Module 11

Synchronization

Overview

- Objectives
- Relevance

Process Synchronization

Signals	`kill(2)`
	`sigaction(2)`
		Sends a signal.
		Catches a signal.

Record locking	`fcntl(2)`	Locks part of a file. Exclusive read and write locks provided can be blocking or non-blocking.

System V semaphores	`semget(2)`
	`semctl(2)`
	`semop(2)`
		Gets a semaphore ID.
		Sets a semaphore explicitly.
		Performs a set of semaphore operations (raise, lower, and wait).

Process Synchronization

Mechanism	**Comment**
mutex lock	For mutual exclusion; simple and fast.
reader-writer lock	Multiple-reader single-writer lock; useful when writes are fast and infrequent compared to reads.
semaphore	Pure Dijkstra semaphore.
condition variable	Useful in producer-consumer situations.

Thread Synchronization

- Coordination
- Single-thread processes
- Multithreaded processes
- Locking mechanisms
- Test and Set

Types of Synchronization Mechanisms

- Mutual exclusion lock
- Multiple-reader single-writer lock
- Semaphore
- Condition variable

Thread Synchronization Within a Process

- Multithreaded POSIX mutex locks
- Rwlocks and shared data structures
- Semaphores regulating bank teller resources
- Condition variables and a threadsafe stack

mutex.c

```
/****************************************************
This program demonstrates the use of a POSIX thread
mutex to coordinate access to a shared resource.
Two threads are created, both of which access a
common counter. Delays are used to cause contention to
prove that the mutexes really work.

This program operates in two modes. With no
commandline arguments, the program operates without
synchronization and the threads fall all over
themselves. With argument “sync” passed on the
commandline, synchronization is used and the
program operates correctly.

/**************************************************/

#include <sys/types.h>/* For general. */
#include <stdio.h>/* Standard IO Library. */
#include <pthread.h>/* Posix threads. */
#include <errno.h>/* Error definitions. */
```

```

#include "utils.h"/* for fractSleep() and printWithTime(). */

#define LOOP_MAX 20/* Each thread counts up this many
iterations.*/



/* Global stuff. */

/* Common counter incremented by all threads. */
int commonCounter= 0;

/* mutex lock.  Note the initializer can be used only
in this form of declaration. */
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;

/* Delay to cause contention. */
double delay = 0.2;

/* Synchronized flag. */
boolean_t synchronized = B_FALSE;
```

```

/* Forward references. */
void * threadMain(void *);

/***************************************************/
main(int argc, char *argv[]) {
/***************************************************
Main program.  Get commandline arguments and spawn threads.
/***************************************************/

  /* Define two threads. */
  pthread_t threadID_1;
  pthread_t threadID_2;

  /* We have an argument and it is "sync": run in
synchronized mode.*/
  if ((argc > 1) && (!(strcmp(argv[1],"sync")))) {
    synchronized = B_TRUE;
  }

  printf ("\n%s running with sync %s\n\n",argv[0],
     (synchronized)?"Enabled":"Disabled");
```

```

  /* Create threads. */
  if ((pthread_create( &threadID_1, NULL,
     threadMain, NULL)) ||
     (pthread_create( &threadID_2, NULL,
     threadMain, NULL))) {
    perror ("Error starting thread(s)");
    exit (errno);
  }

  /* Wait for threads to finish. */
  pthread_join(threadID_1, (void **)NULL);
  pthread_join(threadID_2, (void **)NULL);
}

/***************************************************/
void * threadMain(void *dummy) {
/***************************************************
This is the main for each thread. Get counter
value from global, hold it for a while to cause
contention, then replace it with an incremented
value. While it is held, another thread tries to
```

```
get it and increment it as well.

When running in synchronized mode, get the mutex
lock before accessing the global, and release it
only after we are done with the global. This causes
other threads to wait for the global until we are
finished with it.
/***************************************************/

  int loopCount;/* Loop counter. */
  int temp;/* Used in modification delay. */
  char buffer[80];/* For writing status. */

  for (loopCount = 0; loopCount < LOOP_MAX;
    loopCount++){

    /* Get mutex lock if running in synchronized
       mode. */
    if (synchronized) {
       pthread_mutex_lock(&mutex);
    }

    /* Now that we have the lock, we can safely
       increment the counter without worry that the
       other thread accesses it at the same time. */

    /* Print out initial value. */
    sprintf (buffer, "Common counter incremented "
        "from %d to ", commonCounter);
    write(STDOUT_FILENO, buffer, strlen(buffer));

    /* Modify common counter.  Take a long time to
    do it to cause contention. */
    temp = commonCounter;
    fractSleep(delay);
    commonCounter = temp + 1;

    /* Update status output. */
    sprintf (buffer, "%d\n", commonCounter);
    write(STDOUT_FILENO, buffer, strlen(buffer));

    /* Release mutex lock if running in
    synchronized mode. */
    if (synchronized) {
```

```
      pthread_mutex_unlock(&mutex);
    }

    /* Delay for half as long as the increment
    delay, to cause contention. */
    fractSleep(delay/2.0);
  }
}
/**********************************************/
void fractSleep(float time) {
/**********************************************
  Routine to delay for fractions of a second.
*********************************************/

static struct timespec timeSpec;

/* Truncate fraction in int conversion. */
timeSpec.tv_sec = (time_t)time;

/* Load fraction. */
timeSpec.tv_nsec = (int)((time - timeSpec.tv_sec)*
1000000000);
```

```

/* Delay. */
nanosleep(&timeSpec, NULL);
}
```

