# EECS 440 System Design of a Search Engine

Winter 2021

Lecture 7: Intro to the filesystem

Nicole Hamilton
https://web.eecs.umich.edu/~nham/
nham@umich.edu

# Topics

1. Course details.
2. Handles and ids.
3. Reading stdin, writing stdout.
4. Opening files.
5. Seeking.
6. Memory-mapping files.
7. Walking the directory structure.

# Topics

1. Course Details
2. Handles and ids.
3. Reading stdin, writing stdout.
4. Opening files.
5. Seeking.
6. Memory-mapping files.
7. Walking the directory structure.

# Course details

1. Will try to begin scheduling meetings with the teams next week.

# Topics

1. Course Details.
2. Handles and ids.
3. Reading stdin, writing stdout.
4. Opening files.
5. Seeking.
6. Memory-mapping files.
7. Walking the directory structure.

# cin, cout and cerr

All the streams you're used to using in C++ are built on top of underlying operating system calls.

| | |
|---|---|
| Header | unistd.h |
| Open a file | open( ) |
| Read | read( ) |
| Write | write( ) |
| Close the file | close( ) |

# Handles

On Linux, often called an ID.

Means of referring to system objects.

Opaque datatype

- “Black boxes”
- Not fair to look at the value and try to guess how it’s encoded.

Handles can be inherited or duplicated (even to another process.)

# Inherited handles

| | |
|---|---|
| Stdin | 0 |
| Stdout | 1 |
| Stderr | 2 |

# Intro to the filesystem

1. Course Details.
2. Group photos.
3. Cartoon characters.
4. Handles and ids.
5. Reading stdin, writing stdout.
6. Opening files.
7. Seeking.
8. Memory-mapping files.
9. Walking the directory structure.

# read( ) and write( )

```
// read() attempts to read up to count bytes from
// file descriptor fd into the buffer starting at buf.

ssize_t read(int fd, void *buf, size_t count);

// write() writes up to count bytes from the
// buffer starting at buf to the file referred to
// by the file descriptor fd.

ssize_t write(int fd, const void *buf, size_t count);
```

```
// Simple Linux cat from stdin.

#include <unistd.h>

const ssize_t BufferSize = 4096;

int main( )
   {
   char buffer[ BufferSize ];
   ssize_t bytesRead, bytesWritten;

   do
      {
      bytesRead = read( 0, buffer, sizeof( buffer ) );
      if ( bytesRead > 0 )
         bytesWritten = write( 1, buffer, bytesRead );
      }
   while ( bytesRead > 0 && bytesWritten == bytesRead );
   }
```

# Intro to the filesystem

1. Course Details.
2. Group photos.
3. Cartoon characters.
4. Handles and ids.
5. Reading stdin, writing stdout.
6. Opening files.
7. Seeking.
8. Memory-mapping files.
9. Walking the directory structure.

# Opening and closing files

| | |
|---|---|
| Header | unistd.h |
| Open a file | open( ) |
| Read | read( ) |
| Write | write( ) |
| Close the file | close( ) |

# open( ) parameters

```
int open(const char *pathname, int flags);
int open(const char *pathname, int flags, mode_t mode);
```

Flags are bitwise OR'ed, must include

`O_RDONLY, O_WRONLY or O_RDWR`

Plus optionally:

`O_APPEND, O_CREAT, O_TRUNC` + others

If `O_CREAT` specified, mode bits specify the Linux rwx (read/write/execute) bits to be applied to any new file.

```
// Simple Linux cat file routine.

#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>

const ssize_t BufferSize = 4096;

int main( int argc, char **argv )
   {
   char buffer[ BufferSize ];
   ssize_t bytesRead, bytesWritten;
   int input = open( argv[ 1 ], O_RDONLY );

   do
      {
      bytesRead = read( input, buffer, sizeof( buffer ) );
      if ( bytesRead > 0 )
         bytesWritten = write( 1, buffer, bytesRead );
      }
   while ( bytesRead > 0 && bytesWritten == bytesRead );

   close( input );
   }
```

