CS 3214 Spring 2021 Test 2 Solutions

September 13, 2021

Contents

1 Async (9 pts) 2

2 Concurrency Bugs (15 pts) 7

2.1 Reading Helgrind (6 pts) 7

2.2 A Bug in Mozilla (9 pts) 9

3 Map Reduce, Threaded (12 pts) 12

4 Deadlock Detection (12 pts) 18

5 Are we there yet? Roadblocks to increased multithreading utilization. (12 pts) 26

Rules

- This exam is open book, open notes, and open Internet, but in a read-only way.
- You are not allowed to post or otherwise communicate with anyone else about these problems.
- You are required to cite any sources you use, except for lecture material, source code provided as part of the class materials, and the textbook.

1 Async (9 pts)

In the handout of project 2, we motivated the need for a fork-join threadpool by demonstrating what results when languages provide an `async` facility without also providing an underlying efficient threadpool implementation. We used this C++ program which performs a parallel sum computation:

```cpp
#include <iostream>
#include <vector>
#include <algorithm>
#include <numeric>
#include <future>

template <typename RAIter>
int parallel_sum(RAIter beg, RAIter end)
{
    auto len = std::distance(beg, end);
    if (len < 1000)
        return std::accumulate(beg, end, 0);

    RAIter mid = beg + len/2;
    auto handle = std::async(std::launch::async,
                             parallel_sum<RAIter>, mid, end);
    int sum = parallel_sum(beg, mid);
    return sum + handle.get();
}

int main()
{
    std::vector<int> v(100000, 1);
    std::cout << "The sum is " << parallel_sum(v.begin(), v.end())
              << '\n';
}
```

A C version of such a program was included in the tests accompanying p2 where it was executed by the threadpool you built. In this question, you are asked to explore how much complexity implementing a threadpool added by developing an equally naive `async` implementation in C. Specifically, you should implement the functions `async` and `get` below such that the resulting program computes a parallel sum in the same way the C++ program does, which is by devoting a new and separate thread to each asynchronous task. Note the restrictions on where you may add code.

Your implementation should be reliable and data race free. For the purposes of this question, you do not need to worry about situations in which creating a new thread may fail due to reaching a per-user limit.

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

```c
#include <stddef.h>
#include <stdlib.h>
#include <stdio.h>

typedef int (*async_function_t)(void *);
/*******************************************/
// you may make changes below this line

struct future {
    // implement this
};
/* add any additional functions, variables, and data structures here */

// obtain the result of this asynchronous computation
// if necessary, block until the result is available
// frees the provided future so no future accesses are
// possible
int get(struct future *f)
{
  // implement this
}

// start this asynchronous function in a new thread, passing it args
// returns a dynamically allocate future to refer to the computation
struct future * async(async_function_t fun, void *args)
{
   // implement this
}

// you may not make changes below this line
// ------------------------------------------------------

struct args {
    int *beg, *end;
};

int parallel_sum(struct args *args)
{
    int *beg = args->beg, *end = args->end;

    ptrdiff_t len = end - beg;
    if (len < 1000) {
        int sum = 0;
        while (beg < end)
            sum += *beg++;
        return sum;
    }

    int *mid = beg + len/2;
    struct args rhalf = { .beg = mid, .end = end };
    struct future * handle = async((async_function_t) parallel_sum, &rhalf);
    struct args lhalf = { .beg = beg, .end = mid };
    return parallel_sum(&lhalf) + get(handle);
}

#define N 1000000
int v[N];
int main()
{
    for (int i = 0; i < N; i++)
        v[i] = 1;

    struct args whole = { .beg = v, .end = v + N };
    printf("The sum is %d\n", parallel_sum(&whole));
}
```

