Systems Software

Week 3: Processes, File IO and gProf

Notes By: Jonathan McCarthy

Overview

- Basics of Processes
- System Calls
- Exec and Fork
- Signals and Interrupts

Processes

- The running instance of a program is called a process.
- Multiple processes can be used to perform multiple tasks concurrently.
- This can make use of existing programs in the system environment.
- Programmers can make use of this functionality when writing programs

Processes in Linux

- In a Linux environment, the functions that are used to manipulate processes are found in the unistd.h header file.

- Have a look at the following for more details:
  - http://pubs.opengroup.org/onlinepubs/7908799/xsh/unistd.h.html

Process Id's

- Every process that is running in a Linux environment must be **uniquely identifiable**. Why??
- A **process ID** is used as the unique identifier for a process.
- The process ID's are **16 bit numbers** and are assigned **sequentially** as processes are spawned.
- Every process has a parent, this can be thought of as a tree structure, where the **init process** is root.

Terminal:: View Process

- The ps command can be used to get the processes that are running on the current system:

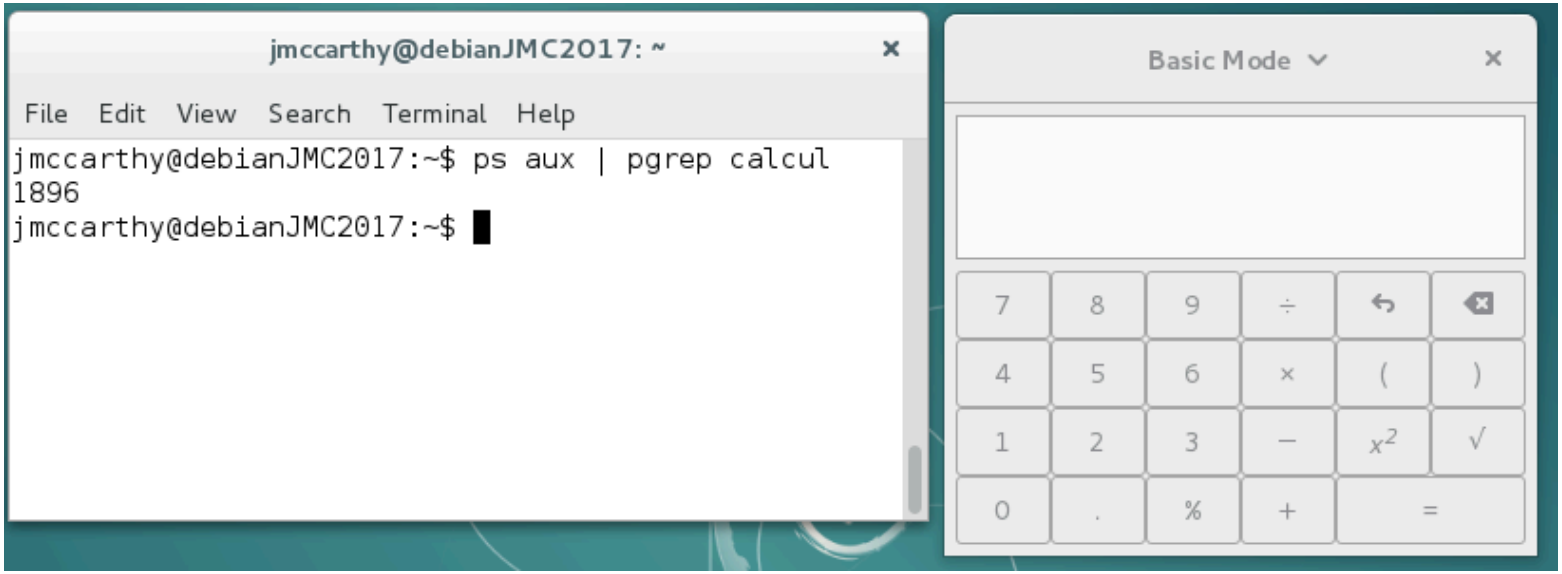
```
jmccarth
File  Edit  View  Search  Terminal  Help
jmccarthy@debianJMC2017:~$ ps
  PID TTY          TIME CMD
 1476 pts/0    00:00:00 bash
 1518 pts/0    00:00:00 ps
jmccarthy@debianJMC2017:~$
```

Terminating a Process

- The kill command is used to kill a process.
- The kill command sends a SIGTERM signal to the process.
- Other signals can be sent to a process, we will see this later in the slides.

Kill Example

- Open a calculator
- Use the PS command with pgrep to find the process
- Kill the process with: **kill 1896**
  - Where 1896 is the process id.



System Calls

- The system call is the fundamental interface between an application and the Linux kernel.
- The following link contains a comprehensive list of system calls:
  - http://man7.org/linux/man-pages/man2/syscalls.2.html

Creating a process

- In Linux there are two main ways to start a process in a C program:
  - System
  - Fork and Exec
- There are some overheads associated with using system, so the preferred option should be fork and exec. There are also some security concerns associated with using system.

