BOSC2013 OO2 - Asynchronicity and C

Sigurt Bladt Dinesen
sidi@itu.dk

November 18, 2013

Contents

1 Multi-threaded sum

1.1 A faster sum of n square-roots

Rewrite the program in such a way that it will actually run faster, than it would have on a single thread.

This is achieved by letting m threads sum the squares of $\frac{n}{m}$ numbers each in parallel, and then summing the results in linear time afterwards. The advantage of this approach is that the use of mutexes is redundant: Each thread keeps its own tentative sum, the collection of which is summed only after all threads finish.

A more elegant approach would be to run the tentative sums, and then have each thread end by locking a mutex, adding its sum to a common sum variable, and unlocking the mutex. The later solution would also offer a performance increase proportional to the ratio $\frac{n}{m}$, which will usually be large.

To conclude the assignment: The runtime of *sumsqrt* is now $\frac{n}{m} + m$ rather than n as it was initially (provided at least m logical cores of course).

A visual representation of the runtime of *sumsqrt* can be seen in figure 1. It can be seen that on one core, the real time and cpu time spend in user land are equal, on two cores the real time is almost halved, although the user land cpu time increases slightly, probably due to threading overhead. It can also be seen that threads beyond the number of cores does virtually nothing to either times. Kernel mode cpu time is omitted as it is negligible here. Had the program used mutexes, that may not have been the case.

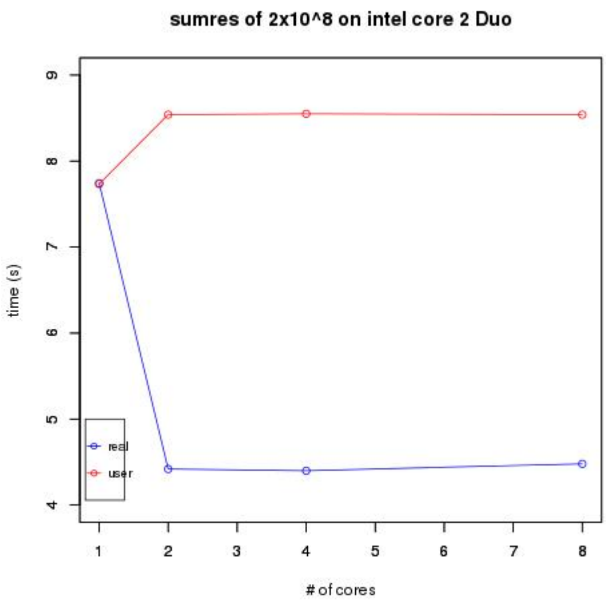

Figure 1: Running sumsqrt on the first 200 million natural numbers on 1, 2, 4, and 8 threads on an 1.6 Ghz Intel core 2 Duo processor. Data was collected over a total of 12 runs.

1.2 Testing

To test *sumsqrt*, first compile by running

```
make sumsqrt
```

in the project root directory. Then *sumsqrt* can be used as

```
sumsqrt [numbers to root and sum] [number of threads]
```

For example

```
sumsqrt 100 4
```

Will calculate the sum of the square roots of numbers 0 though 99, using 4 threads.

2 Multi-threaded FIFO buffer as a linked list

2.1 Parallelized read/write actions

Assume that the list implemetation is used in an asynchronous environment. What issues might occur?

Since *list_remove* and *list_add* work on the same list (albeit on different ends of it), raise conditions would occur if different threads of execution were to run these methods simultaneously. In fact, different threads running just *one* of these methods would be problematic. The trivial example is that of reading from, and writing to, a variable, e.g. to increment its value by a constant.

The execution of such example is illustrated in figure 1, where T_2 sets the shared entity to the result applying a function to it, regardless of changes made by T_1. After execution, S is the result of applying $T_2.f$ to α, when it should have been the result of applying $T_2.f$ to α'. The thing to notice here is that

$$T_2.f(\alpha) \neq T_2.f(\alpha') = T_2.f(T_1.f(\alpha))$$

T_1 *action*	T_2 *action*	S *value*
		α
$T_1.a = S$		α
	$T_2.a = S$	α
$S = T_1.f(S) = \alpha'$		$T_1.f(\alpha)$
	$S = T_2.f(S)$	$T_2.f(\alpha)$

Table 1: Execution of two threads on a shared entity. The figure shows the actions of two threads, T_1 and T_2, operating on a shared value S.