[Solution] The solution involves starting a new thread that receives the future, executes it, and stores the result. This thread is joined in future_get. The techniques used are very similar to those in project 2.

```c
#include <pthread.h>
#include <stddef.h>
#include <stdlib.h>
#include <stdio.h>

typedef int (*async_function_t)(void *);
/******************************************/
// you may make changes below this line

struct future {
    pthread_t t;
    async_function_t task;
    void *args;
    int result;
};

static void * run(void *arg)
{
    struct future * f = arg;
    f->result = f->task(f->args);
    return NULL;
}
```

```c

// obtain the result of this asynchronous computation
// if necessary, block until the result is available
// frees the provided future so no future accesses are
// possible
int get(struct future *f) {
    pthread_join(f->t, NULL);
    int rc = f->result;
    free(f);
    return rc;
}

// start this asynchronous function in a new thread, passing it args
// returns a dynamically allocate future to refer to the computation
struct future * async(async_function_t fun, void *args)
{
    struct future *f = malloc(sizeof *f);
    f->task = fun;
    f->args = args;
    pthread_create(&f->t, NULL, run, f);
    return f;
}

// you may not make changes below this line
// ------------------------------------------------------

struct args {
    int *beg, *end;
};

int parallel_sum(struct args *args)
{
    int *beg = args->beg, *end = args->end;

    ptrdiff_t len = end - beg;
    if (len < 1000) {
        int sum = 0;
        while (beg < end)
            sum += *beg++;
        return sum;
    }

    int *mid = beg + len/2;
    struct args rhalf = { .beg = mid, .end = end };
    struct future * handle = async((async_function_t) parallel_sum, &rhalf);
    struct args lhalf = { .beg = beg, .end = mid };
    return parallel_sum(&lhalf) + get(handle);
}

#define N 1000000
int v[N];
int main()
{
    for (int i = 0; i < N; i++)
        v[i] = 1;

    struct args whole = { .beg = v, .end = v + N };
    printf("The sum is %d\n", parallel_sum(&whole));
}
```

2 Concurrency Bugs (15 pts)

2.1 Reading Helgrind (6 pts)

Your p2 team partner received the following output when using the Helgrind race detection checker on their program.

Explain the information you can gather from the output in your own words. What did your teammate likely do wrong? Be sure to translate and include all relevant characteristics (variable names, line numbers, etc.) in your explanation.

Here is the output Helgrind produced:

```
==21540== Helgrind, a thread error detector
==21540== Copyright (C) 2007-2017, and GNU GPL'd, by OpenWorks LLP et al.
==21540== Using Valgrind-3.13.0 and LibVEX; rerun with -h for copyright info
==21540== Command: ./helgrind
==21540==
==21540== ---Thread-Announcement------------------------------------------
==21540==
==21540== Thread #3 was created
==21540==    at 0x518470E: clone (clone.S:71)
==21540==    by 0x4E4BEC4: create_thread (createthread.c:100)
==21540==    by 0x4E4BEC4: pthread_create@@GLIBC_2.2.5 (pthread_create.c:797)
==21540==    by 0x4C38A27: ??? (in /usr/lib/valgrind/vgpreload_helgrind-amd64-linux.so)
==21540==    by 0x1089A2: main (helgrind.c:37)
==21540==
==21540== ---Thread-Announcement------------------------------------------
==21540==
==21540== Thread #2 was created
==21540==    at 0x518470E: clone (clone.S:71)
==21540==    by 0x4E4BEC4: create_thread (createthread.c:100)
==21540==    by 0x4E4BEC4: pthread_create@@GLIBC_2.2.5 (pthread_create.c:797)
==21540==    by 0x4C38A27: ??? (in /usr/lib/valgrind/vgpreload_helgrind-amd64-linux.so)
==21540==    by 0x10897F: main (helgrind.c:36)
==21540==
==21540== ----------------------------------------------------------------
==21540==
==21540==  Lock at 0x309080 was first observed
==21540==    at 0x4C3603C: ??? (in /usr/lib/valgrind/vgpreload_helgrind-amd64-linux.so)
==21540==    by 0x1088A1: thread2 (helgrind.c:22)
==21540==    by 0x4C38C26: ??? (in /usr/lib/valgrind/vgpreload_helgrind-amd64-linux.so)
==21540==    by 0x4E4B6DA: start_thread (pthread_create.c:463)
==21540==    by 0x518471E: clone (clone.S:95)
==21540==  Address 0x309080 is 0 bytes inside data symbol "lock2"
==21540==
==21540==  Lock at 0x309040 was first observed
==21540==    at 0x4C3603C: ??? (in /usr/lib/valgrind/vgpreload_helgrind-amd64-linux.so)
==21540==    by 0x108849: thread1 (helgrind.c:11)
==21540==    by 0x4C38C26: ??? (in /usr/lib/valgrind/vgpreload_helgrind-amd64-linux.so)
==21540==    by 0x4E4B6DA: start_thread (pthread_create.c:463)
==21540==    by 0x518471E: clone (clone.S:95)
==21540==  Address 0x309040 is 0 bytes inside data symbol "lock1"
==21540==
==21540== Possible data race during read of size 4 at 0x3090A8 by thread #3
==21540== Locks held: 1, at address 0x309080
==21540==    at 0x1088AF: thread2 (helgrind.c:24)
==21540==    by 0x4C38C26: ??? (in /usr/lib/valgrind/vgpreload_helgrind-amd64-linux.so)
```

```
==21540==    by 0x4E4B6DA: start_thread (pthread_create.c:463)
==21540==    by 0x518471E: clone (clone.S:95)
==21540==
==21540== This conflicts with a previous write of size 4 by thread #2
==21540== Locks held: 1, at address 0x309040
==21540==    at 0x108860: thread1 (helgrind.c:13)
==21540==    by 0x4C38C26: ??? (in /usr/lib/valgrind/vgpreload_helgrind-amd64-linux.so)
==21540==    by 0x4E4B6DA: start_thread (pthread_create.c:463)
==21540==    by 0x518471E: clone (clone.S:95)
==21540==  Address 0x3090a8 is 0 bytes inside data symbol "gvar"
==21540==
```

