BSM 308 System Programming

Sakarya University-Computer Engineering

Contents

a) Introduction to System Calls and I/O

b) Cat/Buffering

- http://web.eecs.utk.edu/~jplank/plank/classes/cs360/360/notes/Syscall-Intro/lecture.html
- https://web.eecs.utk.edu/~jplank/plank/classes/cs360/360/notes/Cat/lecture.html

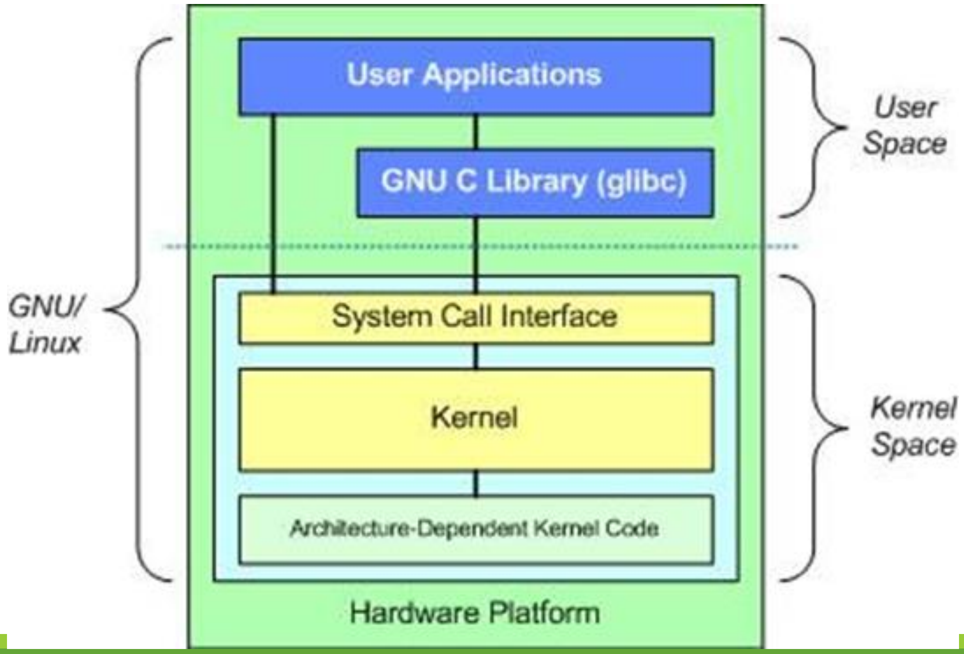
System Calls

- The way that programs talk to the operating system is via ``*system calls*.''
- A system call looks like a procedure call but it's different -- **it is a request to the operating system to perform some activity**.
- System calls are expensive. A procedure call can usually be performed in a few machine instructions.
- A system call requires the computer to save its state, let the operating system take control of the CPU, have the operating system perform some function, have the operating system save its state, and then have the operating system give control of the CPU back to you.

Some examples:

- signal()
- getpid()
- kill()
- stat()
- fork()
- read()

…



Source: http://www.ibm.com/developerworks/library/l-linux-kernel/

System Calls

- When a computer is turned on, the first program that runs is called the "operating system".
- It controls almost all activities on the computer.
- This includes who is logged in, how disks are used, how memory is used, how the CPU is used, and how you talk to other computers.
- The way programs talk to the operating system is through ``system calls''.
- A system call looks like a procedure call, but it is different -- it is a **request for the operating system to perform some activity.**
- System calls are expensive. While a function call can usually be accomplished with a few machine instructions, a system call requires the computer to save its state, the operating system to take control of the CPU, the operating system to perform some function, the operating system to save its state.
- then makes the operating system give you back control of the CPU. (User mode , Kernel mode)

System calls

There are 5 basic system calls that Unix provides for file I/O.

```
1.  int open(const char *path, int flags [ , int mode ] );
2.  int close(int fd);
3.  ssize_t read(int fd, void *buf, size_t count);
4.  ssize_t write(int fd, const void *buf, size_t count);
5.  off_t lseek(int fd, off_t offset, int whence);
```

- You'll notice that they look like normal procedure calls.
- That's how you program with them -- just like normal procedure calls.
- But you should know that they are different:
- **A system call** makes a request to the operating system.
- **A procedure call** jumps to a procedure defined elsewhere in your program.
- This procedure call itself may make a system call (for example, **fopen ()** calls **open ()), but it is a different call.**

System calls

- The reason the OS controls I/O is for security
- The computer must ensure that if there is a bug in my program, it does not crash the system and corrupt other people's programs.
- Work simultaneously or later.
- So when you do disk or display or network I/O, you must go through the operating system and use system calls.
- These five system calls are fully described in the man pages (do ' man -s 2 open ', ' man -s 2 close ', etc.).
- All those annoying types like ssize_t and off_t are ints and longs . They used to be all int , but as machines and files grew, so did they.

