

EECS 498 System Design of a Search Engine

Winter 2025

Lecture 6: Stating, mapped files and hashing

Nicole Hamilton

[https://web.eecs.umich.edu/~nham/
nham@umich.edu](https://web.eecs.umich.edu/~nham/nham@umich.edu)

agenda

1. Course details.
2. Group photos.
3. Review seeking.
4. stat'ing a file.
5. Memory-mapped files.
6. Hashing

agenda

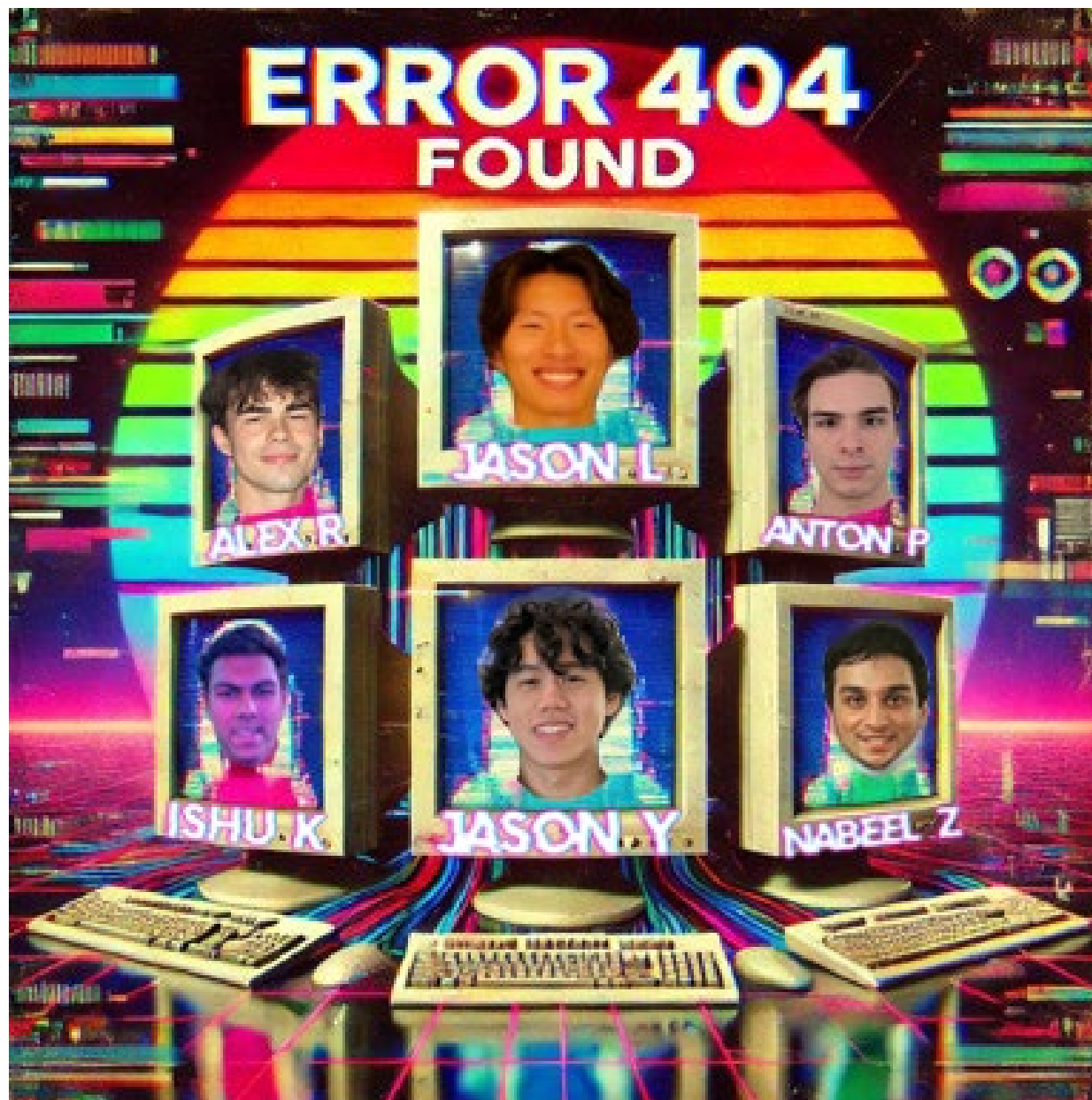
1. Course details.
2. Group photos.
3. Review seeking.
4. stat'ing a file.
5. Memory-mapped files.
6. Hashing

