

Assignment 2 Solutions

cpe 453 Fall 2024

In the beginning there was data. The data was without form and null, and darkness was upon the face of the console; and the Spirit of IBM was moving over the face of the market. And DEC said, "Let there be registers"; and there were registers. And DEC saw that they carried; and DEC separated the data from the instructions. DEC called the data Stack, and the instructions they called Code. And there was evening and there was morning, one interrupt.
-- Rico Tudor, "The Story of Creation or, The Myth of Urk"

— /usr/games/fortune

Due by 11:59:59pm, Wednesday, October 16th.
This assignment may be done with a partner.

Program: Support for Lightweight Processes (liblwp.so)

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 comes down to writing nine functions, described briefly in Table 1, and in more detail below.

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

Table 1: The functions necessary to support threads

The Big Picture

When 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 returns¹ to it, it will start executing at the very first instruction of the thread's body function.

¹This 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.

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.

The Library Functions

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

<code>tid_t lwp_create(lwpfun function, void *argument);</code>	Creates a new lightweight process which executes the given function with the given argument. <code>lwp_create()</code> returns the (lightweight) thread id of the new thread or <code>NO_THREAD</code> if the thread cannot be created.
<code>void lwp_start(void);</code>	Starts the LWP system. Converts the calling thread into a LWP and <code>lwp_yield()</code> s to whichever thread the scheduler chooses.
<code>void lwp_yield(void);</code>	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.
<code>void lwp_exit(int exitval);</code>	Terminates the current LWP and yields to whichever thread the scheduler chooses. <code>lwp_exit()</code> does not return.
<code>tid_t lwp_wait(int *status);</code>	Waits for a thread to terminate, deallocates its resources, and reports its termination status if <code>status</code> is non-NULL. Returns the tid of the terminated thread or <code>NO_THREAD</code> .
<code>tid_t lwp_gettid(void);</code>	Returns the tid of the calling LWP or <code>NO_THREAD</code> if not called by a LWP.
<code>thread tid2thread(tid_t tid);</code>	Returns the <code>thread</code> corresponding to the given thread ID, or <code>NULL</code> if the ID is invalid
<code>void lwp_set_scheduler(scheduler sched);</code>	Causes the LWP package to use the given <code>scheduler</code> to choose the next process to run. Transfers all threads from the old scheduler to the new one in <code>next()</code> order. If <code>scheduler</code> is <code>NULL</code> the library should return to round-robin scheduling.
<code>scheduler lwp_get_scheduler(void);</code>	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 (p.11). 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.

<code>#define LWP_LIVE</code>	status of a live thread
<code>#define LWP_TERM</code>	status of a terminated thread
<code>#define MKTERMSTAT(a,b)</code>	combine status and exit code into an int
<code>#define LWPTERMINATED(s)</code>	true if the status represents a terminated thread
<code>#define LWPTERMSTAT(s)</code>	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, the only place you know to get memory is `malloc(3)` which allocates to you a hunk 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.

