

Memoria Práctica 3 -

Sistemas Operativos

Manuel Cintado y Manuel Suárez Román

Doble Grado Matemáticas e Ingeniería Informática

Pareja 09

Contenido

[Ejercicio 2 3](#_Toc5961748)

[Ejercicio 3 10](#_Toc5961749)

[Código productor 10](#_Toc5961750)

[Código consumidor 15](#_Toc5961751)

[Ejercicio 4 17](#_Toc5961752)

[Código Proceso A 18](#_Toc5961753)

[Código Proceso B 20](#_Toc5961754)

[Código Proceso C 23](#_Toc5961755)

[Código Main 25](#_Toc5961756)

A lo largo de esta memoria, a la hora de introducir el código se han eliminado los #include y los comentarios a fin de no hacer la memoria innecesariamente larga.

Ejercicio 2

En este ejercicio se nos pide implementar un programa que gestione un espacio de memoria compartida de manera que los hijos puedan ir modificando unos datos determinados que mostrará el padre.

a) Código

#define SHM\_NAME "/clientInfo"

#define NAME\_MAX 50

typedef struct {

int previous\_id;

int id;

char name[NAME\_MAX];

} ClientInfo;

ClientInfo \*clienteinformacion;

void manejador\_SIGUSR1(int sig){

printf("recibida SIGUSR1\n");

printf("ID: %d\nPrevious\_id: %d\nName: %s\n", clienteinformacion->id, clienteinformacion->previous\_id, clienteinformacion->name);

}

void manejador\_SIGINT(int sig){

printf("recibida SIGINT\n");

munmap(clienteinformacion, sizeof(\*clienteinformacion));

shm\_unlink(SHM\_NAME);

exit(EXIT\_FAILURE);

}

int main(int argc, char \*argv[]){

int n, i;

pid\_t pid;

int fd\_shm;

int error;

char nombreaux[NAME\_MAX];

struct sigaction act;

if(argc != 2) {

printf("Introduzca el numero de procesos hijo\n");

exit(EXIT\_FAILURE);

}

n = atoi(argv[1]);

sigemptyset(&(act.sa\_mask));

act.sa\_handler = manejador\_SIGINT;

act.sa\_flags = 0;

if(sigaction(SIGINT, &act, NULL) < 0){

perror("Sigaction");

exit(EXIT\_FAILURE);

}

fd\_shm = shm\_open(SHM\_NAME, O\_RDWR | O\_CREAT | O\_EXCL, S\_IRUSR | S\_IWUSR);

if(fd\_shm == -1) {

fprintf (stderr, "Error creando el segmento de memoria compartida \n");

return EXIT\_FAILURE;

}

error = ftruncate(fd\_shm, sizeof(ClientInfo));

if(error == -1) {

fprintf (stderr, "Error redimensionando el segmento de memoria compartida \n");

shm\_unlink(SHM\_NAME);

return EXIT\_FAILURE;

}

clienteinformacion = (ClientInfo \*)mmap(NULL, sizeof(\*clienteinformacion), PROT\_READ | PROT\_WRITE, MAP\_SHARED, fd\_shm, 0);

if(clienteinformacion == MAP\_FAILED) {

fprintf (stderr, "Error mapeando el egmento de memoria compartida \n");

shm\_unlink(SHM\_NAME);

return EXIT\_FAILURE;

}

clienteinformacion->id = 0;

clienteinformacion->previous\_id = -1;

for(i = 0; i < n; ++i){

pid = fork();

if(pid < 0){

perror("fork");

exit(EXIT\_FAILURE);

}

else if(pid == 0){

srand(pid);

sleep(rand()%10 +1);

clienteinformacion->previous\_id ++;

printf("Introduzca el nombre del nuevo usuario ( %d)\n", getpid());

scanf("%s",nombreaux);

strcpy(clienteinformacion->name, nombreaux);

clienteinformacion->id++;

kill(getppid(), SIGUSR1);

exit(EXIT\_SUCCESS);