details

1. Finished reading your introductions.
2. Questions about homework, project, anything else?

agenda

1. Course details.
2. Group photos.
3. Review seeking.
4. stat'ing a file.
5. Memory-mapped files.
6. Hashing





Jimmy Dai

Rickey Dong

Kevin Li

Matthew Wang

Roe Tsimhoni

Danny Kim

Wonbin Jin

Menu

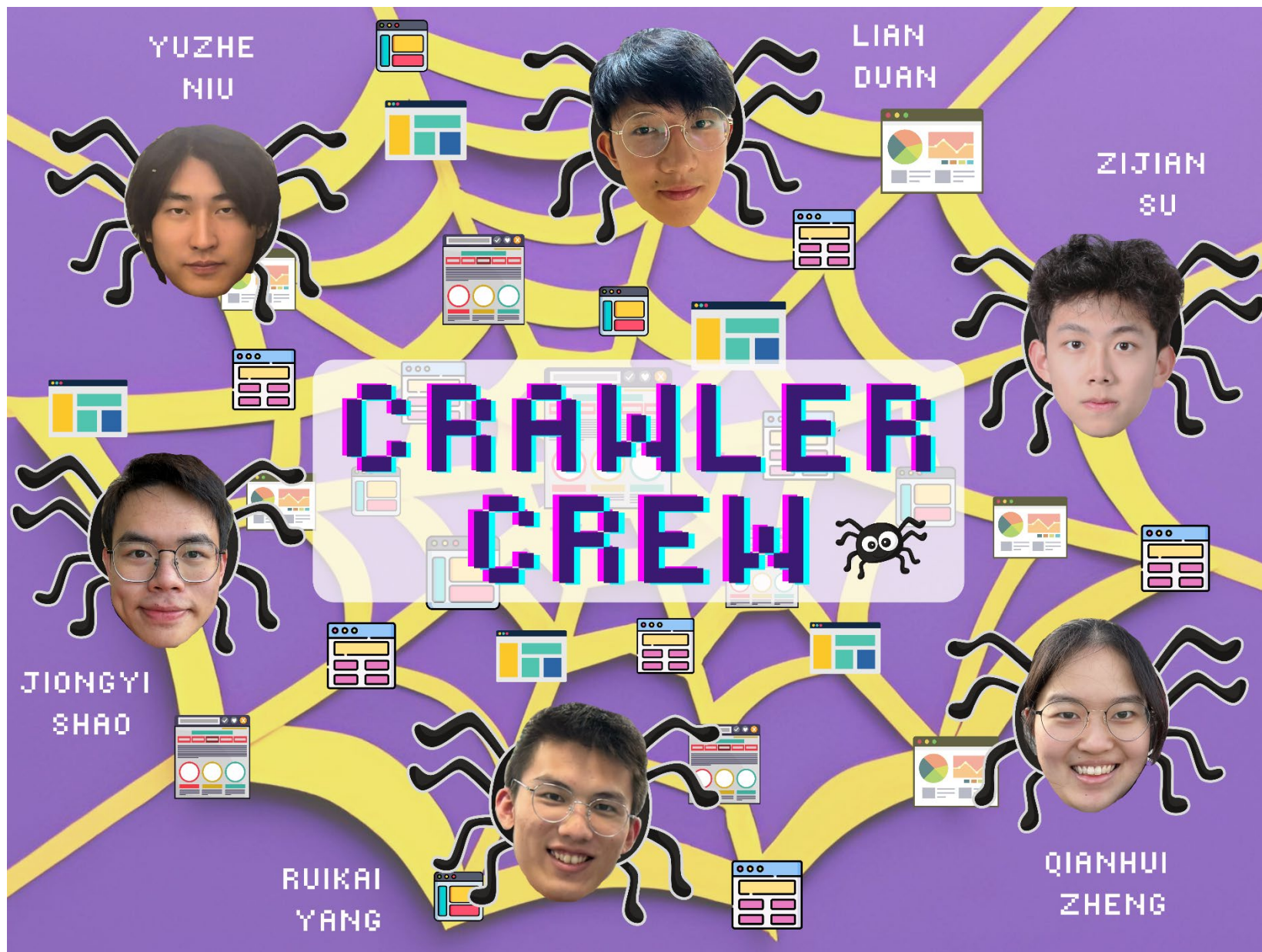
Drinks

- Bubble-Sorted Tea
- Merge-Sorted Mocha
- OOPlong Tea
- Green Tea MatChar
- Java Coffee

Copy and PASTERies

- Query Quiche
- Cookies
 - > Firefox-flavored
 - > Chrome-flavored
 - > Edge-flavored









THE FELLOWSHIP OF THE RING BUFFER





THE MAGNIFICENT SIX

THEY CODED LIKE SIX HUNDRED



VINCENT S.
"CRAWLER"

LUCAS B.
"PARSER"

KAES H.
"CONSTRAINT SOLVER"

XIULIN C.
"RANKER"

ROBERT M.
"INDEX"

NICHOLAS S.
"FRONT END"

agenda

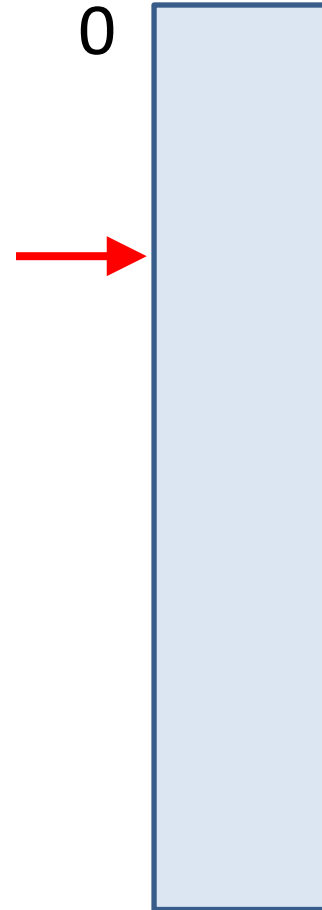
1. Course details.
2. Group photos.
3. Review seeking.
4. stat'ing a file.
5. Memory-mapped files.
6. Hashing

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.

lseek()

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

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

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 <code>offset</code> bytes.
SEEK_CUR	Current location plus <code>offset</code> .
SEEK_END	Size of the file plus <code>offset</code> .

