System and Device Programming

Synchronization Exercises (part A)

Stefano Quer
Dipartimento di Automatica e Informatica
Politecnico di Torino

Exercise

- Use a conditional variable to implement the following behavior
  - A program creates 3 threads
  - Thread 1 and thread 2
    - Update a variable (**count**), increasing it at each iteration
  - Thread 3
    - Awaits until the variable (**count**) reaches a specified value
    - When the value is reached, it displays a message on standard output

Solution

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

#define NUM_THREADS  3
#define COUNT_LIMIT 12

typedef struct cond_s {
  pthread_mutex_t lock;
  pthread_cond_t cond;
  int count;
} cond_t;
```

Conditional variable + protection mutex + condition (counter)

Solution

```
int main () {
  cond_t cond_d;
  pthread_t tids[3];
  setbuf (stdout, 0);
  pthread_mutex_init (&cond_d.lock, NULL);
  pthread_cond_init (&cond_d.cond, NULL);
  cond_d.count = 0;
  pthread_create (&tids[0],NULL,inc,(void *) &cond_d);
  pthread_create (&tids[1],NULL,inc,(void *) &cond_d);
  pthread_create (&tids[2],NULL,watch,(void *) &cond_d);
  pthread_join (tids[0], NULL);
  pthread_join (tids[1], NULL);
  pthread_join (tids[2], NULL);
  pthread_exit(0);
}
```

Conditional variable & mutex initialization

Run threads

Wait for threads

Solution

```
static void *inc (void *args) {
  cond_t *cond_d = (cond_t *) args;
  int stop = 0;
  while (!stop) {
    pthread_mutex_lock (&cond_d->lock);
    cond_d->count++;
    printf ("Increment count [T=%ld] (%d -> %d)\n",
      (long int)pthread_self(),cond_d->count-1,cond_d->count);
    if (cond_d->count == COUNT_LIMIT) {
      pthread_cond_signal (&cond_d->cond);
      printf ("Threshold reached (count=%d).\n", cond_d->count);
    }
    if (cond_d->count >= COUNT_LIMIT) stop = 1;
    pthread_mutex_unlock (&cond_d->lock);
  }
  pthread_exit(0);
}
```

Protect CS

Increase counter

Signal conditional variable

Release CS

When the limit is reached stop the while loop

Solution

```
static void *watch (void *args) {
  cond_t *cond_d = (cond_t *) args;

  printf ("Starting watch on count\n");

  pthread_mutex_lock (&cond_d->lock);
  if (cond_d->count < COUNT_LIMIT) {
    pthread_cond_wait (&cond_d->cond, &cond_d->lock);
    printf ("Watch_count: Condition signal received.\n");
  }
  pthread_mutex_unlock (&cond_d->lock);

  pthread_exit(0);
}
```

Protect CS

while

Wait on conditional variable

Release CS

Output

The generated output

```
Starting watch on count
Increment count [T=139696519681792] (0 -> 1)
Increment count [T=139696528074496] (1 -> 2)
Increment count [T=139696519681792] (2 -> 3)
Increment count [T=139696528074496] (3 -> 4)
Increment count [T=139696519681792] (4 -> 5)
Increment count [T=139696528074496] (5 -> 6)
Increment count [T=139696519681792] (6 -> 7)
Increment count [T=139696528074496] (7 -> 8)
Increment count [T=139696519681792] (8 -> 9)
Increment count [T=139696528074496] (9 -> 10)
Increment count [T=139696519681792] (10 -> 11)
Increment count [T=139696528074496] (11 -> 12)
Threshold reached (count=12).
Watch_count: Condition signal received.
Increment count [T=139696519681792] (12 -> 13)
```

One more increment (by the "other thread") before exiting the while cycle

Exercise

- Implement a Producer-Consumer scheme with an **unbounded** buffer
  - The main thread runs NP producers and NC consumers
  - Producers read strings from an input file
  - Consumers display strings on the output device
  - Producers and consumers communicate using as a buffer a LIFO list
    - The list is implemented with dynamic elements, fields, and pointers
  - Use condition variables to perform synchronization

Solution

Premises

```
#include <stdio.h>
...
#include "pthread.h"

