EECS 440 System Design of a Search Engine

Winter 2019

Lecture 9: Threads, locks and producer/consumer relationships

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

Agenda

1. Course details.
2. Processes.
3. Virtual memory.
4. Threads.
5. Locks.
6. Producer-consumer relationships.
7. Multi-reader/single writer locks.

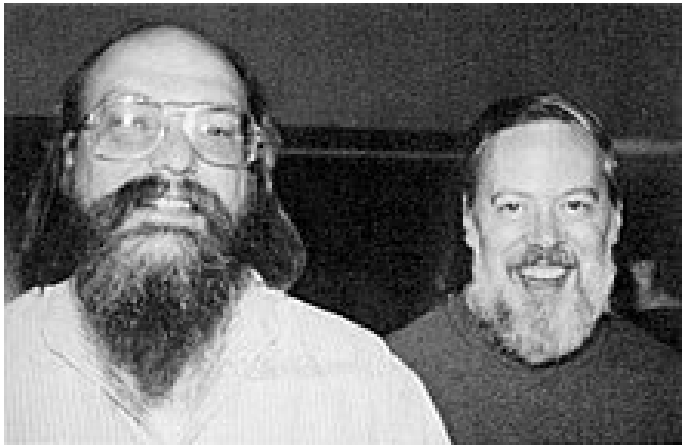
Agenda

1. Course details.
2. Processes.
3. Virtual memory.
4. Threads.
5. Locks.
6. Producer-consumer relationships.
7. Multi-reader/single writer locks.

details

1. HtmlParser AG fixed. All submissions have been rerun.
2. LinuxGetUrl and LinuxGetSsl now due Feb 28.
3. String/vector, one per team, due Mar 7.
4. More shuffling of due dates still possible.
5. Hope to read your bios and your plans this weekend. Apologies for being slow.

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

(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

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

Agenda

1. Course details.
2. Processes.
3. Virtual memory.
4. Threads.
5. Locks.
6. Producer-consumer relationships.
7. Multi-reader/single writer locks.

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.

Unix process creation

System 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.

```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.

```c
#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:  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;
      }
   }
```

Unix process creation

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
);
```

Unix process creation

Why first copy the process only to overwrite it?

Even if it makes for a simpler application programming interface (API), isn't it still expensive and wasteful?

No, because the operating system uses a virtual memory technique called copy-on-write.

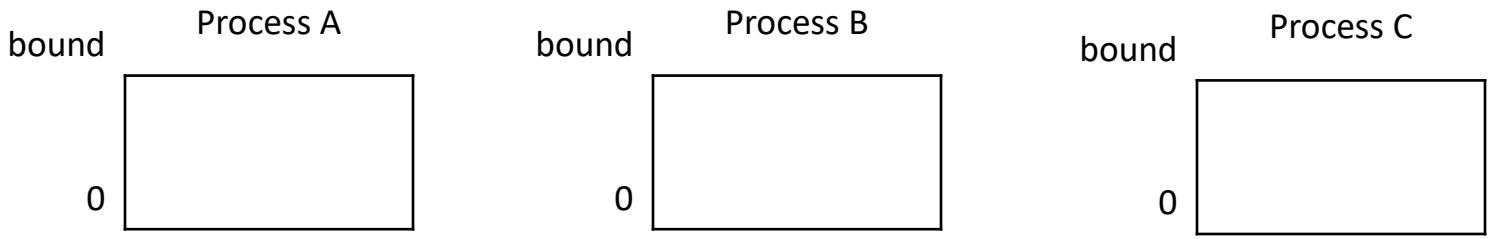
Agenda

1. Course details.
2. Processes.
3. Virtual memory.
4. Threads.
5. Locks.
6. Producer-consumer relationships.
7. Multi-reader/single writer locks.

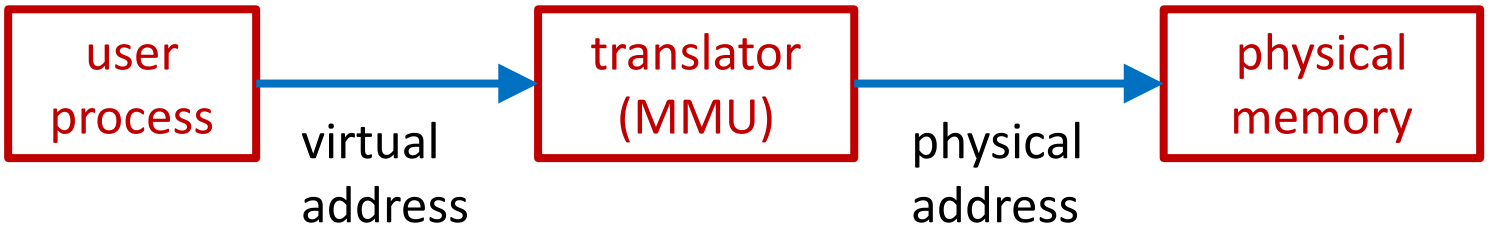
Address Spaces

- Hardware interface:
  - All processes share physical memory

- OS abstraction:



Dynamic address translation



Address independence

Virtual addresses are scoped to 1 process.

Protection

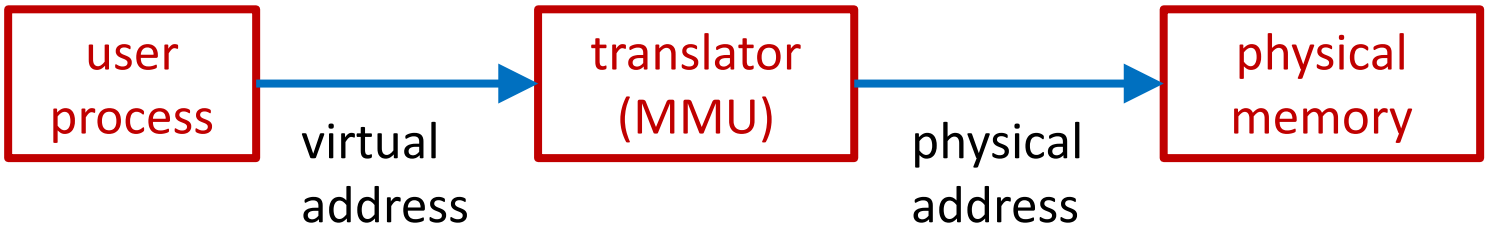
One process can't refer to another's address space.

