Program 1
Last updated: 13 January 2026
This assignment **may not** be done in groups

A Lightweight Process (LWP) Library

This project requires you to implement support for lightweight processes (otherwise known as threads) under 32-bit Linux using the GNU C Compiler (gcc). A lightweight process is an independent thread of control—sequence of executed instructions—executing in the same address space as other lightweight processes. In this assignment you will implement a simple threading library that is non-preemptive (i.e. threads voluntarily yield control of the CPU), single-threaded (i.e. only one thread runs at a time) and executes in use-space (i.e. you will not get any multi-threadings support from the kernel).

The assignment was adopted from versions by Drs. Bellardo and Nico.

Supporting LWPs

Your library must provide seven functions, described briefly in Table 1, and in more detail below.

`new_lwp(function,argument,stacksize)`	Create a new LWP
`lwp_getpid()`	Return PID of the calling LWP
`lwp_exit()`	Terminate the calling LWP
`lwp_yield()`	Yield the CPU to another LWP
`lwp_stop()`	Stop the LWP system
`lwp_start()`	Start the LWP system
`lwp_set_scheduler(scheduler)`	Set a new scheduling function

Table 1: The functions necessary to support threads

Things to know about 32-bit x86

Everything in the rest of this document is intended to provide information needed to implement a lightweight processing library for a 32-bit Intel x86 CPU compiled with gcc. As of Spring 2017, we haven't had a 32-bit machine in the CSSE department. However, you can compile programs in 32-bit mode giving the `-m32` flag to gcc. **This will be necessary in order for the macros**

and calling conventions to work on a 64-bit machine. Further, running in a virtualized environment or the Windows Subsystem for Linux will NOT WORK. It is a very strong suggestion that you do your development and testing on the CSL machines.

What defines a thread

Before we build a threading library, we need to consider what defines a thread. Threads exist in the same memory space as each other (a process), so they can share their code and data segments, but each thread needs its own CPU state (registers) and stack to hold local data, function parameters, and return addresses. They are independent threads of execution.

Thread State

Every thread will have a state of execution, as defined by registers on the CPU. The x86 CPU running in protected mode doing only integer arithmetic has eight registers of interest, shown in Table 2. Since the C programming language has no way of naming registers, macros are provided to allow you to access these registers (described below). You will use these macros to both save and restore the state of a running thread. We call this the thread's *context*.

eax	General Purpose A
ebx	General Purpose B
ecx	General Purpose C
edx	General Purpose D
esi	Source Index
edi	Destination Index
ebp	Base Pointer
esp	Stack Pointer

Table 2: Integer registers of the 32-bit x86 CPU

Stack structure: The gcc calling convention

In addition to its current execution state (as defined by the CPU registers), each thread needs to maintain where it is in the control flow. In other words, what function is it in, what are the arguments and local variables of that function, and where does it go when the function returns. To do this, we use a calling convention and area of process memory image called the stack.

The steps of the 32 bit x86 C calling convention (also described here) are as follows (illustrated in Figure 1a–f):

a. Before a function is called: the *caller* pushes arguments onto the stack in reverse order, then calls the *callee*

b. After the call: the `call` instruction has pushed the return address onto the stack.

c. Before the function body: the *callee* function then executes the following two instructions to set up its frame:

```
pushl %ebp
movl %esp,%ebp
```

Then, it adjusts the stack pointer to leave room for any local variables it allocates.

d. Before the return: while the function executes it may adjust the stack pointer up and down, but before returning, it executes a `ret` instruction. This instruction is equivalent to:

```
movl %ebp,%esp
popl %ebp
```

The effect is to “rewind” the stack back to its state right after the call.

e. After the return, the return address has been popped off the stack, leaving it looking just like it did before the call.

f. After the cleanup, the caller pops off the parameters and the stack is just like it was before.