To create an anonymous mapping with `mmap(2)`, first read the man page. The prototype for `mmap(2)` is

```
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 `pthread(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 8MB².

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.

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 arithmetic⁴) has sixteen registers of interest, shown in Table 4.

<code>rax</code>	General Purpose A	<code>r8</code>	General Purpose 8
<code>rbx</code>	General Purpose B	<code>r9</code>	General Purpose 9
<code>rcx</code>	General Purpose C	<code>r10</code>	General Purpose 10
<code>rdx</code>	General Purpose D	<code>r11</code>	General Purpose 11
<code>rsi</code>	Source Index	<code>r12</code>	General Purpose 12
<code>rdi</code>	Destination Index	<code>r13</code>	General Purpose 13
<code>rbp</code>	Base Pointer	<code>r14</code>	General Purpose 14
<code>rsp</code>	Stack Pointer	<code>r15</code>	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:

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

³It should work with other compilers, but I've tested it with `gcc`.

⁴As well as a bunch more for floating point, but we aren't going to talk about those here. `Swap_rfiles()` saves them, though.

⁵For what it's worth, if an assembly file ends in ".S", the compiler will run it through the C preprocessor. If it's ".s", it won't.

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.

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):

- 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.
- After the call** The `call` instruction has pushed the return address onto the stack.
- 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.

- 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 specification⁶ requires that all stack frames be aligned on a 16 byte boundary⁷. The exact wording is:

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.

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.)

⁶See: <https://software.intel.com/sites/default/files/article/402129/mpx-linux64-abi.pdf>, p 18.

⁷See, that requirement in `malloc` wasn’t just made up to make life hard for you.

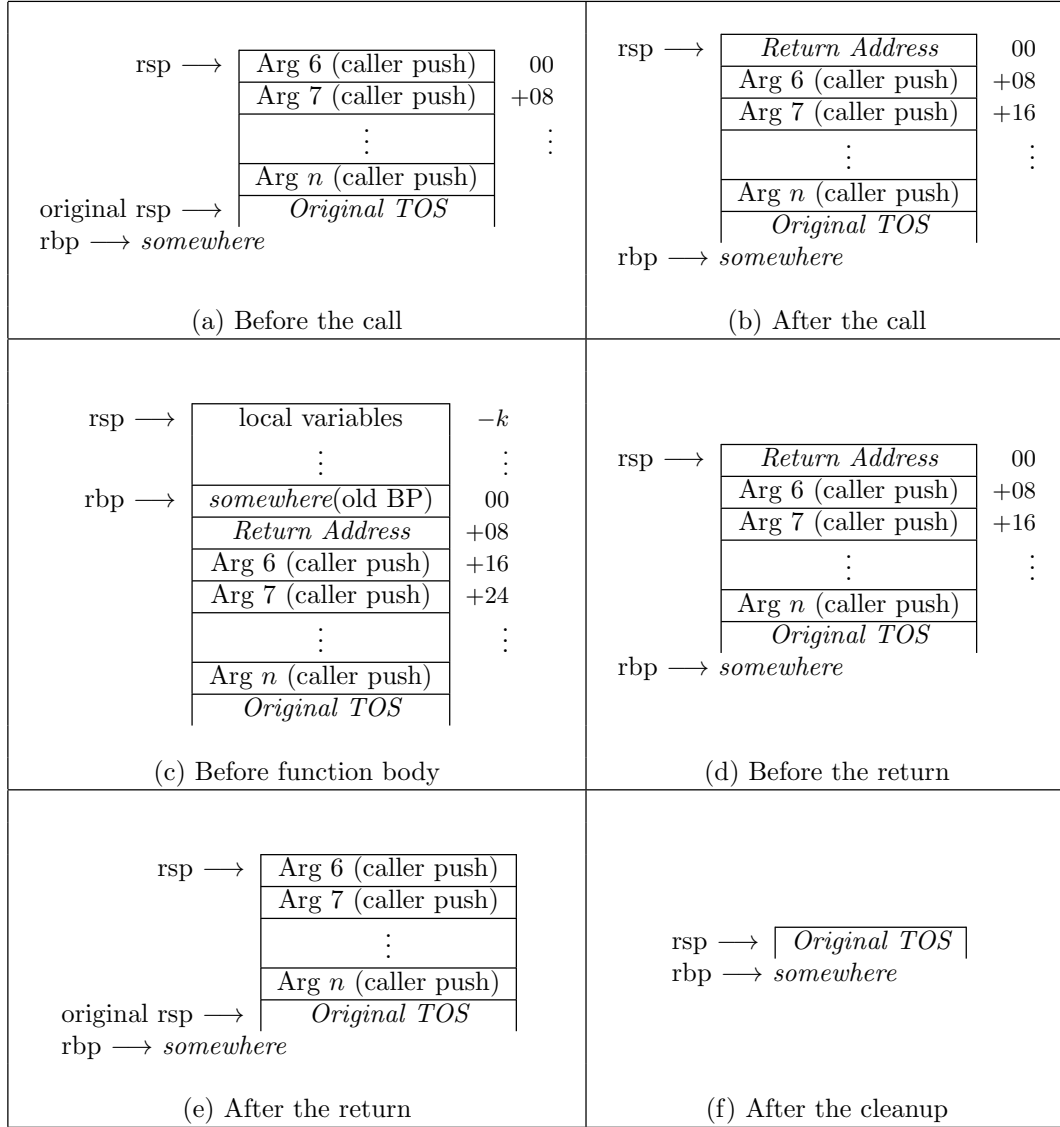


Figure 1: Stack development (Remember that the real stack is upside-down)


