

EECS 498 System Design of a Search Engine

Winter 2026

Lecture 8: Processes and threads

Nicole Hamilton

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

agenda

1. details
2. strategy
3. processes
4. threads
5. producer-consumer relationships

agenda

1. details
2. strategy
3. processes
4. threads
5. producer-consumer relationships

details

1. The group photo survey is now live.
2. You can use the basic C runtime library but you don't need any STL beyond `<string>`, `<vector>` and `<queue>` to do any of the homework.
3. Please do not use STL wrappers around pthread primitives.
4. You may be able to do something much smarter and more performant in your project but the homework is autograded and it what it is.
5. Time to get started on the reading.

Hello, Aardvarks.

Here is how your classmates rated your team photo on a 1-to-10 scale. Ten is best.

Survey question	Team average	Class average
Faces are big enough and clear enough that everyone is easily recognizable. Everyone is identified and their names and the name of their team are easily readable.	9.85	9.24
It's original. It's clearly their own work. They haven't just pasted their faces onto someone else's artwork.	9.23	7.69
It's attractive and visually appealing. It's a nice piece of artwork. This could be a poster.	9.37	7.04
It's funny, clever, or creative. They've chosen a clever name and concept.	9.63	7.53
Excellent technical execution. Good composition, use of color, seamless photo editing. They put some effort into this and it shows.	9.13	7.69
Overall average	9.44	7.84
Scaled and rounded to 100 points	94	78

Comments

1. Wow. Blew me away.
2. Great use of turtles on skateboards versus cats.
3. A little obvious AI.
4. You're mean to cats.

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 T. H. Crowley	1897
Foreword M. D. McIlroy, E. N. Pinson, and B. A. Tague	1899
The UNIX Time-Sharing System D. M. Ritchie and K. Thompson	1905
UNIX Implementation K. Thompson	1931
A Retrospective D. M. Ritchie	1947
The UNIX Shell S. R. Bourne	1971
The C Programming Language D. M. Ritchie, S. C. Johnson, M. E. Lesk, and B. W. Kernighan	1991
Portability of C Programs and the UNIX System S. C. Johnson and D. M. Ritchie	2021
The MERT Operating System H. Lycklama and D. L. Bayer	2049
UNIX on a Microprocessor H. Lycklama	2087
A Minicomputer Satellite Processor System H. Lycklama and C. Christensen	2103
Document Preparation B. W. Kernighan, M. E. Lesk, and J. F. Ossanna, Jr.	2115
Statistical Text Processing L. E. McMahon, L. L. Cherry, and R. Morris	2137
Language Development Tools S. C. Johnson and M. E. Lesk	2155
(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.

365

Electronic version recreated by Eric A. Brewer
University of California at Berkeley

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

Communications
of
the ACM

July 1974
Volume 17
Number 7

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.

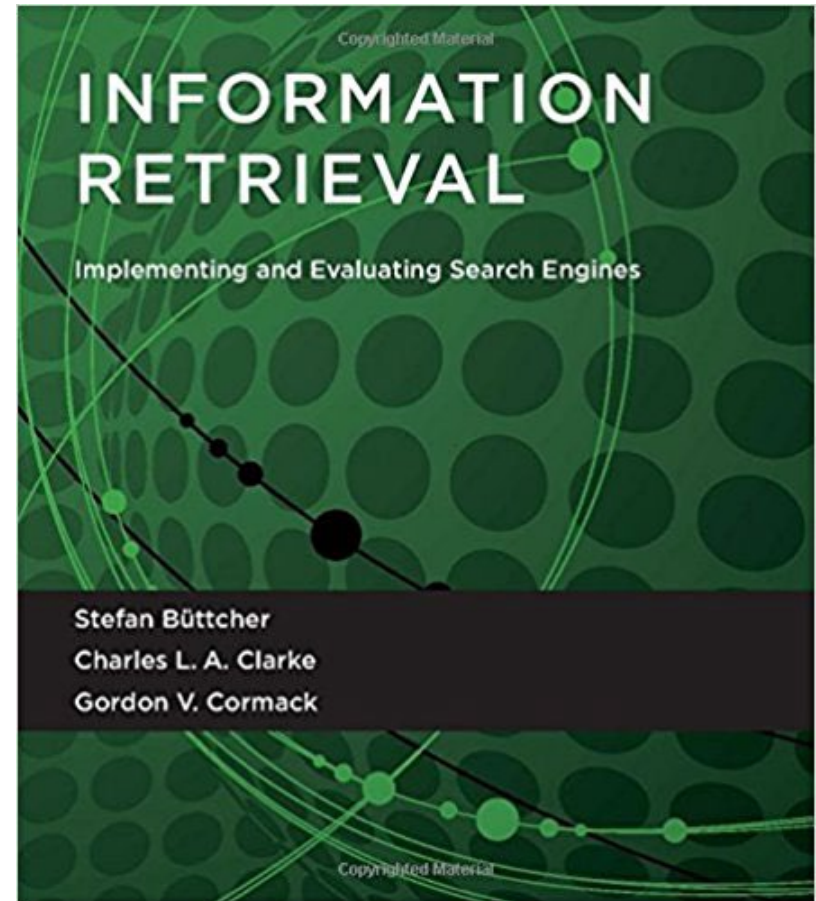
The dynamic ranker compiles the query, searches the index for matching pages and returns a list of the n best pages, each with a score representing its estimated quality as a result. It may also be instrumented to return various levels of debug information, e.g., to allow the scoring calculation be re-run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

I recommend the first 6 chapters.

Posted to Canvas.