#define N 100

typedef struct stack_e {
  char *id;
  /* ... more stuff here ... */
  struct stack_e *next;
} stack_t;

stack_t *stack_d = NULL;
FILE *fp;
pthread_cond_t  cv = PTHREAD_COND_INITIALIZER;
pthread_mutex_t m  = PTHREAD_MUTEX_INITIALIZER;
```

Stack (data + pointer)

Global objects, mutex and condition variable initialization

Solution

Producer

```
while (1) {
  pthread_mutex_lock (&m);
  fscanf (fp, "%s", str);
  tmp = malloc (1 * sizeof (stack_t));
  if (tmp==NULL) exit (1);
  tmp->id = strdup (str);
  tmp->next = stack_d;
  stack_d = tmp;
  pthread_mutex_unlock (&m);
  pthread_cond_signal (&cv);
  if (strcmp (str, "end")==0) {
    fseek (fp, -4, SEEK_END);
    break;
  }
}
```

Begin of CS

Read from file

Push data into the LIFO stack

End of CS

Let "end" be available to stop all consumers

Solution

Consumer

Begin of CS

```
while (1) {
  pthread_mutex_lock(&m);
  while (stack_d == NULL)
    pthread_cond_wait (&cv, &m);
  tmp = stack_d;
  stack_d = stack_d->next;
  strcpy (str, tmp->id);
  free (tmp->id);
  free (tmp);
  pthread_mutex_unlock(&m);
  fprintf (stdout, "Pop element %s: %s\n", str);
  if (strcmp (str, "end")==0)
    break;
}
```

End of CS

Pop the data from LIFO stack

Try to stop in a clean way all consumers

Exercise

- Implement a Producer-Consumer scheme with an **bounded** buffer
  - The main thread runs NP producers and NC consumers
  - Producers read strings from an input file
  - Consumers display strings on the output device

Simplified:
in and **out** N integers

  - Producers and consumers communicate using a buffer of size 1 (a variable)
  - Use condition variables to perform synchronization

Solution 1

Buffer

Premises

Initially empty

```
int buffer;
int count = 0;

pthread_cond_t  cv = PTHREAD_COND_INITIALIZER;
pthread_mutex_t m  = PTHREAD_MUTEX_INITIALIZER;

void enqueue (int value) {
  assert (count==0);
  count = 1;
  buffer = value;
}

int dequeue () {
  assert (count==1);
  count = 0;
  return (buffer);
}
```

Solution 1

```
void *producer(void *arg) {
  int i;
  int loops = int (args):
  for (i=0; i<loops; i++) {
    enqueue (i);
  }
}
```

Producer and Consumer
with NO protection

```
void *consumer(void *arg) {
  int loops = int (args):
  for (i=0; i<loops; i++) {
    int tmp = dequeue ();
    printf ("%d", tmp);
}
```

Logic scheme
Need synchronization

Solution 2

Producer and Consumer
1 producer and 1 consumer

```
void *producer(void *arg) {
  int i;
  int loops = int (args):
  for (i=0; i<loops; i++) {
    pthread_mutex_lock (&m);
    if (count==1)
      pthread_cond_wait(&cv, &m);
    enqueue (i);
    pthread_cond_signal(&cv);
    pthread_mutex_unlock(&m);
  }
}
```

```
void *consumer(void *arg) {
  int i;
  int loops = int (args):
  for (i=0; i<loops; i++) {
    pthread_mutex_lock (&m);
    if (count==0)
      pthread_cond_wait(&cv, &m);
    int tmp = dequeue (i);
    pthread_cond_signal (&cv);
    pthread_mutex_unlock (&m);
    printf ("%d", tmp);
  }
}
```

Broken scheme
(2 problems)

Solution 2

- With more than 1 producer and/or more than 1 consumer the previous solution has a problem
- For example, with 2 consumers
  - The second consumer can enter the CS after the first consumer has been woken up from the wait but before is lock the mutex
  - One of the two will read a wrong data
- The problem is due to the fact that the status can change even after a thread wakes up from a wait

Solution 3

```
void *producer(void *arg) {
  int i;
  int loops = int (args):
  for (i=0; i<loops; i++) {
    pthread_mutex_lock (&m);
    while (count==1)
      pthread_cond_wait(&cv, &m);
    enqueue (i);
    pthread_cond_signal(&cv);
    pthread_mutex_unlock(&m);
  }
}
```

Producer and Consumer
NP producer and NC
consumer

