

USER LEVEL THREAD LIBRARY

J COMPONENT PROJECT REPORT

OPERATING SYSTEMS (CSE 2005)

Submitted to – Prof. Manikandan K,
SCOPE,VIT Vellore.

Submitted by

RAVI PRAKASH (18BCE0719)

In partially fulfillment for the award of the degree of

B.Tech

In

COMPUTER SCIENCE AND ENGINEERING



VIT[®]
Vellore Institute of Technology
(Deemed to be University under section 3 of UGC Act, 1956)

Vellore-632014, Tamil Nadu, India

SCHOOL OF COMPUTER SCIENCE AND ENGINEERING

November, 2019

ABSTRACT

The goal of this project is to implement a basic thread system for Linux. The goal of a thread subsystem is to provide applications that want to use threads a set of library functions (an interface) that the application can use to create and start new threads, terminate threads, or manipulate threads in different ways. We will implement a cooperative User Level Thread (ULT) library for Linux that can replace the default PThreads library.

PROBLEM DEFINATION

Cooperative user level threading is conceptually similar to a concept known as co-routines (Co-routines are general control structures where flow control is cooperatively passed between two different routines without returning), in which a programming language provides a facility for switching execution between different contexts, each of which has its own independent stack.

Cooperative threading is fundamentally different than the pre-emptive threading done by many modern operating systems in that every thread must yield in order for another thread to be scheduled. The goal of a thread subsystem is to provide applications that want to use threads a set of library functions (an interface) that the application can use to create and start new threads, terminate threads, or manipulate threads in different ways. The most well-known and wide-spread standard that defines the interface for threads on Unix-style operating systems is called POSIX threads (or pthreads). The pthreads interface defines a set of functions, a few of which we want to implement for this project. Of course, there are different ways in which the pthreads interface can be realized, and systems have implemented a pthreads subsystem both in the OS kernel and in user mode. **For this project, we aim to implement a few pthreads functions in user mode (as a library) on Linux**

LITERATURE REVIEW

There are several papers which are summarized on threading Summarization is done in form of a table which consists of paper title, advantages, issues and limitations.

Sl.No	Paper Name	Advantages	Issues And Limitations
1	Design and Implementation of Real Time User Level Threads By:Shuichi Oikawa,Hideyuki Tokuda, Tatsuo Nakajima	According to this paper's proposed design it can achieve higher performance than kernel provided threads by reducing the over-head of context switching and also, first class treatment to user-level threads can be retained by appropriate interactions between the kernel and a ULS.	More detailed timing analysis is necessary to the user level threads applications. It also requires to analyze how much pre-empt ability and predictability are improved. Can't determine the overhead of various cases of context switching.
2	Application of threads in Operating System By: A.C Dean,R.West	This paper tells us how threads are important and advantageous than processes as multiple threads allow multiple requests to be satisfied simultaneously, without having to service requests sequentially or to fork off separate processes for every incoming request.	Not much emphasis on the disadvantages including difficulty in concurrency and debugging.

3	<p>Anderson, T.E., B.N. Bershad, E.D. Lazowska and H.M. Levy (1992) Scheduler activations: Effective kernel support for the user-level management of parallelism.</p> <p>ACM Transactions on Computer Systems 10(1), 53-79.</p>	<p>Localization of hardware dependency has the effect of increasing modularity and thus portability. The most obvious advantage of this technique is that a user-level threads package can be implemented on an Operating System that does not support threads. Various thread switch optimizations are made possible when each thread has its own kernel level window save area.</p>	<p>It can't handle nonresident context save areas. Page faults can occur on both the user-level and on the user-level stack. There is no scheduler list manipulation and no locking of data structures.</p>
4	<p>"First-Class User Level Threads" by Marsh, Scott, LeBlanc, and Markatos</p>	<p>No kernel modifications needed to support threads.</p> <p>Efficient:</p> <p>Creation/deletion/switches don't need system calls</p> <p>Flexibility in scheduling: library can use different scheduling algorithms, can be application dependent.</p>	<p>Need to avoid blocking system calls [all threads block]</p> <p>Threads compete for one another Does not take advantage of multiprocessors [no real parallelism]</p>
5	<p>"Design and Implementation of an Efficient I/O Method for a Real-time User Level Thread Library" - By: Kota Abe Toshio Matsuura, Keiichi Yasumoto, Teruo Higashino</p>	<p>This paper helps to overcome the user level thread restrictions. They proposed a new technique for user level multi-thread mechanisms to avoid blocking of whole process caused by file I/O requests. They have confirmed that is method is widely usable and is also efficient.</p>	<p>This paper need to extend their work to multi processor environments. They inform that former relies on non portable method to share address spaces. The latter can share only a part of address space.</p>