`off_t` is a signed type. Return value `off_t = -1` indicates failure and `errno` gives the reason.

Linux

```
$ 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
$
```

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

Linux

```
$ 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
$
```

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?

Linux

```
$ 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
*
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. Review seeking.
4. Memory-mapped files.
5. stat'ing a file.
6. Hashing

Problem with read and write

You have to pick a buffer size and deal with the added complexity of scanning data that might cross a read/write buffer boundary.

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

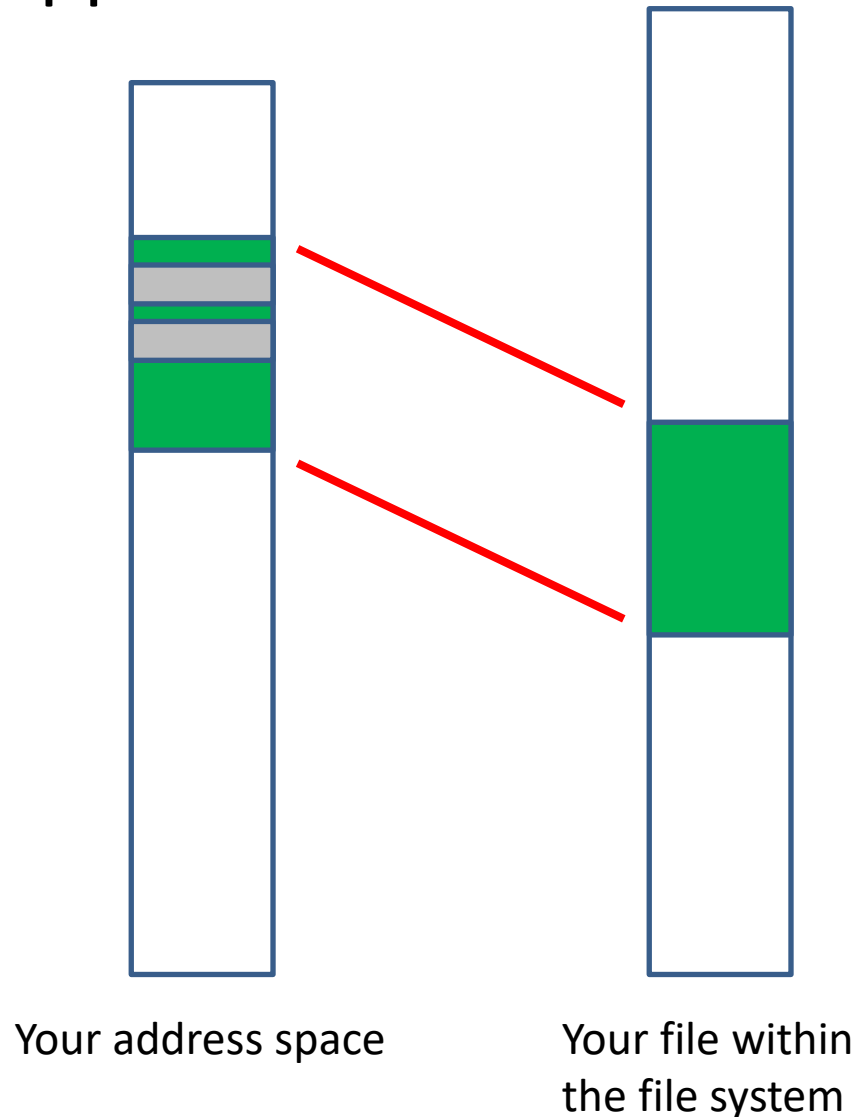
Mapped files

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.



```
#include <sys/mman.h>
```

```
void *mmap(void *addr, size_t length, int prot, int flags,  
           int fd, off_t offset);
```

mmap() creates a new mapping in the virtual address space of the calling process. The starting address for the new mapping can be specified in addr but is usually left as the nullptr. The length argument specifies the length of the mapping (which must be greater than 0).

If addr is NULL, then the kernel chooses the (page-aligned) address at which to create the mapping.

The prot argument describes the desired memory protection of the mapping, rwx.

flags indicate sharing options.