}

}

sigemptyset(&(act.sa\_mask));

act.sa\_handler = manejador\_SIGUSR1;

act.sa\_flags = 0;

if(sigaction(SIGUSR1, &act, NULL) < 0){

perror("Sigaction");

exit(EXIT\_FAILURE);

}

while(wait(NULL)>0);

munmap(clienteinformacion, sizeof(\*clienteinformacion));

shm\_unlink(SHM\_NAME);

exit(EXIT\_SUCCESS);

return EXIT\_SUCCESS;

}

b) Tal y como está planteado el problema en la versión mostrada en el apartado a, no funciona correctamente, pues es necesaria la introducción de los semáforos vistos en la práctica 2, pues en caso contrario se por dicen conflictos de escritura, ya que todos los hijos piden el nombre del nuevo cliente a la vez. Con la introducción de semáforos conseguiremos que esta lectura del nombre se realice de manera secuencial y ordenada.

c)Para solucionar los problemas, simplemente introducimos un semáforo que se encargará de controlar la escritura en el espacio de memoria compartida.

*Código*

#define SHM\_NAME "/clientInfo"

#define SEMA "/sem\_escritura"

#define NAME\_MAX 50

typedef struct {

int previous\_id;

int id;

char name[NAME\_MAX];

} ClientInfo;

ClientInfo \*clienteinformacion;

void manejador\_SIGUSR1(int sig){

printf("recibida SIGUSR1\n");

printf("\tID: %d\n\tPrevious\_id: %d\n\tName: %s\n", clienteinformacion->id, clienteinformacion->previous\_id, clienteinformacion->name);

}

void manejador\_SIGINT(int sig){

printf("recibida SIGINT\n");

munmap(clienteinformacion, sizeof(\*clienteinformacion));

shm\_unlink(SHM\_NAME);

sem\_unlink(SEMA);

exit(EXIT\_FAILURE);

}

int main(int argc, char \*argv[]){

int n, i;

pid\_t pid;

int fd\_shm;

int error;

char nombreaux[NAME\_MAX];

struct sigaction act;

sem\_t \*sem\_lect = NULL;

if(argc != 2) {

printf("Introduzca el numero de procesos hijo\n");

exit(EXIT\_FAILURE);

}

n = atoi(argv[1]);

if((sem\_lect = sem\_open(SEMA, O\_CREAT | O\_EXCL, S\_IRUSR | S\_IWUSR, 1)) == SEM\_FAILED){

perror("sem\_open");

exit(EXIT\_FAILURE);

}

sigemptyset(&(act.sa\_mask));

act.sa\_handler = manejador\_SIGINT;

act.sa\_flags = 0;

if(sigaction(SIGINT, &act, NULL) < 0){

perror("Sigaction");

exit(EXIT\_FAILURE);

}

fd\_shm = shm\_open(SHM\_NAME, O\_RDWR | O\_CREAT | O\_EXCL, S\_IRUSR | S\_IWUSR);

if(fd\_shm == -1) {

fprintf (stderr, "Error creando el segmento de memoria compartida \n");

return EXIT\_FAILURE;

}

error = ftruncate(fd\_shm, sizeof(ClientInfo));

if(error == -1) {

fprintf (stderr, "Error redimensionando el segmento de memoria compartida \n");

shm\_unlink(SHM\_NAME);

return EXIT\_FAILURE;

}

clienteinformacion = (ClientInfo \*)mmap(NULL, sizeof(\*clienteinformacion), PROT\_READ | PROT\_WRITE, MAP\_SHARED, fd\_shm, 0);

if(clienteinformacion == MAP\_FAILED) {

fprintf (stderr, "Error mapeando el egmento de memoria compartida \n");

shm\_unlink(SHM\_NAME);

return EXIT\_FAILURE;

}

clienteinformacion->id = 0;

clienteinformacion->previous\_id = -1;

for(i = 0; i < n; ++i){

pid = fork();

if(pid < 0){

perror("fork");

exit(EXIT\_FAILURE);

}

else if(pid == 0){

while(1){

sem\_wait(sem\_lect);