System

- The system function is part of the stdlib.h library.
- System allows a C program to run a command (similar to one you would run in a terminal window)

jmccarthy@debianJMC20

File Edit View Search Terminal Help

```
#include <stdlib.h>

int main () {
    int returnValue;
    returnValue = system("ls -al");
    return returnValue;
}
~
~
```

Fork

- The fork command can be used to make a **duplicate** copy of its parent process.
- Fork will duplicate a process. The duplicate is referred to as the **child process**.
- Both processes continue executing from the point the programs forked.
- Both have separate and **unique process ID's.**

Fork Example

Open ▾ | forkExample.c ~/Documents/Apps/week3/forkExample | Save

```c
#include <stdio.h>
#include <unistd.h>

int main(int argc, char **argv)
{
    printf("Start\n");

    pid_t pid = fork();

    printf("\nMy PID is: %d", pid);

    if (pid == 0)
    {
        printf("\nHello from the child process!!\n\n\n");
    }
    else if (pid > 0)
    {
        printf("\nHello from the parent process!!\n\n\n");
    }
    else
    {
        printf("Something went wrong!!");
        return 1;
    }

    printf("End");

    return 0;
}
```

- The returned process ID is of type pid_t
- The fork command will return the pid. The parent will have a pid > 0 and the child will have a pid of 0.
- To get the real pid for the child process use the getpid() function.

exec

- The exec function can be used to replace a program instance currently running in a process.
- When the exec function is called the current process stops and a new program starts executing.

exec

jmccarthy@debianJMC2017: ~/Documents/Apps/week3/forkExample

File Edit View Search Terminal Help

```
EXEC(3)                    Linux Programmer's Manual                    E

NAME
       execl, execlp, execle, execv, execvp, execvpe - execute a file

SYNOPSIS
       #include <unistd.h>

       extern char **environ;

       int execl(const char *path, const char *arg, ...);
       int execlp(const char *file, const char *arg, ...);
       int execle(const char *path, const char *arg,
                  ..., char * const envp[]);
       int execv(const char *path, char *const argv[]);
       int execvp(const char *file, char *const argv[]);
       int execvpe(const char *file, char *const argv[],
                   char *const envp[]);

   Feature Test Macro Requirements for glibc (see feature_test_macros(7)):
 Manual page exec(3) line 1/125 19% (press h for help or q to quit)
```

exec

- **execvp and execlp**
  - Take a program name, doesn’t require full page name
- **execv, execvp, execve**
  - Accept an argument list for the new program (null terminated array)
- **execve and execle**
  - Accept an array of null terminated environmental values

Exec example

Open ▾ | execExample.c ~/Documents/Apps/week3/execExample | Save

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h> /* for fork */
#include <sys/types.h> /* for pid_t */
#include <sys/wait.h> /* for wait */

int main()
{

   char *argv[]={"echo","Echo is running now....",NULL};
   //argv[0] is the command to run: echo
   //argv[1] text to be echoed
   //argv[2] null, needed!!
   execv("/bin/echo",argv);
   printf("Im the last thing in this program to output!!");

   return 0;
}
```

jmccarthy@debianJMC2017: ~/Documents/Apps/week3/execExample

File Edit View Search Terminal Help

```
jmccarthy:execExample$ gcc -o execExample execExample.c
jmccarthy:execExample$ ./execExample
Echo is running now....
jmccarthy:execExample$
```

Using fork and Exec together

Open ▾ | forkexecExample.c ~/Documents/Apps/week3/execExample | Save

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h> /* for fork */
#include <sys/types.h> /* for pid_t */
#include <sys/wait.h> /* for wait */

int main()
{
    /*Spawn a child to run the program.*/
    pid_t pid=fork();
    if (pid==0) { /* child process */
        char *argv[]={"echo","Echo is running now....",NULL};
        //argv[0] is the command to run: echo
        //argv[1] text to be echoed
        //argv[2] null, needed!!
        execv("/bin/echo",argv);
        exit(127); /* only if execv fails */
    }
    else { /* pid!=0; parent process */
        printf("The parent keeps doing whats its doign!!");
    }
    return 0;
}
```

- In this example we fork the program and get exec to run a different process in the fork

Signals

- A signal is a software interrupt
- A program needs to be able to handle software interrupts
- A signal can be used to send an asynchronous message to a program.
- Depending on the signal that was sent the program can decide how to proceed.