```

#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;
    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 fsave fsave; /* 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 *tthread;
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 LWP_TERMINATED(s) (((s)>>TERMOFFSET)&LWP_TERM) == LWP_TERM
#define LWP_TERMSTAT(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

```

.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     # retrieve 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

- `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.

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.

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:

$$\begin{array}{lcl} \text{rsp} \longrightarrow & \boxed{\begin{array}{c} \textit{Return Address} \\ \textit{Original TOS} \end{array}} & \begin{array}{l} 00 \\ +08 \end{array} \\ \text{rbp} \longrightarrow & \textit{somewhere} & \end{array}$$

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: I’m often asked, 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).
- But I really want to use my Mac.

Ok...but there are a few things that are different about doing this under MacOS:

- MacOS 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 `lipo(1)` for details.
- Finally, you’ll need to be sure it compiles and runs on Linux, since that’s where it’ll be graded.

- 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.
- Instructions for building and using shared libraries are included in `Asgn1` if you need to review.
- 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.

Supplied Code

There are several pieces of supplied code along with this assignment, all available on the CSL machines in `~pn-cs453/Given/Asgn2`.

File	Description/Location
<code>lwp.h</code>	Header file for <code>lwp.c</code>
<code>fp.h</code>	Header file for preserving floating point state
<code>libPLN.a</code>	precompiled library of <code>lwp</code> functions (for testing)
<code>libsnares.a</code>	precompiled library of snake functions
<code>magic64.S</code>	ASM source for <code>swap_rfiles()</code>
<code>snares.h</code>	header file for snake functions
<code>hungrymain.c</code>	demo program for hungry snakes
<code>snaresmain.c</code>	demo program for wandering snakes
<code>numbersmain.c</code>	demo program with indented numbers

Note: When linking with `libsnares.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.

Coding Standards and Make

See the pages on coding standards and make on the cpe 453 class web page.

What to turn in

Submit via `handin` to the `asgn2` directory of the `pn-cs453` account:

- your well-documented source file(s).
- Your header file, `lwp.h`, suitable for inclusion with other programs. This must be compatible 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.

Solution:

File	Where
Makefile	p.16
fp.h	p.17
lwp.h	p.19
lwp.c	p.21
magic64.S	p.26
rr.h	p.28
rr.c	p.29
snakes.h	p.32
stacks.h	p.33
stacks.c	p.34
tid.h	p.36
tid.c	p.37
util.h	p.40
util.c	p.41

Makefile

```
UNAME := $(shell uname -s)

ifeq ("$(UNAME)","Darwin")
    CC      = clang
    CFLAGS  = -Wall -g -I.
    SHAREDLIBS = liblwp.dylib
else      # everyone else, for now
    CC      = gcc
    CFLAGS  = -Wall -fPIC -g -I.
    SHAREDLIBS = liblwp.so
endif

AR      = ar r

RANLIB  = ranlib

LIBS    = $(SHAREDLIBS)

OBJS    = lwp.o util.o magic64.o rr.o tid.o stacks.o

SRCS    = lwp.c util.c rr.c tid.c magic.s

HDRS    =

EXTRACLEAN = core

all:    $(LIBS)

allclean: clean
        @rm -f $(EXTRACLEAN)
        @-rmdir ./lib ./lib64

clean:
        rm -f $(OBJS) *~ TAGS

$(LIB): $(OBJS)
        $(AR) $@ $(OBJS)
        ranlib $@

depend:
        @echo Regenerating local dependencies.
        @makedepend -Y $(SRCS) $(HDRS)

tags : $(SRCS) $(HDRS)
        etags $(SRCS) $(HDRS)

test: nums
        ./nums

nums: numbersmain.o liblwp.a
        $(CC) $(CFLAGS) -g -o nums numbersmain.o -L. -llwp

numbersmain.o: numbersmain.c
        $(CC) $(CFLAGS) -c numbersmain.c

liblwp.so: $(OBJS)
        $(CC) $(CFLAGS) -shared -o $@ $(OBJS)

%.o: %.c
        $(CC) $(CFLAGS) -c -o $@ $.c

%.o: %.S
        $(CC) $(CFLAGS) -c -o $@ $<

# DO NOT DELETE

lwp.o: util.h tid.h stacks.h
tid.o: tid.h
```



```

/* This was captured from a live FPU. All of these bits are probably not
 * necessary, but that's a task for another day.
 */

```

```

#ifndef LWP_H
#define LWP_H
#include <sys/types.h>

#ifndef TRUE
#define TRUE 1
#endif
#ifndef FALSE
#define FALSE 0
#endif
10

#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 for now"
#endif

typedef unsigned long tid_t;
#define NO_THREAD 0 /* an always invalid thread id */
40

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;
50

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;
60

/* 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);
70

```

```
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 */
```

80

```
#define TERMOFFSET      8
#define MKTERMSTAT(a,b) ((a)<<TERMOFFSET | ((b) & ((1<<TERMOFFSET)-1)) )
#define LWP_TERM        1
#define LWP_LIVE         0
#define LWP_TERMINATED(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);
```

90

```
#endif
```

```

#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <signal.h>
#include <string.h>
#include <lwp.h>
#include <rr.h>
#include "util.h"
#include "tid.h"
#include "stacks.h"
10

#define ALIGNFACTOR 16

extern void dp();
/* Process context information.
*/

static unsigned long tcount = 0; /* how many threads are there */

static thread lwp_running = NULL; /* the currently running LWP */
static thread morgue      = NULL; /* list of exited threads */
static thread waiting     = NULL; /* list of threads waiting */
20

/* local data */
static scheduler sched=&rr_publish; /* RoundRobin; the default */

/* local forward declarations and useful stuff */
static void cswitch(context *from, context *to);

/* useful for "intuitively" building stacks */
#define push(sp,val) (*(--sp)=(unsigned long)(val))
30