/\*Si es el hijo generamos un numero aleatorio\*/

srand(pid);

sleep(rand()%10 +1);

clienteinformacion->previous\_id ++;

printf("Introduzca el nombre del nuevo usuario ( %d)\n", getpid());

scanf("%s",nombreaux);

strcpy(clienteinformacion->name, nombreaux);

clienteinformacion->id++;

/\*Mandamos la señal al padre\*/

kill(getppid(), SIGUSR1);

exit(EXIT\_SUCCESS);

}

}

else

}

}

if(pid > 0){

sigemptyset(&(act.sa\_mask));

act.sa\_handler = manejador\_SIGUSR1;

act.sa\_flags = 0;

if(sigaction(SIGUSR1, &act, NULL) < 0){

perror("Sigaction");

exit(EXIT\_FAILURE);

}

for(i = 0; i < n; ++i){

pause();

sem\_post(sem\_lect);

}

while(wait(NULL)>0);

/\*Liberamos todos los recursos\*/

munmap(clienteinformacion, sizeof(\*clienteinformacion));

shm\_unlink(SHM\_NAME);

sem\_close(sem\_lect);

sem\_unlink(SEMA);

exit(EXIT\_SUCCESS);

}

return EXIT\_SUCCESS;

}

Ejercicio 3

Para desarrollar este problema típico de productor-consumidor hemos empleado un sistema basado en semáforos, uno de los cuales, sem\_general, servía como puerta de acceso a los procesos para coger y/o sacar las letras de la zona de memoria compartida, y otros dos que sirven para indicar le número de letras y de espacios disponible en todo momento.

Código productor

#define ESPACIOS "/sem\_espacios"

#define LETRAS "/sem\_letras"

#define GENERAL "/sem\_general"

#define SHM\_NAME "/mem\_info"

#define MAXIM 10

#define NOMBRE\_ARCHIVO "fich.txt"

void manejador\_SIGINT(int sig){

printf("\nrecibida SIGINT\n");

shm\_unlink(SHM\_NAME);

sem\_unlink(LETRAS);

sem\_unlink(ESPACIOS);

sem\_unlink(GENERAL);

exit(EXIT\_FAILURE);

}

int main(){

sem\_t \*sem\_letras, \*sem\_espacios, \*sem\_general;

int fd\_shm, error;

Cola \*cad;

char auxic;

struct sigaction act;

sigemptyset(&(act.sa\_mask));

act.sa\_flags = 0;

act.sa\_handler = manejador\_SIGINT;

if (sigaction(SIGINT, &act, NULL) < 0) {

perror("sigaction");

exit(EXIT\_FAILURE);

}

if((sem\_letras = sem\_open(LETRAS, O\_CREAT | O\_EXCL, S\_IRUSR | S\_IWUSR, 0)) == SEM\_FAILED){

perror("sem\_open");

exit(EXIT\_FAILURE);

}

if((sem\_espacios = sem\_open(ESPACIOS, O\_CREAT | O\_EXCL, S\_IRUSR | S\_IWUSR, MAXIM)) == SEM\_FAILED){

perror("sem\_open");

sem\_close(sem\_letras);

sem\_unlink(LETRAS);

exit(EXIT\_FAILURE);

}

if((sem\_general = sem\_open(GENERAL, O\_CREAT | O\_EXCL, S\_IRUSR | S\_IWUSR, 1)) == SEM\_FAILED){

perror("sem\_open");

sem\_close(sem\_letras);

sem\_unlink(LETRAS);

sem\_close(sem\_espacios);

sem\_unlink(ESPACIOS);

exit(EXIT\_FAILURE);

}

fd\_shm = shm\_open(SHM\_NAME, O\_RDWR | O\_CREAT | O\_EXCL, S\_IRUSR | S\_IWUSR);

if(fd\_shm == -1) {

fprintf (stderr, "Error creando el segmento de memoria compartida \n");

sem\_close(sem\_letras);

sem\_unlink(LETRAS);

sem\_close(sem\_espacios);

sem\_unlink(ESPACIOS);

sem\_close(sem\_general);

sem\_unlink(GENERAL);

return EXIT\_FAILURE;

