Anexo A

El puente de un solo carril

A.1. Enunciado

Suponga un puente que tiene una carretera con un único carril por el que los coches pueden circular en un sentido o en otro. La anchura del carril hace imposible que dos coches puedan pasar de manera simultánea por el puente. El protocolo utilizado para atravesar el puente es el siguiente:

- Si no hay ningún coche circulando por el puente, entonces el primer coche en llegar cruzará el puente.
- Si un coche está atravesando el puente de norte a sur, entonces los coches que estén en el extremo norte del puente tendrán prioridad sobre los que vayan a cruzarlo desde el extremo sur.
- Del mismo modo, si un coche se encuentra cruzando de sur a norte, entonces los coches del extremo sur tendrán prioridad sobre los del norte.

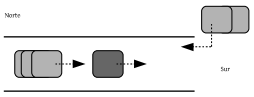

Figura A.1: Esquema gráfico del problema del puente de un solo carril.

A.2. Código fuente

Listado A.1: Makefile para la compilación automática

```
DIROBJ := obj/
DIREXE := exec/
DIRHEA := include/
DIRSRC := src/

CFLAGS := -I$(DIRHEA) -c -Wall
LDLIBS := -lpthread -lrt
CC := gcc

all : dirs manager coche

dirs:
    mkdir -p $(DIROBJ) $(DIREXE)

manager: $(DIROBJ)manager.o $(DIROBJ)semaforoI.o $(DIROBJ)memoriaI.o
    $(CC) -o $(DIREXE)$@ $^ $(LDLIBS)

coche: $(DIROBJ)coche.o $(DIROBJ)semaforoI.o $(DIROBJ)memoriaI.o
    $(CC) -o $(DIREXE)$@ $^ $(LDLIBS)

$(DIROBJ)%.o: $(DIRSRC)%.c
    $(CC) $(CFLAGS) $^ -o $@

clean :
    rm -rf *~ core $(DIROBJ) $(DIREXE) $(DIRHEA)*~ $(DIRSRC)*~
```

Listado A.2: Archivo semaforoI.h

```
#ifndef __SEMAFOROI_H__
#define __SEMAFOROI_H__
#include <semaphore.h>
// Crea un semáforo POSIX.
sem_t *crear_sem (const char *name, unsigned int valor);
// Obtiene un semáforo POSIX (ya existente).
sem_t *get_sem (const char *name);
// Cierra un semáforo POSIX.
void destruir_sem (const char *name);
// Incrementa el semáforo.
void signal_sem (sem_t *sem);
// Decrementa el semáforo.
void wait_sem (sem_t *sem);
#endif
```

Listado A.3: Archivo semaforoI.c

```c
#include <stdio.h>
#include <errno.h>
#include <stdlib.h>
#include <unistd.h>
#include <fcntl.h>
#include <semaforoI.h>

sem_t *crear_sem (const char *name, unsigned int valor) {
  sem_t *sem;
  sem = sem_open(name, O_CREAT, 0644, valor);
  if (sem == SEM_FAILED) {
    fprintf(stderr, "Error al crear el sem.: %s\n", strerror(errno));
    exit(1);
  }
  return sem;
}

sem_t *get_sem (const char *name) {
  sem_t *sem;
  sem = sem_open(name, O_RDWR);
  if (sem == SEM_FAILED) {
    fprintf(stderr, "Error al obtener el sem.: %s\n", strerror(errno));
    exit(1);
  }
  return sem;
}

void destruir_sem (const char *name) {
  sem_t *sem = get_sem(name);
  // Se cierra el sem.
  if ((sem_close(sem)) == -1) {
    fprintf(stderr, "Error al cerrar el sem.: %s\n", strerror(errno));
    exit(1);
  }
  // Se elimina el sem.
  if ((sem_unlink(name)) == -1) {
    fprintf(stderr, "Error al destruir el sem.: %s\n", strerror(errno));
    exit(1);
  }
}

void signal_sem (sem_t *sem) {
  if ((sem_post(sem)) == -1) {
    fprintf(stderr, "Error al modificar el sem.: %s\n", strerror(errno));
    exit(1);
  }
}

void wait_sem (sem_t *sem) {
  if ((sem_wait(sem)) == -1) {
    fprintf(stderr, "Error al modificar el sem.: %s\n", strerror(errno));
    exit(1);
  }
}
```

