Coursework 5 – Cooperative User-level Multithreading Library
Tomás Zilhão Borges

Feature Description

The chosen topic: user-level threads, consists on providing a library that supports the creation and management of multithreading at user-level.

In general, a thread can be seen as a sequential stream of execution within a process and a process can have multiple threads. Furthermore, threads within a process share some resources like the source code, data and files, but each one holds an independent set of registers and stack.

The name given to the procedure of managing multiple threads is multithreading and it is worth implementing for various reasons. Firstly, it improves a program's responsiveness, ie, multithreading may allow a program to continue running even if part of it is blocked or executing a lengthy operation (for example, while downloading a file from the web, we may continue web browsing), therefore increasing responsiveness to the user. Resource economy and sharing resources are two other main benefits, meaning that an application may have multiple points of activity while avoiding allocating new processes (a costly operation) and within the same address space. Lastly, multithreading benefits are even more noticeable in a multiprocessor architecture, where threads may be running in parallel in different processors.

Multithreading is generally split into 2 main categories: user-level and kernel-level multithreading. The main difference is where multithreading is implemented, and real-life applications usually take advantage of both implementations, often providing an efficient collaboration between the 2 types.

We will be mainly focusing on user-level multithreading, which is a simple and fast way of allowing a user-level application to manage multiple threads of executions; it also does not require kernel mode privileges, so the implementation will be mainly based on the user-level library.

Design Considerations

-Relationship between user-space and kernel-space

There are 3 distinct implementation models that define the relationship between user-level and kernel-level threads: many-to-one, one-to-one and many-to-many; this choice will affect the design of the proposed user-level multithreading. Since the xv6 operating system does not currently support kernel-level multithreading, we will be implementing the many-to-one model so that we can correctly cooperate with the xv6 kernel.

Elaborating on the many-to-one model, it consists on providing a user-level library that, without the kernel's knowledge, schedules multiples threads of control onto a process's single kernel thread. Hence, user-level threads provide the illusion of parallelism by multiplexing user threads on a single thread. This model in particular has the advantage of having fast, cheap thread management (it is not required to switch into kernel mode, a costly operation, and we only need to allocate a new set of registers and stack per new thread) and cheap synchronization (not dependent on the kernel scheduler). On the other hand, the fact that we are mapping multiple user threads to a single kernel threads means that when a thread makes a blocking system call, then the entire process will be blocked; in addition, there is no real parallelism (just the illusion), the many-to-one model does not take advantage of multiprocessor architectures, the access to a single kernel thread is the bottleneck.

-Type of Scheduler and Preemptive vs Cooperative

For the type of scheduler, we chose a round-robin scheduler for both its simplicity and for providing the capacity of being improved in a later phase (for example, turning it into a priority based scheduler).

Another important consideration that needs to be made is whether the thread scheduling process should be preemptive or non-preemptive (cooperative). The main difference between them is whether we give to a certain thread the control over when to give up execution or we force the stream of execution to switch to another thread by imposing a time-limit.
Ultimately, a user-level package should provide the freedom to fully control its own threads (and even the freedom to choose between cooperative and preemptive scheduling !), so we will be opting for cooperative scheduling. In addition, a sustainable implementation of a cooperative mechanism can more easily be turned into a preemptive one than the opposite.

-Considerations for xv6 Operating System

Context switching from one thread to another requires to take into consideration some specifics of the xv6 OS. Namely, one should know how to correctly allocate page aligned stacks (4096 bytes, 8912 if we include page guard), and allocate thread contexts that hold properly sized words (64 bits) and work with the context switch function (implemented in assembly).

