CS24 Assignment 6

The files for this assignment are provided in the archive **`cs24hw6.tgz`** on the course website. The archive contains a top-level directory **`cs24hw6`**, in which all other files are contained.

For the questions you need to answer, and the programs you need to write, put them into this **`cs24hw6`** directory. Then, when you have finished the assignment, you can re-archive this directory into a tarball and submit the resulting file:

*(From the directory containing **`cs24hw6`**; replace **`username`** with your username.)*
```
tar -czvf cs24hw6-username.tgz cs24hw6
```

Then, submit the resulting file through the course website.

Problem 1 – IA32 System Calls (20 points)

The files for this problem are in the **`myids`** subdirectory of the **`cs24hw6`** archive.

Write an IA32 assembly language routine to retrieve your user ID and group ID from the kernel (i.e. make the **`getuid()`** and **`getgid()`** system calls). Note that you can find the system call numbers in this file: **`/usr/include/asm/unistd.h`**

The **`getuid()`** and **`getgid()`** calls take no arguments, and they never return an error.

- Provide a file **`get_ids.s`** with an assembly routine to make the system call. The function should take two arguments: the first is a pointer to an integer to receive the user ID, and the second is a pointer to an integer to receive the group ID. The function should retrieve the user ID and store it at the first address, and then retrieve the group ID and store it at the second address. For example, your code should be able to work like this:

  ```
  int uid, gid;
  get_ids(&uid, &gid);
  printf("User ID is %d.  Group ID is %d.\n", uid, gid);
  ```

  Make sure your routine is commented so it is easy to understand.

- Provide **`myids.c`**, which calls the **`get_ids.s`** assembly function and prints the result. *(Hint: you already have some code for that function now…)*

A **`Makefile`** is provided in the archive directory.

You can verify the output of your program by running the **`id`** command and checking your answers.

Problem 2 – Simple User-Space Threads Package (80 points)

The files for this problem are in the **`sthreads`** subdirectory of the **`cs24hw6`** archive.

Complete the provided implementation of **`sthreads`**, a simple user-space thread package for Linux. Specifically, you need to implement the following components:

a) Assembly routines in the file **`glue.s`** *(25 points)*
- Performing a context switch, in the **`__sthread_schedule()`** routine.
- Initializing a new thread context, in the **`__sthread_initialize()`** routine.

b) C functions in the file **`sthread.c`** *(30 points)*
- The main thread scheduler, in the **`__sthread_scheduler()`** function.
- Creating a new thread, in the **`sthread_create()`** function.
- Deleting a finished thread, in the **`__sthread_delete()`** function.

c) Simple test code for the threading package *(25 points)*
- A **`test_arg.c`** program showing that **`sthread_create()`** correctly passes its argument to the thread-function.
- A **`test_ret.c`** program showing that returning from a thread causes the thread to be reclaimed.
- (Your work must also must pass the simple test code supplied to you.)

Very detailed discussion of this problem follows; please read it all! You are expected to follow all specifications and guidelines in this discussion.

Important Note: Writing System/Library Code

From this lab forward, you will be graded more strictly on the robustness and correctness of your code than you have been on past labs. Anytime you write code that is part of an operating system or a library, you need to make sure that your code is robust and will not crash in situations that are reasonable to expect. The remaining CS24 labs will give you an excellent opportunity to practice this.

The `sthreads` Package

The template code for Problem 2 is contained within the **`sthreads`** subdirectory of the Assignment 6 tarball. All parts of the code that you need to implement are marked with the word **`TODO`**. *Many* more details about what to do are given in this section, so read on…

Basic Operating Principles

Threads are much easier to implement than processes because all threads share the same address space, namely that of the process they are running within. (Separate processes leverage the virtual memory abstraction exposed by the kernel and hardware, and that is beyond the scope of CS24.)

