Project 2 | CPE 453
This project **may** be done in groups of 2. One member submits.

Light Weight Processes

This assignment requires you to implement support for lightweight processes (threads) under Linux.
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 assignment originally by Dr. Nico is adopted by versions by Dr. Bellardo and Nico

The Big Picture

Creating threads is to basically create a library that exposes an API for programs to create, control and run threads. This comes down to writing nine functions, described briefly in Table 1, and in more detail below

`lwp_create(function,argument)`	create a new LWP
`lwp_start(void)`	start the LWP system
`lwp_yield(void)`	yield the CPU to another LWP
`lwp_exit(int)`	terminate the calling LWP
`lwp_wait(int *)`	wait for a thread to terminate
`lwp_gettid(void)`	return thread ID of the calling LWP
`tid2thread(tid)`	map a thread ID to a context
`lwp_set_scheduler(scheduler)`	install a new scheduling function
`lwp_get_scheduler(void)`	find out what the current scheduler is

Table 1: The functions necessary to support threads

What you're doing is taking the one real stream of control—the one that calls `main()`, which we will call the original system thread—and sharing it across an arbitrary number of lightweight threads.
Most of the real work will be in `lwp_create()`. `lwp_create()` creates a new thread and sets up its context so that when it is selected by the scheduler to run and `lwp_yield()` uses `swap_rfiles()` to load its context and returns[1] to it, it will start executing at the very first instruction of the thread's body function.

Calling `lwp_yield()` causes a thread to yield control to another thread, and `lwp_exit()` terminates the calling thread and switches to another, if any. The whole system is started off by a call to `lwp_start()` which adds the original system thread to the thread pool, then yields control whichever thread the scheduler should choose.

[1] This is important: none of these thread functions—the ones that are passed to `lwp_create()` to form the program of the new thread—are ever called. They are returned to.

The Library Functions

The semantics of the individual library functions are listed in Table 2 with explanatory notes as necessary below.

`tid_t lwp_create(lwpfun function, void *argument);`	Creates a new lightweight process which executes the given function with the given argument. `lwp_create()` returns the (lightweight) thread id of the new thread or `NO_THREAD` if the thread cannot be created.
`void lwp_start(void);`	Starts the LWP system. Converts the calling thread into a LWP and `lwp_yield()`s to whichever thread the scheduler chooses.
`void 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. If there is no next thread, terminates the program.
`void lwp_exit(int exitval);`	Terminates the current LWP and yields to whichever thread the scheduler chooses. `lwp_exit()` does not return.
`tid_t lwp_wait(int *status);`	Waits for a thread to terminate, deallocates its resources, and reports its termination status if `status` is non-NULL. Returns the tid of the terminated thread or `NO_THREAD`.
`tid_t lwp_gettid(void);`	Returns the tid of the calling LWP or `NO_THREAD` if not called by a LWP.
`thread tid2thread(tid_t tid);`	Returns the `thread` corresponding to the given thread ID, or `NULL` if the ID is invalid
`void lwp_set_scheduler(scheduler sched);`	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` the library should return to round-robin scheduling.
`scheduler lwp_get_scheduler(void);`	Returns the pointer to the current scheduler.

Table 2: The LWP functions

- `lwp_create()`
  Creates a new thread and admits it to the current scheduler. The thread's resources will consist of a context and stack, both initialized so that when the scheduler chooses this thread and its context is loaded via `swap_rfiles()` it will run the given function. This may be called by any thread.

- `lwp_start()`
  Starts the threading system by converting the calling thread—the original system thread—into a LWP by allocating a context for it and admitting it to the scheduler, and yields control to whichever thread the scheduler indicates. It is not necessary to allocate a stack for this thread since it already has one.

- `lwp_yield()`
  Yields control to the next thread as indicated by the scheduler. If there is no next thread, calls `exit(3)` with the termination status of the calling thread (see below).

- `lwp_exit(int status)`
  Terminates the calling thread. Its termination status becomes the low 8 bits of the passed integer. The thread's resources will be deallocated once it is waited for in `lwp_wait()`. Yields control to the next thread using `lwp_yield()`.

- `lwp wait(int *status)`
  Deallocates the resources of a terminated LWP. If no LWPs have terminated and there still exist runnable threads, blocks until one terminates. If status is non-NULL, `*status` is populated with its termination status. Returns the `tid` of the terminated thread or NO_THREAD if it would block forever because there are no more runnable threads that could terminate.
  Be careful not to deallocate the stack of the thread that was the original system thread.

