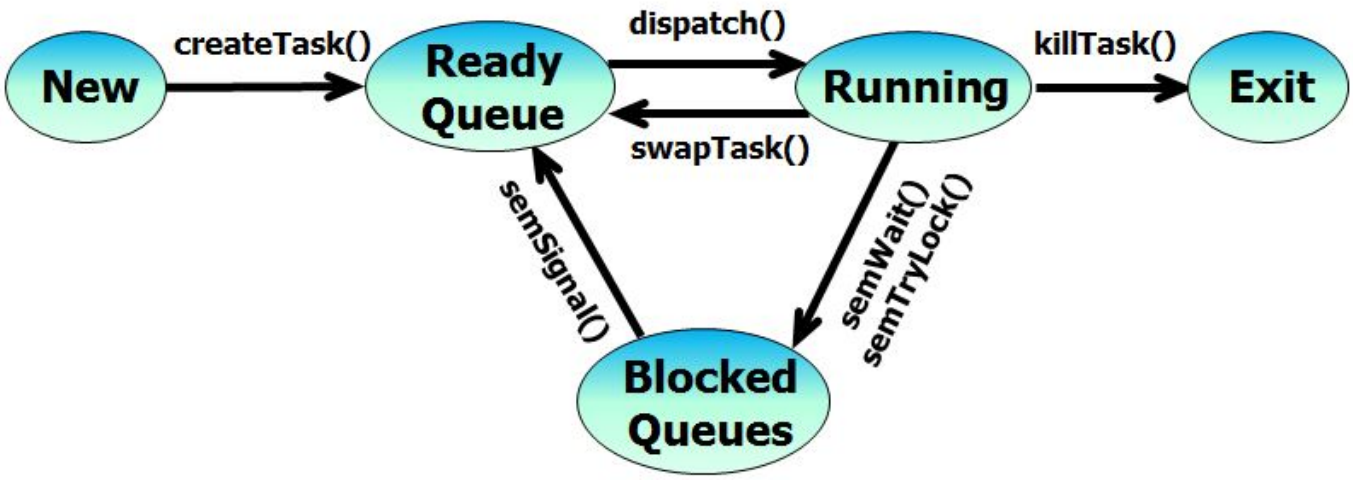
P2 - Multitasking

Purpose

Contemporary operating systems are built around the concept of processes or tasks. A task is an execution stream in the context of a particular task state. Organizing system activities around tasks has proved to be a useful way of separating out different activities into coherent units. To effectively utilize hardware resources, an operating system must interleave the execution of multiple tasks and still provide reasonable response times. These tasks may or may not be related, should not directly affect the state of another task, but usually always need to share resources in a protected, prioritized, and equitable manner.

Project Description

Add a five-state task scheduler to your operating system capable of executing up to 128 tasks in a preemptive, prioritized, round-robin manner. At any point in time, a task will be newly created, ready to run, running, blocked waiting for some event to occur, or exiting. A five-state task scheduler is illustrated as follows:



The following are important guidelines for this scheduling project:

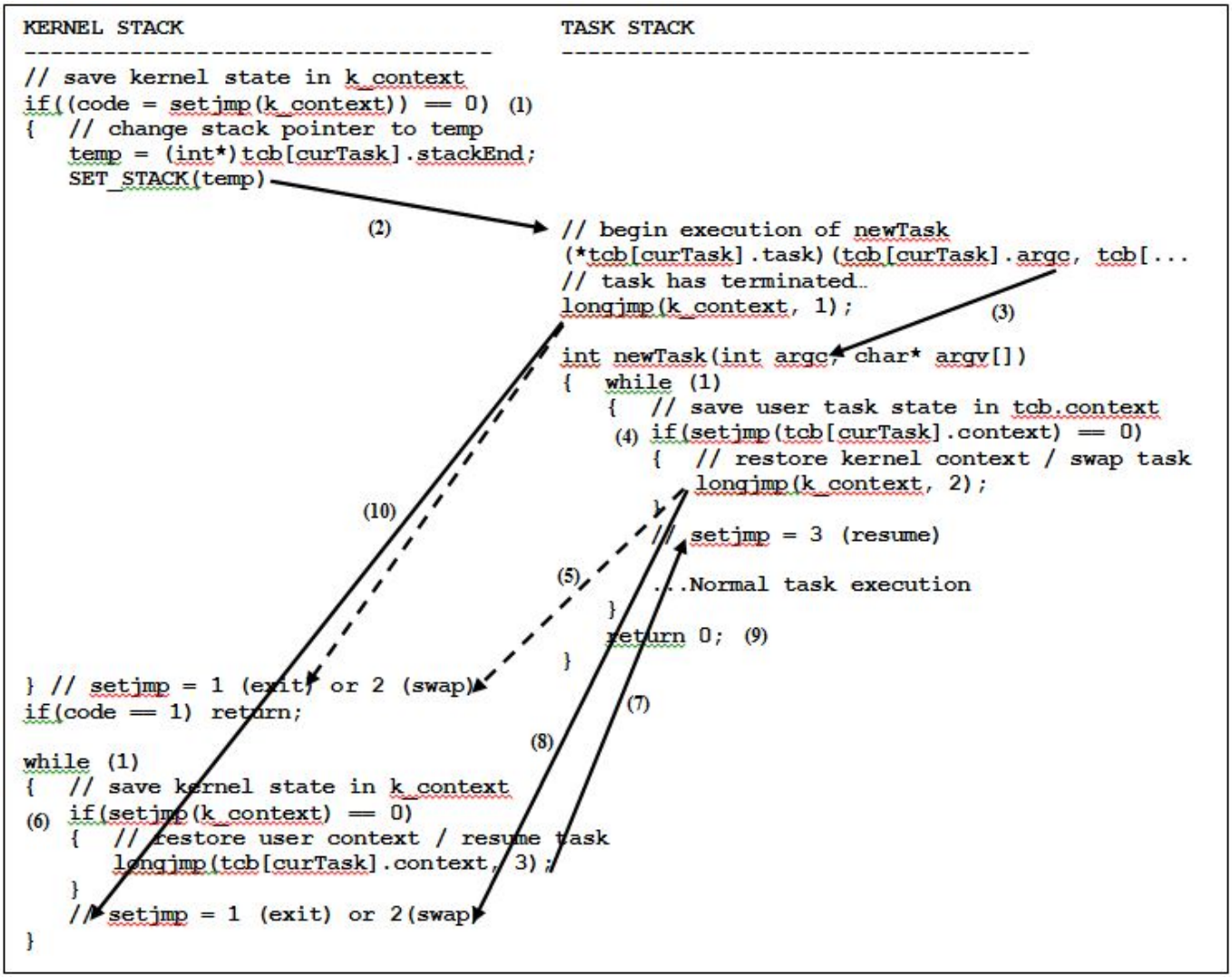
1. Your **scheduler()** function (**os345.c**) should be written in C and be called by the system scheduling loop (while(){ ... } also in **os345.c**).
2. Your **scheduler()** function (**os345.c**) should return the task id of the highest priority, unblocked, ready task from the ready queue. Tasks of the same priority should be

scheduled in a round-robin, FIFO fashion.

3. The **createTask()** function (**os345Tasks.c**) should malloc new argc/argv variables, malloc memory for the new task stack, add the new task to the ready queue, and invoke a context switch.
4. Counting semaphores and blocked queues need to be added to the **semSignal()**, **semWait()**, **semTryLock()** functions (**os345semaphores.c**) and work in conjunction with the scheduler ready queue.
5. The **SWAP** directive should be liberally inserted throughout your tasks. Context switching directives (**SWAP**, **SEM_SIGNAL**) may occur anywhere in a task. Any change of events (**SEM_SIGNAL**) should cause a context switch.
6. Timers are polled during the scheduling loop in the pollInterrupts() function (**os345interrupts.c**).

Multi-tasking in C

A context switch between tasks involves interrupting a task, saving the task state (CPU registers, status, stack values), and then restoring the state of the next scheduled task exactly as it was just before it was interrupted. The swap process requires that each task have its own stack for local variables and function arguments as well as a system or kernel stack (used as a foothold and for privileged functions). Stack state is captured with the C **setjmp** function and restored with the C **longjmp** function. Our operating system will be cooperative preemptive for context switching, hence you must place **SWAP** directives liberally throughout your tasks to force a context switch. The **SET_STACK** directive inserts an assembly language instruction that switches the stack pointer from the kernel stack to a new user stack. This is only done once when a new task is created. The flow of a new task is illustrated below:



(1) The state of the OS kernel is saved in **k_context**. (2) A new stack is created and execution continues off the new task stack. (3) A task (function) begins execution. (4) The task state is saved in the TCB struct **context**. (5) The task does a context switch to the kernel stack with code=2. (6) A new kernel context is again saved in **k_context**. (7) The task is "rescheduled" by restoring the state saved in the task TCB. (8) Subsequent context switches return directly into the scheduling loop. (9) A task terminates by returning from the function. (10) The task either returns just before the scheduling loop (no swap occurred) or within the scheduling loop again. This has all been implemented for you in the **swapTask** and **dispatch** routines.

Context Switching

The before mentioned context switching functionality is accomplished by the **swapTask()** function (**os345.c**). This function is called by the **SWAP** directive and may be called from anywhere in your system when not in the kernel state (for obvious reasons). **SWAP** instructions should be placed liberally throughout your code to simulate preemptive scheduling.

In order to thoroughly test your mutual exclusion logic, TA's are at liberty to ask you add a **SWAP** command anywhere in your code during pass-off. Make sure critical sections of your code are protected by mutex semaphores.

```
#define SWAP swapTask();

// ***************************************************************************
// Do a context switch to next task.
// Save the state of the current task and return to the kernel.
void swapTask()
{
    // either context switch or continue
    if(setjmp(tcb[curTask].context)) return;
    // swap task
    if(tcb[curTask].state == S_RUNNING) tcb[curTask].state = S_READY;
    longjmp(k_context, 2);
} // end swapTask
```

Task Scheduling

Three system functions are called from within the operating system scheduling loop. The **pollInterrupts()** function (**os345.c**) checks for simulated keyboard and timer "interrupts" and signals the corresponding signals/semaphores. Other asynchronous event handlers may be added later to this function. Any task waiting on a signaled semaphore should be moved to the ready state.