6	<p>“User Level DB:a Debugging API for User-Level Thread Libraries”</p> <p>By:Kevin Pouget , Marc P´erache , Patrick Carribault and Herv´e Jourden</p>	<p>This paper introduces the User Level DB Library (ULDB), a generic implementation of the THREAD DB interface. ULDB aims at providing the thread library with a high-level interface, by abstracting away all the generic features.</p>	<p>This method currently implements all the thread semantic features supported by GDB. The support of thread synchronization debugging needs to be investigated and included in the ULDB API and in the debugger as far as GDB is concerned.</p>
7	<p>“Parallel Pthread Library(PPL):User-level Thread Library with Parallelism and Portability”</p> <p>By:Teruki Miyazakit , Chikara Sakamotot , Masayuki Kuwayamat , Keizo Saishot , and Akira Fukudas</p>	<p>This paper described the Parallel Pthread library which is a user-level thread library with parallelism and portability. With parallelism, user-level threads can be executed in parallel rather than concurrently. They implemented the first version of the library to compare the basic performance of the library with other existing user-level thread libraries.</p>	<p>They like to revise the library through implementing PPL to many operating systems and architectures, for their further studies.</p>

EXISTING APPROACH V/S OUR APPROACH

Existing system is about POSIX threads which specify a set of interfaces (functions, header files) for threaded programming commonly known Pthreads. A single process can contain multiple threads, all of which are executing the same program. These threads share the same global memory (data and heap segments), but each thread has its own stack. POSIX also requires that threads share a range of other attributes.

Each of the threads in a process has a unique thread identifier (in the type *pthread_t*). This identifier is returned to the caller of `pthread_create()`, and a thread can obtain its own thread identifier using `pthread_self()`. Thread IDs are guaranteed to be unique only within a process. The system may reuse a thread ID after a terminated thread has been joined, or a detached thread has terminated.

Our approach is a non-preemptive cooperative user-level thread library that operates similarly to the Linux pthreads library, implementing the following function:

- `ult_create`
- `ult_exit`
- `ult_yield`
- `ult_destroy`
- `ult_recv`
- `ult_send`

We are prefixing the typical function names with “ult” to avoid conflicts with the standard library. In this User Level Thread model, one thread yields control of the processor when one of the following conditions is true:

- thread exits by calling `ult_exit`
- thread explicitly yields by calling `ult_yield`

We are also using the functionality provided by the Linux functions `setcontext()`, `getcontext()`, and `makecontext()`. The key idea is that mutexes are essentially useless in a cooperative implementation, but are necessary in a preemptive implementation.

Setcontext(), Getcontext(), and Makecontext()

- `getcontext()`: Backs up the currently running context and changes the currently executing context to a backed up one.
- `Getcontext()` does not move to a new thread, `setcontext()` and `swapcontext()` does. Our thread library should implement at least these 2 features:
- Ability to create a new thread.
- Ability to switch to another thread.

In the first case, we call `getcontext()` to initialize a `ucontext_t`, allocate memory for a stack and set the stack pointer in the `ucontext_t`, then you call `makecontext()` to initialize the context with a starting function.

In the second case, we call `getcontext()` to store the context for the current thread, and `setcontext()` to switch to another thread we have previously stored. We use `swapcontext` combine the `get/setcontext` calls.

Library functions we are implementing in our project are :-

1. `ult_create()`

The original pthread create in posix library syntax and description:

```
#include <pthread.h>
```

```
int pthread_create(pthread_t *thread, const pthread_attr_t *attr)
```

```
void *(*start_routine)(void*), void *arg);
```

The `pthread_create()` function is used to create a new thread, with attributes specified by `attr`, within a process. If `attr` is `NULL`, the default attributes are used. If the attributes specified by `attr` are modified later, the thread's attributes are not affected. Upon successful completion, `pthread_create()` stores the ID of the created thread in the location referenced by `thread`.

The thread is created executing `start_routine` with `arg` as its sole argument. If the `start_routine` returns, the effect is as if there was an implicit call to `pthread_exit()` using the return value of `start_routine` as the exit status. Note that the thread in which `main()` was originally invoked differs from this. When it returns from `main()`, the effect is as if there was an implicit call to `exit()` using the return value of `main()` as the exit status.

✓ In this function we create threads by calling functions `getcontext()` to initialize and `makecontext()`.

Before we call `makecontext()` we dynamically allocate context and update the `uc_stack.ss_sp` and `uc_stack.ss_size`. Since, main thread is considered to be at first thread, we add it to the head of the queue and give it thread id (`th_id`) of 1.

Rest of the threads are given `th_id` incrementally when called to create thread. Each thread created has ACTIVE state by default.

2. `ult_exit()`

The original pthread exit in posix library syntax and description:

```
#include <pthread.h>
```

```
void pthread_exit(void *value_ptr);
```

The `pthread_exit()` function terminates the calling thread and makes the value `value_ptr` available to any successful join with the terminating thread. Any cancellation cleanup handlers that have been pushed and not yet popped are popped in the reverse order that they were pushed and then executed. After all cancellation cleanup handlers have been executed, if the thread has any thread-specific data, appropriate destructor functions will be called in an unspecified order. Thread termination does not release any application visible process resources, including, but not limited to, mutexes and file descriptors, nor does it perform any process level cleanup actions, including, but not limited to, calling any `atexit()` routines that may exist.