  A little more on `lwp_wait()`
  `lwp_wait()`, as specified so far, introduces some nondeterminism into our system, e.g., if there are multiple terminated threads, which one is returned or if there are multiple threads waiting when `lwp_wait()` is called, which one does it get?
  In a real system we may not care, but for a homework it's really useful if we make the same decisions so we can compare results. So, to that end:

  When `lwp_wait()` is called, if there exist terminated threads, it will return the oldest one without blocking. That is, it will return terminated threads in FIFO order and the oldest will be the head of the queue.

  If there are no terminated threads, the caller of `lwp_wait()` will have to block.
  Deschedule it (with `sched->remove()`) and place it on a queue of waiting threads. When another thread eventually calls `lwp_exit()` associate it with the oldest waiting thread—the pointer exited may be useful for this—remove it from the queue, and reschedule it (with `sched->admit()`) so it can finish its call to `lwp_wait()`.

  The only exception to this blocking behavior is if there are no more threads that could possibly block. In that case `lwp_wait()` just returns NO_THREAD. The way it can tell is by using the scheduler's `qlen()` function (see below). Most likely the calling thread will still be in the scheduler at the time of this check, so you're testing for whether `qlen()` is greater than 1.

Thread body functions

The code to be executed by a thread is contained in function whose address is passed to `lwp_create()`. The thread will execute until it either calls `lwp_exit()` or the function returns with a termination status. This thread function takes a single argument, a pointer to anything, that is also passed to `lwp_create()`.

Termination statuses

A thread's status consists of a flag indicating whether it is running (LWP_LIVE) or terminated (LWP_TERM) and an 8-bit integer that can be passed back via `lwp_wait()`.
A thread's termination value is the low 8 bits either of the argument to `lwp_exit()` or of the return value of the thread function. These are combined into a single integer using the macro MKTERMSTAT() which is what is passed back by `lwp_wait()`.
Macros for dealing with termination statuses are given in Table 3.

`#define LWP_LIVE`	status of a live thread
`#define LWP_TERM`	status of a terminated thread
`#define MKTERMSTAT(a,b)`	combine status and exit code into an int
`#define LWPTERMINATED(s)`	true if the status represents a terminated thread
`#define LWPTERMSTAT(s)`	extracts the exit code from a status

Table 3: Macros for thread exit statuses.

Stacks

Every thread needs a stack, and that stack needs to come from somewhere. So far, a way you know to get memory is `malloc(3)`, which allocates to you a junk of memory in a contiguous heap, meaning that if one stack overflows, it can overflow into neighboring regions. In this section we will look at using `mmap(2)` to create stacks in memory regions that are not connected to each other.
`mmap(2)` is a versatile system call that allows processes to map regions of memory shared with other processes, or to map files directly into their memory spaces bypassing the IO system calls. For our purposes, we're just going to use `mmap(2)` to create a region of memory for each of our threads to use as a stack. If a thread's stack overflows, this will generate a SEGV when it touches the first unmapped page, but it will not corrupt its neighbors.

```
void *mmap(where, size, perms, flags, fd, offset);
```

For our stacks, `where` should be NULL (let `mmap(2)` choose), `fd` should be −1 (some implementations require this), and `offset` should be zero. We should offer read and write permission (but not execute) and we should have flags appropriate to a stack:

```
s = mmap(NULL,howbig,PROT_READ|PROT_WRITE,MAP_PRIVATE|MAP_ANONYMOUS|MAP_STACK,-1,0);
```

`mmap(2)` returns a pointer to the memory region on success or MAP_FAILED on failure.

The remaining question is, how big should these stacks be? First, stacks must be a multiple of the memory page size. This can be determined by using `sysconf(3)` to look up the variable _SC_PAGE_SIZE.

Now, like `pthreads(7`) we will use the stack size resource limit if it exists. To get the value of a resource limit, use `getrlimit(2)`. The limit for stack size is RLIMIT_STACK. `getrlimit(2)` reports both hard and soft resource limits. Use the soft one.
If RLIMIT_STACK does not exist or if its value is RLIM_INFINITY, choose a reasonable stack size. I use 8MB[2]. On a sane system, this resource limit will be a multiple of the page size. But what if it's not? Round up to the nearest multiple of the page size. Now you've got your size. Allocate a stack and get on with it.

