Documentación Proyecto 2

- Piersigilli Joaquin LU: 105285
- Utizi Sebastián LU: 105025

Ejercicio 2.2b [Proyecto 1] Fabrica DCIC

Compilación: gcc -o fabrica fabrica_Reentrega.c -lpthread
Ejecución: ./fabrica

Estrategia de resolución: Para este ejercicio se utilizó la metodología dividir y conquistar por lo que se pensó por separado el metodo de resolucion relacionado a los trabajadores que van a utilizar la máquina y a los trabajadores que van a utilizar el baño.
Una vez inicializados los 8 hilos representando a los trabajadores se los envía de a uno a ver qué hacer y, una vez que ese trabajador sabe lo que debe hacer, avisa que ya pueden asignarle alguna tarea a otro.

Para los trabajadores que llegan a trabajar a la máquina se tiene un semáforo “espera” que controla si deben quedarse esperando debido a que la máquina está apagada y se requieren tres trabajadores para encenderla, o si deben pasar directamente a trabajar ya que la máquina está encendida. En el caso que la máquina esté apagada, los dos primeros trabajadores deben esperar a que llegue el tercero, por lo que se bloquean en otro semáforo “tercero” y una vez que éste llega, enciende la máquina y libera a los dos trabajadores que están esperando. Los tres pasan a trabajar un tiempo random hasta que dejan de trabajar. Mientras la máquina esté encendida cualquier trabajador que llegue pasará directamente a trabajar, esto se controla con (sem_trywait(&maquinaEncendida)==-1).
Como a partir de este momento los trabajadores pueden dejar de trabajar en cualquier momento y ésto puede darse simultáneamente, se entra en una sección crítica (ya que si por ejemplo los últimos dos trabajadores dejan de trabajar al mismo tiempo, puede suceder que ambos dejen de trabajar sin apagar la máquina) donde se controla la cantidad de trabajadores que van a quedar en la máquina y si no va a quedar ninguno, éste último la apaga antes de retirarse.

Para los trabajadores que desean utilizar el baño lo primero a verificar es si se trata de una mujer o un hombre, en cualquier caso el código es análogo pero se utilizan distintos semáforos para la sincronización.
La estrategia para una mujer es la siguiente: (Solo se explica para la mujer porque el hombre es igual)
Cuando una mujer va a utilizar el baño, primero se fija si ya hay 2 mujeres y en ese caso se queda esperando. En caso de que no se quede esperando, se debe comprobar si hay alguna otra mujer adentro y en caso de que no haya ninguna, hay que comprobar que no haya hombres, para esto se intenta obtener un mutex “banio” que lo adquiere la primera mujer o el primer hombre que entra a un baño y no se libera hasta que se va la última persona de ese sexo del baño. Entonces, esa mujer es la primera en entrar y obtiene el mutex, si luego llegan más mujeres antes de que ésta se vaya, pasan directamente (siempre respetando el máximo de 2 mujeres a la vez). Cada vez que una mujer se retira del baño, comprueba si es la última y si lo es, libera el mutex, por lo que si había hombres bloqueados en el mutex esperando para entrar al baño, lo adquieren y pueden pasar.

Código:

```
/**,
        Ejercicio 2.2b [Proyecto 1] Fabrica DCIC
        Joaquin Piersigilli, Utizi Sebastian
**/

#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <pthread.h>
#include <semaphore.h>
#include <time.h>

#define CANTP 8

sem_t semPersonas, espera, maquinaEncendida, trabajando, tercero; //semaforos relacionados a la maquina
pthread_mutex_t seccionCritica, banio;
sem_t maxHombres, mutexHombre, cantActualHombres, maxMujeres, mutexMujer, cantActualMujeres; //semaforos relacionados la baño

unsigned long long rdtsc(){
        unsigned int lo,hi;
        __asm__ __volatile__ ("rdtsc" : "=a" (lo), "=d" (hi));
        return ((unsigned long long)hi << 32) | lo;
}

void trabajar(int id_persona){
   printf("%d Me toca trabajar: \n",id_persona);
   fflush(stdout);

   if( 0 == sem_trywait(&espera)){
                printf("maquina esta apagada y el trabajador %d espera que sean 3\n",id_persona);
         fflush(stdout);
          sem_post(&semPersonas); //antes de bloquearse esperando que lleguen mas personas aviso que pueden mandar a hacer tareas a mas personas.
                sem_wait(&tercero);
   }else{
                        if(sem_trywait(&maquinaEncendida)==-1){ //cuando se va a encender la maquina sé que hay exactamente 3 personas trabajando, entonces aumento el semaforo trabajando a 2 (como inicializandolo en el momento que arranca la maquina) y luego lo continuo aumentando el mismo cada vez que se suma alguien a trabajar, de manera que el
```