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);
}
40

static unsigned long *new_intel_stack(unsigned long *sp,lwpfun func, void *arg){
/* construct a stack for the INTEL architecture
*/

unsigned long *ebp;

push(sp,0x12345678); /* Return for wrap (unused) for alignment */
push(sp,lwp_wrap); /* push the wrapper's return address */
push(sp,0); /* push a "saved" base pointer */
50

ebp=sp; /* note the location for use later... */

return ebp;
}

static thread newthread(void) {
/* allocate a new context and give it an id. Returns the thread */
thread new;
new = malloc(sizeof(context));
60
if ( new ) {
new->tid=create_tid(new); /* assign a unique thread id */
new->stack = NULL;
new->stacksize = 0;
new->status = MKTERMSTAT(LWP_LIVE,0);
new->lib_one = NULL;
new->lib_two = NULL;
new->sched_one = NULL;
new->sched_two = NULL;
new->exited = NULL;
70
memset(&new->state,0,sizeof(new->state));
}
}

```

```

    return new;
}

tid_t _attribute_ ((weak))
lwp_create(lwpfun func, void *arg){
    /* generate a context for a newly created thread */
    unsigned long *sp;
    thread new;
    tid_t rvalue = 0;

    if ( !(new = newthread()) ) {
        rvalue = NO_THREAD;
    } else if ( !addstack(new) ){
        free(new);
        rvalue = NO_THREAD;
    } else {
        /* Initialize the stack so that lwp_wrap() is returned to
         * and lwp_wrap() calls exit.
         */
        tcount++;
        sp = (unsigned long *)new->state.SP;
        sp = new_intel_stack(sp, func, arg);

        /* create the context for the new LWP */
        new->status = 0;          /* not terminated */

        new->state.rdi=(unsigned long)func;
        new->state.rsi=(unsigned long)arg;
        new->state.rbp=(unsigned long)sp;
        new->state.fxsave=FPU_INIT;

        sched->admit(new);      /* add to the schedulable pool */
        rvalue = new->tid;      /* return the new tid */
    }

    return rvalue;
}

static int running=FALSE;      /* are we running? */
void _attribute_ ((weak))
lwp_start(){
    /* start the LWP system.
     *
     * Turn the original system thread into a lightweight one, admit
     * and start the system.
     */
    thread self;
    if ( running ) {
        fprintf(stderr, "%s: called while threads are active?\n", _FUNCTION_);
    } else {
        self = newthread();
        if ( self ) {
            tcount++;
            sched->admit(self);
            lwp_running = self;
            if ( lwp_running ) { /* and, if not, how'd that happen? */
                running=TRUE;
                lwp_yield();
            }
        }
    }
}

tid_t _attribute_ ((weak))
lwp_gettid() {
    /* return the tid of the currently running lwp. NO_THREAD if
     * threading is inactive
     */
    tid_t rvalue;
    if ( lwp_running )
        rvalue = lwp_running->tid;
}

```

```

else
    rvalue = NO_THREAD;
return rvalue;
}

//-----
150

#define WAIT_NEXT sched_one

thread append_to_waitlist(thread list, thread t) {
    /* Append the given thread to the given waiting list (threaded by
     * WAIT_NEXT)
     */
    thread l;
    t->WAIT_NEXT = NULL;
160
    if ( !list ) {
        list = t;
    } else {
        for(l=list;l->WAIT_NEXT;l=l->WAIT_NEXT)
            /*wheel*/;
        l->WAIT_NEXT = t;
    }
    return list;
}

170
tid_t lwp_wait(int *status) {
    /* wait for a thread to terminate. When it does, release its
     * resources and return its status is status is non-NULL. Returns
     * the tid of the thread that exited. Returns -1 on error (there is
     * no other runnable thread)
     */
    tid_t which;
    thread this, old;
    this = lwp_running;
    if ( morgue ) { /* there's a thread ready to reap */
180
        this->exited = morgue;
        morgue = morgue->WAIT_NEXT; /* remove from to list of exited lwps */
    } else { /* We're going to have to wait */
        if ( sched->qlen() <= 1 )
            return NO_THREAD; /* nothing to wait for */
        sched->remove(this); /* add to waiting list and wait */
        waiting=append_to_waitlist(waiting,this); /* add to list of waiting lwps */
        lwp_yield(); /* we'll be rescheduled when appropriate */
    }
    /* a thread has terminated */
190
    old = this->exited;
    which = old->tid;
    if ( status )
        *status = old->status;
    if ( old->stack )
        removestack(old);
    free(old);
    return which;
}

200
void _attribute_ ((weak))
lwp_exit(int status){
    /* mark the calling thread as terminated and remove from
     * the scheduler. Resources will be deallocated in lwp_wait()
     */
    thread this;

    this = lwp_running;
    this->status = MKTERMSTAT(LWP_TERM,status);
    sched->remove(this);
210

    tcount--;