# Agenda

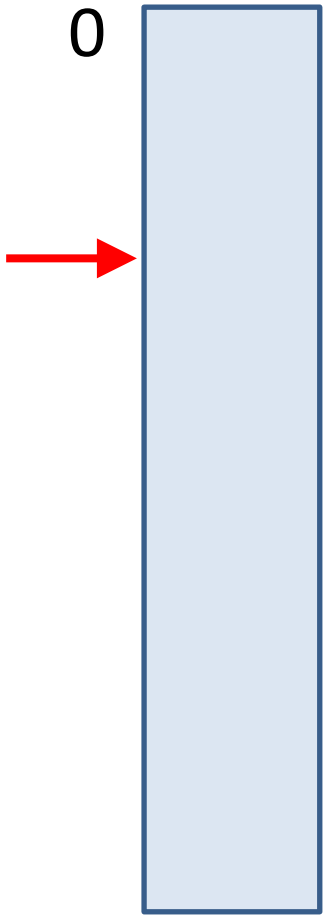
1. Course Details.
2. Group photos.
3. Cartoon characters.
4. Handles and ids.
5. Reading stdin, writing stdout.
6. Opening files.
7. Seeking.
8. Memory-mapping files.
9. Walking the directory structure.

# Seeking

As we read or write a file, we generally think of starting at the beginning, then reading or writing from there.

We have a current location that follows us.

We can reset that position by seeking.




Bytestream that starts at 0 and runs the end of the file.

# Linux

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      Offset is set to offset bytes.
SEEK_CUR      Current location plus offset.
SEEK_END      Size of the file plus offset.
```

off_t is a signed type. off_t = -1 indicates failure, errno set to indicate the error.

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

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

```
$ head -1 LinuxSeekRead.cpp
// Simple Linux file seek and read example.
$ g++ LinuxSeekRead.cpp -o LinuxSeekRead
$ ./LinuxSeekRead
Usage:  LinuxSeekRead filename position bytes
$ ./LinuxSeekRead LinuxSeekRead.cpp 2 30; echo
 Simple Linux file seek and re
$
```

```
// Simple Linux file seek and read example.

// Nicole Hamilton  nham@umich.edu

#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>
#include <stdlib.h>
#include <iostream>
using namespace std;



int main( int argc, char **argv )
   {
   if ( argc != 4 )
      {
      cerr << "Usage:  LinuxSeekRead filename position bytes" << endl;
      return 1;
      }

   int file = open( argv[ 1 ], O_RDONLY );

   if ( file == -1 )
      {
      cerr << "Could not open " << argv[ 1 ] << ", error = " << errno;
      return errno;
      }
```

```
unsigned int position = atoi( argv[ 2 ] ),
    bytes = atoi( argv[ 3 ] );

off_t resultPosition = lseek( file, position, SEEK_SET );

if ( resultPosition == -1 )
   {
   cerr << "Could not seek to " << position <<
      ", error = " << errno;
   return errno;
   }

if ( bytes > 0 )
   {
   char *buffer = new char[ bytes ];
   ssize_t bytesRead;
   if ( bytesRead = read( file, buffer, bytes ) )
      write( 1, buffer, bytesRead );
   else
      {
      cerr << "Could not read " << bytes << " bytes at position " <<
            position << ", error = " << errno;
      return errno;
      }
   }
}
```

If you can seek and read,
you can seek and write.

```
$ head -1 LinuxSeekWrite.cpp
// Simple Linux file seek and write example.
$ g++ LinuxSeekWrite.cpp -o LinuxSeekWrite
$ ./LinuxSeekWrite
Usage:  LinuxSeekWrite filename position
$ echo Hello world how are you > foobar
$ ./LinuxSeekWrite foobar 6
friend
$ cat foobar
Hello friend
ow are you
$
```

```
// Simple Linux file seek and write example.

