Assignment 1: A Cooperative, User-Level Thread Package

In this assignment, you implement a library that defines a user-level threads package. Using your library, a program can create (user-level) threads, destroy them, and run them concurrently on a single processor. This handout provides additional information on the assignment, including valuable background information and implementation details. You should read it carefully and in full.

Contents

1 Logistics

Before starting to program, we highly recommend reading the entirety of this handout. This section describes how to acquire the starter files, compile code, and submit the assignment.

1.1 Setting up your repository

Your assignment is downloaded from, and submitted to, MarkUs using `git`. Login to MarkUs and navigate to CSC369. From there, navigate to 'a1', where you can see (among other things) the URL of the repository. Before cloning the repository, click 'Add Starter Files to Repository'. Now you can clone the repository.

1.2 Working in your repository

Once you have cloned your repository you should find the starter files in the 'a1' subdirectory. Start by reading the `README.md` in that subdirectory, which includes instructions on how to compile everything. The majority of your work should be done in `a1/src/csc369_thread.c`. In addition, you can write unit tests in `a1/tests/check_thread.c`. Finally, you can also write your own example applications. One example is given to you in `a1/examples/handout_example.c`, which is referred to in this handout.

You should, under no circumstances, change `a1/src/csc369_thread.h` or `a1/src/CMakeLists.txt`. It is also unlikely that you will need to change other `CMakeLists.txt`, except perhaps to add more examples or comment out an `add_subdirectory`. An understanding of CMake is not required for this assignment, but you do need to know how to use the command line to build your project. Again, please refer to `a1/README.md` for more information.

1.3 Submitting changes to your repository

You can find the files you have modified by running the `git status` command. You can commit modified files (e.g., `csc369_thread.c`) to your local repository as follows:

```
git add csc369_thread.c
git commit -m "My useful commit message"
```

We suggest committing your changes frequently by rerunning the commands above (with different meaningful messages to the commit command), so that you can go back to see the changes you have made over time. Once you have tested your code, and committed it locally (check that by running git status), you can `git push` it back to MarkUs. You should also do this frequently. We collect and grade the last version pushed to MarkUs before the due date, up to the maximum late penalty. Please see the Syllabus and MarkUs for more details on the late penalty.

2 Background

Threads and processes are key abstractions for enabling concurrency in operating systems. Threads provide the illusion that different parts of a program are executing concurrently. In the de facto standard model of multithreaded execution, threads share the code, the heap, and the runtime system. However, each thread has a separate stack and, naturally, a separate set of CPU registers.

2.1 Kernel- versus user-level threads

You construct user-level threads by implementing a set of functions (see Section 3) that a program can call directly to provide the illusion of concurrency. In contrast, modern operating systems provide kernel threads, and a user program invokes the corresponding kernel thread functions via system calls. Both types of threads use the same core techniques for providing the concurrency abstraction; you would build kernel threads in essentially the same way you build user threads in this assignment. [1] However, there are a few differences between kernel- and user-level threads:

- **Multiprocessing.** User-level threads provide the illusion of concurrency, but on machines with multiple processors kernel-level threads can provide actual concurrency. With user-level threads, the OS schedules the user process on *one CPU*, and the user-level threads package multiplexes the kernel thread associated with the process between one or more user-level threads. With kernel-level threads, the OS is aware of the different (kernel) threads, and it can simultaneously schedule these threads from the same process on different processors.

  A key simplifying assumption for this assignment is that we allow programs to multiplex some number (e.g., m) of user-level threads on one kernel thread. This means that **at most one user-level thread is running at a time** and that your runtime system has complete control over the interleaving of user-level threads with each other.[2]

- **Asynchronous I/O.** When a user-level thread makes a system call that blocks (e.g., reading a file from disk), the OS scheduler moves the process to the Blocked state and will not schedule it until the I/O has completed. Therefore, even if there are other user-level threads within that process, they also have to wait. In contrast, when a kernel thread blocks for a system call, the OS scheduler may choose another kernel thread from the same process to run. Thus, some kernel threads may be running while others are waiting for I/O.

- **Timer interrupts.** The OS scheduler uses timer interrupts to preempt a running kernel thread and switch the CPU to a different runnable kernel thread. Similar to blocking system calls, this stops the execution of all user-level threads in the process until the kernel thread is scheduled to run again. However, to switch between user-level threads that are multiplexed on a single kernel thread, we cannot rely on timer interrupts (those are delivered to the OS, and not to our thread library runtime). Instead, you implement a cooperative threading model, where the currently running user-level thread **must explicitly yield** the processor to another user-level thread by calling a function.