semaforo siempre va a tener 1 persona menos de las que realmente hay, luego, cuando las personas dejan de trabajar decremento el semaforo con un trywait, por lo que al irse la anteultima persona el semaforo pasara a valer 0 y el ultimo producira un error en el trywait y así sé que debo apagar la maquina.

```
                printf("Con %d son 3 y se enciende la maquina\n",id_persona);
                fflush(stdout);
                                sem_post(&maquinaEncendida);
                        sem_post(&tercero);
                        sem_post(&tercero);
                sem_post(&trabajando);
                sem_post(&trabajando);
        }else{
                sem_post(&maquinaEncendida); //para incrementar de nuevo el semaforo que se decremento en el trywait del segundo if
                 sem_post(&trabajando); //a partir del 4to trabajador viene aca y suman de a 1 normalmente
                }

        sem_post(&semPersonas); //antes de ponerme a trabajar aviso q pueden mandar mas gente a hacer tareas
        }

   //trabaja un tiempo random"
   int rnd = rdtsc() % 7000;
   printf("%d va a trabajar %i milisegs\n",id_persona,rnd);
   fflush(stdout);
   usleep(rnd);


    //va a dejar de trabajar una persona, se fija si quedan 0 para saber si hay que apagar la maquina
   pthread_mutex_lock(&seccionCritica);
   if(-1 == sem_trywait(&trabajando)){
        printf("deja de trabajar %d\n",id_persona);
        fflush(stdout);
        printf("Como no queda mas gente trabajando se apaga la maquina\n");
        fflush(stdout);
        sem_wait(&maquinaEncendida);
                sem_post(&espera);
                sem_post(&espera);
   }else{
        printf("deja de trabajar %d\n",id_persona);
        fflush(stdout);
   }
```

```
   pthread_mutex_unlock(&seccionCritica);
}


char que_soy(int id_persona){
        //Asumimos que si el id de la persona es par entonces es mujer
        if (id_persona %2 == 0)
        return 'M';
        else //Si es impar entonces es hombre
        return 'H';
}


void uso_gabinete(int p){
   printf("%d Me toca ir al baño: \n",p);
   sem_post(&semPersonas);
   char sex = que_soy(p);
   if(sex=='M') //una mujer tiene que usar el baño
   {
         sem_wait(&maxMujeres); //si hay 2 mujeres adentro del baño esta se queda esperando
         sem_wait(&mutexMujer); //si no habia 2 mujeres, pide el mutex para entrar a seccion critica

        //si no habia ninguna mujer antes (o sea cantActualMujeres estaba en 0) se tiene q fijar q no haya hombres, en cambio
        //si ya habia 1 mujer, la actual simplemente entra al baño
        if (sem_trywait(&cantActualMujeres) == -1)
        {
                sem_post(&cantActualMujeres); //aumento la cantidad de mujeres
                pthread_mutex_lock(&banio);    //intento obtener el mutex banio, si no puedo quiere decir que lo tiene un hombre
        }else{ //realizo dos post debido a que si habia una mujer, con el trywait el semaforo que indica la cantidad de mujeres se decremento a 0
                sem_post(&cantActualMujeres);
                sem_post(&cantActualMujeres);
        }
         sem_post(&mutexMujer); //libero el mutex de la seccion critica, puede entrar otra mujer

        // uso el baño
        printf("Mujer %i usa el baño \n",p);
        fflush(stdout);
        sleep(3);
```

```
        sem_wait(&mutexMujer);              //seccion critica si mas de una mujer termina de usar el baño
        sem_wait(&cantActualMujeres);       //decremento la cantidad de mujeres
        printf("Mujer %i deja el baño \n",p);
        if (sem_trywait(&cantActualMujeres) == -1) //si no quedan mas mujeres, libero el banio para que puedan entrar hombres en caso de que alguno este bloqueado en el lock mutex banio.
        {
                pthread_mutex_unlock(&banio);
        }else{
                sem_post(&cantActualMujeres);    //si todavia quedan mujeres, incremento el semaforo por el decremento que se produjo al realizar el trywait
        }
        sem_post(&mutexMujer);      //libera mutex de seccion critica para que se vaya otra mujer si es necesario
        sem_post(&maxMujeres);      //se va una mujer, se aumenta la cantidad maxima de mujeres que puede haber

    }
    else //un hombre tiene que usar el baño
    {
        sem_wait(&maxHombres); //si hay 2 hombres adentro del baño este se queda esperando
        sem_wait(&mutexHombre); //si no habia 2 hombres, pide el mutex para entrar a seccion critica

        //si no habia ningun hombre antes (o sea cantActualHombres estaba en 0) se tiene q fijar q no haya mujeres, en cambio
        //si ya habia 1 hombre simplemente entra al baño
        if (sem_trywait(&cantActualHombres) == -1)
        {
                sem_post(&cantActualHombres);
                pthread_mutex_lock(&banio);
        }else{
                sem_post(&cantActualHombres);
                sem_post(&cantActualHombres);
        }
        sem_post(&mutexHombre); //libero el mutex de la seccion critica


        // uso el baño
        printf("Hombre %i usa el baño \n",p);
        fflush(stdout);
```

```
        sleep(3);

        sem_wait(&mutexHombre);
        sem_wait(&cantActualHombres);
        printf("Hombre %i deja el baño \n",p);
        fflush(stdout);

        if (sem_trywait(&cantActualHombres) == -1) //si habia 1 solo hombre (el q se esta yendo), avisa que pueden pasar ya sean hombres o mujeres
        {
                pthread_mutex_unlock(&banio);
        }else{
                sem_post(&cantActualHombres); //por el que decremento el trywait
        }
        sem_post(&mutexHombre);
        sem_post(&maxHombres);


   }
}

void descansar(int p){
        printf("%d Me toca descansar: \n",p);
        sem_post(&semPersonas);
        sleep(5);
}

void que_hacer(int p){

   int rnd = rdtsc() % 89;
   //printf("Random es: %i\n",rnd);
        if(rnd>=0 && rnd<=30){
                trabajar(p);
        }
        else{    if(rnd>=31 && rnd<=60){
                        descansar(p);
        }
                        else{
                       uso_gabinete(p);
        }
        }

}
```

```
void *persona(void *p){

   int id_persona= *(int*)p;
   while(1){
         sem_wait(&semPersonas);
               que_hacer(id_persona);
         sleep(3);
   }

}

int main()
{

         sem_init(&semPersonas,0,1);
         sem_init(&espera,0,2);
         sem_init(&maquinaEncendida,0,0);
         sem_init(&tercero,0,0);
         sem_init(&trabajando,0,0);

         sem_init(&maxMujeres,0,2);
         sem_init(&maxHombres,0,2);
         sem_init(&mutexMujer,0,1);
         sem_init(&mutexHombre,0,1);
         sem_init(&cantActualHombres,0,0);
         sem_init(&cantActualMujeres,0,0);


         pthread_mutex_init(&banio,0);
         pthread_mutex_init(&seccionCritica,0);

         int personas[CANTP];
         pthread_t personas_t[CANTP];
         int i;
         for (i=0; i<CANTP; i++)
         personas[i] = i;

         for (i=0; i<CANTP; i++)
         pthread_create(&personas_t[i], NULL, persona, (void *)&personas[i]);

         //Espero a que todos finalicen
         for (i=0; i<CANTP; i++)
         pthread_join(personas_t[i],NULL);
}
```

Ejercicios del Proyecto 2

Ejercicio 1.1.i.a)

Compilación: gcc -o 11ai 11ai.c
Ejecución: ./11ai
Aclaración: Se debe cortar la execución externamente.