When done with a mapping, it can—and should—be unmapped using `munmap(2)`.

Note: The man page talks about `mmap(2)` being able to create regions that automatically grow downward to support stacks. Apparently in current linux kernels this is. . . aspirational. Still, many megabytes of stack should be good enough for our threads.

[2] Yes, this feels rather large, but a 64-bit address space is huge, so why not?

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`[3]. This is the environment found on the Linux desktop machines in the CSL and `unix[1-5].csc.calpoly.edu`.

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[4]) has sixteen registers of interest, shown in Table 4

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

Table 4: Integer registers of the x86_64 CPU

Since C has no way of naming registers, I have provided some useful tools below that will allow you to access these registers. The assembly language file, `magic64.S`[5] 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 and the floating point state to the `struct registers` pointed to by `old`.

2. if `new != NULL` it loads the 16 register values and the floating point state contained in the `struct registers` pointed to by `new` into the registers.

In this assignment it should never be necessary to load or store a context independently. Always do atomic context switches using `swap_rfiles()`. To assemble `magic64.S`, use `gcc`:
`gcc -o magic64.o -c magic64.S`
The whole function can be seen in Figure 3.

---

[3] It should work with other compilers, but I've tested it with `gcc`.
[4] As well as a bunch more for floating point, but we aren't going to talk about those here. `swap_rfiles()` saves them, though.
[5] For what it's worth, if an assembly file ends in "`.S`", the compiler will run it through the C preprocesser. If it's "`.s`", it won't

```
        .text
        .globl swap_rfiles
        .type  swap_rfiles, @function
  swap_rfiles:
        # void swap_rfiles(rfile *old, rfile *new)
        #
        # "old" will be in rdi
        # "new" will be in rsi
        #
        pushq %rbp              # set up a frame pointer
        movq %rsp,%rbp

        # save the old context (if old != NULL)
        cmpq    $0,%rdi
        je load

        movq %rax,   (%rdi)     # store rax into old->rax so we can use it

        # Now store the Floating Point State
        leaq 128(%rdi),%rax     # get the address
        fxsave (%rax)

        movq %rbx,  8(%rdi)     # now the rest of the registers
        movq %rcx, 16(%rdi)     # etc.
        movq %rdx, 24(%rdi)
        movq %rsi, 32(%rdi)
        movq %rdi, 40(%rdi)
        movq %rbp, 48(%rdi)
        movq %rsp, 56(%rdi)
        movq %r8,  64(%rdi)
        movq %r9,  72(%rdi)
        movq %r10, 80(%rdi)
        movq %r11, 88(%rdi)
        movq %r12, 96(%rdi)
        movq %r13,104(%rdi)
        movq %r14,112(%rdi)
        movq %r15,120(%rdi)

        # load the new one (if new != NULL)
load:   cmpq    $0,%rsi
        je done

        # First restore the Floating Point State
        leaq 128(%rsi),%rax     # get the address
        fxrstor (%rax)

        movq    (%rsi),%rax     # retreive rax from new->rax
        movq   8(%rsi),%rbx     # etc.
        movq  16(%rsi),%rcx
        movq  24(%rsi),%rdx
        movq  40(%rsi),%rdi
        movq  48(%rsi),%rbp
        movq  56(%rsi),%rsp
        movq  64(%rsi),%r8
        movq  72(%rsi),%r9
        movq  80(%rsi),%r10
        movq  88(%rsi),%r11
        movq  96(%rsi),%r12
        movq 104(%rsi),%r13
        movq 112(%rsi),%r14
        movq 120(%rsi),%r15
        movq  32(%rsi),%rsi     # must do rsi last, since it's our pointer

done:   leave
        ret
```

Figure 3: magic64.S: Store one register file and load another

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

In order to build a context in `lwp_create()` that will do the right thing when loaded and returned-to, you will need to know the process by which stack frames are built up and torn down.
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.
This section describes the calling convention which will allow you to both understand and construct the stack frames you will need. These figures show normal stack development. What you will be developing will be distinctly abnormal. 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**
Before the body of a function executes it needs to set up its stack frame that will hold any parameters and local variables that will fit into the registers. To do this, it will execute the following two instructions to set up its frame:
```
pushq %rbp
movq %rsp,%rbp
```
Then, it may adjust 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."

