Assignment 2 Solutions

cpe 453 Winter 2010

```
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, April 21st.[1]
This assignment is to be done individually.

Program: Support for Lightweight Processes (`lwp.c`)

This week's assignment requires you to implement support for lightweight processes (threads) under linux using the GNU C Compiler(gcc). A lightweight process is an independent thread of control—sequence of executed instructions—executing in the same address space as other lightweight processes. Here you will implement a non-preemptive user-level thread package.

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

`new_lwp(function,argument,stacksize)`	create a new LWP
`lwp_getpid()`	return pid of the callingLWP
`lwp_exit()`	terminates the calling LWP
`lwp_yield()`	yield the cpu to another LWP
`lwp_start()`	start the LWP system
`lwp_stop()`	stop the LWP system
`lwp_set_scheduler(scheduler)`	install a new scheduling function

Table 1: The functions necessary to support 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 32-bit Intel x86 CPU compiling with `gcc`. This is the environment found on the Linux machines in the CSL and `vogon` (but not `falcon`).

[1] Note that April 23rd is a furlough day, so, if you're using late days, be aware that the last office hours will be on the 22nd.

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.

In addition to the stack, the x86 CPU running in protected mode doing only integer arithmetic[2] has eight registers of interest, shown in Table 2. Since C has no way of naming registers, I have provided macros below that will allow you to access these registers.

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

Table 2: Integer registers of the x86 CPU

Stack structure: The gcc calling convention

The steps of the convention are as follows (illustrated in Figures 1a–f):

a. **Before the call** Caller pushes parameters onto the stack in reverse order.

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

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

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

Then, it adjusts the stack pointer to leave room for any locals it may need.

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

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

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

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

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

[2] I don't make any guarantees at all about what I'm giving you if you try and do floating point computations in your threads.

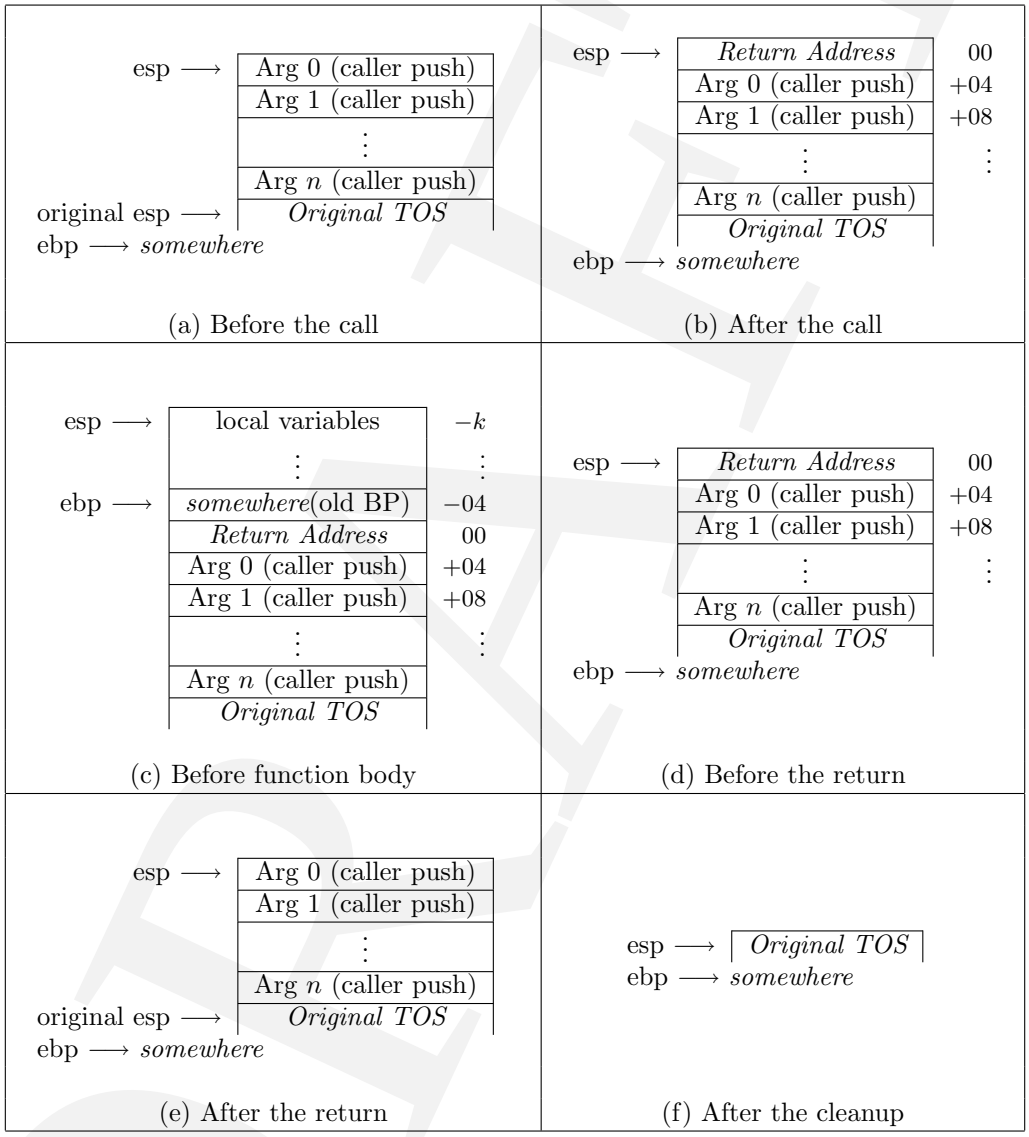

Figure 1: Stack development (Remember that the real stack is upside-down)

More stack structure: The lwp.h macros

`lwp.h` contains four macros that insert assembly instructions into your functions to save or restore context information. These are listed in Table 2.

`SAVE_STATE()`	pushes all registers but `%esp` onto the stack in the order shown in Figure 3.
`RESTORE_STATE()`	pops all registers but %esp onto the stack.
`SetSP(var)`	sets `%esp` to the value of `var`
`GetSP(var)`	sets `var` to the value of `%esp`

Figure 2: Macro functions provided by `lwp.h`.

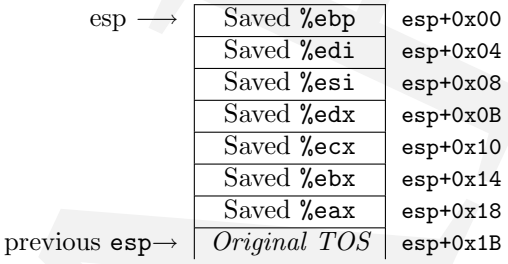

Figure 3: The stack after `SAVE_STATE()`

LWP system architecture

Everything you need is defined in `lwp.h`, Figure 4.

How to get started:

1. Allocate a stack for each LWP
2. Build a stack frame on the stack so that `RESTORE_STATE()` will properly return to the lwp's function with the stack and registers arranged as it will expect. **This involves making the stack look as if the thread called you and was suspended.**