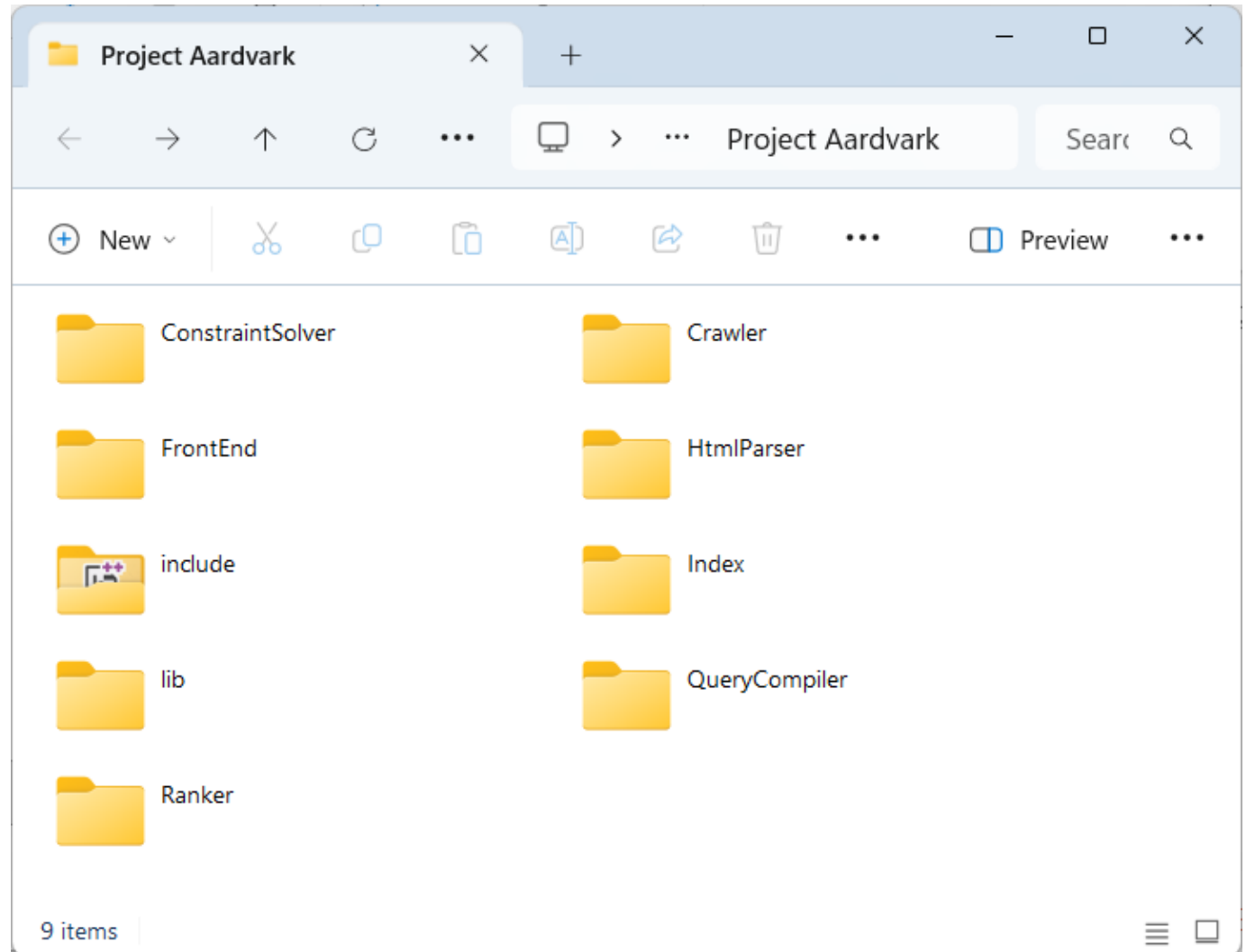
agenda

1. details
2. strategy
3. processes
4. threads
5. producer-consumer relationships

when /where do you start?

Start now.

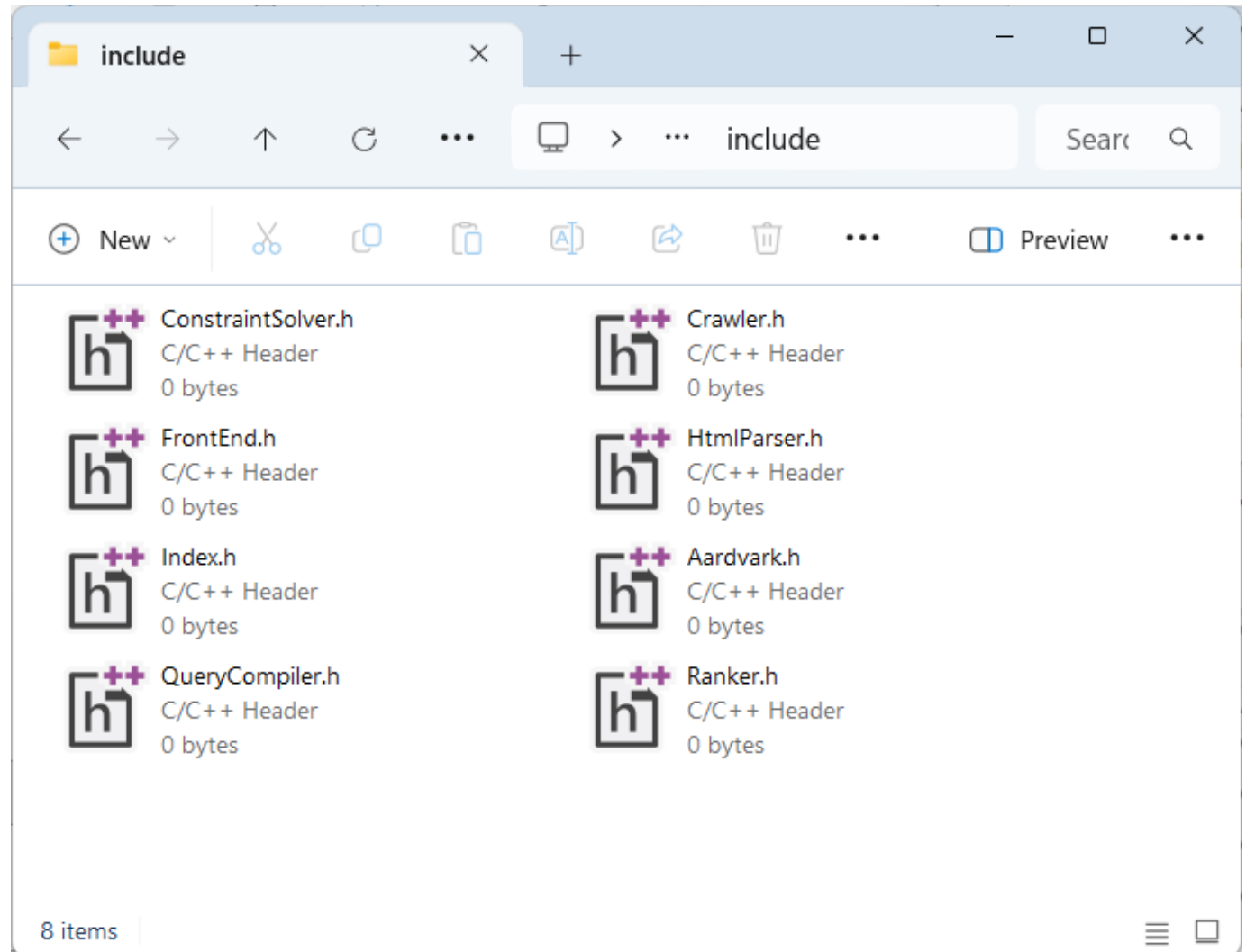
Begin by
laying out
your directory
structure.



define your interfaces

Create header files that describe the external interfaces.