// Nicole Hamilton  nham@umich.edu

#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>
#include <stdlib.h>
#include <iostream>
using namespace std;


int main( int argc, char **argv )
   {
   if ( argc != 3 )
      {
      cerr << "Usage:  LinuxSeekWrite filename position" << endl;
      return 1;
      }
```

```
int file = open( argv[ 1 ], O_WRONLY | O_CREAT, S_IRUSR | S_IWUSR );

if ( file == -1 )
   {
   cerr << "Could not open " << argv[ 1 ] << ", error = " << errno;
   return errno;
   }

unsigned int position = atoi( argv[ 2 ] );

off_t resultPosition = lseek( file, position, SEEK_SET );

if ( resultPosition == -1 )
   {
   cerr << "Could not seek to " << position <<
      ", error = " << errno;
   return errno;
   }

char buffer[ 1024 ];
ssize_t bytesRead, bytesWritten;
while ( bytesRead = read( 0, buffer, sizeof( buffer ) ) )
   write( file, buffer, bytesRead );
}
```

# Seeking past the end

What do you expect should happen if you seek past the end?

How about if you're WAY past the end?

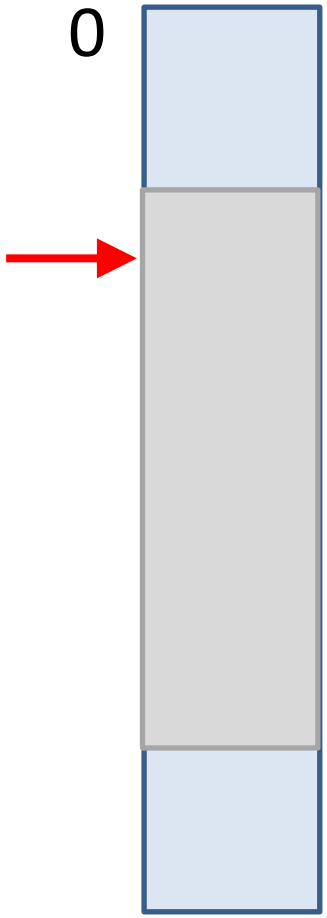
```
$ echo hello world > foobar
$ ls -l foobar
-rwxrwxrwx 3 nicole nicole 12 Jan 24 12:50 foobar
$ ./LinuxSeekWrite foobar 1000000
This is way past the end.
$ ls -l foobar
-rwxrwxrwx 3 nicole nicole 1000026 Jan 24 12:50 foobar
$ od -c foobar
0000000   h   e   l   l   o       w   o   r   l   d  \n  \0  \0  \0  \0
0000020  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0
*
3641100   T   h   i   s       i   s       w   a   y       p   a   s   t
3641120       t   h   e       e   n   d   .  \n
3641132
$
```

# Sparse files

Both Linux and Windows seamlessly support sparse files.

These are files with holes in them where there's no actual data.

Holes take no actual space on disk.




If you read from a hole, you get 0.

# Agenda

1. Course Details.
2. Group photos.
3. Cartoon characters.
4. Handles and ids.
5. Reading stdin, writing stdout.
6. Opening files.
7. Seeking.
8. Memory-mapping files.
9. Walking the directory structure.

# Problem with read and write

Annoying if you're doing any sort of scanning of data that might cross a read/write buffer boundary.

Example:  A simple "wc" (word count) example counts words, breaking on white space.

```
$ head -1 LinuxWcStd.cpp
// Plain vanilla Linux word-count.
$ g++ LinuxWcStd.cpp -o LinuxWcStd
$ ./LinuxWcStd Linux*cpp
346     LinuxCatMT.cpp
117     LinuxForkExec.cpp
594     LinuxGetSsl.cpp
397     LinuxGetUrl.cpp
75      LinuxHelloMT.cpp
460     LinuxListDirectory.cpp
305     LinuxPipe.cpp
201     LinuxSeekRead.cpp
162     LinuxSeekWrite.cpp
325     LinuxSignalDtor.cpp
298     LinuxSignalHandler.cpp
994     LinuxTinyServer.cpp
1087    LinuxTinySslServer.cpp
208     LinuxWcMap.cpp
173     LinuxWcStd.cpp
5742    Total
$
```

