EECS 440 System Design of a Search Engine

Winter 2021

Lecture 8: Mapped files, processes and threads

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

Agenda

1. Course details.
2. stat’ing a file.
3. Seeking.
4. Memory-mapping files.
5. Processes.
6. Threads.

Agenda

1. Course details.
2. stat’ing a file.
3. Seeking.
4. Memory-mapping files.
5. Processes.
6. Threads.

details

1. Still struggling to catch up. I don’t yet have my usual energy.

Reading list

Please read the first 3 main articles by Dennis Ritchie and Ken Thompson.

THE BELL SYSTEM TECHNICAL JOURNAL

JULY/AUGUST 1978
VOL. 57, NO. 6, PART 2

UNIX TIME-SHARING SYSTEM

Preface — 1897
T. H. Crowley

Foreword — 1899
M. D. McIlroy, E. N. Pinson, and B. A. Tague

The UNIX Time-Sharing System — 1905
D. M. Ritchie and K. Thompson

UNIX Implementation — 1931
K. Thompson

A Retrospective — 1947
D. M. Ritchie

The UNIX Shell — 1971
S. R. Bourne

The C Programming Language — 1991
D. M. Ritchie, S. C. Johnson, M. E. Lesk, and B. W. Kernighan

Portability of C Programs and the UNIX System — 2021
S. C. Johnson and D. M. Ritchie

The MERT Operating System — 2049
H. Lycklama and D. L. Bayer

UNIX on a Microprocessor — 2087
H. Lycklama

A Minicomputer Satellite Processor System — 2103
H. Lycklama and C. Christensen

Document Preparation — 2115
B. W. Kernighan, M. E. Lesk, and J. F. Ossanna, Jr.

Statistical Text Processing — 2137
L. E. McMahon, L. L. Cherry, and R. Morris

Language Development Tools — 2155
S. C. Johnson and M. E. Lesk

(Contents continued on outside back cover)

http://emulator.pdp-11.org.ru/misc/1978.07_-_Bell_System_Technical_Journal.pdf

Image source: https://en.wikipedia.org/wiki/Dennis_Ritchie#/media/File:Ken_Thompson_and_Dennis_Ritchie.jpg

The first one is especially helpful.

Here's a better PDF.

I may test you on it.

The UNIX Time-Sharing System

Dennis M. Ritchie and Ken Thompson
Bell Laboratories

UNIX is a general-purpose, multi-user, interactive operating system for the Digital Equipment Corporation PDP-11/40 and 11/45 computers. It offers a number of features seldom found even in larger operating systems, including: (1) a hierarchical file system incorporating demountable volumes; (2) compatible file, device, and inter-process I/O; (3) the ability to initiate asynchronous processes; (4) system command language selectable on a per-user basis; and (5) over 100 subsystems including a dozen languages. This paper discusses the nature and implementation of the file system and of the user command interface.

Key Words and Phrases: time-sharing, operating system, file system, command language, PDP-11

CR Categories: 4.30, 4.32

Copyright © 1974, Association for Computing Machinery, Inc. General permission to republish, but not for profit, all or part of this material is granted provided that ACM's copyright notice is given and that reference is made to the publication, to its date of issue, and to the fact that reprinting privileges were granted by permission of the Association for Computing Machinery.

This is a revised version of a paper presented at the Fourth ACM Symposium on Operating Systems Principles, IBM Thomas J. Watson Research Center, Yorktown Heights. New York, October 15–17, 1973. Authors' address: Bell Laboratories, Murray Hill, NJ 07974.

The electronic version was recreated by Eric A. Brewer, University of California at Berkeley, brewer@cs.berkeley.edu. Please notify me of any deviations from the original; I have left errors in the original unchanged.

1. Introduction

There have been three versions of UNIX. The earliest version (circa 1969–70) ran on the Digital Equipment Corporation PDP-7 and -9 computers. The second version ran on the unprotected PDP-11/20 computer. This paper describes only the PDP-11/40 and /45 [1] system since it is more modern and many of the differences between it and older UNIX systems result from redesign of features found to be deficient or lacking.

Since PDP-11 UNIX became operational in February 1971, about 40 installations have been put into service; they are generally smaller than the system described here. Most of them are engaged in applications such as the preparation and formatting of patent applications and other textual material, the collection and processing of trouble data from various switching machines within the Bell System, and recording and checking telephone service orders. Our own installation is used mainly for research in operating systems, languages, computer networks, and other topics in computer science, and also for document preparation.

Perhaps the most important achievement of UNIX is to demonstrate that a powerful operating system for interactive use need not be expensive either in equipment or in human effort: UNIX can run on hardware costing as little as $40,000, and less than two man years were spent on the main system software. Yet UNIX contains a number of features seldom offered even in much larger systems. It is hoped, however, the users of UNIX will find that the most important characteristics of the system are its simplicity, elegance, and ease of use.

Besides the system proper, the major programs available under UNIX are: assembler, text editor based on QED [2], linking loader, symbolic debugger, compiler for a language resembling BCPL [3] with types and structures (C), interpreter for a dialect of BASIC, text formatting program, Fortran compiler, Snobol interpreter, top-down compiler-compiler (TMG) [4], bottom-up compiler-compiler (YACC), form letter generator, macro processor (M6) [5], and permuted index program.

