SUPSI

Operating Systems

Lab Project: User-level threads

Index

1. Objectives

Modern operating systems are “complex beasts” made of millions of lines of code. Understanding the inner-workings of an operating system by studying the source code is a haunting task. Experimenting with a real kernel is also a time consuming activity. The main problem is that kernel code is typically not developed for teaching purposes, changes are complicated, and debugging at such a low-level is not easy. Since we have (unfortunately) very little time to try to apply the concepts we see in the class in a “real” situation, working inside an existing kernel is not an option. Accordingly, we will approach the problem from a higher level of abstraction: instead of working at the kernel level, we will work on top of an existing kernel. Can we study and experiment with the main concepts of an operating system by working outside the kernel? Yes, if we limit ourselves to just some of the topics found in the course. In this regard, we will focus on task/thread management, scheduling algorithms, and synchronization mechanisms (IPC) with the goal of implementing a thread library. As we saw during the class processes and threads are important abstractions which enable multiple programs to execute concurrently on our computers. Processes represent instances of running programs, and can be viewed as containers which keep track of the resources needed and used by the program. Inside processes, the execution of machine code might simultaneously take different paths, called threads. Each thread is thus an active entity whose execution state is managed by a scheduler. As we saw in the class there exist two ways for implementing threads: either at kernel level, or at user level. Modern operating systems like GNU/Linux or Windows already implement threads, thread scheduling, and thread synchronization primitives at kernel level. Hence, to make things a little bit easier (and perhaps more interesting), the goal of this tutorial is to implement such functionality at the user-level, by developing a simple user-level thread library in C (complete with a scheduler, preemptive behavior, and synchronization mechanisms). This piece of software will behave very similar to a kernel, but will spare us from dealing with the hardware and similar low level concerns: we will learn how to represent the execution context of a thread, how to save and restore that information, how to execute multiple threads “at the same time” (using pseudo-parallelism and preemption), how to design our own scheduler, and how to implement synchronization primitives.

As a prerequisite you need to install a C compiler (typically gcc, which is also available on Windows through Cygwin) and a text (code) editor of your choice. The code presented here is targeted to a GNU/Linux system, however it should compile with no issues on POSIX compatible systems (for example, Apple OSX) too. Because technical discussions on specific machine instructions refer to Intel x86 and AMD64, it is preferable (but not mandatory) to work on such architectures. In order to compile your code you will have to specify the compiler options **`-std=gnu11`** and the linker option **`-lm`**. In this tutorial, the ⌨ symbol indicates some practical exercise / problem that you need to solve before continuing to the next step.

The estimated time to complete an exercise is indicated next to the ⏲ symbol. To complete the whole tutorial (reading and completing the exercises), which will take approximately **35 class hours (~26 effective hours)**, you might need some additional information, such as man pages or online references: feel free to use all the information you need, as the goal of this exercise is to really get a good grasp of some fundamental aspects of an operating system. Do not forget to implement tests for your code, and if you have any doubt or question ask your lecturer!

2. Introduction

Now that we set our goal (that is, implement a user-level thread library)... where do we start? At first glance, we might think of each thread as being some kind of procedure that executes in parallel with other procedures. Consider for example the following code:

```
void thread1() {
        process(take());
}

void thread2() {
        show_progress();
}

void main() {
        while(1) {
                thread1();
                thread2();
        }
}
```

We assume that **take** returns a data unit to be processes (from a global pool), **process** does something on this data, and **show_progress** shows a message to the user indicating the overall progress of the program. The main procedure acts as a scheduler, allowing each thread to be executed. Is it a general and valid solution? It could be, apparently. Execution “bounces” back and forth between **thread1** and **thread2** like in a pseudo-parallel system, but what if the *body* of the **thread1** procedure contains a loop such as:

```
void thread1() {
        for(int i=0; i<10000; i++) {
                process(take());
        }
}
```

In this case the second “thread” would have to wait until the other finishes the loop before getting a chance to execute on the CPU and show a message on the screen, thus not giving the impression of parallel execution. What we need is a way to *yield to the other thread* while executing these procedures, with the possibility of coming back to the same spot afterwards (without restarting the loop):

```
int thread1_running=1;
void thread1() {
        for(int i=0; i<10000; i++) {
                process(take());
                yield();
        }
        thread1_running=0;
}

void thread2() {
        while(thread1_running) {
                show_progress();
                yield();
        }
}

void main() {
        while(1) {
                thread1();
                thread2();
        }
}
```

Clearly this code won't compile for now, because we haven't defined a **`yield`** function: as we will see in this tutorial implementing it is indeed possible, and not very complicated either. Remember that each thread “lives” within the process container, in a shared memory space (that's why you get race conditions when accessing shared data). However some information is not shared between threads, for example the program counter, the stack, the execution state, etc. In the class we called this data the context of execution of a thread. In order to switch execution between threads we need a way to save and restore the context of execution. Ideally, we would need some data structure (supposedly called “execution_context”) and two methods to read and write data from and to it. For example:

```
execution_context thread1_ctx;
execution_context thread2_ctx;

void thread1() {
  for(int i=0; i<10000; i++) {
                // Do some computation
                save_context(thread1_ctx);
                restore_context(thread2_ctx);
        }
}
void thread2() {
      for(int i=0; i<10000; i++) {
                // Do some computation
                save_context(thread2_ctx);
                restore_context(thread1_ctx);
       }
}
void main() {
      while(1) {
                thread1();
                thread2();
       }
}
```

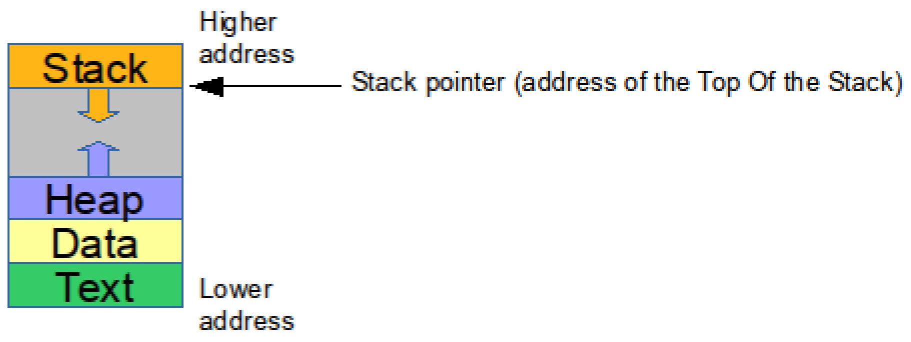
But first, what do we exactly need to save and restore? How can we practically save and restore that information? We will answer these questions starting from the next section, where we will detail the contents of an actual execution context.

3. The execution context

Threads are the active part of a process: each thread shares some part of the execution context with other threads (for example, the address space) but maintains some information required for its execution as “private”. Which of these items are different on each thread? As they pertain to the execution of instructions on the CPU this state information comprises registers, including the program counter (where the current instruction is found in memory) and the stack. As we will demonstrate in the following section, even though the stack resides somewhere in the addressing space of the process, manipulation depends just on two registers: the stack pointer register (typically named **SP**) and the base pointer register (**BP**, also known as **frame pointer**). The first points to the top of the stack, whereas the latter points to the beginning of the current stack frame. If we carefully assign different areas of memory to the stack of each thread (i.e. different values for the **SP** and **BP** registers) then we shouldn't run into any problems.

3.1 Digression: stack management (on x86) during execution

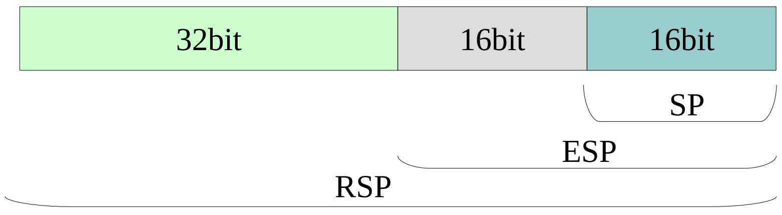
On the Intel x86 architecture the stack is one of the segments (contiguous memory areas) the process' address space is divided into. Other segments are the heap segment (which contains data allocated dynamically using, for example, **`malloc`** and **`free`**), the text segment (which contains the processor instructions of the program) and the data segment (typically divided between uninitialized data and initialized data). The stack grows downward, starting from the highest available address toward lower addresses:

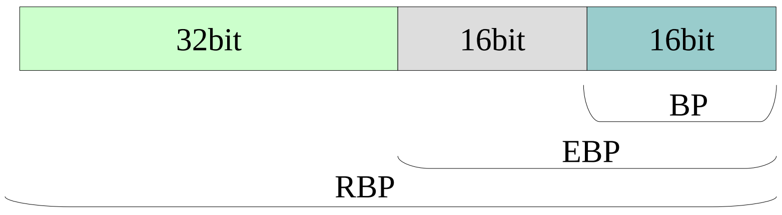


The stack is modified during execution to store local values, parameters during function calls, return values, etc. To understand how the stack grows and shrinks during execution we now focus on function calls: our objective is to show that contextual information about the stack can be reduced to the addresses stored in two registers (SP and BP).

