Project #2
Thread Support Library (TSL)

Assigned: Feb 29, 2024 Document version: 1.3
Due date: March 24, 2024

- This project will be done in groups of 3-4 students. Group size can not be smaller than 3. You will program in C/Linux. Programs will be tested in Ubuntu 22.04 Linux 64-bit. This project needs to be done on a Linux computer with x86-64 (AMD64) architecture. Therefore, at least one member of a group should have a computer with this architecture.
- *Objectives/keywords*: Learn and practice with thread management, context switching, thread scheduling; the implementation details of context switching; internals of thread management and how a user-level threading system can be designed and implemented; keeping track of thread execution; stack discipline, procedure calls, i386 32-bit architecture.

In this project you will develop a basic user-level **thread support library** (**TSL**) that can be used by applications to create and work with threads. The threads will be managed at user level. *Cooperative* threading will be implemented (hence it is easier) as opposed to *preemptive* threading. That means a thread will explicitly and voluntarily yield (give) the CPU to another thread of an application program.

Your library will be a C library for Linux/i386 platforms. You will program in a Linux computer with 64-bit x86-64 architecture, but you will compile with the **-m32 flag** of **gcc** so that the generated code will be **32-bit i386 (IA32) code**. We can run 32-bit i386 programs on a machine with x86-64 (AMD64) architecture (it is backward compatible). An example `Makefile` will be provided. In Linux, you can learn the architecture of your machine by typing `uname -a` or `arch`.

1. Library Interface (API)

You will implement the following functions in your library. You are not allowed to change the function prototypes defined here. Internally, in your library, you can define and use additional functions. The ones below are the **interface functions**, the functions that can be called by applications (API functions).

1. `int` **`tsl_init`**`(int salg)`. You will initialize your library in this function. An application will call this function exactly once before creating any threads. The parameter `salg` is used to indicate the scheduling algorithm the library will use. On success, `0` will be return. On failure, `-1` (`TSL_ERROR`) will be returned.

2. `int` **`tsl_create_thread`**`(void (*tsf)(void *), void *targ)`. This function will create a new thread. The new thread will execute the specified function `tsf`, i.e., the thread start function (or *root* function). The application will define this thread start function, and its address will be passed to `tsl_create_thread()` as the first argument. The start function can

take one parameter of type `void *`. Therefore, the second argument to the `tsl_create_thread()` function, i.e., `targ`, is a pointer that can point to a value or structure that will be passed to the thread start function. If nothing is to be passed, `NULL` can be specified as the second argument to `tsl_create_thread()`.

As the return value, the integer identifier (tid) of the created thread will be returned. Your library will assign a unique positive integer identifier to every thread, including the main thread.

If the new thread could not be created due to some reason, then `-1` (`TSL_ERROR`) will be returned as the return value.

After creating the new thread, the main thread can continue running until it calls the `tsl_yield()` function.

3. `int` **`tsl_yield`**`(int tid)`. Wit this function, a running caller (calling) thread will yield (give) the cpu to some other thread. A context-switch from the caller thread to another thread should happen.

   If the `tid` parameter is equal to the tid of an existing thread, then that thread should be selected as the thread to run next. If `tid` parameter is equal to the special value `0` (`TSL_ANY`) the next thread to run will be selected from the ready queue according to a scheduling algorithm.

   After selecting the next thread to run, the selected thread's saved context will be reloaded to the CPU and the selected thread will start execution.

   If there is no other thread in the ready state other than the calling thread, the scheduler needs to select the calling thread to run next. That means, in this case, your library will save the calling thread context first and then restore it.

   If `tid` parameter is a positive integer but there is no ready thread with that `tid`, the function will return immediately without yielding to any thread. In this case it will return `-1` as the return value. Otherwise, the return value should be the tid of the thread to whom the cpu is yielded.

   Note that this function will not return immediately to the caller, but will return later when the calling thread (caller) is re-scheduled again.

4. `int` **`tsl_exit`**`()`. With this call, a thread will indicate to the library that it wants to get terminated. This function will mark the calling thread as deleted. The TCB and the stack of the thread may not be immediately deallocated. You can keep them until some other thread calls `tsl_join()` operation on this thread.

   The function will not return, since with this call the caller thread is terminated (ended). Another thread will be scheduled (yielded to).