rw.c

```
.
.

/* Multiple Reader Single Writer lock. */
  pthread_rwlock_t rwlock =
        PTHREAD_RWLOCK_INITIALIZER;
.
.
/************************************************
This the main for the writer thread. Get the
appropriate lock, do the read or write, and then
release the lock. Note that trying to get either
lock can force the thread to block if the lock is
unavailable.
*************************************************/
void * writerMain(void *arg) {
  int loopCount;

  for (loopCount = 0; loopCount < LOOP_MAX;
     loopCount++) {
```

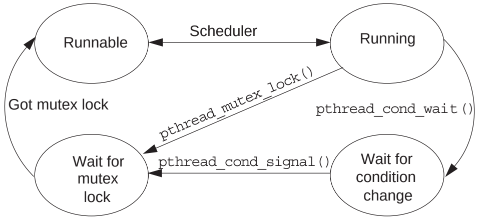
```

    /* Get either the rwlock or the mutex,
    depending on the mode. */
    if (userwlock) {
      pthread_rwlock_wrlock(&rwlock);
    }else {
      pthread_mutex_lock(&mutex);
    }

    doWrite();

    /* Release the lock. */
    if (userwlock) {
      pthread_rwlock_unlock(&rwlock);
    }else {
      pthread_mutex_unlock(&mutex);
    }
  }
}

/***********************************************
This is the main for the reader threads. Get the
```

```
locks as for the writer threads, except specify
that this is a reader and not a writer when getting
the multiple reader single writer lock.
************************************************/
void * readerMain(void * idArg) {
  int id = (int)idArg;/* Arg is thread ID number. */
  int loopCount;

  for (loopCount = 0; loopCount < LOOP_MAX;
     loopCount++) {

    /* Get either the rwlock or the mutex,
       depending on the mode. */
    if (userwlock) {
      pthread_rwlock_rdlock(&rwlock);
    }else {
      pthread_mutex_lock(&mutex);
    }

    doRead();

    /* Release the lock. */
```

```
    if (userwlock) {
      pthread_rwlock_unlock(&rwlock);
    }else {
      pthread_mutex_unlock(&mutex);
    }
  }
}
```

Thread Synchronization Within a Process

Condition Variables



stack.c

```
/****************************************************
This module exemplifies a threadsafe stack. It uses
a mutex to coordinate access to shared data. It
also demonstrates the use of condition variables by
which threads wait for the shared data to change,
and by which threads notify/resume each other when
the state has changed.
/***************************************************/

#include <pthread.h>/* Posix threads. */

#include "stack.h"/* Stack function definitions. */
#include "utils.h"/* for printWithTime(). */

#define STACK_SIZE 3/* Define a small stack size to
        cause contention. */

/* Define the data structure shared between the
  threads. */