Agree on the classes and functions you intend to build.



strategy

Expect all your laptops will run the same code, top to bottom.

(If you need a master and your machine is not the master, how would you debug?)

Each should be a complete but standalone engine with all parts.

Get everything working standalone.

distribute crawling

Divide the web across your laptops.

(How could you do that?)

When you discover a new URL, decide which machine owns it.

If it's yours, add it to your frontier.

If it's not, add it to list for the machine that does own it.

Periodically batch transfer URLs for the other machines.

distribute querying

When you get a query, broadcast it to the sibling machines.

(Do you send the query string or a compiled version?)

Search your own slice of the web.

Merge the results from the sibling machines.

agenda

1. details
2. strategy
3. processes
4. threads
5. producer-consumer relationships

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

Linux uses a sequence of two calls to start a process:

1. `fork( )` creates a copy of current process.
2. `exec( program, args )` replaces current address space with specified program.

Why first copy the process only to overwrite it?

Allows sharing of code, file descriptors, other state information and results in a simple interface.

Windows by contrast, uses a single `CreateProcess( )` system call, but requires a very complex set of arguments to deal with all the possible cases.

Windows CreateProcess

There is no fork().

Creates the child running a new executable, returns a handle to the child.

argv is passed as a string, not an array.

Child process retrieves the command line with GetCommandLine(). C runtime turns that into argc, argv.

Slightly complex rules for words containing spaces or quotes.

Lots of options for debugging, etc.

Two versions.

```
BOOL CreateProcessA(  
    LPCSTR          lpApplicationName,  
    LPSTR           lpCommandLine,  
    LPSECURITY_ATTRIBUTES lpProcessAttributes,  
    LPSECURITY_ATTRIBUTES lpThreadAttributes,  
    BOOL            bInheritHandles,  
    DWORD           dwCreationFlags,  
    LPVOID          lpEnvironment,  
    LPCSTR          lpCurrentDirectory,  
    LPSTARTUPINFOA  lpStartupInfo,  
    LPPROCESS_INFORMATION lpProcessInformation  
);
```

```
BOOL CreateProcessW(  
    LPCWSTR          lpApplicationName,  
    LPWSTR           lpCommandLine,  
    LPSECURITY_ATTRIBUTES lpProcessAttributes,  
    LPSECURITY_ATTRIBUTES lpThreadAttributes,  
    BOOL            bInheritHandles,  
    DWORD           dwCreationFlags,  
    LPVOID          lpEnvironment,  
    LPCWSTR          lpCurrentDirectory,  
    LPSTARTUPINFOW  lpStartupInfo,  
    LPPROCESS_INFORMATION lpProcessInformation  
);
```

process creation

Linux uses a sequence of two calls to start a process:

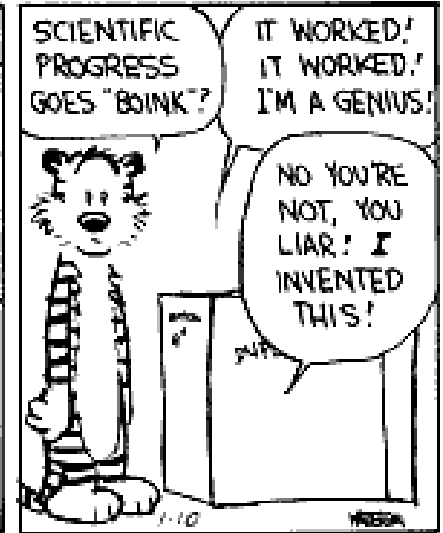
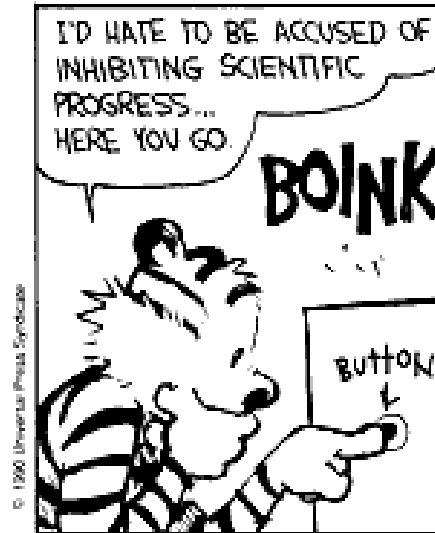
1. `fork( )` creates a copy of current process.
2. `exec( program, args )` replaces current address space with specified program.

Any problems with child being an *exact* clone of parent?

cloning



BROTHER! YOU DOUBTING THOMASES GET IN THE WAY OF MORE SCIENTIFIC ADVANCES WITH YOUR STUPID ETHICAL QUESTIONS! THIS IS A **BRILLIANT** IDEA! HIT THE BUTTON, WILL YA?



Need to know which is the parent and which is the child.

fork and exec

Fork uses the return code to differentiate parent from child.

Child gets return code 0.

Parent gets child's unique process id or *pid*, which it can wait on for the child to complete.

```
int pid = fork( );
if ( pid == 0 )
{
    exec ();  /* child */
}
else
{
    /* parent */
    int status;
    wait( pid, &status, 0 );
}
```

fork bombs

Can easily overwhelm the system by creating thousands of forked copies with malicious code.

In C++:

```
for ( int i = 100; i--; )  
    fork( );
```

In bash:

```
:( ){ :|:& };:
```

copying the address space

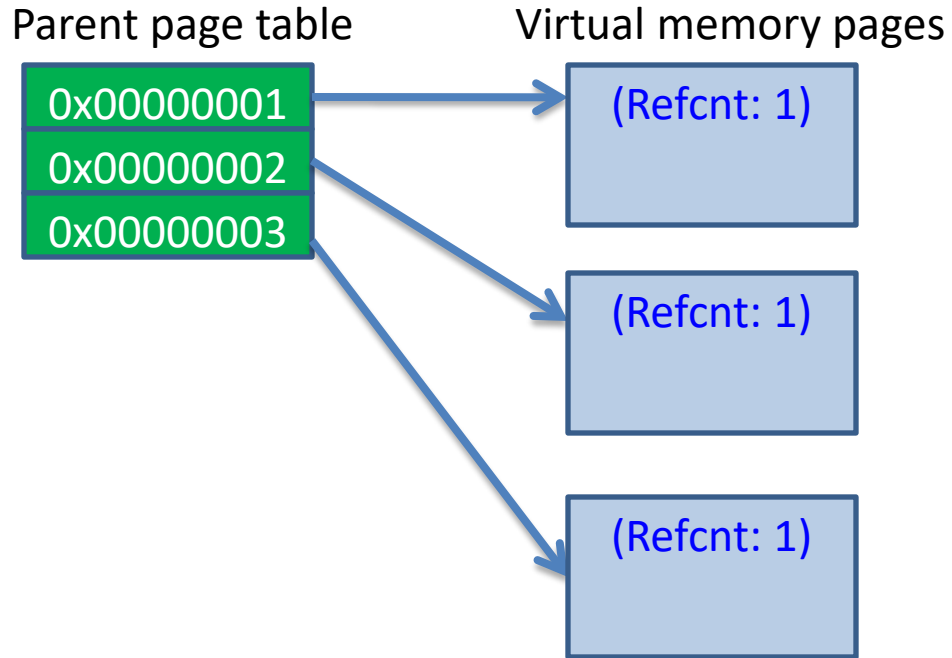
Copying an entire address space is expensive.