3.1.1 A little bit of x86 assembler (GNU / AT&T syntax)

To understand the forthcoming discussion we have to introduce a little bit of x86 assembler. First of all, the Stack Pointer register might be named either ESP (Extended-) or RSP, on x86 32bit and 64bit architectures respectively (SP being 16bit wide). Similarly, the Base Pointer is referred to as EBP (Extended-) or RBP if accessed in 32bit or 64bit mode respectively (BP being 16bit wide):





Some important assembler instructions that we will encounter are:

- **mov *src, dest***
  - Copies the value from *src* to *dest* (**$value** for immediate values, **%reg** for registers, **offset(address)** for memory addresses;
- **push *src***

   - Pushes a new value (word) on the stack (and moves the stack pointer to a lower address);
- **pop *dest***
   - Removes the word at the top of the stack (and moves the stack pointer to a higher address);
- **call *fn***
   - Pushes the return address (address of the instruction following the call) on the stack and jumps to the specified function **fn** (changes the program counter). Function parameters must be already on the stack;
- **ret**
   - Removes the return address from the stack and leaves the function by jumping to this address;
- **leave**
   - Restores the stack pointer (using the base pointer), removes the value at the top of the stack and uses it as new base pointer;
- **add *src, dest***
   - Adds *src* to *dest* (the result is stored in *dest*);
- **sub *src, dest***
   - Subtracts *src* from *dest* (the result is stored in *dest*);

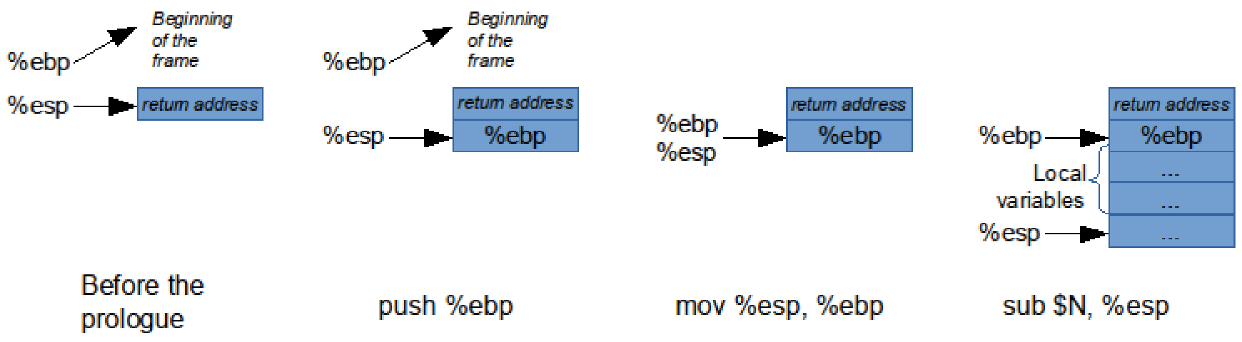
3.1.2 Function calls (x86)

To avoid overwriting existing local values, the function that gets called (referred to as, *the callee*) has to set up a new stack frame (the "private" execution environment of a function). The instructions that perform this task form what is called the **function prologue**: this prologue is required to ensure that we create the necessary space for local variable and that we are able to return at the correct address after the call, restoring the previous state of the stack. The prologue is comprised of the following assembler instructions:

```
① push    %ebp
② mov     %esp, %ebp
③ sub     $N,%esp
```

1) Save the EBP (base pointer) on the stack
2) Copy ESP to EBP
3) Move the stack pointer downward (the stack grows) to make room for local variables (the amount of displacement depends on how many variables are there)

Schematically we can represent the stack operations performed at the beginning of a function as follows:



At the end of a function there should be some cleanup code called **function epilogue** that restores the state of the stack (this code is equivalent to `leave` followed by a `ret`):

```
① mov     %ebp, %esp
② pop     %ebp
③ ret
```

1) Restore the stack pointer
2) Pop the value at the top of the stack and store it as new frame base pointer
3) Pop the return address from the stack and return to the caller

Schematically we can represent the stack operations performed at the end of a function as follows:
When the stack is shrinked, memory areas past the stack pointer are not cleared: contents are are technically still readable, but are not guaranteed to “survive” the next function call. It should be noted that base pointer is not really required for code execution : in fact, current compilers can generate code that uses just the stack pointer (the return address, local variables and parameter positions can be located also as offsets relative to the stack pointer address). For example, the GNU C compiler does not employ the base pointer when the **omit-frame-pointer** flag (or an **-Ox** optimization flag) is set. So, when should we keep the base pointer? Answer: if we need to debug our code, since the base pointer helps producing an execution back-trace by means of a debugger.

3.1.3 Call conventions

Function calls require an agreement between the caller and the callee, commonly referred to as the calling convention. Calling conventions vary between computer architectures: to pass parameters either the stack or the registers (or both) could be used, similarly the return value could be placed on the stack or in a register. Furthermore a calling convention specifies whether the caller or the callee should be responsible for setting up and cleaning up the stack [1].
As an example the C declaration (*cdecl*, typically used by C compilers on 32bit x86 systems) states that the function parameters should be passed on the stack, that the caller is responsible for cleaning up the stack (from the parameters pushed before the call), that parameters are pushed in inverse order, i.e. calling f(1,2,3) pushes 3, 2, and 1, and that return values are passed using a register (integer values or memory addresses, using the **EAX** register, floating point values using the **ST0** register). The caller is also responsible for saving some register (**EAX**, **ECX**, and **EDX**) as well as for cleaning some floating point registers prior to the call.
On 64bit x86 architectures (AMD64) another convention is used: System V AMD64 ABI. In this case the first six parameters are passed using registers (the remaining are still passed using the stack, which would need to be cleaned by the caller afterwards). Let's see how this works with some real C code:

```
int callee(int x, int y) {
    return;
}

void main() {
    int a,b;
    a = 13;
    b = 17;
    callee(a,b);
}
```

We can study the generated assembler instructions with the appropriate compiler switches (for gcc). Supposing that the above code is written in *example.c*, to generate 32bit (cdecl) assembler code in the *example.s32* file use:

```
gcc -m32 -mno-accumulate-outgoing-args -c -g -Wa,-ahl=example.s32 -fverbose-asm
example.c
```

To generate 64bit (System V AMD64 ABI) assembler in the *example.s64* file, use:

```
gcc -c -mno-accumulate-outgoing-args -g -Wa,-ahl=example.s64 -fverbose-asm example.c
```

Comparing the two outputs we can see the aforementioned differences (the `l` after some instructions denotes a variant which works with long words, i.e. 32bit) :

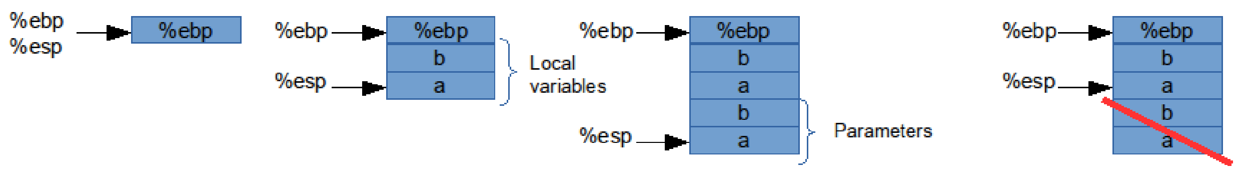
1 Further information can be found at https://en.wikipedia.org/wiki/X86_calling_conventions

```
// 32 bit code
   74 0021 FF75FC           pushl  -4(%ebp)      # b
   75 0024 FF75F8           pushl  -8(%ebp)      # a
   76 0027 E8FCFFFF         call   callee #
   76      FF
   77 002c 83C408           addl   $8, %esp      #,
```

Variables **a** and **b** are local to the main stack frame, thus references are done using an offset (-4 and -8) from the base pointer (integers are 4 bytes wide). Before the call, parameters are pushed on the stack in reverse order. The callee saves the values in temporary registers and then to local values in the stack frame. The result of the addition is stored in the **EAX** register (**%eax**) while the epilogue restores the stack and returns to the caller, as reflected in the following code excerpt:

```
1. // 32 bit code
2.    40 0000 55                pushl  %ebp   #
3.    43 0001 89E5              movl   %esp, %ebp    #,
4.    46 0003 8B450C            movl   12(%ebp), %eax      # y, tmp61
5.    47 0006 8B5508            movl   8(%ebp), %edx# x, tmp62
6.    48 0009 01D0              addl   %edx, %eax    # tmp62, D.1381
7.    50 000b 5D                popl   %ebp   #
8.    53 000c C3                ret
```

After the call the caller has to clean up the stack by moving the stack pointer 8 bytes up (the stack shrinks). This completes the procedure call and restores the stack.



Beginning of main()

Local variables in main()

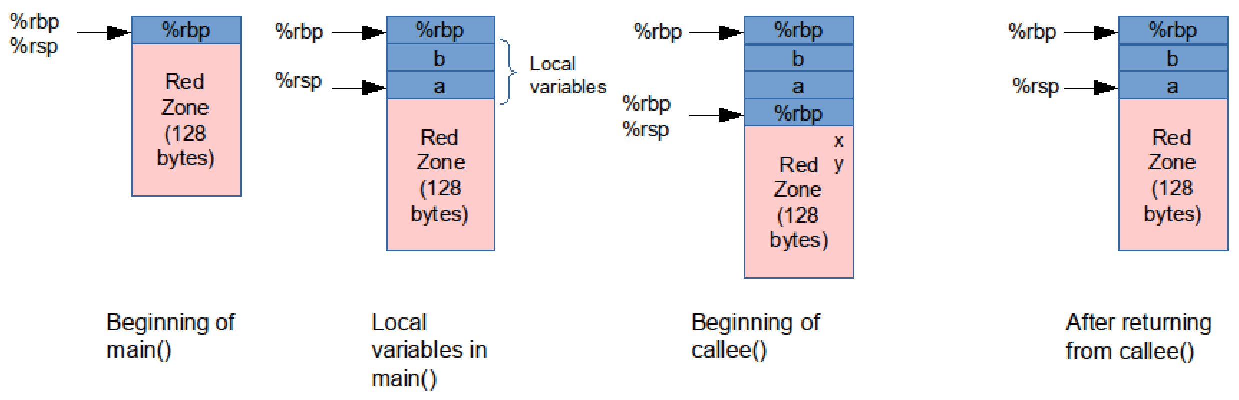
Before calling callee()

After returning from callee()

On a 64bit architecture parameters are passed using the **edi** and **esi** registers (the stack is used only if there are more than 6 parameters):

```
// 64 bit code
   71 0022 8B55FC           movl   -4(%rbp), %edx      # b, tmp59
   72 0025 8B45F8           movl   -8(%rbp), %eax      # a, tmp60
   73 0028 89D6             movl   %edx, %esi    # tmp59,
   74 002a 89C7             movl   %eax, %edi    # tmp60,
   75 002c E8000000         call   callee #
```

The callee retrieves these parameters and saves them as local variables (the q after some instruction denotes a variant which operates on quad words, i.e. 64bit, values). You might notice that the stack pointer is not shifted downwards as in the 32bit version, and that local variables are stored beyond the TOS (at a lower address than %rsp).