   A thread can use this function at the end of its thread start function or wherever it wishes. If a thread omits this function and comes to the end of the thread start function, then the stub function (explained later) to where the thread start function will return will call this `tsl_exit()` function to cause the thread to terminate. In this way, performing a return from the start function of any thread, other than the main thread, results in an implicit call to `tsl_exit()` via the stub function.

   The main thread should be able to call this function as well. In fact this may be recommended. If we want to allow other threads to continue execution, then the main thread should terminate by calling `tsl_exit()`, rather than `exit()`.

If the calling thread is the last to end, then the whole process should be terminated (via `exit()`).

5. `int` **`tsl_join`**`(int tid)`. When called by a thread, this function will not return until the thread with identifier `tid` (call it target thread) terminates. When the target thread terminates (or has already terminated), then the function can return. Before returning, it will deallocate the resources (the stack and TCB) used by the target thread. The return value will be the tid of the target thread. If there is no thread with the indicated `tid`, then `-1` will be returned.
6. `int` **`tsl_cancel`**`(int tid)`. With this function, the calling thread will cancel another thread. The thread to be cancelled (target thread) is indicated with the `tid` argument. The cancellation will be *asynchronous*. That means the target thread will be immediately deleted (terminated), as if it has called `tsl_exit()`.
7. `int` **`tsl_gettid`**`()`. Returns the thread id (tid) of the calling thread; that means, the thread identifier assigned by the library.

2. Library Implementation

You will implement the TSL library that will have the interface (API) and behavior specified above. An application that would like to use the library will include the `tsl.h` header file and will be linked with the library by using the `-l` option of the gcc compiler (as `-ltsl`).

2.1 Thread Control Block

Each thread has a **per-thread state** that represents the working state (execution state) of the thread. A thread's cpu context is a subset of this state that must be saved and restored from/to the processor when switching threads. The **cpu context** of a thread is the cpu state (set of cpu register values) at the current time of execution. If we want to suspend the thread, the cpu context has to be saved. The saved cpu context is restored (reloaded) later to the cpu when the thread is picked up by the scheduler to re-run in the cpu.

Your library will store the context of a thread and other required information in a thread control block (**TCB**) structure. Hence you will will keep one TCB per thread. Since any application has at least one thread (main thread), your library will have at least one TCB created and used for the process (application). If the process then creates N new threads, there will be $N + 1$ threads (including the main thread), yielding the cpu to each other from time to time, and in this manner sharing the cpu and running concurrently.

A possible partial definition of the TCB structure is shown below. It will have a pointer field pointing to a context structure of type `ucontext_t`. The cpu context of a thread will be stored in this structure. A thread will also have its own stack. TCB will have a pointer field pointing to that stack.

```
typedef struct TCB {
    ...
```

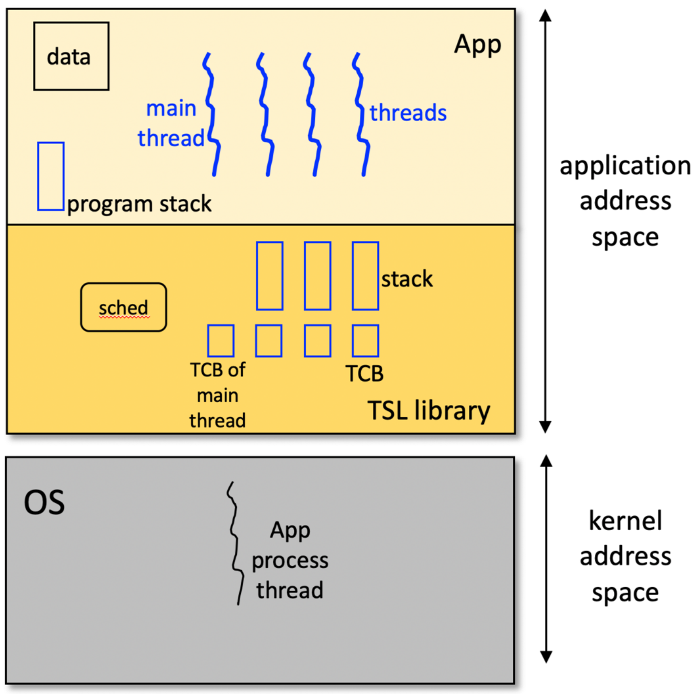

Figure 1: Figure shows the logical place of the tsl library. It will linked with an application. It will manage and schedule the application threads. The scheduling will be cooperative scheduling. This is user-space implementation of threading support (many-to-one mapping). There will be a single thread in the kernel for the whole application.

```
    int             tid;       // thread identifier
    unsigned int    state;     // thread state
    ucontext_t      context;   // pointer to context structure
    char            *stack;    // pointer to stack
    ...
} TCB;
```