Virtual memory

VA only needs to be in physical memory when accessed.

Allows changing translations on the fly.

Dynamic address translation

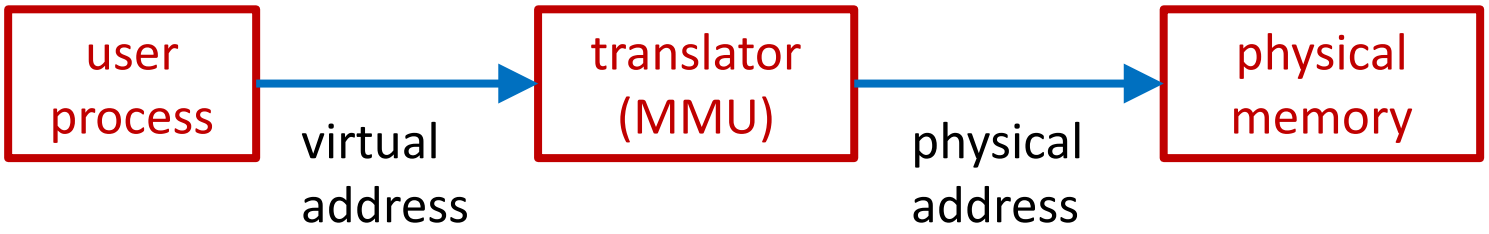


Many ways to implement the translator.

Tradeoffs

1. Flexibility (sharing, growth, virtual memory)
2. Size of data needed to support translation
3. Speed of translation

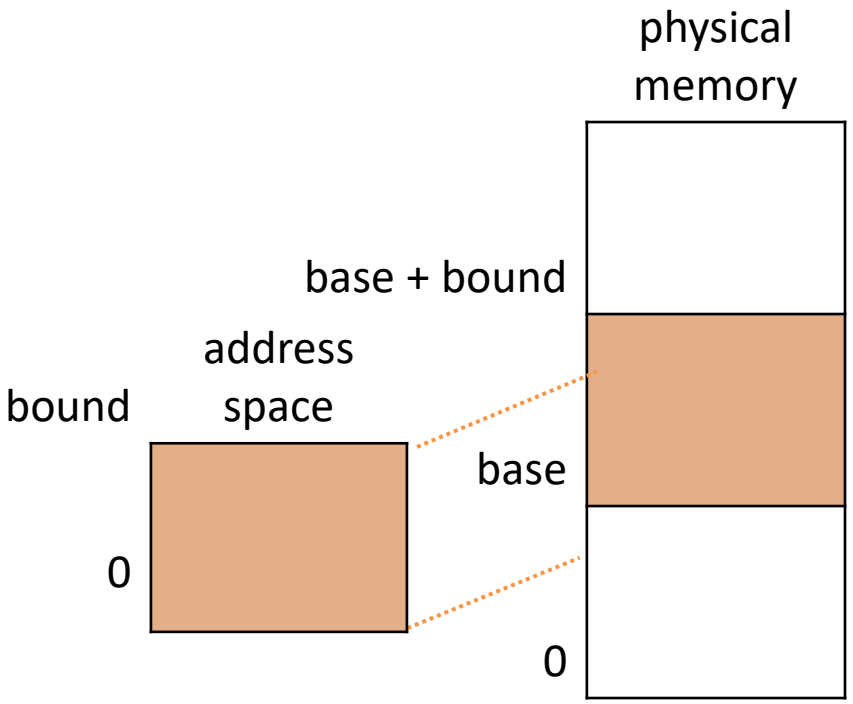
Dynamic address translation



Example MMU strategies:

1. Base and bounds.
2. Segmentation.
3. Paging.

Base and bounds



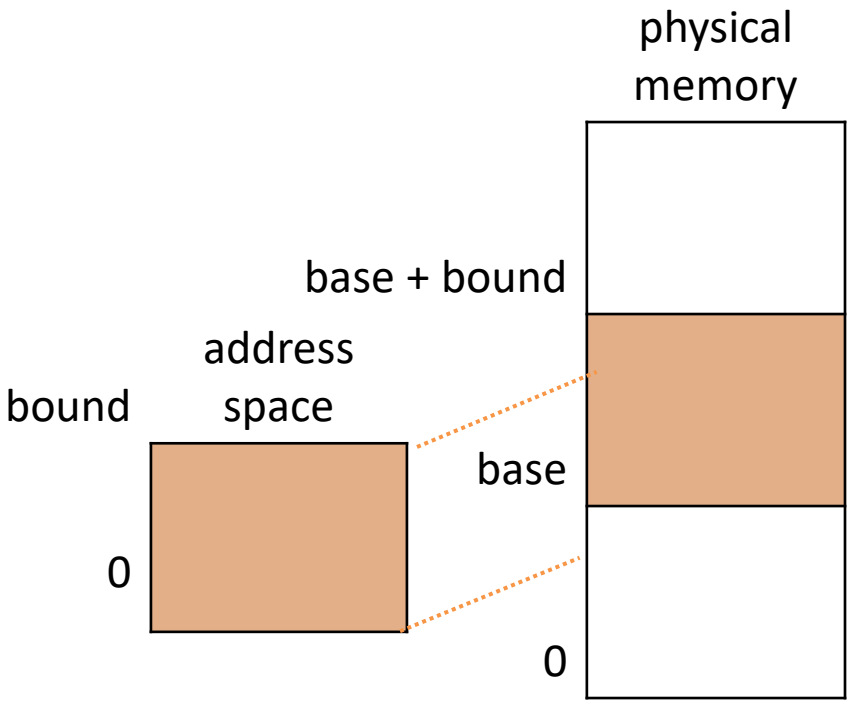
Load each process into a contiguous region of physical memory.