```
$ 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++;
    }
}

```

```

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++;
}

```

More overhead in the setup but a simpler inner loop.

*The void * returned by mmap() can be cast to anything you like.*

*Here, I've cast it to a simple char *.*

But I could cast it to pointer to a very complex struct, e.g.,

*mystruct *s = (mystruct *)mmap(...);*

```
        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. Review seeking.
4. Memory-mapped files.
5. stat'ing a file.
6. Hashing

```
#include <sys/stat.h>
```

```
int stat(const char *pathname, struct stat *statbuf);  
int fstat(int fildes, struct stat *buf);
```

fstat() returns information about a file associated with an open file descriptor in a stat structure defined in <sys/stat.h>.

```

struct stat {
    dev_t      st_dev;          /* ID of device containing file */
    ino_t      st_ino;          /* Inode number */
    mode_t     st_mode;         /* File type and mode */
    nlink_t    st_nlink;        /* Number of hard links */
    uid_t      st_uid;          /* User ID of owner */
    gid_t      st_gid;          /* Group ID of owner */
    dev_t      st_rdev;         /* Device ID (if special file) */
    off_t      st_size;         /* Total size, in bytes */
    blksize_t  st_blksize;      /* Block size for filesystem I/O */
    blkcnt_t   st_blocks;       /* Number of 512B blocks allocated */

    /* Since Linux 2.6, the kernel supports nanosecond
       precision for the following timestamp fields.
       For the details before Linux 2.6, see NOTES. */

    struct timespec st_atim;     /* Time of last access */
    struct timespec st_mtim;     /* Time of last modification */
    struct timespec st_ctim;     /* Time of last status change */

#define st_atime st_atim.tv_sec /* Backward compatibility */
#define st_mtime st_mtim.tv_sec
#define st_ctime st_ctim.tv_sec
};

