P1 - Shell

Purpose

A shell (also called a Command Language Interpreter) is an interface to an Operating System. A shell is a software module that interprets textual commands coming either from the user's keyboard or from a script file and executes the commands either directly or creates a new child process to execute the command. Signal processing is generally at the shell level. Usually, a shell will provide command history and interactive name completion mechanisms.

Project Description

For Project 1, write a shell task for your operating system that supports background tasks and asynchronous signal handlers. Your shell should prompt the user for keyboard input. After a sequence of characters is entered followed by a return, the string is parsed and the indicated command executed, either directly or as a background task.

Follow these steps in completing your Project 1 Shell:

1. Create a new project in your development environment. Download the os345 folder from the link on the course schedule
2. Edit os345config.h (if necessary) to select host OS/IDE/ISA. (Only enable one of the following defines: DOS, GCC, MAC, or NET.) Compile and execute your OS.
3. Modify the function P1_main (os345p1.c) to parse the commands and parameters from the keyboard **inbuffer** string into traditional **argc** and **malloc**'d **argv** C variables. Your shell executes the command directly with a function pointer call, waits for the function to return and then recovers memory (free) before prompting for the next command. Commands and arguments are case insensitive. Quoted strings are treated as one argument and case is preserved within the string.
4. Before a task is scheduled in the function *dispatcher* (os345.c), the function *signals* (os345signals.c) is called. Modify the function *signals* (os345signals.c) to call all pending task signal handlers. Modify the function *createTaskSigHandlers*(os345signals.c) to have a child task inherit all its parent signal handlers. Modify the function *sigAction* (os345signals.c) to register new task signal handlers. Add default signal handlers as needed. Implement all signals and signal handlers as explained below.

Miscellaneous

1. To simulate character interrupts, keyboard characters are read in the function *keyboard_isr* (os345interrupts.c) during the scheduling loop. Add code to handle backspace and other signal functionality.
2. The **SWAP** macro should be liberally placed throughout your user code to ensure timely execution of your polling and scheduling routines - DO NOT put **SWAP**'s in system code.
3. Command parameters are strings, quoted strings ("This is one argument"), decimal numbers, or hexadecimal numbers (ie. 0xa19f).
4. Other than the few required commands, you are to decide the look and feel (syntax and semantics) as well as which commands to implement. You may define command and argument delimiters, but you must address case sensitivity issues. (Is "LS" the same as "ls" or "Ls"?) Commands may be terse and/or verbose (ie, ls and list).
5. Make your shell scalable as you will be adding additional functionality in subsequent projects.

Signals

A signal is a limited form of inter-process communication used in Unix, Unix-like, and other POSIX-compliant operating systems. Essentially, a signal is an asynchronous notification of an event that is sent to a process before it is rescheduled for execution.

When a signal is sent to a process, the operating system interrupts the process' normal flow of execution (when it is scheduled) and executes a signal handler. If the process has previously registered a signal handler, that routine is executed. Otherwise the default signal handler is executed. Execution can be interrupted during any non-atomic instruction.

Blocked processes are not un-blocked by a signal, but rather the signal remains pending until such time as the process is un-blocked and scheduled for execution.

The following table is summarizes how signals might be handled by your operating system:

Signal	*PollInterrupts*	*Scheduler*	*Dispatcher*	*Signal Handler*
SIGCONT	Cntrl-R sigSignal(-1, **SIGCONT**); (Clear SIGSTOPand SIGTSTP in all tasks)	n.a.	Clear **SIGCONT** Call sigContHandler() Schedule task	return;

Signal	*PollInterrupts*	*Scheduler*	*Dispatcher*	*Signal Handler*

SIGINT	Cntrl-X
	sigSignal(0, **SIGINT**);
	semSignal(inBufferReady);
		n.a.	Clear **SIGINT**
			Call sigIntHandler()
			Schedule task
				sigSignal(-1, **SIGTERM**);