Prevent process from accessing data outside its region.

Base register: starting physical address.

Bound register: size of region.

Base and bounds



Pros:

1. Fast.
2. Simple hardware support.

Cons:

1. No virtual memory.
2. External fragmentation.
3. Hard to selectively grow parts of address space.
4. No controlled sharing.

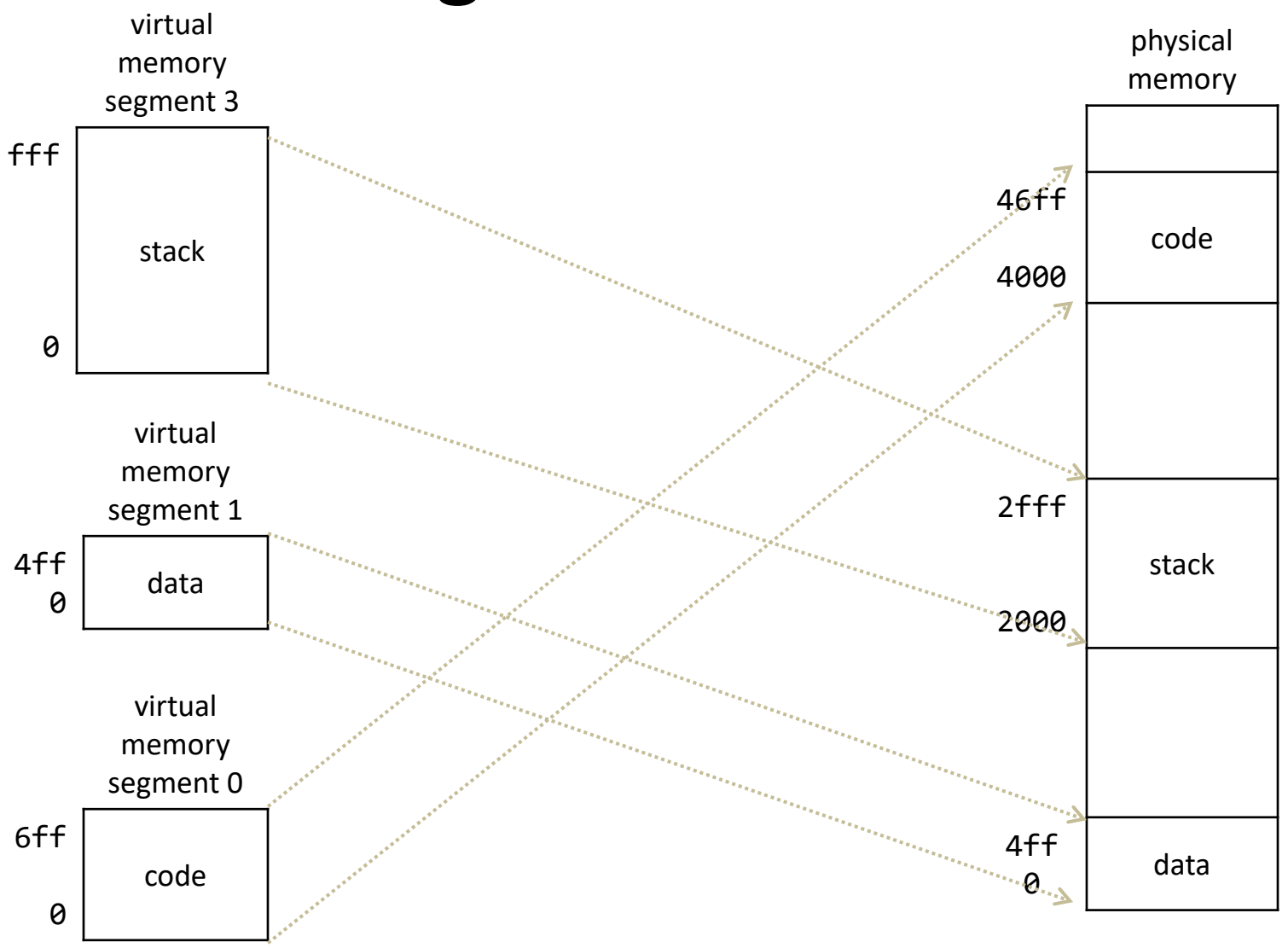
Root cause: Each address space must be contiguous in memory.

Segmentation

Divide address space into segments, regions of memory that are:

1. Contiguous in physical memory.
2. Contiguous in virtual address space.
3. Variable size.

Segmentation



Segmentation

Segment #	Base	Bounds	Description
0	4000	700	code segment
1	0	500	data segment
2	n/a	n/a	unused
3	2000	1000	stack segment

Virtual address is of the form: (segment #, offset)

Physical address = base for segment + offset

Ways to specify the segment number:

1. High bits of address
2. Special register
3. Implicit to instruction opcode

Valid vs. invalid addresses

Segment #	Base	Bounds	Description
0	4000	700	code segment
1	0	500	data segment
2	n/a	n/a	unused
3	2000	1000	stack segment

Not all virtual addresses are valid.

Valid → address is part of virtual address space.

Invalid → virtual address is illegal to access.

Accessing invalid address causes trap to OS.

Reasons for virtual address being invalid?

Invalid segment number.

Offset within valid segment beyond bound.

Protection

Segment #	Base	Bounds	Description
0	4000	700	code segment
1	0	500	data segment
2	n/a	n/a	unused
3	2000	1000	stack segment

Different segments can have different protection.

Code is usually read only (allows fetch, load,...).

Stack and data are usually read/write (allows load, store,...).

Segmentation

Segment #	Base	Bounds	Description
0	4000	700	code segment
1	0	500	data segment
2	n/a	n/a	unused
3	2000	1000	stack segment

Parts of the address space can grow separately.

How would you grow a segment?

If there's contiguous free space, can simply extend the bound.

Otherwise, must move it, perhaps compacting memory.