Signals in Linux

- In Linux there are a pre-defined set of Signals that perform specific tasks.

jmccarthy@debianJMC2017: ~/Documents/Apps/week3/execExample

File Edit View Search Terminal Help

```
jmccarthy:execExample$ kill -l
 1) SIGHUP       2) SIGINT       3) SIGQUIT      4) SIGILL       5) SIGTRAP
 6) SIGABRT      7) SIGBUS       8) SIGFPE       9) SIGKILL     10) SIGUSR1
11) SIGSEGV     12) SIGUSR2     13) SIGPIPE     14) SIGALRM     15) SIGTERM
16) SIGSTKFLT   17) SIGCHLD     18) SIGCONT     19) SIGSTOP     20) SIGTSTP
21) SIGTTIN     22) SIGTTOU     23) SIGURG      24) SIGXCPU     25) SIGXFSZ
26) SIGVTALRM   27) SIGPROF     28) SIGWINCH    29) SIGIO       30) SIGPWR
31) SIGSYS      34) SIGRTMIN    35) SIGRTMIN+1  36) SIGRTMIN+2  37) SIGRTMIN+3
```

Signal Description

jmccarthy@debianJMC2017: ~/Documents/Apps/week3/execExample

dit View Search Terminal Help

Signal	Value	Action	Comment
SIGHUP	1	Term	Hangup detected on controlling terminal or death of controlling process
SIGINT	2	Term	Interrupt from keyboard
SIGQUIT	3	Core	Quit from keyboard
SIGILL	4	Core	Illegal Instruction
SIGABRT	6	Core	Abort signal from **abort**(3)
SIGFPE	8	Core	Floating point exception
SIGKILL	9	Term	Kill signal
SIGSEGV	11	Core	Invalid memory reference
SIGPIPE	13	Term	Broken pipe: write to pipe with no readers
SIGALRM	14	Term	Timer signal from **alarm**(2)
SIGTERM	15	Term	Termination signal
SIGUSR1	30,10,16	Term	User-defined signal 1
SIGUSR2	31,12,17	Term	User-defined signal 2
SIGCHLD	20,17,18	Ign	Child stopped or terminated
SIGCONT	19,18,25	Cont	Continue if stopped
SIGSTOP	17,19,23	Stop	Stop process
SIGTSTP	18,20,24	Stop	Stop typed at terminal
SIGTTIN	21,21,26	Stop	Terminal input for background process
SIGTTOU	22,22,27	Stop	Terminal output for background process

The signals **SIGKILL** and **SIGSTOP** cannot be caught, blocked, or ignored.

- **In a terminal window type:**
  - **man 7 signal**

Signals in Linux

- When a signal is received, the process needs to tell the kernel how to proceed.
- Potentially, there are 3 possible options:
  - Ignore the signal
  - Catch the signal
  - Go with the signal default
- Note: The specific signals for kill cannot be ignored or caught (Sigkill and Sigstop). Why? If the kernel or an administrator need to stop a process they should be able to do so.
- The header file that offers signal functionality is signal.h

Signal Example

Open

signal1.c
~/Documents/Apps/week3/signal1

```c
#include<stdio.h>
#include<signal.h>
#include<unistd.h>

void sig_handler(int sigNum)
{
  if (sigNum == SIGINT)
    printf("SIGINT Interrupt Received\n");
}

int main(void)
{
  if (signal(SIGINT, sig_handler) == SIG_ERR)
  {
     printf("\nSomething went wrong!!\n");
  }

  // Infinite Loop
  while(1) {
    sleep(1);
  }

return 0;
}
```

Signal Example

jmccarthy@debianJMC2017: ~/Documents/Apps/week3/execExample

File Edit View Search Terminal Help

```
jmccarthy:signal1$ ./signal1
SIGINT Interrupt Received
```

jmccarthy@debianJMC2017: ~/Documents/Apps/week3/signal1

File Edit View Search Terminal Help

```
jmccarthy@debianJMC2017:~/Documents/Apps/week3/signal1$ ps -aux | grep signal1
jmccart+  3193  0.0  0.0   4076   644 pts/0    S+   20:55   0:00 ./signal1
jmccart+  3204  0.0  0.1  12728  2232 pts/1    S+   20:58   0:00 grep signal1
jmccarthy@debianJMC2017:~/Documents/Apps/week3/signal1$ kill -2 3193
jmccarthy@debianJMC2017:~/Documents/Apps/week3/signal1$
```

Unix file system organisation

- Unix operates a hierarchical file system.
- It treats everything as either a file or a directory, this makes the file system very efficient and effective.
- The top level directory is root: /
- The root dierctory always contains a certain set of directories
- To get a list of the files and directories use:
  - ls –F /

ls –F /

- Debian

```
jmccarthy@debianJMC2017: ~
File  Edit  View  Search  Terminal  Help
jmccarthy@debianJMC2017:~$ ls -F /
bin/   etc/          lib/         media/  proc/  sbin/  tmp/  vmlinuz@
boot/  home/         lib64/       mnt/    root/  srv/   usr/
dev/   initrd.img@   lost+found/  opt/    run/   sys/   var/
jmccarthy@debianJMC2017:~$
```

