Lab 5 CSC 453 Winter 2020 Can be done as a group of three students maximum

Program: Support for Lightweight Processes (liblwp.so)

This assignment requires you to implement support for lightweight processes (threads) under 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. Here you will implement a non-preemptive user-level thread package.
This comes down to writing nine functions, described briefly , and in more detail below.

lwp create(function,argument,stacksize) create a new LWP
lwp gettid() return thread ID of the calling LWP
lwp exit() terminates the calling LWP
lwp yield() yield the CPU to another LWP
lwp start() start the LWP system
lwp stop() stop the LWP system
lwp set scheduler(scheduler) install a new scheduling function
lwp get scheduler(void) find out what the current scheduler is
tid2thread(tid) map a thread ID to a context

The Big Picture

Most of the magic will be in lwp create(). The job of lwp create() is to set up a thread's context so that when it is selected by the scheduler to run and one of lwp yield(), lwp start(), lwp exit() returns[1] to it, it will start executing where you want it to. With a few minor differences, you'll find that lwp yield(), lwp start(), lwp stop(), and lwp exit() all more or less do the same thing: save one context, pick a thread to run, and load that thread's context.

Things to know

Everything in the rest of this document is intended to provide information needed to implement a lightweight processing package for a 64-bit Intel x86 64 CPU compiling with gcc.

Context: What defines a thread
Before we build a thread support library, we need to consider what defines a thread. Threads exist in the same memory as each other, so they can share their code and data segments, but each thread needs its own registers and stack to hold local data, function parameters, and return addresses.
Registers The x86 64 CPU (doing only integer arithmetic[2]) has sixteen registers of interest, shown in below:
rax General Purpose A r8 General Purpose 8
rbx General Purpose B r9 General Purpose 9
rcx General Purpose C r10 General Purpose 10
rdx General Purpose D r11 General Purpose 11
rsi Source Index r12 General Purpose 12
rdi Destination Index r13 General Purpose 13
rbp Base Pointer r14 General Purpose 14
rsp Stack Pointer r15 General Purpose 15

Since C has no way of naming registers, I have provided some useful tools below that will allow you to access these registers. First there are two macros, Get SP() and Set SP(), which will allow you to get or set the %rsp register directly. Second, is an assembly language file, magic64.S[3] which contains a function void swap rfiles(rfile *old, rfile *new). This does two things:
1. if old != NULL it saves the current values of all 16 registers to the struct registers pointed to by old.
2. if new != NULL it loads the 16 register values contained in the struct registers pointed to
by new into the registers. For convenience, there are also two macros, load context(c) and save context(c) that do half of a swap should you find that desirable.
To assemble magic64.S, use gcc:
gcc -o magic64.o -c magic64.S

Floating Point State
As we said above, in addition to the registers, swap rfiles() also preserves the state of the x87 Floating Point Unit(FPU). This is stored in the last element of the struct rfile, the struct fxsave called fxsave. This structure holds all the FPU state. Important: when you initialize your thread's register file, you will have to initialize this structure to the predefined value FPU INIT like so:
newthread->state.fxsave=FPU INIT;

Stack structure: The gcc calling convention
The extra registers available to the x86 64 allow it to pass some parameters in registers. This makes the overall calling convention a little more complicated, but, in practice, it will be easier for your program since you won't be passing enough parameters to push you out of the registers onto the stack.
The steps of the convention are as follows (illustrated in Figures 1a–f):
a. Before the call Caller places the first six integer arguments into registers %rdi, %rsi, %rdx,%rcx, %r8, and %r9. If there are more, they are pushed onto the stack in reverse order. This is shown in the figure, but you won't encounter more in this assignment.
b. After the call The call instruction has pushed the return address onto the stack.
c. Before the function body If the function has more parameters and local variables than will fit into the registers it will execute the following two instructions to set up its frame:
pushq %rbp
movq %rsp,%rbp
Then, it adjusts the stack pointer to leave room for any locals it may need.
d. Before the return Before returning, the function needs to clean up after itself. To do this, before returning it executes a leave instruction. This instruction is equivalent to:
movq %rbp,%rsp
popq %rbp
The effect is to rewind the stack back to its state right after the call.
e. After the return After the return, the Return address has been popped off the stack, leaving it looking just like it did before the call.
Remember, the ret instruction, while called "return", really means "pop the top of the stack into the program counter."
f. After the cleanup Finally, the caller pops off any parameters on the stack and leaves the stack is just like it was before.