Benefits of Segmentation

Easy to share part of address space.

Process 1

Segment #	Base	Bounds	Description
0	4000	700	code segment
1	0	500	data segment
3	2000	1000	stack segment

Process 2

Segment #	Base	Bounds	Description
0	4000	700	code segment
1	1000	300	data segment
3	500	1000	stack segment

Segmentation

Pros:

1. Can grow each segment independently.
2. Can share segments across address spaces.

Cons:

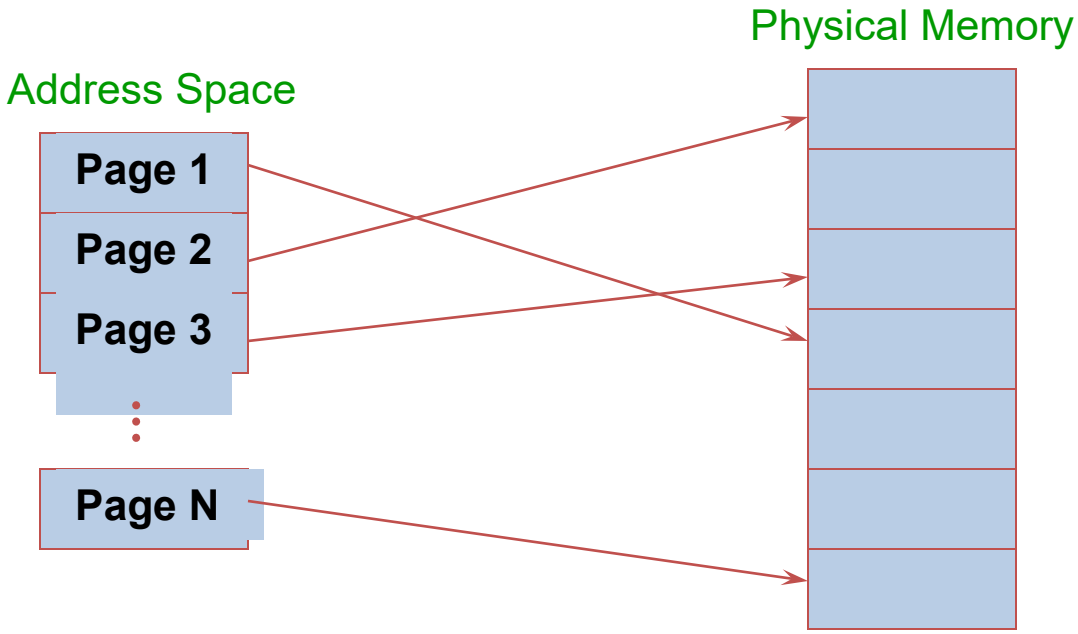
1. Every segment must be smaller than physical memory.
2. Segment allocation is hard.
3. External fragmentation.

Cause: Allocations are of variable amounts of contiguous memory.

Paging

Allocate phys. memory in fixed-size units (pages)

Any free physical page can store any virtual page



Paging

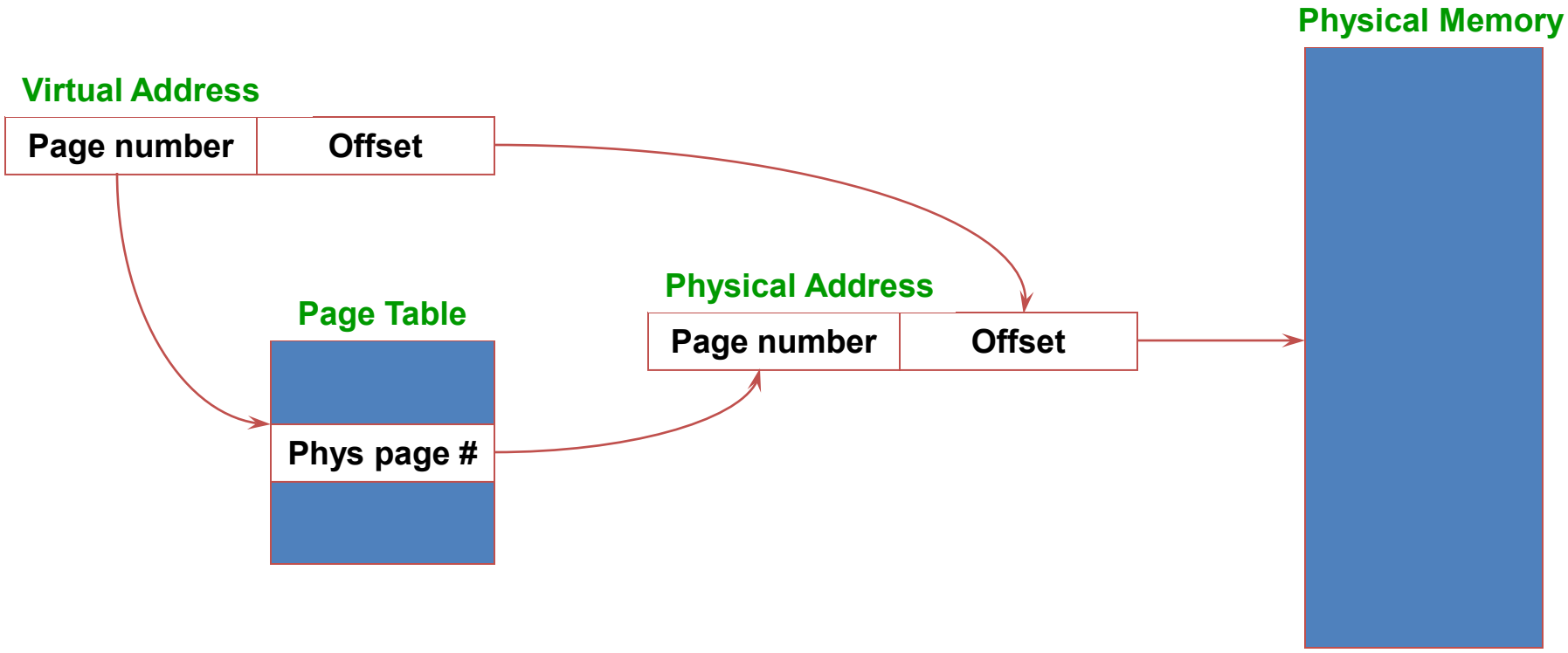
Translation data is the page table.