Estrategia de resolución:

Este ejercicio se resolvió utilizando Pipes para la sincronización.
Se utilizan 4 pipes:
pipeA Se usa para escribir la letra A y habilitar la escritura de la letra B
pipeB Se usa para escribir la letra B y habilitar la escritura de la letra C o la D
pipeCoD Se usa para escribir la letra C o la D y habilitar la escritura de la letra E
pipeE Se usa para escribir la letra E y habilitar la escritura de la letra A

Se utilizan 5 procesos, cada uno correspondiente a una de las letras a escribir.
Al iniciarse los procesos se bloquean esperando que haya un mensaje para leer del pipe correspondiente a la letra que deben escribir, excepto el proceso A ya que al iniciarse, el pipeA ya tiene un mensaje y el proceso de A puede consumirlo al instante, imprimiendo la letra A y enviando un mensaje al pipeB, estando el proceso encargado de imprimir la B bloqueado esperando que éste llegue. Cuando pipeB recibe el mensaje, el proceso B puede continuar su ejecución, imprimiendo la letra B y enviando un mensaje al pipeCoD. Tanto el proceso encargado de imprimir la C como el encargado de imprimir la D están bloqueados esperando que llegue un mensaje al pipeCoD, por lo tanto cuando este llega, cualquiera de los 2 procesos puede continuar su ejecución y en cualquier caso se envía el mensaje al pipeE por lo que el último proceso puede imprimir la E y enviar el mensaje al pipeA para que se repita el patrón.

Código:

```
/**,
        Ejercicio 1.1.a.i
        Joaquin Piersigilli, Utizi Sebastian
**/


#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <sys/wait.h>

int main()
{

        //Declaro los pipes
        int pipeA[2]; //Este pipe se usa para escribir la letra A y habilitar la escritura de la letra B
        int pipeB[2]; //Este pipe se usa para escribir la letra B y habilitar la escritura de la letra C o la D
        int pipeCoD[2]; //Este pipe se usa para escribir la letra C o la D y habilitar la escritura de la letra E
        int pipeE[2]; //Este pipe se usa para escribir la letra E y habilitar la escritura de la letra A

        pid_t   childpid;
        char mensaje;

        //Creo los pipes
        pipe(pipeA);
        pipe(pipeB);
        pipe(pipeCoD);
        pipe(pipeE);

        //Utilizo el lado de escritura del pipeA para poder comenzar con el primer proceso hijo sin que se bloquee el read
        //y luego de una vuelta de la secuencia ABCoDE continue por ese proceso
        write(pipeA[1],"N", 1);

        // Estoy en el proceso padre y creo el
        // Proceso A
        if((childpid = fork()) == -1){  //Hay error en la creacion del proceso hijo
                perror("fork");
                exit(1);
```

```
}

if(childpid == 0){
//Si el fork devolvió 0 estoy en el proceso hijo

        //close(pipeA[0]); Dejo abierto el lado de lectura del pipeA
        close(pipeA[1]);
        close(pipeB[0]);
        //close(pipeB[1]); Dejo abierto el lado de escritura
        close(pipeCoD[0]);
        close(pipeCoD[1]);
        close(pipeE[0]);
        close(pipeE[1]);

        while(1){
        read(pipeA[0], &mensaje, sizeof(char));
        printf("A");
        fflush(stdout);
        write(pipeB[1],"N", 1);
  //  sleep(1);
        }
        exit(0);
}

//Estoy en el proceso padre y creo el
//Proceso B

if((childpid = fork()) == -1){
        perror("fork");
        exit(1);
}

if(childpid == 0){
        //Si el fork devolvió 0 estoy en el proceso hijo

        close(pipeA[0]);
        close(pipeA[1]);
        //close(pipeB[0]); Dejo abierto el lado de lectura del pipeB
        close(pipeB[1]);
        close(pipeCoD[0]);
// close(pipeCoD[1]); Dejo abierto el lado de escritura del pipeCoD
        close(pipeE[0]);
        close(pipeE[1]);
```

```
        while(1){
        read(pipeB[0], &mensaje, sizeof(char));
        printf("B");
        fflush(stdout);
        write(pipeCoD[1],"N", 1);
         //sleep(1);
        }
        exit(0);
}

//Estoy en el proceso padre y creo el
// Proceso C

if((childpid = fork()) == -1){
        perror("fork");
        exit(1);
}

if(childpid == 0){
        //Si el fork devolvió 0 estoy en el proceso hijo

        close(pipeA[0]);
        close(pipeA[1]);
        close(pipeB[0]);
        close(pipeB[1]);
//close(pipeCoD[0]); Dejo abierto el lado de lectura del pipeCoD
        close(pipeCoD[1]);
        close(pipeE[0]);
//close(pipeE[1]); Dejo abierto el lado de escritura del pipeE

        while(1){
        read(pipeCoD[0], &mensaje, sizeof(char));
        printf("C");
        fflush(stdout);
  write(pipeE[1],"N", 1);
 //sleep(1);
        }
        exit(0);
}

//Estoy en el proceso padre y creo el
// Proceso D

if((childpid = fork()) == -1){
```

```
        perror("fork");
        exit(1);
}

if(childpid == 0){
        //Si el fork devolvió 0 estoy en el proceso hijo

        close(pipeA[0]);
        close(pipeA[1]);
        close(pipeB[0]);
        close(pipeB[1]);
        //close(pipeCoD[0]); Dejo abierto el lado de lectura del pipeCoD
        close(pipeCoD[1]);
        close(pipeE[0]);
        //close(pipeE[1]); Dejo abierto el lado de escritura del pipeE

        while(1){
        read(pipeCoD[0], &mensaje, sizeof(char));
        printf("D");
        fflush(stdout);
        write(pipeE[1],"N", 1);
 //    sleep(1);
        }
        exit(0);
}


//Estoy en el proceso padre y creo el
// Proceso E

if((childpid = fork()) == -1){
        perror("fork");
        exit(1);
}

if(childpid == 0){
        //Si el fork devolvió 0 estoy en el proceso hijo

        close(pipeA[0]);
        //close(pipeA[1]); Dejo abierto el lado de escritura del pipeA
        close(pipeB[0]);
        close(pipeB[1]);
        close(pipeCoD[0]);
        close(pipeCoD[1]);
```

```
            //close(pipeE[0]); Dejo abierto el lado de lectura del pipeE
            close(pipeE[1]);

            while(1){
            read(pipeE[0], &mensaje, sizeof(char));
            printf("E ");
            fflush(stdout);
            write(pipeA[1],"N", 1);
             // sleep(1);

            }
            exit(0);
        }
```

//Estoy en el proceso padre luego de haber creado todos los procesos hijos encargados de imprimir cada letra sincronizandose con pipes.

```
        int i;
        for ( i=0; i<6; i++)  //  Espera por la finalización de todos sus hijos.
        wait(NULL);

        exit (1);
}
```

Ejercicio 1.1.ii.a)

Compilación: gcc -o 11aii 11aii.c
Ejecución: ./11aii
Aclaración: Se debe cortar la execución externamente.