This behavior is not a bug of the compiler, but rather an optimization trick implemented on AMD64: “the 128-byte area beyond the location pointed to by %rsp is considered to be reserved and shall not be modified by signal or interrupt handlers. Therefore, functions may use this area for temporary data that is not needed across function calls. In particular, leaf functions may use this area for their entire stack frame, rather than adjusting the stack pointer in the prologue and epilogue. This area is known as the red zone.” [2]

```
// 64 bit code
0000 55                     pushq  %rbp   #
  43 0001 4889E5      movq  %rsp, %rbp   #,
  45 0004 897DFC      movl  %edi, -4(%rbp)      # x, x
  46 0007 8975F8      movl  %esi, -8(%rbp)      # y, y
  48 000a 8B45F8      movl  -8(%rbp), %eax      # y, tmp61
  49 000d 8B55FC      movl  -4(%rbp), %edx      # x, tmp62
  50 0010 01D0        addl  %edx, %eax   # tmp62, D.1734
  52 0012 5D          popq  %rbp   #
  54 0013 C3          ret
```

Because the values of the **x** and **y** parameters are used just to compute the return value (which is directly stored in a register) and never read again, the red zone is an ideal candidate for temporary storage. As expected the end of the function the stack pointer does not need need to be restored.

3.2 Back on route: toward user threads

As we saw in our digression there are just two informations that need to be stored in order to maintain some stack-related contextual information: the **SP** and **BP** registers. During function calls these registers are updated so that each stack frame is kept separated from the previous one. Accordingly, we proved that we could save the whole execution context just by saving all the register values! But how?

setjmp and longjmp

POSIX defines two functions to save the stack context (**`setjmp`** and **`sigsetjmp`**) and two functions to restore a saved state (**`longjmp`** and **`siglongjmp`**):

```
#include <setjmp.h>

int setjmp(jmp_buf env);
int sigsetjmp(sigjmp_buf env, int savesigs);

void longjmp(jmp_buf env, int val);
void siglongjmp(sigjmp_buf env, int val);
```

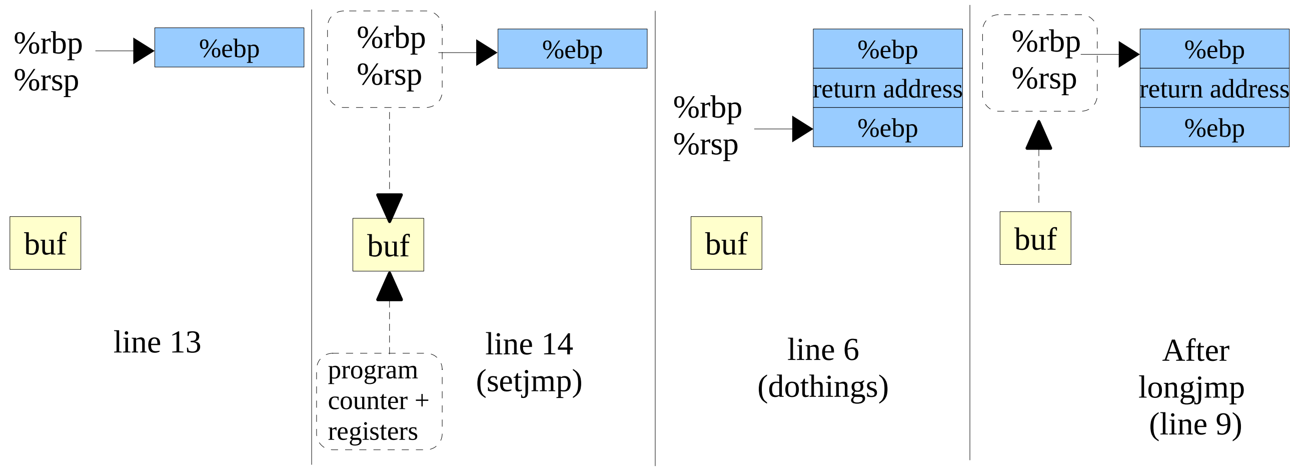
2 https://software.intel.com/sites/default/files/article/402129/mpx-linux64-abi.pdf

The **setjmp** function saves the current execution context (namely register values) in a **jmp_buf** structure. An alternative, named **sigsetjmp** also saves the process' current signal mask (if the **savesigs** parameter is nonzero). To restore a saved context we can use the **longjmp** function, which also requires a numeric value to be returned from the corresponding **setjmp** (which should be nonzero). If the signal mask should also be restored, **siglongjmp** is available. Saving the context means taking a snapshot of the current execution state, creating a point where execution can jump back to: the first time **setjmp** and **sigsetjmp** are called (when they save the current context) they return 0, whereas when they come back they return the value **val** passed to **longjmp**/**siglongjmp**.

3.2.1 Example

```
1.  #include <stdio.h>
2.  #include <setjmp.h>
3.
4.  static jmp_buf buf;
5.
6.      void dothings() {
7.         printf("Now I'm here\n");
8.         sleep(3);
9.         longjmp(buf, 42);
10.        printf("This is never printed\n");
11. }
12.
13. int main() {
14.     if (!setjmp(buf)) { // the first time returns 0
15.         dothings();
16.     } else {
17.         printf("Now I'm there\n");
18.     }
19.     return 0;
20. }
```

This program saves the execution context (at line 14) using **setjmp**. The first call returns 0, and the code continues calling the **dothings** procedure. This procedure prints a string on the console, sleeps for 3 seconds and then jumps back to the previously saved context. Accordingly, the text at line 10 is never printed, as execution resumes from line 14, with a return value equal to 42. Finally the string on line 17 is printed and the program exits. The evolution of the stack can be summarized as follows:



As you can see the contents of the stack after long jump are not lost, just the base pointer and the stack pointer are shifted. A minor drawback deriving from the use of **setjmp** and **longjmp** primitives is related to the use of special registers within thread's code, more specifically, floating point and flags registers are not saved. Nonetheless, to keep things simple, we accept this limitation and in code that might be interrupted by a context switch we will take care of storing data inside local variables declared as **volatile** (preventing the compiler from optimizing those values into registers).

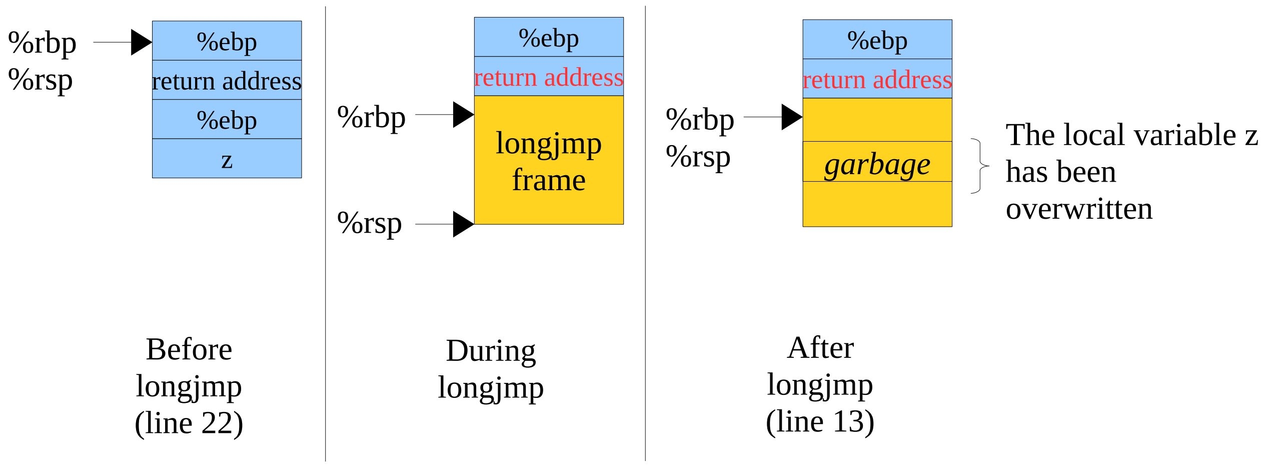
[1] Exercise: copy-paste, compile and test the previous example: try to understand the execution flow and how the stack changes. Change the code and experiment yourself if you like! (15 min)

3.2.2 Almost there...

Being able to save and restore the context enables us to implement a cooperative (non preemptive) thread mechanism: each thread maintains a stack context but needs to explicitly yield to other threads. To develop our solution we must however overcome one final problem... consider this example:

```
1.  #include <stdio.h>
2.  #include <setjmp.h>
3.  #include <stdlib.h>
4.
5.  static jmp_buf main_buf, dothings_buf;
6.
7.  void dothings() {
8.          int z = 1313;
9.          if(!setjmp(dothings_buf)) {
10.                 printf("Now I'm here, z=%d\n", z);
11.                 longjmp(main_buf, 42);
12.         } else {
13.                 printf("Now I'm back here, z=%d\n", z);
14.                 exit(0);
15.         }
16. }
17.
18. int main() {
19.     if (!setjmp(main_buf)) {
20.         dothings();
21.     } else {
22.         longjmp(dothings_buf, 17);
23.     }
24.     return 0;
25. }
```