2.2 Saving and Restoring CPU Context

There is an easy way to get (save) the current cpu context of a running thread: use `getcontext()` call, available in standard C library. You can use this function whenever you want to get and **store** the current cpu state of a running thread. With this function, the execution context of the running (calling) thread at that moment is saved in a context structure (in main memory) of type `ucontext_t`. The context includes the values of the physical cpu registers (physical cpu state) at the time of calling the function. Hence we do not need to write assembly code to save the cpu registers of the currently running thread to memory when we want to make a switch to another thread. We just call this function. It does the whole job of saving the physical register values and then returns. After obtaining the context, we can read and modify some of the fields of the context structure if needed.

There is also a `setcontext()` function. It is used to **load** (restore) the saved cpu state of a thread from a specified context structure into the processor, and in this manner cause the thread to start running at that context (i.e., jump to that cpu state). You will use these functions to perform thread context switch.

Another function, `makecontext()`, is available to switch context automatically. You are not allowed to use the `makecontext()` function in this project, so that you can implement part of context switching yourself and learn the details.

The prototypes of the `getcontext()` and `setcontext()` functions are given below. Please carefully read and study the manual pages of these two functions.

1. `int getcontext(ucontext_t *ucp)`: Get the current context (physical register values) and save it to the context structure pointed by `ucp`. This function returns *twice*.
2. `int setcontext(const ucontext_t *ucp)`: Load the saved context from the context structure pointed by `ucp` to processor so that the execution will continue from that context. This function does not return.

2.3 Context Structure

The `ucontext_t` structure is defined in the header file `/usr/include/sys/ucontext.h`, which is included from `/usr/include/ucontext.h`. The definition is given below.

```
/* Userlevel context.  */
typedef struct ucontext_t
  {
    unsigned long int __ctx(uc_flags);
    struct ucontext_t *uc_link;
    stack_t uc_stack;
    mcontext_t uc_mcontext;
    sigset_t uc_sigmask;
    struct _libc_fpstate __fpregs_mem;
  } ucontext_t;
```

The `uc_mcontext` field points to an architecture (i386) dependent structure in which the physical register values are stored.

2.4 Stub Function

You will use a **stub** function internally in your thread library. The new thread should start executing at that function. The execution then will continue at the thread start function (i.e., root function) specified by the application programmer. In this way, the start function specified by the application will have a wrapper function (i.e., the stub function) to return to. The stub function will be very short and will look like the following.

```
void
stub (void (*tsf) (void*), void *targ)
{
        // new thread will start its execution here
        // then we will call the thread start function
        tsf(targ);  // calling the thread start function
        // tsf will retun to here
        // now ask for termination
        tsl_exit(); // terminate
}
```

2.5 Scheduling Algorithm

When a context switch will happen, your library will select a thread to run next. This selection should be done according to a scheduling algorithm implemented in your library. You will implement at least the following algorithms.

1. `ALG_FCFS` (`1`). Select the ready thread at the head of the ready queue to run next. This will be like Round-Robin scheduling. The yielding thread is added to the tail of the ready queue. A newly created thread is added to the tail of the queue.
2. `ALG_RANDOM` (`2`). Select a ready thread randomly from the ready queue to run next. The yielding thread is added to the tail of the ready queue. A newly created thead is also added to the tail of the queue.
3. ... (`3`). Your scheduling algorithm (if you wish).

2.6 Thread Stack and Stack Discipline

The context structure contains many fields, but you will need to deal with only the stack pointer field (`REG_ESP`) and the program counter, i.e., instruction pointer, (`REG_EIP`) field. These fields are stored in the architecture dependent `mcontext_-t` structure (as an array of register values). The stack pointer field stores the saved value of the stack pointer register of the cpu. In x86 32-bit (**i386** architecture), that register is the **ESP** register and points to the top of the stack. The program counter field stores the value of the instruction pointer register of the cpu. That register is the **EIP** register in i386 architecture and points to the next instruction to be executed.