Estrategia de resolución:

Este ejercicio se resolvió utilizando Pipes para la sincronización.
Se utilizan 5 procesos, cada uno correspondiente a una de las letras a escribir, los llamaremos “proceso A/B/C/D/E”
Se utilizan 13 pipes debido a que algunos se utilizan para tener la referencia de desde qué impresión de la letra se viene, ya que a veces, después de la D puede venir una E o una BB e incluso depende si es la primera impresión de la E o la segunda, se debe seguir un camino u otro, por lo que no alcanza con saber que se imprimió la letra D sino que hay que saber si es su primera aparición en la secuencia o la segunda o la tercera.
Se podrían haber utilizado menos pipes enviando un mensaje en el pipe indicando en qué aparicion de la letra se encuentra, pero consideré que eso no es lo que pide el ejercicio sino realizar la sincronización entera utilizando pipes.

Código:

```
/**,
        Ejercicio 1.1.a.ii
        Joaquin Piersigilli, Utizi Sebastian
**/


#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <sys/wait.h>

//ABCDEAABCDBBCDEABCDE...

//ABCDE AABCD BBCDE

int main()
{

        //Declaro los pipes
        int pipeA[2];
   int pipeAA[2];

        int pipeB1[2];
```

```
int pipeB2[2];
int pipeBB[2];

        int pipeC1[2];
int pipeC2[2];
int pipeC3[2];

int pipeD1[2];
int pipeD2[2];
int pipeD3[2];

        int pipeE1[2];
int pipeE2[2];

        pid_t   childpid;
        char mensaje;

        //Creo los pipes
        pipe(pipeA);
pipe(pipeAA);
        pipe(pipeB1);
        pipe(pipeB2);
pipe(pipeBB);
        pipe(pipeC1);
pipe(pipeC2);
pipe(pipeC3);
pipe(pipeD1);
pipe(pipeD2);
pipe(pipeD3);
pipe(pipeE1);
pipe(pipeE2);

        //Utilizo el lado de escritura del pipeA para poder comenzar con el primer proceso hijo sin que se bloquee el read
        //y luego de una vuelta de la secuencia ABCoDE continue por ese proceso
        write(pipeA[1],"N", 1);

        // Estoy en el proceso padre y creo el
        // Proceso A
        if((childpid = fork()) == -1){  //Hay error en la creacion del proceso hijo
                perror("fork");
                exit(1);
        }
```

```
if(childpid == 0){
//Si el fork devolvió 0 estoy en el proceso hijo

        //close(pipeA[0]);
 close(pipeA[1]);
 //close(pipeAA[0]);
 close(pipeAA[1]);

 close(pipeB1[0]);
 //close(pipeB1[1]);
 close(pipeB2[0]);
 //close(pipeB2[1]);
 close(pipeBB[0]);
 close(pipeBB[1]);

 close(pipeC1[0]);
 close(pipeC1[1]);
 close(pipeC2[0]);
 close(pipeC2[1]);
 close(pipeC3[0]);
 close(pipeC3[1]);


 close(pipeD1[0]);
 close(pipeD1[1]);
 close(pipeD2[0]);
 close(pipeD2[1]);
 close(pipeD3[0]);
 close(pipeD3[1]);


 close(pipeE1[0]);
 close(pipeE1[1]);
 close(pipeE2[0]);
 close(pipeE2[1]);



        while(1){
        read(pipeA[0], &mensaje, sizeof(char));
        printf("A");
        fflush(stdout);
        write(pipeB1[1],"N", 1);
```

```
        read(pipeAA[0], &mensaje, sizeof(char));
        printf("AA");
        fflush(stdout);
        write(pipeB2[1],"N", 1);
        }
        exit(0);
}

//Estoy en el proceso padre y creo el
//Proceso B

if((childpid = fork()) == -1){
        perror("fork");
        exit(1);
}

if(childpid == 0){
        //Si el fork devolvió 0 estoy en el proceso hijo

        close(pipeA[0]);
 close(pipeA[1]);
 close(pipeAA[0]);
 close(pipeAA[1]);

 //close(pipeB1[0]);
 close(pipeB1[1]);
 //close(pipeB2[0]);
 close(pipeB2[1]);
 //close(pipeBB[0]);
 close(pipeBB[1]);

 close(pipeC1[0]);
 //close(pipeC1[1]);
 close(pipeC2[0]);
 //close(pipeC2[1]);
 close(pipeC3[0]);
 //close(pipeC3[1]);


 close(pipeD1[0]);
 close(pipeD1[1]);
 close(pipeD2[0]);
 close(pipeD2[1]);
 close(pipeD3[0]);
```

```
close(pipeD3[1]);


close(pipeE1[0]);
close(pipeE1[1]);
close(pipeE2[0]);
close(pipeE2[1]);