```
// Plain vanilla Linux word-count.
// Nicole Hamilton  nham@umich.edu

#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>
#include <stdlib.h>
#include <iostream>
using namespace std;

int main( int argc, char **argv )
   {
   int total = 0;

   for ( int i = 1;  i < argc;  i++ )
      {
      int words = 0;

      int f = open( argv[ i ], O_RDONLY );
```

```
if ( f != -1 )
   {
   bool midWord = false;
   ssize_t bytes;
   char buffer[ 4096 ], *c;

   while ( bytes = read( f, buffer, sizeof( buffer ) ) )
      for ( c = buffer;  c < buffer + bytes;  c++ )
         switch ( *c )
            {
            case ' ':
            case '\t':
            case '\n':
            case '\r':
               if ( midWord )
                  {
                  midWord = false;
                  words++;
                  }
               break;
            default:
               midWord = true;
            }

   if ( midWord )
      words++;

   close( f );
   cout << words << "\t" << argv[ i ]  << endl;
   total += words;
   }
```

*We iterate over each buffer-full.*

```
    }

if ( argc > 2 )
    cout << total << "\t" << "Total" << endl;
}
```

# Mapped files

Map a whole or part of a file into your address space.

You get a pointer to where it's been mapped and from there you treat it like a big array of chars.

Reading or writing the file only requires dereferencing a pointer into the map.

The operating system's paging system will populate the map as you touch locations.

```
$ head -1 LinuxWcMap.cpp
// Linux word-count using the mapped file support.
$ g++ LinuxWcMap.cpp -o LinuxWcMap
$ ./LinuxWcMap Linux*.cpp
346     LinuxCatMT.cpp
117     LinuxForkExec.cpp
594     LinuxGetSsl.cpp
397     LinuxGetUrl.cpp
75      LinuxHelloMT.cpp
460     LinuxListDirectory.cpp
305     LinuxPipe.cpp
201     LinuxSeekRead.cpp
162     LinuxSeekWrite.cpp
325     LinuxSignalDtor.cpp
298     LinuxSignalHandler.cpp
994     LinuxTinyServer.cpp
1087    LinuxTinySslServer.cpp
208     LinuxWcMap.cpp
173     LinuxWcStd.cpp
5742    Total
$
```

```
// Linux word-count using the mapped file support.
// Nicole Hamilton  nham@umich.edu

#include <sys/types.h>
#include <sys/stat.h>
#include <sys/mman.h>
#include <fcntl.h>
#include <unistd.h>
#include <stdlib.h>
#include <iostream>
using namespace std;


size_t FileSize( int f )
   {
   struct stat fileInfo;
   fstat( f, &fileInfo );
   return fileInfo.st_size;
   }


int main( int argc, char **argv )
   {
   int total = 0;

   for ( int i = 1;  i < argc;  i++ )
      {
      int words = 0;

      int f = open( argv[ i ], O_RDONLY );
```

```
if ( f != -1 )
   {
   size_t fileSize = FileSize( f );

   char *map = ( char * )mmap( nullptr,  fileSize,
      PROT_READ, MAP_PRIVATE, f, 0 );

   if ( map != MAP_FAILED )
      {
      bool midWord = false;
      char *end = map + fileSize;

      for ( char *c = map;  c < end;  c++ )
         switch ( *c )
            {
            case ' ':
            case '\t':
            case '\n':
            case '\r':
               if ( midWord )
                  {
                  midWord = false;
                  words++;
                  }
               break;
            default:
               midWord = true;
            }

      if ( midWord )
         words++;
      }
```

```
      close( f );
      cout << words << "\t" << argv[ i ]  << endl;
      total += words;
      }
   }

if ( argc > 2 )
   cout << total << "\t" << "Total" << endl;
}
```

# Agenda

1. Course Details.
2. Group photos.
3. Cartoon characters.
4. Handles and ids.
5. Reading stdin, writing stdout.
6. Opening files.
7. Seeking.
8. Memory-mapping files.
9. Walking the directory structure.