There is also a host of maintenance, utility, recreation, and novelty programs. All of these programs were written locally. It is worth noting that the system is totally self-supporting. All UNIX software is maintained under UNIX; likewise, UNIX documents are generated and formatted by the UNIX editor and text formatting program.

2. Hardware and Software Environment

The PDP-11/45 on which our UNIX installation is implemented is a 16-bit word (8-bit byte) computer with 144K bytes of core memory; UNIX occupies 42K bytes. This system, however, includes a very large number of device drivers and enjoys a generous allotment of space for I/O buffers and system tables; a minimal system capable of running the

https://people.eecs.berkeley.edu/~brewer/cs262/unix.pdf

An internal paper I wrote at Microsoft as we prepared to go live in January 2005.

Slightly redacted to gain approval from Microsoft for use in this class.

Posted to Canvas.

Dynamic Ranking

by Nicole Hamilton

Ranking is the process by which a search engine decides the order of results. This paper will outline how that's done in a modern search engine and describe one particular algorithm and heuristic scoring method investigated by the author.

Search engine ranking is typically considered as two parts, static ranking and dynamic ranking. Static ranking is an estimate of the importance or quality of the page *without regard to the query*. For example, the New York Times home page is certain to be a high-quality result if it matches at all and we can recognize that with hand-curated list of important sites on the web, as Yahoo! did in its early days or by using algorithmic means to identify important sites, e.g., by noticing that many other sites link to it. Other static attributes include the length of the URL, the title or the page, the domain, e.g., .gov or .edu versus .biz, whether it contains images, lots of links, invisible text, pornographic content, the vocabulary level, spelling errors, and so on. Any data that would inform on the quality of a page independent of a query might be a candidate to be collected for static ranking.

Dynamic ranking, by contrast, is all about making an estimate of the quality of the page as a result *for a particular query* through some means that considers both the static information about the page and the quality of the match between the query and the page content. Though done by an algorithm running in a computer, the objective is to order the results the way a human would.

Basic Dynamic Ranking

The dynamic ranker in a typical search engine is given a query string and a handle to an inverted word index of some number of pages on the web. Typically, an entire engine is composed of thousands of machines, e.g., m rows of n machines, each with an index covering $1/n$-th of the pages known to the engine. As queries arrive from the web service front end, a row is selected, perhaps randomly or by queue length, and the query is broadcast to all the machines in that row. The results from the entire row are collected and top results merged together into a final list

The dynamic ranker compiles the query, searches the index for matching pages and returns a list of the n best pages, each with a score representing its estimated quality as a result. It may also be instrumented to return various levels of debug information, e.g., to allow the scoring calculation be re-run off-line, including redoing it with different scoring parameters.

The query string is a canonicalized form of whatever the user has typed. In Japanese and other Asian languages, the queries are commonly word-broken, e.g., 知多半島道路 becomes 知多+ 半島+ 道路, literally, Chita peninsula road. In Western languages, a naïve word-breaker simply breaks on and discards most punctuation and folds the character set by discarding accents and lower-casing the characters to match the way words are typically indexed.

Typically, the front end will also have prepended some market-specific augmentation string onto the canonicalized query, e.g., to specify that Japanese language pages are preferred in the Japanese market.

Agenda

1. Course details.
2. Homework 1.
3. stat'ing a file.
4. Seeking.
5. Memory-mapping files.
6. Processes.
7. Threads.

```c
#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>.

```c
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
};
```

```cpp
// 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

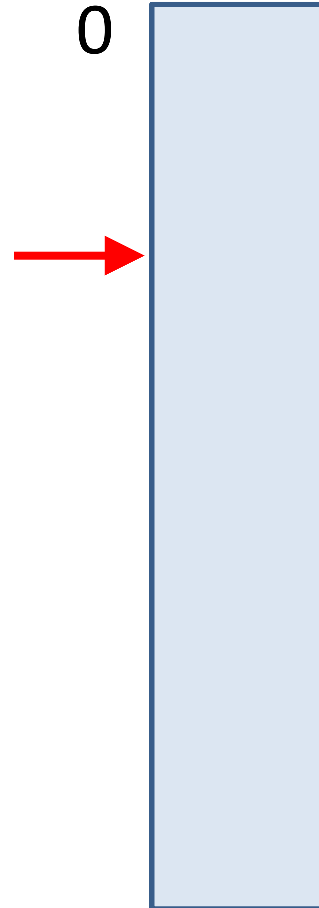
1. Course details.
2. Homework 1.
3. stat'ing a file.
4. Seeking.
5. Memory-mapping files.
6. Processes.
7. Threads.

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 offset bytes.
SEEK_CUR	Current location plus offset.
SEEK_END	Size of the file plus offset.

