CS342 Operating Systems – Spring 2017
Project 2: a Thread Management Library (tlib)

version 0.3

Assigned: Mar 3, 2017
Due date: Mar 18, 2017, 23:55

Objective*: Learn the implementation details of context switching. Learn the internals of thread management and how threading system can be implemented. Learn how to keep track of an execution flow. Learn in detail how program stack is used with nested function calls. Learn the details of x86 cpu architecture and program execution.*

Important Notice*: You are not allowed to look at a solution from Internet, from other sources, from previous semesters, or at the code of your friend. It is considered cheating and will be penalized. Additionally, having the project done by somebody else will be penalized as well. You should come up with your own solution and implementation. You can use your previous solutions for standard data structures like lists.*

In this project you will write a basic *thread management library* (**tlib**) 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 the cpu to another thread of the application program.

You will do the project in a 32 bit Ubuntu Linux machine. Use Ubuntu version 16.04.1. We will prepare and post a virtual machine image with that version. You can do this project in groups of two students each. If you wish you can work individually as well. Some files (like a Makefile, tlib.c, tlib.h) will be provided to you in github so that you can start quickly.

Library Interface (API)

You will implement the following functions in your library. You are not allowed to the change the function prototypes defined here.

void tlib_init(): Initialize your library in this function (if required).

int tlib_create_thread (void (*func)(void *), void *param): Create a new thread. New thread will start its execution at the function *func*. The function will take one parameter *param* which is a pointer that can point to any data type. The calling thread (i.e., the caller of this function) will continue running after the new thread is created. The calling thread may voluntarily leave the cpu by calling the tlib_yield() function. The id of the created thread will be returned. In case a new thread could not be created, a proper error code will be returned. Check the error codes defined in tlib.h header file that is provided in github (note that you are not allowed to change the error codes defined in tlib.h as well). As part of the implementation of this function, you will create and initialize a thread control block (TCB) for the new thread. TCB will be a C structure. The new TCB will be added to the ready queue (which can be a list

of elements where each element keeps one TCB) that needs to be maintained in your library as well.

int delete_thread (int tid): Delete (i.e., terminate) the thread whose id is specified as a parameter to the function. The stack of the deleted thread will be deallocated. The thread control block (TCB) for the deleted thread will be deallocated as well. An error code indicating the result will be returned to the caller: TLIB_SUCCESS will be returned if thread is deleted successfully, otherwise TLIB_ERROR will be returned. If, as a parameter, the special value TLIB_SELF is given, the library will terminate the caller. In this case, the function will not return to the caller, obviously, and some other thread (if available) will be scheduled to run in the cpu. Note that you should be careful about when to deallocate (free) memory for the stack of a thread that was running and that your are deleting.

int tlib_yield (int tid): Yield the cpu (i.e., give the cpu) to the thread with the given id. Hence, upon calling this function, the calling thread is suspended and the specified thread is executed. The caller is added to the ready queue (ready list). Special parameter value TLIB_SELF will cause yielding to itself; in this case, your library will save the calling thread context first and then restore it. This can be used for debugging purposes. The special parameter value TLIB_ANY will cause the library to select any thread in the ready queue and run it. The head of queue can be selected or a thread can be selected randomly. The return value is the id of the thread to whom the cpu is yielded. Not that this function will not return immediately to the caller, but will return later when the calling thread (caller) is re-scheduled again. The function may also fail (may not yield to another thread), for example when the specified thread id does not correspond to a valid thread. Then it will return proper error code to the caller and caller can continue execution.

Library Implementation

You will implement the library whose interface (API) and behavior is specified above. Your library will keep one thread control block (TCB) per thread. Since any application has at least one thread, your library will have at least one TCB created and used for the default main thread.

In a TCB you can keep bookkeeping information about a thread, like its state, etc. Additionally you will keep context information (i.e., cpu context information). It is the cpu state (set of cpu register values) at the time the thread is suspended and will be used to restore the cpu when the thread will be re-executed.

