OPERATING SYSTEMS AND CONCURRENCY.

LAB 3.

DANIYAR NAZARBAYEV.

If part 2 or 3 are not working, try killing the semaphore first. I didn’t bother to create a ctrl+c handler.

Personally I just used this line of code to kill the semaphore.

```
    sem_unlink(SHNAME); exit(1);
```

PART 1.

1. Makefile

```
all: server client

ifdef OSX
CC      = gcc
CCFLAGS = -Wall
LIBS    =

else
CC      = gcc
CCFLAGS = -Wall
LIBS    = -lrt
endif

server: server.c shm.h shm.o
      $(CC) $(CCFLAGS) shm.o -o $@ $< $(LIBS)

client: client.c shm.h shm.o
      $(CC) $(CCFLAGS) shm.o -o $@ $< $(LIBS)

shm.o: shm.c
      $(CC) $(CCFLAGS) -c $< $(LIBS)

clean:
      $(RM) *.out
```

2. server.c

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

#include "shm.h"

int ticket_counter = 1;

int main()
{
    FILE * fp;
    int shmFd;
    struct SHM initData = { ticket_counter, false, false};
    struct SHM * shmData;

    fp = fopen("/tmp/server.log", "a");
    if(fp == NULL) {
        perror("fopen");
        return EXIT_FAILURE;
    }

    fprintf(stderr, "Shared Memory Area created\n");
    shmFd = createSHM(SHNAME);
    shmData = initSHM( shmFd, &initData );

    // Remember the condition value!!!
    while(ticket_counter <= MAX_TICKETS)
    {
      shmData->ticket = ticket_counter;
      shmData->isTaken = false;
      shmData->soldOut = false;

      fprintf(fp, "ticket %d - %s\n", ticket_counter, getTimeStamp());
      fprintf(stderr, "ticket %d - %s\n", ticket_counter, getTimeStamp());

        // Fill in here
      while(shmData->isTaken != true){sleep(MAX_SLEEP);}
      fprintf(fp, "ticket %d taken - %s\n", ticket_counter, getTimeStamp());
      fprintf(stderr, "ticket %d taken - %s\n", ticket_counter, getTimeStamp());
      ticket_counter++;
    }
      shmData->soldOut = true;
fprintf(fp, "tickets sold out - %s\n", getTimeStamp());
fprintf(stderr, "tickets sold out - %s\n", getTimeStamp());

    fprintf(stderr, "Shared Memory Area destroyed\n");
    clearSHM(shmData);
    closeSHM(shmFd);
    destroySHM(SHNAME);
    fclose(fp);

    return EXIT_SUCCESS;
}
```

3. client.c

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


#include "shm.h"

int main()
{
    FILE * fp;
    int shmFd;
    SHMstruct * shmData;

    fp = fopen("/tmp/client.log", "a");
    if(fp == NULL) {
        perror("fopen");
        return EXIT_FAILURE;
    }

    shmFd = loadSHM(SHNAME);
    shmData = accessSHM(shmFd);

    // Remember the condition value!!!
    while(shmData->soldOut != true)
    {
      shmData->isTaken = true;
      fprintf(fp, "ticket %d acquired - %s\n", shmData->ticket, getTimeStamp());
      fprintf(stderr, "ticket %d acquired - %s\n", shmData->ticket,
getTimeStamp());
      while(shmData->isTaken == true && shmData->soldOut == false)
{sleep(MAX_SLEEP);}
    }
```

```
      fprintf(fp, "tickets sold out - %s\n", getTimeStamp());
      fprintf(stderr, "tickets sold out - %s\n", getTimeStamp());

    clearSHM(shmData);
    closeSHM(shmFd);
    fclose(fp);

    return EXIT_SUCCESS;
}
```

4. shm.c

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

#include "shm.h"

// https://www.softprayog.in/programming/interprocess-communication-using-posix-
shared-memory-in-linux

/* Generate a human-readable timestamp */
char * getTimeStamp() {
    time_t ltime = time(NULL);
    return strtok(ctime(&ltime), "\n");
}