Similar problems would occur for the next variables of the list's nodes.

2.2 Thread Safety

Use a mutex to make a thread-safe version of list.c

This is achieved quite simply by using the C pthread library to take a lock before either adding or removing elements in the buffer, and releasing it when done. Packaging all write operations on the list into a critical section ensures that the situation in table 1 can never occur. Note that *add* and *remove* uses the same mutex. This is necessary as the two functions sometimes will operate on the same variables.

3 Producer-Consumer with a bounded buffer

The producer-controller is implemented with the thread safe list implementation from last section. This is not enough however, as we wish to ensure *list_add* and *list_remove* operations will wait if the list is full or empty, respectively. This demand is met by the use of two semaphores, one for adding (*empty*) and one for removing (*full*). When a producer wishes to add a product, preforms *sem_wait* on *empty*, to ensure there are empty slots in the buffer, and then performs a *sem_post* on *full* to indicate consumers that there is one more full slot. Consumer threads do the opposite, that is *sem_post* is performed on *empty* and *sem_wait* on *full*. The *full* semaphore is initialized to 0, and *empty* to the desired buffer size. This corresponds to the notion that *full* represents occupied buffer slots (consumabla slots), and *empty* represent empty slots (available to the producers).

3.1 Termination

Make sure the program does not terminate before all products are produced and consumed

This is done by keeping the thread ids for consumer and producer threads in two lists, and joining on them all in the end of the programs main method. This guarantees that program terminates only when all consumers and producers are done. The number of product consumed or produced by each thread is determined at runtime to be the ratio between the number of products and threads. This ensures that all produced products will be consumed, regardless of whether the number of producers and consumers are equal.

3.2 Testing

To test the producer-consumer program (and with it the FIFO buffer) compile by running

```
make prodcons
```

in the project root directory. Then run prodcons as

```
prodcons [number of prodcer threads] [number of consumer threads]
[buffer size] [number of products]
```

For example,

```
./prodcons 10 10 10 20
```

Will run the producer-consumer program with 10 producing threads, each producing 2 products, 10 consuming threads, each consuming 2 products, a buffer with 10 slots, producing a total of 20 products.

4 Banker's algorithm

4.1 C Matrix Allocation

Allocate memory for the state's vectors and matrices dynamically

The rows of each matrix are allocated with *malloc*, as C does not have a syntax for allocating nested arrays. This memory is never explicitly freed, as the arrays are used until program termination anyway.

4.2 State-safety

Use the Banker's safety algorithm to determine whether a state is safe

The function *resource_request* implements both the request and safety algorithm. It takes an array as argument, representing a resource allocation request, and then runs the safety algorithm as if the request had already been granted.

The *resource_request* and *resource_release* functions both operates on the same matrices, and therefore lock the same mutex before writing to them to avoid race conditions. Using the same mutex ensures that deadlocks can never occur, as a deadlock needs a minimum of two locks to wait on.

Before starting the requesting threads, the program runs the safety algorithm with a request of 0 of all resources, thus ensuring that the current state (equivalent to the state that results of requesting nothing) is safe.

