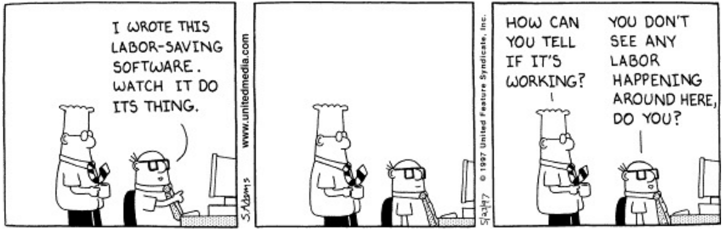
Week 6: Files (Part II), and Processes

CS211: Programming and Operating Systems

Wednesday and Thursday, 19+20 Feb 2020



This week in CS211:

This week of "Programming and Operating Systems", we begin to segue from **Programming** to **Operating Systems**.

The first section is all on files (Programming). Then we'll learn about the concept of a **process** (OS), and then how to write C programs that manipulate processed (Programming + OS).

Recall... Files

Continuing from Week 5, we will learn to manipulate files in C.
We have already seen how to

- Declare an identifier for the file: `FILE *`
- Open the file: `fopen`
- Read from it: `fgetc`, `fgets`
- Close the file: `fclose`

Today:

- **Check and change file counter:** `rewind`, `ftell` and `fseek`.
- **Writing:** `fputc`, `fputs` and `fprintf`

- Declare an identifier for the file, (`FILE *`), e.g.,

```
FILE *datafile;
```

- open the file with `fopen` in `read` mode

```
datafile = fopen("example.txt", "r");
```

If the file can't be opened, the value `NULL` is stored in `datafile`.

- close the file. (`fclose`)

```
fclose(datafile);
```

Between opening and closing the file, we'll want to read from it:

- `fgets`

```
fgets(string, n, fileptr)
```

reads in a line of text from the $fileptr$ stream and stores at most `n` characters in array `sting`. The new line character is stored.
If the string can't be read, because we have reached the end of the file, then `NULL` is returned.

- `fgetc`

```
c = fgetc(fileptr)
```

reads the next character in the file and stores it in the `char` variable `c`. If the end of the file has been reached, `EOF` is returned.

File position indicator

Each time a character is read from the input stream, a counter associated with the stream is incremented.
In Week 5 (`03CountLinesWithfgets.c`) we saw this when we used the `rewind` function:

rewind

`rewind(fileptr)` sets the indicator to the start of the file. This was used in our earlier examples.

There are some other useful function which can be used

- To determine here in the file we are: `ftell`
- To move to a particular location in the file: `fseek`.

File position indicator

ftell

To check the current value of the file position indicator, use:

```
long ftell(FILE *stream);
```

It will return the current value of the file position indicator, in the form of a long int.

For example, if we are at the beginning of the file, then `ftell(file)` should evaluate as *0*.

File position indicator

fseek

To modify the value of the indicator:

`fseek(fileptr, offset, place)`

The value of *offset* is the amount the indicator will be changed by, while `place` is one of

- *SEEK_SET* (0), refers to the start of the stream,
- *SEEK_CUR* (1), refers to the current position of the indicator,
- *SEEK_END* (2), refers to the end of the stream,

For example,

`fseek(file, 0, SEEK_SET)`

is equivalent to

`rewind(file)`.

File position indicator

Example

Here is an easy way of counting the number of characters in a file:

```
fseek(file, 0, SEEK_END);
printf("There are %ld chars in the file\n",
    ftell(file));
```

File position indicator

Example

Write a programme that will open a file and output its contents in reverse.

01Reverse.c

```
int main( void)
{
  FILE *InFile;
   char c;

   InFile=fopen("01Reverse.c", "r");
   if ( InFile == NULL )
   {
     printf("Error - could not open the file\n");
     exit(1);
   }
```

File position indicator

01Reverse.c (cont.)

```c
    // First go to the end of the file
    fseek(InFile, 0, SEEK_END);

    // Now read lines in reverse order
    while (ftell(InFile) != 0)
    {
      c=fgetc(InFile);
      putchar(c);
      fseek(InFile, -2, SEEK_CUR);
    }

  fclose(InFile);
  return(0);
}
```

See also the exercise on Slide 33.

Example: Random Lines

In our next example, we'll write a program that reads a number of lines from a file and then outputs them at random.
It contains the following

- Some comments
- Some `#include directives`
- The beginning of the `main` function, followed by some variable declarations.
- Copies the string `02RandomLines.c` to the array `FileName`; tries to open the file for reading; if that fails, generate an error and exit.
- Reads each line of the file into the two dimensional `char` array `lines[][]`; for each line, increments the variable `NumberOfLines` ; closes the file.