[1]Kernel processes are built using these techniques, too.

[2]More sophisticated systems implement m on n, where m user-level threads are multiplexed across n kernel threads.

2.2 Thread Context

Each thread has its own state (e.g., the thread's program counter, local variables, stack, etc). A thread context is a subset of this state that must be saved/restored from the processor when switching threads. [3] A user-level thread library needs to store the thread context in a per-thread data structure (this structure is sometimes called the "thread control block").

Fortunately, an application can retrieve its current context and store it in a memory location. An application can also set its current context to a predetermined value from a memory location. This can be accomplished through two existing C library calls: `getcontext` and `setcontext`. Study the manual pages (http://linux.die.net/man/2/setcontext) of these two calls.

Notice that `getcontext` saves the current context into a structure of type `struct ucontext`, which is typedef'd as type `ucontext_t`. So, if you allocate a `struct ucontext` and pass a pointer to that memory to a call to `getcontext`, the current registers and other context will be stored to that memory. Later, you can call `setcontext` to copy that state from that memory to the processor, restoring the saved state. The `struct ucontext` is defined in `/usr/include/x86_64-linux-gnu/sys/ucontext.h` on the teach.cs servers. Look at the fields of this struct in detail, especially the `uc_mcontext` and `uc_stack` fields.

[3]Note: To avoid expensive copying, the thread context includes a pointer to the stack, not the entire stack.

3 The Cooperative Threads API

A key simplifying assumption in this assignment is that the threads are cooperative. That is, each thread runs until it explicitly releases the CPU to another thread by yielding or by exiting. [1] The API provides several functions to allow application programs to perform thread management. However, there are a few conventions that application programs must adhere to in order to ensure proper and safe operation. A list of the functions that constitute the user-level threads API are defined in the `csc369_thread.h` file. You implement these functions in `csc369_thread.c`.

3.1 Using the API

A program that uses the API must initialize the library first. After that, it can call other functions from `csc369_thread.h`. Listing 1 shows the `main` function for a simple program that creates, and repeatedly yields to, one thread. The thread that is created will, eventually, run the function `hello_world` with the argument `"Foo"`. But in order for the created thread to run, the main thread must first yield (i.e., `CSC369_ThreadYield`).

```c
int
main()
{
  // Initialize the user-level thread package
  CSC369_ThreadInit();

  fprintf(stdout, "TID(%d): Create a thread\n", CSC369_ThreadId());
  int const foo_tid = CSC369_ThreadCreate(hello_world, "Foo");

  // Wait for thread to finish
  int tid = foo_tid;
  while (tid == foo_tid) {
    fprintf(stdout, "TID(%d): yield\n", CSC369_ThreadId());
    tid = CSC369_ThreadYield();
  }

  return 0; // exit
}
```

Listing 1: An example of a program using the API.

Listing 2 shows a possible implementation of the `hello_world` function. Note that its body alternates between doing some work and yielding to other threads. A thread exits either explicitly or implicitly. It exits explicitly when it calls the `CSC369_ThreadExit` function. It exits implicitly, as is the case in our example, when its function returns. [2]

[1] In contrast, preemptive threading systems allow a scheduler to interrupt a running thread at any time and switch the CPU to running a different thread.

[2] In addition, a thread may call `CSC369_ThreadKill` to force other threads to exit.

```c
void
hello_world(void* arg)
{
  // Do some work
  char const* msg = arg;
  fprintf(stdout, "TID(%d): %s\n", CSC369_ThreadId(), msg);

  CSC369_ThreadYield(); // yield

  // Do some more work
  fprintf(stdout, "TID(%d): Hello, world!\n", CSC369_ThreadId());

  CSC369_ThreadYield(); // yield

  // implicit thread exit
}
```

Listing 2: An example of a function a thread created by the API will execute.

If you were to run the above program, the following output would be generated:

```
TID(0): Create a thread
TID(0): yield
TID(1): Foo
TID(0): yield
TID(1): Hello, world!
TID(0): yield
TID(0): yield
```

3.2 Implementing the API

There are many ways to go about implementing the functions in `csc369_thread.h`. This section briefly reviews each function. But before putting "pen to paper" (i.e., writing a single line of code), think carefully about the state that you will need to track in your library. You can certainly add things later if you forget something. But planning will help when you start implementing functions because you have already defined some structures (like a ready queue) and global (i.e., file scope) data. As you read through the rest of this section, write down notes of potential state that you would need to keep track of (and how you might keep track of it).