        while(1){
        read(pipeB1[0], &mensaje, sizeof(char));
        printf("B");
        fflush(stdout);
        write(pipeC1[1],"N", 1);

        read(pipeB2[0], &mensaje, sizeof(char));
        printf("B");
        fflush(stdout);
        write(pipeC2[1],"N", 1);

        read(pipeBB[0], &mensaje, sizeof(char));
        printf("BB");
        fflush(stdout);
        write(pipeC3[1],"N", 1);
        }
        exit(0);
}

//Estoy en el proceso padre y creo el
// Proceso C

if((childpid = fork()) == -1){
        perror("fork");
        exit(1);
}

if(childpid == 0){
        //Si el fork devolvió 0 estoy en el proceso hijo

        close(pipeA[0]);
 close(pipeA[1]);
 close(pipeAA[0]);
 close(pipeAA[1]);

 close(pipeB1[0]);
```

```
close(pipeB1[1]);
close(pipeB2[0]);
close(pipeB2[1]);
close(pipeBB[0]);
close(pipeBB[1]);

//close(pipeC1[0]);
close(pipeC1[1]);
//close(pipeC2[0]);
close(pipeC2[1]);
//close(pipeC3[0]);
close(pipeC3[1]);


close(pipeD1[0]);
//close(pipeD1[1]);
close(pipeD2[0]);
//close(pipeD2[1]);
close(pipeD3[0]);
//close(pipeD3[1]);


close(pipeE1[0]);
close(pipeE1[1]);
close(pipeE2[0]);
close(pipeE2[1]);

        while(1){
        read(pipeC1[0], &mensaje, sizeof(char));
        printf("C");
        fflush(stdout);
        write(pipeD1[1],"N", 1);

        read(pipeC2[0], &mensaje, sizeof(char));
        printf("C");
        fflush(stdout);
        write(pipeD2[1],"N", 1);

        read(pipeC3[0], &mensaje, sizeof(char));
        printf("C");
        fflush(stdout);
        write(pipeD3[1],"N", 1);

        }
```

```
        exit(0);
}

//Estoy en el proceso padre y creo el
// Proceso D

if((childpid = fork()) == -1){
        perror("fork");
        exit(1);
}

if(childpid == 0){
        //Si el fork devolvió 0 estoy en el proceso hijo

         close(pipeA[0]);
 close(pipeA[1]);
 close(pipeAA[0]);
 close(pipeAA[1]);

 close(pipeB1[0]);
 close(pipeB1[1]);
 close(pipeB2[0]);
 close(pipeB2[1]);
 close(pipeBB[0]);
 //close(pipeBB[1]);

 close(pipeC1[0]);
 close(pipeC1[1]);
 close(pipeC2[0]);
 close(pipeC2[1]);
 close(pipeC3[0]);
 close(pipeC3[1]);


 //close(pipeD1[0]);
 close(pipeD1[1]);
 //close(pipeD2[0]);
 close(pipeD2[1]);
 //close(pipeD3[0]);
 close(pipeD3[1]);


 close(pipeE1[0]);
 //close(pipeE1[1]);
```

```
 close(pipeE2[0]);
 //close(pipeE2[1]);

        while(1){
        read(pipeD1[0], &mensaje, sizeof(char));
        printf("D");
        fflush(stdout);
        write(pipeE1[1],"N", 1);

        read(pipeD2[0], &mensaje, sizeof(char));
        printf("D ");
        fflush(stdout);
        write(pipeBB[1],"N", 1);

        read(pipeD3[0], &mensaje, sizeof(char));
        printf("D");
        fflush(stdout);
        write(pipeE2[1],"N", 1);
        }
        exit(0);
}


//Estoy en el proceso padre y creo el
// Proceso E

if((childpid = fork()) == -1){
        perror("fork");
        exit(1);
}

if(childpid == 0){
        //Si el fork devolvió 0 estoy en el proceso hijo

        close(pipeA[0]);
 //close(pipeA[1]);
 close(pipeAA[0]);
 //close(pipeAA[1]);

 close(pipeB1[0]);
 close(pipeB1[1]);
 close(pipeB2[0]);
 close(pipeB2[1]);
 close(pipeBB[0]);
```

```
close(pipeBB[1]);

close(pipeC1[0]);
close(pipeC1[1]);
close(pipeC2[0]);
close(pipeC2[1]);
close(pipeC3[0]);
close(pipeC3[1]);


close(pipeD1[0]);
close(pipeD1[1]);
close(pipeD2[0]);
close(pipeD2[1]);
close(pipeD3[0]);
close(pipeD3[1]);


//close(pipeE1[0]);
close(pipeE1[1]);
//close(pipeE2[0]);
close(pipeE2[1]);

        while(1){
        read(pipeE1[0], &mensaje, sizeof(char));
        printf("E ");
        fflush(stdout);
        write(pipeAA[1],"N", 1);

        read(pipeE2[0], &mensaje, sizeof(char));
        printf("E ");
        fflush(stdout);
        write(pipeA[1],"N", 1);
        }
        exit(0);
}
```