open

- **Open** requests the operating system to use a file.
- The 'Path ' argument specifies which file you want to use, and the ' flags ' and ' mode ' arguments specify how you want to use it.
- If the operating system approves your request, it returns you a file descriptor .
- This is a non-negative integer. If it returns -1, your access has been denied and you need to check the value of the " errno " variable to determine why.
- All operations you perform on the files will be done through the operating system.
- When you want to do file I/O directly with the operating system, you specify the file by file descriptor.
- Therefore, when you want to do file I/O on a particular file, you must first open that file to get a file handle.

int **open**(char *path, int flags [, int mode]);

- If the operating system approves your request, it will return a ``file descriptor'' to you.
- This is a non-negative integer. If it returns -1, then you have been denied access, and you have to check the value of the variable "errno" to determine why. (That or use perror()).
- All actions that you will perform on files will be done through the operating system.
- Whenever you want to do file I/O, you specify the file by its file descriptor. You must first open that file to get a file descriptor.

Open makes a request to the operating system to use a file.

Path: specifies what file you would like to use.

Flags, Mode: specify how you would like to use the file

Open

mode is required if file is created, ignored otherwise.

mode specifies the protection bits, e.g. 0644 = rw-r--r--

0 (octal notation) –User-Group- Other

0 6 4 4

0644 is the most typical value -- it says "I can read and write it; everyone else can only read it.

0755 :"I can read, write and execute it; and everyone else can only read and execute it.

rwx	oct	meaning
001	01	= execute
010	02	= write
011	03	= write & execute
100	04	= read
101	05	= read & execute
110	06	= read & write
111	07	= read & write & execute

open

- Example: src /o1.c opens the file txt/in1.txt for reading and prints the value of the file descriptor.
- If txt/in1.txt does not exist or you do not have permission to open it, it prints -1 because the open () call failed.
- If txt/in1.txt exists, it prints 3 which means open () request is granted

```
/* This program opens the file "txt/in1.t
   return value of the open() system cal
   non-negative integer (three).  If "txt

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

int main()
{
  int fd;

  fd = open("txt/in1.txt", O_RDONLY);
  printf("%d\n", fd);
  return 0;
}
```

open

❑* mv [source file] [target directory] (move file/directory)

❑Pay attention to the value of flags -- the man page for open () will give you an explanation of flags and how they work. so we need to know what O_RDONLY and all really mean).

❑Here are a few examples of calling bin/o1.

❑Initially, I have a file called txt/in1.txt in my directory, so the open () call succeeded and returned 3.

❑I then renamed it to tmp.txt and now the open () call fails, returning -1. I renamed it and the open () call was successful again and returned 3.

```
UNIX> ls -l txt/in1.txt
-rw-r--r--  1 plank  staff  22 Jan 31 12:50 txt/in1.txt
UNIX> bin/o1
3                                      # The open call succeeds here.
UNIX> mv txt/in1.txt tmp.txt
UNIX> bin/o1
-1                                     # The open call fails here.
UNIX> mv tmp.txt txt/in1.txt
UNIX> bin/o1
3                                      # The open call succeeds again.
UNIX>
```

open

- Second example: src/o2.c tries to open the file "txt/out1.txt" for writing.