    /* if there is a thread waiting, make this its target and reschedule it.
     * If not, put it on the list
     */

```

```

if ( waiting ) {
    thread w = waiting;
    waiting=waiting->WAIT_NEXT;
    sched->admit(w);
    w->exited = this;
} else {
    morgue=append_to_waitlist(morgue,this); /* add to list of exited lwps */
}
lwp_yield();          /* we will never be rescheduled */
}

void __attribute__((weak)) lwp_yield(){
    /* perform a yield */
    thread this;

    this = lwp_running;
    lwp_running = sched->next();
    if (!lwp_running) { /* If there's no next, there's no next */
        if ( tcount )
            fprintf(stderr,"WARNING: next() returned NULL while "
                    "threads exist.  tcount=%ld\n",
                    tcount);
        exit(LWPTERMSTAT(this->status));
    }
    cswitch(this, lwp_running);
}

void cswitch(context *from, context *to) {
    /* do a context switch */
    /*
        block_signals();
        fprintf(stderr,"Start %p\n",(void*)lwp_running->tid);
        if ( from )
            save_context(&from->state);
        fprintf(stderr,"Middle %p\n",lwp_running->tid);
    */
    swap_rfiles(&from->state,&to->state);
    /* if ( to )
        load_context(&to->state);
        fprintf(stderr,"After. %p\n",(void*)(lwp_running?(lwp_running->tid):0));
        unblock_signals();
    */
}

scheduler lwp_get_scheduler(void) {
    /* return the current scheduling suite */
    return sched;
}

void lwp_set_scheduler(scheduler fun) {
    /* initialize the new scheduler, transfer all the threads to
       * the new one, and tear down the old one.
    */
    scheduler old = sched;
    thread t;

    /* Don't change a thing if it's the identity assignment.
       * This prevents RR from chasing its tail, e.g.
    */
    if ( fun == sched )
        return;

    if ( fun )
        sched = fun;
    else
        sched = RoundRobin;

    if ( sched->init )
        sched->init();          /* initialize the new one, if needed */

    /* transfer all the threads */

```



```

while ( ( t==old->next() ) ) {
    old->remove(t);
    sched->admit(t);
}

/* shutdown the old one, if needed */
if ( old->shutdown )
    old->shutdown();
}

void ps() {
    /* print the stack */
    unsigned long *esp,**ebp,*obp;
    int i;
    asm("movq  %rsp,%0": "=r" (esp) : );
    asm("movq  %rbp,%0": "=r" (ebp) : );
    obp = *ebp;
    for(i=-10;i<30;i++) {
        if ( esp+i == obp )
            fprintf(stderr,"obp => <%p> %p\n",esp+i,(void*)esp[i]);
        else
            fprintf(stderr,"      <%p> %p\n",esp+i,(void*)esp[i]);
    }
}

void preg(rfile *r) {
#ifdef __x86_64
    fprintf(stderr, "  rax: 0x%01x\n",r->rax);
    fprintf(stderr, "  rbx: 0x%01x\n",r->rbx);
    fprintf(stderr, "  rcx: 0x%01x\n",r->rcx);
    fprintf(stderr, "  rdx: 0x%01x\n",r->rdx);
    fprintf(stderr, "  rsi: 0x%01x\n",r->rsi);
    fprintf(stderr, "  rdi: 0x%01x\n",r->rdi);
    fprintf(stderr, "  rbp: 0x%01x\n",r->rbp);
    fprintf(stderr, "  rsp: 0x%01x\n",r->rsp);
    fprintf(stderr, "   r8: 0x%01x\n",r->r8);
    fprintf(stderr, "   r9: 0x%01x\n",r->r9);
    fprintf(stderr, "  r10: 0x%01x\n",r->r10);
    fprintf(stderr, "  r11: 0x%01x\n",r->r11);
    fprintf(stderr, "  r12: 0x%01x\n",r->r12);
    fprintf(stderr, "  r13: 0x%01x\n",r->r13);
    fprintf(stderr, "  r14: 0x%01x\n",r->r14);
    fprintf(stderr, "  r15: 0x%01x\n",r->r15);
#elif defined(__i386)
    fprintf(stderr, "  eax: 0x%01x\n",r->eax);
    fprintf(stderr, "  ebx: 0x%01x\n",r->ebx);
    fprintf(stderr, "  ecx: 0x%01x\n",r->ecx);
    fprintf(stderr, "  edx: 0x%01x\n",r->edx);
    fprintf(stderr, "  esi: 0x%01x\n",r->esi);
    fprintf(stderr, "  edi: 0x%01x\n",r->edi);
    fprintf(stderr, "  ebp: 0x%01x\n",r->ebp);
    fprintf(stderr, "  esp: 0x%01x\n",r->esp);
#endif
}

```