3.2.1 CSC369_ThreadInit

You can use this function to perform any initialization that is needed by your threading system. Here, you should also hand-craft the first user thread in the system. To do so, you should configure your thread state data structures so that the (kernel) thread that is running when your program begins (before any other threads are created) will appear as the first user thread in the system (with tid = 0). You will not need to allocate a stack for this thread, because it runs on the (user) stack allocated for this kernel thread by the OS.

3.2.2 CSC369_ThreadId

This function returns the thread identifier of the currently running thread. The return value should lie between 0 and `CSC369_MAX_THREADS - 1`. See Section 3.3 below.

3.2.3 CSC369_ThreadYield and CSC369_ThreadYieldTo

These functions suspend the caller and activate the 'next' thread given. For `CSC369_ThreadYield`, the next thread is the thread at the head of the ready queue to ensure fairness. This policy is called FIFO (first-in, first-out), since the thread that first entered the ready queue (among the threads that are currently in the ready queue) is scheduled first. For `CSC369_ThreadYieldTo`, the identifier of the desired next thread is given as an argument.

In both functions, the caller is put on the ready queue and can be resumed later in a similar fashion. A reasonable policy is to add the caller to the tail of the ready queue to ensure fairness (so all other threads are run before this thread is scheduled again). The functions return the identifier of the thread that took control as a result of the function call. Note that the caller does not get to see the result until it gets its turn to run (later). If the functions fail, then the caller continues execution immediately.

3.2.4 CSC369_ThreadCreate

This function creates a thread whose starting point is the function f. The second argument, arg, is a pointer that will be passed to the function fn when the thread starts executing. The created thread is put on a ready queue but does not start execution yet. The caller of the create function continues to execute after create returns. If successful, the create function returns a thread identifier of type Tid.

3.2.5 CSC369_ThreadExit

This function ensures that the current thread does not run after this call, i.e., **this function should never return**. If there are other threads in the system, one of them should be run. If there are no other threads (i.e., this is the last thread), then the program should exit. Note that a thread that is created later should be able to reuse this thread's identifier, but only after this thread has been destroyed.

3.2.6 CSC369_ThreadKill

This function kills another thread with the given thread identifier. The victim can be the identifier of any available thread. The killed thread should not run any further and the calling thread continues to execute. Upon success, this function returns the identifier of the thread that was killed.

3.3 Solution Requirements

- The first thread in the system (before the first call to `CSC369_ThreadCreate`) should have a thread identifier of 0.

- Your threads system should support the creation of a maximum of `CSC369_MAX_THREADS` concurrent threads by a program (including the initial main thread). Thus, the maximum value of the thread identifier should be `CSC369_MAX_THREADS - 1` (since thread identifiers start from 0). Note that when a thread exits, its thread identifier can be reused by another thread created later.

- Your library **must** maintain a "thread control block" (a thread structure) for each thread that is running in the system. This is similar to the process control block that an operating system implements to support process management. In addition, your library must maintain a queue of the threads that are ready to run, so that when the current thread yields, the next thread in the ready queue can be run. Your library allows running a fixed number of threads, so if it is helpful, you could allocate these structures statically (e.g., as a global array).

- Each thread should have a stack of at least `CSC369_THREAD_STACK_SIZE` bytes. Your implementation **must not** statically allocate all stacks at initialization time (e.g., using a global data structure). Instead, you **must** dynamically allocate a stack (e.g., using `malloc`) whenever a new thread is created (and free one each time a thread is destroyed.)

- Your library must use `getcontext` and `setcontext` to save and restore thread context state (see Implementation Details below). But it **may not** use `makecontext`, `swapcontext` or any other existing C library code to manipulate a thread's context; you need to write the code to do that yourself.

- Your code **must not** make calls to any existing thread libraries (e.g., Linux pthreads), or borrow code from these libraries for this assignment.

- Do not use any code from other students, or any code available on the Internet, or from artificial intelligence. When in doubt, please ask us.