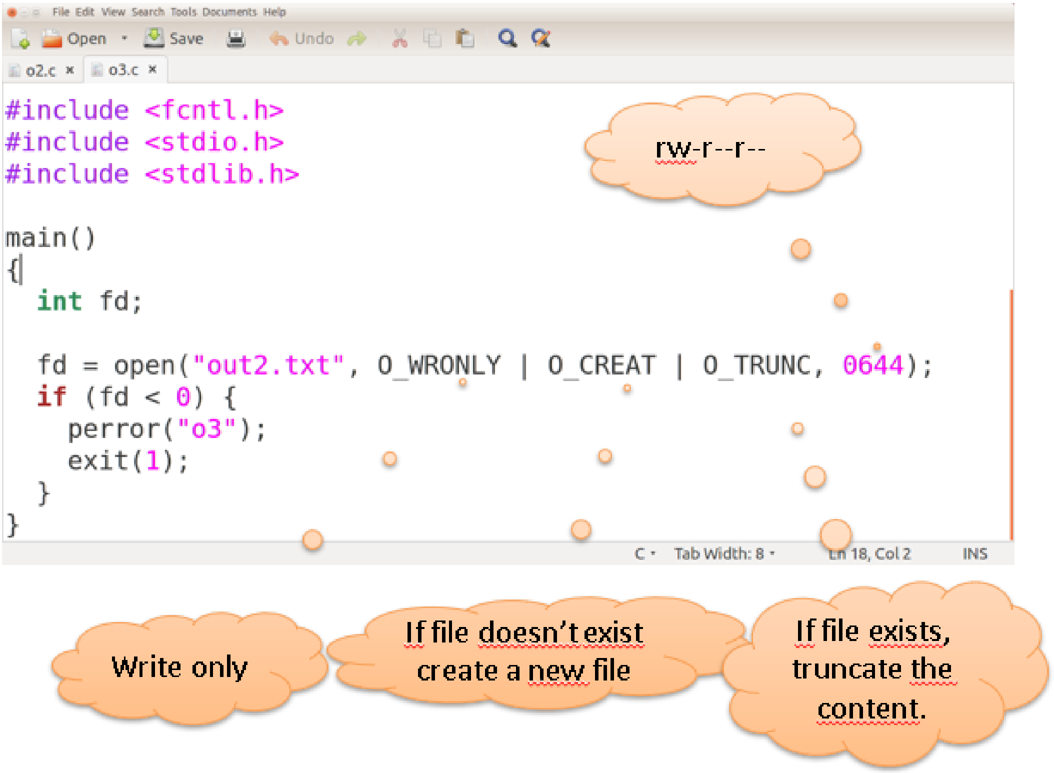
- This fails because txt/out1.txt does not exist anyway. Here's the code -- note that it uses perror() to print why the error occurred .

```
/* This program attempts to open the file
   directory.  Note that this fails becaus
   See src/o3.c for an example of opening

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

int main()
{
  int fd;

  fd = open("txt/out1.txt", O_WRONLY);
  if (fd < 0) {
    perror("txt/out1.txt");
    exit(1);
  }
  return 0;
}
```

```
UNIX> ls -l txt                                              # As you can see, there's no txt/out1.txt
total 8
-rw-r--r--  1 plank  staff  22 Jan 30  2018 in1.txt
-rw-r--r--  1 plank  staff   0 Jan 30  2018 out2.txt
UNIX> bin/o2                                                 # Accordingly, then open() call fails.
txt/out1.txt: No such file or directory
UNIX> echo Hi > txt/out1.txt                                 # I create txt/out1.txt
UNIX> bin/o2                                                 # And now the open() call succeeds
UNIX> cat txt/out1.txt                                       # The program did not change the file.
Hi
UNIX> chmod 0400 txt/out1.txt                                # Here I change the permissions so that I can't open for writing.
UNIX> bin/o2                                                 # And the open() call fails.
txt/out1.txt: Permission denied
UNIX> chmod 0644 txt/out1.txt
UNIX> rm txt/out1.txt                                        # I remove the file
UNIX> bin/o2                                                 # And the open() call fails again.
txt/out1.txt: No such file or directory
UNIX>
```

open

- To open a new file for writing, you must open it with (O_WRONLY | O_CREAT | O_TRUNC) as the flags argument.
- O_CREAT tells you to create the file if it doesn't already exist.
- O_TRUNC tells it, if the file exists, to " truncate " it to zero bytes and delete what's there.



Open examples

```
UNIX> ls -l txt/out2.txt                                     # txt/out2.txt has zero bytes and was last changed in 2018
-rw-r--r--  1 plank  staff  0 Jan 30  2018 txt/out2.txt
UNIX> bin/o3
UNIX> ls -l txt/out2.txt                                     # It still has zero bytes, but the modification time has updated.
-rw-r--r--  1 plank  staff  0 Feb  3 14:56 txt/out2.txt
UNIX> rm txt/out2.txt
UNIX> bin/o3
UNIX> ls -l txt/out2.txt                                     # Now it created the file anew.
-rw-r--r--  1 plank  staff  0 Feb  3 14:57 txt/out2.txt
UNIX> echo "Hi" > txt/out2.txt
UNIX> ls -l txt/out2.txt                                     # The echo command has put "Hi" and a newline into the file.
-rw-r--r--  1 plank  staff  3 Feb  3 14:57 txt/out2.txt
UNIX> bin/o3
UNIX> ls -l txt/out2.txt                                     # bin/o3 has truncated the file.
-rw-r--r--  1 plank  staff  0 Feb  3 14:57 txt/out2.txt
UNIX> echo "Hi Again" > txt/out2.txt
UNIX> chmod 0400 txt/out2.txt
UNIX> ls -l txt/out2.txt                                     # I have put 9 bytes into the file using echo, but the permission is read-only.
-r--------  1 plank  staff  9 Feb  3 14:57 txt/out2.txt
UNIX> bin/o3                                                 # As such, bin/o3 fails to open the file.
txt/out2.txt: Permission denied
UNIX> ls -l txt/out2.txt                                     # And the file is unchanged.
-r--------  1 plank  staff  9 Feb  3 14:57 txt/out2.txt
UNIX> chmod 0644 txt/out2.txt
UNIX> bin/o3
UNIX> ls -l txt/out2.txt                                     # When I change the permissions back to R/W, bin/o3 truncates the file again.
-rw-r--r--  1 plank  staff  0 Feb  3 14:58 txt/out2.txt
```