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

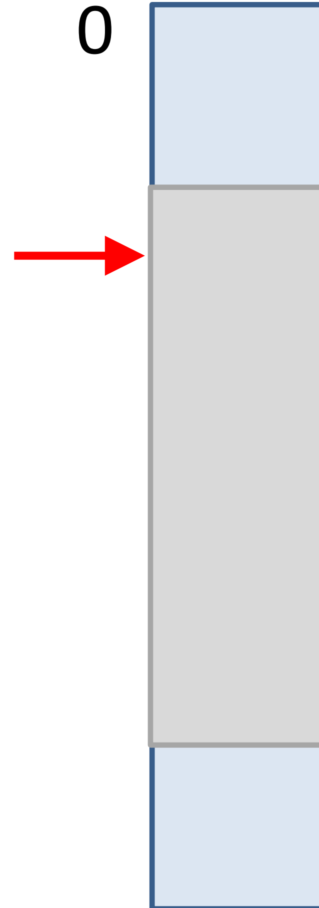
```
$ echo hello world > foobar
$ ls -l foobar
-rwxrwxrwx 3 nicole nicole 12 Jan 24 12:50 foobar
$ ./LinuxSeekWrite foobar 1000000
This is way past the end.
$ ls -l foobar
-rwxrwxrwx 3 nicole nicole 1000026 Jan 24 12:50 foobar
$ od -c foobar
0000000   h   e   l   l   o       w   o   r   l   d  \n  \0  \0  \0  \0
0000020  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \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. Homework 1.
3. stat'ing a file.
4. Seeking.
5. Memory-mapping files.
6. Processes.
7. Threads.

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.

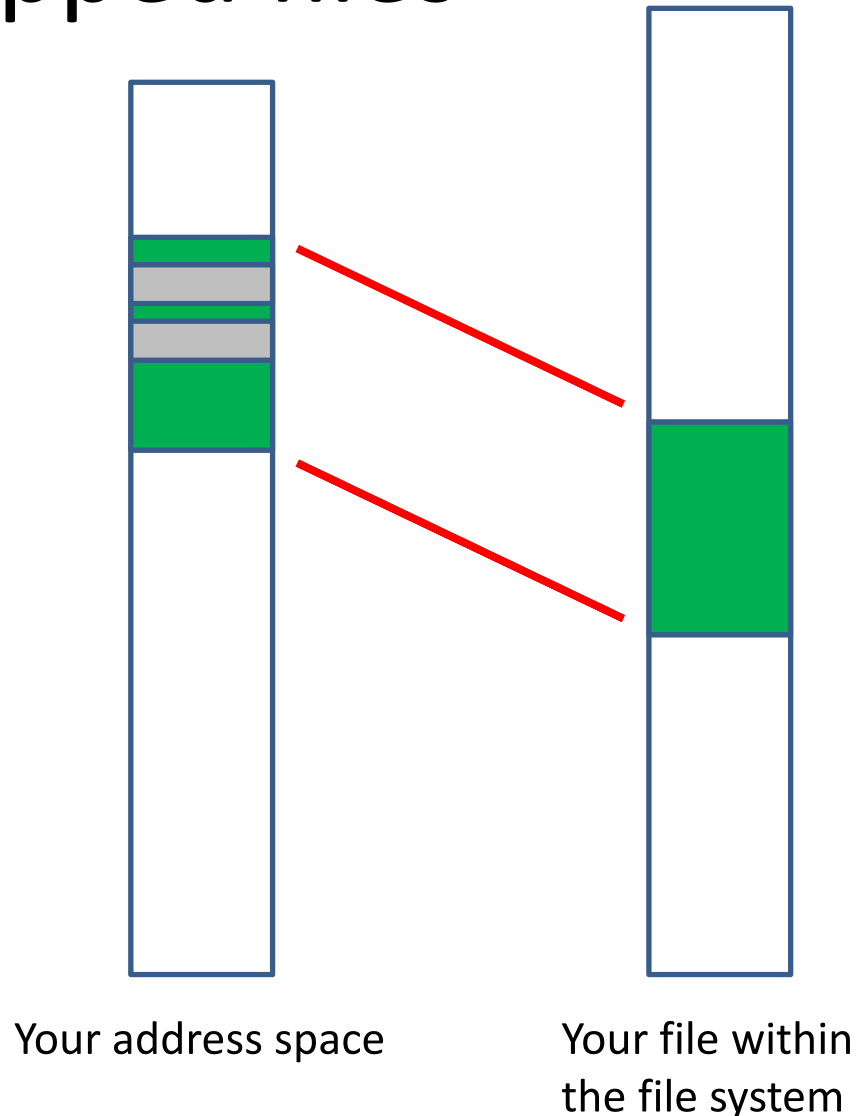
Mapped files

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.



```c
#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. Homework 1.
3. stat'ing a file.
4. Seeking.
5. Memory-mapping files.
6. Processes.
7. Threads.

The Process Model

1. Each process is protected from other processes.
2. Owns resources:
   a. Memory (instructions, stack, data)
   b. Open handles to files, pipes, semaphores, etc.
3. Can also share resources, e.g., blocks of memory.
4. Has "state" information:
   a. Current directory
   b. Environment variables
   c. One or more threads of execution
5. One-way inheritance to children.

Process creation

1. When you type a command into a Unix shell, it creates a child process to run that command.
2. The child process is traditionally created by a fork( ) + exec( ).
3. fork( ) creates an exact duplicate of the calling process and returns 0 to the child and the process id of the child to the parent.
4. exec( ) overlays the current process with a new executable image, but retaining any open handles.