```
void *consumer(void *arg) {
  int i;
  int loops = int (args):
  for (i=0; i<loops; i++) {
    pthread_mutex_lock (&m);
    while(count==0)
      pthread_cond_wait(&cv, &m);
    int tmp = dequeue (i);
    pthread_cond_signal (&cv);
    pthread_mutex_unlock (&m);
    printf ("%d", tmp);
  }
}
```

Broken scheme
(1 problem)

Solution 3

- There is still one problem
  - There is only 1 condition variable
  - A producer (consumer) can wake-up another consumer (producer)

Solution 4

```
void *producer(void *arg) {
  int i;
  int loops = int (args):
  for (i=0; i<loops; i++) {
    pthread_mutex_lock (&m);
    while (count==1)
      pthread_cond_wait(&empty, &m);
    enqueue (i);
    pthread_cond_signal(&full);
    pthread_mutex_unlock(&m);
  }
}
```

Producer and Consumer
NP producer and NC
consumer

```
void *consumer(void *arg) {
  int i;
  int loops = int (args):
  for (i=0; i<loops; i++) {
    pthread_mutex_lock (&m);
    while(count==0)
      pthread_cond_wait(&full, &m);
    int tmp = dequeue (i);
    pthread_cond_signal (&empty);
    pthread_mutex_unlock (&m);
    printf ("%d", tmp);
  }
}
```

Correct scheme

Exercise

- Implement the First Reader-Writer (with the reader precedence) scheme using
  - Mutexes
  - Condition Variables
  - Read-Write Locks

Solution (with mutexes)

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

#define N 20

typedef struct rw_s {
  int nr;
  pthread_mutex_t meR;
  pthread_mutex_t meW;
  pthread_mutex_t w;
} rw_t;
rw_t *rw;
```

With mutexes we have a 1:1 correspondence with the high-level (pseudo-code) solution

Number of concurrent readers and writers

nr, meR, meW, w (see high-level solution)

Solution (with mutexes)

```
int main (void) {
  pthread_t th_a, th_b;
  int i, v[N];
  setbuf (stdout, NULL);
  rw = (rw_t *) malloc (1 * sizeof(rw_t));
  rw->nr = 0;
  pthread_mutex_init (&rw->meR, NULL);
  pthread_mutex_init (&rw->meW, NULL);
  pthread_mutex_init (&rw->w, NULL);
  for (i=0; i<N; i++) {
    v[i] = i;
    pthread_create (&th_a, NULL, reader, (void *) &v[i]);
    pthread_create (&th_b, NULL, writer, (void *) &v[i]);
  }
  free (rw);
  pthread_exit (NULL);
}
```

Init mutex

Run threads
(N readers, N writers)

No joins !

Solution (with mutexes)

```
static void *reader (void *arg) {
  int *p = (int *) arg;
  int i = *p;
  pthread_mutex_lock (&rw->meR);
  rw->nr++;
  if (rw->nr == 1)
    pthread_mutex_lock (&rw->w);
  pthread_mutex_unlock (&rw->meR);
  printf("Thread %d reading\n", i);
  pthread_mutex_lock (&rw->meR);
  rw->nr--;
  if (rw->nr == 0)
    pthread_mutex_unlock (&rw->w);
  pthread_mutex_unlock (&rw->meR);
  pthread_exit (NULL);
}
```

Prologue

CS

Epilogue

Solution (with mutexes)

```
static void *writer (void *arg) {
  int *p = (int *) arg;
  int i = *p;

  pthread_mutex_lock (&rw->meW);
  pthread_mutex_lock (&rw->w);
  printf("Thread %d writing\n", i);
  pthread_mutex_unlock (&rw->w);
  pthread_mutex_unlock (&rw->meW);

  pthread_exit (NULL);
}
```

Prologue

CS

Epilogue

Solution (with condition variables)

Several different solutions are possible

As previous solution

```
...

typedef struct rw_s {
  pthread_mutex_t lock;
  pthread_cond_t turn;
  int nr, nw;
} rw_t;

rw_t *rw;
```

Lock mutex
Conditional variable for the turn
(to readers or to writers)
Condition (2 counters)

Solution (with condition variables)

```
int main (void) {
  pthread_t th_a, th_b;
  int i, v[N];
  setbuf (stdout, NULL);
  rw = (rw_t *) malloc (sizeof(rw_t));
  pthread_mutex_init (&rw->lock, NULL);
  pthread_cond_init (&rw->turn, NULL);
  rw->nr = rw->nw = 0;
  for (i=0; i<N; i++) {
    v[i] = i;
    pthread_create (&th_a, NULL, reader, (void *) &v[i]);
    pthread_create (&th_b, NULL, writer, (void *) &v[i]);
  }
  free (rw);
  pthread_exit (NULL);
}
```

The main program is similar to the one with mutexes

No joins !