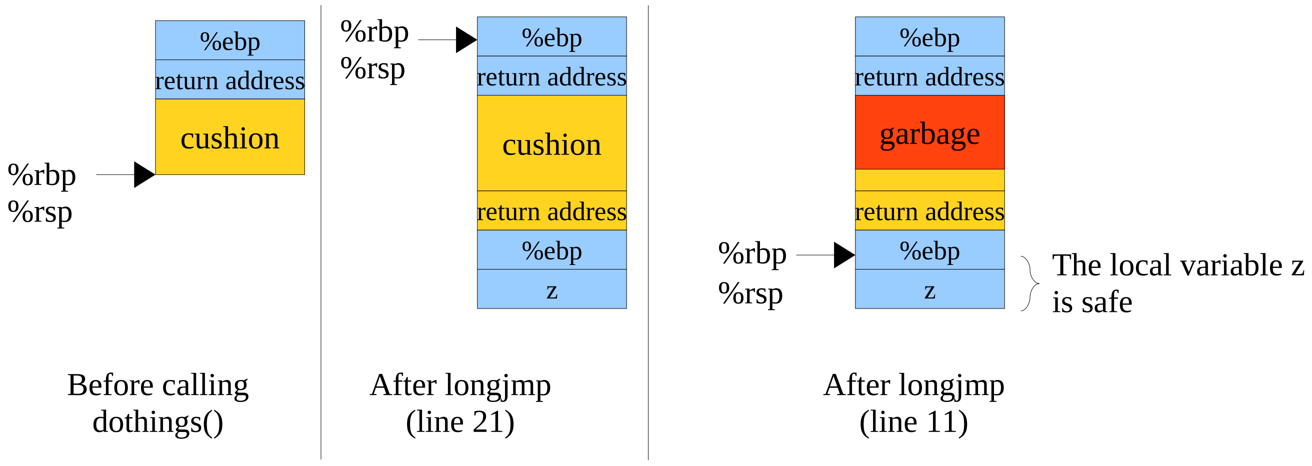
The stack context is saved in the **`main`** procedure, then execution jumps to the **`dothings`** function where the stack context is saved too (in another **`jmp_buf`**). Subsequently the value of the local variable **`z`** is printed and the **`main`** context is restored. Finally, the context of the **`dothings`** function is restored and the value of **`z`** is printed again. If we check out this example we will see that the second time **`z`** is printed its value is 0. What happened? It turns out we have just **smashed the stack**, that is, we overwrote some stack content. Where and why did it happen?



When jumping back from **dothings** to the **main** procedure, we call **longjmp**: this overwrites the **dothings** stack frame (because it was adjacent to the **main** stackframe), and thus the variable **z**.
To solve this problem we need to create some cushion ("empty" space) between the main stack frame and the **dothings** frame. This can be achieved by calling an intermediate procedure that takes up enough space on the stack and subsequently calls the **dothings** procedure.

```
1.  #include <stdio.h>
2.  #include <setjmp.h>
3.  #include <stdlib.h>
4.  static jmp_buf main_buf, dothings_buf;
5.  void dothings() {
6.         int z = 1313;
7.         if(!setjmp(dothings_buf)) {
8.                 printf("Now I'm here, z=%d\n", z);
9.                 longjmp(main_buf, 42);
10.        } else {
11.            printf("Now I'm back here, z=%d\n", z);
12.            exit(0);
13.        }
14. }
15. void cushion() {
16.        char data_cushion[1000];
17.        data_cushion[999] = 1;
18.        dothings();
19. }
20. int main() {
21.     if (!setjmp(main_buf)) {
22.         cushion();
23.     } else {
24.         longjmp(dothings_buf, 17);
25.     }
26.     return 0;
27. }
```

The cushion stack frame is sacrificed but prevents damage to the **dothings** frame. It should be noted that the cushion frame is compromised, thus **dothings** must not return (otherwise it would continue executing in the **cushion** procedure). Please also note that this isn't the only way to safely assign different execution contexts [3], but for this lab it's probably the easiest path to follow.



3 See for example: http://pubs.opengroup.org/onlinepubs/009695399/functions/sigaltstack.html

[2] **Exercise:** copy-paste, compile and test the previous example: try to understand the flow of execution (15 min)

4. Cooperative threads

We now combine the concepts learned in the previous sections to develop a first working (although hard-coded) example with three cooperative threads (which we name *bthreads*):

```c
#include <setjmp.h>
#include <stdio.h>

#define CUSHION_SIZE 10000
#define save_context(CONTEXT) setjmp(CONTEXT)
#define restore_context(CONTEXT) longjmp(CONTEXT, 1)

typedef enum { __BTHREAD_UNINITIALIZED, __BTHREAD_READY } bthread_state;
typedef void *(*bthread_routine) (void *);

void create_cushion_and_call(bthread_routine fn, bthread_state* state);
void* bthread1(void* arg);
void* bthread2(void* arg);
void* bthread3(void* arg);

jmp_buf bthread1_buf, bthread2_buf, bthread3_buf;
bthread_state bthread1_state = __BTHREAD_UNINITIALIZED;
bthread_state bthread2_state = __BTHREAD_UNINITIALIZED;
bthread_state bthread3_state = __BTHREAD_UNINITIALIZED;

void create_cushion_and_call(bthread_routine fn, bthread_state* state)
{
    char cushion[CUSHION_SIZE];
    cushion[CUSHION_SIZE-1] = cushion[0];
    *state = __BTHREAD_READY;
    fn(NULL);
}

void* bthread1(void* arg)
{
    volatile int i;
    for(i=0;i<10000;i++) {
        printf("BThread1, i=%d\n", i);
        /* Yield to next bthread */
        if (!save_context(bthread1_buf)) {
            if (bthread2_state == __BTHREAD_UNINITIALIZED) {
                create_cushion_and_call(bthread2, &bthread2_state);
            } else {
                restore_context(bthread2_buf);
            }
        }
    }
}

void* bthread2(void* arg)
{
    volatile int i;
    for(i=0;i<10000;i++) {
        printf("BThread2, i=%d\n", i);
        /* Yield to next bthread */
        if (!save_context(bthread2_buf)) {
            if (bthread3_state == __BTHREAD_UNINITIALIZED) {
                create_cushion_and_call(bthread3, &bthread3_state);
            } else {
                restore_context(bthread3_buf);
            }
        }
    }
}

void* bthread3(void* arg)
{
    volatile int i;
    for(i=0;i<10000;i++) {
        printf("BThread3, i=%d\n", i);
        /* Yield to next bthread */
        if (!save_context(bthread3_buf)) {
            // We assume that bthread1 is already initialized
            restore_context(bthread1_buf);
        }
    }
}

void main()
{
    create_cushion_and_call(bthread1, &bthread1_state);
}
```

Threads are represented by separate procedures, **bthread1**, **bthread2**, and **bthread3**. The stack context of each thread is maintained in three separate global variables of type **jmp_buf** named **bthread1_buf**, **bthread2_buf** and **bthread3_buf**. Additionally we need a variable to keep track of whether a thread has already been executed or not: this is important because the first call requires the additional step of creating a cushion frame. We achieve this goal by means of the **bthread_state** type, which can represent two possible states: uninitialized (implemented as **__BTHREAD_UNINITIALIZED**) and ready for execution (**__BTHREAD_READY**). All threads start in the uninitialized state, but when the cushion is created the state changes to ready. As you notice we defined a function pointer type **bthread_routine** which represents the "entry point" (or start routine) of our thread. The **create_cushion_and_call** is used to create a stack cushion and launch the thread that follows (which is hard-coded in this example): this function is called only the first time a thread is executed, because subsequent context switches will just need to jump to the previously saved context. To make our code more readable we also defined two macros: **save_context** and **restore_context**. Although it is not always necessary, in our code we should declare local variables that are required to survive a context switch as **volatile** because we need to prevent the compiler from caching the value in a CPU register: unfortunately ***setjmp*** does not save the whole set of registers (in particular, floating point registers are missing) and we want to prevent losing those values when restoring the context. For function arguments the cdecl (on 32 bit x86 systems) convention ensures that values are on the stack, however for System V AMD64 ABI uses registers, thus special care must be taken when dealing with floating point arguments.

---

[3] **Exercise:** Study, compile and test the proposed example code; try to add another thread and verify that your code executes as expected. What happens if a thread exits the for loop earlier than the others? How could you solve this problem? How could you implement a simple priority mechanism where one thread receives more CPU time than the other threads? (45 min)

4.1 Implementing a cooperative threads library

We now want to implement a more flexible approach to the problem. More specifically we are going to create some new types and a “container” structure for the thread context, which would store not only the stack context (**`jmp_buf`**) but also the thread's state and additional scheduling information. Because we do not want to expose this structure to the end-user, we define a thread identifier **`bthread_t`** and use it as a reference to the private data:

```
typedef unsigned long int bthread_t;
```

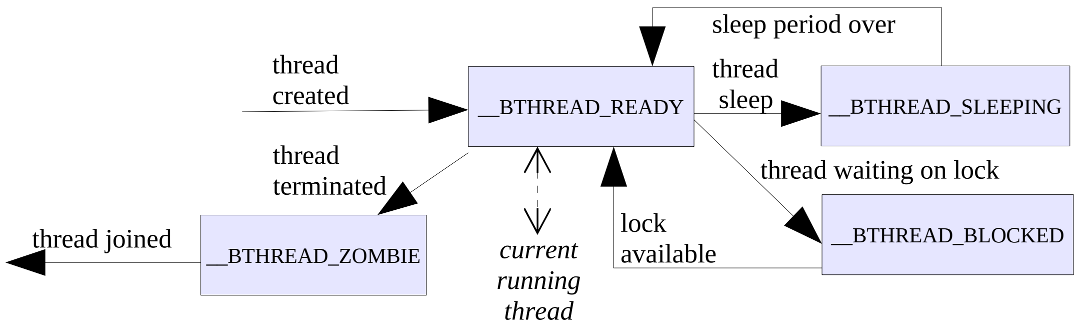
Threads are not always running, therefore we must keep track of their execution state:

```
typedef enum { __BTHREAD_READY = 0, __BTHREAD_BLOCKED, __BTHREAD_SLEEPING,
  __BTHREAD_ZOMBIE} bthread_state;
```

*Note: keep this order and do not add any state before **__BTHREAD_READY** in the definition, otherwise you will have conflicts (since enum types start from 0 by default).*

- **__BTHREAD_READY**
  - The thread is ready to be executed;
- **__BTHREAD_ZOMBIE**
  - The thread has exited but is waiting for another thread to join;
- **__BTHREAD_BLOCKED**
  - The thread is currently blocked due to some synchronization primitive (lock);
- **__BTHREAD_SLEEPING**
  - The thread is sleeping for some amount of time.