}

error = ftruncate(fd\_shm, sizeof(Cola));

if(error == -1) {

fprintf (stderr, "Error redimensionando el segmento de memoria compartida \n");

shm\_unlink(SHM\_NAME);

sem\_close(sem\_letras);

sem\_unlink(LETRAS);

sem\_close(sem\_espacios);

sem\_unlink(ESPACIOS);

sem\_close(sem\_general);

sem\_unlink(GENERAL);

return EXIT\_FAILURE;

}

cad = (Cola \*)mmap(NULL, sizeof(Cola), PROT\_READ | PROT\_WRITE, MAP\_SHARED, fd\_shm, 0);

if(cad == MAP\_FAILED){

fprintf (stderr, "Error mapeando el egmento de memoria compartida \n");

shm\_unlink(SHM\_NAME);

sem\_close(sem\_letras);

sem\_unlink(LETRAS);

sem\_close(sem\_espacios);

sem\_unlink(ESPACIOS);

sem\_close(sem\_general);

sem\_unlink(GENERAL);

return EXIT\_FAILURE;

}

scanf("%c", &auxic);

while(auxic != EOF){

sem\_wait(sem\_espacios);

sem\_wait(sem\_general);

if(insert(cad, auxic) == -1){

perror("insert");

shm\_unlink(SHM\_NAME);

munmap(cad, sizeof(Cola));

sem\_close(sem\_letras);

sem\_unlink(LETRAS);

sem\_close(sem\_espacios);

sem\_unlink(ESPACIOS);

sem\_close(sem\_general);

sem\_unlink(GENERAL);

return EXIT\_FAILURE;

}

sem\_post(sem\_general);

sem\_post(sem\_letras);

scanf("%c", &auxic);

}

sem\_wait(sem\_espacios);

sem\_wait(sem\_general);

if(insert(cad, '\0') == -1){

perror("insert");

shm\_unlink(SHM\_NAME);

munmap(cad, sizeof(Cola));

sem\_close(sem\_letras);

sem\_unlink(LETRAS);

sem\_close(sem\_espacios);

sem\_unlink(ESPACIOS);

sem\_close(sem\_general);

sem\_unlink(GENERAL);

return EXIT\_FAILURE;

}

else{

sem\_post(sem\_general);

sem\_post(sem\_letras);

shm\_unlink(SHM\_NAME);

munmap(cad, sizeof(Cola));

sem\_close(sem\_letras);

sem\_unlink(LETRAS);

sem\_close(sem\_espacios);

sem\_unlink(ESPACIOS);

sem\_close(sem\_general);

sem\_unlink(GENERAL);

return EXIT\_FAILURE;

}

}

Código consumidor

#define ESPACIOS "/sem\_espacios"

#define LETRAS "/sem\_letras"

#define GENERAL "/sem\_general"

#define SHM\_NAME "/mem\_info"

#define MAXIM 10

void manejador\_SIGINT(int sig){

printf("\nrecibida SIGINT\n");

exit(EXIT\_FAILURE);

}

int main(){

sem\_t \*sem\_letras, \*sem\_espacios, \*sem\_general;

int fd\_shm;

Cola \*cad;

char auxic;

struct sigaction act;

sigemptyset(&(act.sa\_mask));

act.sa\_flags = 0;

act.sa\_handler = manejador\_SIGINT;

if (sigaction(SIGINT, &act, NULL) < 0) {

perror("sigaction");

exit(EXIT\_FAILURE);

}

if((sem\_letras = sem\_open(LETRAS, S\_IWUSR | S\_IRUSR, 0)) == SEM\_FAILED){

perror("sem\_open");

exit(EXIT\_FAILURE);

}

if((sem\_espacios = sem\_open(ESPACIOS, S\_IWUSR | S\_IRUSR, MAXIM)) == SEM\_FAILED){

perror("sem\_open");

exit(EXIT\_FAILURE);

}

if((sem\_general = sem\_open(LETRAS, S\_IWUSR | S\_IRUSR, 1)) == SEM\_FAILED){

perror("sem\_open");

exit(EXIT\_FAILURE);