```

```
static char buffer[STACK_SIZE];/* Stack's buffer */
static int index = 0; /* Stack's index. */

/* Mutex lock for exclusive data access. */
static pthread_mutex_t mutex =
        PTHREAD_MUTEX_INITIALIZER;

/* Condition var for coordinating threads. */
static pthread_cond_t conditionVar =
        PTHREAD_COND_INITIALIZER;


/**************************************************/
void push(char oneChar) {
/**************************************************
Push a character onto the stack. Return in the
second argument the stack index corresponding to
where the character is pushed.
/**************************************************/
  char string[25];

  /* Get the lock before accessing the shared data
```

```
  in any way. */
  pthread_mutex_lock(&mutex);

  /* Test the data while under mutex protection, to
     see if the stack is pushable.  Don't change the
     data yet as the mutex lock is given up if
     pthread_cond_wait() is called;  a free lock
     implies consistent data. */

  while (index == STACK_SIZE) {
    printWithTime ("push sleeping...\n");
    pthread_cond_wait(&conditionVar, &mutex);
  }

  /* Stack is pushable.  Push the data. */
   buffer[index++] = oneChar;

  sprintf (string,"Pushed:\tchar %c\tindex %d\n",
      oneChar, index-1);
  printWithTime(string);

  /* Notify waiting threads (poppers in this case)
```

```
    that the stack has changed and, due to the
    operation just completed, there is now
    something to pop. */
  pthread_cond_signal(&conditionVar);

  /* Release the mutex. All done with shared data
    access. */
  pthread_mutex_unlock(&mutex);
}

/**************************************************/
char pop() {
/**********************************************
Pop a character from the stack. Return in the
second argument the stack index corresponding to
where the character is popped.
/**************************************************/
  char toReturn;
  char string[25];

  /* Get the lock before accessing the shared data
    in any way. */
```

```
pthread_mutex_lock(&mutex);

/* Test the data while under mutex protection
  to see if the stack is pushable. Don’t change
  the data yet as the mutex lock is given up if
  pthread_cond_wait() is called; a free lock
  implies consistent data. */

while (index == 0) {
  printWithTime ("pop sleeping...\n");
  pthread_cond_wait(&conditionVar, &mutex);
}

/* Stack is poppable.  Pop the data. */
toReturn = buffer[--index];

sprintf (string, "Pop:\tchar %c\tindex %d\n",
    toReturn, index);
printWithTime(string);

/* Notify waiting threads (pushers in this case)
  that the stack has changed and, due to the
```

```
    operation just completed, there is now something
    to push. */
  pthread_cond_signal(&conditionVar);

  /* Release the mutex. All done with shared data
    access. */
  pthread_mutex_unlock(&mutex);

  return toReturn;
}
```

proc_sem.c

```
.
.
/* This is the semaphore which regulates the bank
  tellers. It operates like a mutex but can manage
  more than one resource. */
sem_t bankLine;
.
.
pthread_mutex_t semMutex =
        THREAD_MUTEX_INITIALIZER;
pthread_cond_t semCond = PTHREAD_COND_INITIALIZER;
.
.
```

```
/* Initialize semaphore to manage NUM_TELLERS
quantity of a resource. Set up so that only
threads of this process can use it. */
if (sem_init(&bankLine, THREADS_OF_THIS_PROCESS,
      NUM_TELLERS)) {
  perror ("Error initializing semaphore");
  exit (errno);
}
.
.
/* This is one of those rushed clients. */
.
.
if (inaHurry) {

/* See if a teller is available. If not,
   sem_trywait returns EAGAIN. */
  if ((sem_trywait(&bankLine) == -1)
      && (errno == EAGAIN)) {
...
```

```
  /* If get here, there is a teller available right
     away. Drop thru and do business. */
  } else {

#if REPORT_NUM_SEM_FREE
      sem_getvalue(&bankLine, &availableTellers);
      pthread_mutex_unlock(&semMutex);
      sprintf (string,
        "Client %d is in a hurry and got a
        teller. Count is now %d\n",
        whoAmI, availableTellers);
      printWithTime(string);
#else
      sprintf (string,
        "Client %d%c is in a hurry and got a
        teller.\n", whoAmI);
      printWithTime(string);
#endif
}

```

```
  /* Not in a hurry. */
  } else {
    /* Wait for next available teller. */

#if REPORT_NUM_SEM_FREE

  /* Calling sem_trywait() in a loop until we get
    the lock, and sleeping on the condition
    variable if the lock is unavailable is
    necessary to prevent deadlock, since we have
    the mutex. */

    while ((sem_trywait(&bankLine) == -1)
        && (errno == EAGAIN)) {
      pthread_cond_wait(&semCond, &semMutex);
}
    sem_getvalue(&bankLine, &availableTellers);
    sprintf (string, "Client %d got a teller.
        Count is now %d\n",whoAmI,
        availableTellers);
    printWithTime(string);