//Estoy en el proceso padre luego de haber creado todos los procesos hijos encargados de imprimir cada letra sincronizandose con pipes.

```
int i;
for ( i=0; i<6; i++)  //  Espera por la finalización de todos sus hijos.
wait(NULL);
exit (1);
}
```

Ejercicio 1.1.b)

Compilación: gcc -o 11bii 11bii.c
Ejecución: ./11bii
Aclaración: Se debe cortar la execución externamente.

Estrategia de resolución:

Análogo al ejercicio anterior sólo que se utilizan colas de mensajes en vez de Pipes, por cada pipe se crea una cola y el algoritmo funciona exactamente igual.

Codigo:

```
/**,
        Ejercicio 1.1.b.ii
        Joaquin Piersigilli, Utizi Sebastian
**/

#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <sys/ipc.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <sys/msg.h>


#define A 1
#define AA 2
#define B1 3
#define B2 4
#define BB 5
#define C1 6
#define C2 7
#define C3 8
#define D1 9
#define D2 10
#define D3 11
#define E1 12
#define E2 13


typedef struct my_msg{  // Estrucutra del mensaje.
   long type;
}msg;
```

```
//ABCDE

//AABCD

//BBCDE
int main()
{

    pid_t childpid;  // ID de procesos.
    int q_id;
    int msg_sz;
    msg buffer;

    q_id = msgget(IPC_PRIVATE, IPC_CREAT | 0666);  //  Se crea la cola de mensajes.
    msg_sz = sizeof(struct my_msg);


    buffer.type=A;
      msgsnd(q_id, &buffer, msg_sz, 0);


    // Estoy en el proceso padre y creo el
          // Proceso A
          if((childpid = fork()) == -1){  //Hay error en la creacion del proceso hijo
                    perror("fork");
                    exit(1);
          }

          if(childpid == 0){
          //Si el fork devolvió 0 estoy en el proceso hijo

           while(1){

                              msgrcv(q_id, &buffer, msg_sz, A, 0);
                              printf("A");
                              fflush(stdout);
                              buffer.type=B1;
                              msgsnd(q_id, &buffer, msg_sz, 0);

                    msgrcv(q_id, &buffer, msg_sz, AA, 0);
                              printf("AA");
                              fflush(stdout);
```

```
                        buffer.type=B2;
                        msgsnd(q_id, &buffer, msg_sz, 0);
        }
        exit(0);
}


//Estoy en el proceso padre y creo el
        //Proceso B
if((childpid = fork()) == -1){
                perror("fork");
                exit(1);
        }

        if(childpid == 0){
        //Si el fork devolvió 0 estoy en el proceso hijo

         while(1){

                msgrcv(q_id, &buffer, msg_sz, B1, 0);
                        printf("B");
                        fflush(stdout);
                        buffer.type=C1;
                        msgsnd(q_id, &buffer, msg_sz, 0);

                msgrcv(q_id, &buffer, msg_sz, B2, 0);
                        printf("B");
                        fflush(stdout);
                        buffer.type=C2;
                        msgsnd(q_id, &buffer, msg_sz, 0);

                msgrcv(q_id, &buffer, msg_sz, BB, 0);
                        printf("BB");
                        fflush(stdout);
                        buffer.type=C3;
                        msgsnd(q_id, &buffer, msg_sz, 0);
        }
        exit(0);
}


//Estoy en el proceso padre y creo el
        // Proceso C
```

```
	if((childpid = fork()) == -1){
		perror("fork");
		exit(1);
	}

	if(childpid == 0){
	//Si el fork devolvió 0 estoy en el proceso hijo

	 while(1){

		 msgrcv(q_id, &buffer, msg_sz, C1, 0);
			printf("C");
			fflush(stdout);
			buffer.type=D1;
			msgsnd(q_id, &buffer, msg_sz, 0);

		 msgrcv(q_id, &buffer, msg_sz, C2, 0);
			printf("C");
			fflush(stdout);
			buffer.type=D2;
			msgsnd(q_id, &buffer, msg_sz, 0);

		 msgrcv(q_id, &buffer, msg_sz, C3, 0);
			printf("C");
			fflush(stdout);
			buffer.type=D3;
			msgsnd(q_id, &buffer, msg_sz, 0);
		}
		exit(0);

}

//Estoy en el proceso padre y creo el
	// Proceso D
	if((childpid = fork()) == -1){
		perror("fork");
		exit(1);
	}

	if(childpid == 0){
	//Si el fork devolvió 0 estoy en el proceso hijo

	 while(1){
```

```
                msgrcv(q_id, &buffer, msg_sz, D1, 0);
                        printf("D");
                        fflush(stdout);
                        buffer.type=E1;
                        msgsnd(q_id, &buffer, msg_sz, 0);

                msgrcv(q_id, &buffer, msg_sz, D2, 0);
                        printf("D ");
                        fflush(stdout);
                        buffer.type=BB;
                        msgsnd(q_id, &buffer, msg_sz, 0);

                msgrcv(q_id, &buffer, msg_sz, D3, 0);
                        printf("D");
                        fflush(stdout);
                        buffer.type=E2;
                        msgsnd(q_id, &buffer, msg_sz, 0);

                }
                exit(0);
}


//Estoy en el proceso padre y creo el
        // Proceso D
        if((childpid = fork()) == -1){
                perror("fork");
                exit(1);
        }

        if(childpid == 0){
        //Si el fork devolvió 0 estoy en el proceso hijo

         while(1){

                msgrcv(q_id, &buffer, msg_sz, E1, 0);
                        printf("E ");
                        fflush(stdout);
                        buffer.type=AA;
                        msgsnd(q_id, &buffer, msg_sz, 0);

                msgrcv(q_id, &buffer, msg_sz, E2, 0);
                        printf("E ");
                        fflush(stdout);
```

```
                        buffer.type=A;
                        msgsnd(q_id, &buffer, msg_sz, 0);

        }
        exit(0);
    }


    int i;
        for(i=0; i<5;i++){
                        wait(NULL);
        }

        msgctl(IPC_PRIVATE, IPC_RMID, NULL);  // Se elimina la cola.

        exit(0);
}
```

Ejercicio 1.2.a

Compilación: gcc -o barbero Ejercicio1.2.a.c -lpthread -lrt
Ejecución: ./barbero

Estrategia de resolución:

La estrategia de resolución de este ejercicio es muy similar a la del ejercicio siguiente. Definimos el struct de memoria compartida, el cual contiene tres semáforos que serán utilizados para la sincronización del barbero y los clientes:

BarberoListo: Representa los dos posibles estados del barbero, trabajando(1), descansando (0). Inicialmente el semáforo toma el valor 1, indicando que el barbero no está trabajando porque todavía no llegó ningún cliente

ClienteListo: Simula la cantidad de clientes esperando en la sala de espera por un corte. Inicialmente el semáforo toma el valor 0, indicando que todavía no llegó ningún cliente.

sillasLibres: representa el número de silla libres que hay en la sala de espera, inicialmente el semáforo toma el valor 10 para simular la sala vacía, cuando llega un nuevo cliente se decrementa el valor del semáforo y cuando el cliente es atendido por el barbero se incrementa el valor del semáforo.

Sincronización: creamos un nuevo proceso por cada uno de los 20 clientes, cada cliente intenta ocupar un lugar en la sala de espera (decrementando el valor del semáforo sillasLibres, si no puede hacerlo porque la sala está llena se bloquea y espera a que haya lugar), luego incrementa el valor del semáforo clienteListo indicando que está listo para ser atendido (envía un mensaje a cliente listo). Finalmente espera a que el barbero termine de trabajar.

Por el lado del barbero contamos con un solo proceso el cual inicialmente espera a que haya un cliente listo, luego cuando este cliente llega incrementa la cantidad de sillas libres en la sala y finalmente indica que termino de trabajar.