Implementation Details

So, the first step should be to create a way of keeping track of all active threads in a given process. This can be done with the implementation of a list where each node represents a single thread. To support the implementation, standard functions for the creation and deletion of nodes and general management of the list should be provided. Each new thread is allocated on the heap as well as its components.

```c
//list of all threads
typedef struct thread_list_struct {
    thread_node *head;
    thread_node *tail;

    thread_node *current;

    int size;

} thread_list;
```

```c
//a node for each thread
typedef struct thread_node_struct {

    int id;
    char stack[STACK_SIZE];
    thread_context context;
    int state; //RUNNING, RUNNABLE

    struct thread_node_struct *next;
    struct thread_node_struct *previous;

} thread_node;
```

```c
/* FUNCTIONALITY */

//returns a new unique id
int new_id();

//adds a new node to the list
void add_thread_node(thread_node * node);

//creates a new node and adds it to the list
//when you create, you add
thread_node * new_thread_node();

//removes a node from the list
void remove_thread_node(thread_node * node);

//removes a node from the list and frees memory
//when you delete, you remove
void delete_thread_node(thread_node * node);
```

Next, we need to implement the ability to execute a context switch, so each thread holds another struct: thread_context, which takes care of saving the return address, the stack pointer and respective registers.
For the execution of the context switch itself, we will be taking advantage of the already implemented swtch.S function that is used in context switches for the kernel process scheduler. Every time we then call swtch() we load from and write to thread_context structures.

```
// Saved registers for context switches.
typedef struct thread_context_struct {
  uint64 ra;//return address: holds the return address from which swtch was called
  uint64 sp; //stack pointer

  // callee-saved
  uint64 s0;
  uint64 s1;
  uint64 s2;
  uint64 s3;
  uint64 s4;
  uint64 s5;
  uint64 s6;
  uint64 s7;
  uint64 s8;
  uint64 s9;
  uint64 s10;
  uint64 s11;

} thread_context;

extern void swtch( thread_context*, thread_context*) asm("swtch");
```

So, now we are in a position of creating a set of functions that is really meant to be used by the user, these include: init_thread() that should initiate threading (initialize the list and allocate a first thread that's already in use); exit_thread() that allows a process to go back to single threaded; create_thread(func) that allows the creation of a new thread holding a reference to a function that should run in it; delete_by_id_thread(id) which allows the deletion of a single thread based on the id returned from create_thread(); yield_thread() which abdicates the current thread from execution and calls schedule_thread() to run the next thread on the list.

```
//switches context to the next thread in the list of available threads
void schedule_thread();

/* USER */

//allows the user to start user-level multithreading
void init_thread();

//allows the user to end user-level multithreading
void exit_thread();

//gives up the current thread, switches context to the next thread in the list
void yield_thread();

//allows the user to create a new thread with a respective function that should
int create_thread(void (*func)());

//allows the user to delete an existing threads from its id (a thread id is ret
void delete_by_id_thread(int id);
```

The full implementation can be found appended to this report, files: tzb_threads.c and tzb_threads.h. A main function and basic testing of the multithreading library can also be found inside the files.

Evaluation of the Feature

So, now we may conclude that a simple and consistent implementation of a cooperative user-level multithreading library was provided.

Still, one can mention certain points in the current implementation that could be improved as well as other major improvements that were not implemented.
Namely, the implementation does not allow a thread to exist and to be out of the scheduler, one would have to delete the thread if it is not meant to be found by the scheduler. A possible solution is to add a third state to the thread (for example, INACTIVE) which would make it invisible to the scheduler.

Regarding scheduling, one can't really choose which thread to run next (it needs to be the next one on the list), a possible improvement would be a priority based scheduler.
Furthermore, even though cooperative vs preemptive greatly depends on the type of application, preemptive is ultimately more versatile and guarantees that no thread causes all the other ones to starve, so that could be an improvement (as well as the ability to turn preemption on and off). A change to preemptive scheduling would require the addition of more mechanisms such as some type of timer and synchronization like wait() or a semaphore mechanism (sleep() and wakeup()), so that threads could correctly cooperate with each other.

As a major improvement, the implementation of kernel-level multithreading would allow a change to a more complex model such as the many-to-many one, where real concurrency is achieved and we can take full advantage of a multiprocessor architecture.