```

The st_mode field tells what this thing is.

```
struct stat {
    dev_t      st_dev;          /* ID of device containing file */
    ino_t      st_ino;          /* Inode number */
    mode_t     st_mode;         /* File type and mode */
    nlink_t    st_nlink;       /* Number of hard links */
    uid_t      st_uid;         /* User ID of owner */
    gid_t      st_gid;         /* Group ID of owner */
    dev_t      st_rdev;        /* Device ID (if special file) */
    off_t      st_size;         /* Total size, in bytes */
    blksize_t  st_blksize;     /* Block size for filesystem I/O */
    blkcnt_t   st_blocks;      /* Number of 512B blocks allocated */

    /* Since Linux 2.6, the kernel supports nanosecond
       precision for the following timestamp fields.
       For the details before Linux 2.6, see NOTES. */

    struct timespec st_atim;    /* Time of last access */
    struct timespec st_mtim;    /* Time of last modification */
    struct timespec st_ctim;    /* Time of last status change */

#define st_atime st_atim.tv_sec      /* Backward compatibility */
#define st_mtime st_mtim.tv_sec
#define st_ctime st_ctim.tv_sec
};
```

```
// Linux stat command to retrieve pathname type
```

```
#include <sys/types.h>
#include <sys/stat.h>
#include <unistd.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";
    }
}
```

```
$ ./stat c* debug
catmt.cpp type = regular file, size = 2798
catmt.sln type = regular file, size = 1059
catmt.vcxproj type = regular file, size = 5446
catmt.vcxproj.filters type = regular file, size = 944