Close

- Tells the operating system that you are done with a file descriptor .
- The operating system can then reuse this file descriptor.
- The src /c1.c program shows some examples of opening and closing the txt/in1.txt file.
- You have to look carefully as it opens the file multiple times without closing it, which is perfectly legal on Unix.
- **Example: src /c1.c**

```
UNIX> bin/c1
Opened the file txt/in1.txt twice:  Fd's are 3 and 4.
Closed both fd's.
Reopened txt/in1.txt into fd2: 3.
Closed fd2.  Now, calling close(fd2) again.
This should cause an error.

c1: Bad file descriptor
UNIX>
```

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

main()
{
  int fd1, fd2;

  fd1 = open("in1.txt", O_RDONLY);
  if (fd1 < 0) { perror("c1"); exit(1); }

  fd2 = open("in1.txt", O_RDONLY);
  if (fd2 < 0) { perror("c1"); exit(1); }

  printf("Opened the file in1.txt twice:  Fd's are %d and %d

  if (close(fd1) < 0) { perror("c1"); exit(1); }
  if (close(fd2) < 0) { perror("c1"); exit(1); }

  printf("Closed both fd's.\n");
```

Open in1.txt

Open in1.txt again

Close the file descriptors

```
Syscall-Intro: gcc -o test c1.c
Syscall-Intro: ./test
Opened the file in1.txt twice:  Fd's are 3 and 4.
Closed both fd's.
```

```c
  printf("Closed both fd's.\n");

  fd2 = open("in1.txt", O_RDONLY);
  if (fd2 < 0) { perror("c1"); exit(1); }

  printf("Reopened in1.txt into fd2: %d.\n", fd

  if (close(fd2) < 0) { perror("c1"); exit(1); }

  printf("Closed fd2.  Now, calling close(fd2) again.\n");
  printf("This should cause an error.\n\n");

  if (close(fd2) < 0) { perror("c1"); exit(1); }
}
```

Open in1.txt again

Close in1.txt

Close in1.txt again

```
Closed both fd's.
Reopened in1.txt into fd2: 3.
Closed fd2.  Now, calling close(fd2) again.
This should cause an error.

c1: Bad file descriptor
```

```
Syscall-Intro: gcc -o test c1.c
Syscall-Intro: ./test
Opened the file in1.txt twice:  Fd's are 3 and 4.
Closed both fd's.
Reopened in1.txt into fd2: 3.
Closed fd2.  Now, calling close(fd2) again.
This should cause an error.

c1: Bad file descriptor
Syscall-Intro:
```

We opened in1.txt again, to see that it will reuse the first file descriptor

Close the file descriptor twice. The second causes an error.

Read

- Read() tells the operating system to read the "size" bytes from the file opened at the file descriptor «fd» and put those bytes in the location pointed to by " buf ".
- Returns how many bytes were actually read.
- **Example: src /r1.c**

```
UNIX> bin/r1
called read(3, c, 10).  returned that 10 bytes  were read.
Those bytes are as follows: Jim Plank

called read(3, c, 99).  returned that 12 bytes  were read.
Those bytes are as follows: Claxton 221

UNIX>
```

Read

- There are a few things to note about this program. First, **the buf** should point to valid memory.
- src /r1.c this is achieved by malloc ()- ing space for c. Alternatively, I could declare c as a static array of 100 characters: char c[100];
- **Second** , I terminate c after read () calls to make sure printf () understands .
- This is important -- there are no NULL characters in text files. When read () reads them, it does not terminate NULL.
- characters as strings in C , **you must terminate them as NULL.**

Read

- **Third, when** read () returns 0, the end of the file has been reached.
- When reading from a file, if read () returns fewer bytes than you want, you've reached the end of the file. This is what happens on the second read() in src /r1.c.
- **Fourth** , notice that the 10th character in the first read () call and the 12th character in the second are both newline characters.
- That's why you get two newlines in the printf () statement. One is in c and the other is in the printf () statement.
- To reiterate, the read call does not read a NULL character. It only reads bytes from the file and the file does not contain any NULL characters. Therefore, you need to explicitly put the NULL character in your string.

Read

- Let's look at a similar program that does not terminate NULL (src /r2.c):

```
/* Showing what happens when you don't NULL terminate. */

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

int main()
{
  char c[100];
  int fd;

  strcpy(c, "ABCDEFGHIJKLMNOPQRSTUVWXYZ");
  fd = open("txt/in1.txt", O_RDONLY);
  if (fd < 0) { perror("r1"); exit(1); }

  read(fd, c, 10);             /* I read 10 bytes, but I don't null terminate. */
  printf("%s\n", c);           /* So this printf() will print the characters from K to Z. */

  read(fd, c, 99);             /* This reads 12 bytes, so it prints M to Z. */
  printf("%s\n", c);

  return 0;
}
```

```
UNIX> bin/r2
Jim Plank
KLMNOPQRSTUVWXYZ
Claxton 221
MNOPQRSTUVWXYZ
UNIX>
```

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

main()
{
  char *c;
  int fd, sz;

  c = (char *) calloc(100, sizeof(char));

  fd = open("in1.txt", O_RDONLY);
  if (fd < 0) { perror("r1"); exit(1); }

  sz = read(fd, c, 10);

  printf("called read(%d, c, 10)."
         "returned that %d bytes  were read.\n",fd, sz);
```

Allocate 100 characters.