   Remember that the base pointer (`ebp`) you restore will be copied to the stack pointer (`esp`) as part of the `leave` instruction before the function returns.
3. When `lwp_start()` is called:
   (a) save the "real" context with `SAVE_STATE()`
   (b) save the "real" stack pointer somewhere where `lwp_stop()` can find it,
   (c) pick one of the lightweight processes to run and switch to its stack
   (d) Load its context with `RESTORE_STATE()` and you should be off and running.

The semantics of the individual functions are defined in Table 3

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

typedef struct context_st {
  unsigned long pid;          /* lightweight process id  */
  unsigned long *stack;       /* pointer to stack returned by malloc() */
  unsigned long stacksize;    /* Size of allocated stack */
  unsigned long *sp;          /* current stack pointer   */
  /*  .... other things if necessary ... */
} lwp_context;


/* Process context information.  "Normally" these would be declared
 * static so that nobody outside the file could look at them, but
 * since we want to make it possible for the user to supply an external
 * scheduling function we need to make these available.
 *   (Not really.  A better way would be to have the user supply a comparison
 *    function, but that would make the scheduler much more complicated.)
 */
extern lwp_context lwp_ptable[];/* the process table          */
extern int lwp_procs;           /* the current number of LWPs  */
extern int lwp_running;         /* the index of the currently running LWP */


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

/* lwp functions */
extern int  new_lwp(lwpfun,void *,size_t);
extern void lwp_exit();
extern unsigned long lwp_getpid();
extern void lwp_yield();
extern void lwp_start();
extern void lwp_stop();
extern void lwp_set_scheduler(schedfun sched);

/* Macros for stack manipulation:
 *
 *  SAVE_STATE()    Pushes all general (non floating–point) registers on the
 *                  stack except the stack pointer.
 *  RESTORE_STATE() Pops all general (non floating–point) registers saved
 *                  by SAVE_STATE() off the stack in reverse order.  RS()
 *                  also copies the base pointer to the stack pointer as
 *                  is done by the "leave" instruction in case the compiler
 *                  optimizes that away.
 *  GetSP(var)      Sets the given variable to the current value of the
 *                  stack pointer.
 *  SetSP(var)      Sets the stack pointer to the current value of the
 *                  given variable.
 *
 * These macros should ONLY be used as the very first or very last
 * act of a function.
 */
#ifdef _i386        /* X86 only code */

#define BAILSIGNAL SIGSTKFLT

#define SAVE_STATE() \
  asm("pushl %%eax":: );\
  asm("pushl %%ebx":: );\
  asm("pushl %%ecx":: );\
  asm("pushl %%edx":: );\
  asm("pushl %%esi":: );\
  asm("pushl %%edi":: );\
  asm("pushl %%ebp":: )

#define GetSP(sp)  asm("movl  %%esp,%0": "=r" (sp) : )

#define SetSP(sp)  asm("movl  %0,%%esp":          : "r" (sp)  )

#define RESTORE_STATE() \
  asm("popl  %%ebp":: );\
  asm("popl  %%edi":: );\
  asm("popl  %%esi":: );\
  asm("popl  %%edx":: );\
  asm("popl  %%ecx":: );\
  asm("popl  %%ebx":: );\
  asm("popl  %%eax":: );\
  asm("movl  %%ebp,%%esp":: )  /* restore esp in case leave is not used */

#else /* END x86 only code */
#error "This stack manipulation code can only be compiled on an x86"
#endif

/* LWP_PROC_LIMIT is the maximum number of LWPs active */
#ifndef LWP_PROC_LIMIT
#define LWP_PROC_LIMIT 30
#endif

#endif
```