```

```
    pthread_mutex_unlock(&semMutex);

#else

    sem_wait(&bankLine);
    sprintf (string,"Client %d got a teller.\n",
        whoAmI);
    printWithTime(string);

#endif
  }

  /* We've got a teller.  Do our business. */
  doBusiness(whoAmI);

#if REPORT_NUM_SEM_FREE
  pthread_mutex_lock(&semMutex);

  /* Done doing business. Teller is available. */
  sem_post(&bankLine);

```

```
  /* Get value of the semaphore to report back as a
     diagnostic. */
  sem_getvalue(&bankLine, &availableTellers);

  pthread_cond_signal(&semCond);
  pthread_mutex_unlock(&semMutex);

  sprintf (string,
      "Client %d done doing business. Available”
      “teller count: %d\n", whoAmI, availableTellers);
  printWithTime(string);
#else

  /* Done doing business.  Teller is available. */
  sem_post(&bankLine);
.
.
```

Using Interprocess Synchronization

- Shared area setup
- Mutex lock initialization
- Cleanup

Using Interprocess Synchronization

```
.
.
/* Shared data is created in this mmapped file. */
#define SHARED_DATA_FILE "/tmp/proc_mutex.dat"

/* Global stuff. */

typedef struct globalStuff {
  pthread_mutex_t sharedMutex;
  int count;
} globalStuff;

/* Handle to shared area. */
struct globalStuff * globalArea;


```

```
/*****************************************************/
void cleanup(int x) {
/*****************************************************
Cleanup handler, called as a signal handler for ^C.
Unlock the mutex before terminating.
/*****************************************************/
  printf ("Cleaning up...\n");
  pthread_mutex_unlock(&globalArea->sharedMutex);
  exit(0);
}
.
.
  /* For interprocess mutex init. */
  pthread_mutexattr_t mutex_attributes;
  /* True if we set up shared area. */
  boolean_t createdHere = B_FALSE;
.
.
```

```

  /* Open the shared area. Err out if it already
     exists (which means we are
     not the first process to use it, and thus not
     to be the creator if it. */
  shared_fd = open(SHARED_DATA_FILE, (O_RDWR |
        O_CREAT | O_EXCL), 0777);

  /* Flag that we are the creator if we are, or
     just open the file otherwise. */
  if ((shared_fd == -1) && (errno == EEXIST)) {
        shared_fd = open(SHARED_DATA_FILE,
        O_RDWR, 0777);
  } else {
    createdHere = B_TRUE;
  }

  /* Some other error opening the file. */
  if (shared_fd < 0) {
    perror ("Error opening shared mutex area");
    exit (errno);
  }
```

```

/* Expand the file to hold the shared data,
   before mmapping it. */
if (createdHere) {
  if (ftruncate(shared_fd,
      sizeof(struct globalStuff))) {
    perror ("ftruncate");
    exit (errno);
  }
}

/* Mmap the file so all processes needing access
   to it can see it. */
globalArea = (struct globalStuff *)mmap(
    NULL, sizeof(struct globalStuff), PROT_READ
    | PROT_WRITE, MAP_SHARED,
    shared_fd, 0);
if (globalArea == (struct globalStuff *)-1) {
  perror ("Error getting shared Mutex virtual
      addr");
  exit (errno);
}
```

```

/* Set up cleanup handler for ^C. */
signal(SIGINT, cleanup);

/* Initialize mutex in shared region if we are
   the shared region creator. */
if (createdHere) {

  /* Initialize mutex attributes list. */
  if (pthread_mutexattr_init(&mutex_attributes)) {
    perror ("pthread_mutexattr_init");
    exit (errno);
  }

  /* Reflect in the attributes that this is an
     interprocess shared mutex. */
  if (pthread_mutexattr_setpshared(
      &mutex_attributes,
      PTHREAD_PROCESS_SHARED)) {
    perror ("pthread_mutexattr_setpshared");
    exit (errno);
  }
```

```

  /* Initialize the mutex in the mmapped area,
     with the attributes above. */
  if (pthread_mutex_init(&globalArea-
      >sharedMutex, &mutex_attributes)) {
    perror ("pthread_mutex_init");
    exit (errno);
  }
.
.
```

System V IPC Semaphores

Command	Functionality	Argument
IPC_RMID	Remove group	Ignored
SETVAL	Set value of particular semaphore	int value*
GETVAL	Get value of particular semaphore	Ignored**
SETALL	Set value of each semaphore in group	ushort_t * array of values*
GETALL	Get value of each semaphore in group	ushort_t * array of values*

* Interpretation of union semun argument.