An implicit call to `pthread_exit()` is made when a thread other than the thread in which `main()` was first invoked returns from the start routine that was used to create it. The function's return value serves as the thread's exit status.

✓ The state of current running thread is changed to DEAD. The memory allocated to the context of current thread is freed here. However, to check for which thread should be run next, the state of current thread's attribute of `joinfrom_th` is changed to ACTIVE and the context of the next ACTIVE thread is set using the `setcontext()`.

3. `ult_yield()`

The pthreads syntax :

```
#include <pthread.h>
```

```
int pthread_yield(void);
```


`pthread_yield()` causes the calling thread to relinquish the CPU. The thread is placed at the end of the run queue for its static priority and another thread is scheduled to run.

On success, `pthread_yield()` returns 0; on error, it returns an error number.

On Linux, this call always succeeds (but portable and future-proof applications should nevertheless handle a possible error return).

✓ Function call to yield finds a next ACTIVE state thread in the queue to swap the context with the currently running thread. Using the `swapcontext()` method of `ucontext.h`.

4.ult_destroy

Pthread Syntax :

```
#include <pthread.h>
```

```
int pthread_mutex_destroy(pthread_mutex_t *mutex);
```

```
int pthread_mutex_init(pthread_mutex_t *restrict mutex,
```

```
const pthread_mutexattr_t *restrict attr);
```

```
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
```

The *pthread_mutex_destroy()* function shall destroy the mutex object referenced by *mutex*; the mutex object becomes, in effect, uninitialized. An implementation may cause *pthread_mutex_destroy()* to set the object referenced by *mutex* to an invalid value. A destroyed mutex object can be reinitialized using *pthread_mutex_init()*; the results of otherwise referencing the object after it has been destroyed are undefined.

It shall be safe to destroy an initialized mutex that is unlocked. Attempting to destroy a locked mutex results in undefined behavior.

