Libtask: a Coroutine Library for C and Unix

write event-driven programs without the hassle of events!
available for FreeBSD, Linux, OS X, and Solaris

libtask.tar.gz

386-ucontext.h	clock.c	power-ucontext.h	taskimpl.h
COPYRIGHT	context.c	primes.c	tcpproxy.c
Makefile	fd.c	print.c	testdelay.c
README	httpload.c	qlock.c	testdelay1.c
amd64-ucontext.h	makesun	rendez.c	
asm.S	mips-ucontext.h	task.c	
channel.c	net.c	task.h	

Libtask is a simple coroutine library. It runs on Linux (ARM, MIPS, and x86), FreeBSD (x86), OS X (PowerPC x86, and x86-64), and SunOS Solaris (Sparc), and is easy to port to other systems.

Libtask gives the programmer the illusion of threads, but the operating system sees only a single kernel thread. For clarity, we refer to the coroutines as "tasks," not threads.

Scheduling is cooperative. Only one task runs at a time, and it cannot be rescheduled without explicitly giving up the CPU. Most of the functions provided in task.h do have the possibility of going to sleep. Programs using the task functions should #include <task.h>.

* Basic task manipulation

```
int taskcreate(void (*f)(void *arg), void *arg, unsigned int stacksize);

        Create a new task running f(arg) on a stack of size stacksize.

void tasksystem(void);

        Mark the current task as a "system" task.  These are ignored
        for the purposes of deciding the program is done running
        (see taskexit next).

void taskexit(int status);

        Exit the current task.  If this is the last non-system task,
        exit the entire program using the given exit status.

void taskexitall(int status);

        Exit the entire program, using the given exit status.

void taskmain(int argc, char *argv[]);

        Write this function instead of main.  Libtask provides its own main.

int taskyield(void);

        Explicitly give up the CPU.  The current task will be scheduled
        again once all the other currently-ready tasks have a chance
        to run.  Returns the number of other tasks that ran while the
        current task was waiting.  (Zero means there are no other tasks
        trying to run.)

int taskdelay(unsigned int ms)

        Explicitly give up the CPU for at least ms milliseconds.
        Other tasks continue to run during this time.

void** taskdata(void);
```

```
        Return a pointer to a single per-task void* pointer.
        You can use this as a per-task storage place.

void needstack(int n);

        Tell the task library that you need at least n bytes left
        on the stack.  If you don't have it, the task library will call abort.
        (It's hard to figure out how big stacks should be.  I usually make
        them really big (say 32768) and then don't worry about it.)

void taskname(char*, ...);

        Takes an argument list like printf.  Sets the current task's name.

char* taskgetname(void);

        Returns the current task's name.  Is the actual buffer; do not free.

void taskstate(char*, ...);
char* taskgetstate(void);

        Like taskname and taskgetname but for the task state.

        When you send a tasked program a SIGQUIT (or SIGINFO, on BSD)
        it will print a list of all its tasks and their names and states.
        This is useful for debugging why your program isn't doing anything!

unsigned int taskid(void);

        Return the unique task id for the current task.
```

*** Non-blocking I/O**

```
There is a small amount of runtime support for non-blocking I/O
on file descriptors.

int fdnoblock(int fd);

        Sets I/O on the given fd to be non-blocking.  Should be
        called before any of the other fd routines.

int fdread(int, void*, int);

        Like regular read(), but puts task to sleep while waiting for
        data instead of blocking the whole program.

int fdwrite(int, void*, int);

        Like regular write(), but puts task to sleep while waiting to
        write data instead of blocking the whole program.

void fdwait(int fd, int rw);

        Low-level call sitting underneath fdread and fdwrite.
        Puts task to sleep while waiting for I/O to be possible on fd.
        Rw specifies type of I/O: 'r' means read, 'w' means write,
        anything else means just exceptional conditions (hang up, etc.)
        The 'r' and 'w' also wake up for exceptional conditions.
```