You might notice that there is no state that represents a running thread because this information is maintained directly by the scheduler (our library will support only one running thread at a time). State transitions depend on the behavior of a thread, and can be summarized as follows:



The thread structure itself contains all the necessary information to support execution:

```
typedef struct {
} bthread_attr_t;

typedef void *(*bthread_routine) (void *);

typedef struct {
    bthread_t tid;
    bthread_routine body;
    void* arg;
    bthread_state state;
    bthread_attr_t attr;
    char* stack;
    jmp_buf context;
    void* retval;
} __bthread_private;
```

The **bthread_routine** type specifies the signature of the thread body routine, whereas the **bthread_attr_t** structure stores additional thread attributes. This structure is not actually used in our implementation (as you will notice further on in this document), but is defined to maintain some similarity with the pthread[4] API. The **__bthread_private** structure contains all the information regarding a thread: an identifier, the body routine and its argument, an execution state, attributes, stack context, and a return value. Additional fields will be added later to support other features such sleep, cancellation requests and priorities.

---

[4] **Exercise:** Copy and save the thread structure code and all other private definitions in a file named **bthread_private.h .** Copy and save public definitions (namely **bthread_t**, **bthread_attr_t**, and **bthread_routine** typedefs in a file named **bthread.h** . (15 min)

4.1.1 Thread queue

Because we need to manage more than one thread in a flexible way will use a specific container: a queue implemented as a circular linked-list. This container, called **TQueue**, exposes different methods to manipulate the queue:

```
struct TQueueNode;
typedef struct TQueueNode* TQueue;

/* Adds a new element at the end of the list, returns its position */
unsigned long int tqueue_enqueue(TQueue* q, void* data);

/* Removes and returns the element at the beginning of the list, NULL if the 
queue is empty */
void* tqueue_pop(TQueue* q);

/* Returns the number of elements in the list */
unsigned long int tqueue_size(TQueue q);

/* Returns a 'view' on the list starting at (a positive) offset distance,
 * NULL if the queue is empty */
TQueue tqueue_at_offset(TQueue q, unsigned long int offset);

/* Returns the data on the first node of the given list */
void* tqueue_get_data(TQueue q);
```

The method **tqueue_at_offset** returns a pointer to the node at **offset** distance from the beginning of the given queue (wrapping around if necessary). To get the data from a node, the **tqueue_get_data** method must be employed. Each node (of type **TQueueNode**) in the linked-list contains a pointer to the next node, as well as a pointer to some data (which we will use as a pointer to a thread structure). The structure of the node, which will remain private (i.e. it will not be exposed in an header file) is:

```
typedef struct TQueueNode {
    struct TQueueNode* next;
```

4 http://pubs.opengroup.org/onlinepubs/7908799/xsh/pthread.h.html

```
    void* data;
} TQueueNode;
```

The last node points to the head of the queue, thus an empty **TQueue** is represented by a **NULL** pointer, i.e.:

```
TQueue t = NULL;
```

---

[5] **Exercise:** Download the source files for the queue from icorsi (**tqueue.c** and **tqueue.h**) and integrate them into your project (5 min)

4.1.2 Scheduler prototype

The scheduler is responsible for initializing and scheduling threads (according to some scheduling policy). In this regard, we define a structure to maintain some private data of the scheduler:

```
typedef struct {
    TQueue queue;
    TQueue current_item;
    jmp_buf context;
    bthread_t current_tid;
} __bthread_scheduler_private;
```

The list of threads is stored in the **queue** field, and **current_item** refer to the list node which contains a pointer to the currently executing thread. The **current_tid** field is used to generate unique thread identifiers. To interact with the scheduler and manage threads the following functions must be implemented:

```
__bthread_scheduler_private* bthread_get_scheduler(); // Private

int bthread_create(bthread_t *bthread, const bthread_attr_t *attr,
                   void *(*start_routine) (void *), void *arg);

int bthread_join(bthread_t bthread, void **retval);

void bthread_yield();

void bthread_exit(void *retval);

void bthread_cleanup(); // Private
```

The signature of these functions as well as their semantics have been kept similar to the *pthread* library in order to simplify porting applications to our user-thread library:

- **bthread_get_scheduler**
    - This private function creates, maintains and returns a static pointer to the singleton instance of **__bthread_scheduler_private**. Fields of this structure need to be initialized as NULL. Other functions will call this method to obtain this pointer. This function should not be accessible outside the library.

- **bthread_create**
    - Creates a new thread structure and puts it at the end of the queue. The thread identifier (stored in the buffer pointed by **bthread**) corresponds to the position in the queue. The thread is not

started when calling this function. Attributes passed through the **`attr`** argument are ignored (thus it is possible to pass a **NULL** pointer). The stack pointer for new created threads is **NULL**.

- **bthread_join**
  - Waits for the thread specified by **`bthread`** to terminate (i.e. **`__BTHREAD_ZOMBIE`** state), by scheduling all the threads. In the following we will discuss some details about this procedure.

- **bthread_yield**
  - Saves the thread context and restores (long-jumps to) the scheduler context. Saving the thread context is achieved using **`sigsetjmp`**, which is similar to **`setjmp`** but can also save the signal mask if the provided additional parameter is not zero (to restore both the context and the signal mask the corresponding call is **`siglongjmp`**). Saving and restoring the signal mask is required for implementing preemption.

- **bthread_exit**
  - Terminates the calling thread and returns a value via **`retval`** that will be available to another thread in the same process that calls **`bthread_join`**, then yields to the scheduler. Between **`bthread_exit`** and the corresponding **`bthread_join`** the thread stays in the **`__BTHREAD_ZOMBIE`** state.

Beside these functions it might be useful to define some additional helper procedures that should remain “private” (i.e. static) within the compilation unit, as well as the two macros that simplify saving and restoring an execution context:

```c
static int  bthread_check_if_zombie(bthread_t bthread, void **retval);
static TQueue bthread_get_queue_at(bthread_t bthread);
#define save_context(CONTEXT) sigsetjmp(CONTEXT, 1)
#define restore_context(CONTEXT) siglongjmp(CONTEXT, 1)
```

- **bthread_check_if_zombie**
  - Checks whether the thread referenced by the parameter **bthread** has reached a zombie state. If it's not the case the function returns 0. Otherwise the following steps are performed: if **retval** is not **`NULL`** the exit status of the target thread (i.e. the value that was supplied to **`bthread_exit`**) is copied into the location pointed to by **`*retval;`** the thread's stack is freed and the thread's private data structure is removed from the queue (*Note: depending on your implementation, you might need to pay attention to the special case where the scheduler's **queue** pointer itself changes!)*; finally the function returns 1.
- **bthread_get_queue_at**
  - Returns a "view" on the queue beginning at the node containing data for the thread identified by **bthread**. If the queue is empty or doesn't contain the corresponding data this function returns NULL.

4.2 Come and join the scheduling loop

The scheduler loop must be started somewhere to initiate the thread execution, but we do not want to introduce a specific procedure. By reasoning we can find a good place to put our scheduler loop: **`bthread_join`**. Because we are implementing user-space threads, each thread executes only while the containing process exists. Accordingly, we can limit our implementation to *joinable* threads and “force” the programmer to join each thread before exiting the program: this way, we ensure that the **bthread_join** procedure will always be called. Here is the skeleton of this function:

```
int bthread_join(bthread_t bthread, void **retval)
{
    volatile __bthread_scheduler_private* scheduler = bthread_get_scheduler();
    scheduler->current_item = scheduler→queue;
    save_context(scheduler→context);
    if (bthread_check_if_zombie(bthread, retval)) return 0;
    __bthread_private* tp;
    do {
       scheduler->current_item = tqueue_at_offset(scheduler->current_item, 1);
       tp = (__bthread_private*) tqueue_get_data(scheduler->current_item);
    } while (tp->state != __BTHREAD_READY);
       // Restore context or setup stack and perform first call
       if (tp->stack) {
              restore_context(tp->context);
       } else {
              tp->stack = (char*) malloc(sizeof(char) * STACK_SIZE);
       #if __x86_64__
              asm __volatile__("movq %0, %%rsp" ::
                     "r"((intptr_t) (tp->stack + STACK_SIZE - 1)));
       #else
              asm __volatile__("movl %0, %%esp" ::
                     "r"((intptr_t) (tp->stack + STACK_SIZE - 1)));
       #endif
              bthread_exit(tp->body(tp->arg));
       }

}
```

First, the **current_item** pointer is reset to point to the start of the **queue**: this step is important as we need to ensure that we always point to a valid element. Next, the scheduler's context is saved: this allows for returning to the scheduler by simply restoring its context. The scheduling loop cycles through all threads until one in the **__BTHREAD_READY** state is found. Please note that each time the loop is repeated we move the **current_item** pointer one position forward. Once a suitable thread has been found we check if its stack has already been initialized, if so we restore the thread's context. Otherwise we setup a new stack (allocating memory on the heap!) and, using a little bit of assembler, we change the stack pointer register. Finally, we call the thread's routing (wrapping it inside **bthread_exit** to ensure that the thread cannot escape). Did you notice that there is no cushion, in contrast to the "hard-coded" example? The answer is simple: our first idea for creating separate stacks was more of a "hack" than a proper solution, and would prevent us from dynamically create and destroy a large number of threads (since we would not be able to recover already allocated stack space).