But Linux is a virtual memory system.

It uses page tables to keep track of what pages of virtual and physical memory belong to what process, which virtual pages are currently in physical memory, and which would have to be fetched from disk.

This allows it to use [copy-on-write](#).

virtual memory



Parent about to fork().

copy-on-write

Maintain reference count for each physical page.

On fork(), copy only the page table of parent.

Increment reference counts by one.

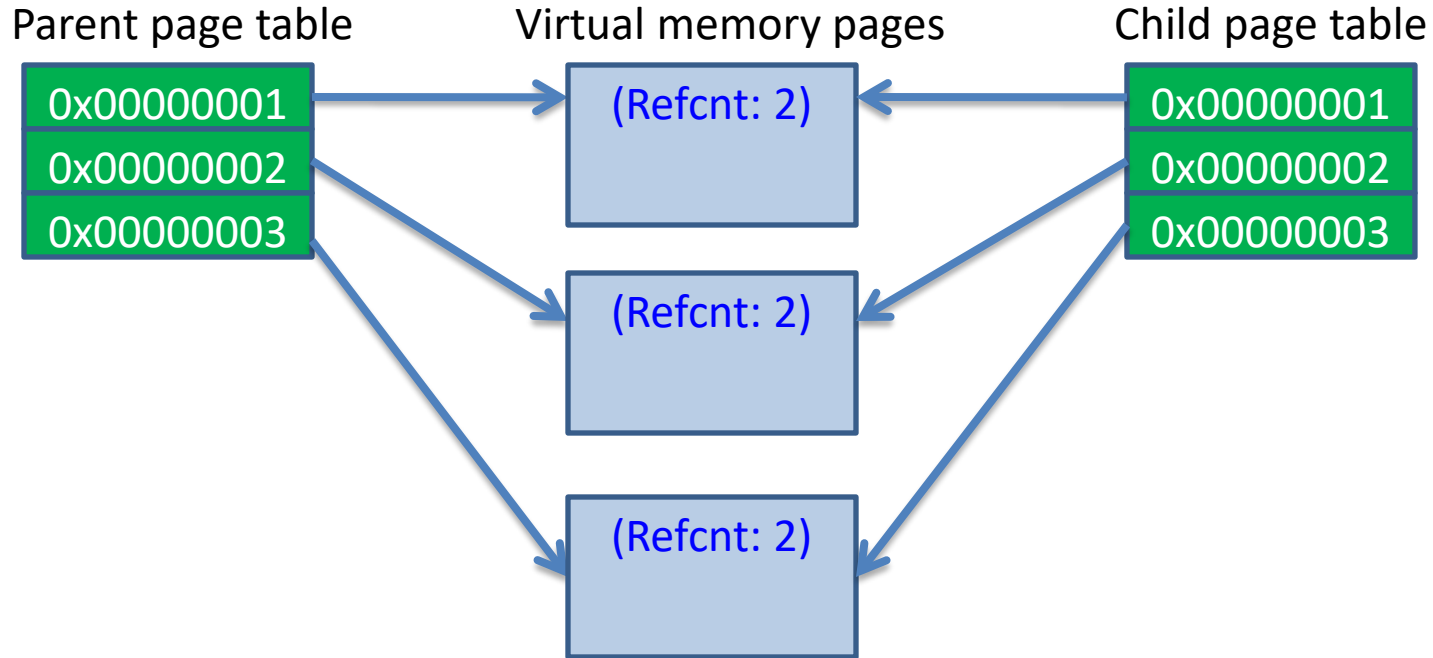
On store by parent or child to page with refcnt > 1:

Make a copy of the page with refcnt of one.

Modify the page table entry of modifier to point to new page.

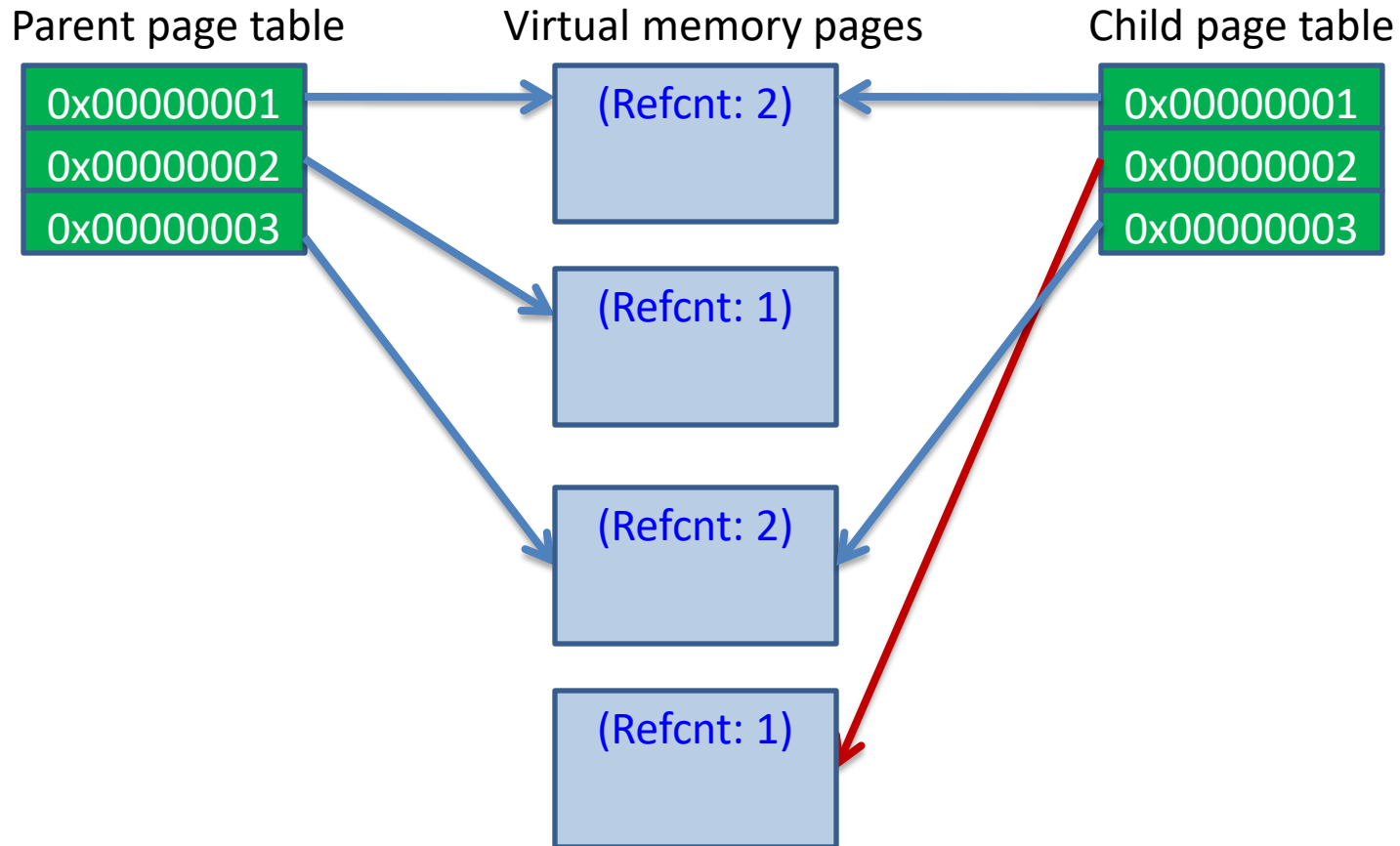
Decrement reference count of old page.