** Ignored as value is returned in function return.

ipcsem.c

```
/*******************************************************
This program demonstrates the use of an IPC semaphore
in coordinating tasks which require simultaneous
multiple resource allocations, each requesting more
than one of the resource.

Allocate a semaphore to represent the resources of a
moving company with five trucks, 12 human movers,
and an insurance policy of $1 Million to be spread
across all concurrent jobs.

Jobs are done on a first come, first served basis.
A job goes only if there are an adequate number of
movers, trucks and insurance to cover the job,
after all other jobs are accounted for. If the job
is a go, decrement the resource count, so that all
resources in use are accounted for when the next
job is requested.
/*************************************************/

```

```
#include <sys/types.h>/* For general. */
#include <sys/ipc.h>/* System 5 IPC defs. */
#include <sys/sem.h> /* System 5 IPC semaphore
        defs.*/
#include <pthread.h>/* Posix threads. */
#include <stdlib.h>/* Needed for delay to work
        properly. */

#include "utils.h" /* For fractSleep() and
        printWithTime(). */

#define NUM_THREADS 10/* Number of simultaneous
        requests. */
#define TIME_BTWN_NEW_THREADS 0.5/* Time between
        intro of new request. */
#define RUNTIME_RANGE 5.0/* Time of longest job. */

```

```
/* These define a single semaphore group consisting
  of three semaphores. All the semaphores in a
  group can be modified together in a single atomic
  operation. The first semaphore represents the
  number of trucks available; the second: the
  number of movers available; the third: the amount
  of insurance in $1000 units available. */

#define NUM_SEMS_IN_GROUP 3
#define TRUCK_SEM 0
#define MOVER_SEM 1
#define INSUR_SEM 2

/* These are used to initialize semaphores with
   total resources managed. */
#define NUM_TRUCKS 5
#define NUM_MOVERS 12
#define AMT_INSUR 1000

```

```
/* Flags passed thru sembuf structure; wait for
   resource. */
#define WAIT 0

#define STRING_SIZE 80

#define FALSE 0
#define TRUE (!FALSE)



/* Define some jobs here. This table specifies how
  many of each resource is requested per each job.
  The values in this table cause contention: for
  example, the first job uses 4 trucks out of 5
  available, making some of the others wait for a
  truck. */

```

```
struct job { int numTrucks; int numMovers;
        int amtInsurance; } jobTable[] = {
  { 4,5,250},/* Use many trucks */
  { 1,2,500},/* Normal amts. */
  { 3,5,1000},/* Lots of insur */
  { 2,8,250},/* Many movers. */
  };

/* Number of jobs in the table. */
int numJobs = sizeof(jobTable) /
         sizeof (struct job);

int semid;/* IPC semaphore identifier (semaphore
         accessed through this).*/

extern int errno;

void *threadMain(void *);


```

```
/**************************************************/
main() {
/**************************************************
Main function. Allocate and initialize resources,
then spawn threads.
**************************************************/

  /* Array of threads, one per request. */
  pthread_t threads[NUM_THREADS];

  /* Needed to convert constant for semctl call.*/
  int numTrucks = NUM_TRUCKS;
  int numMovers = NUM_MOVERS;
  int amtInsur= AMT_INSUR;

  int count;

  /* Allocate a single semaphore group with three
    semaphores in it. */
```

```
  if ((semid = semget(IPC_PRIVATE,
    NUM_SEMS_IN_GROUP, IPC_CREAT | 0600)) == -1){
    perror ("semget");
    exit (errno);
  }

  /* Initialize each semaphore in the group.
      Could also have used SETALL. */
  if ((semctl(semid, TRUCK_SEM, SETVAL,
        &numTrucks))||(semctl(semid,
        MOVER_SEM, SETVAL, &numMovers)) ||
        (semctl(semid, INSUR_SEM, SETVAL,
        &amtInsur))) {
    perror ("Error initializing semaphores");
    goto cleanup;
  }

/* Initialize random number generator used for time
    delays. */
    srand48(time(NULL));