5.ult_send

To Send a value to another thread

Syntax : `void ult_send(int dest, int val);`

6.ult_recv

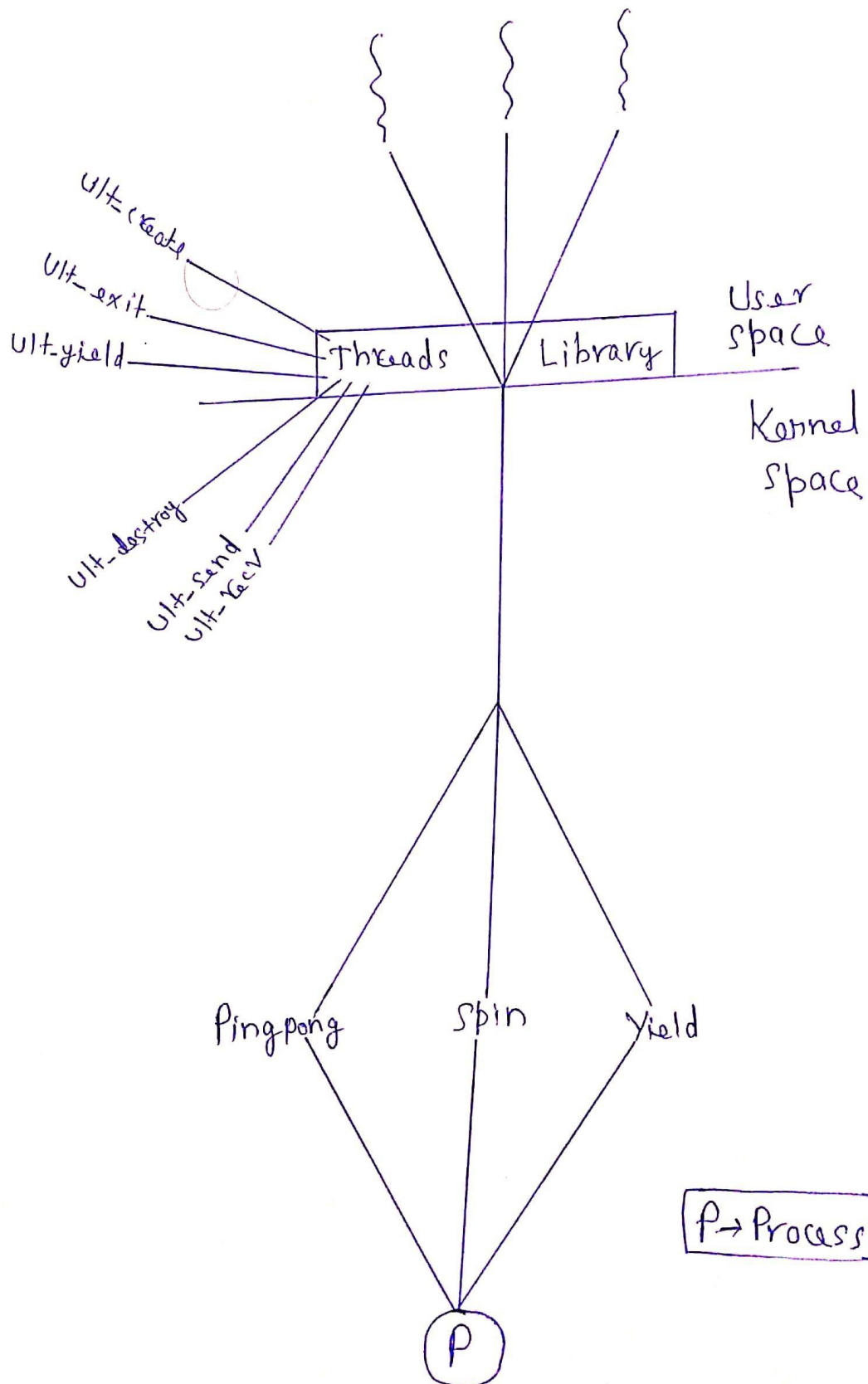
Receive a value from another thread

Syntax : `int ult_recv(int *who);`

COMPARISION BETWEEN OUR MODEL AND EXISTING MODEL

It is often desirable, for reasons of clarity, portability, and efficiency, to write parallel programs in which the number of processes is independent of the number of available processors. Several modern operating systems support more than one process in an address space, but the overhead of creating and synchronizing kernel processes can be high. The Pthreads are slow and inefficient. For instance, threads operations are hundreds of times slower than that of user-level threads. Since kernel must manage and schedule threads as well as processes. It requires a full thread control block (TCB) for each thread to maintain information about threads. As a result there is significant overhead and increased in kernel complexity. In this project we are solving this problem. Following shown are the design, algorithm and implementation of our project.

DESIGN OF OUR APPROACH



ALGORITHM AND CODE OF OUR MODEL

We start with Header file definition which consist of different macro definitions.

Header file – ult.h

```
#ifndef ULT_H
#define ULT_H
#include <ucontext.h>
// Start point of the thread
typedef void (*ult_entry)(void);
// Create (and mark as runnable) a new thread. Will not be run
// until at least the next call to env_yield(). Returns an identifier
// for the thread, or -1 if no new thread can be created.
int ult_create(ult_entry entry);
// Yield to the next available green thread, possibly the current one
void ult_yield(void);
// Indicate that we're done running
void ult_exit(void);
// Kill another green thread
void ult_destroy(int id);
// Get this environment's ID
int ult_getid(void);
// Receive a value from another green thread
int ult_recv(int *who);
// Send a value to another green thread
void ult_send(int dest, int val);
#endif
```

C file – ult.c

```
#define _GNU_SOURCE
#include <signal.h>
#include <stdlib.h>
#include <sys/mman.h>
#include <time.h>
#include "ult.h"
#define UNUSED __attribute__((unused))
#define TIMERSIG SIGRTMIN
// Environment structure, containing a status and a jump buffer.
typedef struct Env {
int status;
ucontext_t state;
int state_reentered;
int ipc_sender;
int ipc_value;
} Env;
#define NENV      1024
// Status codes for the Env
#define ENV_UNUSED  0
#define ENV_RUNNABLE 1
#define ENV_WAITING  2
#define ENV_STACK_SIZE 16384
static Env envs[NENV];
static int curenv;

void umain(void);
```

```
/* provided by user */
```

```
/* Define a "successor context" for the purpose of calling env_exit */
```

```
static ucontext_t exiter = {0};
```

```
/* Preemption timer */
```

```
timer_t timer;
```

```
const struct itimerspec ts = { {0, 0},
```

```
    {0, 1000000000},
```

```
};
```

```
static void
```

```
make_stack(ucontext_t *ucp)
```

```
{
```

```
// Reuse existing stack if any
```

```
    if (ucp->uc_stack.ss_sp)
```

```
        return;
```

```
    ucp->uc_stack.ss_size = ENV_STACK_SIZE;
```

```
}
```

```
int ult_create(ult_entry entry)
```

```
{
```

```
// Find an available environment
```

```
int env;
```

```
for (env = 0; (env < NENV); env++) {
```

```
    if (envs[env].status == ENV_UNUSED) {
```

```
        // This one is usable!
```

```

    break; } }
if (env == NENV) {
    // No available environments found
    return -1; }
envs[env].status = ENV_RUNNABLE;

    getcontext(&envs[env].state);
    make_stack(&envs[env].state);
    envs[env].state.uc_link = &exiter;
    makecontext(&envs[env].state, entry, 0);
    // Creation worked. Yay.
    return env; }

static void ult_schedule(void)
{
    int attempts = 0;
    int candidate;
    while (attempts < NENV)
    {
        candidate = (curenv + attempts + 1) % NENV;
        if (envs[candidate].status == ENV_RUNNABLE)
            curenv = candidate;
        /* Request delivery of TIMERSIG after 10 ms */
        timer_settime(timer, 0, &ts, NULL);
        setcontext(&envs[curenv].state);
    }
    attempts++; }