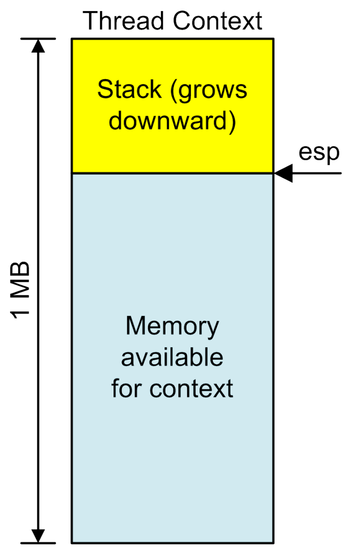
Therefore, the only context we need to preserve for each thread is the state of the thread's registers (including the flags), and the thread's stack. For threads to work correctly we must preserve *all* registers, not just the callee-save registers. However, we are only going to focus on the integer registers; you do not have to preserve the floating-point registers, the SSE/MMX registers, etc.

This week we will be implementing a *cooperative multitasking* library, which means that one thread manually *yields* execution to another thread. This makes it easier to implement context switching, because the yield operation invokes the context-switch directly.

This also means that we can save the currently running thread's execution context onto the thread's own stack (i.e. push all registers and **`eflags`** onto the stack), and then the thread's stack-pointer becomes the pointer to the thread's context. To switch to another thread, we simply save the old thread's stack-pointer somewhere, then set **`esp`** to the new thread's context, and then finally restore the thread's execution context by popping that thread's registers and flags off of that thread's stack.

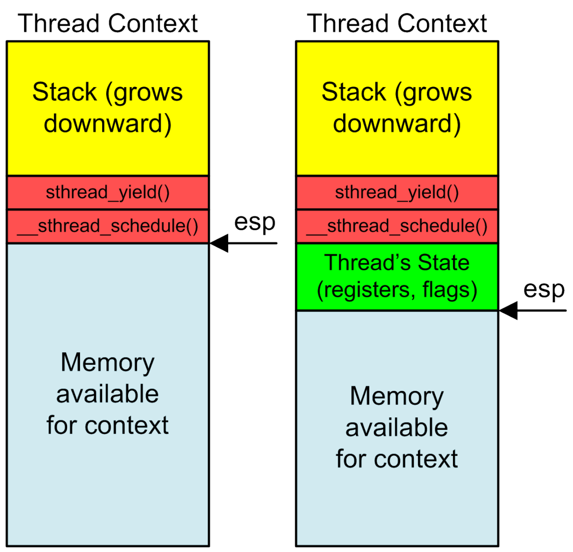
Pictures are worth a thousand words, so here are some diagrams of what is described above.

When a thread is created, we allocate space for the thread's stack. It is easy to change what stack is being used; we just set **`%esp`** to be the current thread's stack. As the thread executes, it will fill up the thread's context as the stack grows downward. We will end up with something like this (*not to scale…*):



Since this is a cooperative multitasking library, a thread must manually yield execution to another thread. This is done in several ways, either by calling **`sthread_yield()`**, **`sthread_block()`** (which blocks this thread before turning over execution to another thread), or by returning from the thread's function (which marks this thread as finished before switching to another thread). All of these functions have one thing in common: they all eventually call **`__sthread_schedule()`**, which performs the context-switch.