}

fd\_shm = shm\_open(SHM\_NAME, O\_RDWR, S\_IRUSR | S\_IWUSR);

if(fd\_shm == -1) {

fprintf (stderr, "Error creando el segmento de memoria compartida \n");

return EXIT\_FAILURE;

}

cad = (Cola \*)mmap(NULL, sizeof(Cola), PROT\_READ | PROT\_WRITE, MAP\_SHARED, fd\_shm, 0);

if(cad == MAP\_FAILED){

fprintf (stderr, "Error mapeando el egmento de memoria compartida \n");

return EXIT\_FAILURE;

}

while(1){

sem\_wait(sem\_letras);

sem\_wait(sem\_general);

if((auxic = delete(cad)) == -1){

perror("delete");

munmap(cad, sizeof(Cola));

exit(EXIT\_FAILURE);

}

else if(auxic != '\0'){

printf("%c", auxic);

sem\_post(sem\_general);

sem\_post(sem\_espacios);

}

else{

printf("\nEncontrado el final del fichero\n");

sem\_close(sem\_general);

sem\_close(sem\_letras);

sem\_close(sem\_espacios);

exit(EXIT\_SUCCESS);

}

}

}

Ejercicio 4

En este ejercicio tratamos el envío y la recepción de mensajes entre procesos, la única complejidad añadida que puede tener el programa es la de como evaluar cuando el programa debe finalizar, lo cual es simple pues mq\_receive devuelve el número de bytes recibidos, que son los mismos que envía el segundo proceso al tercero.

Código Proceso A

#define MAXIMO 2048

int main(int argc, char\* argv[]) {

mqd\_t queue;

struct mq\_attr attributes;

struct stat status;

char\* msg;

char aux = 'a';

int f, i;

attributes.mq\_flags = 0;

attributes.mq\_maxmsg = 10;

attributes.mq\_curmsgs = 0;

attributes.mq\_msgsize = 2048;

if(argc != 3){

printf("Introduzca los parametros correctamente\n" );

exit(EXIT\_FAILURE);

}

f = open(argv[1], O\_RDONLY, S\_IRUSR);

if(f < 0){

perror("open");

exit(EXIT\_FAILURE);

}

queue = mq\_open(argv[2],

O\_CREAT | O\_WRONLY, /\* This process is only going to send messages \*/

S\_IRUSR | S\_IWUSR, /\* The user can read and write \*/

&attributes);

if(queue == (mqd\_t)-1) {

perror("mq\_open");

return EXIT\_FAILURE;

}

if(fstat(f, &status)<0) {

perror("fstat");

return EXIT\_FAILURE;

}

msg = (char\*)mmap(NULL, status.st\_size, PROT\_READ, MAP\_PRIVATE, f, 0);

if(msg == MAP\_FAILED){

perror("mmap");

return EXIT\_FAILURE;

}

for(i = 0 ; i < status.st\_size; i =+ MAXIMO){

if(status.st\_size < MAXIMO){

if(mq\_send(queue, (char \*)msg, status.st\_size, 1) == -1) {

perror("mq\_send");

return EXIT\_FAILURE;

}

}

if(mq\_send(queue, (char \*)msg, MAXIMO, 1) == -1) {

perror("mq\_send");

return EXIT\_FAILURE;

}

msg = msg + 2048;

}

if(status.st\_size%2048 != 0) {

if(mq\_send(queue, (char\*)msg, status.st\_size%MAXIMO, 1) == -1) {

perror("mq\_send");

return EXIT\_FAILURE;

}

}

else{

if(mq\_send(queue,(char\*)&aux, 1, 1) == -1) {

perror("mq\_send");

return EXIT\_FAILURE;

}

}

munmap(msg, status.st\_size);

mq\_close(queue);

return EXIT\_SUCCESS;