Virtual address is split into:

1. Virtual page # (high bits of address, e.g., bits 31-12).
2. Offset (low bits of address, e.g., bits 11-0, for 4 KB page size).

Virtual page #	**Physical page #**
0	105
1	15
2	283
3	invalid
...	invalid
1048575	invalid

Page Lookups



Paging

- Pros
  1. Simple memory allocation
  2. Flexible sharing
  3. Easy to grow address space

- Cons
  1. 32-bit virtual address, 4 KB pages, 4 byte PTEs
  2. Page table size?

Page table size

Page size is typically 4 KB or 8 KB.

Some architectures support multiple page sizes.

Each process with a 32-bit address space with 4-byte page table entries requires:

$$\left(\frac{2^{32}}{4096}\right) * 4 = 4\ MB$$

A 64-bit address space with 8-byte entries requires:

$$\left(\frac{2^{64}}{4096}\right) * 8 = 3.6 * 10^{16} = 36\ PB$$

Paging

Pros

1. Simple memory allocation.
2. Flexible sharing.
3. Easy to grow address space.

Cons

1. Large page table size.

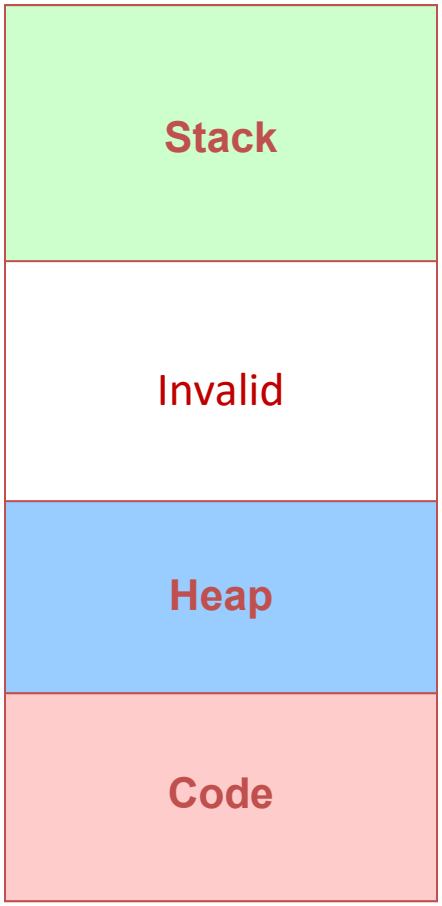
But the vast majority of all page table entries will be marked invalid.

Sparse Address Space

Most processes use only a tiny fraction of their 32 or 64-bit address space.

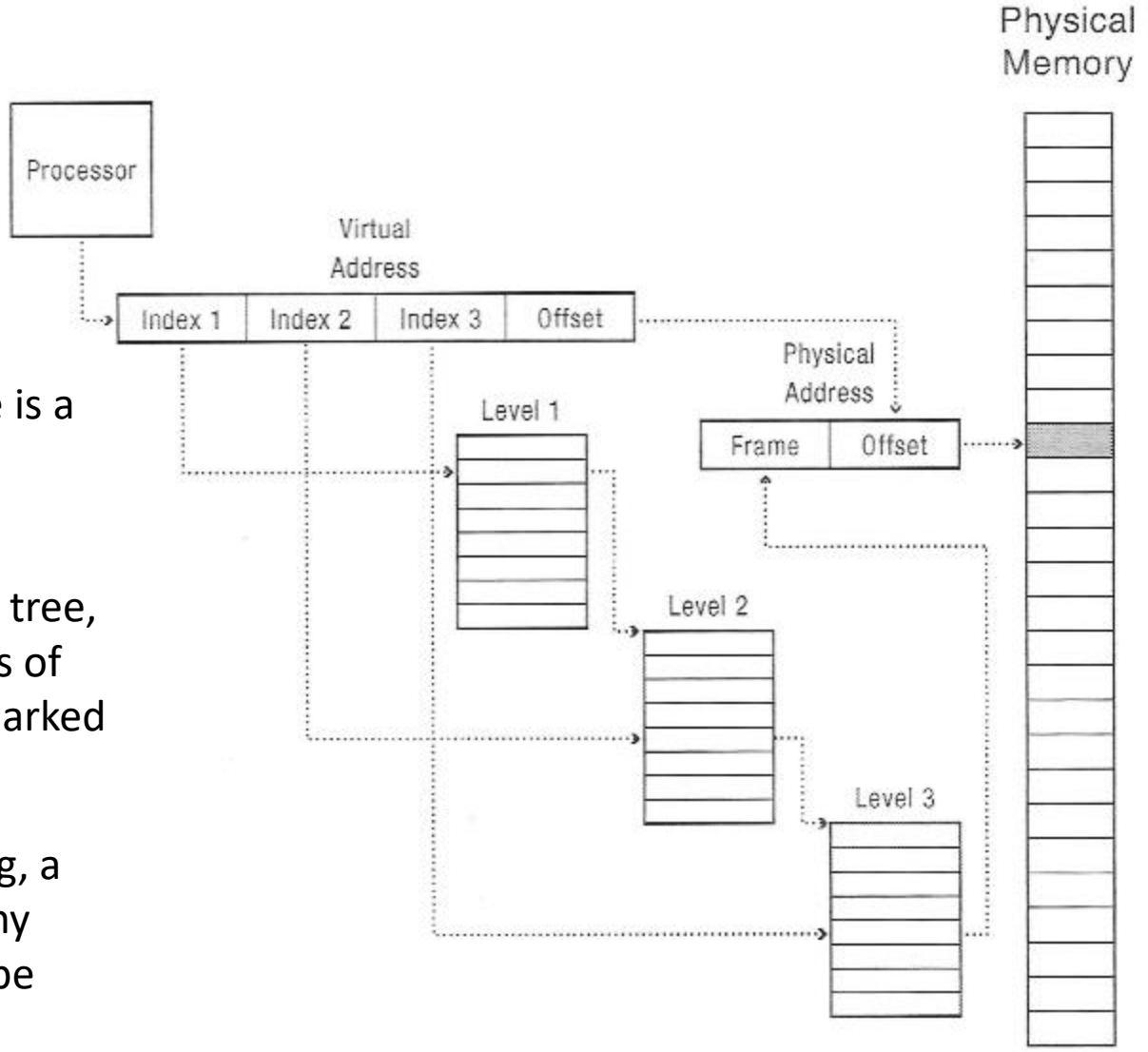
They usually have a huge hole in the middle.