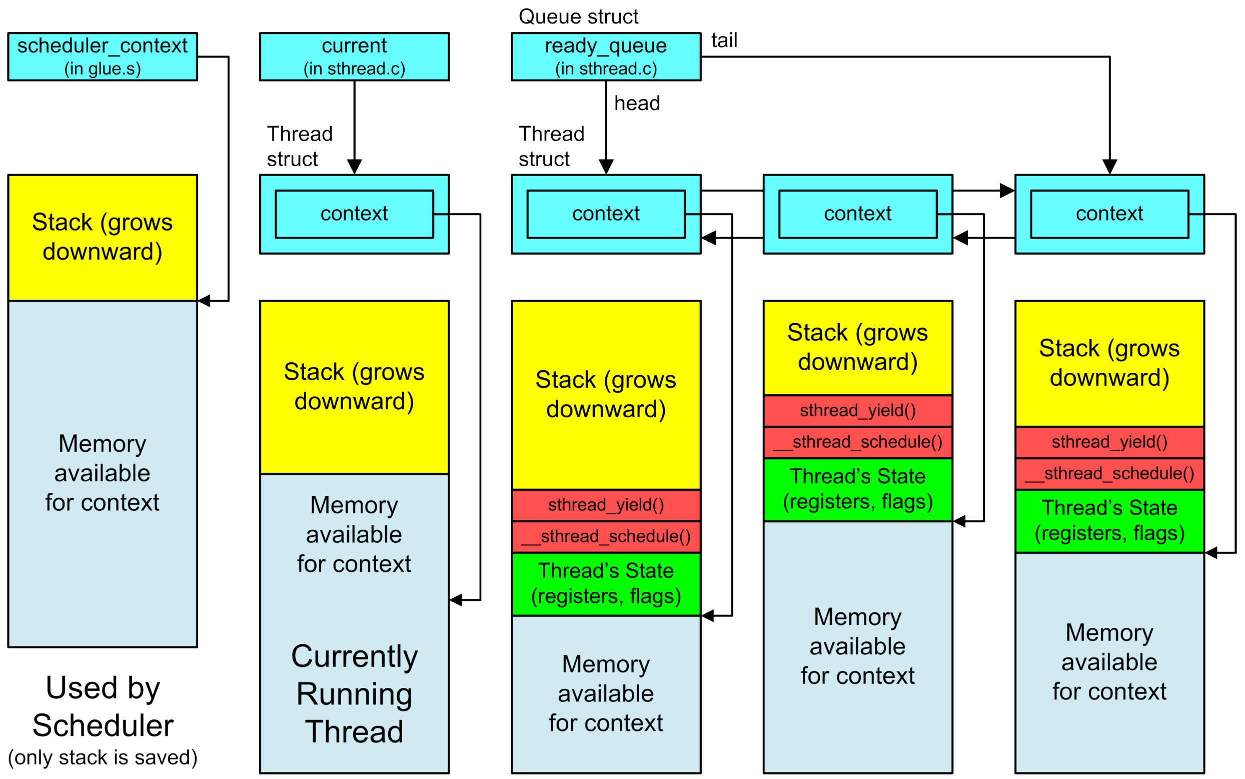
Here is what the thread's context will look like at the point that **`sthread_yield()`** has been invoked, and it in turn has invoked the scheduler. Now the scheduler must save the current thread's execution context (i.e. the general-purpose registers) and then switch to another thread's context. To simplify this implementation, we will save the machine context onto the stack as well. This will result in the thread context looking like the second picture to the right.



Notice that now **`%esp`** will point to the location to use when restoring the thread's context the next time we execute it. Therefore, we can save the current value of **`%esp`** as the current thread's context.

Restoring a thread's execution is simply the reverse of this process. The thread's context-pointer is stored back into **`%esp`**, then the thread's registers and flags are restored from the context. This allows **`__sthread_schedule()`** to **`ret`** back to **`sthread_yield()`**, which can then return back into the thread-function, exactly where **`sthread_yield()`** was invoked in the first place. Pretty awesome.

This final diagram summarizes how the major portions of the system fit together:



As described earlier, every thread of execution has its own stack area. This diagram shows four threads (one running, plus three waiting to run), as well as the separate stack area used by the scheduler (which is not really a separate thread in this implementation, just the scheduler's work-area).

When a thread is waiting to run, the **context**-pointer is the last value of **%esp** when the thread was being suspended; thus, this is the location where the machine context begins.

When a thread begins to be executed (i.e. switched from "ready" to "running"), **%esp** is set to the thread's context-pointer, and then the context is restored. This means that the **context**-pointer is no longer the location of the machine context, but this is fine because the **context**-pointer is updated the next time the thread is suspended. (This is also why each Thread struct keeps a separate **memory**-pointer, so that we actually know where the thread's memory resides. The **context**-pointer is updated every single time the thread is suspended.)

The **ready_queue** is a doubly-linked list of threads ready for execution. This week's scheduler is very simple: When a thread is suspended, it is dumped onto the end of this queue. Then, the first thread in the queue is retrieved and begins to be executed. This is a very simple round-robin scheduler.

High-level thread API

The user API (Application Programming Interface) is defined in the file **sthread.h**. The file **test.c** contains an example program using threads.