```

```
exit(0); }
```

```
void ult_yield(void)
{ envs[curenv].state_reentered = 0;
  getcontext(&envs[curenv].state);
  if (envs[curenv].state_reentered++ == 0)
  {
    // Save successful, find the next process to run.
    ult_schedule();
  }
  // We've re-entered. Do nothing.
}
```

```
void ult_exit(void)
{
  envs[curenv].status = ENV_UNUSED;
  ult_schedule();
}
```

```
void ult_destroy(int env)
{ envs[env].status = ENV_UNUSED;
}
```

```
int ult_getid(void)
{
  return curenv;
}
```



```

int ult_recv(int *who)
{
    envs[curenv].status = ENV_WAITING;
    ult_yield();
    if (who) *who = envs[curenv].ipc_sender;
    return envs[curenv].ipc_value;
}

```

```

void ult_send(int toenv, int val)
{
    while (envs[toenv].status != ENV_WAITING)
        ult_yield();
    envs[toenv].ipc_sender = curenv;
    envs[toenv].ipc_value = val;
    envs[toenv].status = ENV_RUNNABLE; }

```

```

static void preempt(int signum UNUSED, siginfo_t *si UNUSED, void
*context UNUSED)
{
    ult_yield();
}

```

```

static void enable_preemption(void)
{
    struct sigaction act = {
        .sa_sigaction = preempt, .sa_flags = SA_SIGINFO, };
    struct sigevent sigev = { .sigev_notify = SIGEV_SIGNAL, .sigev_signo =
TIMERSIG, .sigev_value.sival_int = 0, };
}

```

```
sigemptyset(&act.sa_mask);
sigaction(TIMERSIG, &act, NULL);
timer_create(CLOCK_PROCESS_CPUTIME_ID, &sigev, &timer);
}
```

```
static void initialize_threads(ult_entry new_main) {
curenv = 0;
```

```
getcontext(&exiter);
make_stack(&exiter);
makecontext(&exiter, ult_exit, 0);
```

```
ult_create(new_main);
setcontext(&envs[curenv].state);
}
```

```
int main(int argc UNUSED, char* argv[] UNUSED)
{
enable_preemption();
initialize_threads(umain);
return 0;
}
```

The above two files are definitions basically the source code definitions that we are going to use during implementation of

(i) Pingpong

It is basically a counter between two processes.

(ii) Spin

It is basically a test preemption by forking off a child process that just spins forever.

(iii) Yield

It causes the calling thread to relinquish the CPU. The thread is placed at the end of the run queue for its static priority and another thread is scheduled to run.

IMPLEMENTATION AND RESULT ANALYSIS

(i) Pingpong.c

// Ping-pong a counter between two processes.

// Only need to start one of these -- splits into two with fork.

```
#include <stdio.h>
```

```
#include <ult.h>
```

```
void pingpong(void);
```

```
void umain(void)
```

```
{
```

```
    int who = ult_create(pingpong);
```

```
        // get the ball rolling
```

```
    printf("send 0 from %x to %x\n", ult_getid(), who);
```

```
    ult_send(who, 0);
```

```
    pingpong(); }
```

```
void pingpong(void)
```

```
{
```

```
    int who;
```

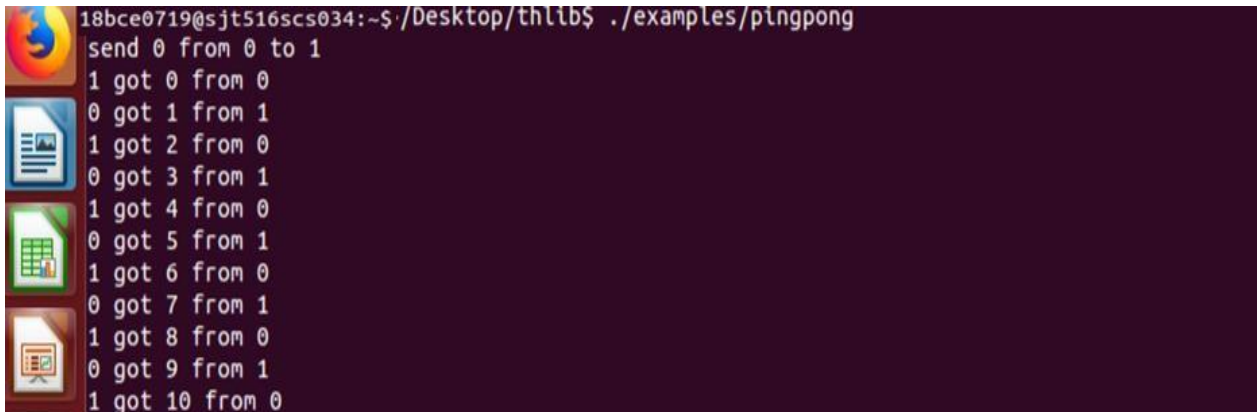
```
    while (1)
```

```

{
int i = ult_recv(&who);
printf("%x got %d from %x\n", ult_getid(), i, who);
if (i == 10)
return;
i++;
ult_send(who, i);
if (i == 10)
return;
} }