[6] **Exercise:** Implement the aforementioned scheduler methods (the prototypes for “public” methods go into **bthread.h**, whereas all private prototypes and the implementation goes into **bthread_private.h** and **bthread.c**). Add code to handle zombie threads in **bthread_join** (180 min)

4.3 Yield to the scheduler

In our current implementation threads must explicitly yield to the scheduler (because preemption is not yet available): a good way to ensure periodical yields is to implement and use a yielding *printf* variant using this simple macro:

```
#define bthread_printf(...)        \
    printf(__VA_ARGS__); \
    bthread_yield();
```

4.4 Thread sleep

Threads might decide to sleep for a while using the following procedure:

```
void bthread_sleep(double ms);
```

The **ms** parameter specifies the number of milliseconds the thread must sleep. To implement sleeping we must add a new field in the thread structure:

```
double wake_up_time;
```

When calling **bthread_sleep** the state of the thread is set to **__BTHREAD_SLEEPING** and then the thread must yield to the scheduler. The wake up time is computed from the current time, obtained using:

```
double get_current_time_millis()
{
    struct timeval tv;
    gettimeofday(&tv, NULL);
    return (tv.tv_sec) * 1000 + (tv.tv_usec) / 1000 ;
}
```

Each time the thread is considered for execution the scheduler must first check if it's sleeping: in that situation the scheduler compares the current time with the wake up time and if necessary changes the state to **__BTHREAD_READY** to enable execution.

[7] **Exercise:** Implement and test **bthread_sleep**. (⏱ 45 min)

4.5 Cancellation points

As we saw in the class, a thread can also request cancellation of another thread: cancellation happens as soon as the thread receiving the request calls **testcancel**. To implement cancellation points in our library we need to keep track of cancellation requests by adding a flag to the thread structure which will be initialized to 0 and set to 1 when someone ask for cancellation:

```
    int cancel_req;
```

Cancellation (through **bthread_exit**) happens when the recipient thread executes **bthread_testcancel** . The return value of a cancelled thread is -1. The prototypes of these methods are:

```
int bthread_cancel(bthread_t bthread);

void bthread_testcancel(void);
```

[8] **Exercise:** Implement and test **bthread_cancel** and **bthread_testcancel**. (⏱ 45 min)

4.6 Preemption using a timer

Preemption prevents a thread from never releasing control to the scheduler (through **bthread_yield**), and can be implemented by means of a timer signal that periodically interrupts the executing thread and returns control to the scheduler. To set such a timer we employ **setitimer**:

```c
static void bthread_setup_timer()
{
    static bool initialized = false;

    if (!initialized) {
        signal(SIGVTALRM, (void (*)()) bthread_yield);
        struct itimerval time;
        time.it_interval.tv_sec = 0;
        time.it_interval.tv_usec = QUANTUM_USEC;
        time.it_value.tv_sec = 0;
        time.it_value.tv_usec = QUANTUM_USEC;
        initialized = true;
        setitimer(ITIMER_VIRTUAL, &time, NULL);
    }
}
```

The length of the quantum (in microseconds, a millionth of a second) is defined by the macro **QUANTUM_USEC**. Each time the alarm fires a **SIGVTALRM** is sent to the process and the **bthread_yield** procedure is called. Note: if the above solution does not work on your system replace **ITIMER_VIRTUAL** with **ITIMER_REAL**. The signal handler executes on the current thread's stack, hence **bthread__yield** will save the correct execution context. Preemption can also cause some issues in our library. For example, if the scheduler is preempted we might generate a race condition. To avoid this problem we to define two “private” procedures to temporarily block/unblock the timer signal (using **sigprocmask**):

```c
void bthread_block_timer_signal();
void bthread_unblock_timer_signal();
```

Timer signals must be enabled when thread execution is started/resumed but must be blocked prior to any context save (**sigsetjmp**), for example during **bthread__yield**. These procedures can be called from threads (which could be dangerous) but can also provide some initial low-level atomicity (useful for implementing synchronization primitives). Finally, to ensure that access to the console does not end up in a deadlock (since we are using signals), an async-safe *printf* function must be implemented (replacing the previous definition of the same-name macro). Here instead of a variadic macro we use a variadic function:

```c
void bthread_printf(const char* format, …) // requires stdlib.h and stdarg.h
{
        bthread_block_timer_signal();
        va_list args;
        va_start (args, format);
        vprintf (format, args);
        va_end (args);
        bthread_unblock_timer_signal();
}
```

[9] **Exercise:** Complete the library by implementing preemption. You might want to try different quantums lengths to determine the best value. Pay attention to functions that should remain “private” within your library. Also, ensure that variables declared in preemptable regions of code that need to be

preserved across context switches are declared as **`volatile`** to prevent the compiler from caching the value in a CPU register (⏲ 90 min)

5. Experimenting with different scheduling policies

Up until now our library implements a simple round-robin scheduling policy. It might be interesting to implement other scheduling policies, such as priority scheduling, lottery scheduling or random scheduling. To allow the programmer choose the most appropriate policy we implement a customizable scheduling architecture. First we define a signature for the scheduling procedure:

```c
typedef void (*bthread_scheduling_routine)();
```

Upon execution the scheduling procedure will update the **current_item** pointer in the scheduler's private structure. The actual scheduling procedure is stored as a field in the same structure:

```c
bthread_scheduling_routine scheduling_routine;
```

The scheduling loop will also need to be modified to take into account the custom scheduling procedure (a fallback to round-robin scheduling in the case of a NULL pointer must also be considered).

[10] **Exercise:** implement **random scheduling** and **priority scheduling** functions. You are going to modify the thread structure to add the necessary scheduling parameters, moreover you would need to implement a public function to set the global scheduling policy and per-thread parameters (for example, thread priority). Concerning priority scheduling a simple scheme can be implemented, with higher priority threads receiving a longer quantum or more than one consecutive quantum. FInally implement lottery scheduling. *Do not forget automated testing routines! (⏲ 180 min)*

6. Synchronization primitives

Among the last pieces of the thread library are synchronization primitives, namely mutexes, semaphores, barriers, and condition variables. Their implementation will enable us to run the *"classical problems"* we saw (or will see) during the class. We start with the mutex (**tmutex.h**), and leave other primitives as an exercise:

```c
#ifndef __TMUTEX_H__
#define __TMUTEX_H__

#include "tqueue.h"

typedef struct {
    void* owner;
    TQueue waiting_list;
} bthread_mutex_t;

// Defined only for "compatibility" with pthread
typedef struct {} bthread_mutexattr_t;

// attr is ignored
int bthread_mutex_init(bthread_mutex_t* m, const bthread_mutexattr_t *attr);

int bthread_mutex_destroy(bthread_mutex_t* m);

int bthread_mutex_lock(bthread_mutex_t* m);

int bthread_mutex_trylock(bthread_mutex_t* m);

int bthread_mutex_unlock(bthread_mutex_t* m);

#endif
```

And here is the implementation of the mutex (**tmutex.c**):

```c
#include <assert.h>
#include "tmutex.h"
#include "bthread_private.h"

int bthread_mutex_init(bthread_mutex_t* m, const bthread_mutexattr_t *attr)
{
    assert(m != NULL);
    m->owner = NULL;
    m->waiting_list = NULL;
    return 0;
}

int bthread_mutex_destroy(bthread_mutex_t* m)
{
    assert(m->owner == NULL);
    assert(tqueue_size(m->waiting_list) == 0);
    return 0;
}

int bthread_mutex_lock(bthread_mutex_t* m)
{
    bthread_block_timer_signal();
    __bthread_scheduler_private* scheduler = bthread_get_scheduler();
    volatile __bthread_private* bthread =  (__bthread_private*) tqueue_get_data(scheduler->current_item);
    if (m->owner == NULL) {
        m->owner = bthread;
        bthread_unblock_timer_signal();
    } else {
        bthread->state = __BTHREAD_BLOCKED;
        tqueue_enqueue(&m->waiting_list, bthread);
        while(bthread->state != __BTHREAD_READY) {
            bthread_yield();
        };
    }
    return 0;
}

int bthread_mutex_trylock(bthread_mutex_t* m)
{
    bthread_block_timer_signal();
    __bthread_scheduler_private* scheduler = bthread_get_scheduler();
    __bthread_private* bthread =  (__bthread_private*) tqueue_get_data(scheduler->current_item);
    if (m->owner == NULL) {
        m->owner = bthread;
        bthread_unblock_timer_signal();
    } else {
        bthread_unblock_timer_signal();
        return -1;
    }
    return 0;
}

int bthread_mutex_unlock(bthread_mutex_t* m)
{
    bthread_block_timer_signal();
    assert(m->owner != NULL);
    assert(m->owner == tqueue_get_data(scheduler->current_item));
    __bthread_private* unlock = tqueue_pop(&m->waiting_list);
    if (unlock != NULL) {
        m->owner = unlock;
        unlock->state = __BTHREAD_READY;
        bthread_yield();
        return 0;
    } else {
        m->owner = NULL;
    }
    bthread_unblock_timer_signal();
    return 0;
}
```

When a thread tries to lock the mutex we check if it's available. If not, we put the thread in the **__BTHREAD_BLOCKED** state. To avoid race-conditions we must execute mutex procedures atomically with the timer signal disabled. As you can see we re-use our queue to store the list of threads that are currently waiting for the mutex. When the mutex is released we pick the first thread from the queue and resume its execution.
The interface of other synchronization primitives is as follows, starting with the semaphore:

```c
#ifndef __TSEMAPHORE_H__
#define __TSEMAPHORE_H__

#include "tqueue.h"

typedef struct {
    int value;
    TQueue waiting_list;
} bthread_sem_t;

// pshared is ignored, defined for compatibility with pthread
int bthread_sem_init(bthread_sem_t* m, int pshared, int value);

int bthread_sem_destroy(bthread_sem_t* m);

int bthread_sem_wait(bthread_sem_t* m);

int bthread_sem_post(bthread_sem_t* m);

#define bthread_sem_up(s) \
       bthread_sem_post(s);

#define bthread_sem_down(s) \
       bthread_sem_wait(s);
#endif
```