/* Create Shared Memory Segment
 *
 * Function creates a named SHM file descriptor on the filesystem.
 *
 * @param shname Name of SHM
 * @return file descriptor
 */
int createSHM(char * shname)
{
int fd = shm_open(shname, O_CREAT | O_RDWR, S_IRUSR | S_IWUSR);
if (fd == -1){perror("shm_open");}

if (ftruncate(fd, sizeof(struct SHM)) == -1){perror("ftruncate");}

return fd;
}

/* Load Shared Memory
 *
 * Function loads an existing named SHM, or gracefully fails
 * when the SHM does not exist.
 *
 * @param shname Name of SHM region
 * @return file descriptor
 */
int loadSHM(char * shname)
{
```

```
int fd = shm_open(shname, O_RDWR, S_IRUSR | S_IWUSR);
if (fd == -1){perror("shm_open");}

if (ftruncate(fd, sizeof(struct SHM)) == -1){perror("ftruncate");}

return fd;
}

/* Access Existing SHM
 *
 * From an existing SHM file descriptor, allocate the SHMstruct and
 * return its pointer.
 *
 * @param fd File descriptor of existing SHM
 * @return Pointer to SHMstruct
 */
struct SHM * accessSHM(int fd) {
      struct SHM * temp = mmap(NULL, sizeof(struct SHM),
PROT_READ | PROT_WRITE, MAP_SHARED, fd, 0);
      if (temp == MAP_FAILED){perror("mmap");}
      return temp;
}

/* Initialise SHM
 *
 * From an existing SHM file descriptor, initilise the SHMstruct variable.
 *
 * HINT: use accessSHM()
 *
 * @param fd File descriptor of SHM
 * @return Pointer to SHMstruct
 */
struct SHM * initSHM(int fd, struct SHM *data) {
      struct SHM * temp = accessSHM(fd);
      temp->ticket = data->ticket;
      temp->isTaken = data->isTaken;
      temp->soldOut = data->soldOut;
      return temp;
}

/* De-allocate SHMstruct
 *
 * Function de-allocates an already existing SHMstruct.
 *
 * @param shm Pointer to SHMstruct
 */
void clearSHM(struct SHM * shm)
{
      if(munmap(shm, sizeof(struct SHM)) == -1){perror("munmap");}
}

/* Close SHM file descriptor
 *
 * Function closes an existing SHM file descriptor.
 *
 * @param fd File descriptor of SHM
 */
void closeSHM(int fd)
{
      if(close(fd) == -1){perror("close shm");}
}

/* Unlink SHM
 *
 * Function destroys an existing SHM assuming that its
 * allocated memory has been de-allocated.
 *
 * @param shname Name of SHM
 */
void destroySHM(char * shname)
{
      if(shm_unlink(shname) == -1){perror("shm destroy");}
}
```

5. shm.h

```
#ifndef _shm_h_
#define _shm_h_

#include <stdbool.h>

#define SHNAME "/shmserver" // shared memory
#define MAX_TICKETS 10
#define MAX_SLEEP 1 // seconds

typedef struct SHM {
    int ticket;
    bool isTaken;
    bool soldOut;
} SHMstruct;

extern char * getTimeStamp();

extern int createSHM(char *shname);
extern int loadSHM( char *shname);

extern SHMstruct* initSHM( int fd, SHMstruct *data);
extern SHMstruct * accessSHM(int fd);

extern void clearSHM(SHMstruct * shm);
extern void closeSHM(int fd);
extern void destroySHM(char * shname);

#endif
```

PART 2.

1. Makefile

```
all: server client

ifdef OSX
CC      = gcc
CCFLAGS = -Wall
LIBS    =

else
CC      = gcc
CCFLAGS = -Wall
LIBS    = -lrt -pthread
endif

server: server.c shm.h shm.o
	$(CC) $(CCFLAGS) shm.o -o $@ $< $(LIBS)

client: client.c shm.h shm.o
	$(CC) $(CCFLAGS) shm.o -o $@ $< $(LIBS)

shm.o: shm.c
	$(CC) $(CCFLAGS) -c $< $(LIBS)