So we only need to represent that part of the page table that isn't marked invalid.



Virtual page #	Physical page #
0	105
1	15
2	283
3	invalid
...	invalid
1048572	invalid
1048573	1078
1048574	48136
1048575	60

Multi-level Paging



A standard page table is a simple array.

Multi-level paging generalizes this into a tree, filling in only the parts of the tree that aren't marked invalid.

With multilevel paging, a lot of the entries in any given page table will be null.

Image source: Anderson & Dahlin, *Operating Systems: Principles and Practice*, p. 398.

When a process starts, a new L1 page table is allocated, then filled in with L2 and L3 leaves as new pages are made valid.

When a process ends, the entire tree of L1, L2 and L3 tables is deleted.

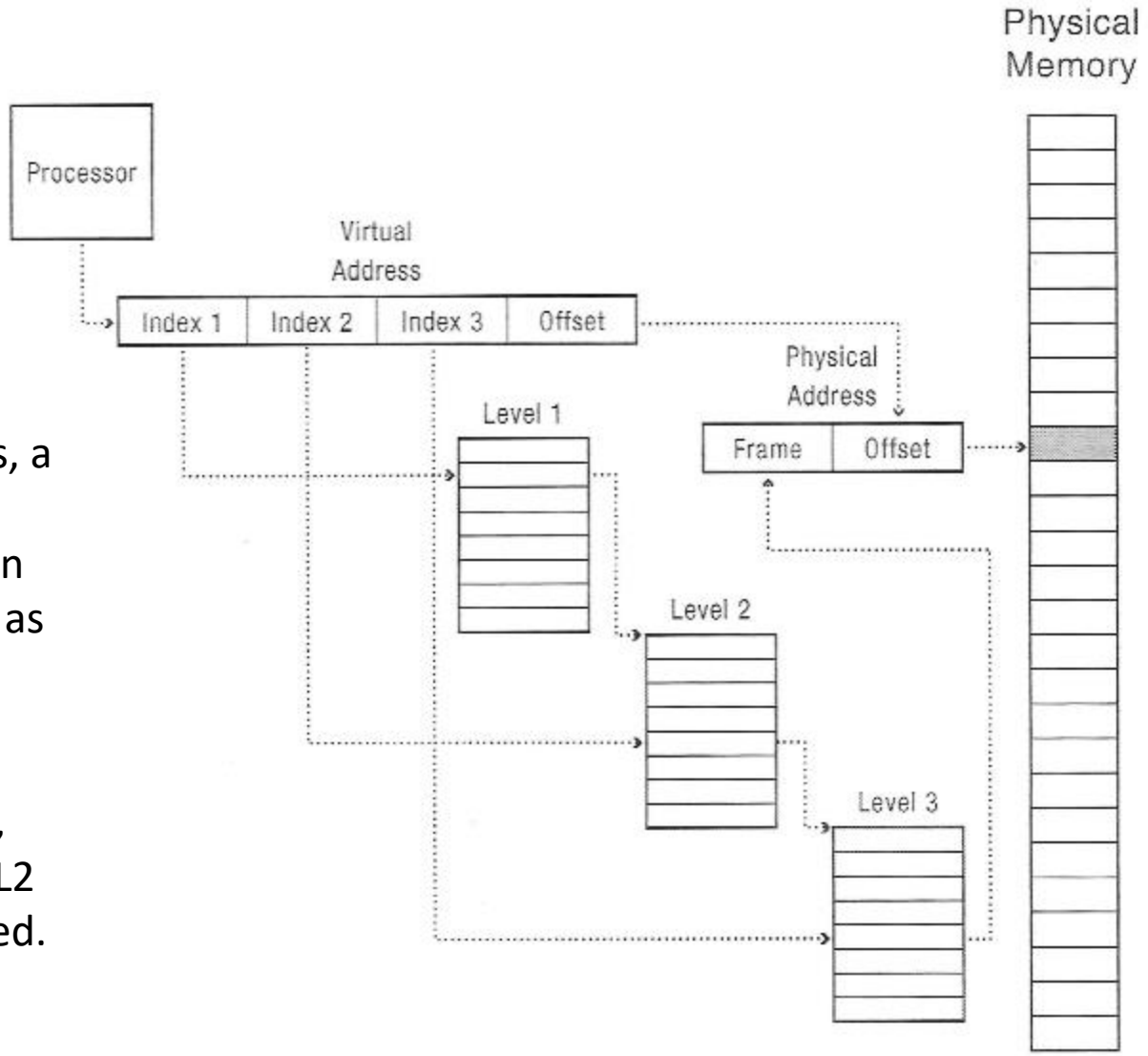
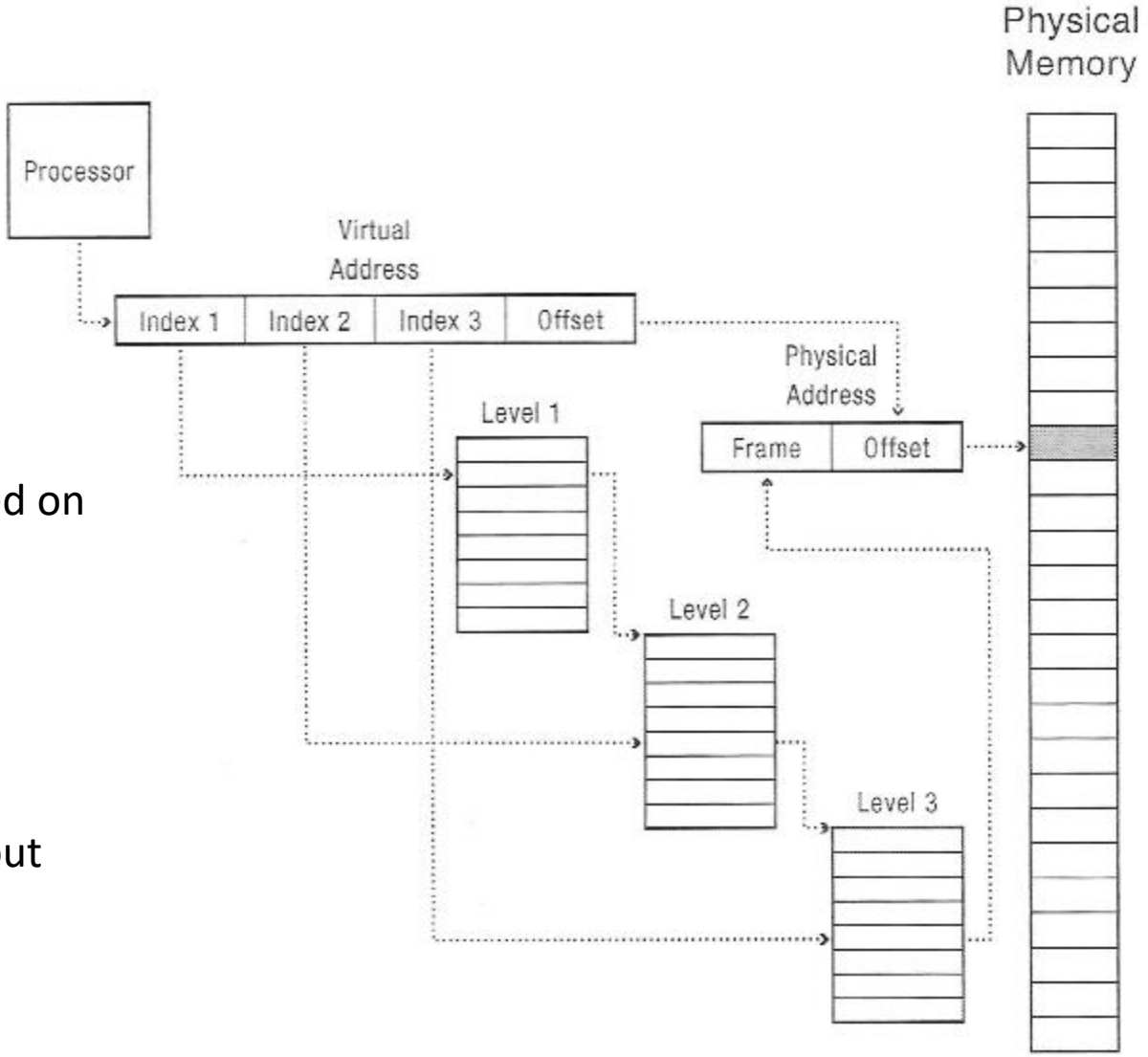