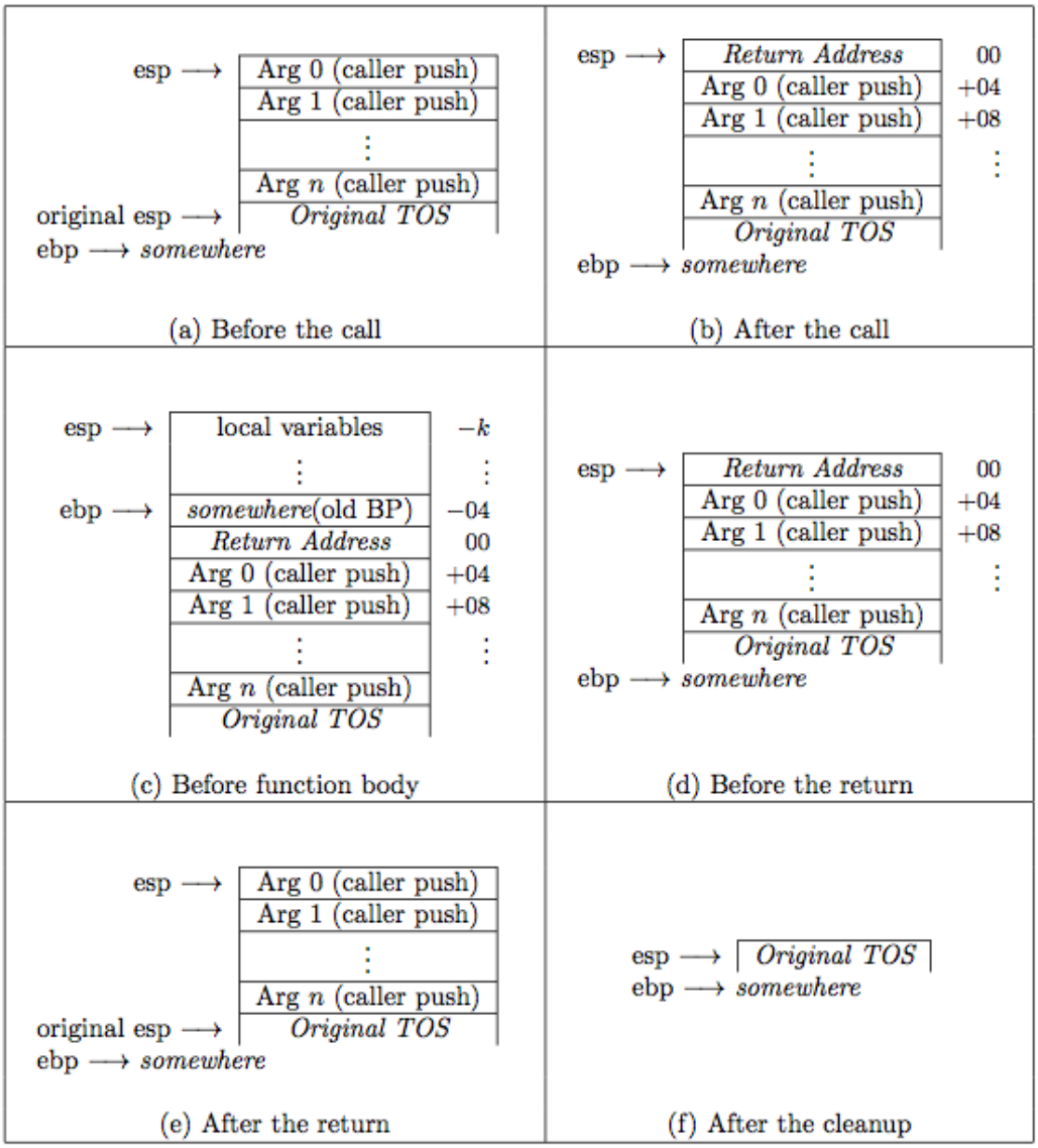

Figure 1: Stack development (Remember that the real stack is upside-down)

The lwp.h macros

`lwp.h` contains four macros that insert assembly instructions into your functions to save or restore a thread's context information. These macros should be very helpful to you in completing this assignment. At no point should it be necessary to write any assembly code yourself.