```

#ifdef __APPLE__
#define FNAME _swap_rfiles
#else
/* everyone else */
#define FNAME swap_rfiles
#endif

.text
.globl FNAME
#ifdef __APPLE__
.type swap_rfiles, @function
#endif
FNAME:
    # 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     # retrieve rax from new->rax
    movq 8(%rsi),%rbx     # etc.
    movq 16(%rsi),%rcx
    movq 24(%rsi),%rdx
    movq 32(%rsi),%rsi
    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

```

magic64.S

```
done:  leave
        ret
```

```
#ifndef RRH
#define RRH

#include <lwp.h>

extern scheduler RoundRobin;
extern struct scheduler rr_publish; /* for static initialization */

#endif
```

```

#include <lwp.h>
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <rr.h>

static thread qhead=NULL;
static int len=0; /* current queue length */
#define tnext sched_one
#define tprev sched_two

static __attribute__((unused)) void pq(void) {
    /* print the scheduler's queue. Useful for debugging */
    thread t;
    printf("Q: ");
    t=qhead;
    if ( t ) {
        do {
            printf("%ld%s",t->tid,((t->tnext!=qhead)?", ":""));
            t=t->tnext;
        } while ( t != qhead );
    }
    putchar('\n');
}

static void rr_admit(thread new) {

    len++; /* adjust length */
    /* add to queue */
    if ( qhead ) {
        new->tnext = qhead;
        new->tprev = qhead->tprev;
        new->tprev->tnext = new;
        qhead->tprev = new;
    } else {
        qhead = new;
        qhead->tnext = new;
        qhead->tprev = new;
    }
    /* printf("admit: ");pq(); */
}

static void rr_remove(thread victim) {
    len--; /* adjust length */
    /* cut out of queue */
    victim->tprev->tnext = victim->tnext;
    victim->tnext->tprev = victim->tprev;

    /* what if it were qhead? */
    if ( victim == qhead ) {
        if ( victim->tnext != victim )
            qhead = victim->tnext; /* preserve who would've been next */
        else
            qhead = NULL;
    }
    /* printf("remove: ");pq(); */
}

static thread rr_next() {
    thread t;

    t = qhead;
    if ( qhead ) {
        qhead = qhead->tnext;
    }

    return t;
}

static int rr_glen(void) {
    /* return number of ready threads */

```

```

    return len;
}

struct scheduler rr_publish = {NULL, NULL, rr_admit, rr_remove, rr_next, rr_qlen};
scheduler RoundRobin = &rr_publish;

/*****/
_attribute__((unused)) void
dpl() {
    thread l;
    if ( !qhead )
        fprintf(stderr, "qhead is NULL\n");
    else {
        fprintf(stderr, "queue:\n");
        l = qhead;
        do {
            fprintf(stderr, " (tid=%lu tnext=%p tprev=%p)\n", l->tid, l->tnext,
                l->tprev);
            l = l->tnext;
        } while ( l != qhead );
        fprintf(stderr, "\n");
    }
}

#ifdef TESTING
#define N 5
static thread nt(int tid) {
    /* allocate a new context and give it an id. Returns the thread */
    thread new;
    new = malloc(sizeof(context));
    if ( new ) {
        new->tid = tid;
        new->stack = NULL;
        new->stacksize = 0;
        new->status = MKTERMSTAT(LWP_LIVE, 0);
        new->lib_one = NULL;
        new->lib_two = NULL;
        new->sched_one = NULL;
        new->sched_two = NULL;
        new->exited = NULL;
        memset(&new->state, 0, sizeof(new->state));
    }
    return new;
}

int main(int argc, char *argv[]) {
    thread t;
    int i;

    printf("Creation:\n");
    for(i=0; i<N; i++) {
        t = nt(i+1);
        RoundRobin->admit(t);
        pq();
    }
    printf("\nAdvancement:\n");
    for(i=0; i<N; i++) {
        t = RoundRobin->next();
        printf("scheduling %ld ", t->tid);
        pq();
    }
    printf("\nRemoving in the middle:\n");
    printf(" Before ");
    pq();
    t = qhead->tnext->tnext;
    RoundRobin->remove(t);
    printf(" After ");
    pq();

    printf("\nReadmitting:\n");
    printf(" Before ");
    pq();
}

```

```
RoundRobin->admit(t);
printf(" After      ");
pq();

printf("\nRemoving qhead:\n");
for(i=0;i<N;i++) {
    printf(" Before ");
    pq();
    RoundRobin->remove(qhead);
    printf(" After      ");
    pq();
}
}
#endif
```

150

```
#ifndef SNAKEH
#define SNAKEH
#include <lwp.h>

typedef enum {NW,N,NE,W,E,SW,S,SE} direction;
#define NUMDIRS 8

typedef struct point_t {
    int x;
    int y;
} sn_point;

typedef struct snake_st {
    direction dir;
    int len;
    int color; /* color pair to use if colored snakes */
    sn_point *body;
    tid_t lw_pid; /* useful for playing with scheduling */
    struct snake_st *others; /* a utility link to find all snakes again */
} *snake;