LWP system architecture
Everything you need is defined in lwp.h, fp.h and magic64.S, included in Figures 2 and 3.At the heart of lwp.h is the definition of a struct threadinfo t which defines a thread's context. This contains:

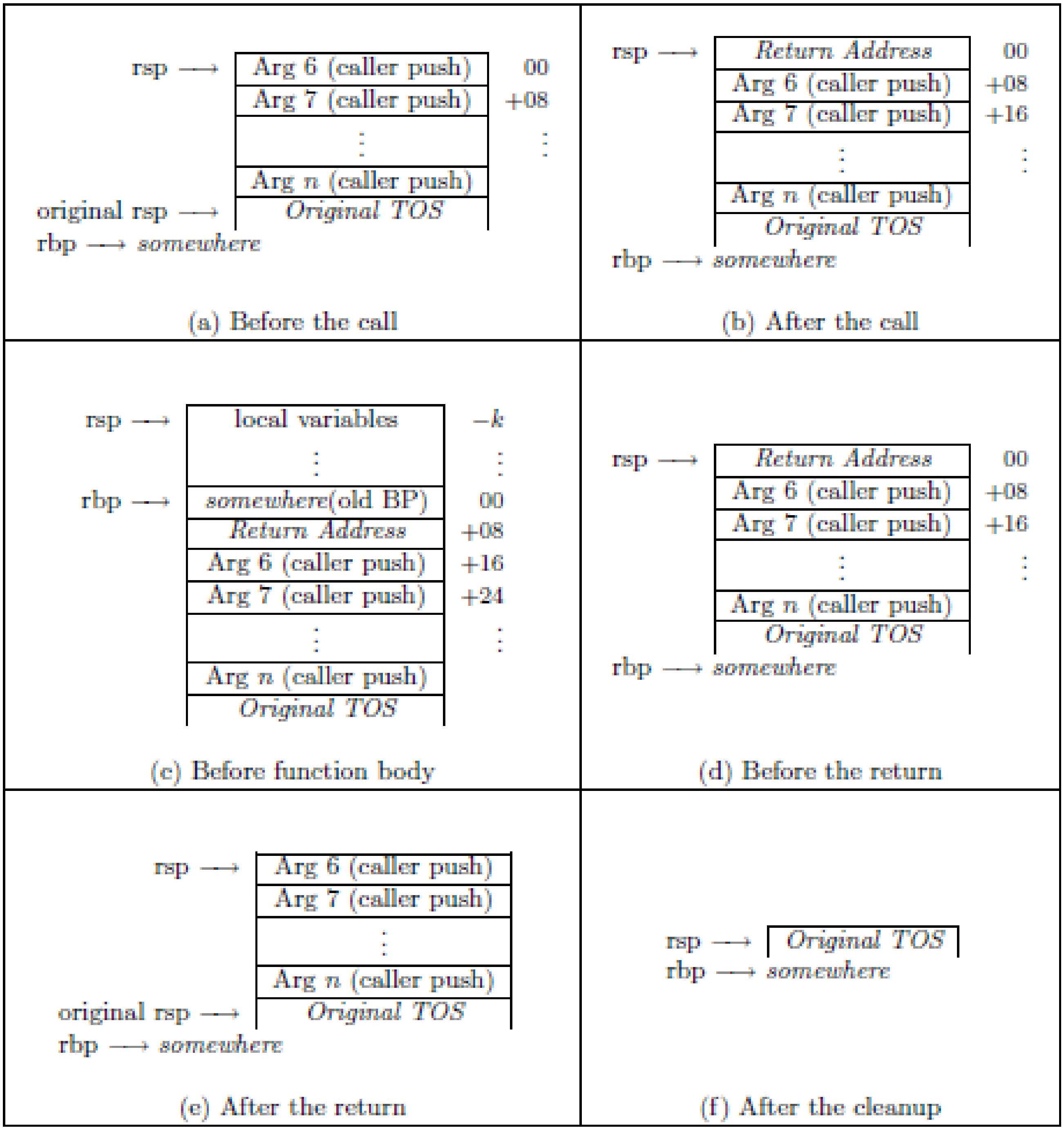

Figure 1: Stack development (Remember that the real stack is upside-down)

```c
#ifndef LWPH
#define LWPH
#include <sys/types.h>

#ifndef TRUE
#define TRUE 1
#endif
#ifndef FALSE
#define FALSE 0
#endif


#if defined(__x86_64)
#include "fp.h"
typedef struct __attribute__ ((aligned(16))) __attribute__ ((packed))
registers {
  unsigned long rax;          /* the sixteen architecturally-visible regs. */
  unsigned long rbx;
  unsigned long rcx;
  unsigned long rdx;
  unsigned long rsi;
  unsigned long rdi;
  unsigned long rbp;
  unsigned long rsp;
  unsigned long r8;
  unsigned long r9;
  unsigned long r10;
  unsigned long r11;
  unsigned long r12;
  unsigned long r13;
  unsigned long r14;
  unsigned long r15;
  struct fxsave fxsave;   /* space to save floating point state */
} rfile;
#else
  #error "This only works on x86_64 for now"
#endif

typedef unsigned long tid_t;
#define NO_THREAD 0             /* an always invalid thread id */

typedef struct threadinfo_st *thread;
typedef struct threadinfo_st {
  tid_t         tid;            /* lightweight process id  */
  unsigned long *stack;         /* Base of allocated stack */
  size_t        stacksize;      /* Size of allocated stack */
  rfile         state;          /* saved registers         */
  thread        lib_one;        /* Two pointers reserved   */
  thread        lib_two;        /* for use by the library  */
  thread        sched_one;      /* Two more for            */
  thread        sched_two;      /* schedulers to use       */
} context;

typedef void (*lwpfun)(void *); /* type for lwp function */

/* Tuple that describes a scheduler */
typedef struct scheduler {
  void   (*init)(void);            /* initialize any structures     */
  void   (*shutdown)(void);        /* tear down any structures      */
  void   (*admit)(thread new);     /* add a thread to the pool      */
  void   (*remove)(thread victim); /* remove a thread from the pool */
  thread (*next)();                /* select a thread to schedule   */
} *scheduler;

/* lwp functions */
extern tid_t lwp_create(lwpfun,void *,size_t);
extern void  lwp_exit(void);
extern tid_t lwp_gettid(void);
extern void  lwp_yield(void);
extern void  lwp_start(void);
extern void  lwp_stop(void);
extern void  lwp_set_scheduler(scheduler fun);
extern scheduler lwp_get_scheduler(void);
extern thread tid2thread(tid_t tid);