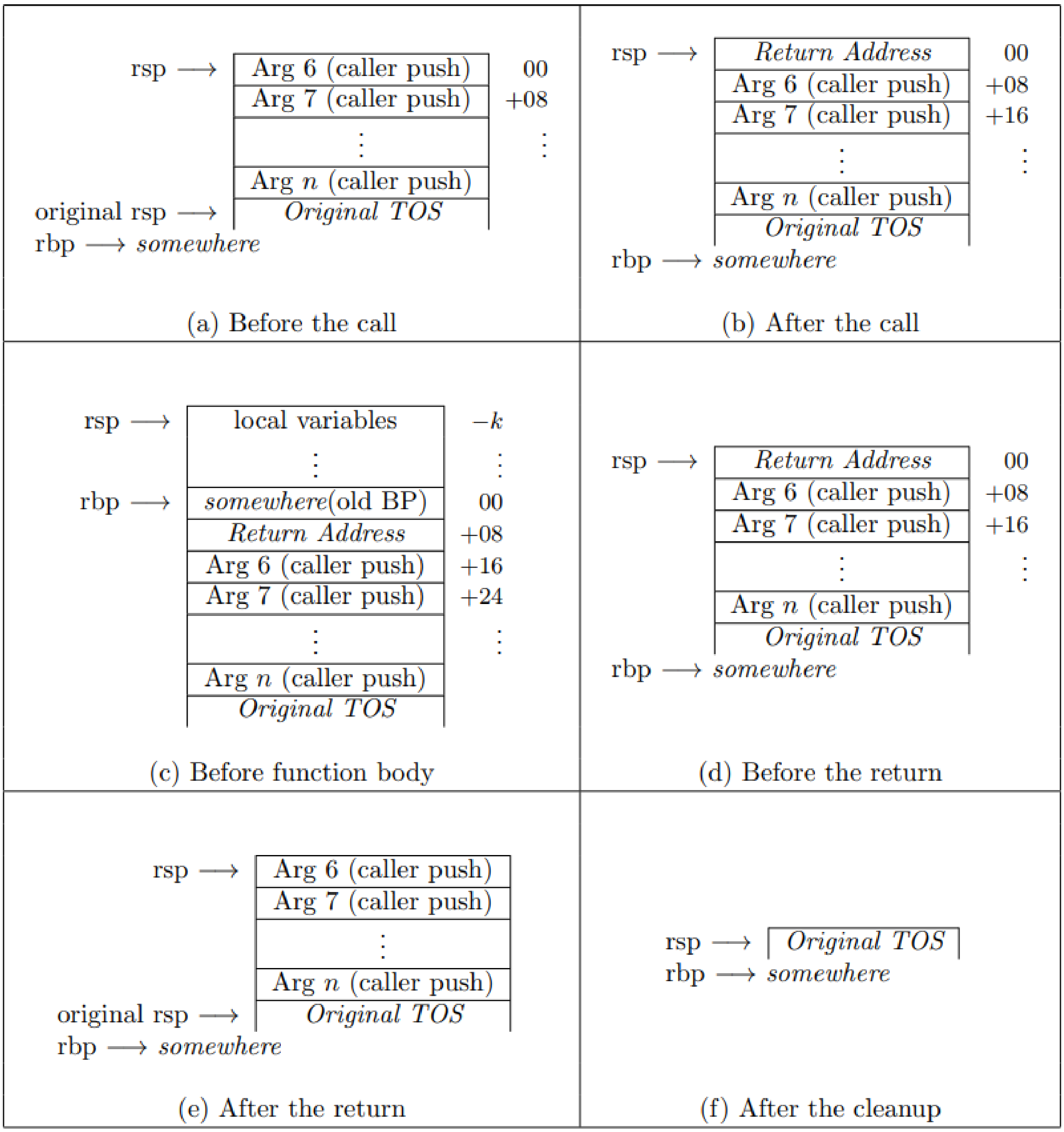
f) **After the cleanup**
Finally, the caller pops off any parameters on the stack and leaves the stack is just like it was before.

Note: Intel's Application Binary Interface specification[6] requires that all stack frames be aligned on a 16 byte boundary[7]. The exact wording is:

[6] See: https://software.intel.com/sites/default/files/article/402129/mpx-linux64-abi.pdf, p 18.
[7] See, that requirement in malloc wasn't just made up to make life hard for you.

The end of the input argument area shall be aligned on a 16 (32 or 64, if __m256 or __m512 is passed on stack) byte boundary.