5 Appendix/code

5.1 Multi-threaded sum

sumsqrt.c

```c
#include <pthread.h>
#include <stdio.h>
#include <math.h>

typedef struct runpars{
        int lower;
        int upper;
        float res;
} runpars;

void *runner(void *params);
void runThread(pthread_t *tidP, runpars *params);
void spawnNThreads(int n, runpars pars[]);

int main(int argc, char* argv[]){
        if(argc != 3){
                fprintf(stderr, "usage: sumsqrt [n] [threads]\n");
                return -1;
        }
        if(atoi(argv[1]) < 0) {
                fprintf(stderr,"%d - n must be >= 0\n", atoi(argv[1]));
                return -1;
        }
        if(atoi(argv[2]) <= 0) {
                fprintf(stderr,"%d - threads must be = 0\n", atoi(argv[2]));
                return -1;
        }

        int upper = atoi(argv[1]);
        int nThreads = atoi(argv[2]);
        int stepSize = upper/nThreads;
        struct runpars pars[nThreads];
        int i;
        for (i=0; i<nThreads; i++){
                int first = i * stepSize;
                int last = (i == nThreads-1) ? upper : (i+1) * stepSize -1;
                pars[i] = (runpars) {.lower = first, .upper = last, .res = 0.0};
                //printf("%d to %d\n", pars[i].lower, pars[i].upper);
        }
```

```
        spawnNThreads(nThreads, pars);
        float sum = 0.0;
        for (i=0; i<nThreads; i++){
                sum += pars[i].res;
        }
        printf("sum of squares 0 though %d = %f\n", upper
            , sum);

        return 0;
}

void spawnNThreads(int n, runpars pars[]){
        pthread_t ids[n];
        int i;
        for (i=0; i<n; i++){
                runThread(&(ids[i]), &(pars[i]));
        }
        for (i=0; i<n; i++)
                pthread_join(ids[i], NULL);
}

void runThread(pthread_t *tidP, runpars *params){
        pthread_attr_t attrP;
        pthread_attr_init(&attrP);
        pthread_create(tidP, &attrP, runner, params);
        pthread_attr_destroy(&attrP);
}

void *runner(void *params){
        struct runpars* pars = (struct runpars*) params;
        int i, upper = pars->upper;
        for(i = pars->lower; i <= upper; i++){
                pars->res += sqrtf(i);
        }
        pthread_exit(0);
}
```

5.2 Multi-threaded FIFO buffer as a linked list

list/main.c

```
/*****************************************************************************

         main.c

         Implementation of a simple FIFO buffer as a
             linked list defined in list.h.

*****************************************************************************/
```

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include "list.h"
#include "assert.h"

// FIFO list;
List *fifo;
const int elmsPerThread = 500;
const int numberOfThreads = 3;

void *asyncRunner(void* peh){
        int i;
        for(i=0; i<elmsPerThread; i++){
                list_add(fifo, node_new());
        }
        pthread_exit(NULL);
}
void asyncListTest(){ //this test is fake and gay, and not unit-like at all
        fifo = list_new();
        pthread_t *ids[numberOfThreads];
        int i;
        for(i=0; i<numberOfThreads; i++){
                pthread_t* tid = (pthread_t *) malloc(sizeof(pthread_t));
                ids[i] = tid;
                pthread_create(tid, NULL, asyncRunner, NULL);
        }
        for(i=0; i<numberOfThreads; i++){
                pthread_join(*ids[i], NULL);
        }
        int removeN=0;
        int actualN=0;
        int assertedN = fifo->len;
        Node *n = fifo->first->next;
        while(n != NULL) {
                actualN++;
                n = n->next;
        }

        n = list_remove(fifo);
        while(n != NULL){
                removeN++;
                n = list_remove(fifo);
        }
        //printf("asserted: %i\nremoved: %i\nactual: %i\nelmsPer: %i\n#of threads: %i\nprod: %i\n",
```

```
		assertedN, removeN, actualN, elmsPerThread, numberOfThreads, elmsPerThread*numberOfThreads);
	assert(assertedN == actualN);			//fifo->len is the same as the actual length
	assert(removeN == actualN);			//remove will yield as many elms as there actually are in the list
	assert(elmsPerThread*numberOfThreads == actualN);	//there are actually the expected number of elements
	assert(fifo->len == 0);				//the list is now empty
}

int main(int argc, char* argv[])
{
	asyncListTest();
	return 0;
	fifo = list_new();

	list_add(fifo, node_new_str("s1"));
	list_add(fifo, node_new_str("s2"));

	Node *n1 = list_remove(fifo);
	if (n1 == NULL) { printf("Error no elements in list\n"); exit(-1);}
	Node *n2 = list_remove(fifo);
	if (n2 == NULL) { printf("Error no elements in list\n"); exit(-1);}
	printf("%s\n%s\n", n1->elm, n2->elm);

	return 0;
}
```

list/list.h

```
/******************************************************************************

   list.h

   Header file with definition of a simple linked list.