Linux fork( )

1. Creates a child copy of the current process.
2. Returns 0 to the child, process ID to the parent.
3. Typically followed by an exec( ) in the child to overwrite the child process with a new executable file.
4. Multiple versions of exec( ) give various options, e.g., search path, etc.
5. Wait on the process ID for the child to complete.
6. Accomplished by sharing VM page table entries, marking them all read-only, then using copy-on-write when either process makes a change.

```
$ g++ LinuxForkExec.cpp -o LinuxForkExec
$ ./LinuxForkExec wc LinuxForkExec.cpp
parent waiting for child
child starting wc
 38 117 861 LinuxForkExec.cpp
child has exited with status = 0
$
```

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

pid_t fork(void);
```

fork() creates a new process by duplicating the calling process. The new process is referred to as the child process. The calling process is referred to as the parent process.

The child process and the parent process run in separate memory spaces. At the time of fork() both memory spaces have the same content.

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

pid_t fork(void);
```

The child process is an exact duplicate of the parent process except for the following points:

1. The child has its own unique process ID.

2. The child's parent process ID is the same as the parent's process ID.

3. The child does not inherit its parent's memory locks, timers, pending signals and outstanding asynchronous I/O.

```
#include <unistd.h>

extern char **environ;

int execl(const char *path, const char *arg, ..., NULL */);
int execlp(const char *file, const char *arg, ...
               /* (char  *) NULL */);
int execle(const char *path, const char *arg, ...
               /*, (char *) NULL, char * const envp[] */);
int execv(const char *path, char *const argv[]);
int execvp(const char *file, char *const argv[]);
int execvpe(const char *file, char *const argv[],
               char *const envp[]);
```

The exec() family of functions replaces the current process image with a new process image.

The initial argument for these functions is the name of a file that is to be executed.

```c
#include <sys/wait.h>

pid_t waitpid(pid_t pid, int *stat_loc, int options);
```

The waitpid() function suspends execution of the calling thread until child process terminates then returns information about its exit status.

```
$ g++ LinuxForkExec.cpp -o LinuxForkExec
$ ./LinuxForkExec wc LinuxForkExec.cpp
parent waiting for child
child starting wc
 38 117 861 LinuxForkExec.cpp
child has exited with status = 0
$
```

```
#include <sys/types.h>
#include <unistd.h>
#include <sys/wait.h>
#include <iostream>
using namespace std;

int main( int argc, char **argv )
   {
   if ( --argc == 0 )
      {
      cerr << "Usage:  CreateProcess command arguments" << endl;
      return 1;
      }

   pid_t processId = fork( );
   if ( processId )
      {
      // parent process
      cout << "parent waiting for child" << endl;
      int waitStatus;
      waitpid( processId, &waitStatus, 0 );
      cout << "child has exited with status = " << WEXITSTATUS( waitStatus )
            << endl;
      }
   else
      {
      // child process
      argv++;
      cout << "child starting " << *argv << endl;
      execvp( *argv, argv );
      cout << "this never prints" << endl;
      }
   }
```

Agenda

1. Course details.
2. Homework 1.
3. stat’ing a file.
4. Seeking.
5. Memory-mapping files.
6. Processes.
7. Threads.

Threads vs. Processes

Processes provide concurrency *between* applications:

1. High startup costs.
2. One-way inheritance.
3. Lots of “firewalling.”
4. Errant apps can’t scribble on others.

Threads provide concurrency *within* an application:

1. Very low cost to spawn.
2. Only a scheduler entry is created.
3. Everything else is shared.
4. No protection between threads.

What is a thread?

A simple flow of control that can be separately scheduled.

Its “state” consists of:

1. An instruction pointer,
2. A stack,
3. A register set,
4. Its scheduling priority,
5. Any semaphores it owns.

The operating system
Virtual memory, scheduling, file system,
i/o devices

Process 1

Memory image, open files, current directory, a running program.

Process 2

Memory image, open files, current directory, a running program.

...

Process n

Memory image, open files, current directory, a running program.

Shared process
Memory image, open files, current directory,
a running program, argc, argv, envp

Thread 1

Instruction pointer, register set, stack pointer, Scheduling priority, locks held

Thread 2

Instruction pointer, register set, stack pointer, Scheduling priority, locks held

…

Thread n

Instruction pointer, register set, stack pointer, Scheduling priority, locks held