- **Thread *sthread_create(f, arg)** creates a new thread that begins by calling function **f** with the argument **arg**.
- The **void sthread_yield()** function is called by a thread to yield execution to another thread.
- The **void sthread_block()** function blocks execution of the current thread.
- The **void sthread_unblock(Thread *t)** function is used to unblock a thread that was previously blocked, and make it available for execution.
- The **void sthread_start()** function should be called once in the main routine to start the thread scheduler. This function never exits.

Architecture

The thread package has two parts. The **glue.s** file contains the code to perform a context-switch, and the C file **sthread.c** defines a thread scheduler, as well as the functions that implement the thread API.

Scheduler Interface

The main scheduler is written in C, in the file **sthread.c**. The main data structures in the scheduler are defined as follows. (See **sthread.c** for more detailed commenting.)

```
typedef enum {
    ThreadRunning,
    ThreadReady,
    ThreadBlocked,
    ThreadFinished
} ThreadState;

struct _thread {
    ThreadState state;
```

```
        void *memory;
        ThreadContext *context;
        struct _thread *prev;
        struct _thread *next;
    };

    /* sthread.h:  typedef struct _thread Thread; */
    static Thread *current;
    static Queue ready_queue;
    static Queue blocked_queue;
```

The state of the process is defined in the **`state`** field. A thread can be in one of four states:

> **ThreadRunning** – The thread is currently executing, or is about to be executing. **At any time there should be exactly one thread in the ThreadRunning state.** The value in the **`current`** variable points to this running thread.
>
> **ThreadReady** – The thread is ready for execution, but another thread is currently running. All threads in this state are saved in the **`ready_queue`**.
>
> **ThreadBlocked** – Thread has been explicitly blocked (with a call to **`sthread_block()`**) and is not ready for execution. All threads in this state are stored in the **`blocked_queue`**.
>
> **ThreadFinished** – The thread has returned from its thread-function and is therefore finished executing. Threads that return from their thread-function are automatically reclaimed by the scheduler, which then starts executing another thread.

The queues are implemented as doubly linked lists of processes. The queue code is provided, including the following three functions.

```
    /* Adds the thread to the appropriate queue */
    void queue_add(Thread *t);

    /* Fetches a process from the specified queue */
    Thread *queue_take(Queue *q);

    /* Remove the process from the queue */
    void queue_remove(Queue *q, Thread *t);
```

Note that **`queue_add()`** will store the specified thread into **`ready_queue`** if the thread's state is **`ThreadReady`**, or it will store it into **`blocked_queue`** if the thread's state is **`ThreadBlocked`**. If the thread is in neither of these states, the function will abort the program with an error message.

The other two functions require a queue to be specified. For this assignment, you will focus primarily on **`ready_queue`**; we will work with **`blocked_queue`** in the next assignment.

Scheduler Implementation

Thread scheduling is performed by the **`__sthread_scheduler(context)`** function, which takes the machine context of the current thread, and returns the context of the next thread to be executed.

For this lab, you are to implement a simple scheduler that performs the following operations:

- Properly handle the currently running thread. This is straightforward, based on the current state of the thread.

  If the current thread is in the *running* or *blocked* states: If the thread isn't in the "blocked" state (e.g. by a call to **`sthread_block()`**, which also invokes the scheduler), you can simply change the thread's state to "ready". Then, enqueue the thread using **`queue_add()`**, which will store the thread into the proper queue based on its state.

  If the current thread is in the *finished* state: call **`__sthread_delete()`** on the current thread. Do not enqueue the thread; it is done!!

- Select the next thread from the ready queue for execution. If such a thread is available, **`current`** must be updated to point to this new thread, and the thread's state must be updated to "running." Then, the new thread's context should be returned from **`__sthread_scheduler()`**, which will allow the thread's context to be restored.

  However, if there are no ready threads, the system could be in one of two states:
  - If there are also no blocked threads then it means that all threads in the program have completed successfully. Print a message to this effect, then terminate the program with a "success" status code by calling **`exit(0)`**.
  - If, however, there is one or more blocked thread in the blocked queue, the program has become deadlocked. Again, print a message to this effect, and then terminate with an "error" status code using **`exit(1)`**.