The C calling convention in i386 architecture causes the stack behave as below when we call a procedure (i.e., a function) [6].

A called function has a frame (sequence of 32 bit memory words) allocated in the stack to store its local variables, parameters, etc. When the function returns, the respective frame is deallocated by moving the stack pointer properly (moving up - towards larger addresses). The figure shows the stack frame for a function that is currently running. The stack pointer register (ESP) points to the top of the stack (stores the address of the valid word at the top of the stack). Stack grows downward, towards lower addresses in i386 architecture. The **EBP** register (**stack frame pointer**) points to the stack frame of the currently running function, i.e., to the location just after where the return address (old EIP) is stored. The **EBP** register is used to access the parameters and local variables of the function. All these can be accessed relative to the memory address stored in the EBP register. Even though the stack pointer (ESP) may change during function execution, EBP register value is fixed (hence it is a *base* - reference point - to find the local variables and parameters of the function).

When a function is **called** in a C program from a caller function, the compiler (the caller) pushes the **arguments** (parameters) of the function from right to left. In the example, there are n parameters pushed to the stack by the caller. The **return address** in the caller function is also pushed to the stack (as old EIP, after func arg

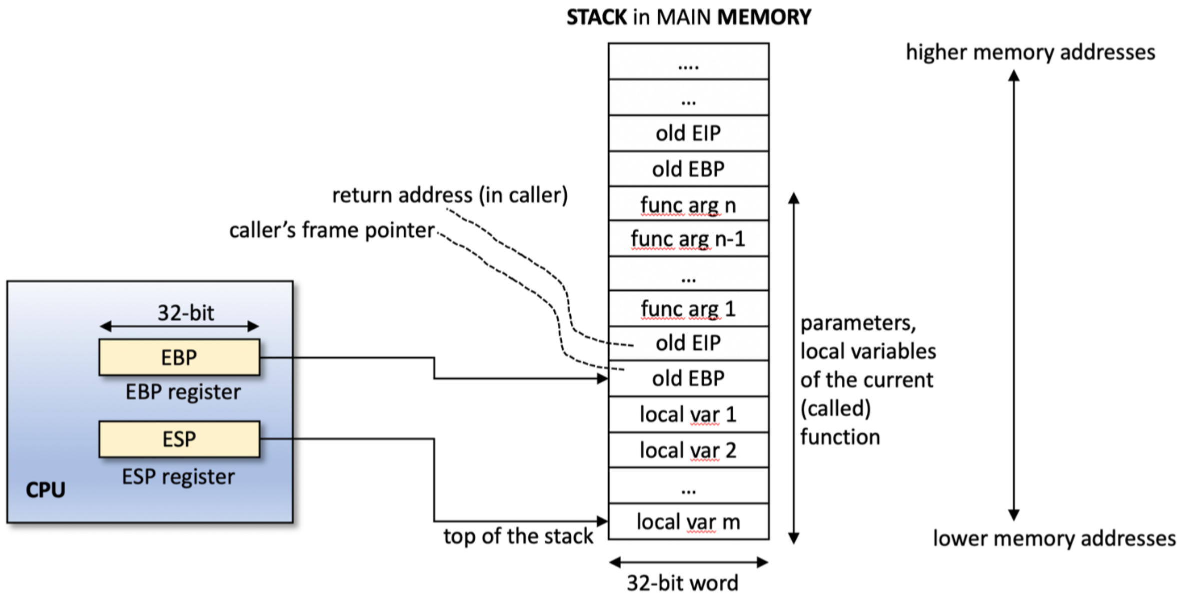

Figure 2: Procedure (function) call in a thread. A thread has a stack in memory.

1, in the figure), when `call` machine instruction is executed to jump to the called function. That means, the execution will continue at this address after the callee (the called function) returns. This is the address of the instruction just after the `call` machine instruction (used to call a procedure).

When the called function starts running, it first pushes (saves) the EBP register value to the stack (the pushed value is the old EBP – caller's frame pointer) and updates the EBP register to point to the top of the stack, where the old EBP value resides. Now EBP register points to the stack frame of the called (current) function. The called function can allocate further space in the stack for its **local variables** (if any) by moving the stack pointer (towards lower addresses). The stack pointer should point to the top valid item in the stack.