copy-on-write



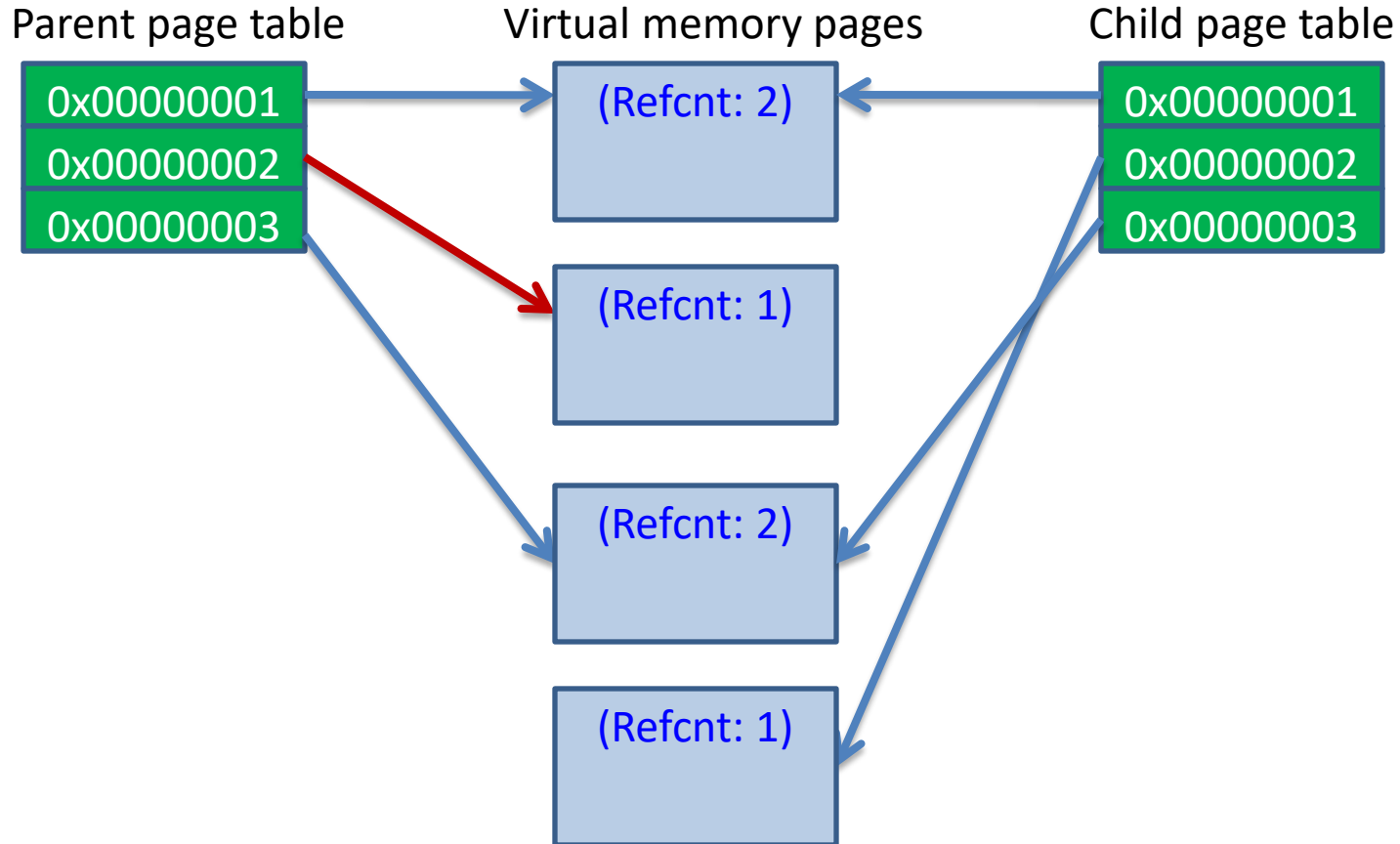
Copy-on-write of parent address space.

copy-on-write



Child modifying page 2 causes a copy to be made.

