Este *handler* se encarga de iniciar los IPCs y, si esto no genera un error, envía una señal a la Vista *kill(vista_pid,SIGUSR2*) notificando el correcto funcionamiento de los IPCs. A partir de ahí, la Vista comienza a leer.

La Vista lee únicamente si la Aplicación está viva.

Dado que se le notificó previamente sobre una Vista leyendo del buffer, cuando la Aplicación termina su trabajo notifica a la Vista de la misma manera: *kill(vista_pid, SIGUSR2*) pero no cierra los IPCs hasta recibir una respuesta de vuelta. Se decidió esto para darle la posibilidad a la Vista de leer información que aún no haya leído, antes de cerrar los IPCs.

Semáforos y Memoria Compartida

Ambos son inicializados con nombres definidos por el *protocolo*. Su uso es el común.

El buffer de la información es mapeado según el *file descriptor* de memoria compartida, habiendo truncado el *file* descriptor al tamaño requerido por el buffer.

Utilizamos 2 semáforos, *readSem* y *writeSem*, sus nombres están definidos en el archivo fuente. Su uso es “cruzado”; la Aplicación habilita a la Vista a leer haciendo *sem_post(readSem)*, y la Vista habilita a la Aplicación a escribir haciendo *sem_post(writeSem)*, mientras que cada uno espera en su semáforo para leer/escribir.

Comunicación Aplicación - Esclavos

La comunicación entre la aplicación y los esclavos se lleva a cabo con pipes.

Para que el proceso padre, la aplicación, se comunique con sus esclavos, se utilizan 2 pipes para cada uno de sus hijos. De esta manera, el padre puede elegir qué mandarle a cada hijo, porque sabe donde escribirle para que sólo su hijo pueda leer, y además sabe que una vez que el hijo termine su función, va a mandar el resultado a un pipe específico. Para que el padre sepa cuándo leer, usa la función *select*. De esta manera, va a saber cuantos hijos terminaron de procesar el md5 y si están dispuestos a recibir más trabajos. Una vez que no hayan más paths para procesar, el padre simplemente espera a que sus hijos terminen de trabajar.

Estructuras de datos

Para almacenar datos en la aplicación, se decidió utilizar listas en su gran mayoría. Los datos guardados en la aplicación con mayor importancia son los archivos, los procesos hijos, y los hashes.

Para guardar los archivos, se decidió ordenarlos por tamaño, para que los esclavos trabajen de la manera más parecida posible en cuanto a tiempo. Se logró aquello almacenando estructuras (nodos) que describen lo que necesitamos sobre los archivos (los *paths*, el tamaño) en una lista doblemente encadenada.

Para guardar los PID de los esclavos, se utilizaron estructuras (nodos) que describen el PID del proceso en una lista simplemente encadenada. Se decidió esto sobre almacenarlos sobre un arreglo porque el tamaño del arreglo podría variar, dependiendo de cuántos archivos se pasen como parámetros.

Por último, se guardaron los hashes de cada archivo en un puntero a punteros a caracteres. Se inicializa con un tamaño igual a la cantidad de archivos procesados, y en cada posición se ubica el *path* de un archivo procesado. Luego, se encuentra el *path* del archivo para ubicar en conjunto su hash. Se decidió de esta manera para llevar la cuenta de cuáles archivos fueron procesados por los esclavos y cuáles no.

Instrucciones

Compilación

Compilamos con gcc los siguientes flags:

- -Wall: opcional para mostrar warnings
- -pthread: requerido para utilizar los semáforos
- -lrt: requerido para utilizar la memoria compartida

Ejemplo:

$ gcc -o view.o view.c -Wall -lrt -pthread ; gcc -o aplication.o aplication.c -Wall -lrt -pthread ; gcc -o slaves.o slaves.c -Wall -lrt -pthread

Ejecución

La aplicación puede ser corrida por su cuenta sin necesidad de la Vista.

```
$: ./aplication.o /*
```

Al iniciar, la Aplicación esperará algunos segundos a que apareca una Vista, si no ocurre esto continuará con su trabajo. Para saltear la espera apretar 'enter'(\n).