To **return** to the caller, a function simply copies the frame pointer (EBP register value) to the stack pointer register (ESP), pops the top stack item into EBP register (i.e., restores the previous frame pointer - pointer to caller's frame - to the EBP register), and uses the `ret` machine instruction to pop the **old instruction pointer** off the stack into the processor's instruction register (**EIP**). In this way execution returns to the caller function.

2.7 Library Initialization

The `tsl_init()` function will initialize your library. As part of this initialization, a ready queue structure (**runqueue** structure) should be created and initialized as well. It will keep a list of TCBs corresponding to the threads that are in ready state. If you wish you can create and initialize other queues. It is up to you how to manage the set of TCBs of the threads that can be in various states. You can keep all TCBs in a single data structure or in multiple data structures.

You will also need to allocate a TCB for the main thread of the application (process). Its state will be `RUNNING` initially. You will also assign a unique thread identifier to the main thread. It can be `1`).

2.8 Creating a New Thread

While creating a new thread, you will first create and initialize a thread control block (TCB) for the new thread. The state of the new thread will be `READY`. It will be assigned a positive unique thread identifier (tid) as well. The new TCB will be added to the ready queue.

You will also allocate space (with `malloc`) for a stack for the new thread. While running, the new thread will use this stack for its function calls. The stack should be pointed by the TCB of the thread. The size of the stack should be `TSL_STACKSIZE`.

The maximum number of new threads (not including the main thread) that can be created will be `TSL_MAXTHREADS`.

The TCB of a thread will keep information about the thread, including the context information. The **context** information will be initially obtained from the context of the running thread (creator thread). As stated above, you can obtain the context of the running thread with `getcontext`() function.

After obtaining **an initial context** with `getcontext()` into a context structure, you will change three things.

1. You will change the instruction pointer (`REG_EIP`) field of the context structure to point to the thread to run (i.e., point to the stub function). The stub function will call the thread start function defined by the application programmer.
2. You will allocate (with `malloc()`) and *initialize* a new stack.
3. You will change the stack pointer field (`REG_ESP`) of the context structure to point to the top of the new stack.

The `malloc()` call will allocate space for the new stack and will return a pointer to it (to a low address). The bottom of the stack will be that pointer value plus `TSL_STACKSIZE` (a higher address). Remember that in i386 architecture, stack grows *downward* (from high memory addresses to low memory addresses).

The rest of the context structure can (will) stay untouched.

Since we made the context's `REG_EIP` field to point to the stub function, when the context is loaded into cpu with `setcontext()`, the thread will start executing at the stub function (`REG_EIP` field value will be loaded into EIP register of the cpu and execution will start automatically from that address). The stub function will call the thread start function. When thread start function returns, the stub function will call the `tsl_exit()` to terminate the thread. Stub function does not have to return somewhere, because it will be the last function the thread will execute.

Stub function takes two arguments: address of the user specified thread function (tsf) and a parameter to that function (targ).

In your thread creation code, you will *initialize the stack* by putting these two arguments (`tsl` and `targ`) into the stack and updating the *stack pointer* field of the context structure accordingly (should point to the top of the stack). In this way the stub function will be ready to execute. The figure shows the state of the stack just before stub function starts executing.

When the stub function gets executed (when thread is scheduled), it will call the **tsf** function. This call will push the return address (EIP) to the stack and EIP will be updated to point to the beginning of the tsf, and tsf execution will start. The current EBP value will be pushed to the stack, and EBP will be updated to

