Ivy Truong
CSCI 3800
04/21/2022
Homework 5

1. a) Yes

   b) For signal handlers to be reentrant (async signal safe), the functions called must not use static data structures, do not call malloc or free, and the functions must not be part of the standard I/O library (which involves modifying global resources). The main reason that these conditions must be satisfied is because there are some functions that involve rewriting and accessing the same memory location that can be rewritten by the information returned to the signal handler, since the signal can be caught at any time and can produce unpredictable results if using non reentrant functions. Any variable that is stored in memory that can be accessed by multiple other processes have the potential to be overwritten or corrupted. For example, anything that uses static or global data structures can be overwritten by any process because it is visible to the processes that call the function, while calling malloc or free uses heap memory to store the variables which can be overwritten by the stack data.

   c) One specific condition that makes strtok() functions non reentrant is that it uses a static data structure to save the new modified string after splitting the previous string by the delimiter. This makes the function non reentrant because static data structures are always present even after the function called using the static data structure is returned to the main program. Static data structures are not unique to the program that called the function, so any function can modify the value and overwrite any previous value on accident, causing unpredictable results. In this specific case, if strtok() is called before the signal handler and then during the signal handler, the static variable modified during the signal handler will override any progress made when strtok() was called before the signal handler executed.
   Link to view strtok() function source file: https://elixir.bootlin.com/glibc/glibc-2.17.90/source/string/strtok.c

   d) The function strtok_r() resolved the reentrancy issue in the strtok() by adding another parameter, char **save_ptr, that saves the new string modified. By adding another parameter to the function, it makes is so that each call of strtok_r() will have to save their own variable of the new modified string within the program that called the function instead of saving a 'global' static variable within the strtok() function itself. This allows multiple programs to call the strtok_r() function without having to worry about another program overwriting the last modified string and allows multiple programs that have to use this function for different unique strings to execute at the same time.
   Link to view strtok_r() function source file: https://elixir.bootlin.com/glibc/glibc-2.24/source/string/strtok_r.c

2. a) Screenshot of commands executed

```
[truongiv@csci-gnode-02 ~]$ sleep 1000 &
[1] 13253
[truongiv@csci-gnode-02 ~]$ sleep 1100 &
[2] 13254
[truongiv@csci-gnode-02 ~]$ sleep 1200 &
[3] 13255
[truongiv@csci-gnode-02 ~]$
```

```
[truongiv@csci-gnode-02 ~]$ jobs
[1]   Running                 sleep 1000 &
[2]-  Running                 sleep 1100 &
[3]+  Running                 sleep 1200 &
[truongiv@csci-gnode-02 ~]$ fg
sleep 1200
^Z
[3]+  Stopped                 sleep 1200
[truongiv@csci-gnode-02 ~]$ jobs
[1]   Running                 sleep 1000 &
[2]-  Running                 sleep 1100 &
[3]+  Stopped                 sleep 1200
[truongiv@csci-gnode-02 ~]$ fg %2
sleep 1100
^Z
[2]+  Stopped                 sleep 1100
[truongiv@csci-gnode-02 ~]$ jobs
[1]   Running                 sleep 1000 &
[2]+  Stopped                 sleep 1100
[3]-  Stopped                 sleep 1200
[truongiv@csci-gnode-02 ~]$ bg
[2]+ sleep 1100 &
[truongiv@csci-gnode-02 ~]$ bg %3
[3]+ sleep 1200 &
[truongiv@csci-gnode-02 ~]$
```

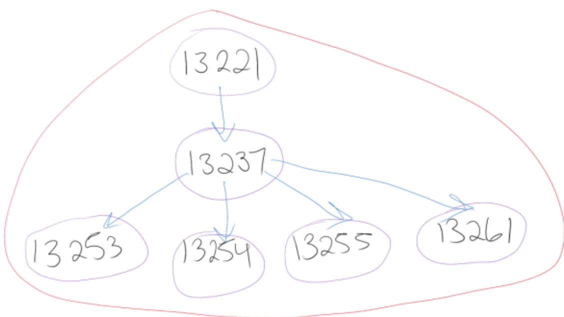
```
[truongiv@csci-gnode-02 ~]$ jobs
[1]   Running                 sleep 1000 &
[2]-  Running                 sleep 1100 &
[3]+  Running                 sleep 1200 &
[truongiv@csci-gnode-02 ~]$ ps -o pid,ppid,pgid,sid,comm
  PID  PPID  PGID   SID COMMAND
13221 13220 13221 13221 tcsh
13237 13221 13237 13221 bash
13253 13237 13253 13221 sleep
13254 13237 13254 13221 sleep
13255 13237 13255 13221 sleep
13261 13237 13261 13221 ps
[truongiv@csci-gnode-02 ~]$
```

b) Command to bring job [1] to foreground: fg %1

Command to terminate job [2] running in background: kill %2

```
[truongiv@csci-gnode-02 ~]$ fg %1
sleep 1000
^Z
[1]+  Stopped                 sleep 1000
[truongiv@csci-gnode-02 ~]$ bg %1
[1]+ sleep 1000 &
[truongiv@csci-gnode-02 ~]$ kill %2
[truongiv@csci-gnode-02 ~]$ jobs
[1]   Running                 sleep 1000 &
[2]-  Terminated              sleep 1100
[3]+  Running                 sleep 1200 &
[truongiv@csci-gnode-02 ~]$
```

c) Diagram with Key showing Relationship between processes below:



- PID
- PPID (arrow points to child process)
- PGID
- SID

3. a) Yes

   b) Source code tell_wait.c included in submission to answer this question.

4. a) A thread is a piece of a program that runs separately from the rest of the program, this is usually called by the process to create it, and multiple threads can exist in a process. Threads do not have a data segment or heap but contains its own stack, and shares resources with the process that called it. Lastly, threads must live within a process for it to exist. A process is a heavy weight resource that executes the whole program. A process has many resources included after creation such as code, data, and heap segments. Lastly, when a process is created, there must be at least one thread in it, which runs the program. One of the main differences between them is that threads share its memory and resources of the process that called it (lightweight), while processes have their own memory and resources allocated (heavyweight). Another difference is that when a thread terminates, its stack is reclaimed, while when a process terminates, all its resources are claimed, and threads terminate.

b)

Threads		Processes	
Pros	Cons	Pros	Cons
Can run portions of code concurrently by partitioning the code to run codes of the same type (simpler software design)	Shared state: resources are shared among multiple threads and global variables can be modified by any thread	Simple: separate workspace for each process	High overhead, expensive to manage
Can overlap I/O with computation	Many library functions are not thread safe	Reliable	When created it takes a lot of memory space
Can handle asynchronous operations	Crash in one thread may crash the entire process	If a process is blocked, the other processes continue their execution	Expensive context switching
Faster performance since threads share resources from the process that called them	If a user level thread is blocked, all other threads are blocked		Complicated synchronization: difficult to interprocess communications to share memory and file descriptors among multiple processes
IPC cooperation: shared address space incurs less overhead than IPC			Takes a lot of time to create and terminate a process
Faster to create threads and context switch since they share resources from the process that called them			

-Notes from lecture slides on canvas, Module 21 Threads Primer.

-Outside source: https://www.tutorialspoint.com/difference-between-process-and-thread