*** Network I/O**

```
These are convenient packaging of the ugly Unix socket routines.
They can all put the current task to sleep during the call.

int netannounce(int proto, char *address, int port)

        Start a network listener running on address and port of protocol.
        Proto is either TCP or UDP.  Port is a port number.  Address is a
        string version of a host name or IP address.  If address is null,
```

```
        then announce binds to the given port on all available interfaces.
        Returns a fd to use with netaccept.
        Examples: netannounce(TCP, "localhost", 80) or
                netannounce(TCP, "127.0.0.1", 80) or netannounce(TCP, 0, 80).

int netaccept(int fd, char *server, int *port)

        Get the next connection that comes in to the listener fd.
        Returns a fd to use to talk to the guy who just connected.
        If server is not null, it must point at a buffer of at least
        16 bytes that is filled in with the remote IP address.
        If port is not null, it is filled in with the report port.
        Example:
                char server[16];
                int port;

                if(netaccept(fd, server, &port) >= 0)
                        printf("connect from %s:%d", server, port);

int netdial(int proto, char *name, int port)

        Create a new (outgoing) connection to a particular host.
        Name can be an ip address or a domain name.  If it's a domain name,
        the entire program will block while the name is resolved
        (the DNS library does not provide a nice non-blocking interface).
        Example: netdial(TCP, "www.google.com", 80)
                or netdial(TCP, "18.26.4.9", 80)
```

* **Time**

```
unsigned int taskdelay(unsigned int ms)

        Put the current task to sleep for approximately ms milliseconds.
        Return the actual amount of time slept, in milliseconds.
```

* **Example programs**

In this directory, tcpproxy.c is a simple TCP proxy that illustrates most of the above. You can run

```
        tcpproxy 1234 www.google.com 80
```

and then you should be able to visit http://localhost:1234/ and see Google.

Other examples are:

- primes.c - simple prime sieve
- httpload.c - simple HTTP load generator
- testdelay.c - test taskdelay()

* **Building**

To build, run make. You can run make install to copy task.h and libtask.a to the appropriate places in /usr/local. Then you should be able to just link with -ltask in your programs that use it.

On SunOS Solaris machines, run makesun instead of just make.

* **Contact Info**

Please email me with questions or problems.

Russ Cox
rsc@swtch.com

* **Stuff you probably won't use at first**
* **but might want to know about eventually**

```
void tasksleep(Rendez*);
int taskwakeup(Rendez*);
int taskwakeupall(Rendez*);
```

A Rendez is a condition variable. You can declare a new one by just allocating memory for it (or putting it in another structure) and then zeroing the memory. Tasksleep(r) 'sleeps on r', giving up the CPU. Multiple tasks can sleep on a single Rendez. When another task comes along and calls taskwakeup(r), the first task sleeping on r (if any) will be woken up. Taskwakeupall(r) wakes up all the tasks sleeping on r. They both return the actual number of tasks awakened.

```
void qlock(QLock*);
int canqlock(QLock*);
void qunlock(QLock*);
```

You probably won't need locks because of the cooperative scheduling, but if you do, here are some. You can make a new QLock by just declaring it and zeroing the memory. Calling qlock will give up the CPU if the lock is held by someone else. Calling qunlock will not give up the CPU. Calling canqlock tries to lock the lock, but will not give up the CPU. It returns 1 if the lock was acquired, 0 if it cannot be at this time.

```
void rlock(RWLock*);
int canrlock(RWLock*);
void runlock(RWLock*);

void wlock(RWLock*);
int canwlock(RWLock*);
void wunlock(RWLock*);
```

RWLocks are reader-writer locks. Any number of readers can lock them at once, but only one writer at a time. If a writer is holding it, there can't be any readers.

```
Channel *chancreate(int, int);
etc.
```

Channels are buffered communication pipes you can use to send messages between tasks. Some people like doing most of the inter-task communication using channels.

For details on channels see the description of channels in http://swtch.com/usr/local/plan9/man/man3/thread.html and http://swtch.com/~rsc/thread/ and also the example program primes.c, which implements a concurrent prime sieve.