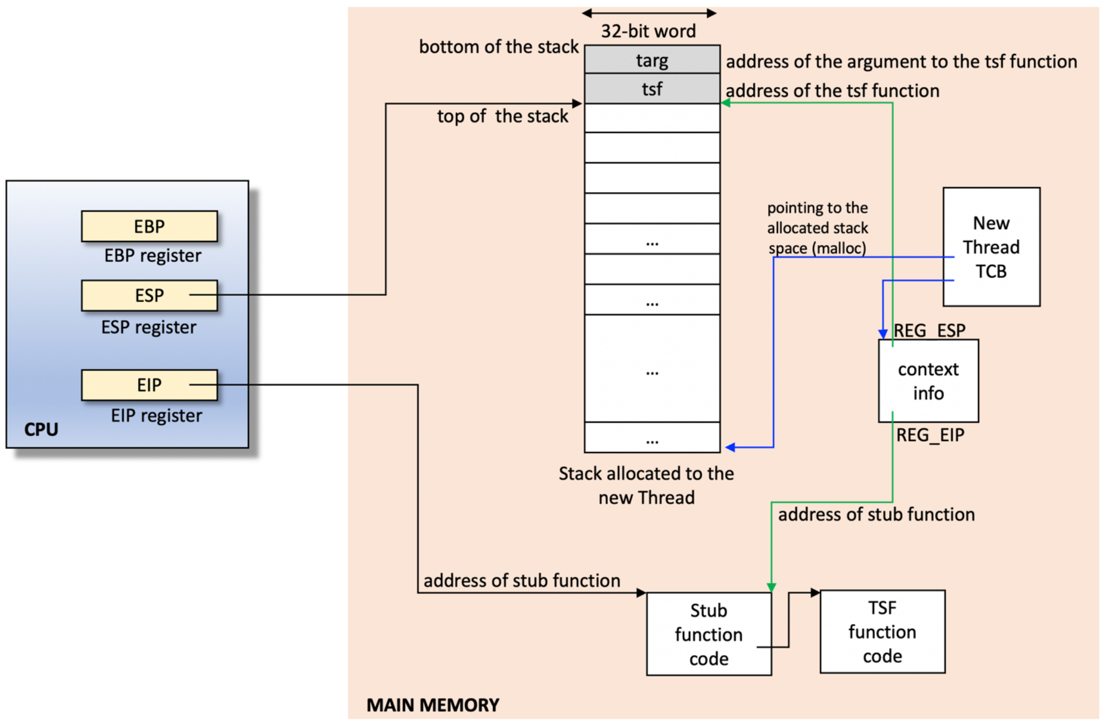

Figure 3: Stack and memory state prepared by `tsl_create_thread()` function for a new thread. The state of the thread will be `READY` and its TCB will be added to the ready queue. Later, before running the new thread in the cpu, the cpu registers will be loaded from the related context structure and therefore EIP and ESP registers will point as in the figure (just before new thread's stub starts executing).

point to the top of the stack at that moment. Depending on the tsf, space for local variables can be allocated on the stack by updating the ESP (stack pointer) register.

2.9 Yielding to Another Thread (Context Switch)

A thread will call `tsl_yield()` to give the cpu to some other thread. Inside `tsl_yield()`, the state of the caller thread will be changed from `RUNNING` to `READY` and the caller thread's TCB will be added to the ready queue. The current CPU context (of the caller) thread needs to be saved as well (with `getcontext()` call).

Another thread will be selected from the ready queue to run next. If the `tid` paramater of the `tsl_yield()` function is positive, then the respective thread (if exists) will be selected to run next. If the `tid` parameter is set to be `0` (`TSL_ANY`), the selection will be done according to a scheduling algorithm. After selection, the selected thread's saved context (reachable from the TCB of the selected thread) will be reloaded to the CPU (via `setcontext()` call) and the selected thread will start execution immediately.

Note that `getcontext()` call will return twice.

1. When the running thread X calls `getcontext()`, the current context of X (cpu state) will be saved into a context structure and `getcontext()` will return immediately. This is the first return. The calling thread X will continue running until it yields the cpu to some other thread.

2. When another thread finally yields back to this thread X by calling `setcontext()`, the cpu registers will be reloaded from the saved context of X, and X will continue execution starting at that context (which was obtained while `getcontext()` was running earlier). Hence, execution of X will continue from that point, from inside `getcontext()`, and it will return again. This is the second return.

Think about what action you should take in each case. Think about also how you can tell in which case you are. *Hint*: in the first case, you need to context switch to another thread by calling `setcontext()`. Before that you need to change your state to be `READY` (from `RUNNING`). In the second case, somebody called `setcontext`, and you are re-running. You need to change your state to be `RUNNING` (if it is not done already).

Not that you will mainly call `getcontext()` from inside `tsl_yield()`, which is doing a context switch to another thread. You will also call it in `tsl_init()` and `tsl_create_thread()` where you will allocate and initialize a TCB for the main thread and a new thread.