- Mac

```
jmccarthy — -bash — 80×24
Jonathans-MBP:~ jmccarthy$ ls -F /
Applications/                   etc@
Library/                        home/
Network/                        installer.failurerequests
System/                         net/
Users/                          private/
Volumes/                        sbin/
bin/                            tmp@
cores/                          usr/
dev/                            var@
Jonathans-MBP:~ jmccarthy$
```

Bin Directory

- The Bin is a standard subdirectory for the root dir. Its main function is to store executable programs that are ready to run (these can be used as part of the boot sequence, to offer user functionality and/or repairing a system)

```
jmccarthy@debianJMC2017:/bin$ pwd
/bin
jmccarthy@debianJMC2017:/bin$ ls
bash            getfacl         netstat           sync
bunzip2         grep            nisdomainname     systemctl
busybox         gunzip          ntfs-3g           systemd
bzcat           gzexe           ntfs-3g.probe     systemd-ask-password
bzcmp           gzip            ntfs-3g.secaudit  systemd-escape
bzdiff          hciconfig       ntfs-3g.usermap   systemd-inhibit
bzegrep         hostname        ntfscat           systemd-machine-id-setup
bzexe           ip              ntfscluster       systemd-notify
bzfgrep         journalctl      ntfscmp           systemd-tmpfiles
bzgrep          kbd_mode        ntfsfallocate     systemd-tty-ask-password-agent
bzip2           kill            ntfsfix           tailf
bzip2recover    kmod            ntfsinfo          tar
bzless          less            ntfsls            tempfile
bzmore          lessecho        ntfsmove          touch
cat             lessfile        ntfstruncate      true
chacl           lesskey         ntfswipe          udevadm
chgrp           lesspipe        open              ulockmgr_server
chmod           ln              openvt            umount
chown           loadkeys        pidof             uname
chvt            login           ping              uncompress
cp              loginctl        ping6             unicode_start
```

Dev Directory

- The dev directory contains specific device files. These files are created during installation processes.