Read 10 characters

Null termination

Read 99 characters.

```c
         "returned that %d bytes  were read.\n",fd, sz);

  c[sz] = '\0';

  printf("Those bytes are as follows: %s\n", c);

  sz = read(fd, c, 99);

  printf("called read(%d, c, 99)."
         "returned that %d bytes  were read.\n",fd, sz);

  c[sz] = '\0';

  printf("Those bytes are as follows: %s\n", c);

  close(fd);
}
```

Write

- Write() is just like read (), only it writes bytes instead of reading them. Returns the number of bytes actually written, which is always "size".

```c
/* This program opens the file "out3.txt" in the current directory
   for writing, and writes the string "cs360\n" to it. */

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

int main()
{
  int fd, sz;

  fd = open("txt/out3.txt", O_WRONLY | O_CREAT | O_TRUNC, 0644);
  if (fd < 0) { perror("txt/out3.txt"); exit(1); }

  sz = write(fd, "cs360\n", strlen("cs360\n"));

  printf("called write(%d, \"cs360\\n\", %ld).  it returned %d\n",
         fd, strlen("cs360\n"), sz);

  close(fd);
  return 0;
}
```

```
UNIX> bin/w1
called write(3, "cs360\n", 6).  it returned 6
UNIX> cat txt/out3.txt
cs360
UNIX>
```

Write

- You should consider different combinations of O_CREAT and O_TRUNC and their effects on typing.
- specifically at src /w2.c. This allows you to specify the combination of O_WRONLY, O_CREAT and O_TRUNC that you use in your open () call:
- **src /w2.c**

Write

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

main()
{
  int fd, sz;

  fd = open("out3.txt", O_WRONLY | O_CREAT | O_TRUNC, 0644);
  if (fd < 0) { perror("r1"); exit(1); }

  sz = write(fd, "cs360\n", strlen("cs360\n"));

  printf("called write(%d, \"cs360\\n\", %ld).  it returned %d\n",
         fd, strlen("cs360\n"), sz);

  close(fd);
}
```

```
Syscall-Intro: gcc -o test w1.c
Syscall-Intro: ./test
called write(3, "cs360\n", 6).  it returned 6
Syscall-Intro: cat out3.txt
cs360
Syscall-Intro:
```

cat - concatenate files and print on the standard output

lseek

- All open files have a "file pointer" associated with them .
- When the file is opened, the file pointer points to the beginning of the file.
- As the file is read or written, the file pointer moves.
- For example, after the first read in r1.c, the file pointer points to the 11th byte in txt/in1.txt.
- You can manually move the file pointer with lseek ().

```
LSEEK(2)            Linux Programmer's Manual            LSEEK(2)

NAME
       lseek - reposition read/write file offset

SYNOPSIS
       #include <sys/types.h>
       #include <unistd.h>

       off_t lseek(int fd, off_t offset, int whence);

DESCRIPTION
       lseek()  repositions  the  file offset of the open file
       description associated with the file descriptor  fd  to
       the  argument  offset according to the directive whence
       as follows:

       SEEK_SET
              The file offset is set to offset bytes.

       SEEK_CUR
              The file offset is set to its  current  location
              plus offset bytes.

       SEEK_END
              The  file  offset is set to the size of the file
              plus offset bytes.
```

lseek

- lseek's ' -whence -from' variable specifies how to search, starting from the beginning of the file, the current value of the pointer, and the end of the file.
- The return value is the position of the pointer after lseek .
- src /l1.c. It does a bunch of searching in txt/in1.txt.
- Watch and make sure everything makes sense.
- sys / types.h and unistd.h ? I typed " man -s 2 lseek ".
- **src /l1.c**

Standard Input , Output and Error

- Now every process in Unix starts with three file descriptors already defined and open:
  - File descriptor 0 is standard input.
  - File descriptor 1 is standard output.
  - File descriptor 2 is standard error.
- So, when writing a program, you can read from standard input using read (0, ...) and write to standard output using write(1, ...).

```
#include <sys/types.h>
#include <unistd.h>
#include <fcntl.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
main()
{
  char *c;
  int fd, sz, i;

  c = (char *) calloc(100, sizeof(char));
  fd = open("in1.txt", O_RDONLY);
  if (fd < 0) { perror("r1"); exit(1); }

  sz = read(fd, c, 10);
  printf("We have opened in1.txt,"
        "and called read(%d, c, 10).\n", fd);
  printf("It returned that %d bytes"
        "  were read.\n", sz);
  c[sz] = '\0';
  printf("Those bytes are as follows: %s\n", c);
```

For example, in r1.c, after the first read, the file pointer points to the 11th byte in in1.txt.

Read 10 characters (byte) from file.