/* Colors range from 1 (blue on black) to 8 ( black on black).
 */
#define MAX_VISIBLE_SNAKE 7

extern int start_windowing();
extern void end_windowing();
extern snake new_snake(int y, int x, int len, int dir, int color) ;
extern void free_snake(snake s);
extern void draw_all_snakes();
extern void run_snake(snake *s);
extern void run_hungry_snake(snake *s);
extern void kill_snake();
extern unsigned int get_snake_delay();
extern void set_snake_delay(unsigned int msec);
extern snake snakeFromLWpid(tid_t lw_pid);

#endif
```

10

20

30


```
#ifndef STACKSH
void *addstack(thread t);
void removestack(thread t);

#if defined (__x86_64)      /* X86 only code */
#define SP rsp
#else /* END x86 only code */
#error "This stack manipulation code can only be compiled on an x86"
#endif

#endif
```

10

```

#include <lwp.h>
#include <stdio.h>
#include <stdint.h>
#include <stdlib.h>
#include <string.h>
#include <sys/mman.h>
#include <sys/resource.h>
#include <sys/time.h>
#include <unistd.h>
#include "stacks.h"
10

#define DEFAULTSTACK (1<<20)
#define ALIGNFACTOR 16UL
#define WORDSIZE sizeof(unsigned long)

static size_t getstacksize(void) {
    /* Get the stack resource limit. Perhaps later try to catch the
     * SEGV (would require sigaltstack(2))
     * If this is not an even multiple of the page size, round up.
     */
    size_t ss = DEFAULTSTACK,ps;
    struct rlimit rlim;
    if ( -1 != getrlimit(RLIMIT_STACK, &rlim) ) {
        if (rlim.rlim_cur != RLIM_INFINITY )
            ss = rlim.rlim_cur;
        } else {
            perror("getrlimit(RLIMIT_STACK)");
        }
        /* round up to a multiple of page size, if possible */
        if ( -1 == (ps = sysconf(_SC_PAGE_SIZE)) ) {
            perror("sysconf(_SC_PAGE_SIZE)"); /* oh, well */
            exit(EXIT_FAILURE);
        } else {
            if ( ss % ps )
                ss += ps - ss % ps; /* round up */
        }
    }

    return ss;
}
40

void *addstack(thread t) {
    /* Add a stack to the given thread. Returns a pointer to the
     * new stack, or NULL on error.
     * The thread's %rsp is set to a value at the bottom appropriate
     * for an aligned stack frame.
     */
    unsigned long *s=NULL;
    size_t howbig;

    if ( t ) {
        howbig = getstacksize();
        50

        s = mmap(NULL,
            /* Where */
            howbig, /* How big? */
            PROT_READ|PROT_WRITE, /* Protections */
            MAP_PRIVATE|MAP_ANONYMOUS|MAP_STACK, /* flags */
            -1, /* fd (some require -1) */
            0); /* offset */

        if ( s == MAP_FAILED ) {
            60
            s = NULL;
        } else {
            uintptr_t is;
            t->stack = s;
            t->stacksize = howbig / WORDSIZE;
            is = (uintptr_t) (t->stack + t->stacksize);
            t->state.SP = is - is%ALIGNFACTOR;
        }
    }
    return s;
}
70

```

```
void removestack(thread t) {
    /* Remove (and deallocate) a stack from the given thread.
    */
    if ( t ) {
        if ( munmap(t->stack,t->stacksize * WORDSIZE) )
            perror("munmap");
        t->stack = NULL;
        t->stacksize = 0;
        t->state.SP = 0;
    }
}

#ifdef DEBUG
int main(int argc, char *argv[]) {
    printf("Stack size will be %ld\n",getstacksize());
    return 0;
}
#endif
```

80

90

```
#ifndef TIDH
#define TIDH
#include <lwp.h>

tid_t  create_tid(thread lwp);
thread tid2thread(tid_t tid);
int    tid2nonce(tid_t tid);

#endif
```

```

/*
 * We're going to do some magic here.  On a 32-bit machine the tid will just
 * be the pointer to the thread and we have no real error checking.  On a
 * 64-bit one, though, there's (currently) really only a 47-bit virtual
 * address space, so we can use the remaining 17-bits to hold a nonce to
 * verify that the address isn't being re-used.
 */
#include <lwp.h>
#include <setjmp.h>
#include <stdint.h>
#include <stdio.h>
#include <stdlib.h>
#include <signal.h>
#include "tid.h"

static tid_t read_tid_segv_proof(thread lwp);

typedef void (*sigfun)(int);