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

#ifndef _LIST_H
#define _LIST_H
```

```

/* structures */
typedef struct node {
  void *elm; /* use void type for generality; we cast the element's type to void type */
  struct node *next;
} Node;

typedef struct list {
  int len;
  Node *first;
  Node *last;
} List;

/* functions */
List *list_new(void);            /* return a new list structure */
void list_add(List *l, Node *n); /* add node n to list l as the last element */
Node *list_remove(List *l);      /* remove and return the first element from list l*/
Node *node_new(void);            /* return a new node structure */
Node *node_new_str(char *s);     /* return a new node structure, where elm points to new copy of string s */

#endif
```

list/main.c

```
/*******************************************************************************

          main.c

          Implementation of a simple FIFO buffer as a linked list defined in list.h.

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

#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include "list.h"
#include "assert.h"

// FIFO list;
List *fifo;
const int elmsPerThread = 500;
const int numberOfThreads = 3;
```

```c

void *asyncRunner(void* peh){
	int i;
	for(i=0; i<elmsPerThread; i++){
		list_add(fifo, node_new());
	}
	pthread_exit(NULL);
}
void asyncListTest(){ //this test is fake and gay, and not unit-like at all
	fifo = list_new();
	pthread_t *ids[numberOfThreads];
	int i;
	for(i=0; i<numberOfThreads; i++){
		pthread_t* tid = (pthread_t *) malloc(sizeof(pthread_t));
		ids[i] = tid;
		pthread_create(tid, NULL, asyncRunner, NULL);
	}
	for(i=0; i<numberOfThreads; i++){
		pthread_join(*ids[i], NULL);
	}
	int removeN=0;
	int actualN=0;
	int assertedN = fifo->len;
	Node *n = fifo->first->next;
	while(n != NULL) {
		actualN++;
		n = n->next;
	}

	n = list_remove(fifo);
	while(n != NULL){
		removeN++;
		n = list_remove(fifo);
	}
	//printf("asserted: %i\nremoved: %i\nactual: %i\nelmsPer: %i\n#of threads: %i\nprod: %i\n", assertedN, removeN, actualN, elmsPerThread, numberOfThreads, elmsPerThread*numberOfThreads);
	assert(assertedN == actualN);                    //fifo->len is the same as the actual length
	assert(removeN == actualN);                      //remove will yield as many elms as there actually are in the list
	assert(elmsPerThread*numberOfThreads == actualN);    //there are actually the expected number of
```

```
elements
        assert(fifo->len == 0);                                   //the list is now empty
}

int main(int argc, char* argv[])
{
        asyncListTest();
        return 0;
        fifo = list_new();

        list_add(fifo, node_new_str("s1"));
        list_add(fifo, node_new_str("s2"));

        Node *n1 = list_remove(fifo);
        if (n1 == NULL) { printf("Error no elements in list\n"); exit(-1);}
        Node *n2 = list_remove(fifo);
        if (n2 == NULL) { printf("Error no elements in list\n"); exit(-1);}
        printf("%s\n%s\n", n1->elm, n2->elm);

        return 0;
}
```

list/Makefile

```
all: fifo liblist.a

OBJS = list.o main.o
LIBS= -lpthread
CFLAGS= "-ggdb"

fifo: ${OBJS}
        gcc -o $@ ${OBJS} ${LIBS}

liblist.a: list.o
        ar rcs liblist.a list.o

clean:
        rm -rf *o fifo
```

5.3 Producer-Consumer with a bounded buffer

prodcons.c

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

```
#include <stdio.h>
#include <stdlib.h>
#include "list/list.h"

typedef struct runpars{
        int start;
        int end;
} runpars;

void *producer(void *params);
void *consumer(void *params);
void runThread(pthread_t *tidP, runpars *params);
pthread_t* spawnNThreads(int n, void*(*fun) (void *));
void sleeper(float waitMs);
int prodN;
int consN;
int bufferSize;
int productsN;
int producerLoad;
int consumerLoad;
float SLEEP_TIME_MS = 1.0;
List *buffer;
sem_t empty, full;