```

Execution Snapshot of Pingpong.c



```

18bce0719@sjt516scs034:~$ ./Desktop/thlib$ ./examples/pingpong
send 0 from 0 to 1
1 got 0 from 0
0 got 1 from 1
1 got 2 from 0
0 got 3 from 1
1 got 4 from 0
0 got 5 from 1
1 got 6 from 0
0 got 7 from 1
1 got 8 from 0
0 got 9 from 1
1 got 10 from 0

```

(ii) Spin.c

// Test preemption by forking off a child process that just spins forever.

// Let it run for a couple time slices, then kill it.

```
#include <stdio.h>
```

```
#include <unistd.h>
```

```
#include <ult.h>
```

```

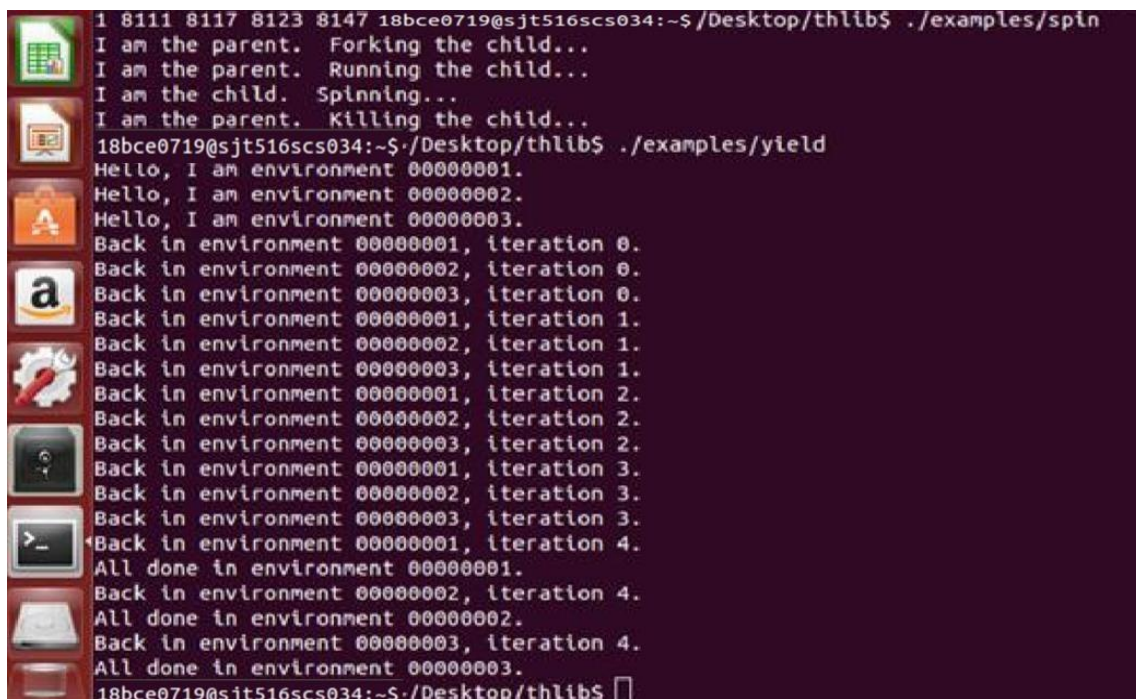
void child(void)
{
printf("I am the child. Spinning...\n");
while (1)
/* do nothing */; }

void umain(void) {
int env;

printf("I am the parent. Forking the child...\n");
env = ult_create(child);
printf("I am the parent. Running the child...\n");
ult_yield();
printf("I am the parent. Killing the child...\n");
ult_destroy(env);
}

