System Software - Assignment 1
Owen Kane - C13383511

I started my solution to this assignment my creating a daemon.

```
// Create a child process
int pid = fork();
int pid2 = fork();
int ppid;
int ppid2;


FILE *pFile;
char buffer[256];
ppid = getpid();
ppid2 = getpid();

if (pid > 0) {
    printf("Starting Process\nCreating Parent and child");
    // if PID > 0 :: this is the parent
    // this process performs printf and finishes
    sleep(5);  // uncomment to wait 10 seconds before process ends
    exit(EXIT_SUCCESS);
} else if (pid == 0) {
    // Step 1: Create the orphan process
    printf("Child process is now a orphan process\n");

    // Step 2: Elevate the orphan process to session leader, to loose controlling TTY
    // This command runs the process in a new session
    if (setsid() < 0) { exit(EXIT_FAILURE); }

    // We could fork here again , just to guarantee that the process is not a session leader
    int pid = fork();
    if (pid > 0) {
        exit(EXIT_SUCCESS);
    } else {
        printf("Daemon process created and running\n");
        // Step 3: call umask() to set the file mode creation mask to 0
        // This will allow the daemon to read and write files
        // with the permissions/access required
        umask(0);
```

I did this by forking a process and killing its parent, this child process is now a orphan process and is made session leader. Once this is done, we create a run a new session.
After creating another orphan process I began to implement the functionality outlined by the CTO.

```
        umask(0);

        // Step 4: Change the current working dir to root.
        // This will eliminate any issues of running on a mounted drive,
        // that potentially could be removed etc..
        if (chdir("/") < 0 ) { exit(EXIT_FAILURE); }

        // Step 5: Close all open file descriptors
        /* Close all open file descriptors */
        int x;
        for (x = sysconf(_SC_OPEN_MAX); x>=0; x--)
        {
            close (x);
        }

        while(1) {
            int value = getTime();
        }
    }
}
```

I created a infinitive loop after completing the daemon setup. This was to insure the process to backup the the website was always running.
We could also always find out the current status of the backup process by examining the value returned by the getTime() function.

timer.c

```
int getTime() {

    int argc;
    char **argv;
    sleep(1);
    int value = 1;
    time_t now;
    struct tm newyear;
    double seconds;
    newyear = *localtime(&now);
    newyear.tm_hour = 23;
    newyear.tm_min = 59;
    newyear.tm_sec = 0;
    time(&now);  /* get current time; same as: now = time(NULL)  */

    seconds = difftime(now,mktime(&newyear));
    if (seconds  == 0) {

        int lockedValue = lock();
        char backup[100] = "/Users/owenkane/Desktop/SS/Backup/www/";
        char toBackUp[100] = "/Users/owenkane/Desktop/SS/Intranet/www";
        char str_timestamp[80];

        strftime(str_timestamp, 80, "%B_%d_%H", &newyear);
        strcat(backup, str_timestamp);

        char *command = "/bin/cp";
        char *arguments[] = { "cp", "-R", toBackUp, backup, NULL };

        int unlockedValue = unlock();
```

In this function I backup the the files store in the Intranet directory. First I setup a time variables to allow the user to chose when the files are backed up.
I then subtract the current time from the user set time and when difference is 0, the backup process begins.

```
    if (seconds  == 0) {

        int lockedValue = lock();
        char backup[100] = "/Users/owenkane/Desktop/SS/Backup/www/";
        char toBackUp[100] = "/Users/owenkane/Desktop/SS/Intranet/www";
        char str_timestamp[80];

        strftime(str_timestamp, 80, "%B_%d_%H", &newyear);
        strcat(backup, str_timestamp);

        char *command = "/bin/cp";
        char *arguments[] = { "cp", "-R", toBackUp, backup, NULL };

        int unlockedValue = unlock();


        if(lockedValue == 1) {
            logFile("lock", "Success");
        } else {
            logFile("lock", "Failure");
        }

        if(unlockedValue == 1) {
            logFile("Transfer", "Success");
            logFile("Unlock", "Success");
            execvp(command, arguments);
        } else {
            logFile("Unlock", "Failure");
        }
    }
    return value;
```

When the backup process begins, the files are locked so they cannot be changed during the process. I then use the “cp” command in execvp() to copy the contents of the selected directories. Once the process is over the directories are unlocked. We also log the the status of the these events in the log file.