clean:
	$(RM) *.out
```

2. server.c

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <fcntl.h>           /* For O_* constants */
#include <sys/stat.h>        /* For mode constants */

#include "shm.h"
#include <semaphore.h>

int ticket_counter = 1;

int main()
{
    FILE * fp;
    int shmFd;
    struct SHM initData = { ticket_counter, false, false};
    struct SHM * shmData;

    fp = fopen("/tmp/server.log", "a");
    if(fp == NULL) {
        perror("fopen");
        return EXIT_FAILURE;
    }

    fprintf(stderr, "Shared Memory Area created\n");
    shmFd = createSHM(SHNAME);
    shmData = initSHM( shmFd, &initData );
```

```
      sem_t * semiphore = sem_open(SHNAME, O_CREAT, 0644, 1);

int previous_num = 0;

    // Remember the condition value!!!
    while(ticket_counter < MAX_TICKETS)
    {
      if(previous_num != ticket_counter){
            fprintf(fp, "ticket %d - %s\n", ticket_counter, getTimeStamp());
            fprintf(stderr, "ticket %d - %s\n", ticket_counter, getTimeStamp());
      }

previous_num = ticket_counter;


      sem_wait(semiphore);
            if(shmData->isTaken == true){
                  fprintf(fp, "ticket %d taken - %s\n", ticket_counter, 
getTimeStamp());
                  fprintf(stderr, "ticket %d taken - %s\n", ticket_counter, 
getTimeStamp());
                  ticket_counter++;
                  sleep(MAX_SLEEP);
                  shmData->ticket = ticket_counter;
                  shmData->isTaken = false;
            }
      sem_post(semiphore);
    }

sem_wait(semiphore);
shmData->soldOut = true;
sem_post(semiphore);

fprintf(fp, "tickets sold out - %s\n", getTimeStamp());
fprintf(stderr, "tickets sold out - %s\n", getTimeStamp());

    fprintf(stderr, "Shared Memory Area destroyed\n");
    clearSHM(shmData);
    closeSHM(shmFd);
    destroySHM(SHNAME);
    fclose(fp);
      sem_close(semiphore);
      sem_unlink(SHNAME);

    return EXIT_SUCCESS;
}
```

3. client.c

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <fcntl.h>           /* For O_* constants */
#include <sys/stat.h>        /* For mode constants */


#include "shm.h"
#include <semaphore.h>

int counter = 1;
int main()
{
    FILE * fp;
    int shmFd;
    SHMstruct * shmData;

    fp = fopen("/tmp/client.log", "a");
    if(fp == NULL) {
        perror("fopen");
        return EXIT_FAILURE;
    }

    shmFd = loadSHM(SHNAME);
    shmData = accessSHM(shmFd);

      sem_t * semiphore = sem_open(SHNAME, O_RDWR);
bool cond = false;

    // Remember the condition value!!!
    while(cond != true)
    {

      sem_wait(semiphore);
            cond = shmData->soldOut;
            if(shmData->isTaken == false){
                  shmData->isTaken = true;
                  fprintf(fp, "ticket %d acquired - %s\n", shmData->ticket,
getTimeStamp());
                  fprintf(stderr, "ticket %d acquired - %s\n", shmData->ticket,
getTimeStamp());
}
      sem_post(semiphore);
    }

fprintf(fp, "tickets sold out - %s\n", getTimeStamp());
fprintf(stderr, "tickets sold out - %s\n", getTimeStamp());

    clearSHM(shmData);
    closeSHM(shmFd);
    fclose(fp);

      sem_close(semiphore);