```
  printf("Those bytes are as follows: %s\n",

  i = lseek(fd, 0, SEEK_CUR);
  printf("lseek(%d, 0, SEEK_CUR) returns that the"
        " current offset of the file is %d\n\n", fd, i);

  printf("now, we seek to the beginning "
        "of the file and call read(%d, c, 10)\n",fd);
```

Saving file '/home/bilg/Documents/sisprog/Syscall-Intro/l1.c'...   C   Tab Width: 8   Ln 39, Col 1   INS

Current position of the file pointer

```c
lseek(fd, 0, SEEK_SET);
sz = read(fd, c, 10);
c[sz] = '\0';
printf("The read returns the following bytes: %s\n", c);

printf("now, we do lseek(%d, -6, SEEK_END)."
               "  It returns %ld\n",
               fd, lseek(fd, -6, SEEK_END));

printf("If we do read(%d, c, 10),"
      "we get the following bytes: ", fd);
```

Beginning of the file

Go back 6 characters from the end of the file

```c
sz = read(fd, c, 10);
c[sz] = '\0';
printf("%s\n", c);

printf("Finally, we do lseek(%d, -1, SEEK_SET)."
               "  This return s -1.\n", fd);
printf("perror() tells us why: ");
fflush(stdout);

i = lseek(fd, -1, SEEK_SET);
perror("l1 :");
}
```

Read 10 characters (bytes) from the last file location.

Position the file pointer to the beginning and try to read one character behind. (error)

Standard Input, Output and Error

Now, every process in Unix starts out with three file descriptors predefined and open:

File descriptor 0 is standard input.

File descriptor 1 is standard output.

File descriptor 2 is standard error.

Thus, when you write a program, you can read from standard input, using read(0, ...), and write to standard output using write(1, ...).

```
#include <unistd.h>

int main()
{
  char c;

  while (read(0, &c, 1) == 1) write(1, &c, 1);
  return 0;
}
```

```
UNIX> bin/simpcat < txt/in1.txt
Jim Plank
Claxton 221
UNIX>
```

simpcat

- Three equivalent ways to write a simple cat program that reads from standard input and writes to standard output.

src/simpcat1.c	src/simpcat2.c	src/simpcat3.c

src/simpcat1.c

```c
/* Using getchar()/putchar(). */

#include <stdio.h>
#include <fcntl.h>
#include <stdio.h>

int main()
{
  char c;

  c = getchar();
  while(c != EOF) {
    putchar(c);
    c = getchar();
  }
  return 0;
}
```

src/simpcat2.c

```c
/* Using read()/write(). */

#include <unistd.h>

int main()
{
  char c;

  while(read(0, &c, 1) == 1) {
    write(1, &c, 1);
  }
  return 0;
}
```

src/simpcat3.c

```c
/* Using fread()/fwrite(). */

#include <stdio.h>

int main()
{
  char c;
  int i;

  i = fread(&c, 1, 1, stdin);
  while(i > 0) {
    fwrite(&c, 1, 1, stdout);
    i = fread(&c, 1, 1, stdin);
  }
  return 0;
}
```

Test results of Simpcat function

```
abc:2_Cat$ time ./simpcat1 <large.txt> /dev/null

real    0m1.640s
user    0m1.373s
sys     0m0.008s
abc:2_Cat$ time ./simpcat2 <large.txt> /dev/null

real    0m42.368s
user    0m6.398s
sys     0m32.746s
abc:2_Cat$ time ./simpcat3 <large.txt> /dev/null