Listado A.4: Archivo memoriaI.h

```
#ifndef __VARIABLEI_H__
#define __VARIABLEI_H__
// Crea un objeto de memoria compartida y devuelve el descriptor de archivo.
int crear_var (const char *name, int valor);
// Obtiene el descriptor asociado a la variable.
int obtener_var (const char *name);
// Destruye el objeto de memoria compartida.
void destruir_var (const char *name);
// Modifica el valor del objeto de memoria compartida.
void modificar_var (int shm_fd, int valor);
// Devuelve el valor del objeto de memoria compartida.
void consultar_var (int shm_fd, int *valor);
#endif
```

Listado A.5: Archivo memoriaI.c (1 de 3)

```
#include <errno.h>
#include <fcntl.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/mman.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <unistd.h>
#include <memoriaI.h>

int crear_var (const char *name, int valor) {
  int shm_fd, *p;

  // Abre el objeto de memoria compartida.
  shm_fd = shm_open(name, O_CREAT | O_RDWR, 0666);
  if (shm_fd == -1) {
    fprintf(stderr, "Error al crear la variable: %s\n", strerror(errno));
    exit(1);
  }

  // Establecer el tamaño.
  if (ftruncate(shm_fd, sizeof(int)) == -1) {
    fprintf(stderr, "Error al truncar la variable: %s\n", strerror(errno));
    exit(1);
  }

  // Mapeo del objeto de memoria compartida.
  p = mmap(NULL, sizeof(int), PROT_READ | PROT_WRITE, MAP_SHARED, shm_fd, 0);
  if (p == MAP_FAILED) {
    fprintf(stderr, "Error al mapear la variable: %s\n", strerror(errno));
    exit(1);
  }

  *p = valor;
  munmap(p, sizeof(int));

  return shm_fd;
}
```

Listado A.6: Archivo memoriaI.c (2 de 3)

```c
/* includes previos... */

#include <memoriaI.h>

int obtener_var (const char *name) {
  int shm_fd;

 // Abre el objeto de memoria compartida.
  shm_fd = shm_open(name, O_RDWR, 0666);
  if (shm_fd == -1) {
    fprintf(stderr, "Error al obtener la variable: %s\n", strerror(errno));
    exit(1);
  }

  return shm_fd;
}

void destruir_var (const char *name) {
  int shm_fd = obtener_var(name);

  if (close(shm_fd) == -1) {
    fprintf(stderr, "Error al destruir la variable: %s\n", strerror(errno));
    exit(1);
  }

  if (shm_unlink(name) == -1) {
    fprintf(stderr, "Error al destruir la variable: %s\n", strerror(errno));
    exit(1);
  }
}

void modificar_var (int shm_fd, int valor) {
  int *p;

  // Mapeo del objeto de memoria compartida.
  p = mmap(NULL, sizeof(int), PROT_READ | PROT_WRITE,
       MAP_SHARED, shm_fd, 0);
  if (p == MAP_FAILED) {
    fprintf(stderr, "Error al mapear la variable: %s\n", strerror(errno));
    exit(1);
  }

  *p = valor;

  munmap(p, sizeof(int));
}
```

Listado A.7: Archivo memoriaI.c (3 de 3)

```
/* includes previos... */

#include <memoriaI.h>

void consultar_var (int shm_fd, int *valor) {
  int *p, valor;

  // Mapeo del objeto de memoria compartida.
  p = mmap(NULL, sizeof(int), PROT_READ | PROT_WRITE,
       MAP_SHARED, shm_fd, 0);
  if (p == MAP_FAILED) {
    fprintf(stderr, "Error al mapear la variable: %s\n", strerror(errno));
    exit(1);
  }

  *valor = *p;

  munmap(p, sizeof(int));
}
```