Solution (with condition variables)

```
static void *reader (void *arg) {
  int *p = (int *) arg;
  int i = *p;
  pthread_mutex_lock (&rw->lock);
  while (rw->nw > 0)
    pthread_cond_wait (&rw->turn, &rw->lock);
  rw->nr++;
  pthread_mutex_unlock (&rw->lock);
  printf ("Thread %2d reading\n", i);
  pthread_mutex_lock (&rw->lock);
  rw->nr--;
  if (rw->nr==0) pthread_cond_broadcast (&rw->turn);
  pthread_mutex_unlock (&rw->lock);
  pthread_exit (NULL);
}
```

Prologue

CS

Epilogue

Solution (with condition variables)

```
static void *writer (void *arg) {
  int *p = (int *) arg;
  int i = *p;
  pthread_mutex_lock (&rw->lock);
  while (rw->nw > 0 || rw->nr > 0)
    pthread_cond_wait (&rw->turn, &rw->lock);
  rw->nw++;
  pthread_mutex_unlock (&rw->lock);
  printf ("Thread %2d writing\n", i);
  pthread_mutex_lock (&rw->lock);
  rw->nw--;
  pthread_mutex_unlock (&rw->lock);
  pthread_cond_broadcast (&rw->turn);
  pthread_exit (NULL);
}
```

Prologue

CS

Epilogue

Solution (with reader-writer locks)

Precedence depends on how RW locks are implemented

This is all we need with RW locks

The main program is similar to the one with mutexes

```
#define N 20

pthread_rwlock_t rw;

int main (void) {
  pthread_t th_a, th_b;
  int i, v[N];
  setbuf (stdout, NULL);
  pthread_rwlock_init (&rw, NULL);
  for (i=0; i<N; i++){
    v[i] = i;
    pthread_create (&th_a, NULL, reader, (void *) &v[i]);
    pthread_create (&th_b, NULL, writer, (void *) &v[i]);
  }
  pthread_exit(NULL);
}
```

Solution (with reader-writer locks)

```
static void *reader(void *arg) {
  int *p = (int *) arg;
  int i = *p;
  pthread_rwlock_rdlock (&rw);
  printf ("Thread %d reading\n", i);
  pthread_rwlock_unlock (&rw);
  pthread_exit (NULL);
}
static void *writer(void *arg) {
  int *p = (int *) arg;
  int i = *p;
  pthread_rwlock_wrlock (&rw);
  printf ("Thread %d writing\n", i);
  pthread_rwlock_unlock (&rw);
  pthread_exit (NULL);
}
```

Prologue

CS

Epilogue

Prologue

CS

Epilogue

Exercise

- Implement a **priority semaphore**, i.e., a semaphore in which
  - Each thread has an intrinsic priority
    - The priority is an integer value
    - The **higher** priority corresponds to the **lower** value
  - The semaphore has a priority queue associated with it where threads await to be signalled
  - When a call to the signal (sem_post) function wakes-up a thread, threads must be woken-up following their priority
    - We have to awake the threads with the higher priority among the ones waiting on that semaphore

Solution

```
... headers ...
#define N 4

typedef struct list_s {
  sem_t *sem;
  int priority;
  struct list_s *next;
} list_t;
typedef struct my_sem_s {
  sem_t *mutex;
  int cnt;
  list_t *sem_list;
} my_sem_t;

... prototypes ...
my_sem_t *my_sem;
```

List element:
Semaphore
Priority
Next pointer

We create a new ADT, a priority semaphore. This ADT associates a different semaphore to each thread and we insert thread in a priority queue (a heap)

For the sake of simplicity the heap is substituted by an **ordered list** (linear cost instead of logarithmic cost)