3.4 Hints and Advice

This assignment does not require writing a large number of lines of code. It does require you to think carefully about the code you write. Before you dive into writing code, it will pay to spend time planning and understanding the code you are going to write. If you think the problem through from beginning to end, this assignment will not be too hard. If you try to hack your way out of trouble, you will spend many frustrating nights in the lab.

As a start, here are some questions you should answer before you write code:

- What fields are needed in your thread control block? Perhaps the most important is the thread state (e.g., running, etc.). Think about all the states that you need to support.

- `getcontext` 'returns' twice. When it is called directly, it initializes the context structure that is passed as an argument, and then execution continues after the `getcontext` call. Then, when `setcontext` is called later, execution returns to the instruction following the `getcontext` call, which appears to be a second 'return', since the code continues running from the instruction following `getcontext`. For this assignment, you use this behavior: once when you create a context, and again when you switch to that context. What action should you take in each case? How can you tell which case you are in?

- Most threads are created with `CSC369_ThreadCreate`, but the initial thread is there before your library is invoked. Nonetheless, the original thread must be able to yield to let other threads run, and other threads must be able to yield and let the original thread run. How is this going to work?

- A hard bug to find would be an overflow or underflow of the stack you allocate. How might such a bug manifest itself? What defensive programming strategies can you use to detect stack overflow in a more controlled manner as the system runs?

- Note that when the initial thread in a C process returns, it calls the exit system call, which causes the OS to destroy the process, even if you have other user level threads in the process that want to run. How will you ensure that the program exits only when the last thread in your system exits?

- It is dangerous to use memory once it has been freed. In particular, you should not free the stack of the currently running thread in `CSC369_ThreadExit` while it is still running. So how will you make sure that the thread stack is eventually deallocated? How will you make sure that another thread that is created in between does not start using this stack (and then you inadvertently deallocate it)? You should convince yourself that your program would work even if you used a debugging malloc library that overwrites a block with dummy data when that block is free'd.

- What are the similarities and differences between a thread yield and a thread exit? Think carefully. It may be useful to encapsulate all that is similar in a common function, which may help reduce bugs make your code simpler.

- Use a debugger. As an exercise, put a breakpoint at the instruction after you copy the current thread's state using `getcontext`. You can print the current values of the registers (in `gdb`, type `info registers`).

  You can also print the values stored in your thread struct and the thread context. For example, say `current` is a pointer to the thread structure associated with the currently running thread, and `context` is a field in this structure that stores the thread context. Then, in `gdb`, you can use `p/x current->context` to print the context stored by a call to `getcontext`. You may find this particularly useful in making sure that the state you 'restore' when you run a newly-created thread for the first time makes sense.

4 Implementation Details

This section outlines details of the implementation that you should know before starting the assignment.

4.1 The Stub Function

When you create a new thread, you want it to run a function. A thread exits implicitly when it returns from its function [1] To implement a similar implicit thread exit, rather than having your thread begin by running the function directly (i.e., the one provided as an argument to `CSC369_ThreadCreate`), you should start the thread in a "stub" function. The stub function calls the desired function of the thread [2]

In other words, `CSC369_ThreadCreate` should initialize a thread so that it starts in a stub function like the one in Listing 3. When the thread runs for the first time, it will execute `MyThreadStub`, which will call `f`. If the `f` function returns, it will return to the stub function which will call `CSC369_ThreadExit` to terminate the thread.

```c
void
MyThreadStub(void (*f)(void *), void *arg)
{
        f(arg);
        CSC369_ThreadExit();
}
```

Listing 3: An example of a stub function.

4.2 Thread Contexts

You will use `getcontext` and `setcontext` in two ways. First, to suspend a currently running thread (to run another one), you will use `getcontext` to save its state and later use `setcontext` to restore its state. Second, to create a new thread, you will use `getcontext` to create a valid context, but you will leave the current thread running; you (the current thread, actually) will then change a few registers in this valid context to initialize it as a new thread, and put this new thread into the ready queue; at some point, the new thread will be chosen by the scheduler, and it will run when `setcontext` is called on this new thread's context.

When creating a thread, you cannot *just* make a copy of the current thread's context (using `getcontext`). You also need to make a few changes to initialize the new thread:

[1] Much like the main program thread is destroyed by the OS when it returns from its main function in C, even when the main function doesn't invoke the exit system call.

[2] Much like main is actually called from the crt0 stub function in UNIX.