copy-on-write



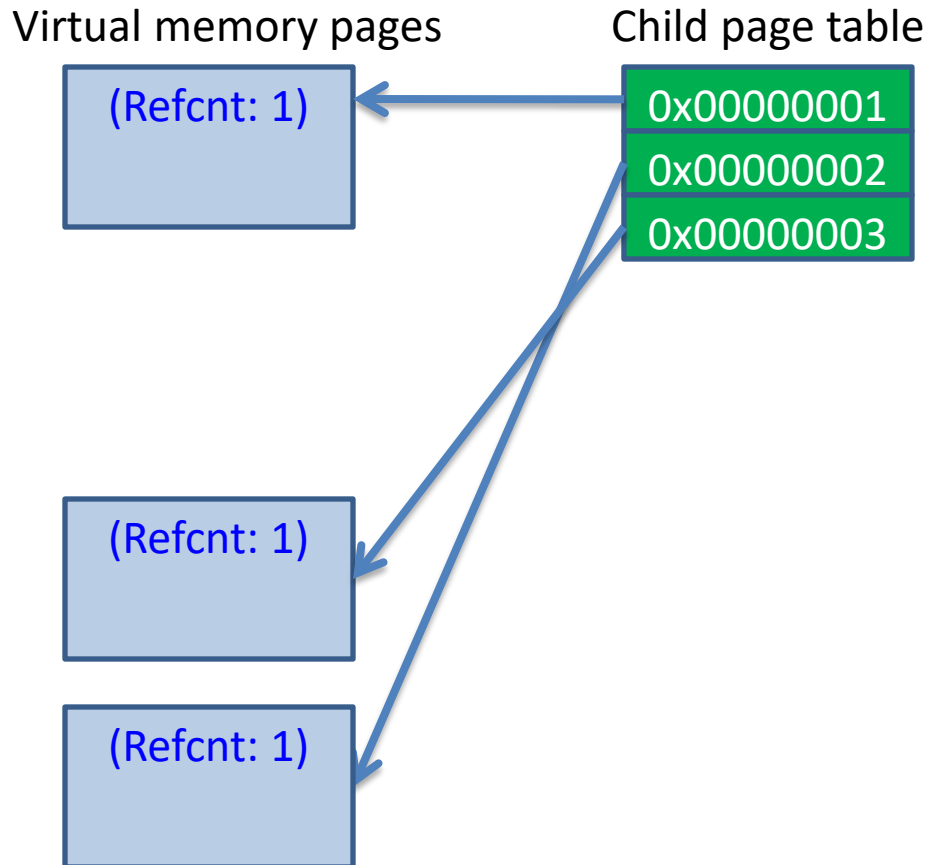
Parent modifying page 2 does not require copying.

copy-on-write

When the parent exits, its page table is deleted and the ref counts decremented.

If a ref count becomes 0, that page is freed.

The child may continue running.



making exec() faster

exec() initializes code in the address space.

Naive solution: read file, copy into memory.

Can we do better?

Observation: most code never accessed.

Load code on-demand from memory-mapped files.

file-backed vs. swap-backed

Swap-backed pages

Block on disk chosen by pager.

A process's writes to a page visible only to that process.

Modifications lost after process exit.

File-backed pages

Block on disk chosen by app.

Any process's write to a page visible to other processes that map the same block.

Modifications persist across process lifetimes.

implementing a shell

Here's how a shell works.

```
while ( true )
{
    print the prompt;
    read the input;
    parse into a command + arguments;
    fork a copy of the shell process;
    if ( child )
    {
        redirect i/o as requested;
        exec the new program with the
            specified arguments;
    }
    else
        //parent
        if ( !background execution )
            wait for child to finish;
}
```

```
$ 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>
```

```
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.

