Memoria Práctica 3 - Sistemas Operativos

Manuel Cintado y Manuel Suárez Román

Doble Grado Matemáticas e Ingeniería Informática

Pareja 09

Contenido

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);
}
```