Listado A.8: Proceso manager

```
/* includes previos... */
#include <memoriaI.h>
#include <semaforoI.h>

#define COCHES 30       #define MAX_T_LANZAR 3  #define PUENTE "puente"
#define MUTEXN "mutexn" #define MUTEXS "mutexs" #define COCHESNORTE "cn" #define COCHESSUR "cs"

pid_t pids[COCHES];
void liberarecursos (); void finalizarprocesos (); void controlador (int senhal);

int main (int argc, char *argv[]) {
  pid_t pid_hijo; int i;

  srand((int)getpid());
  // Creación de semáforos y segmentos de memoria compartida.
  crear_sem(PUENTE, 1); crear_sem(MUTEXN, 1); crear_sem(MUTEXS, 1);
  crear_var(COCHESNORTE, 0); crear_var(COCHESSUR, 0);

  if (signal(SIGINT, controlador) == SIG_ERR) {
    fprintf(stderr, "Abrupt termination.\n"); exit(EXIT_FAILURE);
  }

  for (i = 0; i < COCHES; i++) {   // Se lanzan los coches...
    switch (pid_hijo = fork()) {
    case 0:
      if ((i % 2) == 0) execl("./exec/coche", "coche", "N", PUENTE, MUTEXN, COCHESNORTE, NULL);
      else              execl("./exec/coche", "coche", "S", PUENTE, MUTEXS, COCHESSUR, NULL);
      break;
    }
    sleep(rand() % MAX_T_LANZAR);
  }

  for (i = 0; i < COCHES; i++) waitpid(pids[i], 0, 0);
  liberarecursos(); return EXIT_SUCCESS;
}

void liberarecursos () {
  destruir_sem(PUENTE); destruir_sem(MUTEXN); destruir_sem(MUTEXS);
  destruir_var(COCHESNORTE); destruir_var(COCHESSUR);
}

void finalizarprocesos () {
  int i; printf ("\n--- Finalizando procesos --- \n");
  for (i = 0; i < COCHES; i++)
    if (pids[i]) {
      printf ("Finalizando proceso [%d]... ", pids[i]);
      kill(pids[i], SIGINT); printf ("<Ok>\n");
    }
}

void controlador (int senhal) {
  finalizarprocesos(); liberarecursos();
  printf ("\nFin del programa (Ctrol + C)\n"); exit(EXIT_SUCCESS);
}
```

Listado A.9: Proceso coche

```
/* includes previos... */
#include <memoriaI.h>
#include <semaforoI.h>

#define MAX_TIME_CRUZAR 5

void coche (); void cruzar ();

int main (int argc, char *argv[]) {
  coche(argv[1], argv[2], argv[3], argv[4]);
  return 0;
}

/* salida permite identificar el origen del coche (N o S) */
void coche (char *salida, char *id_puente, char *id_mutex, char *id_num_coches) {
  int num_coches_handle, valor;
  sem_t *puente, *mutex;

  srand((int)getpid());

  puente = get_sem(id_puente);
  mutex = get_sem(id_mutex);
  num_coches_handle = obtener_var(id_num_coches);

  /* Acceso a la variable compartida 'num coches' */
  wait_sem(mutex);

  consultar_var(num_coches_handle, &valor);
  modificar_var(num_coches_handle, ++valor);
  /* Primer coche que intenta cruzar desde su extremo */
  if (valor == 1) {
    wait_sem(puente); /* Espera el acceso al puente */
  }

  signal_sem(mutex);

  cruzar(salida); /* Tardará un tiempo aleatorio */

  wait_sem(mutex);

  consultar_var(num_coches_handle, &valor);
  modificar_var(num_coches_handle, --valor);
  /* Ultimo coche en cruzar desde su extremo */
  if (valor == 0) {
    signal_sem(puente); /* Libera el puente */
  }

  signal_sem(mutex);
}

void cruzar (char *salida) {
  if (strcmp(salida, "N") == 0) printf("%d cruzando de N a S...\n", getpid());
  else                          printf("%d cruzando de S a N...\n", getpid());

  sleep(rand() % MAX_TIME_CRUZAR + 1);
}
```