```
#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 execlx(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.

```
#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: LinuxForkExec 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. details
2. strategy
3. processes
4. threads
5. producer-consumer relationships

processes vs. threads

Processes provide concurrency *between* applications.

Threads provide concurrency *within* an application.

Typically, many running processes

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.

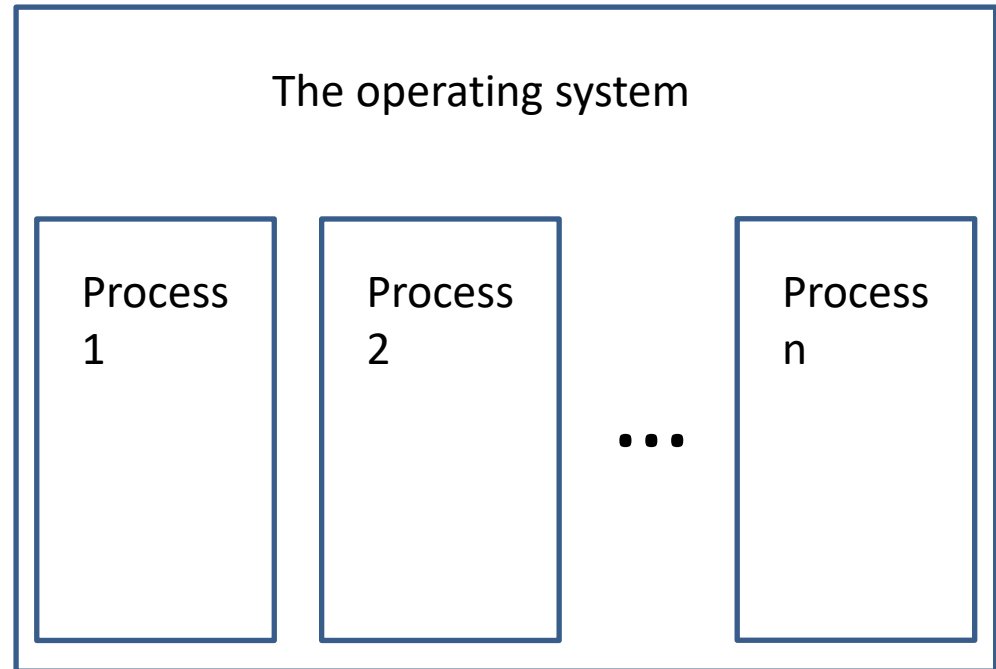
Processes provide concurrency between applications.

High startup costs.

One-way inheritance.

Lots of “firewalling.”

Errant apps can’t scribble on others.



Within a process

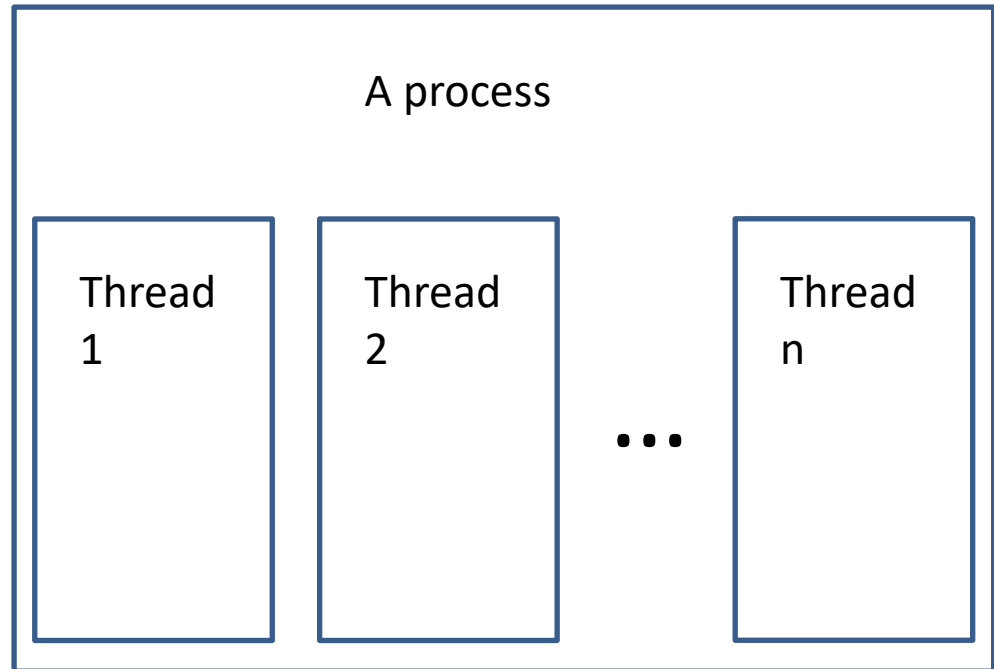
Threads provide concurrency *within* an application:

Very low cost to spawn.

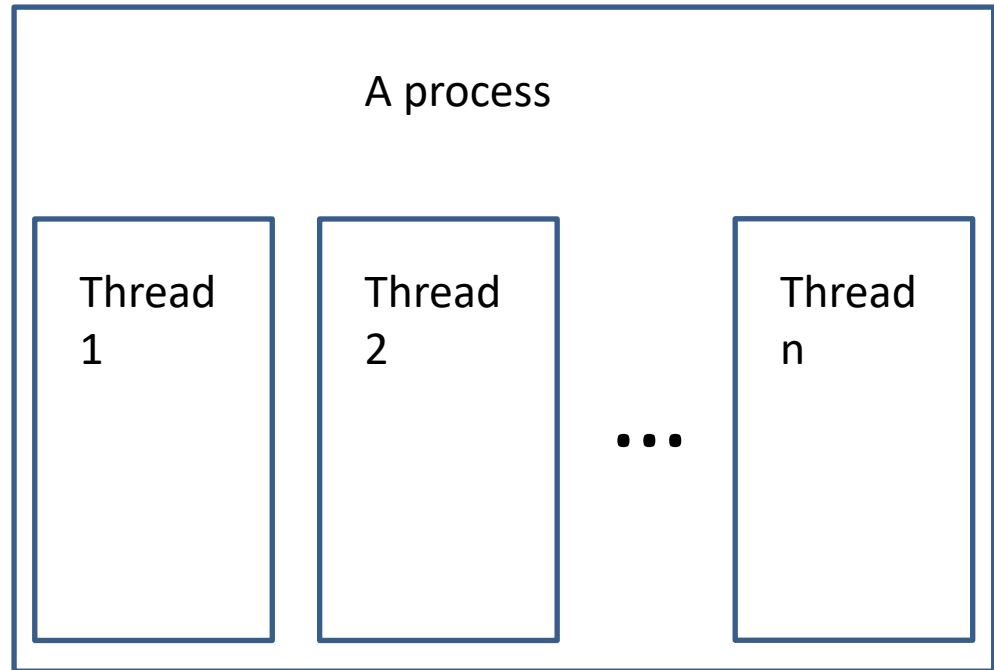
Only a scheduler entry is created.

Everything else is shared.

No protection between threads.

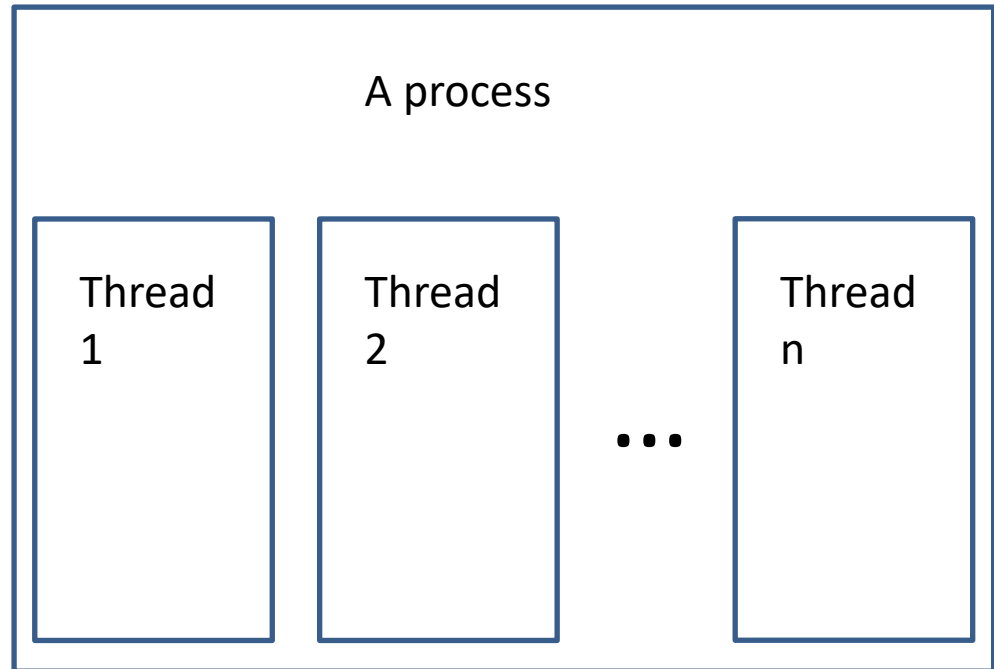


A thread is simple flow of control that can be separately scheduled.



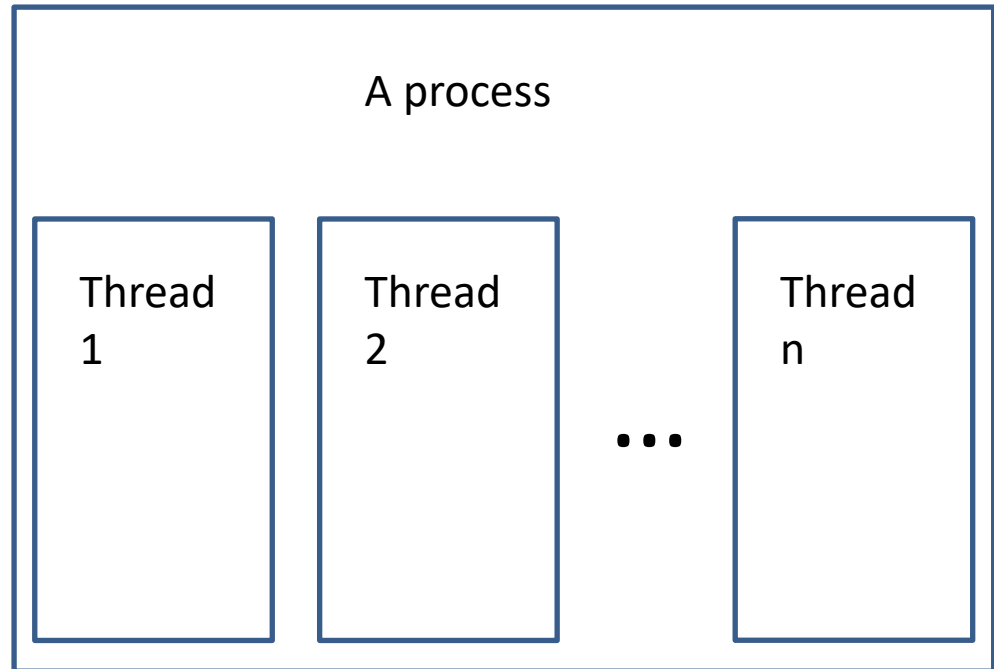
A thread's "state" consists of:

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



Every other thread within the process shares:

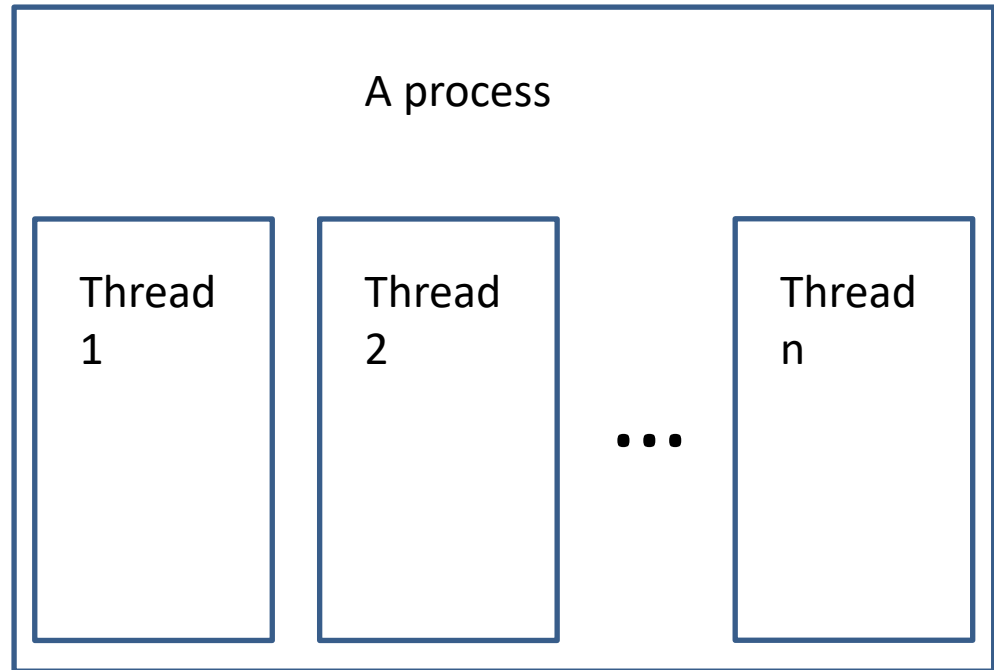
1. Memory (instructions and data),
2. Open handles to files, processes, pipes, etc.,
3. Current directory, and
4. Environment variables.



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.



Within a process

Shared process state
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

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
$
```

```
#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.