This means that the address of the bottom (lowest in memory) element of the argument area needs to be evenly divisible by 16, even if there isn't an argument area. That is, the address above the frame's return address must be evenly divisible by 16 (equivalently, the saved base pointer's address must be evenly divisible by 16).
Be aware of this as you build your stacks. If your stack frame is not properly aligned, all you will see is a SEGV.



(a) Before the call (b) After the call (c) Before function body (d) Before the return (e) After the return (f) After the cleanup

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

LWP system architecture

Everything you need is defined in `lwp.h`, `fp.h`, and `magic64.S`, two of which are included in Figures 2 and 3 (for the third, see "Supplied Code" later on).

At the heart of `lwp.h` is the definition of a `struct threadinfo_st` which defines a thread's context. This contains:

- 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—the pointer originally returned by `mmap(2)`, see above—so that it can later be unmapped.

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

- A *status* integer that encodes the current status of a thread (running or terminated) and an exit status if terminated.

- Four pointers:

  `lib_one` and `lib_two` are reserved for the use of the library internally, for any purpose or no purpose at all. (Many people find these useful to maintain a global linked list of all threads for implementing `tid2thread()` or perhaps for keeping track of threads that are waiting.)

  `sched_one` and `sched_two` are reserved for use by schedulers, for any purpose or no purpose at all. Most schedulers need to keep lists of threads, so this makes that convenient.

  Neither the scheduler nor the library may make any assumptions about what the other is doing

These, along with each's stack, hold all the state we need for each thread.

```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         */
  unsigned int  status;         /* exited? exit status?    */
  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       */
  thread        exited;         /* and one for lwp_wait()  */
} context;

typedef int (*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)(void);            /* select a thread to schedule   */
  int    (*qlen)(void);            /* number of ready threads       */
} *scheduler;

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

/* for lwp_wait */
#define TERMOFFSET        8
#define MKTERMSTAT(a,b)   ( (a)<<TERMOFFSET | ((b) & ((1<<TERMOFFSET)-1)) )
#define LWP_TERM          1
#define LWP_LIVE          0
#define LWPTERMINATED(s)  ( (((s)>>TERMOFFSET)&LWP_TERM) == LWP_TERM )
#define LWPTERMSTAT(s)    ( (s) & ((1<<TERMOFFSET)-1) )

/* prototypes for asm functions */
void swap_rfiles(rfile *old, rfile *new);

#endif
```

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

Scheduling

The lwp library's default scheduling policy is round robin—that is, each thread takes its turn then goes to the back of the line when it yields—but client code can install its own scheduler with `lwp_set_scheduler()`. The lwp `scheduler` type is a pointer to a structure that holds pointers to six 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.

- **`int qlen()`**
  Return the number of runnable threads. This will be useful for `lwp_wait()` in determining if waiting makes sense

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:

```
struct scheduler rr_publish = {NULL, NULL, rr admit, rr remove, rr next, rr qlen};
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. Write the default round robin scheduler. This consists almost entirely of keeping a list, and then you will have a scheduler, and it feels good to have started.

2. Then, in `lwp_create()`:

   (a) Allocate a stack and a context for each LWP.

   (b) Initialize the stack frame and context so that when that context is loaded in `swap_rfiles()`, 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 `swap_rfiles()` and was suspended.**

   How to do this? Figure out where you want to end up, then work backwards through the endgame of `swap_rfiles()` to figure out what you need it to look like when it's loaded.

   You know that the end of `swap_rfiles()` (and every function) is:
   ```
   leave
   ret
   ```

   And that `leave` really means:
   ```
   movq %rbp, %rsp ; copy base pointer to stack pointer
   popq %rbp ; pop the stack into the base pointer
   ```

   and `ret` means pop the instruction pointer, so the whole thing becomes:
   ```
   movq %rbp, %rsp ; copy base pointer to stack pointer
   popq %rbp ; pop the stack into the base pointer
   popq %rip ; pop the stack into the instruction pointer
   ```

   Consider that what you're doing, really, is creating a stack frame for `swap_rfiles()` to tear down—in lieu of the one it created on the way in, on a different stack—and creating the caller's half of `lwpfun's` stack frame since nobody actually calls it. (c) admit() the new thread to the scheduler.

   (c) `admit()` the new thread to the scheduler.

3. When `lwp_start()` is called:

   (a) Transform the calling thread—the original system thread—into a LWP. Do this by creating a context for it and `admit()`ing it to the scheduler, but don't allocate a stack for it. Use the stack it already has. Make sure not to deallocate this later (leave it NULL in the context or flag it some other way).

   (b) `lwp_yield()` to whichever thread the scheduler picks

   (c) The idea here is that once the original system thread calls `lwp_start()` it is transformed into just another thread (other than that you shouldn't free its stack). From here on out, the system continues until there are no more runnable threads.

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 the thread's 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 (and they will in this case), this will simply be:

rsp ⟶	*Return Address*	00
	Original TOS	+08
rbp ⟶ *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. You could 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.

Note: What is this "original TOS"? This is the alleged past of this thread. Of course, it doesn't have a past, so it doesn't exist. This thread came from nowhere.

About that thread "going back"

The termination of the thread function poses an interesting challenge: If it calls `lwp_exit()` with an exit status, all is well and it's clear how to proceed.
But what if it doesn't? If the thread function returns, the value that it returns is supposed to become its exit status. If we simply return to `lwp_exit()` as suggested above, the return value is in the location where return values are to be found (%rax) rather than in the register where `lwp_exit()` will look for its argument (%rdi). No amount of stack trickery will get us what we want here. The easiest way to deal with this is to remember that you are a programmer:
Instead of invoking the thread function directly, wrap it in a little function like the one in Figure 4 that calls the thread function with its argument, then calls `lwp_exit()` with the result. (This is, in fact, completely analogous to how `main()` is called. The process really begins with `_start()`.)