Additional Notes:

> **The very first call to the `__sthread_scheduler()` function will pass a `NULL` context!** This is because there isn't a "current thread" running yet.
>
> **The `__sthread_scheduler()` function needs to detect situations where there are one or fewer ready threads.** For example, if only one thread is running, and then it yields, the same thread should just resume running.

Creating and Deleting Threads

The other functions that you need to implement in `sthread.c` are the `sthread_create(f, arg)` and `__sthread_delete(Thread *)` functions.

Here are the steps for creating a thread:

- Heap-allocate a stack for the new thread using **`malloc()`**. Use **`DEFAULT_STACKSIZE`** for your **`malloc()`** call, defined at the top of **`sthread.c`**.

- Heap-allocate a new **`Thread`** structure for the thread, and initialize its **`state`**, **`memory`**, and **`context`** members. (Other members are updated by **`queue_add()`**.)

- Initialize the machine context by calling **`__sthread_initialize_context(stack, f, arg)`**, where the **`stack`** variable points to the *end* of the stack that was allocated. (Don't forget that stacks grow *downward* on IA32…)

- Place the new thread on the **`ready`** queue.

Thread deletion simply consists of deallocating the memory used by the thread; no cleanup beyond this is necessary.

Machine Contexts

The thread's **`context`** field points to its machine context. In this simple thread package, we consider only the integer machine context including the general-purpose registers **`%eax`** … **`%esp`**, and the **`eflags`** register. The functions to manipulate the context are defined in the file **`glue.s`**.

The assembly routine **`__sthread_schedule()`** saves the machine context, calls the C scheduler **`__sthread_scheduler()`** with the saved context, and restores the returned context.

To keep the implementation simple, a thread's machine context is stored on its stack. To save the context, push the general purpose and **`eflags`** registers on the thread's stack; the stack pointer **`%esp`** is now a pointer to the machine context. The restore operation does the opposite: pop the values of the registers, and **`ret`** to the calling function. (For access to the **`eflags`** register, you can use the **`pushfl`** and **`popfl`** instructions.)

You will also need to implement the **`__sthread_initialize_context(stack, f, arg)`** routine to save the initial machine context for a thread. Once you have written **`__sthread_schedule()`**, this function is generally straightforward, although you must construct the stack such that the **`ret`** at the end of **`__sthread_schedule()`** will begin executing the specified function **`f`** with the argument **`arg`**. In addition, **`f`** will return when the thread is finished, so the return-address that **`f`** returns to should be the **`__sthread_finish()`** function provided in **`sthread.c`**. (The address of this function can be specified as a literal value by writing **`$__sthread_finish`** in the assembly code.)

Testing

The test program in **`test.c`** is very exciting to watch the first time your threading library works, but not very useful in testing the last two features specified in the previous section. Therefore you should create two more test programs, each which informally exercises these features. The tests don't need to formally unit-test your code; just write programs that exercise these features such that when they are run, it is obvious whether the features are working correctly or not.

- **The first program you must write is `test_arg.c`.** This program should simply attempt to pass an argument to the thread-functions, to ensure that the argument is positioned properly in the thread's machine context such that the functions can retrieve and use it.

  The thread functions are defined to take a **`void *`** argument, but you can pass anything that is the size of a **`void *`**; just cast it to a **`void *`** when calling **`sthread_create()`**, and in your thread-function, cast it back to the proper type. (As gross as this is, it is a very common practice with threading APIs.)

  You can pass whatever kind of argument you would like; if you want to pass a simple integer value, it will be acceptable (albeit boring); if you want to pass a pointer to a variable that both threads can manipulate, that is even better!

- **The second program you must write is `test_ret.c`.** This program should create at least four threads, ensuring that each thread runs for a different length of time, and then terminate the thread by returning from the thread-function.

  One simple way to implement this program would be to have a single thread-function that takes an argument specifying how many times to loop before terminating. The thread-function could print some output and then yield within the loop, and when the loop is completed, the thread-function can return.

  If your threading library is working correctly, you should see your threads terminating one by one, and when all threads are completed then the program should exit.

Additional Notes:

You do not need to write a program to exercise the "blocked" functionality; you will work with that next week.