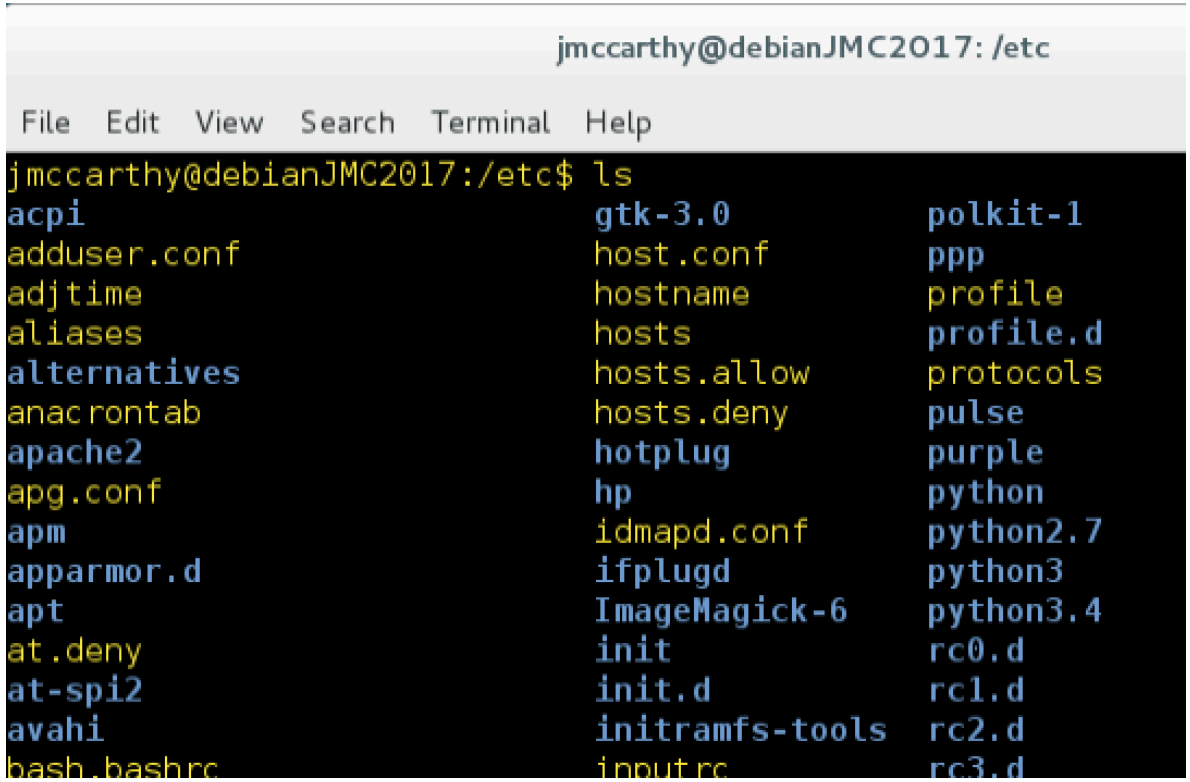
jmccarthy@debianJMC2017: /dev

File Edit View Search Terminal Help

```
jmccarthy@debianJMC2017:/dev$ ls
autofs          mapper          sr0     tty28  tty51     vcs
block           mcelog          stderr  tty29  tty52     vcs1
bsg             mem             stdin   tty3   tty53     vcs2
btrfs-control   mqueue          stdout  tty30  tty54     vcs3
bus             net             tty     tty31  tty55     vcs4
cdrom           network_latency tty0    tty32  tty56     vcs5
char            network_throughput tty1 tty33  tty57     vcs6
console         null            tty10   tty34  tty58     vcs7
core            port            tty11   tty35  tty59     vcsa
cpu             ppp             tty12   tty36  tty6      vcsa1
cpu_dma_latency psaux           tty13   tty37  tty60     vcsa2
cuse            ptmx            tty14   tty38  tty61     vcsa3
disk            pts             tty15   tty39  tty62     vcsa4
dri             random          tty16   tty4   tty63     vcsa5
dvd             rfkill          tty17   tty40  tty7      vcsa6
```

Etc Directory

- The etc directory is used to store files that are used in the administration and management of user accounts, file system, devices drivers etc….



Lib Directory

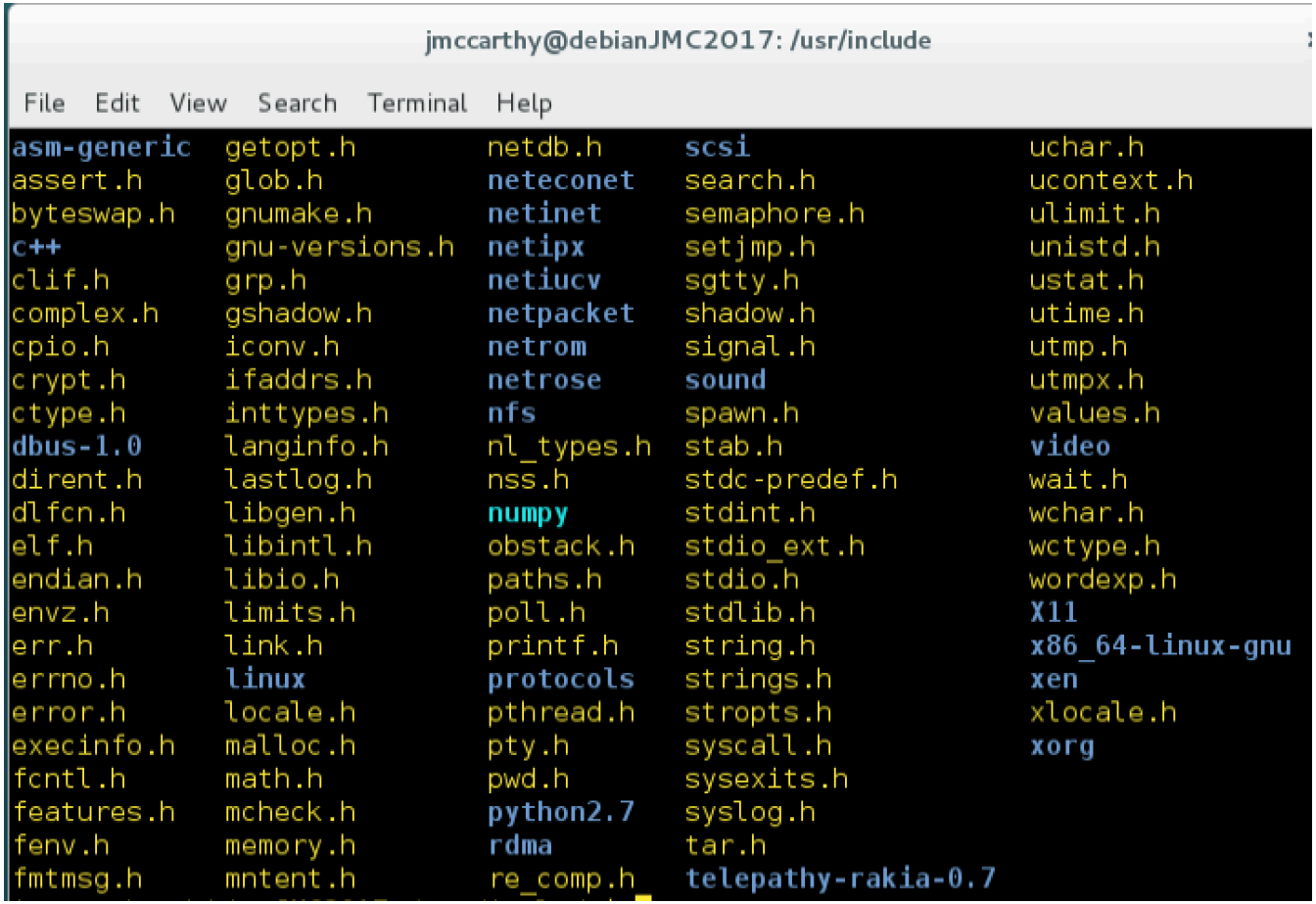
- The lib directory contains kernel modules and shared library images.
- These are used to boot the system and to run commands in the root filesystem
- A lib file has a .so extension. (These are dll files in Windows!!)
- The main concept is to allow programs to include functionality be referencing the lib file.
- The files in the lib folder can be described as system functionality.