real    0m5.290s
user    0m4.672s
sys     0m0.032s
```

Simpcat

- So, what's going on? /dev/null is a special file in Unix that you can write to, but it never stores anything on disk.
- We're using it so that you don't create 25M files in your home directory as this wastes disk space. "Large.txt" is a 25,000,000-byte file. This means that in simpcat1.c, getchar() and putchar() are being called 25 million times each, as are read() and write() in simpcat2.c, and fread() and fwrite() in simpcat3.c.
- Obviously, the culprit in simpcat2.c is the fact that the program is making system calls instead of library calls. Remember that a system call is a request made to the operating system. This means at each read/write call, the operating system has to take over the CPU (this means saving the state of the simpcat2 program), process the request, and return (which means restoring the state of the simpcat2 program).
- This is evidently far more expensive than what simpcat1.c and simpcat3.c do.

simpcat

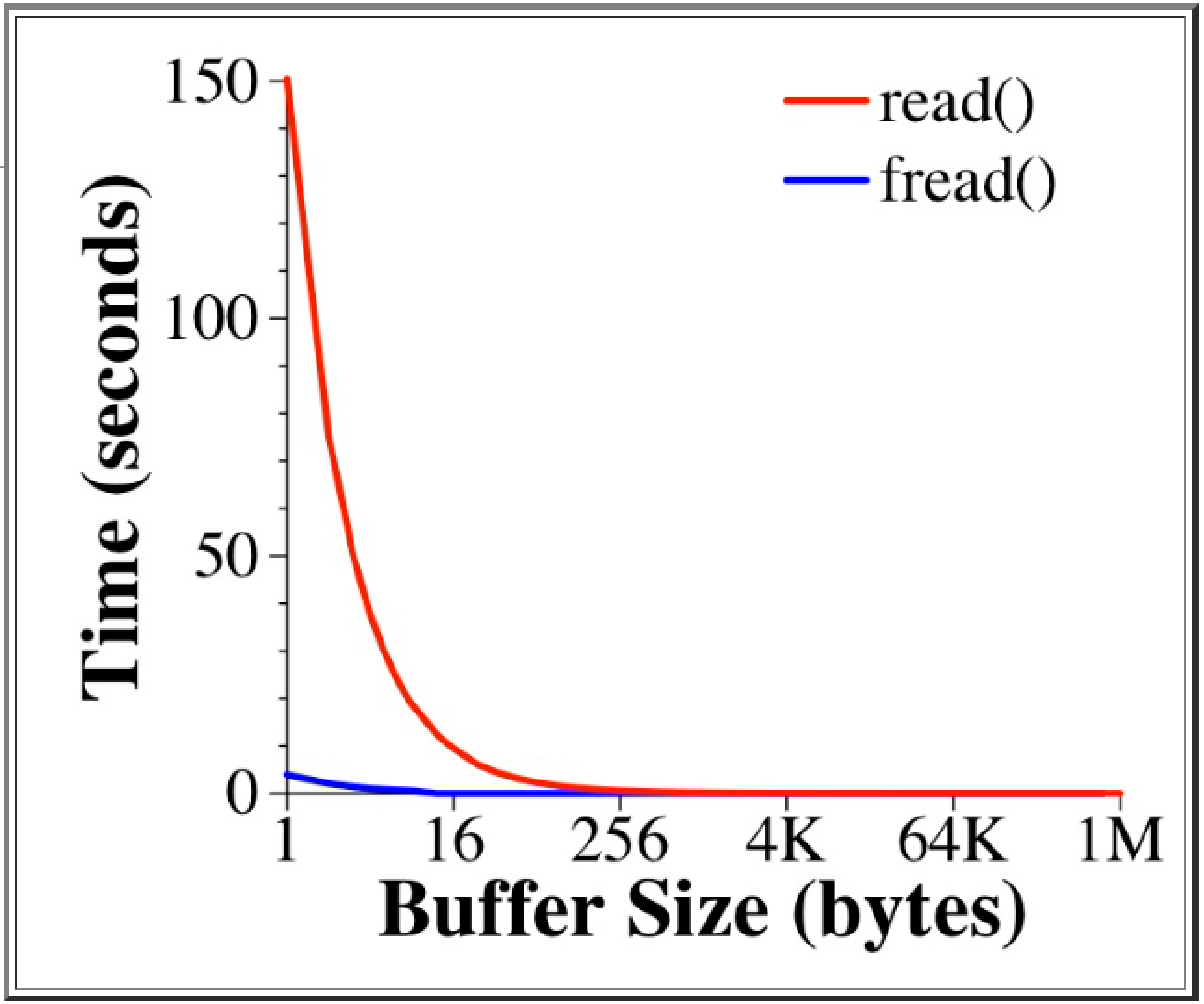
*time bin /simpcat4 128 < data/large.txt > /dev/ null

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

int main(int argc, char **argv)
{
  int bufsize;
  char *c;
  int i;

  bufsize = atoi(argv[1]);
  c = (char *) malloc(bufsize*sizeof(char));
  i = 1;
  while (i > 0) {
    i = read(0, c, bufsize);
    if (i > 0) write(1, c, i);
  }
  return 0;
}
```

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

int main(int argc, char **argv)
{
  int bufsize;
  char *c;
  int i;