/* Macros for stack pointer manipulation:
 *
 *  GetSP(var)      Sets the given variable to the current value of the
 *                  stack pointer.
 *  SetSP(var)      Sets the stack pointer to the current value of the
 *                  given variable.
 */
#if defined (__x86_64)        /* X86 only code */
  #define BAILSIGNAL SIGSTKFLT
  #define GetSP(sp)  asm("movq  %%rsp,%0": "=r" (sp) : )
  #define SetSP(sp)  asm("movq  %0,%%rsp":            : "r" (sp)  )
#else /* END x86 only code */
  #error "This stack manipulation code can only be compiled on an x86"
#endif

#if defined (__APPLE__)
  #undef BAILSIGNAL
  #define BAILSIGNAL SIGABRT
#endif

/* prototypes for asm functions */
#define load_context(c) (swap_rfiles(NULL,c))
#define save_context(c) (swap_rfiles(c,NULL))
void swap_rfiles(rfile *, rfile *to);

#endif
```

Figure 2: Definitions and prototypes for LWP: `lwp.h`

The thread's thread ID. This must be a unique integer that stays the same for the lifetime of the thread. It's what a thread may use to identify itself. (NO THREAD is defined to be 0 and is always invalid.) You may assume that there will never be more than $2^{64} - 2$ threads, so a counter is just fine.

• A pointer to the base of the thread's allocated stack space so that it can later be free()d.

• A struct registers that contains a copy of all the thread's stored registers.

• Four pointers:

– lib one and lib two are reserved for the use of the library internally. (E.g., to maintain a global linked list of all valid threads for implementing tid2thread().)

– sched one and sched two are reserved for use by schedulers.

Together, these hold all the state we need for each thread.

Scheduling

The lwp scheduler is a structure that holds pointers to five functions. These are:

void init(void) This is to be called before any threads are admitted to the scheduler. It's to allow the scheduler to set up. This one is allowed, to be NULL, so don't call it if it is.

void shutdown(void) This is to be called when the lwp library is done with a scheduler to allow it to clean up. This, too, is allowed, to be NULL, so don't call it if it is.

void admit(thread new) Add the passed context to the scheduler's scheduling pool.

void remove(thread victim) Remove the passed context from the scheduler's scheduling pool.

thread next() Return the next thread to be run or NULL if there isn't one.

Changing schedulers will involve initializing the new one, pulling out all the threads from the old one (using next() and remove()) and admitting them to the new one (with admit()), then shutting down the old scheduler.