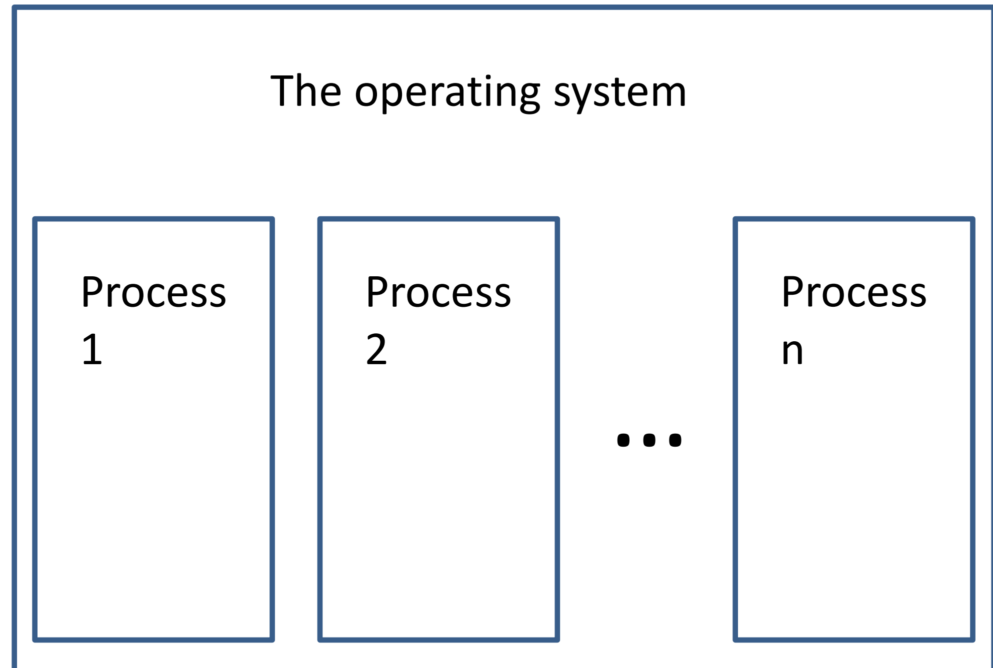
Processes provide concurrency between applications.

High startup costs.

One-way inheritance.

Lots of “firewalling.”

Errant apps can’t scribble on others.



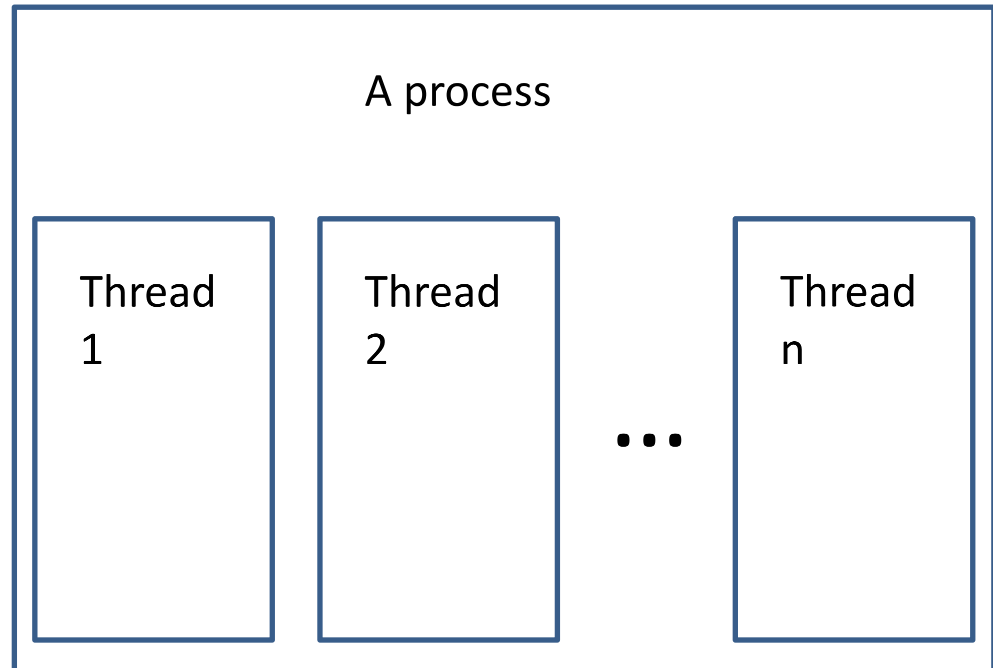
Threads provide concurrency *within* an application:

Very low cost to spawn.

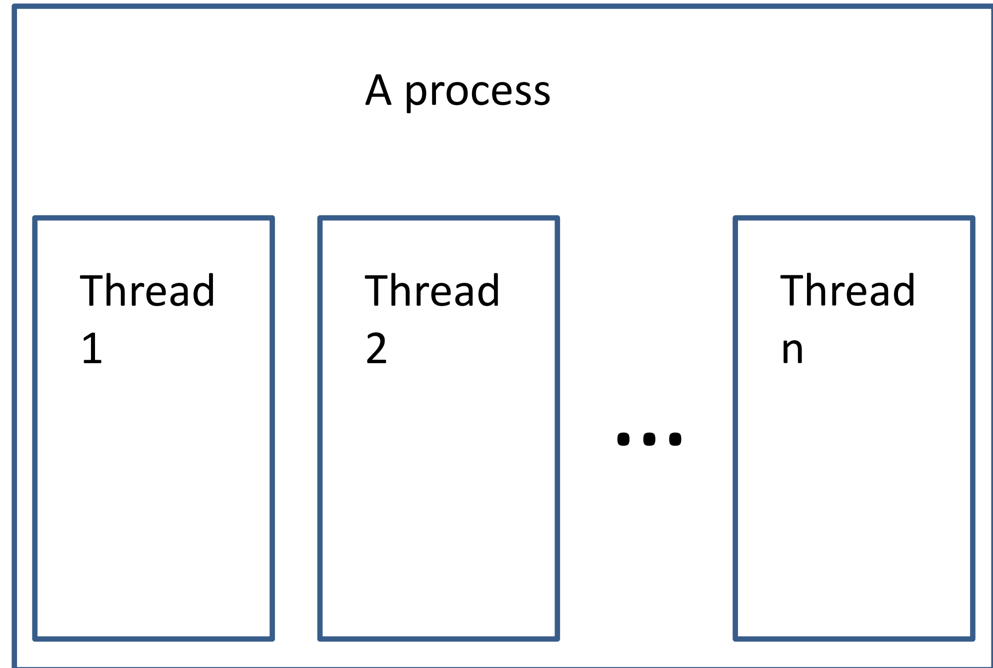
Only a scheduler entry is created.