  bufsize = atoi(argv[1]);
  c = (char *) malloc(bufsize*sizeof(char));
  i = 1;
  while (i > 0) {
    i = fread(c, 1, bufsize, stdin);
    if (i > 0) fwrite(c, 1, i, stdout);
  }
  return 0;
}
```

simpcat

- These let us read in more than one byte at a time.
- This is called buffering: You allocate a region of memory in which to store things, so that you can make fewer system/procedure calls.
- Note that fread() and fwrite() are just like read() and write(), except that they go to the standard I/O library instead of the operating system.



simpcat

- First, what can we now infer about the standard I/O library? Buffering is used!
- In other words, when you first call getchar () or fread (), it performs a read() of a large number of bytes into a buffer.
- So subsequent calls to getchar () or fread () will be fast. When you try to fread () large memory segments , the two exhibit the same behavior in that fread () doesn't need to be buffered -- it just calls read ().
- So why is getchar () faster than fread (c, 1, 1, stdin)?
- <u>Because getchar () is optimized for reading one character and fread () is not.</u>
- Think about it -- fread () needs to read four arguments, and if it's executing code for small values of the size, it needs to at least figure out that the size is small before executing the code. getchar () is written to be really fast for single characters.

simpcat

What's the lesson behind this?

1. Buffering is a good way to cut down on too many system calls.

2. If you are reading small chunks of bytes, then use **getchar()** or **fread()**. They do buffering for you.

3. If you are doing single character I/O, use **getchar()** (or **fgetc()**).

4. If you are reading large chunks of bytes, then **fread()** and **read()** work about the same. However, you should use **fread()**, since it makes your programming more consistent, and because it does a little more error checking for you.

Standard I/O vs. System calls .

- System calls work with integer file descriptors .
- Standard I/O calls define a structure called FILE and work with pointers to those structures. It can be used in code optimization.

```
System Call                    Standard I/O call
-----------                    -----------------
open                           fopen
close                          fclose
read/write                     getchar/putchar
                               getc/putc
                               fgetc/fputc
                               fread/fwrite
                               gets/puts
                               fgets/fputs
                               scanf/printf
                               fscanf/fprintf
lseek                          fseek
```

simpcat

- For example, here are the versions of the cat program that should be called with the file name as arguments.

To exemplify, the following are versions of the program **cat** which must be called with **filename** as their arguments. [Cat1.c] uses system calls, and [cat2.c] uses the standard I/O library. Both use an 8K buffer for the **read()/fread()** and **write()/fwrite()** calls. Read the man page for **open** ("man 2v open") and **fopen** ("man 3s fopen") to understand their arguments.

Try:

```
UNIX> sh -c "time bin/cat1 data/large.txt > /dev/null"        # As you can see,

real    0m0.010s
user    0m0.003s
sys     0m0.006s
UNIX> sh -c "time bin/cat2 data/large.txt > /dev/null"        # Their performance is the same.

real    0m0.010s
user    0m0.003s
sys     0m0.006s
UNIX>
```

simpcat

- Finally, src / fullcat.c contains a version of cat that is very similar to the real version -- if you omit a filename, it prints standard input to standard output.

https://web.eecs.utk.edu/~jplank/plank/classes/cs360/360/notes/Cat/src/fullcat.c

- Otherwise, it prints every file specified in the command line arguments. Notice how it resembles both simpcat1.c and cat2.c.

Chars/ ints

- You'll notice that getchar () is defined to return an int , not a character. Relatedly, see simpcat1a.c:

- The only difference between simpcat1a.c and simpcat1.c is that c is an int instead of a character . Now, why would this matter? See below

```
UNIX> ls -l src/simpcat1.c bin/simpcat1
-rwxr-xr-x  1 plank  staff  12604 Feb  5 12:17 bin/simpcat1
-rw-r--r--  1 plank  staff    466 Feb  5 12:15 src/simpcat1.c
UNIX> bin/simpcat1 > tmp1.txt
^C
UNIX> bin/simpcat1 < bin/simpcat1 > tmp1.txt
UNIX> bin/simpcat1 < src/simpcat1.c > tmp2.txt
UNIX> ls -l tmp1.txt tmp2.txt
-rw-r--r--  1 plank  staff  3919 Feb  7 23:37 tmp1.txt
-rw-r--r--  1 plank  staff   466 Feb  7 23:38 tmp2.txt
UNIX>
```

Notice anything wierd? Now:

```
UNIX> bin/simpcat1a < bin/simpcat1 > tmp3.txt
UNIX> ls -l tmp3.txt
-rw-r--r--  1 plank  staff  12604 Feb  7 23:38 tmp3.txt
UNIX>
```

```
int main()
{
  int c;

  c = getchar();
  while(c != EOF) {
    putchar(c);
    c = getchar();
  }
  return 0;
}
```

```
int main()
{
  char c;

  c = getchar();
  while(c != EOF) {
    putchar(c);
    c = getchar();
  }
  return 0;
}
```

Simpcat1 uses a char to copy character. When a byte that contains 255 is read, it is recorded as -1 which means EOF. This breaks while loop and copying stops.

```
abc:2_Cat$ ./simpcat1 < simpcat3 > file1
abc:2_Cat$ ls -l file1
-rw-r--r-- 1 abc abc 650 Apr  7 15:23 file1
abc:2_Cat$ ./simpcat1a < simpcat3 > file1
abc:2_Cat$ ls -l file1
-rw-r--r-- 1 abc abc 10976 Apr  7 15:23 file1
```

*This has to do with what happens when getchar () reads 255 characters.