```

```
  /* Spawn the threads. The argument passed to
      threadMain is a job table index, so multiple
      requests can be made using the same table entry
      when the index wraps. Delay to model jobs being
      staggered instead of coming all at once. */
  for (count = 0; count < NUM_THREADS; count++) {
    if (pthread_create(
      &threads[count], NULL, threadMain,
      (void *)(count % numJobs))) {
    perror ("Error starting reader threads");
    goto cleanup;
    }
  fractSleep(TIME_BTWN_NEW_THREADS);
  }

  /* Wait for threads to finish. */
  for (count = 0; count < NUM_THREADS; count++) {
    pthread_join(threads[count], (void **)NULL);
  }

```

```
cleanup:

  /* Delete the semaphore.  This is not done
      automatically by the system. */
  if (semctl(semid, 0, IPC_RMID, NULL)) {
    perror ("semctl IPC_RMID:");
  }
}


/**************************************************/
  void *threadMain(void * arg) {
  ************************************************
  Here is where a thread starts executing. A
  thread in executing this function represents a
  task for the moving company, and requests the
  resources it needs.
/**************************************************/

```

```
  /* The argument passed in is a table index. */
  int jobNum = (int)arg;

  /* Local string for message composition. */
  char string[STRING_SIZE];

  sprintf (string,
     "Job # %d requesting %d trucks, %d people, "
     "$%d000 insurance...\n",
     jobNum, jobTable[jobNum].numTrucks,
     jobTable[jobNum].numMovers,
     jobTable[jobNum].amtInsurance);
  printWithTime(string);

    /* Get the resources needed.  Wait for them if
    necessary. */
  if (reserve(semid, jobTable[jobNum])) {
    perror ("reserve");
    return (NULL);
  }

```

```
  sprintf (string,
     "Job # %d got %d trucks, %d people"
     ", %d000 insurance and is running...\n",
     jobNum, jobTable[jobNum].numTrucks,
     jobTable[jobNum].numMovers,
     jobTable[jobNum].amtInsurance);
  printWithTime(string);

  /* Delay to simulate the time the resources are
    in use. */
  fractSleep(drand48() * RUNTIME_RANGE);

  sprintf (string,
     "Job # %d done;  returning %d "
     "trucks, %d people, %d000 insurance...\n",
     jobNum, jobTable[jobNum].numTrucks,
     jobTable[jobNum].numMovers,
     jobTable[jobNum].amtInsurance);
  printWithTime(string);

```

```
  /* Release resources. */
  if (release(semid, jobTable[jobNum])) {
    perror ("release");
  }
}


/**************************************************/
int reserve(int semid, struct job thisJob) {
/************************************************
  Reserve resources required for a job. This
  wrapper function passes negative values into
  playWithSemaphores(), since negative values
  represent resource allocations.
/**************************************************/
  return (playWithSemaphores(
     semid,-thisJob.numTrucks, -thisJob.numMovers,
     -thisJob.amtInsurance));
}


```

```
/**************************************************/
int release(int semid, struct job thisJob) {
/**************************************************
  Release resources of a job completed. This
  wrapper function passes positive values into
  playWithSemaphores(), since positive values
  represent resource deallocations.
****************************************************/
  return (playWithSemaphores(
     semid, thisJob.numTrucks, thisJob.numMovers,
     thisJob.amtInsurance));
}


/**************************************************/
static int playWithSemaphores(
   int semid, int numTrucks, int numMovers,
   int amtInsurance) {
/***************************************************
  This is the workhorse function that allocates /
  deallocates resources from the semaphore group.
*****************************************************/
```

```

/* There is one operation per semaphore for this
    example. This allocates an array of semaphore
    operations, all of which is carried out with a
    single atomic operation (system call to
    semop()).*/

  struct sembuf ops[NUM_SEMS_IN_GROUP];

/* One operation per semaphore. Note that a
    negative value to ops[x].sem_op allocates a
    resource, while a positive value releases it. */

  ops[0].sem_num = TRUCK_SEM;
  ops[0].sem_op = numTrucks;
  ops[1].sem_num = MOVER_SEM;
  ops[1].sem_op = numMovers;
  ops[2].sem_num = INSUR_SEM
  ops[2].sem_op = amtInsurance;

```

```
/* All semaphore operations are to be handled in
    the same way. */
  ops[0].sem_flg = ops[1].sem_flg =
      ops[2].sem_flg = WAIT;

/* "The call" that does the work. */
  return (semop(semid, ops, NUM_SEMS_IN_GROUP));
}

```

Exercise: Synchronization

- Objectives
- Tasks
- Discussion
- Solutions

Module 12

Sockets