There is an easy way to get the cpu context of a running thread: you can use the getcontext() library function to get and store the cpu state of a running thread. It will store the cpu state in a specified memory area. There is also setcontext() function. It will restore the cpu state of a thread from a specified context variable (structure) and will trigger the execution of that thread with that context. You will use these functions to perform thread context switch. There is also makecontext() function, which performs context switching 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 the manual pages of these two functions (man getcontext).

```
int getcontext(ucontext_t *ucp);
int setcontext(const ucontext_t *ucp)
```

When getcontext() is called in a program (in a thread), the context (cpu state) of the calling thread at that moment is stored in a structure of type *ucontext_t* pointed by the user specified pointer *ucp*. You can then read and write fields in the context structure. The ucontext_t structure is defined in header file <ucontext.h>. In your Linux system, you can find the ucontext.h file in /usr/include/i386-linux-gnu/sys.

You will use a stub function internally in your thread library. The new thread will start executing at that function. The execution then will continue at the thread start function (i.e., root function) specified by the application. In this way, the function specified by the application will have some wrapper function to return to. The stub function will be very short and will look like the following (already implemented in the provided code in github). Hence when preparing a thread control block and a context for a new tread, the instruction pointer field (EIP) in the context variable (structure) needs to be made to point to this stub function.

```
stub (void (*tstartf) (void*), void *param)
{
        // thread starts here
        int ret;
        tstartf (param); // thread start function specified by the application
        ret = tlib_delete_thread (TLIB_SELF)
        exit(0)
}
```

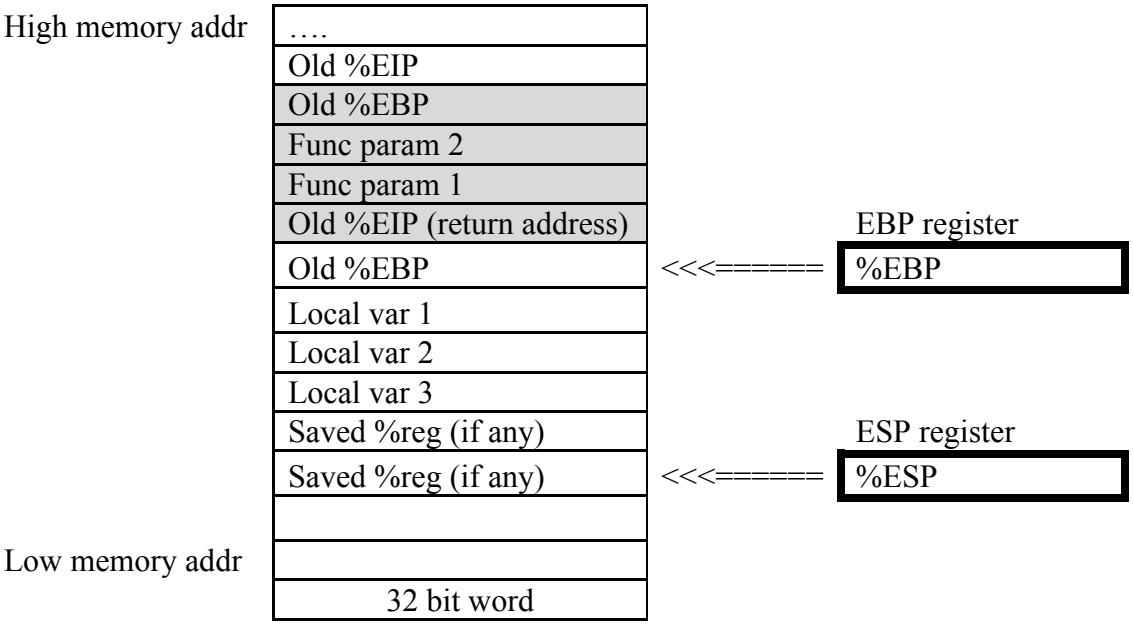
To create a new thread, you will create a TCB and a stack for the new thread. The TCB will keep state, etc., and also the context. The context will be initially copied from the context of the calling thread (creator thread). As stated above, you can obtain the context of the running (caller) thread with the getcontext() function. After obtaining an intial context, you will change just two fields in it. The rest of the conext structure can stay untouched. You will do the following as part of the setting the context for the new thread: 1) change the program counter (instruction pointer) field of context variable to point to the thread to run (i.e., to the stub function which takes the thread start function as parameter); 2) *allocate* (will malloc) and *initialize* a new stack; 3) change the stack pointer field of the context variable to point to the top of the new stack allocated. Note that stack grows downward.

Note that getcontext(), that is called by a tread, will return twice. Once when you create a context (1), and again when you switch to that context (2). Think about what action you will take in each case. Think about also how you can tell in which case you are.