Si queremos correr la Vista junto con la Aplicación, debemos **primero** correr la Aplicación en el *background* y **luego** la Vista, ó correr ambos programas en distintas *Shells* de la misma instancia del sistema respetando el orden.

La Aplicación imprime en salida estándar y en un archivo, cuyo nombre está definido en "viewProtocol.h", su PID. De esta manera la Vista puede levantar el PID de la Aplicación "automáticamente" o recibirlo por línea de comando.

```
Caso 1:
     $: ./aplication.o /* &
     $: ./view.o

Caso 2:
     $: ./aplication.o /*
     $: ./view.o         ←--- en otra instancia de shell

Caso 3:
     $: ./aplication.o /*
     $: ./view.o [PID]   ←--- en otra instancia de shell
```

La Aplicación se encarga de ejecutar a los procesos esclavos por su cuenta.

Ambos procesos se ejecutaron sobre el sistema provisto por la cátedra dada la imágen de docker : “agodio/itba-so”

En cuanto al testeo decidimos comparar el hash de cada archivo generado directamente con *md5sum*, con la lista de resultados generada por la Aplicación e impresa en el archivo.

Para ejecutar el test, **primero** eliminamos cualquier archivo cuyo nombre comience con “md5Hash_”, **luego** ejecutamos la aplicación, **luego** ejecutamos el test. Se debe ejecutar en este orden ya que el test abre el último y único archivo creado por la aplicación y lo lee iterando linea por linea los hashs.

Para ejecutar el test se debe correr de la siguiente manera:

$: ./unityTest.o

Limitaciones

La limitación más importante toma lugar en la cantidad de archivos que se pueden procesar: el *path* de todos los archivos tiene que ocupar como mucho BUFFER_SIZE-1 por pipe. Esto se debe al hecho de que se decidió usar un protocolo entre los esclavos y la aplicación para el pasado de archivos en los pipes, donde se usa una constante BUFFER_SIZE, un número que indica el tamaño que debe ocupar el buffer para leer de los pipes que le llegan a los esclavos. De esta manera, cuando se quiere distribuir entre los esclavos más de 1 archivo para cada uno, los paths de cada archivo en el buffer no puede ocupar más de BUFFER_SIZE.

Por otro lado, no logramos hacer que el proceso Vista no retenga el avance del proceso Aplicación. Al, usar semáforos, los procesos están obligados a esperarse entre sí. Es cierto que la Vista es quien más espera, sin embargo, cuando la Vista lee de la memoria compartida, retiene a la Aplicación, si esta quiere escribir. Lo ideal sería que la Aplicación no tenga que esperar para nada a la Vista.

Problemas

Durante el desarrollo de la comunicación entre la Vista y la Aplicación, teníamos el problema de que en caso de que alguno de los procesos terminara inesperadamente, los IPCs no se cerraban correctamente y permanecían abiertos en el sistema.

Además, en el caso de que no se haya deseado iniciar la Vista, los IPCs se inicializaban sin necesidad de compartir memoria. Para solucionar esto, decidimos implementar la comunicación con señales.

Por otra parte, cuando se realizó la comunicación entre los esclavos y la aplicación, la cantidad de pipes requerida para la comunicación era casi la mitad, ya que se intentó usar señales para la comunicación de los esclavos hacia los padres (para enviar los hashes). Cuando no se pudo realizar con el tiempo que se contaba, decidimos que la comunicación sea absolutamente con pipes: 2 para cada proceso hijo.