- You need to change the saved program counter register in the context to point to a stub function, which should be the first function the thread runs.
- You need to change the saved argument registers, described below, in the context to hold the arguments that are to be passed to the stub function.
- You need to allocate a new per-thread stack using malloc.
- You need to change the saved stack pointer register in the context to point to the top of the new stack. (Warning: in x86-64, stacks grow down!)

In the real world, you would take advantage of an existing library function, `makecontext`, to make these changes to the copy of the current thread's context. The advantage of using this function is that it abstracts away the details of how a context is saved in memory, which simplifies things and helps portability. The disadvantage is that it abstracts away the details of how a context is saved in memory, which might leave you unclear about exactly what's going on. In the spirit of "there is no magic", for this assignment you should not use `makecontext` or `swapcontext`. Instead, you must manipulate the fields in the saved `ucontext_t` directly.

The context structure contains many data fields, but you only need to deal with four of them when creating new threads: the stack pointer, the program counter, and two argument registers. Other than that, you don't need to worry about the fields within the context variable, as long as you do not tamper with them. Also, it is a good idea to use variables that have been initialized through a `getcontext` call in order to not have bizarre behavior.

Figure 4.1 shows what registers and memory looks looks like when any given function is executing. [3] Notice that while a procedure executes, it can allocate stack space by moving the stack pointer down (stack grows downwards). However, it can find local variables, parameters, return addresses, and the old frame pointer (old `%rbp`) by indexing relative to the frame pointer (`%rbp`) register because its value does not change during the lifetime of a function call.

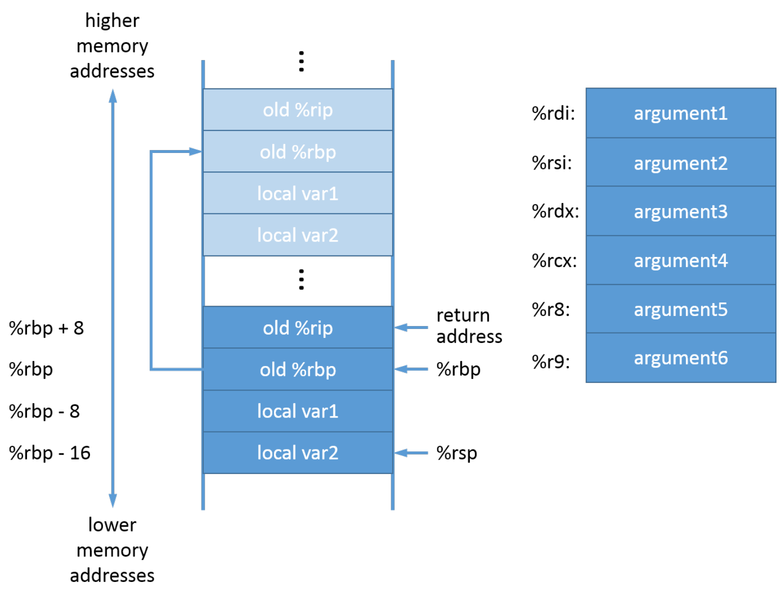

Figure 4.1: A view of memory and some registers when a function is executing

When a function needs to make a function call, it copies the arguments of the "callee" function (the function to be called) to the registers shown on the right in the x86-64 architure. For example, the

[3]Based on the Posix C calling conventions, see: `http://en.wikipedia.org/wiki/X86_calling_conventions#System_V_AMD64_ABI`.

`%rdi` register will contain the first argument, the `%rsi` register will contain the second argument, etc. Then the caller saves the current instruction pointer (`%rip`) into the stack (shown as "return address" in the figure), and changes the instruction pointer to the callee function. At this point the stack pointer (`%rsp`) points to the return address (shown in the figure). Note that the stack pointer points to the last pushed value in the stack.

The callee function then starts executing. It first pushes the the frame pointer value of the caller function (shown as old `%rbp`) into the stack, and then sets the frame pointer (`%rbp`) to the current stack pointer (`%rbp = %rsp`), so that it points to the old frame pointer value. Then the callee function decrements the stack pointer (shown as `%rsp`), and uses the space between the frame pointer and the stack pointer for its local variables, for saving or spilling other registers, etc. As an example, these three steps are performed by the first three instructions (push, mov and sub) in our main function (from Section 3.1), shown on lines 11-13 in Listing 4. The callee locates its variables, parameters, and the return address, by using addresses relative to the fixed frame pointer (`%rbp`).