2.10 Terminating the Running Thread (Thread Exit)

A thread will call `tsl_exit()` to get terminated. The `tsl_exit()` will mark the thread as ended (you can change its state to something like `ENDED`).

The stack and TCB of the calling (running) thread will not be deleted (deallocated) immediately; because a running thread should not deallocate its stack by itself. Therefore, this function will just mark the thread as `ENDED` (it will be no longer schedulable). But the resources of the thread (its TCB and stack) will only be deleted (freed), when some other thread calls `tsl_join()` function on this thread. Hence, that other thread, calling `tsl_join()`, will free the stack and TCB of the thread that called `tsl_exit()`.

2.11 Waiting for a Thread to Terminate (Join)

When a thread calls the `tsl_join()` function, it will wait (yield the cpu) until some other thread (target thread), whose tid is specified as an argument to the function, terminates. Up to that time `tsl_join()` will not return.

3. Testing / Application

You need to write some applications to test your library. We will also write test applications that will be linked with your library to create and use threads.

A program that will use your library will basically operate as follows.

```
#include "tsl.h"
int main(int argc, char *argv[])
{
    // initialize the libary (tsl_init)
    // create threads (tsl_create)
    // wait for the threads to finish (tsl_join)
    // exit or tsl_exit
}

// start function of thread i
```

```
void *tsf_i(void *targ)
{
    // do some work
    // yield  (tsl_yield)
    // repeat if needed
    // return or tsl_exit
}
```

4. Experiments and Report

You will design and conduct some performance experiments. You will then interpret them. You will put your results, interpretations, and conclusions in a report.pdf file. You will include this file as part of your uploaded package. Experiments and the related report will be **20% of the project grade**.

5. Submission

Submission will be the same as the previous project; except a group will have 3-4 students now. Hence adapt the following for a group that has 3-4 students.

Put all your files into a directory named with your Student ID. If the project is done as a group, the IDs of both students will be written, separated by a dash '−'. In a `README.txt` file, write your name, ID, etc. (if done as a group, all names and IDs will be included). The set of files in the directory will include `README.txt`, `Makefile`, and program source file(s). We should be able to compile your program(s) by just typing `make`. No binary files (executable files or .o files) will be included in your submission. Then **tar** and **gzip** the directory, and submit it to **Moodle**.

For example, a project group with student IDs `21404312` and `214052104` will create a directory named `21404312-214052104` and will put their files there. Then, they will tar the directory (package the directory) as follows:

```
tar cvf 21404312-214052104.tar 21404312-214052104
```

Then they will gzip (compress) the tar file as follows:

```
gzip 21404312-214052104.tar
```

In this way they will obtain a file called `21404312-214052104.tar.gz`. Then they will upload this file to **Moodle**. For a project done individually, just the ID of the student will be used as a file or directory name.

6. Tips, Clarifications, and Additional Requirements

Tips, clarifications, additional requirements about the project will be added to the following *web-page*:

`https://www.cs.bilkent.edu.tr/~korpe/courses/cs342spring2024/dokuwiki/doku.php?id=p2-s24`

It will also be linked from course's Moodle page (under project document). Please check this project web-page regularly to follow the clarifications. It may include additional requirements or specifications. Further *useful information*, *tips*, and *explanations* on this page will help you in developing the project. It will be a dynamic, living page, that may be updated (based on your questions as well) until the project deadline.

An initial project skeleton (including a `Makefile`) is added to github at the following address. You can start with this.
`https://github.com/korpeoglu/cs342spring2024-p2.git`

References

[1] x86 (i386) C Function-call Conventions.
`http://unixwiz.net/techtips/win32-callconv-asm.html`

[2] The C library getcontext and setcontext functions.
`https://man7.org/linux/man-pages/man3/getcontext.3.html`

[3] ULT package, Mike Dahlin. UT, Austin.
`https://www.cs.utexas.edu/users/dahlin/Classes/UGOS/labs/labULT/proj-ULT.html`

[4] x86 and PC architecture `https://pdos.csail.mit.edu/6.828/2004/lec/l2.html`

[5] PC Hardware and x86, Frans Kaashoek.
`https://pdos.csail.mit.edu/6.S081/2017/lec/l-x86.pdf`

[6] Stack Discipline, Dave Eckhardt. `https://www.cs.cmu.edu/~410/lectures/L02_Stack.pdf`