CreateProcess.cpp type = regular file, size = 822
CreateProcess.sln type = regular file, size = 1437
CreateProcess.vcxproj type = regular file, size = 5461
CreateProcess.vcxproj.filters type = regular file, size = 959
debug type = directory, size = 0
$
```

```
int main( int argc, char **argv )
{
    if ( argc < 2 )
    {
        cerr << "Usage:  stat pathnames" << endl;
        return 1;
    }

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

agenda

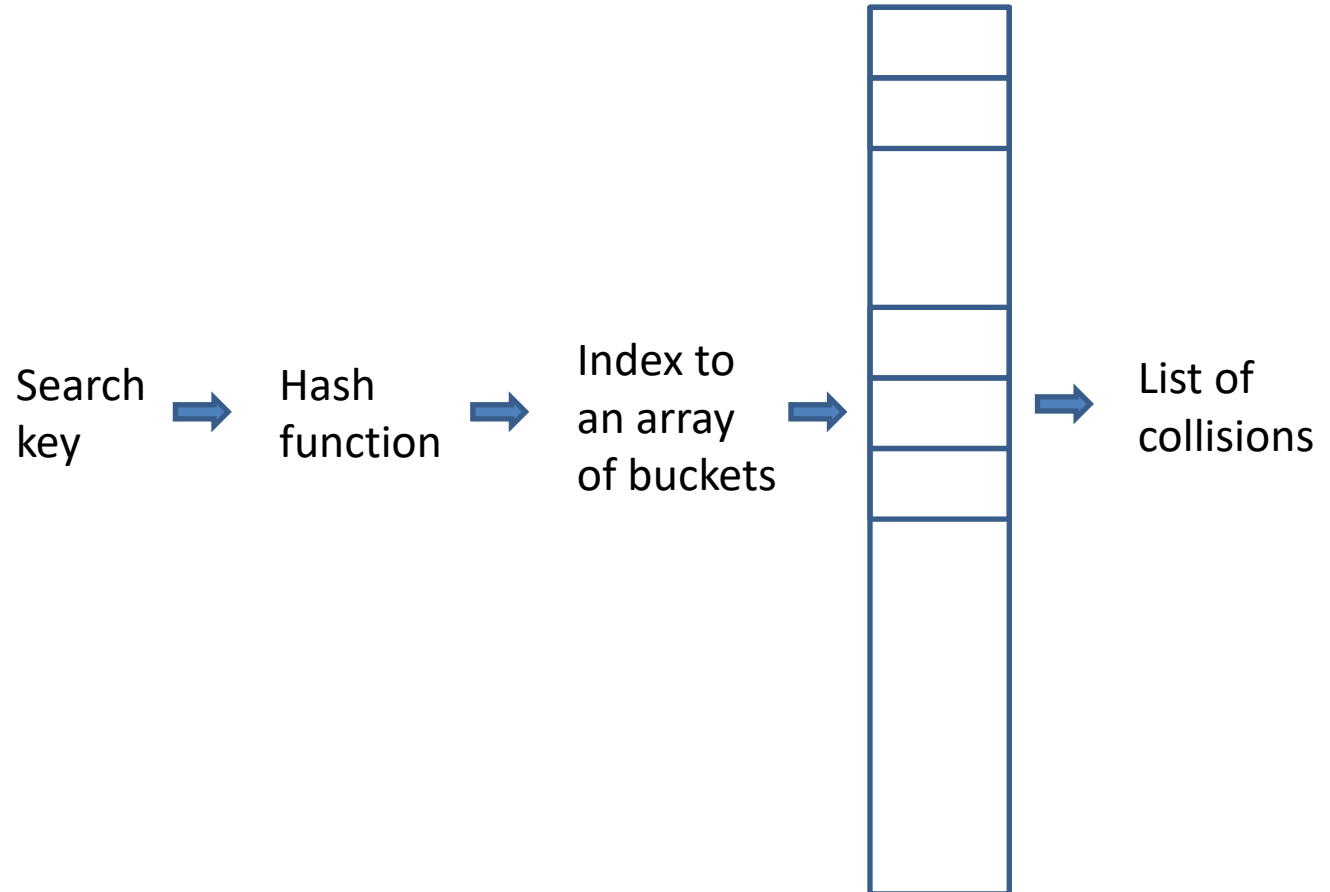
1. Course details.
2. Group photos.
3. Review seeking.
4. Memory-mapped files.
5. stat'ing a file.
6. Hashing

Hashing

Create an array of buckets to hold the items you wish to index based on a key field.

Create a simple function that can map the key to a bucket index.

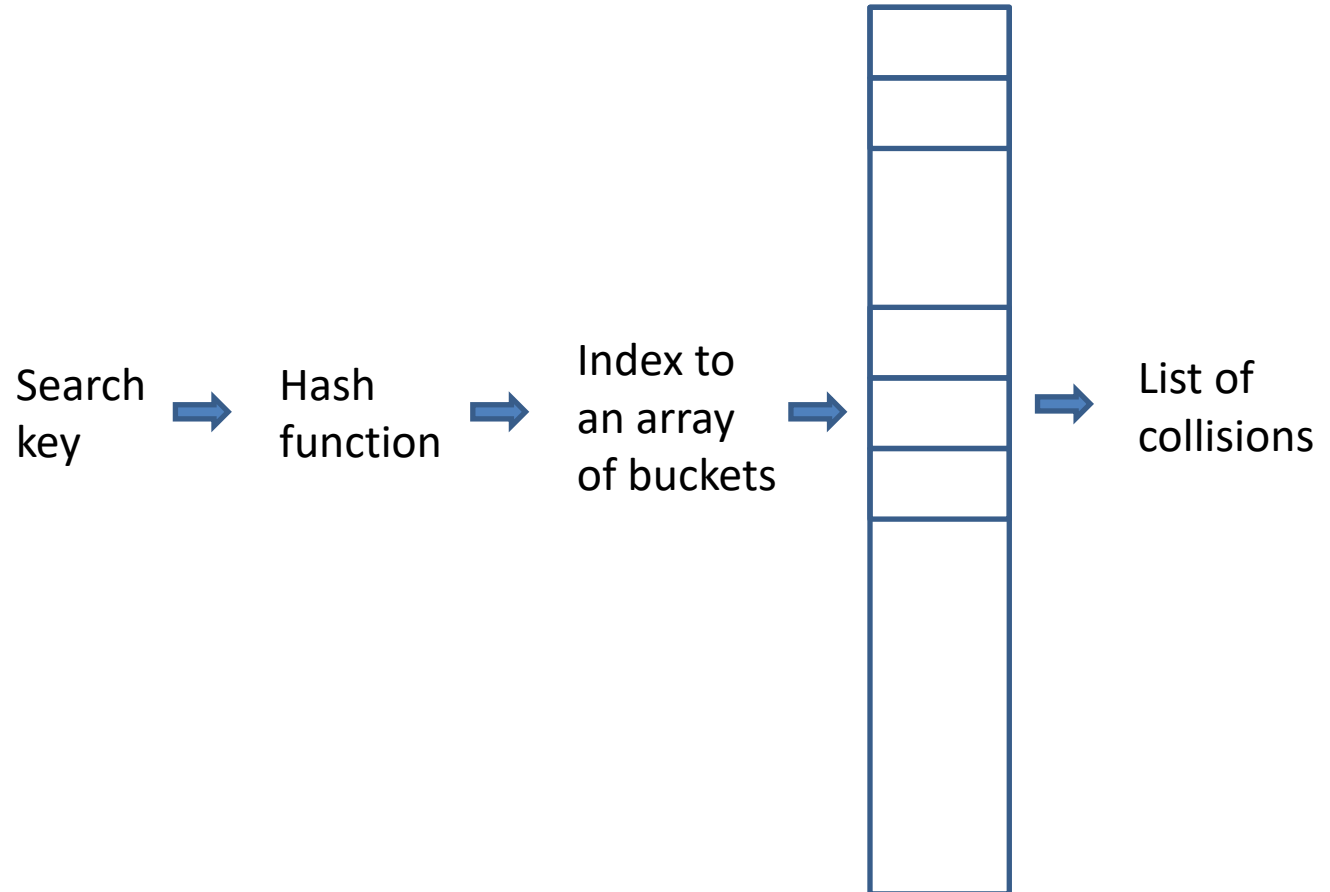
If multiple different keys map to the same bucket, that's called a collision.



Hashing

Expensive to grow the table or deal with collisions, so we make the table pretty big.

No serious developer would use hashing except when they expect *lots* of entries that don't have to be kept in sort order, so there's no point in starting with only two buckets.

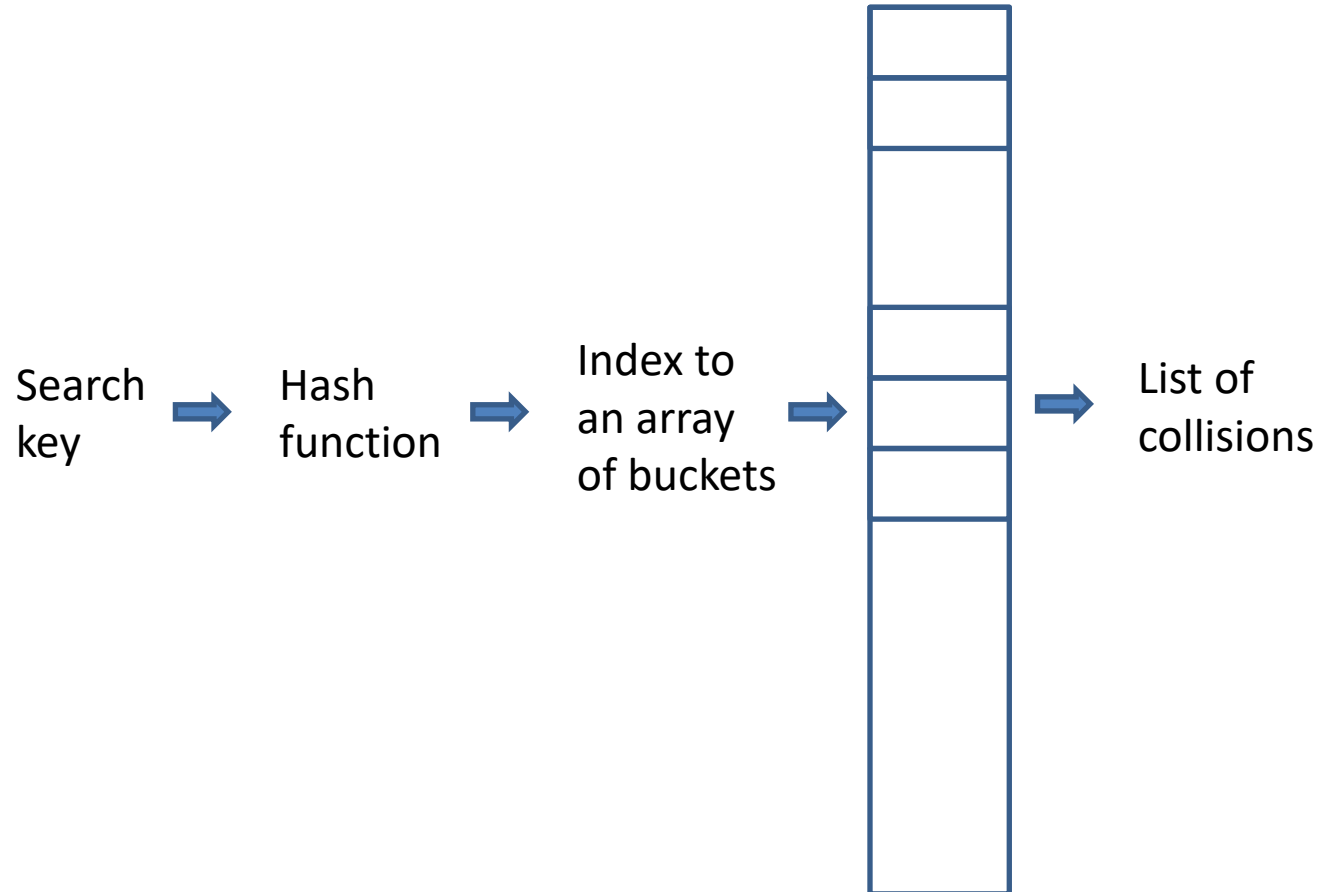


Hashing

Start with 1024 or 4096 or some other big power-of-two number of buckets that you expect matches the problem.

Use the low bits of the hash function as an index.

You may/may not support growing the hash table.



Hash function

Even a simple hash can work pretty well.

But what's the problem with this one?