Example: Random Lines

- Set the integer variable `Deleted` to `0`.
- Until all lines have been "***deleted***",
  - generate a random number between `0` and `NumberOfLines`
  - If the corresponding line has not yet been deleted,
    - > display the line,
    - > "delete" the line by setting the first char to `\0`
    - > increment the `Deleted` variable.

`02RandomLines.c`

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

int main(void )
{
  int i, NumberOfLines=0, Deleted, WhichLine;
  char lines[100][100], FileName[30];
  FILE *infile;
```

Example: Random Lines

02RandomLines.c

```
  strcpy(FileName, "02RandomLines.c");
  infile = fopen(FileName, "r");
  if (infile == NULL)
  {
    printf("Error: can't open %s for reading",
           FileName);
    exit(EXIT_FAILURE);
  }

  for (i=0; (fgets(lines[i], 99, infile)) != NULL; i++)
    NumberOfLines++;

  fclose(infile);
```

Example: Random Lines

02RandomLines.c

```
// Now display non-empty lines in a random order
Deleted=0;
while(Deleted < NumberOfLines)
{
  WhichLine = rand()%NumberOfLines;
  if (lines[WhichLine][0] != '\0')
  {
    printf("%s", lines[WhichLine]);
    lines[WhichLine][0]='\0';
    Deleted++;
  }
}
return(EXIT_SUCCESS);
```

Writing data to a file

Finally, we will student how to create a new file and write data to it.

First, as usual, declare a file pointer:

```
FILE *outfile;
```

Then open a new file in `w`rite mode:

```
outfile=fopen("NewList.txt", "w");
```

To write to the file, use one of

- `fprintf(FILE *stream, ...)`: works just like `printf()` except that its first argument is the output stream.
- `fputc(char c, FILE *stream)`: writes the character `c` to the stream,
- `fputs(char *str, FILE *stream)`: writes the string *str* to the stream, without its trailing `'\0'`

Writing data to a file

Example

Write a program that copies every fifth line from an input file into an output file.

03DeleteLines.c

```
int main(void)
{
  FILE *infile, *outfile;
  char InFileName[99], OutFileName[99], Line[99];
  int i;

  printf("Enter the name of the input file: ");
  scanf("%s", InFileName);
  printf("Enter the name of the output file: ");
  scanf("%s", OutFileName);
```

Writing data to a file

03DeleteLines.c

```
  infile = fopen(InFileName, "r");
  if (infile == NULL)
  {
    printf("Can't open %s in read mode\n",
           InFileName);
    exit(EXIT_FAILURE);
  }
  outfile = fopen(OutFileName, "w");
  if (outfile == NULL)
  {
    printf("Can't open %s in write mode\n",
           OutFileName);
    exit(EXIT_FAILURE);
  }
```

Writing data to a file

03DeleteLines.c

```
  i=0;
  while ( fgets(Line, 99, infile) != NULL )
  {
    i++;
    if (i%5 == 0)
      fputs(Line, outfile);
  }

  fclose(infile);
  fclose(outfile);