int main(int argc, char* argv[]){
        setbuf(stdout,NULL);
        if(argc != 5){
                fprintf(stderr, "usage: prodcons [procucers] [consumers] [bufferSize] [products]\n");
                return -1;
        }
        if(atoi(argv[1]) <= 0) {
                fprintf(stderr,"%d - producers must be > 0\n", atoi(argv[1]));
                return -1;
        }
        if(atoi(argv[2]) <= 0) {
                fprintf(stderr,"%d - consumers must be > 0\n", atoi(argv[2]));
                return -1;
        }
        if(atoi(argv[3]) <= 0) {
                fprintf(stderr,"%d - bufferSize must be > 0\n", atoi(argv[3]));
                return -1;
        }
        if(atoi(argv[4]) <= 0) {
                fprintf(stderr,"%d - products must be > 0\n", atoi(argv[4]));
```

```
                return -1;
        }

        buffer = list_new();
        prodN = atoi(argv[1]);
        consN = atoi(argv[2]);
        bufferSize = atoi(argv[3]);
        productsN = atoi(argv[4]);
        producerLoad = productsN / prodN;
        consumerLoad = productsN / consN;
        printf("Starting! Here are the stats\n"
                        "prodN: %i\n"
                        "consN: %i\n"
                        "buffer: %i\n"
                        "products: %i\n"
                        "produderload: %i\n"
                        "consumerload: %i\n",
                        prodN, consN, bufferSize, productsN, producerLoad, consumerLoad);

        sem_init(&full, 0, 0);
        sem_init(&empty, 0, bufferSize);
        srand(time(NULL));

        pthread_t *consIds = spawnNThreads(consN, consumer);
        pthread_t *prodIds = spawnNThreads(prodN, producer);
        int i;
        for(i=0; i<prodN; i++){
                pthread_join(prodIds[i], NULL);
        }
        for(i=0; i<consN; i++){
                pthread_join(consIds[i], NULL);
        }
        free(prodIds);
        free(consIds);
        return 0;
}

pthread_t* spawnNThreads(int n, void *(*fun)(void *)){
        pthread_t *ids = (pthread_t*) malloc(sizeof(pthread_t)*n);
        int i;
        for (i=0; i<n; i++){
                struct runpars *par = malloc(sizeof(runpars));
                if(fun==producer){
                par->start = producerLoad*i;
```

```
			par->end = i==n-1 ? productsN -1 : (producerLoad * (i+1))-1;
			printf("produce steps: %i - %i\n", par->start, par->end);
			}else{
			par->start = consumerLoad*i;
			par->end = i==n-1 ? productsN -1 : (consumerLoad * (i+1))-1;
			printf("consume steps: %i - %i\n", par->start, par->end);
			}
			pthread_create(&ids[i], NULL, fun, par);
		}
		return ids;
}

void *producer(void *params){
		struct runpars* pars = (struct runpars*) params;
		int i;
		for(i = pars->start; i < pars->end; i++){
			Node *n = node_new();
			n->elm = (void*) i;
			sem_wait(&empty);
			list_add(buffer, n);
			sem_post(&full);
			sleeper(SLEEP_TIME_MS);
			int fullC,emptyC;
			sem_getvalue(&full,&fullC);
			sem_getvalue(&empty,&emptyC);
			printf("producer %i made item %i, full: %i, empty: %i\n", pars->start/producerLoad, (int) n->elm, fullC, emptyC);
		}
		free(pars);
		pthread_exit(0);
}

void *consumer(void *params){
		struct runpars* pars = (struct runpars*) params;
		int i;
		for(i = pars->start; i < pars->end; i++){
			sem_wait(&full);
			Node *n = list_remove(buffer);
			sem_post(&empty);
			sleeper(SLEEP_TIME_MS);
			int fullC,emptyC;
			sem_getvalue(&full,&fullC);
			sem_getvalue(&empty,&emptyC);
			printf("consumer ate item %i, full: %i,
```

```
                     empty: %i\n", (int) n->elm, fullC, emptyC);
        }
        free(pars);
        pthread_exit(0);
}

void sleeper(float waitMs){
        //random value between 0 and waitMs
        waitMs = (float)rand() * waitMs / (float)RAND_MAX;
        usleep((int)waitMs * 1000); //times 1000 for microseconds;
}
```

makefile