}

Código Proceso B

#define MAXIMO 2048

int main(int argc, char\* argv[]) {

mqd\_t queue, queue2;

struct mq\_attr attributes, attributes2;

char \*msg;

int i, aux = MAXIMO;

attributes.mq\_flags = 0;

attributes.mq\_maxmsg = 10;

attributes.mq\_curmsgs = 0;

attributes.mq\_msgsize = 2048;

attrbutes2.mq\_flags = 0;

attributes2.mq\_maxmsg = 10;

attributes2.mq\_curmsgs = 0;

attributes2.mq\_msgsize = 2048;

msg = (char\*)malloc(sizeof(char)\*MAXIMO);

if(argc != 3){

printf("Introduzca los parametros correctamente\n" );

exit(EXIT\_FAILURE);

}

queue = mq\_open(argv[1], O\_RDONLY,

S\_IRUSR | S\_IWUSR,

&attributes);

if(queue == (mqd\_t)-1) {

perror("mq\_open");

return EXIT\_FAILURE;

}

queue2 = mq\_open(argv[2],

O\_CREAT | O\_WRONLY,

S\_IRUSR | S\_IWUSR,

&attributes2);

if(queue2 == (mqd\_t)-1) {

perror("mq\_open");

return EXIT\_FAILURE;

}

while(aux == MAXIMO)

aux = mq\_receive(queue, msg, sizeof(char)\*MAXIMO, NULL);

if (aux == -1) {

perror("mq\_receive");

return EXIT\_FAILURE;

}

for (i = 0; i < aux; i++){

if(msg[i] < 'z' && msg[i] >= 'a'){

msg[i]++;

}

else if(msg[i] == 'z'){

msg[i] = 'a';

}

}

if(mq\_send(queue2, msg, aux, 1) == -1) { //parará cuando la cola esté vacia

fprintf (stderr, "Error sending message\n");

return EXIT\_FAILURE;

}

}

mq\_close(queue);

mq\_unlink(argv[1]);

mq\_close(queue2);

return EXIT\_SUCCESS;

}

Código Proceso C

#define MAXIMO 2048

int main(int argc, char\* argv[]) {

mqd\_t queue;

struct mq\_attr attributes;

char\* msg;

int aux = MAXIMO;

attributes.mq\_flags = 0;

attributes.mq\_maxmsg = 10;

attributes.mq\_curmsgs = 0;

attributes.mq\_msgsize = 2048;

msg = (char\*)malloc(sizeof(char)\*2048);

if(argc != 2){

printf("Introduzca los parametros correctamente\n" );

exit(EXIT\_FAILURE);

}

queue = mq\_open(argv[1],

O\_RDONLY,

S\_IRUSR | S\_IWUSR,

&attributes);

if(queue == (mqd\_t)-1) {

perror("mq\_open");

return EXIT\_FAILURE;

}

while(aux == MAXIMO){

aux = mq\_receive(queue, msg, sizeof(char)\*MAXIMO, NULL);

if (aux == -1) {

perror("mq\_receive");

return EXIT\_FAILURE;

}

printf("%s", msg);

}

mq\_close(queue);

mq\_unlink(argv[1]);

return EXIT\_SUCCESS;

}

Código Main

#define COLA1 "/cola1"

#define COLA2 "/cola2"

int main(){

pid\_t pid1, pid2, pid3;

pid1 = fork();

if (pid1 < 0){

perror("fork");

exit(EXIT\_FAILURE);

}

if (pid1 == 0){

if (execl("./ejercicio4\_A", "./ejercicio4\_A", "file.txt", COLA1, (char\*)NULL) == -1) {

perror("execl");

return EXIT\_FAILURE;

}

}

pid2 = fork();

if (pid2 < 0){

perror("fork");

exit(EXIT\_FAILURE);

}

if (pid2 == 0){

if (execl("./ejercicio4\_B", "./ejercicio4\_B", COLA1, COLA2, (char\*)NULL) == -1) {

perror("execl");

return EXIT\_FAILURE;

}

}

pid3 = fork();

if (pid3 < 0){

perror("fork");

exit(EXIT\_FAILURE);

}

if (pid3 == 0){

if (execl("./ejercicio4\_C", "./ejercicio4\_C", COLA2, (char\*)NULL) == -1) {

perror("execl");

return EXIT\_FAILURE;

}

}

while(wait(NULL)>0);

exit(EXIT\_SUCCESS);

}