Figure 4: Definitions and prototypes for LWP: `lwp.h`

`new_lwp(function,argument,stacksize);`	Creates a new lightweight process which calls the given function with the given argument. The new processes's stack will be `stacksize` words. The LWP's process table entry will include: `pid` a unique integer process id `stack` a pointer to the memory region for this thread's stack `stacksize` the size of this thread's stack **in words** `sp` this thread's current stack pointer (top of stack) `new_lwp()` returns the (lightweight) process id of the new thread, or -1 if more than `LWP_PROC_LIMIT` threads already exist.
`lwp_getpid();`	Returns the pid of the calling LWP. The return value of `lwp_getpid()` is undefined if not called by a LWP.
`lwp_yield();`	Yields control to another LWP. Which one depends on the scheduler. Saves the current LWP's context, picks the next one, restores that thread's context, and returns.
`lwp_exit();`	Terminates the current LWP, frees its resources, and moves all the others up in the process table. If there are no other threads, calls `lwp_stop()`.
`lwp_start();`	Starts the LWP system. Saves the original context and stack pointer (for `lwp_stop()` to use later), picks a LWP and starts it running. If there are no LWPs, returns immediately.
`lwp_stop();`	Stops the LWP system, restores the original stack pointer and returns to that context. (Wherever `lwp_start()` was called from. `lwp_stop()` does not destroy any existing contexts, and thread processing will be restarted by a call to `lwp_start()`.
`lwp_set_scheduler(scheduler);`	Causes the LWP package to use the function `scheduler` to choose the next process to run. `(*scheduler)()` must return an integer in the range $0 \ldots$ `lwp_procs` $- 1$, representing an index into `lwp_ptable`, or -1 if there is no thread to schedule. If `scheduler` is `NULL`, or has never been set, the scheduler should do round-robin scheduling.