Common Directories

- **Mnt Directory**
  - The mnt directory is used to mount external drives and devices etc…
- **Sys Directory**
  - The sys directory is used to hold system config files.
- **Lost+Found**
  - Any files that have not saved correctly due to system error etc… thety will be stored in the lost+found directory.

Usr Directory

- The usr directory contains the majority of data in the system. Usr contains user related programs (as opposed to system programs).



Permissions

- **Permissions types**
  - Read
  - Write
  - Execute
- **Permission Groups**
  - Owner
  - Group
  - All Users

jmccarthy@debianJMC2017: ~/Documents/Apps/week3/fileExample1

File Edit View Search Terminal Help

```
jmccarthy@debianJMC2017:~/Documents/Apps/week3/fileExample1$ ls -al
total 60
drwxr-xr-x 2 jmccarthy jmccarthy 4096 Feb  8 22:23 .
drwxr-xr-x 7 jmccarthy jmccarthy 4096 Feb  8 20:07 ..
-rwxr-xr-x 1 jmccarthy jmccarthy 7520 Feb  8 20:11 fileExample1
-rw-r--r-- 1 jmccarthy jmccarthy  318 Feb  8 20:11 fileExample1.c
-rwxr-xr-x 1 jmccarthy jmccarthy 7392 Feb  8 20:48 fileExample2
-rw-r--r-- 1 jmccarthy jmccarthy  310 Feb  8 20:47 fileExample2.c
-rwxr-xr-x 1 jmccarthy jmccarthy 7080 Feb  8 22:10 fileExample3
-rw-r--r-- 1 jmccarthy jmccarthy  179 Feb  8 22:10 fileExample3.c
-rwxr-xr-x 1 jmccarthy jmccarthy 7176 Feb  8 22:23 fileExample4
-rw-r--r-- 1 jmccarthy jmccarthy  237 Feb  8 22:23 fileExample4.c
-rw-r--r-- 1 jmccarthy jmccarthy   36 Feb  8 22:19 temp.txt
jmccarthy@debianJMC2017:~/Documents/Apps/week3/fileExample1$
```

```
jmccarthy@debianJMC2017:~/Documents/Apps/week3/fileExample1$ chmod 777 temp.txt
jmccarthy@debianJMC2017:~/Documents/Apps/week3/fileExample1$ ls -al temp.txt
-rwxrwxrwx 1 jmccarthy jmccarthy 36 Feb  8 22:19 temp.txt
jmccarthy@debianJMC2017:~/Documents/Apps/week3/fileExample1$
```

Types of files

- There are two types of files we will be dealing with:
  - Text files
  - Binary files
- Text files store basic plain text.
- Binary files are 1's and 0's and can store larger amounts of data and is more secure.

File Operations

- What can we do:
  - Create a new file
  - Open a file
  - Read a file
  - Close a file

fopen

```
FOPEN(3)                    Linux Programmer's Manual                    FO

NAME
       fopen, fdopen, freopen - stream open functions

SYNOPSIS
       #include <stdio.h>

       FILE *fopen(const char *path, const char *mode);

       FILE *fdopen(int fd, const char *mode);

       FILE *freopen(const char *path, const char *mode, FILE *stream);
```

↗ From the man pages

fopen

- The fopen() function opens the file whose name is the string pointed to by path and associates a stream with it.
- The argument mode points to a string beginning with one of the following:

Mode	Description
r	Open a file for reading. Stream starts at beginning of file
r+	Open for reading and writing. Stream at beginning of file
w	Create a file or overwrite what was there. Stream at beginning of file.
w+	Open for reading and writing. The file is created if it does not exist, otherwise it is truncated. The stream is positioned at the beginning of the file.
a	Open for appending (writing at end of file). The file is created if it does not exist. The stream is positioned at the end of the file.
a+	Open for reading and appending (writing at end of file). The file is created if it does not exist. The initial file position for reading is at the beginning of the file, but output is always appended to the end of the file.