Everything else is shared.

No protection between threads.

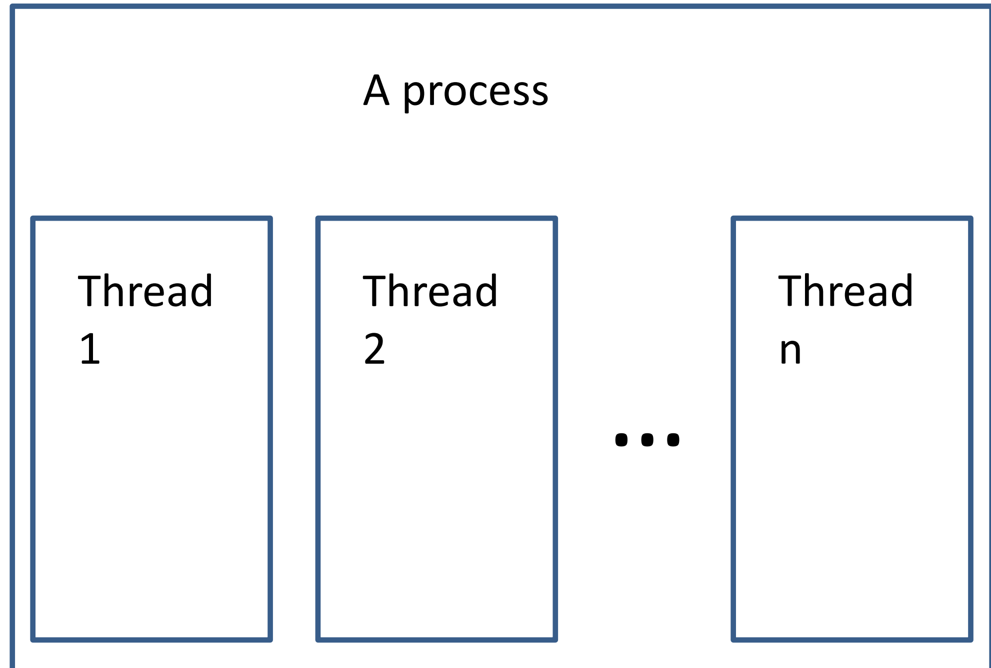


A thread is simple flow of control that can be separately scheduled.



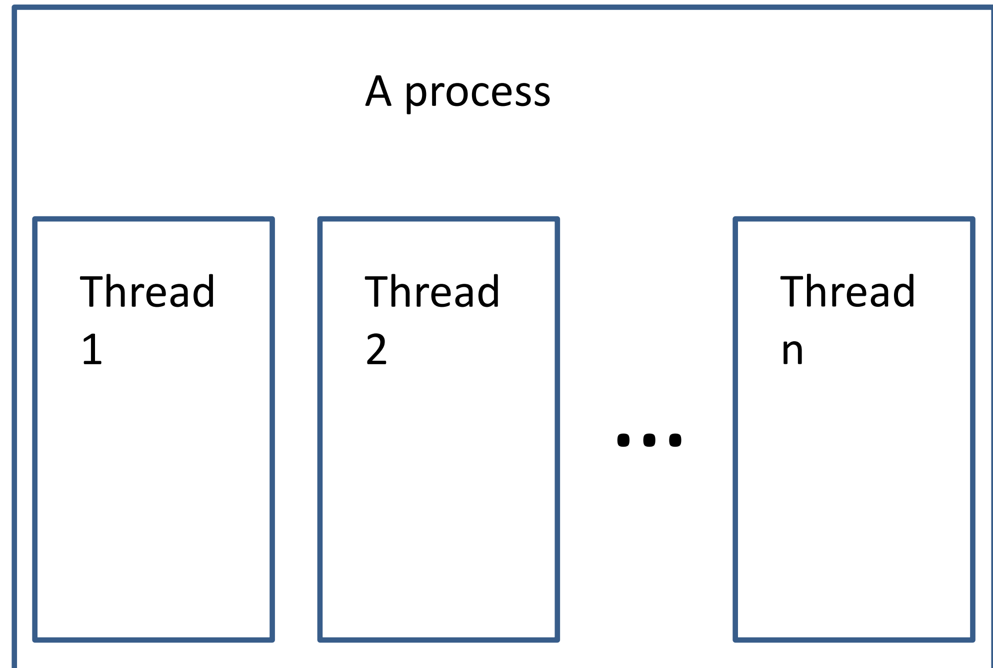
A thread’s “state” consists of:

1. An instruction pointer,
2. A stack,
3. A register set,
4. Its scheduling priority, and
5. Any semaphores it owns.



Every other thread within the process shares:

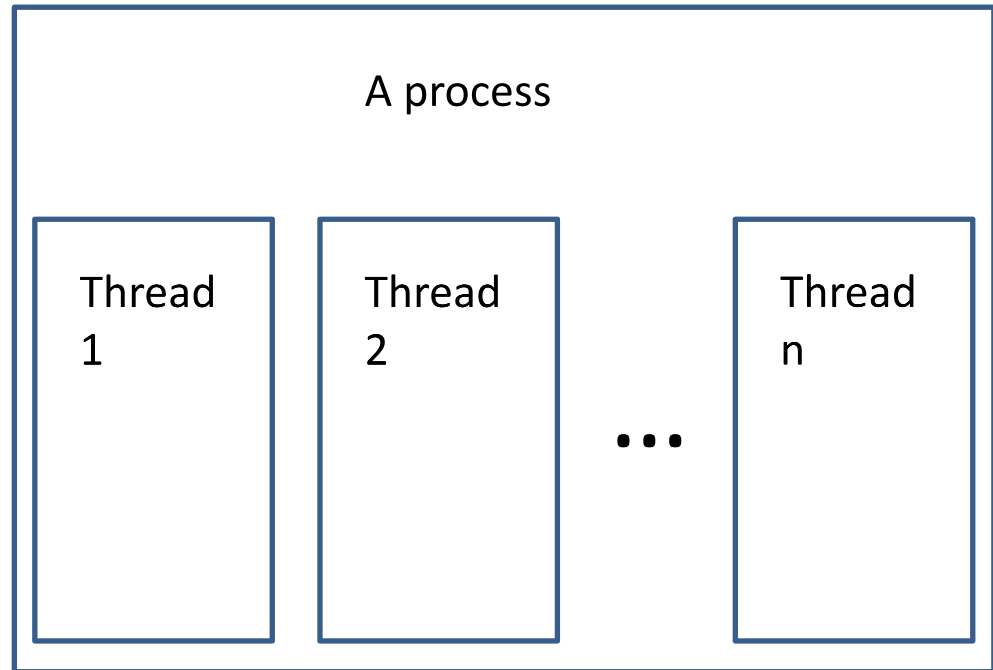
1. Memory (instructions and data),
2. Open handles to files, processes, pipes, etc.,
3. Current directory, and
4. Environment variables.



A child thread begins completely asynchronously unless you create it in a suspended state.

If you have an SMP, the kernel may transparently run any given thread on any given processor.

Usually there's "affinity" for the last processor a thread on which a thread ran.



The argument for threads

1. Allows overlapped activities.
2. Slow activities like I/O can be moved off the critical path.
3. Just because I/O has stalled doesn't mean you can't do other things while you wait.
4. Much lighter cost to create a thread than to create a process.
5. Lower context switching cost when the scheduler picks a new thread.
6. Much less cost to share objects between threads because they all share the same memory space.

```
$ head -1 LinuxHelloMT.cpp
// Simple multi-threaded hello world program.
$ g++ LinuxHelloMT.cpp -pthread -o LinuxHelloMT
$ ./LinuxHelloMT
Starting child
Waiting for child
Hello from the child!
Child has exited
$
```

```c
#include <pthread.h>

int pthread_create(pthread_t *thread, const pthread_attr_t *attr,
         void *(*start_routine) (void *), void *arg);

int pthread_join(pthread_t thread, void **retval);
```

The pthread_create() function starts a new thread in the calling process. The new thread starts execution by invoking start_routine(); arg is passed as the sole argument of start_routine().

The pthread_join() function waits for the thread specified by thread to terminate. If that thread has already terminated, then pthread_join() returns immediately.

```cpp
// Simple multi-threaded Linux hello world program.

#include <stdlib.h>
#include <pthread.h>
#include <iostream>
using namespace std;

void *Hello( void *p )
   {
   cout << "Hello from the child!" << endl;
   }

int main( int argc, char **argv )
   {
   cout << "Starting child" << endl;
   pthread_t child;

   pthread_create( &child, nullptr, Hello, nullptr );

   cout << "Waiting for child" << endl;
   pthread_join( child, NULL );

   cout << "Child has exited" << endl;
   }
```

Agenda

1. Course details.
2. Homework 1.
3. stat'ing a file.
4. Seeking.
5. Memory-mapping files.
6. Processes.
7. Threads.
8. Bonus: Producer/consumer relationships

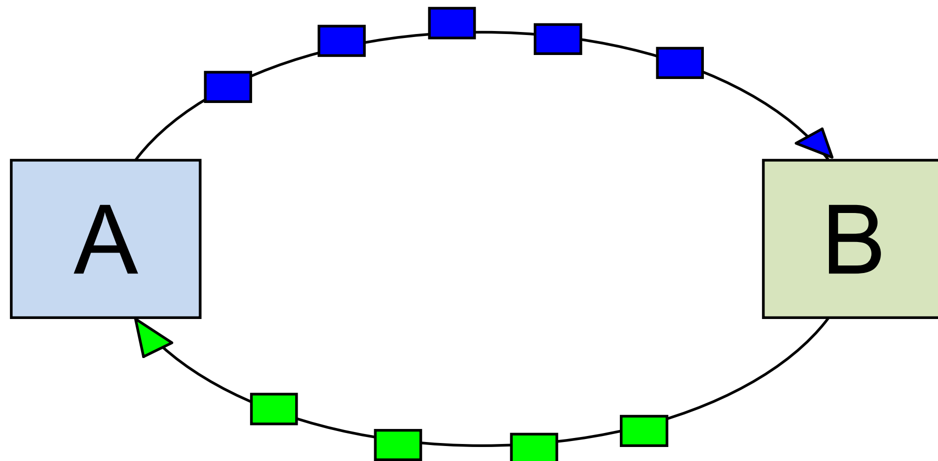
Producer-Consumer relationships

1. Basic notion: Two threads that cooperate so that each consumes what the other produces.
2. Must share data, locking it before any access.
3. Sleeping when waiting for input.