`SAVE_STATE()`	Push all registers but `%esp` onto the stack in the order shown in Figure 2.
`RESTORE_STATE()`	Pop all registers but `%esp` onto the CPU
`SetSP(var)`	Set `%esp` to the value of `var`
`GetSP(var)`	Set `var` to the value of `%esp`

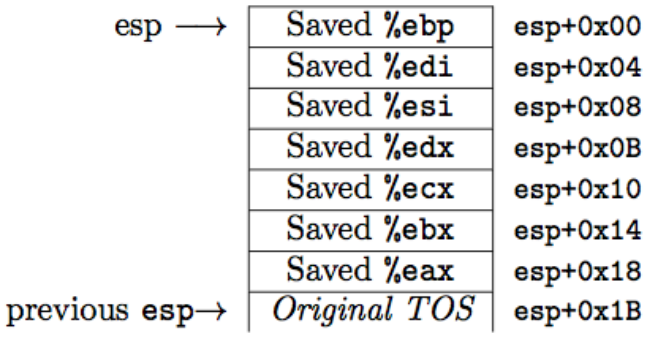

Figure 2: The stack after SAVE_STATE()

.

These macros should be used for saving the CPU state of a running thread when it yields or a call `lwp_stop` is made, such that the thread can continue where it left off in its execution when it is scheduled again.

How to get started

There are basically two “hard parts” to this assignment. The first is setting up a thread’s stack so that it can run. We use a bit of trickery to make this happen:

1. Allocate memory to be used as a stack for the thread being created.
2. Build a stack frame on the stack so when `RESTORE_STATE()` is called, it will properly return to the thread’s function with the stack and registers arranged as it the processor and thread will expect.
    - Remember that the base pointer (ebp) you restore will be copied to the stack pointer (esp) as part of the leave instruction before the function returns.
3. When `lwp_start()` is called:
    (a) save the “real” context (of the main thread) with `SAVE_STATE()`.
    (b) save the “real” stack pointer somewhere where you can find it again.
    (c) schedule one of the lightweight processes to run and switch to its stack.
    (d) load the thread’s context with `RESTORE_STATE()` and you should be off and running.

LWP functions

Your threadings library will implement seven functions. The semantics of these functions are defined in Table 3.

`new_lwp(function, argument, stacksize)`	Called by the user program. Creates a new lightweight process (thread) whose functionality is defined by the given `function` with the given `argument`. The new LWP's stack will be `stacksize` words. The LWP's process table entry will include: `pid` — A unique integer process id `stack` — A pointer to the lowest address of the memory region for this thread's stack `stacksize` — The size of this thread's stack in **words** `sp` — This thread's current stack pointer (top of stack) `new_lwp()` returns the LWP process id of the newly allocated thread, or -1 if more than `LWP_PROC_LIMIT` threads already exist.
`lwp_getpid()`	Called by a thread. Returns the pid of the calling LWP. The return value of `lwp_getpid()` is undefined if not called by an LWP.
`lwp_yield()`	Called by a thread. Yields control to another thread. Which thread depends on the scheduler. Saves the current thread's context (on its stack), schedules the next thread, restores that thread's context, and returns.
`lwp_exit()`	Called by a thread. Terminates the current LWP, removes it from the process table, and moves all the other's up in the table. If no threads remain, it should restore the current stack pointer and return to that context.
`lwp_start()`	Called by the user program. Starts (or resumes) the LWP system. Saves the original context and stack pointer (for `lwp_stop` or `lwp_exit` to use later), schedules an LWP, and starts it running. Returns immediately if there are no LWPs.
`lwp_stop()`	Called by a thread. Stops the LWP system, restores the original stack pointer and returns to that context (wherever `lwp_start()` was called from). `lwp_stop()` does not destroy any existing contexts, and thread processing will be restarted by a call to `lwp_start()`.
`lwp_set_scheduler(`	Called by the user program. Causes the LWP library to use the

`scheduler);`	function `scheduler` to choose the next LWP to run. `(*scheduler)()` is of type `schedfun` and must return an integer in the range `0`...`lwp_procs-1`. If `scheduler` is `NULL`, or never been set, the library should do round robin scheduling.