```
prodcons: prodcons.o
        gcc -g -static -Wall -I./list -L./list -o $@ prodcons.o -llist -lpthread

report: report/report.pdf
report/report.pdf: report/report.tex
        pdflatex --output-directory report/tmp report/report.tex
        pdflatex --output-directory report/tmp report/report.tex #horribad, use latexmk or latex-MK instead
        mv report/tmp/report.pdf report/report.pdf

sumsqrt: sumsqrt.o
        gcc -ggdb sumsqrt.o -o sumsqrt -lpthread -lm

sumsqrt.o: sumsqrt.c
        gcc -ggdb -c sumsqrt.c

clean:
        @@rm prodcons prodcons.o sumsqrt sumsqrt.o report/tmp/* report/report.pdf 2>/dev/null
```

5.4 Banker's Algorithm

banker.c

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/time.h>
#include <pthread.h>

typedef struct state {
```

```
  int *resource;
  int *available;
  int **max;
  int **allocation;
  int **need;
} State;

// Global variables
int m, n;
State *s = NULL;
pthread_mutex_t mutex;

// Mutex for access to state.
pthread_mutex_t state_mutex;

void printavail(){
  printf("Avaialble: [");
  int i;
  for(i=0; i<n; i++){
    printf("%i,", s->available[i]);
  }
  puts("]");
}

/* Random sleep function */
void Sleep(float wait_time_ms)
{
  // add randomness
  wait_time_ms = ((float)rand())*wait_time_ms / (float)
      RAND_MAX;
  usleep((int) (wait_time_ms * 1e3f)); // convert from ms
       to us
}

//return 1 if a is <= b for all a[i] b[i], else return 0
int arrayLE(int* a, int* b, int len)
{
  int i;
  for(i=0; i<len; i++){
    if(a[i] > b[i]){
      return 0;
    }
  }
  return 1;
}

/* Allocate resources in request for process i, only if
    it
   results in a safe state and return 1, else return 0 */
int resource_request(int pi, int *request)
```

```
{
  if(!arrayLE(request, s->available, n))
    return 0;

  int work[n];
  int finish[m];
  int tmpNeed[n];
  int i;
  for(i=0;i<m;i++){
    finish[i]=0;
  }
  for(i=0;i<n;i++){
    work[i]=s->available[i] - request[i];
  }
  for(i=0;i<n;i++){
    tmpNeed[i]=s->need[pi][i] - request[i];
  }
  int k=0;
  while(1){
    while(!(!finish[k] && arrayLE(tmpNeed, work, n)) && k<m){
      k++;
    }

    if(!(!finish[k] && arrayLE(tmpNeed, work, n))){
      break;
    }

    for(i=0; i<n; i++){
      work[i] = work[i] + s->allocation[pi][k] + request[i];
    }
    finish[k]=1;
  }
  k=1;
  for(i=0; i<m; i++){
    if (!finish[i])
      k=0;
    break;
  }
  if(k){
    pthread_mutex_lock(&mutex);
    for(i=0; i<n; i++){
      s->allocation[pi][i] = s->allocation[pi][i] + request[i];
      s->available[i] = s->available[i] - request[i];
      s->need[pi][i] = s->need[pi][i] - request[i];
    }
    printavail();
    pthread_mutex_unlock(&mutex);
```

```
  }
  return k;
}

/* Release the resources in request for process i */
void resource_release(int i, int *request)
{
  int j;
  pthread_mutex_lock(&mutex);
  for(j=0; j<n; j++){
    s->available[j] = s->available[j] + request[j];
    s->allocation[i][j] = s->allocation[i][j] - request[j];
    s->need[i][j] = s->need[i][j] + request[j];
  }
  pthread_mutex_unlock(&mutex);
}

/* Generate a request vector */
void generate_request(int i, int *request)
{
  int j, sum = 0;
  while (!sum) {
    for (j = 0;j < n; j++) {
      request[j] = s->need[i][j] * ((double)rand())/ (double)RAND_MAX;
      sum += request[j];
    }
  }
  printf("Process %d: Requesting resources.\n",i);
}

/* Generate a release vector */
void generate_release(int i, int *request)
{
  int j, sum = 0;