```

Execution Snapshot of Spin.c



```

1 8111 8117 8123 8147 18bce0719@sjt516scs034:~$ ./Desktop/thlib$ ./examples/spin
I am the parent. Forking the child...
I am the parent. Running the child...
I am the child. Spinning...
I am the parent. Killing the child...
18bce0719@sjt516scs034:~$ ./Desktop/thlib$ ./examples/yield
Hello, I am environment 00000001.
Hello, I am environment 00000002.
Hello, I am environment 00000003.
Back in environment 00000001, iteration 0.
Back in environment 00000002, iteration 0.
Back in environment 00000003, iteration 0.
Back in environment 00000001, iteration 1.
Back in environment 00000002, iteration 1.
Back in environment 00000003, iteration 1.
Back in environment 00000001, iteration 2.
Back in environment 00000002, iteration 2.
Back in environment 00000003, iteration 2.
Back in environment 00000001, iteration 3.
Back in environment 00000002, iteration 3.
Back in environment 00000003, iteration 3.
Back in environment 00000001, iteration 4.
All done in environment 00000001.
Back in environment 00000002, iteration 4.
All done in environment 00000002.
Back in environment 00000003, iteration 4.
All done in environment 00000003.
18bce0719@sjt516scs034:~$ ./Desktop/thlib$

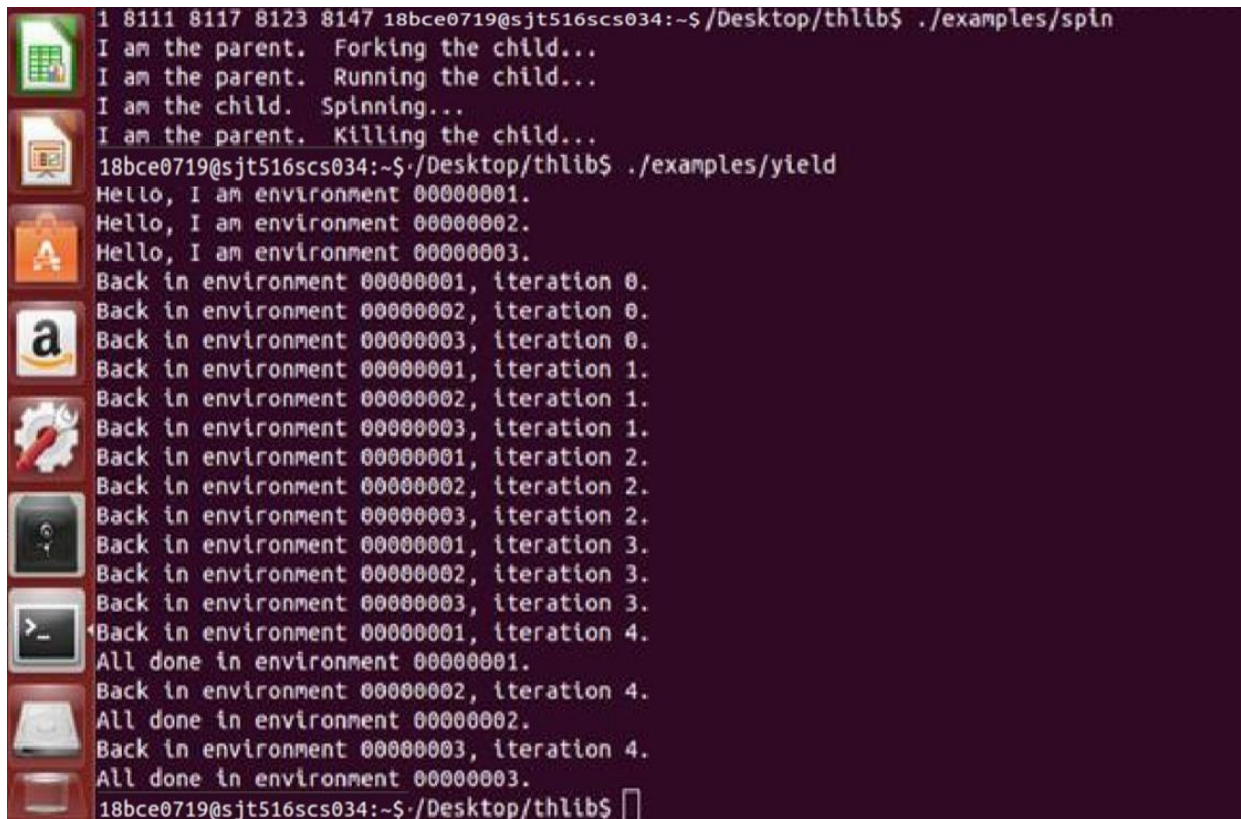
```

(iii) Yield.c

```
#include <stdio.h>
#include <ult.h>
static void yield_thread(void)
{
    int i;
    printf("Hello, I am environment %08x.\n", ult_getid());
    for (i = 0; i < 5; i++)
    {
        ult_yield();
        printf("Back in environment %08x, iteration %d.\n", ult_getid(), i);
    }
    printf("All done in environment %08x.\n", ult_getid());
}

void umain(void)
{
    int i;
    for (i = 0; i < 3; i++)
    {
        ult_create(yield_thread);
    }
    ult_exit();
}
```