To return to the caller, a procedure simply copies the frame pointer (`%rbp`) to the stack pointer (`%rsp = %rbp`), effectively releasing the current frame. Then it pops the top stack item into `%rbp` to restore the `%rbp` of the caller function, and then uses the ret instruction to pop the old instruction pointer off the stack into the instruction register (`%rip`), returning control to the caller function. These steps are performed by the last two instructions (`leaveq`, `retq`) in Listing 4.

One complication with the Posix C x86-64 calling convention is that it requires the frame pointer `%rbp` to aligned to 16 bytes. This byte alignment means that the value of `%rbp`, or the stack location (stack address) to which `%rbp` points to, must be a multiple of 16. **Otherwise, system libraries may crash.** When you are creating a new thread that will execute the stub function, you will need to setup the stack so that this calling convention is followed. In particular, you will need to think about the byte alignment for the stack pointer and the frame pointer when control is transferred to `MyThreadStub`.

```
(gdb) b main
Breakpoint 1 at 0x12a5: file handout_example.c, line 24.
(gdb) r
Starting program: handout_example

Breakpoint 1, main () at handout_example.c:24
24      {
(gdb) disassemble main
Dump of assembler code for function main:
=> 0x00005555555552a5 <+0>:     endbr64
   0x00005555555552a9 <+4>:     push   %rbp
   0x00005555555552aa <+5>:     mov    %rsp,%rbp
   0x00005555555552ad <+8>:     sub    $0x10,%rsp
   0x00005555555552b1 <+12>:    callq  0x5555555558ad <CSC369_ThreadInit>
   0x00005555555552b6 <+17>:    callq  0x55555555592b <CSC369_ThreadId>
   0x00005555555552bb <+22>:    mov    %eax,%edx
   0x00005555555552bd <+24>:    mov    0x2d5c(%rip),%rax        # 0x555555558020 <stdout@@GLIBC_2.2.5>
   0x00005555555552c4 <+31>:    lea    0xd5e(%rip),%rsi        # 0x555555556029
   0x00005555555552cb <+38>:    mov    %rax,%rdi
   0x00005555555552ce <+41>:    mov    $0x0,%eax
   0x00005555555552d3 <+46>:    callq  0x5555555550f0 <fprintf@plt>
   0x00005555555552d8 <+51>:    lea    0xd64(%rip),%rsi        # 0x555555556043
   0x00005555555552df <+58>:    lea    -0xbd(%rip),%rdi        # 0x555555555229 <hello_world>
   0x00005555555552e6 <+65>:    callq  0x55555555593b <CSC369_ThreadCreate>
   0x00005555555552eb <+70>:    mov    %eax,-0x4(%rbp)
   0x00005555555552ee <+73>:    mov    -0x4(%rbp),%eax
   0x00005555555552f1 <+76>:    mov    %eax,-0x8(%rbp)
   0x00005555555552f4 <+79>:    jmp    0x555555555325 <main+128>
   0x00005555555552f6 <+81>:    callq  0x55555555592b <CSC369_ThreadId>
   0x00005555555552fb <+86>:    mov    %eax,%edx
   0x00005555555552fd <+88>:    mov    0x2d1c(%rip),%rax        # 0x555555558020 <stdout@@GLIBC_2.2.5>
   0x0000555555555304 <+95>:    lea    0xd3c(%rip),%rsi        # 0x555555556047
   0x000055555555530b <+102>:   mov    %rax,%rdi
   0x000055555555530e <+105>:   mov    $0x0,%eax
   0x0000555555555313 <+110>:   callq  0x5555555550f0 <fprintf@plt>
   0x0000555555555318 <+115>:   mov    $0x0,%eax
   0x000055555555531d <+120>:   callq  0x555555555aeb <CSC369_ThreadYield>
   0x0000555555555322 <+125>:   mov    %eax,-0x8(%rbp)
   0x0000555555555325 <+128>:   mov    -0x8(%rbp),%eax
   0x0000555555555328 <+131>:   cmp    -0x4(%rbp),%eax
   0x000055555555532b <+134>:   je     0x5555555552f6 <main+81>
   0x000055555555532d <+136>:   mov    $0x0,%eax
   0x0000555555555332 <+141>:   leaveq
   0x0000555555555333 <+142>:   retq
```

Listing 4: Disassembly of our earlier example from Section 3.1