Sources:
The xv6 book
https://www.tutorialspoint.com/operating_system/os_multi_threading.htm
https://cs.brown.edu/research/thmon/thmon2a.html
https://www.youtube.com/watch?v=LOfGJcVnvAk
https://pdos.csail.mit.edu/6.828/2018/homework/xv6-uthread.html
http://www.scs.stanford.edu/17sp-cs240/labs/lab1/
https://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.142.472&rep=rep1&type=pdf
https://www.cs.ucr.edu/~heng/teaching/cs179f-winter20/lec1.pdf
https://www.cs.bgu.ac.il/~os122/wiki.files/Operating%20Systems%20-%20assignment%202.pdf

APPENDIX:

```
/*
tzb_threads.h
Operating Systems Coursework 5 at University of Leeds
Cooperative User-level Multithreading Library
Tomás Zilhão Borges, 201372847
*/

#include "kernel/types.h"
#include "user/user.h"
#include "kernel/fcntl.h"

#define STACK_SIZE  8192 //page aligned!! 2 pages (4096 bytes), does the second page work as page guard somewhow??

//States of a thread
#define RUNNING     1
#define RUNNABLE    0


// Saved registers for context switches.
typedef struct thread_context_struct {
  uint64 ra;//return address: holds the return address from which swtch was called
  uint64 sp; //stack pointer

  // callee-saved
  uint64 s0;
  uint64 s1;
  uint64 s2;
  uint64 s3;
  uint64 s4;
  uint64 s5;
  uint64 s6;
  uint64 s7;
  uint64 s8;
  uint64 s9;
  uint64 s10;
  uint64 s11;

} thread_context;

//a node for each thread
typedef struct thread_node_struct {

    int id;
    char stack[STACK_SIZE]; //8192
    thread_context context;
    int state; //RUNNING, RUNNABLE

    struct thread_node_struct *next;
```

```
    struct thread_node_struct *previous;

} thread_node;

//list of all threads
typedef struct thread_list_struct {
    thread_node *head;
    thread_node *tail;

    thread_node *current;

    int size;

} thread_list;


extern void swtch( thread_context*, thread_context*) asm("swtch");


/* FUNCTIONALITY */

//returns a new unique id
int new_id();

//adds a new node to the list
void add_thread_node(thread_node * node);

//creates a new node and adds it to the list
//when you create, you add
thread_node * new_thread_node();

//removes a node from the list
void remove_thread_node(thread_node * node);

//removes a node from the list and frees memory
//when you delete, you remove
void delete_thread_node(thread_node * node);

//switches context to the next thread in the list of available threads
void schedule_thread();

/* USER */

//allows the user to start user-level multithreading
void init_thread();

//allows the user to end user-level multithreading
void exit_thread();

//gives up the current thread, switches context to the next thread in the list of available threads
void yield_thread();
```

```c
//allows the user to create a new thread with a respective function that should run in it
int create_thread(void (*func)());

//allows the user to delete an existing threads from its id (a thread id is returned by spawn_thread())
void delete_by_id_thread(int id);

/* TESTING */
void mythread();
int main(int argc, char *argv[]);
void print_thread_node(thread_node * node);
```

```c
/*
tzb_threads.c
Operating Systems Coursework 5 at University of Leeds
Cooperative User-level Multithreading Library
Tomás Zilhão Borges, 201372847
*/


#include "tzb_threads.h"


thread_list *list; //not allowing the creation of more than 1 list


//allows the user to start multithreading
void exit_thread(){

    if(list->current != list->head){ //changing back to main thread before ending it all
        printf("change to main thread to exit threading\n");
        return;
    }
    thread_node * temp = 0;
    if(list->head->next != 0) {
        temp = list->head->next;
    }
    while(temp!= 0){
        delete_thread_node(temp);
        temp = list->head->next;
    }

    free(list->head);
    free(list);

}

//allows the user to end multithreading
```

```
void init_thread()
{
    list = malloc(sizeof(thread_list));
    list->head = 0;
    list->tail = 0;
    list->current = 0;
    list->size = 0;

    thread_node * thread_0 = new_thread_node(); //first/main thread
    thread_0->state = RUNNING;

    list->current = thread_0;
}

//switches context to the next thread in the list of available threads
void schedule_thread(){

    thread_node * temp;
    if(list->current->next != 0){
        temp = list->current->next;
    }else{ //can't go any further, go back to the beginning
        temp = list->head;
    }

    while(temp != list->current)
    {
        if( (temp->state == RUNNABLE) ){