Lock.c

```
int lock()
{
    int value = 1;
    char mode[] = "1111";
    char buf[100] = "/Users/owenkane/Desktop/SS/Intranet/";
    int i;
    i = strtol(mode, 0, 8);
    return value;

}

int unlock() {
    sleep(10);
    int value = 1;
    char mode[] = "07777";
    char buf[100] = "/Users/owenkane/Desktop/SS/Intranet/";
    int i;
    i = strtol(mode, 0, 8);
    return value;
}
```

To do this I edit the privilege value depending if they are to be locked or unlocked.

logFile.c

```c
void logFile(char message[], char status[]){
    FILE *fp;
    fp = fopen("/Users/owenkane/Desktop/SS/log.txt","a");

    time_t now;
    time(&now);/* get current time; same as: now = time(NULL)  */

    char *buf;
    buf=(char *)malloc(10*sizeof(char));
    buf=getlogin(); // get user logged in
    printf("\n %s \n",buf);

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

    fprintf(fp,"\n-----New-Entry-----");
    fprintf(fp,"Time    : %s\n",ctime(&now));
    fprintf(fp,"Status  : %s\n",status);
    fprintf(fp,"Message : %s\n",message);
    fprintf(fp,"User    : %s\n",buf);
    fprintf(fp,"\n--------------------");
    fclose(fp);
}
```

To log files I take action competed as a message and the status of it (success or fail) as a parameters of the function. I the get the time and the user logged in who is currently logged in (Who caused the action) with the getlogin() function. This is then written to the log.txt file.

sync.c

Unfortunately I wasn't able to prompt the daemon to run the backup of changed Intranet site file to the live site, instead a created a new file just to do this. This rsync command is a linux command to backup only changed files from one directory to another.

Main.c cont

```
if (pid > 0) {
    sleep(2);
    exit(EXIT_SUCCESS);
} else if (pid == 0) {
    /*
    This function creates a file with all the files updated in the last 24 hours
    by piping outputs of execlp()s, however when introduced to the whole program it fails to work.
    */
    find_updated();
}
```

I created another child process to create a file of the last files that have been modified in the last 24 hours. However when I introduced this to the rest of the code it failed to run unfortunately.

find_updated.c

```
int find_updated() {
  char data[4096];
  FILE *f = fopen("updated.txt", "w");

  // Create pipe between find and awk
  pipe(pipefd1);

  // fork (find)
  pid = fork();

  if (pid == 0) { //If child
    dup2(pipefd1[1], 1);

    // close fds
    close(pipefd1[0]);
    close(pipefd1[1]);

    execlp("find", "find", "/Users/owenkane/Desktop/SS/Intranet/www", "-mtime", "-1", "-ls",  NULL);

    _exit(1);
  }
```

Here I create a child process that will communicate with other process to join the output of two separate execlp() to find the files that have been edited in the last 24 hours.
I first created the file, forked the process and set up the pipes so they could communicate output, despite not being in the same process. I used the above find command to find files that have been edited in the last 24 hours, however this returns far too much information that we need as we just require the paths of these files.

```
// Create pipe between awk and sort
pipe(pipefd2);

// fork (awk)
pid = fork();

if (pid == 0) {
  // get the input from pipe 1
  dup2(pipefd1[0], 0);

  // set the output to pipe 2
  dup2(pipefd2[1], 1);

  // close fds
  close(pipefd1[0]);
  close(pipefd1[1]);
  close(pipefd2[0]);
  close(pipefd2[1]);

  // exec (awk), specifying user, date and filepath
  execlp("awk", "awk", "{print $11}", NULL);

  // error check
  perror("bad exec grep root");
  _exit(1);
}
```

I then create the pipe link between the find command and the awk command that will be executed to refine the output.
In the new process that is created I get the input from the find command and set the the output of the second pipe. I then use the awk command to refine the output of the previous find command to just view the path of the file.

```
  // close unused fds
  close(pipefd1[0]);
  close(pipefd1[1]);

  // fork (sort)
  if (pid == 0) {
    // get the input from pipe 2
    dup2(pipefd2[0], 0);

    // close fds
    close(pipefd2[0]);
    close(pipefd2[1]);

    _exit(1);
  } else {
    close(pipefd2[1]);
    int nbytes = read(pipefd2[0], data, sizeof(data));
    //printf("%.*s", nbytes, data);
    fprintf(f, "%.*s", nbytes, data);
    close(pipefd2[0]);
    fclose(f);
  }

  return 1;
}
```

Once I have completed both commands I close the unused file descriptors, get the output from out second pipe, close its file descriptors and then write the result (the refined paths) to the file and close the file.

Makefile

```
CC=gcc
objects = main.o timer.o locked.o log.o find_updated.o
headers = main.h timer.h locked.h log.h find_updated.h

myprog : $(objects)
    $(CC) -o prog $(objects) -lm

main.o : main.c $(headers)
    $(CC) -c main.c

timer.o : timer.c
    $(CC) -c timer.c

locked.o : locked.c
    $(CC) -c locked.c

log.o : log.c
    $(CC) -c log.c

find_updated.o : find_updated.c
    $(CC) -c find_updated.c

clean:
    rm prog main.o timer.o locked.o log.o find_updated.o
```

I created a makefile to describe the process needed to recompile and link a program. It contains a target, a target is usually the name of a file that is generated by a program. However a target can also be the name of an action to carry out, such as ‘clean’ here, which deletes the files which should be removed before recompiling. A makefile also defines the prerequisite which are required as input to create the target. It also defines a “recipe”, which is an action that make carries out. A recipe may have more than one command, either on the same line or each on its own line.