

# CSS 430 Operating Systems

## Program 2A: User Thread Context Switching

### 1. Purpose

This assignment implements a very simple “user thread library” and its round-robin scheduler, say `sthread` (not `pthread`). It exercises how to capture and restore the current execution environment with `setjmp( )` and `longjmp( )`, to retrieve stack and base pointers from the CPU register set with `asm( )`, and to copy the current activation record (i.e., the stack area of the current function) into a scheduler’s memory space upon a context switch to a next thread. We also use `signal( )` and `alarm( )` to guarantee a minimum time quantum to run the current thread.

### 2. User Threads

Unlike kernel threads, user threads are not identified by operating systems. In other words, a process that spawns multiple user threads but not kernel threads is considered as a single-threaded process from the OS viewpoint, yet some applications can take advantage of many user threads to simulate micro-communication among many independent entities, each instantiated as a user thread.

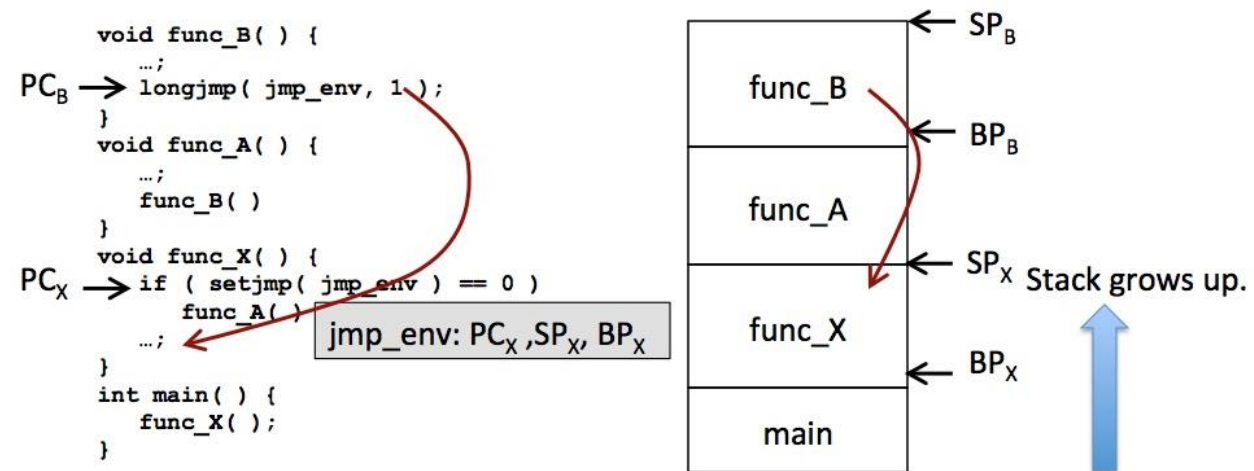
The minimum resources of each user thread are two-fold: (1) its on-going CPU register contents and (2) its stack. The former can be captured and restored as the `jmp_env` structure with `setjmp( )` and `longjmp( )`, whereas the latter needs to be identified as the current activation record that is a stack space between the current stack pointer (SP) and based pointer (BP). To manage multiple user threads, we need to allocate to each thread a thread control block (TCB) that includes its `jmp_env` and activation record.

### 3. How to Capture the Current Execution Environment

Whenever a program in execution (i.e., a process) calls a function, it creates a new activation record on top of the current stack. This activation record includes the return address to the caller function, using which the control can go back to the caller function upon a return statement.

*What if the control wants to go back to the caller function or even the caller of caller without completing the current function, thus without returning from it?* Assume that `func_X( )` called `func_A( )`; `func_A( )` called `func_B( )`; and the control wants to jump from `func_B( )` back to a certain point of `func_X( )` rather than a cascading return from B to A and A to X. To facilitate this long jump, Linux provides the following functions:

<code>int setjmp(jmp_buf env)</code>	Saves the current CPU register contents to <code>env</code> and returns 0. Later when <code>longjmp</code> is called, the control comes back to <code>setjmp</code> and returns a value given from <code>longjmp</code> .
<code>void longjmp(jmp_buf env, int val)</code>	Retrieves the CPU register contents from <code>env</code> and thus control goes back to <code>setjmp</code> as returning <code>val</code> .