fopen – Writing to File

Open

fileExample1.c
~/Documents/Apps/week3/fileExample1

```c
#include <stdio.h>
#include <stdlib.h>

int main()
{
   char *name[50];
   FILE *fptr;
   fptr = fopen("temp.txt","w");

   if(fptr == NULL)
   {
      printf("Error!");
      exit(1);
   }

   printf("Enter name: ");
   scanf("%s",&name);

   fprintf(fptr,"%s",name);
   fclose(fptr);

   return 0;
}
```

- Using fopen to create the file, if it doesn't exist.
- If the file pointer is null something went wrong, exit
- Use fprintf to write the name to the file
- int fprintf(FILE *stream, const char *format, ...);
- Use fclose to close the file pointer.

fopen – Reading from File

Open

fileExample2.c
~/Documents/Apps/week3/fileExample1

```c
#include <stdio.h>
#include <stdlib.h>

int main()
{
    char *name[50];
    FILE *fptr;
    fptr = fopen("temp.txt","r");

    if (fptr == NULL){
        printf("Error! opening file");
        exit(1);
    }

    fscanf(fptr,"%s",&name);

    printf("\n\nValue of n= %s\n\n",name);
    fclose(fptr);

    return 0;
}
```

- Using fopen to open the file, null returned if can't open file
- If the file pointer is null something went wrong, exit
- Use fscanf to read the name from the file
- int fscanf(FILE *stream, const char *format, ...);
- Use fclose to close the file pointer.

fputs

jmccarthy@debianJMC2017: ~/Documents/Apps/week3/fileExample1

File Edit View Search Terminal Help

```
PUTS(3)                    Linux Programmer's Manual                    PUTS(3)

NAME
       fputc, fputs, putc, putchar, puts - output of characters and strings

SYNOPSIS
       #include <stdio.h>

       int fputc(int c, FILE *stream);

       int fputs(const char *s, FILE *stream);

       int putc(int c, FILE *stream);

       int putchar(int c);

       int puts(const char *s);

DESCRIPTION
       fputc() writes the character c, cast to an unsigned char, to stream.

       fputs()  writes  the  string  s to stream, without its terminating null
       byte ('\0').
```

fputs example

Open

fileExample3.c

~/Documents/Apps/week3/fileExample1

```
#include <stdio.h>

main() {
   FILE *fp;

   fp = fopen("temp.txt", "w+");
   fprintf(fp, "Writing with fprintf...\n");
   fputs("Writing with fputs...\n", fp);
   fclose(fp);
}
```

fgets Example

Open ▾ fileExample4.c ~/Documents/Apps/week3/fileExample1

fileExample4.c

```c
#include <stdio.h>

main() {

   FILE *fp;
   char buff[255];
   int i;

   fp = fopen("temp.txt", "r");

   for (i = 0 ; i < 3 ; i++)
   {
      fgets(buff, 255, fp);
      printf("Line %d: %s\n", i , buff );
   }

   fclose(fp);

}
```

Open ▾ temp.txt ~/Documents/Apps/week3/

```
Bruce Banner
Barry Allen
Clark Kent
```

jmccarthy@debianJMC2017: ~/Documents/Apps/week3/fileExample1

File Edit View Search Terminal Help

```
jmccarthy@debianJMC2017:~/Documents/Apps/week3/fileExample1$ ./fileExample4
Line 0: Bruce Banner

Line 1: Barry Allen

Line 2: Clark Kent

jmccarthy@debianJMC2017:~/Documents/Apps/week3/fileExample1$
```

fread and fwrite – Binary Files

jmccarthy@debianJMC2017: ~/Documents/Apps/week3/fileExample1

File Edit View Search Terminal Help

```
FREAD(3)                    Linux Programmer's Manual                    FREAD(3)

NAME
       fread, fwrite - binary stream input/output

SYNOPSIS
       #include <stdio.h>

       size_t fread(void *ptr, size_t size, size_t nmemb, FILE *stream);

       size_t fwrite(const void *ptr, size_t size, size_t nmemb,
                     FILE *stream);

DESCRIPTION
       The  function  fread()  reads  nmemb  elements of data, each size bytes
       long, from the stream pointed to by stream, storing them at  the  loca-
       tion given by ptr.

       The  function  fwrite()  writes nmemb elements of data, each size bytes
       long, to the stream pointed to by stream, obtaining them from the loca-
       tion given by ptr.
```

fseek

```
NAME
       fgetpos, fseek, fsetpos, ftell, rewind - reposition a stream

SYNOPSIS
       #include <stdio.h>

       int fseek(FILE *stream, long offset, int whence);

       long ftell(FILE *stream);

       void rewind(FILE *stream);

       int fgetpos(FILE *stream, fpos_t *pos);
       int fsetpos(FILE *stream, const fpos_t *pos);
```

fseek

DESCRIPTION

The **fseek**() function sets the file position indicator for the stream pointed to by stream. The new position, measured in bytes, is obtained by adding offset bytes to the position specified by whence. If whence is set to **SEEK_SET**, **SEEK_CUR**, or **SEEK_END**, the offset is relative to the start of the file, the current position indicator, or end-of-file, respectively. A successful call to the **fseek**() function clears the end-of-file indicator for the stream and undoes any effects of the **ungetc**(3) function on the same stream.