Barrier:

```c
#ifndef __TBARRIER_H__
#define __TBARRIER_H__

#include "tqueue.h"
```

```c

typedef struct {
    TQueue waiting_list;
    unsigned count;
    unsigned barrier_size;
} bthread_barrier_t;

// Defined only for "compatibility" with pthread
typedef struct {} bthread_barrierattr_t;

// attr is ignored
int bthread_barrier_init(bthread_barrier_t* b,
                                       const bthread_barrierattr_t* attr,
                                       unsigned count);

int bthread_barrier_destroy(bthread_barrier_t* b);

int bthread_barrier_wait(bthread_barrier_t* b);

#endif
```

Condition variable:

```c
#ifndef __TCONDITION_H__
#define __TCONDITION_H__

#include "tqueue.h"
#include "tmutex.h"

typedef struct {
    TQueue waiting_list;
} bthread_cond_t;

// Defined only for "compatibility" with pthread
typedef struct {} bthread_condattr_t;

// attr is ignored
int bthread_cond_init(bthread_cond_t* c, const bthread_condattr_t *attr);

int bthread_cond_destroy(bthread_cond_t* c);

int bthread_cond_wait(bthread_cond_t* c, bthread_mutex_t* mutex);

int bthread_cond_signal(bthread_cond_t* c);

int bthread_cond_broadcast(bthread_cond_t* c);


#define bthread_cond_notify(s) \
       bthread_cond_signal(s);

#define bthread_cond_notifyall(s) \
       bthread_cond_broadcast(s);

#endif
```

[11] Exercise: Implement the remaining synchronization primitives (semaphore, barrier, and condition variable). Test and verify that mutual exclusion works correctly. (360 min)

7. Execution tracing

In order to provide some facilities for debugging we want to implement a tracing facility that outputs some information on the *standard output* while a program using the library is executing. For that we need to implement a macro that we will use whenever something interesting happens inside our program (thread creation, thread execution, state changes, etc.)

```
#ifdef TRACING
#define trace(...) printf (stderr, __VA_ARGS__)
#else
#define trace(...)
#endif
```

[12] Exercise: add the **trace** macro to your code and add trace points. Also implement an **atomic_trace** macro to be used when pre-emption is enabled. (45 min)

8. Classic synchronization problems

In this last part of the laboratory we report the source code of classic synchronization problems, which can be studied and used to verify the user-level thread library.

8.1 Dining philosophers

```
#include <stdio.h>
#include <stdlib.h>
#include <stdint.h>
#include "bthread.h"
#include "tmutex.h"
#include "tsemaphore.h"

#define N 5
#define LEFT(i) (i+N-1) % N
#define RIGHT(i) (i+1) % N
#define THINKING 0
#define HUNGRY 1
#define EATING 2

int state[N];
bthread_mutex_t mutex;
bthread_sem_t philosopher_semaphore[N];

void* philosopher (void* arg);
void take_forks(int i);
void put_forks(int i);
void test(int i);
void think(int i);
void eat(int i);

void think(int i)
{
    bthread_printf("Philosopher %d is thinking...\n", i);
    bthread_sleep(200);
}

void eat(int i)
{
```

```c
    bthread_printf("Philosopher %d is eating...\n", i);
    bthread_sleep(300);
}

void* philosopher (void* arg)
{
    volatile int i;
    i = (intptr_t) arg;
    while(1) {
        think(i);
        take_forks(i);
        eat(i);
        put_forks(i);
    }
    bthread_printf("\tPhilosopher %d dead\n", i);
}

void take_forks(int i)
{
    bthread_mutex_lock(&mutex);
    state[i] = HUNGRY;
    bthread_printf("\tPhilosopher %d is hungry\n", i);
    test(i);
    bthread_mutex_unlock(&mutex);
    bthread_sem_wait(&philosopher_semaphore[i]);
}

void put_forks(int i)
{
    bthread_mutex_lock(&mutex);
    state[i] = THINKING;
    test(LEFT(i));
    test(RIGHT(i));
    bthread_mutex_unlock(&mutex);
}

void test(int i)
{
    if (state[i] == HUNGRY
        && state[LEFT(i)] != EATING
        && state[RIGHT(i)] != EATING) {
        state[i] = EATING;
        bthread_printf("\tPhilosopher %d got forks\n", i);
        bthread_sem_post(&philosopher_semaphore[i]);
    }
}

int main(int argc, char *argv[])
{
    int j;
    for (j=0; j<N; j++) {
        bthread_sem_init(&philosopher_semaphore[j], 0, 0);
        state[j] = THINKING;
    }
    bthread_mutex_init(&mutex, NULL);

    volatile bthread_t philosophers[N];
    int i;
    for (i=0; i<N; i++) {
        bthread_create(&philosophers[i], NULL, philosopher, (void*) (intptr_t) i);
    }

    for (i=0; i<N; i++) {
        bthread_join(philosophers[i], NULL);
    }

        for (j=0; j<N; j++) {
            bthread_sem_destroy(&philosopher_semaphore[j]);
        }
        bthread_mutex_destroy(&mutex);
        printf("Exiting main\n");
        return 0;
    }
```

8.2 Producer consumer

```c
#include <stdio.h>
#include <stdlib.h>
#include "bthread.h"
#include "tmutex.h"
#include "tsemaphore.h"

#define N 10

int stock = 0;
bthread_mutex_t stock_mutex;
bthread_sem_t items_in_stock;
bthread_sem_t space_in_stock;

void* producer(void* arg)
{
    for(;;) {
        bthread_sem_wait(&space_in_stock);
        bthread_mutex_lock(&stock_mutex);
        if (stock < N) {
            /* Produce an item */
            stock = stock + 1;
            bthread_printf("Producer: now there are %d items\n", stock);
        } else {
            bthread_printf("Producer: stock is full\n");
        }
        bthread_mutex_unlock(&stock_mutex);
        bthread_sem_post(&items_in_stock);
    }
}

void* consumer(void* arg)
{
    for(;;) {
        bthread_sem_wait(&items_in_stock);
        bthread_mutex_lock(&stock_mutex);
        if (stock > 0) {
            /* Consume an item */
            stock = stock - 1;
            bthread_printf("Consumer: now there are %d items\n", stock);
        } else {
            bthread_printf("Consumer: stock is empty\n");
        }
        bthread_mutex_unlock(&stock_mutex);
        bthread_sem_post(&space_in_stock);
    }

}

int main(int argc, char *argv[])
{
    bthread_mutex_init(&stock_mutex, NULL);
    bthread_sem_init(&items_in_stock, 0, 0);
    bthread_sem_init(&space_in_stock, 0, N);

    volatile bthread_t producer1, consumer1, consumer2;

    bthread_create(&producer1, NULL, producer, NULL);
    bthread_create(&consumer1, NULL, consumer, NULL);
    bthread_create(&consumer2, NULL, consumer, NULL);

    bthread_join(producer1, NULL);
    bthread_join(consumer1, NULL);
    bthread_join(consumer2, NULL);

    bthread_sem_destroy(&items_in_stock);
    bthread_sem_destroy(&space_in_stock);
    bthread_mutex_destroy(&stock_mutex);
    printf("Exiting main\n");
    return 0;
}
```

8.3 Producer consumer with condition variable

```
#include <stdio.h>
#include <stdlib.h>
#include "bthread.h"
#include "tmutex.h"
#include "tcondition.h"

#define N 10

int stock = 0;
bthread_mutex_t stock_mutex;
bthread_cond_t items_in_stock;
bthread_cond_t space_in_stock;

void* producer(void* arg)
{
    for(;;) {
        bthread_mutex_lock(&stock_mutex);

        while (stock == N) {
            bthread_cond_wait(&space_in_stock, &stock_mutex);
        }

        if (stock < N) {
            /* Produce an item */
            stock = stock + 1;
            bthread_printf("Producer: now there are %d items\n", stock);
        } else {
            bthread_printf("Producer: stock is full\n");
        }
        bthread_mutex_unlock(&stock_mutex);
        bthread_cond_signal(&items_in_stock);
    }
}

void* consumer(void* arg)
{
    for(;;) {
        bthread_mutex_lock(&stock_mutex);

        while (stock == 0) {
            bthread_cond_wait(&items_in_stock, &stock_mutex);
        }

        if (stock > 0) {
            /* Consume an item */
            stock = stock - 1;
            bthread_printf("Consumer: now there are %d items\n", stock);
        } else {
            bthread_printf("Consumer: stock is empty\n");
        }
        bthread_mutex_unlock(&stock_mutex);
        bthread_cond_signal(&space_in_stock);
    }

}

int main(int argc, char *argv[])
{
    bthread_mutex_init(&stock_mutex, NULL);
    bthread_cond_init(&items_in_stock, NULL);
    bthread_cond_init(&space_in_stock, NULL);

    volatile bthread_t producer1, consumer1, consumer2;

    bthread_create(&producer1, NULL, producer, NULL);
    bthread_create(&consumer1, NULL, consumer, NULL);
    bthread_create(&consumer2, NULL, consumer, NULL);

    bthread_join(producer1, NULL);
    bthread_join(consumer1, NULL);
    bthread_join(consumer2, NULL);

    bthread_cond_destroy(&items_in_stock);
    bthread_cond_destroy(&space_in_stock);
    bthread_mutex_destroy(&stock_mutex);
    printf("Exiting main\n");
    return 0;
}
```