If `func_X( )` is a thread scheduler, it can save its execution with `setjmp( )` just before launching a new thread that runs `func_A( )`. Without finishing `func_A( )`, this thread can relinquish its execution temporarily back to the scheduler with `longjmp( )`. Then, the scheduler can launch another thread that runs `func_B( )`.

Having launched all user threads, how can the scheduler resume the thread execution of `func_A( )` and `func_B( )` in turn? An answer is that each thread should call `setjmp(env_A)` or `setjmp(env_B)` to save its own CPU register contents including program counter (PC), SP, and BP, so that the scheduler can switch to `func_A( )` by calling `longjmp(env_A, 1)` or to `func_B( )` with `longjmp(env_B, 1)`.

However, the remaining problem is that, when the scheduler resumes the thread `func_A( )`, the latest activation record in the stack has been overwritten with `func_B( )`'s information. Therefore, we need to capture and save each thread's activation record in the heap. The current thread's activation record is the space between SP and BP.

# CSS 430 Operating Systems

## Program 2A: User Thread Context Switching

For security purposes, Linux mangles `jmp_env` contents and makes it quite hard to locate SP and BP contents in `jmp_env`. (Note that MacOS can easily locate SP and BP as `jmp_env[4]` and `[2]`.) We will use asm declarations that embed assembly language code in C++.

```
register void *sp asm ("sp");
register void *bp asm ("bp");
```

Including all these pieces of information, we define TCB as follows:

```
class TCB {
public:
    TCB( ) : sp( NULL ), stack( NULL ), size( 0 ) {}
    jmp_buf env; // the execution environment captured by setjmp( )
    void* sp;    // the stack pointer (when retrieving a thread, we don't need to use bp)
    void* stack; // the temporary space to maintain the latest stack contents
    int size;    // the size of the stack contents
};
```

To save and retrieve the current thread's activation record from or into stack, we will call `memcpy( )`:

Saving into TCB:

```
cur_tcb->size = (int)((long long int)bp - (long long int)sp);
cur_tcb->sp = sp;
memcpy( cur_tcb->stack, sp, cur_tcb->size );
```

Retrieving from TCB:

```
memcpy( cur_tcb->sp, cur_tcb->stack, cur_tcb->size );
```

### 4. Signaling

This assignment's user thread library is based on **non-preemptive scheduling** that switches to a next thread only when the current thread voluntarily relinquishes the CPU with `sthread_yield( )`. However, similar to `pthread_yield( )`, we want to guarantee a certain amount of time quantum, say 5 seconds in Prog2A, to run the current thread. If `sthread_yield( )` is called after the current 5-sec period, we switch to a next thread, otherwise simply ignore this call and return back to the current thread. Without bothering the current thread execution, we must check if 5 seconds have elapsed. For this purpose, we will use "signaling". The scheduler calls:

```
signal( SIGALRM, sig_alarm );
alarm( 5 );
```

These statements direct the operating system to receive an alarm interrupt upon 5 seconds and to call the user-specified `sig_alarm( )` as an interrupt handler. In the user-defined `sig_alarm` function, we can simply set a Boolean value `alarmed` true, so that `sthread_yield( )` switches to a next thread as resetting `alarmed`.

```
static bool alarmed = false;
static void sig_alarm( int signo ) {
    alarmed = true;
}
```

### 5. The sthread library functions

Prog2A's `sthread` library in `sthread.cpp` includes the following functions to launch a new user thread, to switch to a next thread, and to terminate the current thread.

Library functions	Actions
<code>#define scheduler_init( )</code>	Initializes the <code>sthread</code> scheduler. It must be called before launching any <code>sthreads</code> .
<code>#define scheduler_start( )</code>	Starts the <code>sthread</code> scheduler. It must be called after launching all <code>sthreads</code> .
<code>#define sthread_create( function, arguments )</code>	Launches a new thread that invokes a given function as passing arguments to it.
<code>#define capture( )</code>	Captures the current thread's <code>jmp_env</code> and activation record into <code>cur_tcb</code> . This is a helper function called from <code>sthread_init( )</code> and <code>sthread_yield( )</code> .
<code>#define sthread_init( )</code>	Is called by each user thread as soon as it starts for going back to the <code>main( )</code> program.
<code>#define sthread_yield( )</code>	Is called by each user thread to voluntarily yield the CPU. Only after a timer interrupt, calls <code>capture( )</code> and goes back to the scheduler. When the control comes back from the scheduler, retrieve this thread activation record from <code>cur_tcb-&gt;stack</code> .