[Solution] The Helgrind messages shows that 2 threads access a shared 32-bit variable `gvar` concurrently. These accesses were concurrent because the programmer failed to synchronize access to them with a lock that was held when both threads accessed the variable. Instead, Helgrind reports that Thread #3 held a mutex named `lock2` when accessing the variable on line helgrind.c:24 whereas Thread #2 held a mutex named `lock1` when accessing the same variable on line 13.

Here is the program that was used to create this Helgrind output:

```
1  #include <pthread.h>
2  #include <stdio.h>
3
4  pthread_mutex_t lock1 = PTHREAD_MUTEX_INITIALIZER;
5  pthread_mutex_t lock2 = PTHREAD_MUTEX_INITIALIZER;
6
7  static void *
8  thread1(void * _tn)
9  {
10     int * shared = (int *) _tn;
11     pthread_mutex_lock(&lock1);
12     for (int i = 0; i < 1000000; i++)
13         (*shared)++;
14     pthread_mutex_unlock(&lock1);
15     return NULL;
16 }
17
18 static void *
19 thread2(void * _tn)
20 {
21     int * shared = (int *) _tn;
22     pthread_mutex_lock(&lock2);
23     for (int i = 0; i < 1000000; i++)
24         (*shared)++;
25     pthread_mutex_unlock(&lock2);
26     return NULL;
27 }
28
29 static int gvar;
```

```

int
main()
{
    int N = 2;
    pthread_t t[N];
    pthread_create(t + 0, NULL, thread1, &gvar);
    pthread_create(t + 1, NULL, thread2, &gvar);

    for (int i = 0; i < N; i++)
        pthread_join(t[i], NULL);

    printf("shared = %d\n", gvar);
    return 0;
}
```

2.2 A Bug in Mozilla (9 pts)

Concurrency bugs can arise when different threads interact. The following program contains such a concurrency bug. It was reconstructed from a bug that in the past affected systems such as Mozilla's codebase.

```
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>
#include <string.h>
#include <unistd.h>
#include <pthread.h>

typedef void *(*worker_func_t)(void *);

enum worker_state { PENDING, STARTED, FINISHED };
struct worker {
    pthread_t thread_id;
    enum worker_state state;
};

pthread_mutex_t glock = PTHREAD_MUTEX_INITIALIZER;      // protects worker1 and *worker1
struct worker *worker1;

struct worker *start_worker(worker_func_t wfun, void *wdata)
{
    struct worker *w = malloc(sizeof *w);
    w->state = PENDING;
    pthread_create(&w->thread_id, NULL, wfun, wdata);
24      return w;
25  }
26
27  static void *
28  worker1_fun(void *_)
29  {
30      pthread_mutex_lock(&glock);
31      worker1->state = STARTED;
32      pthread_mutex_unlock(&glock);
33
34      printf("worker working\n");
35
36      pthread_mutex_lock(&glock);
37      worker1->state = FINISHED;
38      pthread_mutex_unlock(&glock);
39      return NULL;
40  }
41
42  int
43  main()
44  {
45      struct worker * w = start_worker(worker1_fun, NULL);
46      pthread_mutex_lock(&glock);
47      worker1 = w;
48      pthread_mutex_unlock(&glock);
49      pthread_join(worker1->thread_id, NULL);
50      printf("all done\n");
51  }
```

1. (3 pts) Determine the bug and explain how the program would fail if the bug manifested itself.