8.4 Sleeping barber

```
#include <stdio.h>
#include <stdlib.h>
#include "bthread.h"
#include "tmutex.h"
#include "tsemaphore.h"
#include <stdbool.h>

#define CHAIRS 3

typedef enum BarberState { WORKING, SLEEPING } BarberState;
BarberState state = SLEEPING;
int customers_count = 0;
bthread_mutex_t customers_count_state_mutex;
bthread_sem_t customers_semaphore;
bthread_sem_t barber_semaphore;

void do_hairs()
{
    bthread_printf("Cutting... cutting... cutting...\n");
}

void get_hairdo()
```

```c
{
    bthread_printf(":)\n");
}

void* barber(void* arg)
{
    for(;;) {
        bthread_mutex_lock(&customers_count_state_mutex);
        if (customers_count > 0) {
            state = WORKING;
            bthread_sem_post(&customers_semaphore);
            customers_count -= 1;
            bthread_mutex_unlock(&customers_count_state_mutex);
            do_hairs();
        } else {
            state = SLEEPING;
            bthread_mutex_unlock(&customers_count_state_mutex);
            bthread_printf("%s - Going to sleep\n", (char*) arg);
            bthread_sem_wait(&barber_semaphore);
            bthread_printf("%s - Yaaws!\n", (char*) arg);
        }
    }
}

void* customer(void* arg)
{
    for(;;) {
        bthread_mutex_lock(&customers_count_state_mutex);
        if (state == SLEEPING) {
            bthread_sem_post(&barber_semaphore);
            bthread_printf("%s - Wake up barber!\n", (char*) arg);
        }
        if (customers_count < CHAIRS) {
            customers_count += 1;
            bthread_mutex_unlock(&customers_count_state_mutex);
            bthread_sem_wait(&customers_semaphore);
            get_hairdo();
        } else {
            bthread_mutex_unlock(&customers_count_state_mutex);
            bthread_printf("%s - No chairs, I'm going home...\n", (char*) arg);
        }
        bthread_sleep(1500);
    }
}

int main(int argc, char *argv[])
{
    volatile bthread_t b, c1, c2;

    bthread_mutex_init(&customers_count_state_mutex, NULL);
    bthread_sem_init(&customers_semaphore, 0, 0);
    bthread_sem_init(&barber_semaphore, 0, 0);

    bthread_create(&b, NULL, &barber, (void*) "Figaro");
    bthread_create(&c1, NULL, &customer, (void*) "Count Almaviva");
    bthread_create(&c2, NULL, &customer, (void*) "Rosina");

    bthread_join(b, NULL);
    bthread_join(c1, NULL);
    bthread_join(c2, NULL);

    bthread_sem_destroy(&barber_semaphore);
    bthread_sem_destroy(&customers_semaphore);
    bthread_mutex_destroy(&customers_count_state_mutex);
    printf("Exiting main\n");

    return 0;
}
```

8.5 Readers and writers (solution favorable to readers)

```c
#include <stdio.h>
#include <stdlib.h>
#include "bthread.h"
#include "tsemaphore.h"

bthread_sem_t readers_count_sem;
bthread_sem_t db_sem;
int readers_count = 0;

void read_data()
{
    bthread_printf("Reading data\n");
}

void process_data()
{
    bthread_printf("Processing data\n");
}

void* reader(void* arg)
{
    for(;;) {
        bthread_sem_wait(&readers_count_sem);
        readers_count += 1;
        if (readers_count == 1) {
            bthread_sem_wait(&db_sem);
        }
        bthread_printf("There are %d readers\n", readers_count);
        bthread_sem_post(&readers_count_sem);
        read_data();
        bthread_sem_wait(&readers_count_sem);
        readers_count -= 1;
        bthread_printf("There are %d readers\n", readers_count);
        if (readers_count == 0) {
            bthread_sem_post(&db_sem);
        }
        bthread_sem_post(&readers_count_sem);
        process_data();
    }
}

void produce_data()
{
    bthread_printf("Produce data\n");
}

void write_data()
{
    bthread_printf("Writing data\n");
    bthread_sleep(2000);
}

void* writer(void* arg)
{
    for(;;) {
        produce_data();
        bthread_sem_wait(&db_sem);
        write_data();
        bthread_sem_post(&db_sem);
        bthread_sleep(1000);
    }
}


int main(int argc, char *argv[])
{
    bthread_sem_init(&db_sem, 0, 1);
    bthread_sem_init(&readers_count_sem, 0, 1);

    volatile bthread_t readers[5];
    volatile bthread_t writers[5];

    int i;
    for (i=0; i<5; i++) {
        bthread_create(&readers[i], NULL, reader, NULL);
        bthread_create(&writers[i], NULL, writer, NULL);
    }

    for (i=0; i<5; i++) {
        bthread_join(readers[i], NULL);
        bthread_join(writers[i], NULL);
    }

    bthread_sem_destroy(&readers_count_sem);
    bthread_sem_destroy(&db_sem);

    return 0;
}
```

8.6 Readers and writers (solution favorable to writers)

```c
#include <stdio.h>
#include <stdlib.h>
#include "bthread.h"
#include "tsemaphore.h"

bthread_sem_t readers_pass_sem;
bthread_sem_t readers_count_sem;
bthread_sem_t writers_count_sem;
bthread_sem_t writers_pass_sem;
bthread_sem_t db_sem;
int readers_count = 0;
int writers_count = 0;

void read_data()
{
    bthread_printf("Reading data\n");
}

void process_data()
{
    bthread_printf("Processing data\n");
}

void* reader(void* arg)
{
    for(;;) {
        bthread_sem_wait(&readers_pass_sem);
        bthread_sem_wait(&writers_pass_sem);
        bthread_sem_wait(&readers_count_sem);
        readers_count += 1;
        if (readers_count == 1) {
            bthread_sem_wait(&db_sem);
        }
        bthread_printf("There are %d readers\n", readers_count);
        bthread_sem_post(&readers_count_sem);
        bthread_sem_post(&writers_pass_sem);
        bthread_sem_post(&readers_pass_sem);

        read_data();
        bthread_sem_wait(&readers_count_sem);
        readers_count -= 1;
        bthread_printf("There are %d readers\n", readers_count);
        if (readers_count == 0) {
            bthread_sem_post(&db_sem);
        }
        bthread_sem_post(&readers_count_sem);
        process_data();
    }
}

void produce_data()
{
    bthread_printf("Produce data\n");
}

void write_data()
{
    bthread_printf("Writing data\n");
    bthread_sleep(2000);
}

void* writer(void* arg)
{
    for(;;) {
        produce_data();

        bthread_sem_wait(&writers_count_sem);
        writers_count += 1;
        if (writers_count == 1) {
            /* Take writers' pass */
            bthread_sem_wait(&writers_pass_sem);
        }
        bthread_sem_post(&writers_count_sem);

        bthread_sem_wait(&db_sem);
        write_data();
        bthread_sem_post(&db_sem);

        bthread_sem_wait(&writers_count_sem);
        writers_count -= 1;
        if (writers_count == 0) {
            /* Give back writers' pass */
            bthread_sem_post(&writers_pass_sem);
        }
        bthread_sem_post(&writers_count_sem);

        bthread_sleep(1000);
    }
}

int main(int argc, char *argv[])
{
    bthread_sem_init(&db_sem, 0, 1);
    bthread_sem_init(&readers_count_sem, 0, 1);
    bthread_sem_init(&writers_count_sem, 0, 1);
    bthread_sem_init(&writers_pass_sem, 0, 1);
    bthread_sem_init(&readers_pass_sem, 0, 1);

    volatile bthread_t readers[5];
      volatile bthread_t writers[5];

          int i;
          for (i=0; i<5; i++) {
              bthread_create(&readers[i], NULL, reader, NULL);
              bthread_create(&writers[i], NULL, writer, NULL);
          }

          for (i=0; i<5; i++) {
              bthread_join(readers[i], NULL);
              bthread_join(writers[i], NULL);
          }

          bthread_sem_destroy(&writers_count_sem);
          bthread_sem_destroy(&writers_pass_sem);
          bthread_sem_destroy(&readers_count_sem);
          bthread_sem_destroy(&db_sem);
          bthread_sem_destroy(&readers_pass_sem);

          return 0;
      }
```

8.7 Readers and writers (fair solution)

```
#include <stdio.h>
#include <stdlib.h>
#include "bthread.h"
#include "tsemaphore.h"

bthread_sem_t arrival_order;
bthread_sem_t readers_count_sem;
bthread_sem_t db_sem;
int readers_count = 0;
int writers_count = 0;

void read_data()
{
    bthread_printf("Reading data\n");
}

void process_data()
{
    bthread_printf("Processing data\n");
}

void* reader(void* arg)
{
    for(;;) {
        bthread_sem_wait(&arrival_order);
        bthread_sem_wait(&readers_count_sem);
        readers_count += 1;
        if (readers_count == 1) {
            bthread_sem_wait(&db_sem);
        }
        bthread_printf("There are %d readers\n", readers_count);
        bthread_sem_post(&readers_count_sem);
```

```c
        bthread_sem_post(&arrival_order);

        read_data();
        bthread_sem_wait(&readers_count_sem);
        readers_count -= 1;
        bthread_printf("There are %d readers\n", readers_count);
        if (readers_count == 0) {
            bthread_sem_post(&db_sem);
        }
        bthread_sem_post(&readers_count_sem);
        process_data();
    }
}

void produce_data()
{
    bthread_printf("Produce data\n");
}

void write_data()
{
    bthread_printf("Writing data\n");
    bthread_sleep(2000);
}

void* writer(void* arg)
{
    for(;;) {
        produce_data();

        bthread_sem_wait(&arrival_order);
        bthread_sem_wait(&db_sem);
        bthread_sem_post(&arrival_order);
        write_data();
        bthread_sem_post(&db_sem);

        bthread_sleep(1000);
    }
}

int main(int argc, char *argv[])
{
    bthread_sem_init(&db_sem, 0, 1);
    bthread_sem_init(&readers_count_sem, 0, 1);
    bthread_sem_init(&arrival_order, 0, 1);

    volatile bthread_t readers[5];
    volatile bthread_t writers[5];

    int i;
    for (i=0; i<5; i++) {
        bthread_create(&readers[i], NULL, reader, NULL);
        bthread_create(&writers[i], NULL, writer, NULL);
    }

    for (i=0; i<5; i++) {
        bthread_join(readers[i], NULL);
        bthread_join(writers[i], NULL);
    }

    bthread_sem_destroy(&arrival_order);
    bthread_sem_destroy(&readers_count_sem);
    bthread_sem_destroy(&db_sem);

    return 0;
}


```