            break;
        }

        temp = temp->next; // Move to next node
    }

    if(temp != list->current){ //we found another thread, do context switch

        temp->state = RUNNING;
        thread_node * aux = list->current;
        list->current = temp;
        swtch(&aux->context, &list->current->context);


    }else{ //if no other thread and current still RUNNABLE (should i check for it?)
        list->current->state = RUNNING;

    }


}

//gives up the current thread, switches context to the next thread in the list of available threads
```

```
void yield_thread() {

  list->current->state = RUNNABLE;
  schedule_thread();
}

//allows the user to create a new thread with a respective function that should run in it
int create_thread(void (*func)()){

    thread_node * node = new_thread_node();

    node->context.sp = (uint64) (node->stack + STACK_SIZE); //sp to the top of the stack
    node->context.ra = (uint64) func; //save return address

    return node->id;

}

//returns a new unique id
int new_id(){

    static int number = 0;
    return number++; //first return will be 0
}

//adds a new node to the list
void add_thread_node(thread_node * node){

    list->size++;

    if (list->head == 0)
    {
        list->head = list->tail = node;
    }
    else //adding at the end of list
    {
        node->previous = list->tail;
        list->tail->next = node;
        list->tail = node;
    }

}

//creates a new node and adds it to the list
//when you create, you add
thread_node * new_thread_node(){

    if(list== 0)
    {
        printf("initialize threading first\n");
        return 0;
```

```
    }

    thread_node * node = malloc(sizeof(thread_node));
    node->id = new_id();


    node->state = RUNNABLE;
    node->next =0;
    node->previous=0 ;

    add_thread_node(node);

    return node;

}

//removes a node from the list
void remove_thread_node(thread_node * node){


    thread_node * temp = list->head;
    while(temp != 0)
    {
        if(node->id == temp->id){

            if (list->head->id == node->id) {
                list->head = node->next;
            }

            if (node->next) {
                node->next->previous = node->previous;
            }

            if (node->previous) {
                node->previous->next = node->next;
            }

            list->size--;
            break;
        }

        temp = temp->next; // Move to next node
    }


}

//removes a node from the list and frees memory
void delete_thread_node(thread_node *node){

    remove_thread_node(node);
```

```
    free(node);
}

//allows the user to deelete an existing threads from its id (a thread id is returned by spawn_thread())
void delete_by_id_thread(int id){

    if(id == 0){
        printf("can´t delete main thread\n");
        return;
    }

    thread_node * temp = list->head;
    while(temp != 0)
    {
        if(id == temp->id){
            break;
        }

        temp = temp->next; // Move to next node
    }
    if(temp == 0){
        printf("id does not match any thread's id\n");
        return;
    }

    if(temp->state == RUNNING){
        printf("please yield before deleting this thread\n");
        return;
    }
    else{
        delete_thread_node(temp);
    }

}


void print_thread_node(thread_node * node){

    printf("thread id: %d\n", node->id );
    printf("thread state: %d\n", node->state);

}




void mythread()
{
    // TEST#2
    //delete_by_id_thread(list->current->id);
```

```
    int count = 0;
    int i;
    for (i = 0; i < 6; i++) {
        printf( "my thread running %d\n", list->current->id);
        printf( "counter %d\n", count++);

        yield_thread();
    }
    printf("my thread: exit\n");
    yield_thread();

}

int main(int argc, char *argv[]){


    // TEST#1
    init_thread();
    create_thread(mythread);
    create_thread(mythread);

    int count = 0;
    int i;
    //printf( "my thread running %d\n", list->current->id);
    for (i = 0; i < 7; i++) {
        printf( "my thread running %d\n", list->current->id);
        printf( "counter %d\n", count++);

        yield_thread();
    }
    printf("my thread: exit\n");
    //printf("tail: %d\n", list->tail->id);
    exit_thread();


    // TEST#2, NEEDS CHANGES IN MYTHREAD
/*
    init_thread();
    int a_id = create_thread(mythread);
    create_thread(mythread);

    yield_thread();

    delete_by_id_thread(a_id);
    printf("SIZE: %d\n",list->size);

*/

    printf("end\n");
    exit(0);
```

```
}
```