



SOFE 3950U / CSCI 3020U: Operating Systems

TUTORIAL #8: Signals and Data Structures Part II

Objectives

- Learn the fundamentals of signals and data structures in C
- Gain experience writing multiprocessor code and data structures

Important Notes

- Work in groups of **four** students
 - All reports must be submitted as a PDF on Canvas, if source code is included submit everything as an archive (e.g. zip, tar.gz)
- Save the file as <tutorial_number>_<first student's id>.pdf (e.g. tutorial8_100123456.pdf)

Notice

It is recommended for this lab activity and others that you save/bookmark the following resources as they are very useful for C programming.

- <http://en.cppreference.com/w/c>
- <http://www.cplusplus.com/reference/clibrary/>
- <http://users.ece.utexas.edu/~adnan/c-refcard.pdf>
- <http://gribblelab.org/CBootcamp>

The following resources are helpful as you will need to use signals and data structures to complete the task.

- http://www.gnu.org/software/libc/manual/html_node/Standard-Signals.html#Standard-Signals
- http://www.gnu.org/software/libc/manual/html_node/Signaling-Another-Process.html
- http://www.gnu.org/software/libc/manual/html_node/Process-Completion.html#Process-Completion
- http://www.gnu.org/software/libc/manual/html_node/Signaling-Another-Process.html
- <http://www.thegeekstuff.com/2013/02/c-binary-tree/>
- http://www.learn-c.org/en/Binary_trees

Conceptual Questions

1. What is an Abstract Data Type (ADT)?
2. Explain the difference between a queue (FIFO) and a stack (LIFO).
3. Name and briefly explain three types of data structures.
4. Explain what a binary tree is, what are some common operations of a binary tree?
5. Explain what a hash table (dictionary) is, what are common operations of a hash table?

Application Questions

All of your programs for this activity can be completed using the template provided, where you fill in the remaining content. A makefile is not necessary, to compile your programs use the following command in the terminal. **If you do not have clang then replace clang with gcc, if you are still having issues please use -std=gnu99 instead of c99.**

```
clang -Wall -Wextra -std=c99 <program name>.c -o <program name>
```

Example:

```
clang -Wall -Wextra -std=c99 question1.c -o question1
```

You can then execute and test your program by running it with the following command.

```
./<program name>
```

Example:

```
./question1
```

Template

```
#include <stddef.h>
#include <stdlib.h>
#include <stdio.h>
#include <stdbool.h>
#include <unistd.h>
#include <signal.h>
#include <sys/types.h>
#include <sys/wait.h>
```

```
int main(void)
{ ... }
```

Notice

You must have the program **process** in the same location as your source code, compile the included source file **sigtrap.c** as process using the following commands. You can use **clang** or **gcc**, you will need to use **-std=gnu99** in order for the code to compile properly.

```
clang -Wall -Wextra -std=gnu99 sigtrap.c -o process
```

1. Create a program that does the following.
 - Create a structure called **proc** that contains the following
 - **parent**, character array of 256, name of the parent process
 - **name**, character array of 256 length
 - **priority**, integer for the process priority
 - **memory**, integer for the memory in MB used by process
 - Create a binary tree data structure called **proc_tree** which contains the proc data structure.
 - Create the necessary functions to interact with your binary tree data structure, you will need to add items to your tree and iterate through it.
 - Your program then reads the contents of a file called **process_tree.txt (7 LINES)**, which contains a **comma separated** list of the parent, name, priority, and memory.
 - Read the contents of the file and create your binary tree, add the children to the parent based on the name of the parent.
 - Print the contents of your binary tree (you likely need to use **recursion!**) displaying the contents of each parent, and the children of each parent.
2. Create a simple host dispatch shell that does the following.
 - Create a structure called **proc** that contains the following
 - **name**, character array of 256 length
 - **priority**, integer for the process priority
 - **pid**, integer for the process id
 - **address** integer index of memory in **avail_mem** allocated
 - **memory**, integer for the memory required
 - **runtime**, integer for the running time in seconds
 - **suspended**, boolean indicating process has been suspended
 - Create a **FIFO** queue called **priority** which will be populated with real time priority processes (priority 0).
 - Create a second **FIFO** queue called **secondary**, which will be populated with secondary priority processes.

- Create an array of **length 1024** called **avail_mem**, use **#define MEMORY 1024**, **initialize it to 0** to indicate all memory is free.
- Read in the processes from the file **processes_q2.txt**, the file contains a comma separated list of the **name**, **priority**, **memory**, and **runtime** you must initialize the **pid** and **address** to **0** in your process structure, it is set when you execute the processes.
- When reading the file **processes_q2.txt** add each process with a priority of 0 to the **priority** queue, add the remaining processes to the **secondary** queue.
- Iterate through all of the processes in the **priority** queue first, **pop()** each item from the queue and execute the **process** binary using fork and exec.
 - Mark the memory needed in the **avail_mem** array as used (**1**), set the **address** member of the struct to the starting index where the memory is allocated in the **avail_mem** array.
 - Before **process** is executed, print the **name**, **priority**, **pid**, **memory**, and **runtime** of the process.
 - Run the process for the specified **runtime** and then send it the signal **SIGTSTP** to terminate it.
 - Ensure that you use the **waitpid** function to wait until the process has terminated.
 - Free the memory in **avail_mem** used by the process (**set the array entries to 0**).
- Then iterate through all of the processes in the **secondary** queue, **pop()** each item from the queue and execute the **process** binary using fork and exec.
 - If there is **enough memory** available in **avail_mem** array then proceed, otherwise **push()** it back on the queue.
 - Mark the memory needed in the **avail_mem** array as used (**1**), set the **address** member of the struct to the starting index where the memory is allocated in the **avail_mem** array.
 - Before **process** is executed, print the **name**, **priority**, **pid**, **memory**, and **runtime** of the process.
 - **If** the process has already been suspended (**suspended** is true, and **pid** set) then send **SIGCONT** to the process to resume it.
 - Run the process for **1 second** then send **SIGTSTP** to the process to suspend it.
 - If the process was just created, set the **pid** member in the process struct that was returned from **pop()** to the process id returned from **exec()**.
 - Decrement the **runtime** member in the process struct by **1**.

- Set the **suspended** member in the process struct to **true**, indicating the processes has been suspended.
- Add the process back to the **secondary** queue using **push()**
- Repeat this for every process in the **secondary queue**.
- For any item in the **secondary** queue that **only has 1 second** of **runtime** left
 - Run the process for the specified **runtime** and then send it the signal **SIGINT** to terminate it.
 - Ensure that you use the **waitpid** function to wait until the process has terminated.
 - **Do not** add the process back to the queue.
 - Free the memory in **avail_mem** used by the process (**set the array entries to 0**).
- Once all of the processes have been executed the main program terminates.