```
unsigned Hash( const char *p )  
{  
    unsigned h = 0;  
    for ( ; *p; p++ )  
        h = ( h << 1 ) ^ *p;  
    return h;  
}
```

Fowler-Noll-Vo hash function

A more
complex
hash
using big
prime
numbers.

```
size_t fnvHash ( const char *data, size_t length )
{
    static const size_t FnvOffsetBasis =
        146959810393466560
    static const size_t FnvPrime =
        1099511628211ul;

    size_t hash = FnvOffsetBasis;
    for ( size_t i = 0; i < length; ++i )
    {
        hash *= FnvPrime;
        hash ^= data[ i ];
    }
    return hash;
}
```

Polynomial function

CRCs (cyclic redundancy codes) use shift registers.

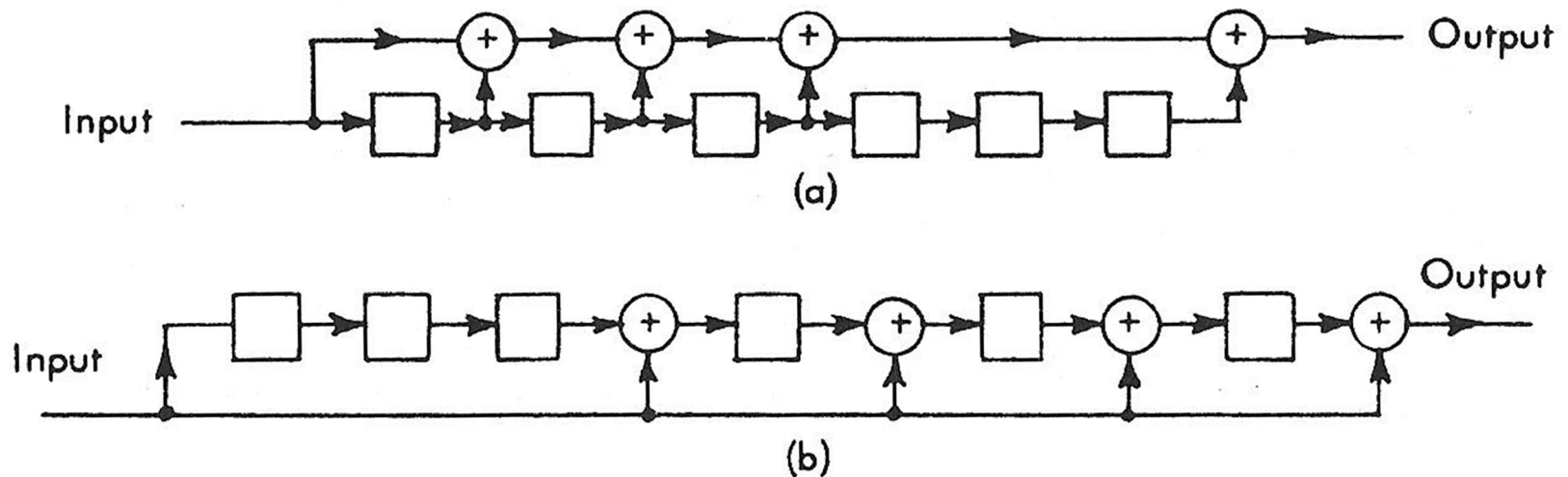


Figure 7.5. Circuits for multiplying by $X^6 + X^5 + X^4 + X^3 + 1$.

Source: W. Wesley Peterson and E.J. Weldon, Jr, *Error-Correcting Codes*, Second Edition, MIT Press, 1972, pp 178-179.

Polynomial function

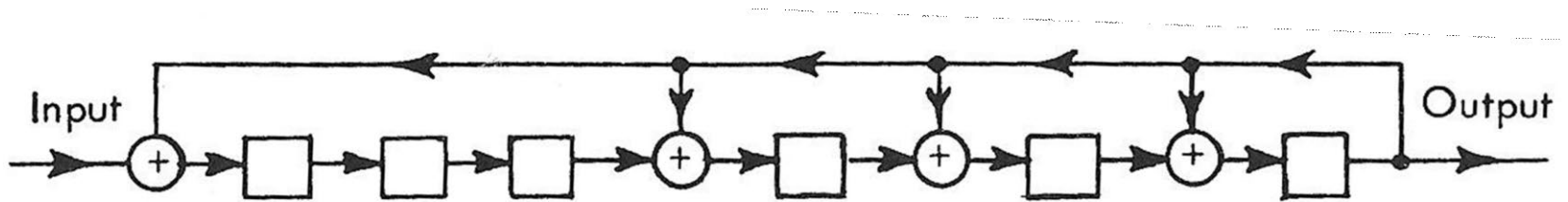


Figure 7.7. A circuit for dividing by $X^6 + X^5 + X^4 + X^3 + 1$.