Código:

```
/**
   Barbero dormilon (Memoria compartida)
   Joaquin Piersigilli, Utizi Sebastian
**/
#include <sys/shm.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <fcntl.h>
#include <sys/mman.h>
#include <stdio.h>
#include <stdlib.h>
```

```
#include <unistd.h>
#include <semaphore.h>

//Defino el numero de clientes del barbero
#define NCLIENTES 20
//Defino el numero de sillas de la sala de espera
#define NSILLAS 10

typedef struct Memoria
{
   sem_t clienteListo; //Simulo la cantidad de clientes esperando, listos para un corte
   sem_t barberoTermino; //Simulo cuando el barbero termino un corte y esta libre
   sem_t sillasLibres; //Simulo la cantidad de sillas que quedan libres en la sala
} shared_data;

int main()
{
   int i;
   int clientes [NCLIENTES];
   //Creo y abro el nuevo espacio de memoria
   shared_data *mem;
   int memDescriptor;
   memDescriptor = shm_open("Memoria compartida", O_CREAT | O_RDWR, 0666);
   if(memDescriptor == -1)
   {
      perror("Error al crear la memoria compartida.");
      exit(1);
   }
   ftruncate(memDescriptor, sizeof(struct Memoria));
   mem = mmap(0, sizeof(struct Memoria), PROT_WRITE, MAP_SHARED, memDescriptor,
0);

   //Inicializo el struct
   sem_init(&(mem->barberoTermino), 1, 0);
   sem_init(&(mem->clienteListo), 1, 0);
   sem_init(&(mem->sillasLibres), 1, NSILLAS);

//Barbero
   int pid = fork();
   if(pid < 0)
   {
      perror("Error al crear al barbero.");
      exit(1);
   }
```

```
   if(pid == 0)
   {
      //Mientras haya clientes que atender
      while (1)
      {
         printf("El barbero esta descansando esperando a un cliente\n");
         fflush(stdout);
         sem_wait(&(mem->clienteListo));
         fflush(stdout);
         //Libero la silla ocupada por el cliente
         sem_post(&(mem->sillasLibres));
         printf("El barbero le corta el pelo a un cliente\n");
         fflush(stdout);
         sleep(2);
         printf("El barbero termino de trabajar.\n");
         fflush(stdout);
         sem_post(&(mem->barberoTermino));
      }
   }

//Cliente
   for(i=0; i<NCLIENTES; i++)
   {
      clientes[i] = fork();
      if(clientes[i] < 0)
      {
         perror("Error al crear los clientes.");
         exit(1);
      }
      if(clientes[i] == 0)
      {
         printf("El cliente: %d acaba de llegar.\n", i);
         fflush(stdout);
         //Espero a que haya lugar en la sala de espera
         sem_wait(&(mem->sillasLibres));
         printf("El cliente %d ocupa una silla de la sala.\n", i);
         fflush(stdout);
         //Espero a que la silla del barbero este libre
         sem_post(&(mem->clienteListo));
         sem_wait(&(mem->barberoTermino));
         exit(0);
      }
   }
```

```
    //Espero a que terminen todos los clientes
    for(i=0; i<NCLIENTES; i++)
    {
        wait(NULL);
    }

    return 0;
}
```

Ejercicio 1.2.b

Compilacion: gcc -o barbero Ejercicio1.2.b.c -lpthread -lrt
Ejecucion: ./barbero

Estrategia de resolución:

Creamos una cola de mensajes definimos tres tipos distintos:
BarberoListo: se envía un mensaje a la cola cuando el barbero termina de trabajar, es decir, permite modelar cuando el barbero está trabajando y cuando está descansando esperando por un nuevo cliente. Inicialmente hay un mensaje en esta cola, indicando que el barbero no esta trabajando porque todavía no llego ningun cliente
ClienteListo: se envía un mensaje a la cola cuando un nuevo cliente llega a la sala de espera, para avisarle al barbero que hay un nuevo cliente al cual atender.
sillasLibres: representa el número de silla libres que hay en la sala de espera, inicialmente se envian 10 mensajes a esta cola para simular la sala vacía, cuando llega un nuevo cliente consume un mensaje y cuando el cliente es atendido por el barbero se envía un mensaje a la cola.
Sincronización: creamos un nuevo proceso por cada uno de los 20 clientes, cada cliente ocupa un lugar en la sala de espera (recibe un mensaje de la cola de sillas libres), luego espera a que el barbero esté listo (recibe un mensaje de que el barbero está listo) y finalmente envía un mensaje de que está listo para ser atendido (envía un mensaje a cliente listo).
Por el lado del barbero contamos con un solo proceso el cual inicialmente espera a que haya un cliente listo (espera a recibir un mensaje en cliente listo), luego cuando este cliente llega incrementa la cantidad de sillas libres en la sala (envia un mensaje a sillas libres), finalmente envía un mensaje a la cola barberoListo para indicar que termino de trabajar.

```
/**
   Barbero dormilon (Cola de mensajes)
   Joaquin Piersigilli, Utizi Sebastian
**/

#include <sys/ipc.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <sys/msg.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <pthread.h>

#define barberoListo 1
#define clienteListo 2
```

```
#define sillasLibres 3
#define NCLIENTES 20
#define NSILLAS 10

typedef struct my_msg   // Estrucutra del mensaje.
{
   long type;
} msg;

int main()
{
   pid_t pid;  // ID de procesos.
   int q_id;
   int msg_sz;
   msg buffer;
   int i;
   int clientes[NCLIENTES];

   q_id = msgget(IPC_PRIVATE, IPC_CREAT | 0666);  //  Se crea la cola de mensajes.
   msg_sz = sizeof(struct my_msg);

   //Inicializo las sillas libres
   buffer.type = sillasLibres;
   for (i=0; i<NSILLAS; i++)
   {
      msgsnd(q_id, &buffer, msg_sz, 0);
   }

   //Inicialmente el barbero duerme y no trabaja
   buffer.type = barberoListo;
   msgsnd(q_id, &buffer, msg_sz, 0);

   //Barbero
   pid = fork();
   if(pid==0)
   {
      while(1)
      {
         msgrcv(q_id, &buffer, msg_sz, clienteListo, 0);
         buffer.type = sillasLibres;
         msgsnd(q_id, &buffer, msg_sz, 0);

         printf("El barbero le corta el pelo a un cliente\n");
         fflush(stdout);
```

```
            buffer.type=barberoListo;
            msgsnd(q_id, &buffer, msg_sz, 0);
        }
    }

    //Clientes
    for(i=0; i<NCLIENTES; i++)
    {
        clientes[i]=fork();

        if(clientes[i]==0)  //estoy en el hijo
        {
            printf("Llego un nuevo cliente\n");
            fflush(stdout);
            //Ocupo una silla en la sala
            msgrcv(q_id, &buffer, msg_sz, sillasLibres, 0);
            printf("El cliente entro en la sala\n");
            fflush(stdout);
            msgrcv(q_id, &buffer, msg_sz, barberoListo, 0);
            //El barbero esta listo y le aviso que hay un cliente esperando
            buffer.type=clienteListo;
            msgsnd(q_id, &buffer, msg_sz, 0);

            exit(0);
        }
    }

    //Espero a que terminen todos los hijos
    for(i=0; i<NCLIENTES; i++)
    {
        wait(NULL);
    }

    msgctl(IPC_PRIVATE, IPC_RMID, NULL);  // Se elimina la cola.

    return 0;
}
```

Ejercicio 2.1

Compilación: gcc -o Ejercicio2.1 Ejercicio2.1.c
Ejecución: ./Ejercicio2.1 <Ruta> <Permisos> donde: Ruta representa la ruta absoluta del archivo al cual se le desean cambiar los permisos y <permisos> representa los permisos que se le desean asignar, notar que los permisos deben ser ingresados en octal. Por ejemplo:
./Ejercicio 2.1 /home/usuario/Escritorio/archivo.txt 777

Para la resolución de este ejercicio utilizamos el system call chmod, su uso es análogo al comando chmod.
Decidimos tomar la decisión de que el usuario ingrese los comandos directamente en octal, ya que consideramos que el objetivo del enunciado no era la lectura y conversión de parámetros ingresados al momento de ejecutar el programa. Nos centramos en la asignación de permisos mediante la invocación al system call chmod.