Priority semaphore:
Mutex to protect the sem
Semaphore counter (list length)
Queue list

Solution

```
int main (void) {
  pthread_t th_a;
  int i, *pi;
  setbuf (stdout, 0);
  my_sem = my_sem_init (0);
  for (i=1; i<=N; i++) {
    pi = (int *) malloc (sizeof (int));
    *pi = i * 10;
    pthread_create (&th_a, NULL, runner, pi);
  }
  for (i=0; i<N; i++) {
    my_sem_signal (my_sem);
    sleep(1);
  }
  pthread_exit (0);
}
```

Init priority semaphore

Run N threads

Wake threads in order

Solution

```
my_sem_t *my_sem_init (int value) {
  my_sem_t *s;
  s = (my_sem_t *) malloc (sizeof (my_sem_t));
  s->cnt = value;
  s->mutex = (sem_t *) malloc (sizeof (sem_t));
  sem_init (s->mutex, 0, 1);

  s->sem_list = (list_t *) malloc (sizeof (list_t));
  s->sem_list->priority = INT_MIN;
  s->sem_list->next = (list_t *) malloc (sizeof (list_t));
  s->sem_list->next->priority = INT_MAX;
  s->sem_list->next->next = NULL;
  return s;
}
```

Init the priority semaphore and related waiting list

Create list with sentinel and dummy elements

Solution

Thread function

```
static void *runner (void *arg) {
  int *i = (int *) arg;
  pthread_detach (pthread_self ());
  sleep (random () % 5);
  printf ("request from %u %d\n",
    (unsigned int) pthread_self (), *i);

  my_sem_wait (my_sem, *i);

  printf ("service of   %u %d\n",
    (unsigned int) pthread_self (), *i);
  return 0;
}
```

All threads just wait on the priority semaphore

Priority = thread id

Solution

```
void my_sem_wait (my_sem_t * s, int prio) {
  sem_t *new_sem;


  sem_wait (s->mutex);
  if (--s->cnt < 0) {
    new_sem = enqueue_sorted (s->sem_list, prio);
    sem_post (s->mutex);
    sem_wait (new_sem);
    sem_destroy (new_sem);
  } else {
    sem_post (s->mutex);
  }
}
```

Enqueue only when the counter reaches 0

Enqueue the thread (on the ordered list)

Wait on the semaphore until it is signaled (by my_sem_signal)

Non-blocking wait

Solution

```
void my_sem_signal (my_sem_t * s) {
  sem_t *sem;

  sem_wait (s->mutex);
  if (++s->cnt <= 0) {
    sem = dequeue_sorted (s->sem_list);
    sem_post (sem);
  }
  sem_post (s->mutex);
}
```

Wake-up a waiting thread

Solution

```
sem_t *enqueue_sorted (list_t * head, int priority) {
  list_t *p, *new_elem;
  p = head;
  while (priority > p->next->priority)
    p = p->next;
  new_elem = (list_t *) malloc (sizeof (list_t));
  new_elem->sem = (sem_t *) malloc (sizeof (sem_t));
  sem_init (new_elem->sem, 0, 0);
  new_elem->priority = priority;
  new_elem->next = p->next;
  p->next = new_elem;
  printf("%u with priority = %d  waits on sem %p\n",
    (unsigned int) pthread_self(), priority, new_elem->sem);
  return new_elem->sem;
}
```

Search insertion position

Set all fields of new element

Insert new thread in correct position

Solution

```
sem_t *dequeue_sorted (list_t * head) {
  list_t *p;
  sem_t *s;


  p = head->next;
  s = head->next->sem;
  printf("%u with priority = %d awakes from sem %p\n",
    (unsigned int) pthread_self(), head->next->priority, s);
  head->next = head->next->next;
  free (p);
  return s;
}
```

Extract from list head … once skipped the dummy element

Get corresponding semaphore

Extract element

Return semaphore to signal

Specification

- Implement a program that
  - Takes as argument a number **K**
  - Creates **K** threads with code **TA** and **2*K** threads with code **TB**, then exit
    - All threads are identified by their creation index (from **0** to **K-1** and from **0** to **2*K-1,** respectively)
  - Threads
    - Sleep a random number of seconds
    - Then, they coordinate their effort such that
      - For each set of 3 threads running, the first thread is a thread TA and the other two are threads TB
      - All threads must print their id
      - The last thread TB must also print a newline