2. (3 pts) Is the bug you found a data race? Data races are defined in lecture; or more formally in the C11 memory model [URL] as follows:

   > When an evaluation of an expression writes to a memory location and another evaluation reads or modifies the same memory location, the expressions are said to conflict. A program that has two conflicting evaluations has a data race unless either
   >
   > - both conflicting evaluations are atomic operations
   > - one of the conflicting evaluations *happens-before*[1] another

   Justify your answer, referring to line numbers in the code if this is necessary.

3. (3 pts) Suggest a way to fix the bug.

[1]Here, *happens-before* refers to the cross-thread relationship between events, see Slide 11.

[Solution]

1. The program does not guarantee that the global variable 'worker1' is set on line 47 before it is accessed on line 31. If line 31 executes before the assignment on line 47 it would result in a segmentation violation due to accessing a NULL pointer. (In fact, this can be observed when running this code under valgrind.)
2. This is not a data race since the mutex glock serializes these 2 operations. It is an ordering violation.
3. It can be fixed in a number of ways, for instance,
   (a) switching lines 45 and 46, thus ensuring that line 31 executes after line 48.
   (b) Moving 'wdata' into 'struct worker', passing 'w' instead of 'wdata' on line 23 and setting 'worker1' in 'worker1_fun'.
   (c) Using a semaphore (initialized with 0) that is waited for at the beginning of 'worker1_fun' and signaled after the 'worker1 = w' assignment.

The original Mozilla bug was both a data race and an ordering violation and was reported in: *Learning from mistakes: a comprehensive study on real world concurrency bug characteristics* by Lu et al at ASPLOS 2008 (Figure 2). [URL]

3 Map Reduce, Threaded (12 pts)

In Test 1, you were asked to simulate a MapReduce network using inter-process communication (IPC), specifically Unix pipes. In this problem, we explore how to perform the same simulation using inter-thread communication using what we call "thread pipes." A thread pipe, like a Unix pipe, supports three operations:

- `pipe_send`: sends a message of `MSG_SIZE` bytes through the pipe. Messages are indivisible (atomic).
- `pipe_receive`: receives the next message of `MSG_SIZE` from the pipe If multiple threads call `pipe_receive`, any of them may receive the message.
- `pipe_initialize`: initializes a thread pipe object in place