  while (!sum) {
    for (j = 0;j < n; j++) {
      request[j] = s->allocation[i][j] * ((double)rand()) / (double)RAND_MAX;
      sum += request[j];
    }
  }
  printf("Process %d: Releasing resources.\n",i);
}

/* Threads starts here */
void *process_thread(void *param)
{
  /* Process number */
```

```
  int i = (int) (long) param, j;
  /* Allocate request vector */
  int *request = malloc(n*sizeof(int));
  while (1) {
    /* Generate request */
    generate_request(i, request);
    while (!resource_request(i, request)) {
      /* Wait */
      Sleep(100);
    }
    /* Generate release */
    generate_release(i, request);
    /* Release resources */
    resource_release(i, request);
    /* Wait */
    Sleep(1000);
  }
  free(request);
}

int main(int argc, char* argv[])
{
  pthread_mutex_init(&mutex, NULL);

  /* Get size of current state as input */
  int i, j;
  printf("Number of processes: ");
  scanf("%d", &m);
  printf("Number of resources: ");
  scanf("%d", &n);

  int res[n];
  int avail[n];
  int **max[m];
  int **alloc[m];
  int **need[m];

  for(i=0; i<m; i++){
    max[i] = malloc(sizeof(int)*n);
    alloc[i] = malloc(sizeof(int)*n);
    need[i] = malloc(sizeof(int)*n);
  }

  State p = (State)
  {
    .resource = (int*) res,
    .available = (int*) avail,
    .max = (int**) max,
    .allocation = (int**) alloc,
    .need = (int**) need
```

```
  };
  s = &p;
  /* Get current state as input */
  printf("Resource vector: ");
  for(i = 0; i < n; i++)
    scanf("%d", &s->resource[i]);
  printf("Enter max matrix: ");
  for(i = 0;i < m; i++)
    for(j = 0;j < n; j++){
      scanf("%d", &s->max[i][j]);
    }
  printf("Enter allocation matrix: ");
  for(i = 0; i < m; i++)
    for(j = 0; j < n; j++) {
      scanf("%d", &s->allocation[i][j]);
    }
  printf("\n");

  /* Calcuate the need matrix */
  for(i = 0; i < m; i++)
    for(j = 0; j < n; j++)
      s->need[i][j] = s->max[i][j] - s->allocation[i][j];

  /* Calcuate the availability vector */
  for(j = 0; j < n; j++) {
    int sum = 0;
    for(i = 0; i < m; i++)
      sum += s->allocation[i][j];
    s->available[j] = s->resource[j] - sum;
  }

  /* Output need matrix and availability vector */
  printf("Need matrix:\n");
  for(i = 0; i < n; i++)
    printf("R%d ", i+1);
  printf("\n");
  for(i = 0; i < m; i++) {
    for(j = 0; j < n; j++)
      printf("%d  ",s->need[i][j]);
    printf("\n");
  }
  printf("Availability vector:\n");
  for(i = 0; i < n; i++)
    printf("R%d ", i+1);
  printf("\n");
  for(j = 0; j < n; j++)
    printf("%d  ",s->available[j]);
  printf("\n");

  /* If initial state is unsafe then terminate with error
```

```
      */
  int req[n];
  for(i=0; i<n; i++)
    req[i]=0;

  for(i=0; i<m; i++)
    if (!resource_request(i, req))
      exit -1;

  /* Seed the random number generator */
  struct timeval tv;
  gettimeofday(&tv, NULL);
  srand(tv.tv_usec);

  /* Create m threads */
  pthread_t *tid = malloc(m*sizeof(pthread_t));
  for (i = 0; i < m; i++)
    pthread_create(&tid[i], NULL, process_thread, (void *) (long) i);

  /* Wait for threads to finish */
  pthread_exit(0);
  free(tid);

  /* Free state memory */
}

// vim: set ts=2 sw=2 et:
```

input.txt

```
4
3

9 3 6

3 2 2
6 1 3
3 1 4
4 2 2

0 0 0
0 0 0
0 0 0
0 0 0
```

banker/Makefile

```
all: banker

OBJS = banker.o
```

```
LIBS= -lpthread
CFLAGS= "-ggdb"

banker: ${OBJS}
        gcc -o $@ ${OBJS} ${LIBS}

clean:
        rm -rf *o banker
```