    return EXIT_SUCCESS;
}
```

4. shm.c

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

#include "shm.h"
```

```
// https://www.softprayog.in/programming/interprocess-communication-using-posix-
shared-memory-in-linux

/* Generate a human-readable timestamp */
char * getTimeStamp() {
    time_t ltime = time(NULL);
    return strtok(ctime(&ltime), "\n");
}

/* Create Shared Memory Segment
 *
 * Function creates a named SHM file descriptor on the filesystem.
 *
 * @param shname Name of SHM
 * @return file descriptor
 */
int createSHM(char * shname)
{
int fd = shm_open(shname, O_CREAT | O_RDWR, S_IRUSR | S_IWUSR);
if (fd == -1){perror("shm_open");}

if (ftruncate(fd, sizeof(struct SHM)) == -1){perror("ftruncate");}

return fd;
}

/* Load Shared Memory
 *
 * Function loads an existing named SHM, or gracefully fails
 * when the SHM does not exist.
 *
 * @param shname Name of SHM region
 * @return file descriptor
 */
int loadSHM(char * shname)
{
int fd = shm_open(shname, O_RDWR, S_IRUSR | S_IWUSR);
if (fd == -1){perror("shm_open");}

if (ftruncate(fd, sizeof(struct SHM)) == -1){perror("ftruncate");}

return fd;
}

/* Access Existing SHM
 *
 * From an existing SHM file descriptor, allocate the SHMstruct and
 * return its pointer.
 *
 * @param fd File descriptor of existing SHM
 * @return Pointer to SHMstruct
 */
struct SHM * accessSHM(int fd) {
      struct SHM * temp = mmap(NULL, sizeof(struct SHM),
PROT_READ | PROT_WRITE, MAP_SHARED, fd, 0);
      if (temp == MAP_FAILED){perror("mmap");}
      return temp;
}

/* Initialise SHM
 *
 * From an existing SHM file descriptor, initilise the SHMstruct variable.
 *
 * HINT: use accessSHM()
 *
 * @param fd File descriptor of SHM
 * @return Pointer to SHMstruct
 */
struct SHM * initSHM(int fd, struct SHM *data) {
      struct SHM * temp = accessSHM(fd);
      temp->ticket = data->ticket;
      temp->isTaken = data->isTaken;
      temp->soldOut = data->soldOut;
      return temp;
}

/* De-allocate SHMstruct
 *
 * Function de-allocates an already existing SHMstruct.
 *
 * @param shm Pointer to SHMstruct
 */
void clearSHM(struct SHM * shm)
{
      if(munmap(shm, sizeof(struct SHM)) == -1){perror("munmap");}
}

/* Close SHM file descriptor
 *
 * Function closes an existing SHM file descriptor.
 *
 * @param fd File descriptor of SHM
 */
void closeSHM(int fd)
{
      if(close(fd) == -1){perror("close shm");}
}

/* Unlink SHM
 *
 * Function destroys an existing SHM assuming that its
 * allocated memory has been de-allocated.
 *
 * @param shname Name of SHM
 */
void destroySHM(char * shname)
{
      if(shm_unlink(shname) == -1){perror("shm destroy");}
}
```

5. shm.h

```
#ifndef _shm_h_
#define _shm_h_

#include <stdbool.h>

#define SHNAME "/shmserver" // shared memory
#define MAX_TICKETS 20
#define MAX_SLEEP 1 // seconds

typedef struct SHM {
    int ticket;
    bool isTaken;
    bool soldOut;
} SHMstruct;

extern char * getTimeStamp();

extern int createSHM(char *shname);
extern int loadSHM( char *shname);

extern SHMstruct* initSHM( int fd, SHMstruct *data);
extern SHMstruct * accessSHM(int fd);

extern void clearSHM(SHMstruct * shm);
extern void closeSHM(int fd);
extern void destroySHM(char * shname);

#endif
```

PART 3.

1. Makefile

```
all: server client

ifdef OSX
CC      = gcc
CCFLAGS = -Wall
LIBS    =

else
CC      = gcc
CCFLAGS = -Wall
LIBS    = -lrt -pthread
endif

server: server.c shm.h shm.o
      $(CC) $(CCFLAGS) shm.o -o $@ $< $(LIBS)

client: client.c shm.h shm.o
      $(CC) $(CCFLAGS) shm.o -o $@ $< $(LIBS)

shm.o: shm.c
      $(CC) $(CCFLAGS) -c $< $(LIBS)

clean:
      $(RM) *.out
```

2. server.c

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <fcntl.h>           /* For O_* constants */
#include <sys/stat.h>        /* For mode constants */

#include "shm.h"
#include <semaphore.h>

int ticket_counter = 1;

int main()
{
    FILE * fp;
    int shmFd;
    struct SHM initData = { {ticket_counter, ++ticket_counter, +
+ticket_counter}, {false, false, false}, false};
    struct SHM * shmData;

    fp = fopen("/tmp/server.log", "a");
    if(fp == NULL) {
        perror("fopen");
        return EXIT_FAILURE;
    }