Thread Context

The context structure contains many fields, but you will need to deal with only the stack pointer field and program counter (instruction pointer) field as stated above. The stack pointer field stores the value of the stack pointer register of the cpu. In Intel x86 machines, that register is called ESP and points to the top of the stack (i.e., it points to the 32-bit word valid value at the top of the stack). The program counter field stores the value of the instruction pointer register of the cpu. That register is called EIP in Intel x86 cpu architecture and points to the next instruction to be executed.

The C calling convention in Intel x86 32-bit cpu architecture causes the stack behave as the following when a procedure (function is called). Each function called has a frame (sequence of 32 bit memory words) allocated in the stack. When function returns the respective frame is deallocated by moving the stack pointer properly. The gray area in the figure below is showing the stack frame for the caller function. The part below that is the frame of the current function (last called function). The stack pointer register (ESP) points to the top of the stack (stores the address of the word at the top of the stack). Stack grows downward, towards lower addresses in x86 architecture. The EBP register (stack frame pointer) keeps the start of the stack frame for a function and it can be used to access the parameters and local variables of the function and the return address (i.e., the address to jump when the function returns). All these can be accessed relative to memory address stored in the EBP register.



When a function is called in a C program (i.e., a procedure called), the *compiler* pushes the parameters to the function from right to left. In the example above, there are 2 parameters pushed into the stack by the caller (the gray area in the stack above belongs to caller). The compiler also pushes the current instruction pointer (EIP) into the stack. This is the return address. That means, the execution will continue at this address after the callee (the called function) returns. And this makes sense, since this is the adrress of the instruction after the procedure call in the program. The procedures (functions) are called (jumped to) with the *call* machine instruction (i.e., *call function-name*).

When called and starts running, the callee (i.e., the function that is called, to which the white area below the gray area belongs in the example above) pushes (saves) the EBP register value into the stack (old EBP – caller's frame pointer) and updates the EBP register to point to the old frame pointer at the top of the stack (the *compiler* inserts such a code to do this into to the beginning of the function that is called). Hence the EBP register will point to the start of the stack frame that will be used by the called function. It allocates space in the stack for the local variables defined in the callee. The stack pointer is then advanced to point to the top of the stack. The callee can access those parameters using the current value in the EBP register. It can access local variables and also the parameters of the current function by use of the EBP.

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 - frame for caller) and uses the *ret* machine instruction to pop the old instruction pointer off the stack into the processor's instruction register (EIP), which causes the execution return to the caller function.

In this project, when a thread is created, you will make the EIP field in the context of a thread control block to point to the stub function. Hence the new thread will start executing in the stub function when thread is scheduled. That means the sub function will start executing not with a normal C function call. And stub function does not have to return somewhere, because it will be the last function thread has to execute. Hence we don't care where the EBP register was pointing at the time that stub gets executed.

Stub function takes two parameters: address of the user specified thread function (tstartf) and a parameter to that function (param). In your thread create code, you need to put these into the stack and update the top of stack pointer accordingly. Then, when stub gets executed (when thread is scheduled), it will call tstartf function. This is normal C function call. Hence the compiler would have generated the machine code to make this call. And this call (calling tstartf) will update automatically the EBP register to point to the top of the stack at that moment, and then update the top of the stack beyond the local variables. Hence it is ok if we don't update the EBP ourselves. In the figure above, you can think that the gray area is the stack frame for stub. And the white area below that is the stack frame for tstartf. It is OK that the EBP and Old %EIP value in the gray area are not correct.

Testing: You need to write some applications that will test the library. We will write apps that will be linked with your library to crate and use threads.

Submissions: You will submit through Moodle as usual using the same steps in the previous project. Not that you only need to include your README.md file, tlib.c file, tlib.h file. You don't need to include your test programs or the Makefile. We will have our own Makefile to compile and link your library with our program(s).

References:

[1]. ULT package, Mike Dahlin. A project in UT, Austin, from which this project is inspired and adapted.
https://www.cs.utexas.edu/users/dahlin/Classes/UGOS/labs/labULT/proj-ULT.html

[2]. Stack Discipline by Dave Eckhardt. Very good slides that explain how stack is used while functions are called in C.
https://www.cs.cmu.edu/~410/lectures/L02_Stack.pdf

Clarifications and additional information:
--- Clarifications may be added to the website of the course.
--- A project skeleton is put to github. You can clone and use it.
https://github.com/korpeoglu/cs342spring2017_p2.git