```
static void lwp_wrap(lwpfun fun, void *arg) {
  /* Call the given lwpfunction with the given argument.
   * Calls lwp_exit() with its return value
   */
  int rval;
  rval=fun(arg);
  lwp_exit(rval);
}
```

Figure 4: A useful wrapper for the thread function.

Tricks, Tools, and Useful Notes

Just some things to consider while designing and building your library:

- a segmentation violation may mean
  – a stack overflow
  – stack corruption
  – an attempt to access a stack frame that is not properly aligned
  – all the other usual causes

- Use the CSL linux machines (or your own).

- 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.

- Remember that stacks start in high memory and grow towards low memory. You can find the high end of your stack region through the magic of arithmetic.

- Also remember that pointer arithmetic is done in terms of the size of the thing pointed-to.

- I defined the stack member of the context structure to be an `unsigned long *` to make it easy to treat the stack as an array of unsigned longs and index it accordingly.

- Despite the fact that it is possible to load and save contexts independently, don't do it. The compiler feels free—rightly—to move the stack pointer to allocate or deallocate local storage on the stack. If you save your context in one place and load it in another, your thread will go through a time warp and saved data may be corrupted. Use `swap_rfiles` to perform an atomic context switch.

- Finally, remember that there doesn't have to be a next thread. If `sched->next()` returns NULL, `lwp_yield()` will exit as described above.

- 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 an archive, the program to do so is `ar(1)`. The `r` flag means "replace" to insert new files into the archive:
  ```
  % ar r libstuff.a obj1.o obj2.o ...objn.o
  ```

Supplied Code

There are several pieces of supplied code along with this project, all available on the CSL machines in **~pn-cs453/Given/Asgn2**[8]

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.

Assignment

Turn in this assignment on Canvas. Name of the submitted file must be **project2_submission.tar**

What to turn in

1. Your source files (.c, .h, etc.)

   Your header file, `lwp.h`, suitable for inclusion with other programs. This must be compatabile with the distributed one, but you may extend it

1. A make file (called `Makefile`) that will build `liblwp.a` on `unix[1-4]` 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. Refer to the example makefile if you need more guidance on this. [a sample Makefile is provided on Canvas.]

2. A README.txt file with:
   a. Your names on the very first line
   b. Special instructions for the program
   c. If your program does not work properly, state why and what is missing
   d. Anything else you want me to consider while grading

[8] Choose a directory and move there. Use `cp -r ~pn-cs453/Given/Asgn2 .` to copy all the files over.

Files shared on Canvas:
`-Makefile`
`-libPLN.a`
`-libsnakes.a`

Sample runs

You can run the demos for the project yourself. The demos are found here:
`~pn-cs453/Given/Asgn2/demos`
Copy those files into your directory of choice.

```
LD_LIBRARY_PATH=../lib64/
export LD_LIBRARY_PATH

make nums
./nums

make snakes
./snakes

make hungry
./hungry
```