  return(EXIT_SUCCESS);
}
```

Further points

Issues concerning the use of files in C, but which we haven't covered, include

- There are in fact 6 modes a file can have: `r`, `w`, `a`, `r+`, `w+`, `a+`.
- To open a binary file, also include the letter `b` as part of the mode.
- `freopen()` attaches a new file to an existing stream
- `tmpfile()` opens a temporary file in binary read/write (w+b) mode. The file is automatically deleted when it is closed or the program terminates.
- `fflush()` flushes a buffer
- `remove()` and `rename()` can be used to manipulate files in a directory.
- `int feof(FILE *stream)` returns a nonzero character if the file position indicator is at the end of the file.

The Process

As we now move towards the "Operating System" part of the course, the need to learn some classical OS Theory. The presentation given here is quite standard, and you should find equivalent descriptions in any OS text-book.

Material from this point one relates to Chapters 4 and 5 of Operating Systems: Three Easy Pieces by Remzi H. Arpaci-Dusseau and Andrea C. Arpaci-Dusseau:

Processes: `http://pages.cs.wisc.edu/~remzi/OSTEP/cpu-intro.pdf`

Process API: `http://pages.cs.wisc.edu/~remzi/OSTEP/cpu-api.pdf`

"A Process... is a running programme" (OSTEP, p25)

Most OS will give the impression that many programmes are running at one time. The user/programmer does not know or care of the CPU is currently busy: the OS gives them the impression that it is available for their (exclusive) use.

This is made possible by abstracting the concept of a running program as a **process**.

"*The OS creates this illusion by virtualizing the CPU*": we will study, later, how this scheduling achieved. For now, we will take it that we need the concept of the **process** to do this.

Every process consists of:

- the ***Process Text*** - the code of the program
- the ***program counter*** – the address of the next instruction to be executed.
- the ***process stack*** (temporary data, e.g., local variables, return addresses, etc)
- the ***data section*** – global variables.

A process is ***not*** (just) a program: if two users run the same program at the same time they create different processes.
A program is a ***passive*** entity, whereas a process is ***active/dynamic***.
Often, the terms *process* and *job* can be used interchangeably.

Process API

Here is a minimal set of operations that an OS must be able to apply to a process.

Create
: a new process, e.g., when you click on an icon.

Destroy
: (or terminate) a process,

Wait
: that is **pause** the process until some other event occurs.

Suspend and resume:
: like wait, but invoked more explicitly.

Status
: report: information about a process, such as how long it has run for, how much memory has been allocated to it, etc.

The **state** of a process is defined (in part) by the current activity of that process:

- **new**: The process is being created
- **running**: Instructions are being executed
- **blocked**: (also called "waiting"). The process is waiting for some event to occur
- **ready**: The process is waiting to be assigned to a processor
- **terminated**: The process has finished execution.

Process Creation

A parent process creates children processes, which, in turn create other processes, forming a tree of processes.
After a parent creates a subprocess it may:

- execute ***concurrently*** with the child
  or
- ***wait*** until child terminates before it continues.

The parent may share all, some or none of its resources with the child (resources include memory space, open files, the terminal, etc.)
It is usually the case that the child will share the parent's memory only in the sense that it receives a copy.
The child can then mimic the parents execution, or it might over-write (or "*over-lay*") its memory space with other instructions.

All processes have a unique ***PID*** – a process identification number. If we create a subprocess in a C program using the `fork()` function, a new process is created:

- The new process run concurrently with its parent unless we instruct the parent to `wait()`
- The subproc is given a copy of the parents memory space.
- At the time of creation, the two processes are almost identical, except that the `fork()` returns the child's PID to the parent and 0 to the child.

In order to use this function, we must include the *`unistd.h`* header file.

04Fork.c

```c
// An example of forking a process

#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>


int main(void )
{
   int pid1, mypid;

   pid1 = fork();
   mypid = getpid();

   printf("I am %d\t", mypid);
   printf("Fork returned %d\n", pid1);
   return(0);
}
```

When I compile and execute this (e.g., on https://www.onlinegdb.com/) I get something like

```
I am 7791. Fork returned 0
I am 7790. Fork returned 7791
```

IMPORTANT: *unistd.h* is not included in the installation of `code::blocks` on campus. Try

- https://www.onlinegdb.com/online_c_compiler
- https://www.jdoodle.com/c-online-compiler
- https://paiza.io/projects/
- https://rextester.com/l/c_online_compiler_gcc
- But not https://www.tutorialspoint.com/compile_c_online.php or http://www.compileonline.com/ or https://www.codechef.com/. Also problematic: https://repl.it/languages/c and https://ideone.co

05Fork2.c

```
// An example of forking two processes
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>

int main(void )
{
  int pid1, pid2, mypid;

  pid1 = fork();
  pid2 = fork();
  mypid = getpid();

  printf("I am %d\t", mypid);
  printf("1st fork returned %d\t", pid1);
  printf("2nd fork returned %d\n", pid2);
  return(0);
}
```

Running that we might get:

```
I am 7802. 1st Fork returned 7803. 2nd Fork returned 7805
I am 7803. 1st Fork returned 0. 2nd Fork returned 7804
I am 7804. 1st Fork returned 0. 2nd Fork returned 0
I am 7805. 1st Fork returned 7803. 2nd Fork returned 0
```

Discuss: Why do we get this output?

Process Creation

Example 3: getppid()

The parent knows the child's PID because it is returned by `fork()`. The child can find out its parent's PID, by using the `getppid()` function:

06ParentsPID.c

```c
int main(void )
{
  int pid1;
  pid1 = fork();
  printf("I am %d\t", getpid());
  printf("fork returned %5d\t", pid1);
  printf("My parent is %d\n", getppid());
  return(0);
}
```

```
OUTPUT:
I'm proc 7825. fork() returned 0. My partent is 7824
I'm proc 7824. fork() returned 7825. My partent is 5394
```

Exercises

Exercise (Exer 6.1)

In the `04CountLinesWithfgetc.c` from Week 5, we used `rewind()` to move the file position indicator to the start of the file, before counting the number of lines, and then rewind it when we are done. This means that, after any call to `file_length()` the file position indicator is set to the start of the file; that is, we lose the current position.
Improve the code so that in the `file_length()` function

- *first stores the current file position;*
- *then `rewind`s the file;*
- *counts the the number of lines;*
- *resets the file position indicator.*