```
// 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. details
2. strategy
3. processes
4. threads
5. producer-consumer relationships

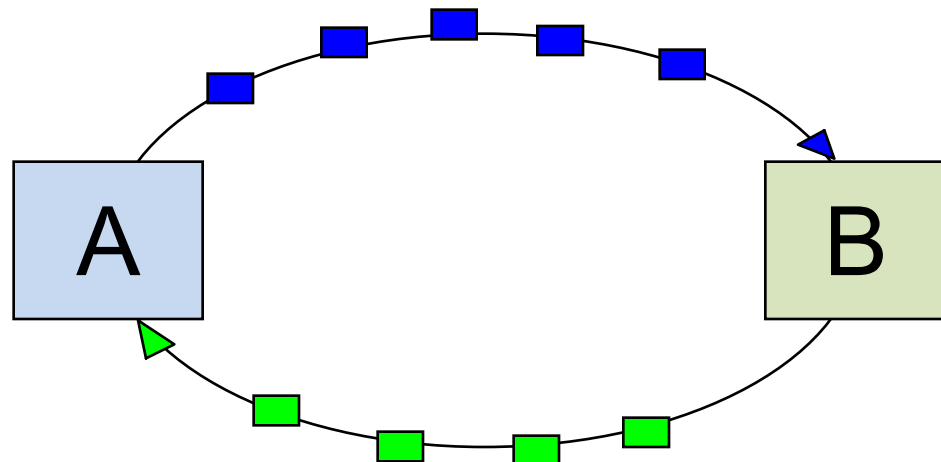
producer-consumer relationships

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.

a multi-threaded cat program

Our multi-threaded cat utility will use 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.

We'll use condition variables for signaling and mutexes for locking.

```
#include <pthread.h>

pthread_cond_t cond = PTHREAD_COND_INITIALIZER;

int pthread_cond_signal(pthread_cond_t *cond);
int pthread_cond_broadcast(pthread_cond_t *cond);
int pthread_cond_wait(pthread_cond_t *cond,
                      pthread_mutex_t *mutex);
int pthread_cond_destroy(pthread_cond_t *cond);
```

A condition variable allows threads to suspend execution until some predicate on shared data is satisfied.

A condition variable must always be associated with a mutex, to avoid the race condition where a thread prepares to wait on a condition variable and another thread signals the condition just before the first thread actually waits on it.

```
tcsh-9% head -1 LinuxCatMT.cpp
// Linux multi-threaded cat routine.
tcsh-10% wc LinuxCatMT.cpp
  131  328 2569 LinuxCatMT.cpp
tcsh-11% g++ --std=c++17 LinuxCatMT.cpp -o LinuxCatMT
tcsh-12% ./LinuxCatMT < LinuxCatMT.cpp | wc
    131    328    2569
tcsh-13%
```

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

```
#include <unistd.h>
#include <stdlib.h>
#include <pthread.h>
```

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

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

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

```

template< typename T > class SharedList
{
private:

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

public:

    SharedList( ) : top( nullptr ), bottom( nullptr )
    {
        pthread_mutex_init( &lock, nullptr );
        available = PTHREAD_COND_INITIALIZER;
    }

    ~SharedList( )
    {
        pthread_mutex_destroy( &lock );
        Node< T > *p;
        while ( top )
        {
            p = top;
            top = top->next;
            delete p;
        }
    }
}

```

```

Node< T > *Get( )
{
    Node< T > *a;
    Node< T > *a;
    pthread_mutex_lock( &lock );
    while ( !( a = top ) )
        pthread_cond_wait( &available, &lock );
    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;
    pthread_cond_signal( &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 );
```

```
return nullptr;
```

```
}
```

```
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( )
{
    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();
}
```