A note on function pointers:

Remember, the name of a function is its address, so you can pass a pointer to a function just by using its name. For example, my round robin scheduler is defined like so:

static struct scheduler rr publish = {NULL, NULL, rr admit, rr remove, rr next};
scheduler RoundRobin = &rr publish;
Calling a function pointer is just a matter of dereferencing it and applying it to an argument.
E.g.:
thread nxt;
nxt = RoundRobin->next()

How to get started
1. Allocate a stack for each LWP.
2. Build a stack frame on the stack so that when that context is loaded it will properly return to the lwp's function with the stack and registers arranged as it will expect. This involves making the stack look as if the thread called you and was suspended.
3. When lwp start() is called:
(a) save the "real" context somewhere where lwp stop() can find it,
(b) pick one of the lightweight processes to run (by calling the scheduler).
(c) Load its context with swap rfiles() and you should be off and running.
Remember, what you are trying to do is to build a context so that when lwp yield() selects it, loads its registers, and returns, it starts executing its very first instruction with the stack pointer pointing to a stack that looks like it had just been called.
If the arguments fit into registers, this will simply be:

rsp $\longrightarrow$	*Return Address*	00
	Original TOS	+08

rbp $\longrightarrow$ *somewhere*

But what is this return address? It's supposed to be the place where the thread function should go "back" to after it's done, but it didn't come from anywhere. Use lwp exit(). That way either it calls lwp exit() or it returns there, but one way or the other when it's done, lwp exit() will be called.

Tricks, Tools, and Useful Notes

• a segmentation violation may mean
– a stack overflow.
– stack corruption
– all the other usual causes
• Use the CSL linux machines
• But I really want to use my Mac.
Ok. . . but there are a few things that are different about doing this under OSX:
– OSX requires all stack frames to be 16-byte aligned.
– Dynamic libraries have the suffix .dylib
– The path the loader searches for dynamic libraries is DYLD LIBRARY PATH.
– It is possible to compile multiple architectures of library into a single .dylib file. See lido(1) for details.
– Finally, you'll need to be sure it compiles and runs on Linux, since that's where it'll be graded.

`lwp_create(function,argument,stacksize);`	Creates a new lightweight process which executes the given function with the given argument. The new processes's stack will be `stacksize` words. `lwp_create()` returns the (lightweight) thread id of the new thread ir −1 if the thread cannot be created.
`lwp_gettid(void);`	Returns the tid of the calling LWP or `NO_THREAD` if not called by a LWP.
`lwp_yield(void);`	Yields control to another LWP. Which one depends on the scheduler. Saves the current LWP's context, picks the next one, restores that thread's context, and returns.
`lwp_exit(void);`	Terminates the current LWP and frees its resources. Calls `sched->next()` to get the next thread. If there are no other threads, calls `lwp_stop()`.
`lwp_start(void);`	Starts the LWP system. Saves the original context (for `lwp_stop()` to use later), picks a LWP and starts it running. If there are no LWPs, returns immediately.
`lwp_stop(void);`	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(scheduler);`	Causes the LWP package to use the given `scheduler` to choose the next process to run. Transfers all threads from the old scheduler to the new one in `next()` order. If `scheduler` is `NULL`, or has never been set, the scheduler should do round-robin scheduling.
`lwp_get_scheduler(void);`	Returns the pointer to the current scheduler.
`tid2thread(tid);`	Returns the `thread` corresponding to the given thread ID, or `NULL` if the ID is invalid

Table 3: The LWP functions

If you want to find out what your compiler is really doing, use the gcc -S switch to dump the assembly output.
gcc -S foo.c
will produce foo.s containing all the assembly.

• lwp exit() is probably the second trickiest part of this assignment because you must be careful not to free() a stack that you're still using.
• Remember that stacks start in high memory and grow towards low memory.
• Also remember that pointer arithmetic is done in terms of the size of the thing pointed-to.
• Instructions for building and using shared libraries are included in Asgn1 if you need to review.
• Note that lwp stop() does not necessarily mean stop forever. It should be possible to call lwp stop() then later call lwp start() to restart.
• Finally, remember that there doesn't have to be a next thread. If sched->next() returns NULL, lwp yield(), lwp exit(), and lwp start() should restore the original system context. (as lwp stop() does).

Supplied Code

There are several pieces of supplied code

File Description/Location
lwp.h Header file for lwp.c

fp.h Header file for preserving floating point state
libPLN.a precompiled library of lwp functions (for testing)
libsnakes.a precompiled library of snake functions
magic64.S ASM source for swap rfiles()
snakes.h header file for snake functions
hungrymain.c demo program for hungry snakes
snakemain.c demo program for wandering snakes
numbersmain.c demo program with indented numbers
Note: When linking with libsnakes.a it is also necessary to link with the standard library ncurses using -lncurses on the link line. Ncurses is a library that supports text terminal manipulation.

What to turn in

Submit to polylearn

• your well-documented source file(s).
• Your header file, lwp.h, suitable for inclusion with other programs. This must be compatabile
with the distributed one, but you may extend it. A makefile (called Makefile) that will build liblwp.so from your source when invoked with
no target or with the target "liblwp.so".
• A README file that contains:
– Your name(s), including your login name(s) in parentheses (e.g. "(pnico)").
– 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.

Sample runs

We did these in class. If you want, though, you can use the provided libPLN.a to build your own
samples.