SIGTERM	n.a.	n.a.	Clear **SIGTERM** Call sigTermHandler() Do not schedule task	killTask(curTask); (Note: killTask does not terminate shell)
SIGTSTP	Cntrl-W sigSignal(-1, **SIGTSTP**);	n.a.	Clear **SIGTSTP** Call sigTstpHandler() Do not schedule task	sigSignal(-1, **SIGSTOP**);
SIGSTOP	n.a.	TASK NOT SCHEDULED	n.a.	No handler needed

Handling Control Signals

Certain combinations of characters entered at the controlling terminal of a running process cause the system to signal the process. Signals are processed just before a process is scheduled for execution. For project 1, the following control character signals are to be handled:

- Cntrl-X sends an INT signal (**SIGINT**) to the shell process (task 0), clears the input buffer, and semSignal's the input buffer ready (**inBufferReady**); the shell SIGINT handler should send a TERMINATE signal (**SIGTERM**) to all tasks.
- Cntrl-R sends a CONTINUE signal (**SIGCONT**) to all tasks; clears **SIGSTOP** and **SIGTSTP** signals from all tasks.
- Cntrl-W sends a STOP signal (**SIGTSTP**) to all tasks.

Signal handlers can be installed with the **sigAction()** system call. If a signal handler is not installed for a particular signal, the default handler is used.

Requirements

Your Shell is to be demonstrated in person to the professor as follows:

1. Your Shell task should parse the command line arguments into traditional **argc** and **argv** C variables. Use **malloc** to allocate space for **argv** and the string arguments. Recover the allocated space with the **free** function when a task/function terminates/returns. **argc** is the number of command line arguments. **argv** is a pointer to an array of pointers that point to the character string arguments. **argv[0]** is the first parameter (the command), so **argc** will always be at least 1.

2. Implement the following shell commands:

    - **Add** - add all numbers in command line (decimal or hexadecimal).
    - **Args** - list all parameters on the command line, numbers or strings.

3. Implement background execution of programs. If the command line ends with an ampersand (&), your shell should create a new task to execute the command line. Otherwise, your shell should directly call the command function (and wait for the function to return.) Use the **createTask** function to create a background process.

4. Add **signal** functionality to your shell such that:

    - Cntrl-X terminates (kills) all tasks except task 0 (shell).
    - Cntrl-W pauses the execution of all tasks.
    - Cntrl-R continues the execution of all tasks after a pause.

5. Additional functionality may be added to your Shell for bonus credit such as:

    - Help implemented using the *more* filter.
    - Quoted strings are treated as one argument and case is preserved within the string.
    - Be able to edit command line (insert / delete characters). *NOTE: delete deletes the character at (to the right of) the cursor, backspace deletes the character to the left (on mac the delete(←) key does the backspace functionality not delete, you need to press fn + delete to get delete functionality)
    - Chain together multiple commands separated by some delimiter.
    - Wild cards.
    - List / Set command line variables.
    - Implement aliasing.
    - Calculator - perform basic binary operations.
    - Allow for other number bases such as binary (%) and octal (0).
    - Command line recall and editing of one or more previous commands. (Additional Bonus)

Grading

There are 10 points possible for Project 1. The grading criteria will be as follows:

Points	*Requirement*
2	Your shell parses the command line into **argc** and **malloc**'d **argv** argument variables. The function **createTask()** also **malloc**'s and copies **argv** argument variables which are passed to a new task. All **malloc**'d memory is appropriately recovered - shell **free**'s its own and memory **malloc**'d in createTask) is **free**'d by **sysKillTask()**. (Don't create any memory leaks! Don't worry at this time about any pre-existing system memory leaks such as with the **reset** command.)
2	Commands and arguments are case insensitive. Backspace is implemented (delete character to the left) and correctly handles input buffer under/overflow.
3	The signals **SIGCONT**, **SIGINT**, **SIGTSTP**, **SIGTERM**, and **SIGSTOP** function properly as described above.
1	The required shell commands **add** and **args** are correctly implemented. (The add command handles hexadecimal as well as decimal arguments.)
2	Your shell supports background execution of all commands.
-1/2	For each school day late.

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

Points	*Requirement*
1	For each additional shell function of your choice implemented. (See above item 5. Excludes recall.)
2	Implement command line recall/edit.