```

/*
Ethernet CRC
Polynomial    x^32 + x^26 + x^23 + x^22 + x^16 + x^12 + x^11 + x^10 +
               x^8 + x^7 + x^5 + x^4 + x^2 + x^1 + 1

```

Represent the polynomial as an unsigned number with each bit corresponding to a term of the polynomial set to one. E.g., the x^5 term means bit 5 (counting from the LSB as bit 0) is on. The x^{32} term is actually the feedback term and is "off the end" of the polynomial.

```

*/

#define Ethernet_Polynomial    0x04c11db7
#define Poly                    Ethernet_Polynomial

```

```
/* Calculate the CRC table entries.  crc(x) returns table entry x.  Use this
   routine to fill a table crctab[256] with the appropriate values.
*/
```

```
ulong crc( register ulong i )
{
    i <<= 24;

    i = (i << 1) ^ ((i & 0x80000000) ? Poly : 0);
    i = (i << 1) ^ ((i & 0x80000000) ? Poly : 0);
    i = (i << 1) ^ ((i & 0x80000000) ? Poly : 0);
    i = (i << 1) ^ ((i & 0x80000000) ? Poly : 0);
    i = (i << 1) ^ ((i & 0x80000000) ? Poly : 0);
    i = (i << 1) ^ ((i & 0x80000000) ? Poly : 0);
    i = (i << 1) ^ ((i & 0x80000000) ? Poly : 0);
    i = (i << 1) ^ ((i & 0x80000000) ? Poly : 0);

    return i;
}
```

```
static const unsigned long CRCTable[ ] =  
    { 0x00000000, 0x04c11db7, 0x09823b6e, 0x0d4326d9,  
      0x130476dc, 0x17c56b6b, 0x1a864db2, 0x1e475005,  
      0x2608edb8, 0x22c9f00f, 0x2f8ad6d6, 0x2b4bcb61,  
      0x350c9b64, 0x31cd86d3, 0x3c8ea00a, 0x384fbdbd,  
      0x4c11db70, 0x48d0c6c7, 0x4593e01e, 0x4152fda9,  
      0x5f15adac, 0x5bd4b01b, 0x569796c2, 0x52568b75,  
      0x6a1936c8, 0x6ed82b7f, 0x639b0da6, 0x675a1011,  
      . . .  
      0xbcb4666d, 0xb8757bda, 0xb5365d03, 0xb1f740b4    };
```

```
// To calculate a CRC, use the UpdateCRC( crc, c ) procedure to  
// accumulate the data into the CRC. The crc argument is the  
// accumulated CRC, c is the next character of data to be added.  
// Initially, crc = 0.
```

```
unsigned long UpdateCRC( unsigned long crc, unsigned char c )  
{  
    return ( ( ( crc ) << 8 ) ^ CRCTable[ ( ( crc ) >> 24 ) ^ ( c ) ] );  
}
```

Buckets

Can either contain the object being indexed or a pointer to the object.

If it's the object itself, you need at least as many buckets as objects and each bucket will be larger.

If it's a pointer, it can be to a list.

Collisions

You have to assume that you will have collisions.

When searching a hash bucket, you must compare the search keys to make sure that what you found is what you wanted.

If the key fields are long, you might decide to add a field to a bucket entry with the complete hash of the key and compare that before comparing the entire field.

Collisions

Several ways to deal with collisions.

1. Increment the bucket number modulo the number of buckets and try there.
2. Use a second hash (or just different bits of the same hash) and try there.
3. Create a list off each bucket.
4. Try (1) or (2) and if not successful, create list.

Collisions

When using a list for collisions, the insertion order matters.

The STL map template pushes collisions onto the front of the list.

Why did they do that and is that a good choice?

Collisions

If query performance is more important than build time, collisions should be placed onto the end.

Why?

Collisions

If query performance is more important than build time, collisions should be placed onto the end.

Why?

Because you expect to encounter the most frequent terms earlier in the build and you want those at the front.

Collisions

If you use a list for collisions, most people normally think of something like a linked list.

Since you don't know what will have to go into the list, one way to deal with collisions by pushing new node onto the list (or onto the end).

Collisions

But suppose you already know what the data looks like.

Could you use that information to build a better performant, smaller memory footprint, more efficient hash table to hold exactly that known data?

If you know the data

You might try several hash algorithms to see which gives you the lowest rate of collisions.

You could tune the number of buckets.

You could compress the collision lists. Instead of a linked list, it could be an array or a serialized stream.

Compressing the list

Buckets contain pointers or, even better, offsets into one large contiguous buffer containing the items stored in the hash table.

Each list is a contiguous series of bytes representing a concatenation of the entries on a collision list.

If you only need to read them serially, no problem if the entries vary in size, as long as you can tell them apart.

You do need to know when you've reached the end, typically with a sentinel.