Implement the functions `pipe_initialize`, `pipe_send`, and `pipe_receive` so that the following program works. *Note that the execution result will be the same as was observed in Test 1.*

```c
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>
#include <string.h>
#include <unistd.h>
#include <pthread.h>
#include <sys/wait.h>
#include "list.h"

#define THREADS 7
#define NPIPES 11
#define TOTAL 100
#define MSG_SIZE 20

/* you may not change anything above this line */

struct thread_pipe {
   // implement this
};

// define any other data structures here

/* Initialize a thread pipe object in place */
int
pipe_initialize(struct thread_pipe *pipe)
{
}

/* Send a msg of size MSG_SIZE to a thread pipe
```

```c
 * The pipe does not need to impose a limit on the number of
 * items currently contained in the pipe.
 */
void
pipe_send(struct thread_pipe *pipe, char msg[MSG_SIZE])
{
}

/* Receive a msg of size MSG_SIZE to a thread pipe.
 * Blocks until a msg is in the pipe.
 */
void
pipe_receive(struct thread_pipe *pipe, char msg[MSG_SIZE])
{
}

/*
 * You may not make any changes below this line
 */
static struct thread_pipe pip[NPIPES]; //driver-A, A-B, A-C, A-D, B-E, C-E, C-F, D-F, E-G, F-G
static int a_to_b, a_to_c, a_to_d, c_to_e, c_to_f, total;

/* Read `total` messages from read end of pipe pair in `fromfd` and
 * round-robin distribute them to pipes at `(tofd + i) % stride`
 *
 * For simplicity, we send always MSG_SIZE-byte messages.
 */
static void process(const char *name, struct thread_pipe *from, struct thread_pipe *to,
                    int total, int stride)
{
  char msgi[MSG_SIZE], msgo[MSG_SIZE];

  for (int count = 0; count < total; count++) {
    pipe_receive(from, msgi);
    snprintf(msgo, sizeof msgo, "%s%s", name, msgi);
    pipe_send(&to[count%stride], msgo);
  }
}

static int divceil(int x, int y)
{
  return (x + y - 1) / y;
}

static void *task_a(void *_)
{
  process("A", &pip[0], &pip[1], total, 3);
  return NULL;
}

static void *task_b(void *_)
{
  process("B", &pip[1], &pip[4], a_to_b, 1);
  return NULL;
}

static void *task_c(void *_)
{
  process("C", &pip[2], &pip[5], a_to_c, 2);
  return NULL;
}

static void *task_d(void *_)
{
  process("D", &pip[3], &pip[7], a_to_d, 1);
  return NULL;
}

static void *task_e(void *_)
{
  process("E", &pip[4], &pip[8], a_to_b, 1);
  process("E", &pip[5], &pip[8], c_to_e, 1);
  return NULL;
}

static void *task_f(void *_)
{
  process("F", &pip[6], &pip[9], c_to_f, 1);
  process("F", &pip[7], &pip[9], a_to_d, 1);
  return NULL;
}

static void *task_g(void *_)
{
  process("G", &pip[8], &pip[10], a_to_b + c_to_e, 1);
  process("G", &pip[9], &pip[10], c_to_f + a_to_d, 1);
  return NULL;
}

typedef void * (*thread_func_t)(void *);
void shuffle_threads(thread_func_t *funcs, size_t n)
{
```

```c
  for (int i = 0; i < n - 1; i++) {
    int j = i + rand() / (RAND_MAX / (n - i) + 1);
    thread_func_t t = funcs[j];
    funcs[j] = funcs[i];
    funcs[i] = t;
  }
}

int
main(int ac, char *av[])
{
  // compute how many messages will be sent
  total = ac > 1 ? atoi(av[1]) : TOTAL;
  a_to_b = divceil(total, 3);
  a_to_c = divceil(total-1, 3);
  a_to_d = divceil(total-2, 3);
  c_to_e = divceil(a_to_c, 2);
  c_to_f = divceil(a_to_c-1, 2);

  // creating all the pipes
  for (int i = 0; i < NPIPES; i++)
    if (pipe_initialize(&pip[i])<0) {
      fprintf(stderr, "Pipe creation error\n");
      return EXIT_FAILURE;
    }

  // let's create the threads next
  pthread_t t[THREADS];
  thread_func_t tasks[THREADS] = { task_a, task_b, task_c, task_d, task_e, task_f, task_g };

  // for a bit of challenge, shuffle the order in which they are started
  srand(ac > 2 ? atoi(av[2]) : time(NULL));
  shuffle_threads(tasks, THREADS);

  for (int i = 0; i < THREADS; i++)
    pthread_create(t+i, NULL, tasks[i], NULL);

  // send numbers to A thread
  for (int i = 0; i < total; i++) {
    char msg[MSG_SIZE];
    snprintf(msg, sizeof msg, "%d\n", i);
    pipe_send(&pip[0], msg);
  }

  // get numbers from last thread
  for (int i = 0; i < total; i++) {
    char msg[MSG_SIZE];
    pipe_receive(&pip[10], msg);
    printf("%s", msg);
  }

  // join threads
  for (int i = 0; i < THREADS; i++)
    pthread_join(t[i], NULL);
}
```

[Solution] The solution is a simple application of the producer/consumer queue idea, for instance implemented with a mutex to protect a list of messages and a condition variable to signal availability. We only show the required functions below.

```
struct thread_pipe {
    pthread_mutex_t lock;
    pthread_cond_t msg_available;
    struct list items;
};

struct pipe_item {
    char msg[MSG_SIZE];
    struct list_elem elem;
};

/* Initialize a thread pipe object in place */
int
pipe_initialize(struct thread_pipe *pipe)
{
    list_init(&pipe->items);
    int rc = pthread_mutex_init(&pipe->lock, NULL);
    return rc ? rc : pthread_cond_init(&pipe->msg_available, NULL);
}

/* Send a msg of size MSG_SIZE to a thread pipe
 * The pipe does not need to impose a limit on the number of
 * items currently contained in the pipe.
 */
void
pipe_send(struct thread_pipe *pipe, char msg[MSG_SIZE])
{
    struct pipe_item *item = malloc(sizeof *item);
    pthread_mutex_lock(&pipe->lock);
    list_push_back(&pipe->items, &item->elem);
    memcpy(item->msg, msg, MSG_SIZE);
    pthread_cond_signal(&pipe->msg_available);
```

```c
    pthread_mutex_unlock(&pipe->lock);
}

/* Receive a msg of size MSG_SIZE to a thread pipe.
 * Blocks until a msg is in the pipe.
 */
void
pipe_receive(struct thread_pipe *pipe, char msg[MSG_SIZE])
{
    pthread_mutex_lock(&pipe->lock);
    while (list_empty(&pipe->items))
        pthread_cond_wait(&pipe->msg_available, &pipe->lock);

    struct pipe_item * item = list_entry(list_pop_front(&pipe->items),
                                         struct pipe_item, elem);
    memcpy(msg, item->msg, MSG_SIZE);
    free(item);
    pthread_mutex_unlock(&pipe->lock);
}
```