Table 3: The LWP functions

Tricks and Tools

- a core dump may mean
  - a stack overflow.
  - stack corruption
  - all the other usual causes
- For playing with the snakes, use an xterm, not a Gnome terminal. The Gnome terminal has very poor refresh characteristics. To run an xterm, just run "xterm".
- Use the CSL linux machines
- 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.
- Using precompiled libraries.

  To use a precompiled library file, lib*name*.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 `-l`*name* flag tells it to include the archive file `lib`*name*`.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
  ```
- For less credit.

  For a lower grade (but a working program) you may omit the `lwp_set_scheduler()` functionality and just do round-robin scheduling. If you choose this route, be sure to document it in your README.

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
lwp.h	Header file for lwp.c
libPLN.a	precompiled library of lwp functions (for testing)
libsnakes.a	precompiled library of snake functions
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.

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 compatabile with the distributed one, but you may extend it.
- A makefile (called `Makefile`) that will build `liblwp.a` from your source when invoked with no target or with the target "`liblwp.a`".
- A README file that contains:
  - Your name, including your login name 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.9
lwp.c	p.10
lwp.h	p.14

```
CC     = gcc

CFLAGS  = -Wall -g -I.

AR     = ar r

RANLIB  = ranlib

LIB    = liblwp.a

OBJS   = lwp.o

SRCS   = lwp.c

HDRS   =

EXTRACLEAN = core liblwp.a

all:    $(LIB)

allclean: clean
        @rm -f $(EXTRACLEAN)

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)