Execution Snapshot of Yield.c



```
1 8111 8117 8123 8147 18bce0719@sjt516scs034:~$ ./Desktop/thlib$ ./examples/spin
I am the parent. Forking the child...
I am the parent. Running the child...
I am the child. Spinning...
I am the parent. Killing the child...
18bce0719@sjt516scs034:~$ ./Desktop/thlib$ ./examples/yield
Hello, I am environment 00000001.
Hello, I am environment 00000002.
Hello, I am environment 00000003.
Back in environment 00000001, iteration 0.
Back in environment 00000002, iteration 0.
Back in environment 00000003, iteration 0.
Back in environment 00000001, iteration 1.
Back in environment 00000002, iteration 1.
Back in environment 00000003, iteration 1.
Back in environment 00000001, iteration 2.
Back in environment 00000002, iteration 2.
Back in environment 00000003, iteration 2.
Back in environment 00000001, iteration 3.
Back in environment 00000002, iteration 3.
Back in environment 00000003, iteration 3.
Back in environment 00000001, iteration 4.
All done in environment 00000001.
Back in environment 00000002, iteration 4.
All done in environment 00000002.
Back in environment 00000003, iteration 4.
All done in environment 00000003.
18bce0719@sjt516scs034:~$ ./Desktop/thlib$
```

From the results above, we can conclude that overhead created in POSIX threads which creates kernel complexity and slows down the processes do not affect our model. Our model is independent of the POSIX threads therefore prevents complexity and does not have overheads.

CONCLUSION

We successfully implemented our model which is made by user level threads, which by the results is advantageous over POSIX threads. Modern operating systems support more than one process in an address space, but the overhead of creating and synchronizing kernel processes can be high. Since kernel must manage and schedule threads as well as processes, it requires a full thread control block (TCB) for each thread to maintain information about threads which increases kernel complexity which is reduced with the libraries we made using user level threads. We implemented user level thread library using create, exit, yield, destroy, receive, send, get_id functions which are similar to the posix library in the linux system and implementing this library in real life applications.

FUTURE WORK

User level threads have certain disadvantages which should be solved upon the near future. User-level threads requires non-blocking systems call i.e., a multithreaded kernel. Otherwise, entire process will blocked in the kernel, even if there are runnable threads left in the processes. For example, if one thread causes a page fault, the process blocks and there is a lack of coordination between threads and operating system kernel. Therefore, process as whole gets one time slice irrespective of whether process has one thread or 1000 threads within. It is up to each thread to relinquish control to other threads

REFERENCES

- [1] K. Abe, T. Matsuura, K. Taniguchi, "An implementation of portable lightweight process mechanism under BSD UNIX", Trans. of Information Processing Society of Japan, vol. 36, no. 2, pp. 296-303, 1995.
- [2] Information technology - Portable Operating System Interface (POSIX) Part1: System Application Program Interface (API) [C Language], 1996.
- [3] R. Mechler, A portable real time threads environment, 1997.
- [4] F. Mueller, "A library implementation of POSIX threads under UNIX", Proc. the Winter 1993 USENIX Technical Conf., pp. 29-41, 1993.
- [5] H. Oguma, A. Kaieda, H. Morimoto, M. Suzuki, Y. Nakayama, "A study of light-weight process library on SMP computers", Information Processing Society of Japan Sig Notes, vol. 97, no. 112, pp. 13-18, 1997.
- [6] S. Oikawa, H. Tokuda, "User-level real-time threads", Proc. 11th IEEE Workshop on Real-Time Operating Systems and Software. RTOSS '94, pp. 7-11, 1994-May.
- [7] C.Provenzano, Pthreads, [online] Available:
<http://www.mit.edu:8001/people/proven/pthreads.html>.
- [8] C. Sakamoto, T. Miyazaki, M. Kuwayama, K. Saisho, A. Fukuda, "Design and implementation of Parallel Pthread Library PPL with prallelism and portability", Trans. of the Institute of Electronics Information and Communication Engineers, vol. J80-D-I, no. 1, pp. 42-49, 1997.
- [9] H. Tatsumoto, K. Yasumoto, K. Abe, T. Higashino, T. Matsuura, K. Taniguchi, "Design and implementation of Timed LOTOS Compiler with

real-time threads", Proc. of Multimedia Distributed Cooperative and Mobile Symposium (DICO'98), pp. 555-562, 1998.

- [10] K. Yasumoto, T. Higashino, K. Abe, T. Matsuura, K. Taniguchi, "A LOTOS compiler generating multi-threaded object codes", Proc. of 8th IFIP International Conference on Formal Description Techniques (FORTE'95), pp. 271-286, 1995-Oct.
- [11] Ulrich Drepper, Ingo Molnar, "User Level DB: a Debugging API for User- Level Thread Libraries", The native POSIX thread library for Linux, 2005-Mar.
- [12] F. Herrman, C. Kaiser, "Fire - fly : A Multiprocessor Workstation", Lightweight Processes , SunOS Programming Utilities and Libraries Implementing POSIX Threads under UNIX , Proc . Second Software Engineering Research Forum , 1992.