      //sem_unlink(SHNAME); exit(1);
    fprintf(stderr, "Shared Memory Area created\n");
    shmFd = createSHM(SHNAME);
    shmData = initSHM( shmFd, &initData );

      sem_t * semiphore = sem_open(SHNAME, O_CREAT, 0644, 5);

int previous_num[3] = {0};
sem_wait(semiphore);
    // Remember the condition value!!!
    while(ticket_counter <= MAX_TICKETS)
    {

      int x = 0;
      while(x<3){
            //fprintf(stderr, "ticket %d\n", shmData->ticket[x]);
            if(shmData->ticket[x] != 100 && previous_num[x] != shmData-
>ticket[x]){
                  fprintf(fp, "ticket %d - %s\n", shmData->ticket[x],
getTimeStamp());
                  fprintf(stderr, "ticket %d - %s\n", shmData->ticket[x],
getTimeStamp());
            }
      previous_num[x] = shmData->ticket[x];
      x++;
      }

      x = 0;
      while(x<3){
            if(shmData->ticket[x] != 100 && shmData->isTaken[x] == true){
                  fprintf(fp, "ticket %d taken - %s\n", shmData->ticket[x],
getTimeStamp());
                  fprintf(stderr, "ticket %d taken - %s\n", shmData->ticket[x],
getTimeStamp());
                  ticket_counter++;
                  sleep(MAX_SLEEP);
                        //reset
                        if(ticket_counter<MAX_TICKETS) {
                        shmData->ticket[x] = ticket_counter;
                        shmData->isTaken[x] = false;
                        } else {
                              shmData->ticket[x] = 100;
                        }
            }
      x++;
      }

    }

shmData->soldOut = true;
sem_post(semiphore);

fprintf(fp, "tickets sold out - %s\n", getTimeStamp());
fprintf(stderr, "tickets sold out - %s\n", getTimeStamp());

    fprintf(stderr, "Shared Memory Area destroyed\n");
    clearSHM(shmData);
    closeSHM(shmFd);
    destroySHM(SHNAME);
    fclose(fp);
      sem_close(semiphore);
      sem_unlink(SHNAME);

    return EXIT_SUCCESS;
}
```

```
void s_pause(sem_t * semiphore){
      while(sem_wait(semiphore) != 0){}
}

void s_resume(sem_t * semiphore){
      while(sem_post(semiphore) != 0){}
}
```

3. client.c

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <fcntl.h>           /* For O_* constants */
#include <sys/stat.h>        /* For mode constants */


#include "shm.h"
#include <semaphore.h>

int counter = 1;

int main()
{
    FILE * fp;
    int shmFd;
    SHMstruct * shmData;

    fp = fopen("/tmp/client.log", "a");
    if(fp == NULL) {
        perror("fopen");
        return EXIT_FAILURE;
    }

    shmFd = loadSHM(SHNAME);
    shmData = accessSHM(shmFd);

      sem_t * semiphore = sem_open(SHNAME, O_RDWR);
bool cond = false;
int x = 100;
int y;

    // Remember the condition value!!!
    while(cond != true)
    {
      sem_wait(semiphore);
      if(x==100){
      sem_getvalue(semiphore, &x);
      fprintf(stderr, "%d\n", x);
            switch(x){
                  case 3:
                        y = 0;
                        break;
                  case 2:
                        y = 1;
                        break;
                  case 1:
                        y = 2;
                        break;
                  default:
                        y = 100;
                        x = 100;
            }
      }
            sleep(1);
      cond = shmData->soldOut;
      if(y != 100 && shmData->isTaken[y] == false){

            shmData->isTaken[y] = true;
            fprintf(fp, "ticket %d acquired - %s\n", shmData->ticket[y],
getTimeStamp());
            fprintf(stderr, "ticket %d acquired - %s\n", shmData->ticket[y],
getTimeStamp());
      }
      sem_post(semiphore);
    }

fprintf(fp, "tickets sold out - %s\n", getTimeStamp());
fprintf(stderr, "tickets sold out - %s\n", getTimeStamp());

    clearSHM(shmData);
    closeSHM(shmFd);
    fclose(fp);

      sem_close(semiphore);

    return EXIT_SUCCESS;
}

void s_pause(sem_t * semiphore){
      while(sem_wait(semiphore) != 0){}
}

void s_resume(sem_t * semiphore){
      while(sem_post(semiphore) != 0){}
}
```