# DO NOT DELETE

lwp.o: lwp.h
```

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <signal.h>
#include "lwp.h"

/* Process context information.  "Normally" these would be declared
 * static so that nobody outside the file could look at them, but
 * since we want to make it possible for the user to supply an external
 * scheduling function we need to make these available.
 *   (Not really.  A better way would be to have the user supply a comparison
 *    function, but that would make the scheduler much more complicated.)
 */
lwp_context lwp_ptable[LWP_PROC_LIMIT];
int lwp_procs   = 0;
int lwp_running = -1;

/* local data */
static int      nextid  = 0;  /* used for issuing process ids */
static unsigned long *SavedStack;    /* saved stack */
static schedfun Scheduler=NULL; /* function pointer for scheduler function */

/* local forward declarations and useful stuff */
#define STACK_MAGIC 0x00ABCDEF
#define MAGIC_DEPTH 32          /* place is in a little from the end */
static void check_stacks();     /* check to see if the magic number is
                                 * where it should be
                                 */

extern void lwp_set_scheduler(schedfun sched){
  /* set the lwp system's scheduler function to the given function.
   * must return an integer.
   */
  Scheduler = sched;
}

#define push(sp,val) (*(--sp)=(unsigned)(val))

unsigned long *new_intel_stack(unsigned long *sp,lwpfun func, void *arg){
  /* mock up a stack for the INTEL architecture
   * First, a frame that returns to lwp_exit() should our function actually
   * return.
   */

  unsigned long *ebp;

  push(sp,arg);             /* argument for function */
  push(sp,lwp_exit);        /* just in case this lwp tries to return */
  push(sp,func);            /* push the function's return address */
  push(sp,0x1abcdef1);      /* bogus "saved" base pointer */
  ebp=sp;                   /* note the location for use below... */
  push(sp,0x6c6f7453);      /* push an initial eax */
  push(sp,0x66206e65);      /* push an initial ebx */
  push(sp,0x206d6f72);      /* push an initial ecx */
  push(sp,0x746e6957);      /* push an initial edx */
  push(sp,0x32207265);      /* push an initial esi */
  push(sp,0x21363030);      /* push an initial edi */
  push(sp,ebp);             /* push an initial ebp */

  return sp;

}


int new_lwp(lwpfun func,void *arg, size_t ssize){
  unsigned long *stack,*sp;
  int newpid;
  int rvalue = 0;


  if ( lwp_procs == LWP_PROC_LIMIT   ) {
    rvalue = -1;              /* Too many processes already */
```

```
  } else {
    stack = malloc(ssize * sizeof(unsigned));
    if ( !stack ) {
      rvalue = -1;
    } else {
      /* Initialize the stack  */
      /* place a magic number at the beginning of the region (the limit
        * of the new stack) that can be checked later.
        */
      stack[MAGIC_DEPTH]=STACK_MAGIC;

      /* Initialize the stack with a return to lwp_exit(), and
        * build an activation for our function call
        */
      sp = stack + ssize;

      sp = new_intel_stack(sp,func,arg);

      /* create the context for the new LWP */
      newpid=nextid++;                          /* unique process id */
      lwp_ptable[lwp_procs].pid=newpid;
      lwp_ptable[lwp_procs].stack=stack;          /* base of stack */
      lwp_ptable[lwp_procs].stacksize=(unsigned)ssize; /* size of  stack */
      lwp_ptable[lwp_procs].sp=(unsigned long*)sp;     /* stack pointer */
      lwp_procs++;

      rvalue = newpid;                          /* return the new pid */
    }
  }

  return rvalue;
}

int pick_lwp() {
  /* choose the next lwp to run.   If Schedfun is defined, apply it,
    * otherwise, do simple round-robin.
    * returns -1 on error;
    */
  int next;
  if ( lwp_procs > 0 ) {
    if ( Scheduler != NULL ) {
      next = (*Scheduler)();
    } else {
      next = (lwp_running+1);
    }
    next = next%lwp_procs;        /* keep it in range... */
  } else {
    next = -1;
  }

  return next;
}

void lwp_start(){
  /* start the LWP system.
    *
    * We save the original stack and move to one of the lightweight stacks.
    */
  int next;
  unsigned long *newstack,*thisstack;

  SAVE_STATE();
  GetSP(thisstack);

  SavedStack = thisstack;
  next = pick_lwp();

  if ( next == -1 ) {
    fprintf(stderr,"%s:  No lightweight process to run.\n",__FUNCTION__);
    newstack = SavedStack;
  } else {
    lwp_running  = next;
```

```
    newstack = lwp_ptable[next].sp;
  }
  /* restore the state and we're off */
  SetSP(newstack);
  RESTORE_STATE();
}

extern int lwp_getpid() {
  /* return the pid of the currently running lwp.   Undefined if
   * threading is inactive
   */
  return lwp_ptable[lwp_running].pid;
}

extern void lwp_stop(){
  /* terminate the running of the LWP system and restore the original
   * stack
   */
  unsigned long *sp;
  SAVE_STATE();
  GetSP(sp);
  lwp_ptable[lwp_running].sp = sp;
  SetSP(SavedStack);
  RESTORE_STATE();
}

static void lwp_exit2(){
  /* actually do the termination of the current process and
   * selection of the next process
   */
  int i;

  free((void*)lwp_ptable[lwp_running].stack);
  for(i=lwp_running;i<lwp_procs-1;i++)
    lwp_ptable[i]=lwp_ptable[i+1];
  lwp_procs--;                  /* there's one fewer now */
  if ( lwp_procs ) {
    lwp_running--;              /* back up running by one to account for
                                 * the shift.   Keeps orientation the same
                                 */
    lwp_running = pick_lwp();
    SetSP(lwp_ptable[lwp_running].sp);
    RESTORE_STATE();
  } else {
    /* there are no more, we're done, so call lwp_stop().
     */
    lwp_stop();
  }
}

extern void lwp_exit(){
  /* Move off the thread's stack onto the system stack,
   * then call lwp_exit2() where it will be free()d.
   *   We do not RESTORE_STATE, because we want to leave that
   * all in place for the eventual lwp_stop().   Do be sure not
   * to touch any stack below SavedStack
   *   This move is done in two function calls in order to not
   * disturb the local variables on the stack.
   */
  SetSP(SavedStack);

  lwp_exit2();                  /* do the real exit.   Does not return */
}

extern void lwp_yield(){
  /* voluntarily relinquish the CPU to whoever's next.   Save this
   * process's state and restore the other one.
   */

  int next;
  unsigned long *sp;
```

```
  SAVE_STATE();
  GetSP(sp);

  check_stacks();               /* check for stack corruption */

  next = pick_lwp();            /* pick whoever's next */

  lwp_ptable[lwp_running].sp = sp;
  lwp_running=next;

  SetSP(lwp_ptable[next].sp);
  RESTORE_STATE();
}

static void check_stacks() {
  /* make sure the magic number we put on the stack when we created it
   * is still there.
   *    This isn't foolproof---it's an alarm rather than a prevention
   * mechanism---but it will catch some problems.
   */
  int i;
  for(i=0;i<lwp_procs;i++){
    if (lwp_ptable[i].stack[MAGIC_DEPTH] != STACK_MAGIC) {
      fprintf(stderr,"Stack overflow, lwp_pid %d.    Aborting\n",
              (int)lwp_ptable[i].pid);
      kill(getpid(),BAILSIGNAL); /* send ourselves a stackfault signal */
      lwp_stop();
    }
  }
}
```