```
$ head -1 LinuxListDirectory.cpp
// Linux recursive list directory
$ g++ LinuxListDirectory.cpp -o LinuxListDirectory
$ ./LinuxListDirectory -1 x64
x64, type = directory
x64/., type = directory
x64/.., type = directory
x64/Debug, type = directory
$
```

```
// Linux recursive list directory

// Nicole Hamilton  nham@umich.edu

#include <sys/types.h>
#include <sys/stat.h>
#include <unistd.h>
#include <dirent.h>
#include <iostream>
using namespace std;


const char *Filetype( mode_t mode )
   {
   switch ( mode & S_IFMT )
      {
      case S_IFSOCK:
         return "socket";
      case S_IFLNK:
         return "symbolic link";
      case S_IFREG:
         return "regular file";
      case S_IFBLK:
         return "block device";
      case S_IFDIR:
         return "directory";
      case S_IFCHR:
         return "character device";
      case S_IFIFO:
         return "FIFO";
      default:
         return "unknown";
      }
   }
```

```
bool DotName( const char *name )
   {
   return name[ 0 ] == '.' &&
         ( name[ 1 ] == 0 || name[ 1 ] == '.' && name[ 2 ] == 0 );
   }
```

```
void ListDirectory( string filePath, const int maxDepth )
   {
   DIR *handle = opendir( filePath.c_str( ) );
   if ( handle )
      {
      struct dirent *entry;
      while ( entry = readdir( handle ) )
         {
         struct stat statbuf;
         string childPath = filePath + '/';
         childPath += entry->d_name;
         if ( stat( childPath.c_str( ), &statbuf ) )
            cerr << "stat of " << filePath << " failed, errno = " << errno << endl;
         else
            {
            cout << childPath << ", type = " << Filetype( statbuf.st_mode );
            if ( ( statbuf.st_mode & S_IFDIR ) == S_IFDIR )
               {
               cout << endl;
               if ( !DotName( entry->d_name ) && ( maxDepth > 0 || maxDepth == -1 ) )
                  ListDirectory( childPath, maxDepth < 0 ? -1 : maxDepth - 1 );
               }
            else
               cout << ", size = " << statbuf.st_size << endl;
            }
         }
      closedir( handle );
      }
   else
      cerr << "open handle failed, errno = " << errno;
   }
```

```
int main( int argc, char **argv )
   {
   if ( argc < 2 )
      {
      cerr << "Usage:  linuxlistdirectory [ -r | -- ] pathnames" << endl
           << "   -r   Recursive" << endl
           << "   -int maxDepth (default 0)" << endl;
      return 1;
      }

   argv++;
   argc--;
   int maxDepth = 0;

   if ( argv[ 0 ][ 0 ] == '-' )
      {
      if ( argv[ 0 ][ 1 ] == 'r' && argv[ 0 ][ 2 ] == 0 )
         maxDepth = -1;
      else
         maxDepth = atoi( argv[ 0 ] + 1 );
      argv++;
      argc--;
      }
```

```
for ( int i = 0;  i < argc;  i++ )
   {
   struct stat statbuf;
   if ( stat( argv[ i ], &statbuf ) )
      cerr << "stat of " << argv[ i ] << " failed, errno = " << errno << endl;
   else
      {
      cout << argv[ i ] << ", type = " << Filetype( statbuf.st_mode );
      if ( ( statbuf.st_mode & S_IFMT ) == S_IFDIR )
         {
         cout << endl;
         ListDirectory( argv[ i ], maxDepth < 0 ? -1 : maxDepth - 1 );
         }
      else
         cout << ", size = " << statbuf.st_size << endl;
      }
   }
}
```