Image source: Anderson & Dahlin, *Operating Systems: Principles and Practice*, p. 398.

Multi-level Paging



Questions:

What must be changed on a context switch?

How would you share memory between processes?

What's not to like about this strategy?

Image source: Anderson & Dahlin, *Operating Systems: Principles and Practice*, p. 398.

Multi-level Paging

What must be changed on a context switch?

Pointer to a level 1 page table.

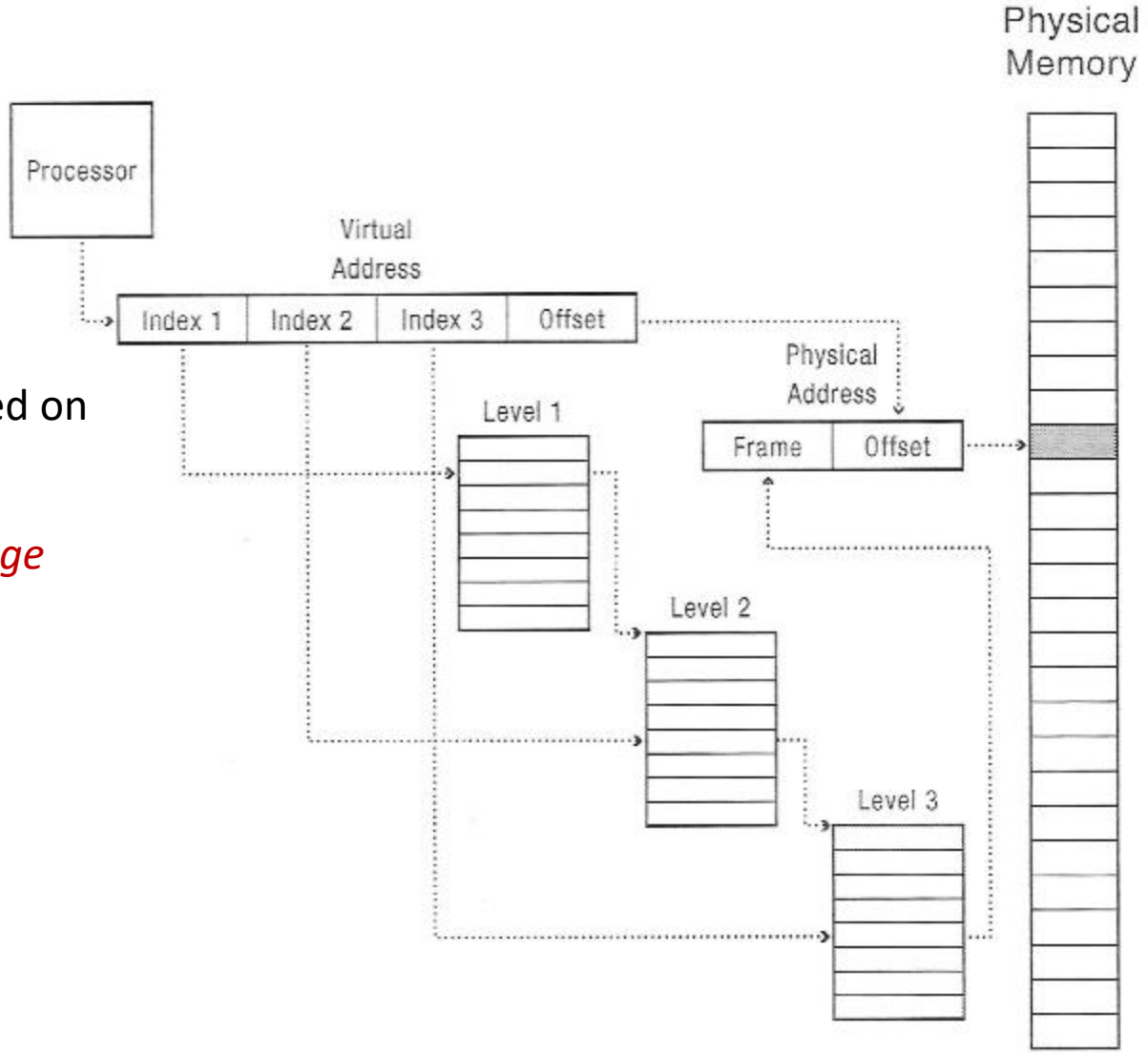