4. shm.c

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

#include "shm.h"

// https://www.softprayog.in/programming/interprocess-communication-using-posix-
shared-memory-in-linux

/* Generate a human-readable timestamp */
char * getTimeStamp() {
    time_t ltime = time(NULL);
    return strtok(ctime(&ltime), "\n");
}
```

```
/* Create Shared Memory Segment
 *
 * Function creates a named SHM file descriptor on the filesystem.
 *
 * @param shname Name of SHM
 * @return file descriptor
 */
int createSHM(char * shname)
{
int fd = shm_open(shname, O_CREAT | O_RDWR, S_IRUSR | S_IWUSR);
if (fd == -1){perror("shm_open");}

if (ftruncate(fd, sizeof(struct SHM)) == -1){perror("ftruncate");}

return fd;
}

/* Load Shared Memory
 *
 * Function loads an existing named SHM, or gracefully fails
 * when the SHM does not exist.
 *
 * @param shname Name of SHM region
 * @return file descriptor
 */
int loadSHM(char * shname)
{
int fd = shm_open(shname, O_RDWR, S_IRUSR | S_IWUSR);
if (fd == -1){perror("shm_open");}

if (ftruncate(fd, sizeof(struct SHM)) == -1){perror("ftruncate");}

return fd;
}

/* Access Existing SHM
 *
 * From an existing SHM file descriptor, allocate the SHMstruct and
 * return its pointer.
 *
 * @param fd File descriptor of existing SHM
 * @return Pointer to SHMstruct
 */
struct SHM * accessSHM(int fd) {
      struct SHM * temp = mmap(NULL, sizeof(struct SHM),
PROT_READ | PROT_WRITE, MAP_SHARED, fd, 0);
      if (temp == MAP_FAILED){perror("mmap");}
      return temp;
}

/* Initialise SHM
 *
 * From an existing SHM file descriptor, initilise the SHMstruct variable.
 *
 * HINT: use accessSHM()
 *
 * @param fd File descriptor of SHM
 * @return Pointer to SHMstruct
 */
struct SHM * initSHM(int fd, struct SHM *data) {
      struct SHM * temp = accessSHM(fd);
      memcpy (temp, data, sizeof (struct SHM));
      return temp;
}
```

```c
/* De-allocate SHMstruct
 *
 * Function de-allocates an already existing SHMstruct.
 *
 * @param shm Pointer to SHMstruct
 */
void clearSHM(struct SHM * shm)
{
      if(munmap(shm, sizeof(struct SHM)) == -1){perror("munmap");}
}

/* Close SHM file descriptor
 *
 * Function closes an existing SHM file descriptor.
 *
 * @param fd File descriptor of SHM
 */
void closeSHM(int fd)
{
      if(close(fd) == -1){perror("close shm");}
}

/* Unlink SHM
 *
 * Function destroys an existing SHM assuming that its
 * allocated memory has been de-allocated.
 *
 * @param shname Name of SHM
 */
void destroySHM(char * shname)
{
      if(shm_unlink(shname) == -1){perror("shm destroy");}
}
```

5. shm.h

```c
#ifndef _shm_h_
#define _shm_h_

#include <stdbool.h>

#define SHNAME "/shmserver" // shared memory
#define MAX_TICKETS 21
#define MAX_SLEEP 1 // seconds

typedef struct SHM {
    int ticket[3];
    bool isTaken[3];
    bool soldOut;
} SHMstruct;

extern char * getTimeStamp();

extern int createSHM(char *shname);
extern int loadSHM( char *shname);

extern SHMstruct* initSHM( int fd, SHMstruct *data);
extern SHMstruct * accessSHM(int fd);

extern void clearSHM(SHMstruct * shm);
extern void closeSHM(int fd);
extern void destroySHM(char * shname);

#endif
```