Ejercicio 2.2:

Compilación: gcc -o Ejercicio2.2 Ejercicio2.2.c
Ejecución: ./Ejercicio2.2

Estrategia de resolución:

Dado que tanto dirección ingresada como el resultado mostrado estan en formato decimal, todos los cálculos son realizados en decimal.
Obtenemos la dirección ingresada y la convertimos a decimal para poder operarla.
Para poder determinar el numero de pagina, dividimos la direccion ingresada por el tamaño de cada página, considerando que el tamaño de página está dado en KB, por lo que debemos dividir la dirección ingresada por 4096B (4KB = 4096B).
Finalmente para obtener el desplazamiento obtenemos el módulo de la división de la dirección ingresada por el tamaño de la página, nuevamente expresado en B y no en KB.

Codigo:

```
/**,
   Ejercicio 2.2 Manejo de direcciones
   Joaquin Piersigilli, Utizi Sebastian
**/


#include <stdio.h>
#include <stdlib.h>
```

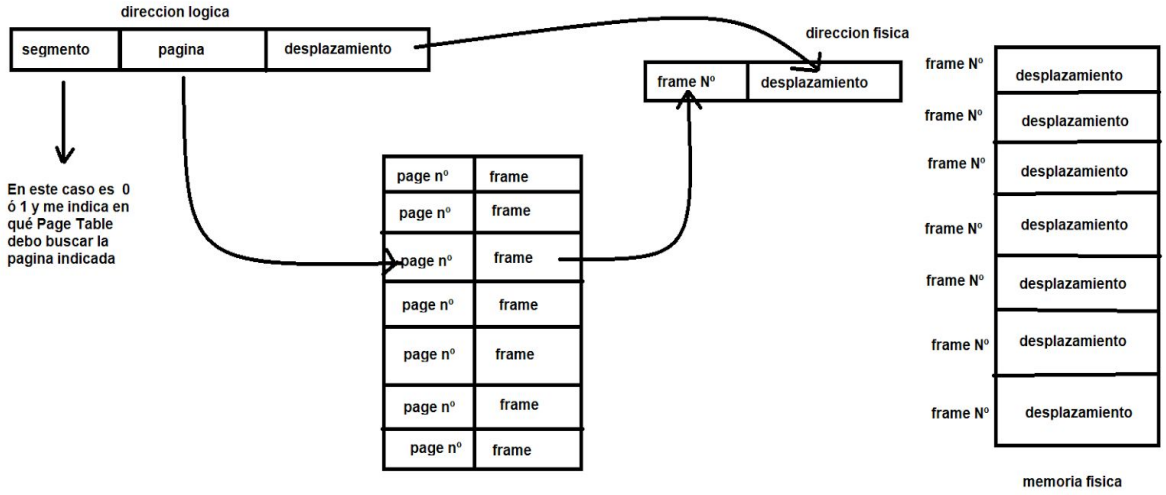
```
#define PAG 4096 //Defino el tamanio de la pagina
                 //4KB = 4096B

int main()
{
   char direccionString [20];
   int direccion;
   int desplazamiento;
   int pagina;

   printf("Ingrese una direccion en decimal:");
   scanf("%s", &direccionString);
   direccion = atoi(direccionString);

   pagina = direccion / PAG;
   desplazamiento = direccion % PAG;

   printf("Pagina: %i, desplazamiento: %i\n", pagina, desplazamiento);

   return 0;
}
```

Ejercicio 2.3:

Segment 0		Segment 1	
Read/Execute		Read/Write	
Virtual Page #	Page frame #	Virtual Page #	Page frame #
0	2	0	On Disk
1	On Disk	1	14
2	11	2	9
3	5	3	6
4	On Disk	4	On Disk
5	On Disk	5	13
6	4	6	8
7	3	7	12

La dirección lógica está dividida en segmento (segment), página (page) y desplazamiento (offset). El segmento funciona como índice en la Segment Table, una vez accedido ese

índice obtengo la dirección base a partir de la cual está la Page Table correspondiente. En este caso sólo hay 2 segmentos y la imagen nos brinda para cada uno las páginas que hay. Una vez conocida la Page Table, la página brindada en la dirección lógica funciona como índice, y accediendo a él conseguimos el Frame que se usará para la dirección física.



En este caso usando paginado bajo demanda, quiere decir que las direcciones se traen del disco a memoria a medida que se necesitan, y por lo tanto, si accede a un Page Frame que está en Disco, se produce un trap llamado Page Fault.

a) Fetch desde segmento 1, página 1, desplazamiento 3.

Al ingresar al segmento 1 y a la página 1, obtenemos el Frame “14” por lo tanto la dirección física a la que se accede es Frame=14, Desplazamiento=3 o (1110 | 0011) y se puede realizar el Fetch ya que para esto se necesitan permisos de lectura y en el segmento 1 estos permisos se tienen.

b) Store en segmento 0, página 0, desplazamiento 16.

No hace falta realizar las operaciones para saber a qué dirección se accede ya que se produce un Protection Fault debido a que el segmento 0 no tiene permisos de escritura.

c) Fetch desde segmento 1, página 4, desplazamiento 28.

Para buscar el Frame necesario para obtener la dirección de memoria física debemos posicionarnos en la Page Table asociada al segmento 1, una vez hecho esto nos posicionamos en la page 4 de dicha Page Table y al intentar obtener el Frame vemos que

éste se encuentra en el Disco, y por lo tanto, como es paginado bajo demanda, se produce una Page Fault.

d) Jump a la locación en segmento 1, página 3, desplazamiento 32.

En este caso no hace falta buscar la locación correspondiente al segmento 1, página 3, desplazamiento 32 ya que la instrucción Jump se realiza durante la ejecución de un programa para que éste continúe su ejecución en la instrucción indicada por el Jump, como en este caso esa instrucción se encuentra en el segmento 1, se produce una Protection Fault ya que el segmento 1 no tiene permisos de Ejecución.