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

typedef struct context_st {
  unsigned long pid;          /* lightweight process id  */
  unsigned long *stack;       /* pointer to stack returned by malloc() */
  unsigned long stacksize;    /* Size of allocated stack */
  unsigned long *sp;          /* current stack pointer   */
  /*  .... other things if necessary ... */
} lwp_context;


/* Process context information.  "Normally" these would be declared
 * static so that nobody outside the file could look at them, but
 * since we want to make it possible for the user to supply an external
 * scheduling function we need to make these available.
 *   (Not really.  A better way would be to have the user supply a comparison
 *    function, but that would make the scheduler much more complicated.)
 */
extern lwp_context lwp_ptable[];/* the process table          */
extern int lwp_procs;          /* the current number of LWPs  */
extern int lwp_running;        /* the index of the currently running LWP */


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

/* lwp functions */
extern int  new_lwp(lwpfun,void *,size_t);
extern void lwp_exit();
extern unsigned long lwp_getpid();
extern void lwp_yield();
extern void lwp_start();
extern void lwp_stop();
extern void lwp_set_scheduler(schedfun sched);

/* Macros for stack manipulation:
 *
 *  SAVE_STATE()    Pushes all general (non floating–point) registers on the
 *                  stack except the stack pointer.
 *  RESTORE_STATE() Pops all general (non floating–point) registers saved
 *                  by SAVE_STATE() off the stack in reverse order.  RS()
 *                  also copies the base pointer to the stack pointer as
 *                  is done by the "leave" instruction in case the compiler
 *                  optimizes that away.
 *  GetSP(var)      Sets the given variable to the current value of the
 *                  stack pointer.
 *  SetSP(var)      Sets the stack pointer to the current value of the
 *                  given variable.
 *
 * These macros should ONLY be used as the very first or very last
 * act of a function.
 */
#ifdef _i386        /* X86 only code */

#define BAILSIGNAL SIGSTKFLT

#define SAVE_STATE() \
  asm("pushl %%eax":: );\
  asm("pushl %%ebx":: );\
  asm("pushl %%ecx":: );\
  asm("pushl %%edx":: );\
  asm("pushl %%esi":: );\
  asm("pushl %%edi":: );\
  asm("pushl %%ebp":: )

#define GetSP(sp)  asm("movl  %%esp,%0": "=r" (sp) : )

#define SetSP(sp)  asm("movl  %0,%%esp":           : "r" (sp)  )

#define RESTORE_STATE() \
```

```
  asm("popl  %%ebp":: );\
  asm("popl  %%edi":: );\
  asm("popl  %%esi":: );\
  asm("popl  %%edx":: );\
  asm("popl  %%ecx":: );\
  asm("popl  %%ebx":: );\
  asm("popl  %%eax":: );\
  asm("movl  %%ebp,%%esp":: )  /* restore esp in case leave is not used */

#else /* END x86 only code */
#error "This stack manipulation code can only be compiled on an x86"
#endif

/* LWP_PROC_LIMIT is the maximum number of LWPs active */
#ifndef LWP_PROC_LIMIT
#define LWP_PROC_LIMIT 30
#endif

#endif
```