The **ftell**() function obtains the current value of the file position indicator for the stream pointed to by stream.

The **rewind**() function sets the file position indicator for the stream pointed to by stream to the beginning of the file. It is equivalent to:

```
(void) fseek(stream, 0L, SEEK_SET)
```

except that the error indicator for the stream is also cleared (see **clearerr**(3)).

gprof

- gprof - A GNU Profiler For Performance Analysis Of Programs
- Gprof is a software profiler tool to measure the performance of an application.
- The profiler will analyse which functions are taking time to execute.
- Individual functions can be measured to compare their performance against the calling parent function.

Install gprof

- Most distributions will have gprof installed be default.
  - ***apt-get install binutils***

gprof usage

- Step 1: Make sure gprof is installed
- Step 2: compile the program with the –Wall –pg flags
- gcc -Wall enables all compiler's warning messages.
- -pg

gProf

- Profiling works by changing how every function in your program is compiled so that when it is called, it will stash away some information about where it was called from. From this, the profiler can figure out what function called it, and can count how many times it was called. This change is made by the compiler when your program is compiled with the `-pg' option, which causes every function to call mcount as one of its first operations.

Source: https://sourceware.org/binutils/docs/gprof/Implementation.html

gProf

- The mcount routine, included in the profiling library, is responsible for recording in an in-memory call graph table both its parent routine (the child) and its parent's parent.
- Profiling also involves watching your program as it runs, and keeping a histogram of where the program counter happens to be every now and then.
- Typically the program counter is looked at around 100 times per second of run time, but the exact frequency may vary from system to system.

Source: https://sourceware.org/binutils/docs/gprof/Implementation.html

gProf

- gprof is an **instrumenting profiler**, it is profiling the same code you would compile in release without profiling instrumentation. There is an overhead associated with the instrumentation code itself. Also, the instrumentation code may alter instruction and data cache usage.
- A **sampling profiler** works on non instrumented code by looking at the target program's program counter at regular intervals using operating system interrupts. It can also query special CPU registers to give you even more insight of what's going on.

gProf

```
Flat profile:

Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  ms/call  ms/call  name
 96.02      0.58     0.58        1   576.13   576.13  yet_another_test
  5.05      0.61     0.03        1    30.32   606.45  test
  0.00      0.61     0.00        1     0.00   576.13  another_test
  0.00      0.61     0.00        1     0.00     0.00  some_other_test
```

gProf

```
 %          the percentage of the total running time of the
time        program used by this function.

cumulative a running sum of the number of seconds accounted
 seconds   for by this function and those listed above it.

 self       the number of seconds accounted for by this
seconds    function alone.  This is the major sort for this
           listing.

calls      the number of times this function was invoked, if
           this function is profiled, else blank.
```

gProf

```
 self      the average number of milliseconds spent in this
ms/call    function per call, if this function is profiled,
       else blank.

 total     the average number of milliseconds spent in this
ms/call    function and its descendents per call, if this
       function is profiled, else blank.

name       the name of the function.  This is the minor sort
           for this listing. The index shows the location of
       the function in the gprof listing. If the index is
       in parenthesis it shows where it would appear in
       the gprof listing if it were to be printed.
```

Gprof – Call Graph

```
granularity: each sample hit covers 2 byte(s) for 1.65% of 0.61 seconds

index % time    self  children    called     name
                0.03    0.58       1/1           main [2]
[1]     100.0   0.03    0.58       1         test [1]
                0.00    0.58       1/1           another_test [3]
-----------------------------------------------
                                                  <spontaneous>
[2]     100.0   0.00    0.61                 main [2]
                0.03    0.58       1/1           test [1]
                0.00    0.00       1/1           some_other_test [5]
-----------------------------------------------
                0.00    0.58       1/1           test [1]
[3]      95.0   0.00    0.58       1         another_test [3]
                0.58    0.00       1/1           yet_another_test [4]
-----------------------------------------------
                0.58    0.00       1/1           another_test [3]
[4]      95.0   0.58    0.00       1         yet_another_test [4]
-----------------------------------------------
                0.00    0.00       1/1           main [2]
[5]       0.0   0.00    0.00       1         some_other_test [5]
-----------------------------------------------
```

gprof – Call Table

```
This table describes the call tree of the program, and was sorted by
the total amount of time spent in each function and its children.

Each entry in this table consists of several lines.  The line with the
index number at the left hand margin lists the current function.
The lines above it list the functions that called this function,
and the lines below it list the functions this one called.
```

- See the output of a gprof file for more details!!

Questions