Table 3: LWP functions

Tricks and Tools

- Remember, the stack grows toward smaller addresses
- A segmentation fault may mean:
  - a stack overflow
  - stack corruption
  - all the other usual causes
- You will get a lot of seg faults in the development of this library. In fact, you will seg faulty somewhere until each and every function has been correctly implemented.
- A working 32-bit and 64-bit version of the library have been provided for you. You can use them to compare your library's functionality.
- For playing with the `snakes` or `hungry` example programs, you will need to link to the `ncurses` library (which may or may not be installed on the CSL machines). If a 32-bit version of that library doesn't exist, don't worry about it, and just use `numbersmain` as your working example.
- Use unix[1-4].csc.calpoly.edu or your own 32-bit x86 compatible machine or virtual machine (not an ARM-based machine, like an Apple M1). Your final submission will be tested on unix2.csc.calpoly.edu.
- If you want to find out what code your compiler is really creating, use the `gcc -S` switch to dump the assembly output.
  ```
  % gcc -S foo.c
  ```
  will produce `foo.s` containing all the assembly. Get comfortable reading this assembly. It will help your development.
- Using precompiled libraries:

  To use a precompiled library file, `libname.a`, you can do one of two things. First, you can simply include it on the link line like any other object file:
  ```
  % gcc -o prog prog.o thing.o libname.a
  ```
  Second, you can use C's library finding mechanism. The -L option gives a directory in which to look for libraries and the -lname flag tells it to include the archive file libname.a:
  ```
  % gcc -o prog prog.o thing.o -L. -lname
  ```
- Building a library:

  To build a library (historically called an *archive*, hence the .a extension), the program to do so is `ar`. The "r" flag means "replace" to insert new files into the archive:
  ```
  % ar r libstuff.a obj1.o obj2.o ... objn.o
  ```
- Use the debugger GDB – if you don't know it, learn it! Some commands that may be

particularly useful are:

- `info registers` – Prints the current register contents
- `x/32 $sp` – Displays the first 32 4-byte entries on the stack
- `disassemble` – Disassembles the current program

Supplied Code

There are several pieces of supplied code along with this assignment. An compressed tar archive of all of them, including a sample file, and a pre-compiled LWP library can be copied from this location on the CSC Unix systems:

`unix*.csc.calpoly.edu:/home/znjp/www/CPE453_LWP_FILES.tgz`

File	Description
`lwp.h`	Header file for the lwp.c you will be writing.
`liblwp-i386.a`	Precompiled 32-bit library of the LWP functions (for testing against). Compare your program's output with this.
`liblwp-x86_64.a`	Precompiled 64-bit library of the LWP functions (for testing against). It is included just for fun.
`numbersmain.c`	A simple demo program that uses threads to print indented numbers. This is the program you should use for testing. It exercises all functionality EXCEPT calling `lwp_getpid` and restarting the threads after a call to `lwp_stop`. You can modify it so that it does both.
`snakesmain.c` and `hungrymain.c`	More advanced demo programs, requiring the `ncurses` library (see above)
`Makefile`	A sample Makefile for building the above. NOTE: it builds executables for the host architecture (based on `uname`). It will need to be modified to always build for 32-bit.

One more time for the back: When linking with `libsnakes.a` it is also necessary to link with the `ncurses` library using `-lncurses` on the link line. Ncurses is a library that supports text terminal manipulation. The CSL machines likely won't have a 32-bit version of `ncurses` installed.

Deliverables

Submit a **tar.gz** file with all your source code (no binaries). It must include the following:

1. An `lwp.c` source file, which defines all of the required LWP functions.
2. A copy of the `lwp.h` header file
3. A makefile (called `Makefile`) that will build a 32-bit version of `liblwp.a` on a CSL UNIX machine from your source when invoked with no target or with the appropriate target (`liblwp.a`). The makefile must also remove all binary and object files when invoked with the "clean" target. You may refer to the provided makefile if you need a starting point.
4. A README file that contains:
    - Your name.
    - Your answer to this short question: "How is this LWP library different from a real thread management library? Name two ways in which this LWP library API or functionality could be improved."
    - Any special instructions.
    - Any other thing you want me to know while I am grading it.
    - The README file should be plain text, i.e, not a Word document, and should be named "`README`", all capitals with no extension.

In addition, you must schedule a brief interview with me, where you should be prepared to answer questions about your assignment. In particular, how threads begin running and are scheduled.

Using tar/gzip

Your submission must be `tar`'d and `gzip`'d archive. Only one file should be submitted, and that is your archive file containing all files required by Deliverables. As per convention, your submission should have either the `tar.gz` or `.tgz` extension. An example of how to use tar and gzip follows.

Let's say you have three files you want to submit for this assignment. Then use the command:

```
# tar -zcvf lwp.tgz lwp.c lwp.h README Makefile
```