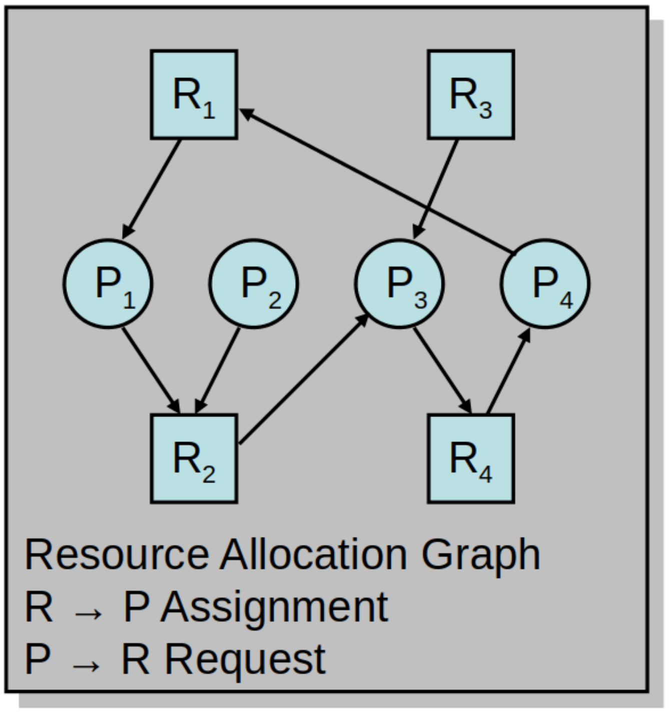

Figure 1: This resource allocation graph shows a deadlock situation that arose with 4 processes and 4 resources.

4 Deadlock Detection (12 pts)

Consider the resource allocation graph shown in Figure 1 which is reproduced from the lecture on deadlocks. In this problem, you are asked to completed a multi-threaded program that produces the exact deadlock shown in this graph and that can detect when it deadlocks.

In the following program the 4 involved processes P1, P2, P3, and P4 are represented as threads 1 through 4, executing functions `P1fun`, `P2fun`, `P3fun`, and `P4fun`, respectively. These threads (once implemented) should acquire resources such that the deadlock situation shown in Figure 1 arises. A barrier divides the actions of each thread in two sequential parts. All threads execute their second part after all threads have executed their first part. However, the threads execute their respective actions within each part concurrently with each other.

- (6 pts) Implement each thread such that the program deadlocks and the resource acquisition graph shown in the Figure results, no matter the order in which the scheduler may schedule those threads.
- (6 pts) Implement the function `check_for_deadlock` such that it detects a cycle in the resource acquisition graph when a process attempts to acquire a resource. Your deadlock detection algorithm should be general and not assume the specific graph shown in the Figure. Follow the required output exactly, which will be explained below.

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

struct Process;
static _Thread_local struct Process *current;

pthread_mutex_t glock = PTHREAD_MUTEX_INITIALIZER;

struct Resource {
    struct Process *heldby; // process currently holding the resource
    char *name;
    sem_t sem;          // represents the resource
};

struct Process {
    struct Resource *acquires; // resource this process is trying to acquire
    pthread_t thread;
    char *name;
};