/* Sanity check: This won't compile if a pointer won't fit safely in
 * an unsigned long
 */
#define TESTSIZE ((sizeof(uintptr_t)==sizeof(tid_t))?1:-1)
typedef int verify_sizes_match[TESTSIZE];
#undef TESTSIZE

/* linux x86_64 currently only uses a 47-bit virtual address space.
 * cram a nonce into the upper 17 bits.
 * 386 doesn't, so the nonce goes to nothing.
 */
#if defined(_x86_64)
#define NONCEMASK 0x1ffff
#define NONCESHIFT 47
#else
#error "This only works on x86 for now"
#endif

#define THREADMASK (~(NONCEMASK<<NONCESHIFT))

int tid2nonce(tid_t tid) {
    /* return the nonce portion of the tid */
    return (tid>>NONCESHIFT)&NONCEMASK;
}

tid_t create_tid(thread lwp) {
    /* Build a tid out of a nonce and a thread address.  This will
     * be ignored in 32-bit mode.
     */
    tid_t res;
    static int nonce;

    nonce++;
    if ( (long)lwp & ~THREADMASK ) {
        /* pointer bleeds into the nonce space.  We're cooked. */
        fprintf(stderr,"%s: pointer too big.  Time to refactor.\n",
            __FUNCTION__);
        exit(EXIT_FAILURE);
    }

    res = ((nonce&NONCEMASK)<<NONCESHIFT) | (tid_t)lwp;

    return res;
}

thread tid2thread(tid_t tid) {
    /* If non-NULL, extract the thread pointer from a tid.
     * Checks to see if it points to a valid thread with this tid.
     * Returns NULL if invalid;
     */
    thread lwp;
    tid_t stored_tid;

```

```

lwp = (thread) (tid&THREADMASK);

if ( lwp != NULL ) {
    stored_tid = read_tid_segv_proof(lwp);

    if ( stored_tid != tid ) {
        lwp=NULL;
    }
}
}
return lwp;
}

#define OOPS 42
static jmp_buf backhere;

static void catch_segvs(int sig){
    /* catch segvs */
    siglongjmp(backhere, OOPS);
}

static tid_t read_tid_segv_proof(thread lwp){
    struct sigaction sa, old;
    tid_t res;

    /* set up the handler */
    sa.sa_handler=(sigfun)catch_segvs;
    sigemptyset(&sa.sa_mask);
    sa.sa_flags=0;
    if ( -1 == sigaction(SIGSEGV,&sa,&old) ) {
        perror("sigaction");
        exit(EXIT_FAILURE);
    }

    /* try it */
    if (sigsetjmp(backhere,1) == OOPS) {
        res = NO_THREAD;
    } else {
        res = lwp->tid;
    }

    /* restore whatever the old one was */
    if ( -1 == sigaction(SIGSEGV,&old,NULL) ) {
        perror("sigaction");
        exit(EXIT_FAILURE);
    }

    return res;
}

#ifdef DEBUG_TID
int main () {
    thread t;
    tid_t tid;

    read_tid_segv_proof(NULL);
    read_tid_segv_proof(NULL);

    printf(" NONCEMASK: %p\n",NONCEMASK);
    printf("THREADMASK: %p\n",THREADMASK);
    printf("NONCESHIFT: %p\n",NONCESHIFT);

    t = malloc(sizeof (context));
    tid = create_tid(t);
    t->tid = tid;

    printf("Submitted:  t:  %p tid:  %p\n", t, tid);

    t = tid2thread(tid);
    printf("Recovered:  %p tid:  %p\n", t, tid);

```

```
t = tid2thread((tid&THREADMASK));

read_tid_segv_proof(NULL);
read_tid_segv_proof(NULL);

exit(0);

}

void lwp_stop(){
    fprintf(stderr,"%s called\n",__FUNCTION__);
    exit(1);
}
#endif
```

150

```
#ifndef UTILH
#define UTILH
void block_signals() ;
void unblock_signals() ;
#endif
```



```
#include <signal.h>
#include <lwp.h>

#ifdef PREEMPTIVE
static int howdeep=0;
#endif

void block_signals() {
    #ifdef PREEMPTIVE
    /* block alarms during the delicate bit */
    sigset_t alrm;
    sigemptyset(&alrm);
    sigaddset(&alrm,SIGALRM);
    if ( -1 == sigprocmask(SIG_BLOCK, &alrm, NULL ) ) {
        perror("sigprocmask");
    }
    howdeep++;
    #endif
}

void unblock_signals() {
    #ifdef PREEMPTIVE
    howdeep--;
    if ( ! howdeep ) {
        /* unblock alarms.  We're good to go */
        sigset_t alrm;
        sigemptyset(&alrm);
        sigaddset(&alrm,SIGALRM);
        if ( -1 == sigprocmask(SIG_UNBLOCK, &alrm, NULL ) ) {
            perror("sigprocmask");
        }
    }
    #endif
}
```