Example

A multi-threaded cat utility that uses one thread to read input and one to write output.

1. Activity A consumes empty buffers and produces full buffers.
2. Activity B consumes full buffers and produces empty buffers.
3. A pool of buffers is used to minimize blocking.



Producer-Consumer Strategy

When an activity needs input,

1. It locks the input list.
2. If the input list is not empty then
   It takes an item and releases the lock.
   else
   It clears the "data available" event,
   Releases the lock,
   Sleeps on "data available
   Starts over at the top.

When an activity has output,

1. It locks the output list,
2. Puts the item on the list, signals "data available" and releases the lock.

```c
#include <semaphore.h>

int sem_init(sem_t *sem, int pshared, unsigned int value);
```

sem_init( ) initializes an unnamed semaphore.

```c
#include <pthread.h>

int pthread_mutex_lock(pthread_mutex_t *mutex);
int pthread_mutex_trylock(pthread_mutex_t *mutex);
int pthread_mutex_unlock(pthread_mutex_t *mutex);
```

pthread_mutex_lock, pthread_mutex_trylock, pthread_mutex_unlock — lock and unlock a mutex

```c
#include <semaphore.h>

int sem_wait(sem_t *sem);

int sem_trywait(sem_t *sem);

int sem_timedwait(sem_t *sem, const struct timespec *abs_timeout);
```

sem_wait, sem_timedwait, sem_trywait allow you to lock a semaphore.

```c
#include <semaphore.h>

int sem_post(sem_t *sem);
```

sem_post( ) unlocks a semaphore.

```
$ head -1 LinuxCatMT.cpp
// Linux multi-threaded cat routine.
$ g++ LinuxCatMT.cpp -pthread -o LinuxCatMT
$ wc LinuxCatMT.cpp
 135  346 2688 LinuxCatMT.cpp
$ ./LinuxCatMT < LinuxCatMT.cpp | wc
    135     346    2688
$
```

```
// Linux multi-threaded cat routine.

#include <unistd.h>
#include <stdlib.h>
#include <pthread.h>
#include <semaphore.h>
#include <cassert>



struct Buffer
   {
   char Block[ 1024 ];
   ssize_t Length;
   };


template< typename T > struct Node
   {
   Node *next;
   T Data;

   Node( ) : next( nullptr )
      {
      }
   };
```

```cpp
template< typename T > class SharedList
   {
   private:

      Node< T > *top, *bottom;
      sem_t available;
      pthread_mutex_t lock;

   public:

      SharedList( ) : top( nullptr ), bottom( nullptr )
         {
         pthread_mutex_init( &lock, nullptr );
         // Mac OSX:  available = sem_open( "/semaphore", O_CREAT, 0644, 1 ) );
         sem_init( &available, 0, 0 );
         }

      ~SharedList( )
         {
         pthread_mutex_destroy( &lock );
         sem_destroy( &available );
         }
```

```
   Node< T > *Get( )
      {
      Node< T > *a;
      sem_wait( &available );
      pthread_mutex_lock( &lock );
      a = top;
      assert( a );
      if ( ( top = a->next ) == nullptr )
         bottom = nullptr;
      a->next = nullptr;
      pthread_mutex_unlock( &lock );
      return a;
      }

   void Put( Node< T > *a )
      {
      pthread_mutex_lock( &lock );
      if ( bottom )
         bottom = bottom->next = a;
      else
         top = bottom = a;
      sem_post( &available );
      pthread_mutex_unlock( &lock );
      }
};
```

```
SharedList< Buffer > Empty, Full;


void *Reader( void *p )
   {
   Node< Buffer > *e;
   ssize_t length;

   do
      {
      e = Empty.Get( );
      e->Data.Length = read( 0, e->Data.Block, sizeof( e->Data.Block ) );
      length = e->Data.Length;  /* Why? */
      Full.Put( e );
      }
   while ( length > 0 );
   }


void Writer( void )
   {
   Node< Buffer > *f;
   ssize_t length;

   while ( f = Full.Get( ), f->Data.Length > 0 )
      {
      write( 1, f->Data.Block, f->Data.Length );
      Empty.Put( f );
      }
   }
```

```
int main( int argc, char **argv )
   {
   SharedList< Buffer > empty, full;
   // put 5 empty nodes on the empty list;
   for ( int i = 5; i--;  )
      Empty.Put( new Node< Buffer > );

   pthread_t child;

   /* Spawn the reader as a child thread. */
   pthread_create( &child, nullptr, Reader, nullptr );

   /* Do the writing in this thread. */
   Writer();
   }
```