/* Process `p` is about to request resource `r`.
 * Check for deadlock.
 */
static void
check_for_deadlock(struct Process *p, struct Resource *r)
{
    printf("checking deadlock as process %s wants %s", p->name, r->name);
    /* if the attempt by process p to acquire resource r would
     * cause a deadlock, output this and then abort()
            printf(" - found DEADLOCK\n");
            abort();
     */
    printf(" - no deadlock\n");
}

void acquire(struct Resource *r)
{
    pthread_mutex_lock(&glock);
    check_for_deadlock(current, r);
    if (r->heldby == NULL) {
        r->heldby = current;
        current->acquires = NULL;
    } else {
        current->acquires = r;
    }
    pthread_mutex_unlock(&glock);
    if (current->acquires)
        sem_wait(&r->sem);
}

struct Process P1 = { .name = "P1" },
               P2 = { .name = "P2" },
               P3 = { .name = "P3" },
               P4 = { .name = "P4" };

struct Resource R1 = { .name = "R1" },
                R2 = { .name = "R2" },
                R3 = { .name = "R3" },
                R4 = { .name = "R4" };

static pthread_barrier_t barrier;

static void * P1fun(void *process)
{
    current = process;
    // you may add code here
    pthread_barrier_wait(&barrier);
    // you may add code here
    return NULL;
}

static void * P2fun(void *process)
{
    current = process;
    // you may add code here
    pthread_barrier_wait(&barrier);
    // you may add code here
    return NULL;
}

static void * P3fun(void *process)
{
    current = process;
    // you may add code here
    pthread_barrier_wait(&barrier);
    // you may add code here
    return NULL;
}
```

```

static void * P4fun(void *process)
{
    current = process;
    // you may add code here
    pthread_barrier_wait(&barrier);
    // you may add code here
    return NULL;
}

int
main()
{
    sem_init(&R1.sem, 0, 0);
    sem_init(&R2.sem, 0, 0);
    sem_init(&R3.sem, 0, 0);
    sem_init(&R4.sem, 0, 0);
    pthread_barrier_init(&barrier, NULL, 4);
    pthread_create(&P1.thread, NULL, P1fun, &P1);
    pthread_create(&P2.thread, NULL, P2fun, &P2);
    pthread_create(&P3.thread, NULL, P3fun, &P3);
    pthread_create(&P4.thread, NULL, P4fun, &P4);

    pthread_join(P1.thread, NULL);
    pthread_join(P2.thread, NULL);
    pthread_join(P3.thread, NULL);
    pthread_join(P4.thread, NULL);
}
```

Note that since the program is intended to function as an example of a system that is guaranteed to deadlock, a `release()` operation is not implemented.

The expected output of this program will be similar to the following:

```
checking deadlock as process ?? wants ?? - no deadlock
checking deadlock as process ?? wants ?? - no deadlock
...
checking deadlock as process ?? wants ?? - found DEADLOCK
Aborted (core dumped)
```

where the ?? are elided (since you need to find them as part of task 1). The output order does not have to be deterministic.

[Solution]

A possible output is shown here:

```
checking deadlock as process P1 wants R1 - no deadlock
checking deadlock as process P4 wants R4 - no deadlock
checking deadlock as process P3 wants R3 - no deadlock
checking deadlock as process P3 wants R2 - no deadlock
checking deadlock as process P3 wants R4 - no deadlock
checking deadlock as process P2 wants R2 - no deadlock
checking deadlock as process P1 wants R2 - no deadlock
checking deadlock as process P4 wants R1 - found DEADLOCK
Aborted (core dumped)
```

For part 1, the solution requires 2 phases: in phase 1, every process should acquire the resource that the resource allocation graph shows it holds (e.g., P1 holds R1, P3 holds R2 and R3, and P4 holds R3). In Phase 2, they will be trying to acquire the resource shown in the request edges. The last attempt will lead to detectable deadlock

For part 2, the deadlock detection can be performed with a simple loop following the request and assignment edges. A cycle would lead back to the process currently trying to acquire a resource.

The complete solution is shown below:

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

struct Process;
static _Thread_local struct Process *current;

pthread_mutex_t glock = PTHREAD_MUTEX_INITIALIZER;

struct Resource {
    struct Process *heldby; // process currently holding the resource
    char *name;
    sem_t sem;          // represents the resource
};

struct Process {
    struct Resource *acquires; // resource this process is trying to acquire
    pthread_t thread;
    char *name;
};

/* Process `p` is about to request resource `r`.
 * Check for deadlock.
 */
static void
check_for_deadlock(struct Process *p, struct Resource *r)
{
    printf("checking deadlock as process %s wants %s", p->name, r->name);
    for (;;) {
        struct Process *nextp = r->heldby;
        if (nextp == p) {
            printf(" - found DEADLOCK\n");
            abort();
        }
        if (nextp != NULL && nextp->acquires) {
            // printf(" and process %s wants %s", nextp->name, r->name);
            r = nextp->acquires;
        } else
            break;
    }
    printf(" - no deadlock\n");
}

void acquire(struct Resource *r)
{
    pthread_mutex_lock(&glock);
    check_for_deadlock(current, r);
    if (r->heldby == NULL) {
        r->heldby = current;
        current->acquires = NULL;
    } else {
        current->acquires = r;
    }
    pthread_mutex_unlock(&glock);
    if (current->acquires)
        sem_wait(&r->sem);
}

struct Process P1 = { .name = "P1" },
               P2 = { .name = "P2" },
               P3 = { .name = "P3" },
               P4 = { .name = "P4" };

struct Resource R1 = { .name = "R1" },
                R2 = { .name = "R2" },
                R3 = { .name = "R3" },
                R4 = { .name = "R4" };

static pthread_barrier_t barrier;

static void * P1fun(void *process)
{
    current = process;
    // you may add code here
    acquire(&R1);
```

```c
    pthread_barrier_wait(&barrier);
    // you may add code here
    acquire(&R2);
    return NULL;
}

static void * P2fun(void *process)
{
    current = process;
    // you may add code here
    pthread_barrier_wait(&barrier);
    // you may add code here
    acquire(&R2);
    return NULL;
}

static void * P3fun(void *process)
{
    current = process;
    // you may add code here
    acquire(&R3);
    acquire(&R2);
    pthread_barrier_wait(&barrier);
    // you may add code here
    acquire(&R4);
    return NULL;
}

static void * P4fun(void *process)
{
    current = process;
    // you may add code here
    acquire(&R4);
    pthread_barrier_wait(&barrier);
    // you may add code here
    acquire(&R1);
    return NULL;
}

int
main()
{
    sem_init(&R1.sem, 0, 0);
    sem_init(&R2.sem, 0, 0);
    sem_init(&R3.sem, 0, 0);
    sem_init(&R4.sem, 0, 0);
    pthread_barrier_init(&barrier, NULL, 4);
    pthread_create(&P1.thread, NULL, P1fun, &P1);
    pthread_create(&P2.thread, NULL, P2fun, &P2);
    pthread_create(&P3.thread, NULL, P3fun, &P3);
    pthread_create(&P4.thread, NULL, P4fun, &P4);

    pthread_join(P1.thread, NULL);
    pthread_join(P2.thread, NULL);
    pthread_join(P3.thread, NULL);
    pthread_join(P4.thread, NULL);
}
```

5 Are we there yet? Roadblocks to increased multithreading utilization. (12 pts)

Project 2 inspired a number of you to think about the power of multithreading and how it can affect the program that you write. Specifically, Eric H. posed a timely and interesting question about the current state of the usage of parallel programming, part of which I quote/paraphrase below.

Project 2 specification linked to a specific take by researchers from Berkeley who argue that "parallel programming models, software systems, and a supporting architecture" are key to our incorporation of parallel systems. However, the documentation also highlights the large number of software implementations that do not make use of the latest multi-core processors that have become more common in everyday technology.

[Solution] Some of the reasons/explanations are provided below as an example. We will be accepting a broad range of clearly-articulated answers for these questions.

1. (4 pts) What are some of the challenges in accomplishing the goal of widespread parallel programming? (Name at least two.)
   [Solution] Reasons can range from difficulty of ensuring correctness in parallel programs, complex legacy codes, not every program/application has parallelism, emergence of low-power low-end processors such as in IoT devices, lack of easy-to-use tools, etc.

2. (4 pts) Why do you think there is a mismatch between the software and hardware advancements in parallel computing? (Name at least two reasons.)
   [Solution] Reasons can range from lack of coordination between software and hardware researchers/developers, lack of powerful tools to exploit the hardware features, application needs, porting of legacy code, difficulty in training new people in hardware-software codesign, etc.

3. (4 pts) What steps do you believe must be taken to expedite the adoption of parallel computing systems?
   [Solution] Better education and training, use of innovative ways to exploit parallelism such as function-as-a-service, development of new tools and techniques, novel applications, etc. can be some of the ways to expedite the adoption of parallel computing systems.

Note: This is an open-ended question. Please provide brief answers with well-formed relevant arguments.