Citas

- https://www.linuxprogrammingblog.com/code-examples/signal-waiting-sigtimedwait

```
sigset_t mask;
sigset_t orig_mask;
struct timespec timeout;
pid_t pid;

sigemptyset (&mask);
sigaddset (&mask, SIGCHLD);

if (sigprocmask(SIG_BLOCK, &mask, &orig_mask) < 0) {
    perror ("sigprocmask");
    return 1;
}

pid = fork_child ();

timeout.tv_sec = 5;
timeout.tv_nsec = 0;

do {
    if (sigtimedwait(&mask, NULL, &timeout) < 0) {
        if (errno == EINTR) {
            /* Interrupted by a signal other than SIGCHLD. */
            continue;
        }
        else if (errno == EAGAIN) {
            printf ("Timeout, killing child\n");
            kill (pid, SIGKILL);
        }
        else {
            perror ("sigtimedwait");
            return 1;
        }
    }

    break;
} while (1);
```

- https://stackoverflow.com/questions/25261/set-and-oldset-in-sigprocmask

```
sigset_t x;
sigemptyset (&x);
sigaddset(&x, SIGUSR1);
sigprocmask(SIG_BLOCK, &x, NULL)
```

- https://stackoverflow.com/questions/3395690/md5sum-of-file-in-linux-c

```
FILE * file = popen(md5SumFunc, "r");

int i;
int ch;
pathNameHash[size] = ' ';

for (i = 1; i < 33 && isxdigit(ch = fgetc(file)); i++)
{
    pathNameHash[size+i] = ch;
}
pathNameHash[size+i] = '\0';
```

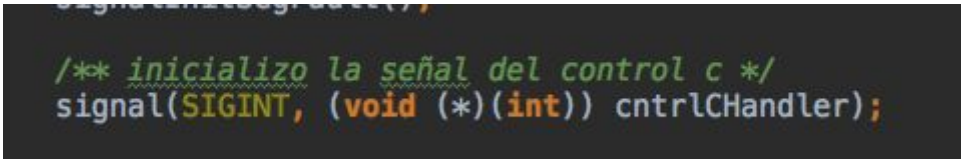
- https://stackoverflow.com/questions/10202941/segmentation-fault-handling

```
void signalInitSegFault()
{
    struct sigaction sigIntHandler;
    sigIntHandler.sa_handler = (void (*)(int)) segFaultHandler;
    sigemptyset(&sigIntHandler.sa_mask);
    sigIntHandler.sa_flags = 0;
    sigaction(SIGINT, &sigIntHandler, NULL);
    sigaction(SIGSEGV, &sigIntHandler, NULL);

}

void segFaultHandler(int signum, siginfo_t *info, void *ucontext)
{
    exit(EXIT_FAILURE);
}
```

- https://stackoverflow.com/questions/17766550/ctrl-c-interrupt-event-handling-in-linux



```
/** inicializo la señal del control c */
signal(SIGINT, (void (*)(int)) cntrlCHandler);
```

- https://stackoverflow.com/questions/8149569/scan-a-directory-to-find-files-in-c

```
char * hasFile()
{
    int found = 1;
    DIR *dp;
    struct dirent *entry;
    struct stat statbuf;
    char * rt = NULL;

    if((dp = opendir(".")) == NULL)
    {
        fprintf(stderr,"cannot open directory: %s\n", ".");
        return NULL;
    }
    while(found && (entry = readdir(dp)) != NULL)
    {
        lstat(entry->d_name,&statbuf);
        if(S_ISDIR(statbuf.st_mode))
        {
            /* Found a directory, but ignore . and .. */
            if(strcmp(".",entry->d_name) == 0 || strcmp("..",entry->d_name) == 0)
                continue;
        }
        else
        {
            if(isFile(entry->d_name))
            {
                found = 0;
                rt = entry->d_name;
            }
        }
    }
    closedir(dp);
    return rt;
}
```

Link de GitHub
https://github.com/jimenalozano/SOtp1

hash del commit
53a2afeb29369f5eba1c47d612b85ac18fdc24d9