The **scheduler()** function (**os345.c**) returns the task index (**curTask**) of the highest priority ready task. Tasks of the same priority should be scheduled in a round-robin, FIFO fashion. If there are no tasks ready for execution (all are suspended), then a -1 is returned and no task is scheduled.

A task is executed or rescheduled from the **dispatcher()** function (**os345.c**).

Task Format

A task is a C function and has its own stack. Consequently, it is re-entrant and variables retain their values throughout the scope and/or life of the function. Generally, tasks are repeated over and over as external events occur. To accomplish this, one might surround the task code with a **while(1)** and put a **SEM_WAIT** at the top of the loop. (The task must periodically give up control for other tasks to execute.)

The question is asked, "If the highest priority task is always executing, won't there be tasks that never execute?" The answer is "Yes!" Starvation can occur and needs to be handled by proper usage of semaphores as well as judicious placement of **SWAP** commands.

A task (function) terminates when it returns to the operating system.

```
// ********************************************************************************
// signal task
int signalTask(int argc, char* argv[])
{
    int count = 0;              // task variable
    while(1)
    {
        SEM_WAIT(semSignal);    // wait for signal
        printf("\n%s  Task[%d], count=%d", taskName, curTask, ++count);
    }
    return 0;                   // terminal task
} // end signalTask
```

Task Creation

A task is scheduled for execution by the **createTask()** function (**os345tasks.c**). There are five arguments for the createTask function: a task name, task address, priority, argc, and argv arguments. The task priority ranges from 1 to **SHRT_MAX** (+32767 – found in **limits.h**), low to high priority. The **createTask()** function makes copies of argc and argv arguments in new malloc'd variables which are passed to the task when it is first called.

```
// create task signal1
createTask("signal1",           // task name
           signalTask,          // task
           VERY_HIGH_PRIORITY,  // task priority
           argc,                // task argc
           argv);               // task argv
```

Grading and Pass-off

Your scheduler is to be demonstrated in person the professor. The assignment is worth 10 points, which will be awarded as follows:

Points	*Requirement*

Points	*Requirement*
3	Replace the simplistic 2-state scheduler with a 5-state, preemptive, prioritized, round-robin scheduler using ready and blocked task queues. (Be sure to handle the **SIGSTOP** signal.)
2	Implement counting semaphores within the semSignal, semWait, and semTryLock functions. Add blocked queues to your semSignal and semWait semaphore functions. Validate that the **SEM_SIGNAL** / **SEM_WAIT** / **SEM_TRYLOCK** binary and counting semaphore functions work properly with your scheduler. (Note: **SEM_TRYLOCK** will be tested in Project 3.)
2	Modify the **createTask()** function (os345tasks.c) to insert the new task into a prioritized, round-robin ready queue. Modify if necessary the **killTask()** and **sysKillTask()** functions such that individual tasks can be terminated and resources recovered. (This would include all semaphores created by the killed task.) Modify the list tasks command to display all tasks in the ready and blocked queues in execution/priority order indicating the task name, if the task is ready, paused, executing, or blocked, and the task priority. If the task is blocked, list the reason for the block.
2	Add a 10 second timer (**tics10sec**) **counting** semaphore to the **timer_isr()** function (**os345interrupts.c**). Include the <time.h> header and call the C function **time(time_t *timer)**. **SEM_SIGNAL** the **tics10sec** semaphore every 10 seconds. Create a reentrant, high priority task that blocks (**SEM_WAIT**) on the 10 second timer semaphore (**tics10sec**). When activated, output a message with the current task number and time and then block again.
1	Upon entering main, your shell (CLI) should be scheduled as task 0. Have the **project2** command schedule timer tasks 1 through 9 and observe that they are functioning correctly. The shell task blocks (**SEM_WAIT**) on the binary semaphore **inBufferReady**, while the "TenSeconds" tasks block on the counting semaphore **tics10sec**. The "ImAlive" tasks do not block but rather immediately swap (context switches) after incrementing their local counters. The high priority "Signal" tasks should respond immediately when semaphore signaled.
-1/2	For each school day late.

In addition, **after completing the above requirements**, the following bonus points may be awarded:

Points	*Requirement*

Points	*Requirement*
1	Implement buffered pseudo-interrupt driven character output and demonstrate that it works by using a *my_printf* function in place of *printf* (for Project 2 only.) Use a consumer / producer model with *my_printf* putting printable characters in an output buffer (producer) and pollInterrupts getting and printing the buffered characters one at a time (consumer).
1	Implement time slices that adjust task execution times when scheduled. Modify the **createTask()** function (**os345tasks.c**) to include the time slice when the task is created. (Each call to the **swapTask()** function (**os345.c**) constitutes one time slice.