Specification

- Execution example
  - pgrm 8
    - A0 B7 B6
    - A1 B8 B5
    - A2 B9 B10
    - A7 B11 B12
    - A4 B4 B13
    - A5 B14 B15
    - A3 B3 B0
    - A6 B2 B1

One TA

Solution

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include "pthread.h"
#include "semaphore.h"

sem_t *sa, *sb;
int n;

static void *TA (void *);
static void *TB (void *);
```

Global variables:
1 semaphore for threads TA
1 semaphore for threads TB
1 counter (1:TA, 2:TB, 3:TB and newline)

Solution

Main (extract)

```
num_threads = atoi(argv[1]);
sa = (sem_t *) malloc (sizeof(sem_t));
sb = (sem_t *) malloc (sizeof(sem_t));
sem_init (sa, 0, 1);
sem_init (sb, 0, 0);
setbuf (stdout, 0);
for (i=0; i<2*num_threads; i++){
  pi = (int *) malloc (sizeof(int)); *pi = i;
  pthread_create (&th, NULL, TB, pi);
}
for (i=0; i<num_threads; i++){
  pi = (int *) malloc (sizeof(int)); *pi=i;
  pthread_create (&th, NULL, TA, pi);
}
pthread_exit(0);
```

No joins !

Solution

```
static void *TA (void *arg) {
  int id;
  int *pi = (int *) arg;
  id = *pi;

  pthread_detach (pthread_self ());

  sleep (1);
  sem_wait (sa);
  printf ( "A%d ", id);
  n++;
  sem_post (sb);

  return 0;
}
```

Detach thread (no need to join)

Once TA first

Then one TB

Solution

```
static void *TB (void *arg) {
  int id;
  int *pi = (int *) arg;
  id = *pi;
  pthread_detach (pthread_self ());
  sleep(1);
  sem_wait (sb);
  n++;
  printf ("B%d ", id);
  if (n>=3) {
    printf ("\n"); n = 0; sem_post (sa);
  } else {
    sem_post (sb);
  }
  return 0;
}
```

One TB runs

Newline and run TA again

Run one more TB

Specification

- Modify the previous program such that
  - The main program runs 1 thread TA and 1 thread TB
  - TA and TB include an infinite cycle in which they display 'A' and 'B', respectively
    - For each set of 3 print instructions, we must have 1 character A and 2 characters B in any position
  - Execution example

```
pgrm
     ABB
     BBA
     BAB
     etc.
```

Solution

Same main
One more semaphore (**me**)
sem_init (sa, 0, 1);
sem_init (sb, 0, 2);
sem_init (me, 0, 1);

```
int main (int argc, char **argv) {
  pthread_t th;

  sa = (sem_t *) malloc (sizeof(sem_t));
  sb = (sem_t *) malloc (sizeof(sem_t));
  me = (sem_t *) malloc (sizeof(sem_t));
  sem_init (sa, 0, 1);
  sem_init (sb, 0, 2);
  sem_init (me, 0, 1);
  setbuf(stdout, 0);
  pthread_create (&th, NULL, TA, NULL);
  pthread_create (&th, NULL, TB, NULL);
  pthread_exit(0);
}
```

Solution

```
static void *TA (void) {
  pthread_detach (pthread_self ());
  while (1) {
    sleep (rand()%2);
    sem_wait (sa);
    sem_wait (me);
    printf ( "A");
    n++;
    if (n>=3) {
      printf ("\n");
      n = 0;
      sem_post (sa); sem_post (sb); sem_post (sb);
    }
    sem_post (me);
  }
  return 0;
}
```

Loop forever

Wait for a random time

The last thread wakes-up one A and two B threads

Solution

```
static void *TB (void *arg) {
  pthread_detach (pthread_self ());
  while (1) {
    sleep (rand()%2);
    sem_wait (sb); sem_wait (me);
    printf ("B");
    n++;
    if (n>=3) {
      printf ("\n");
      n = 0; sem_post (sa); sem_post (sb); sem_post (sb);
    }
    sem_post (me);
  }
  return 0;
}
```

Loop forever

Wait for a random time

The last thread wakes-up one A and two B threads