# CSS 430 Operating Systems

## Program 2A: User Thread Context Switching

<code>#define pthread_exit()</code>	Is called when the current thread terminates itself. It deallocates <code>cur_tcb-&gt;stack</code> and jumps to the scheduler.
<code>static void sig_alarm( int signo )</code>	Is called upon a timer interrupt and sets <i>alarmed</i> true.
<code>static void scheduler()</code>	Is the logic of a simple round-robin thread scheduler. The scheduler maintains a list of TCBs in <code>queue&lt;TCB*&gt; thr_queue</code> and resumes the top thread of this queue every 5 seconds.

**Note that all these library functions except `sig_alarm()` and `scheduler()` must be implemented as macro declarations with `#define`.** This is because we do not want to insert any function calls and their activation records between the scheduler and user threads.

### 6. Statement of Work

You can find `sthread.cpp` and `driver.cpp` from Canvas Files: Files/code/prog2. The former is the `sthread` template file and the latter tests the library by spawning three user threads.

**Task 1:** Complete the `sthread.cpp` by implementing:

- (1) `pthread_yield()`
- (2) `capture()`

The lines of code (LoC) for this implementation would be less than 20 lines. All the other functions and `driver.cpp` have been implemented. The compilation and execution can be done through:

```
g++ driver.cpp
./a.out
scheduler: initialized
func1: Bothell 0
func1: Bothell 1
...
func3: Tacoma 9
scheduler: no more threads to schedule
```

**Taks2:** Illustrate the entire stack layers when a timer interrupt (`=sig_alarm`) was just asserted while `func3` is running its *for* loop. Of course, we assume that `driver.cpp` launched all the three threads, `func1`, `func2`, and `func3`; it started the scheduler; and the scheduler has chosen `func3` as the current thread to resume.

### 7. What to Turn in

Total 20pts.

	Materials	Points	Note
1	<code>sthread.cpp</code> <ol style="list-style-type: none"> <li>a. Code organization: <b>+2pts</b> <ul style="list-style-type: none"> <li>• Well organized: 2pts,</li> <li>• Poor comments or bad organization: 1pt, or</li> <li>• No comments and horrible code: 0pts</li> </ul> </li> <li>b. Correctness: <b>+10pts</b> <ul style="list-style-type: none"> <li>• <code>capture()</code> captures <code>sp</code>, <code>bp</code>, and <code>stack</code>: <b>+3pts</b>,</li> <li>• <code>capture()</code> pushes the current <code>tcb</code> into <code>queue</code>: <b>+1pts</b></li> <li>• <code>pthread_yield()</code> works only when <code>alarmed</code>: <b>+2pts</b>,</li> <li>• <code>pthread_yield()</code> correctly uses <code>setjmp/longjmp</code>: <b>2pts, and</b></li> <li>• <code>pthread_yield()</code> retrieves the new thread's <code>stack</code>: <b>+2pts</b></li> </ul> </li> </ol>	12	Submit this <code>cpp</code> file individually. <b>Additionally, you should also submit <code>driver.cpp</code> file individually, even if you did not change it.</b> Submitting <code>driver.cpp</code> together with <code>sthread.cpp</code> makes the grading process easier.
2	An execution snapshot in <code>jpg</code> or <code>png</code> . <ul style="list-style-type: none"> <li>• Correct RR context switch among <code>func1</code>, <code>func2</code>, and <code>func3</code> threads: 4pts,</li> <li>• Incorrect RR context switch: 3pts, or</li> <li>• No outputs: 0pts</li> </ul>	4	Include these two items in a pdf file named: <code>FirstNameLastName_prog2A.pdf</code> . Submit this pdf file individually.
	An illustration of the stack layers <ul style="list-style-type: none"> <li>• Correct: 4pts</li> <li>• Wrong: 2pts, or</li> <li>• No illustrations: 0pts</li> </ul>	4	