Image source: Anderson & Dahlin, *Operating Systems: Principles and Practice*, p. 398.

Multi-level Paging

How would you share memory between processes?

Share either individual pages or large blocks of pages by sharing a level 1, 2 or 3 entry.

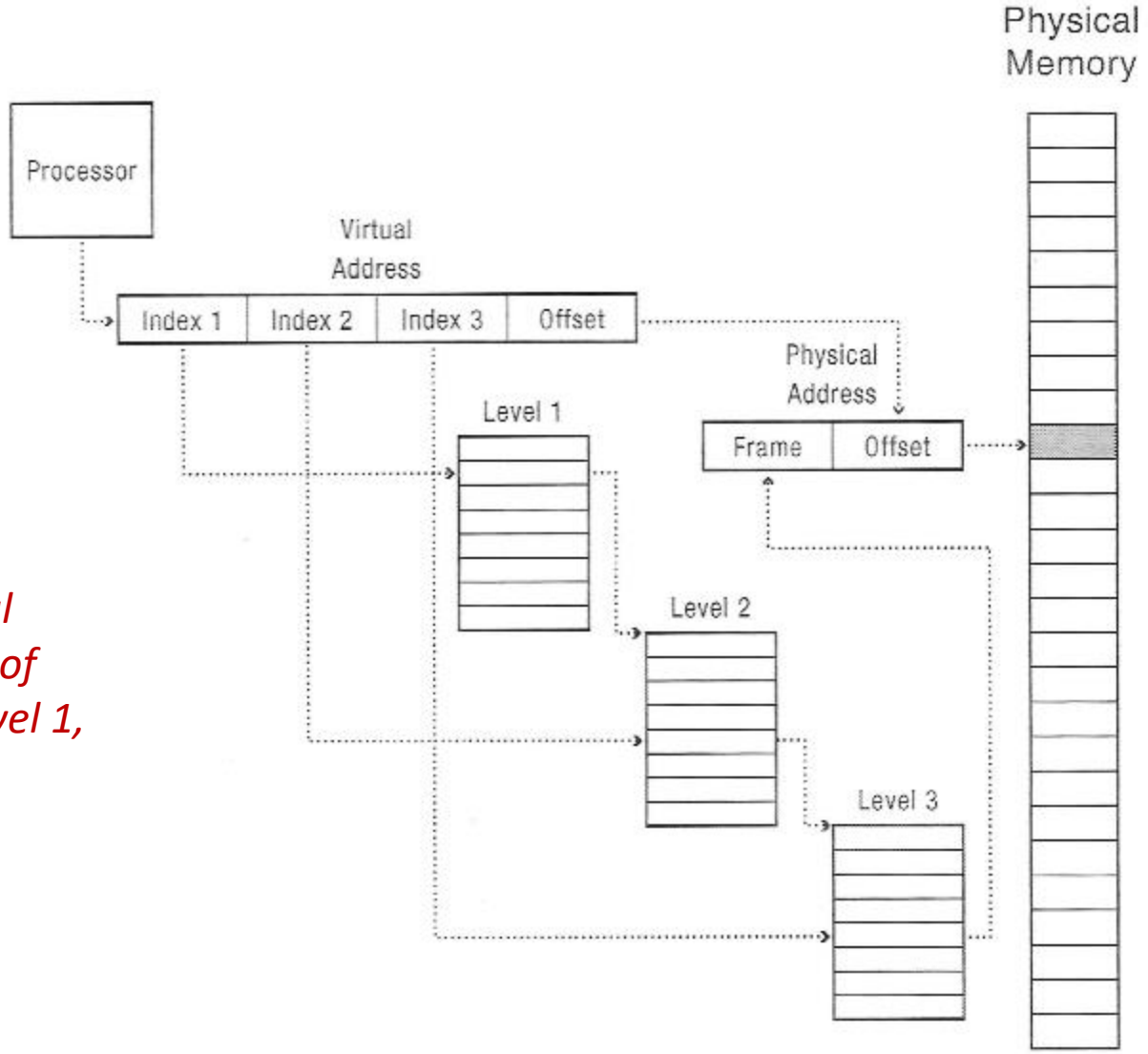


Image source: Anderson & Dahlin, *Operating Systems: Principles and Practice*, p. 398.

Multi-level Paging

What's not to like about this strategy?

Every memory access by a user application requires multiple table lookups.

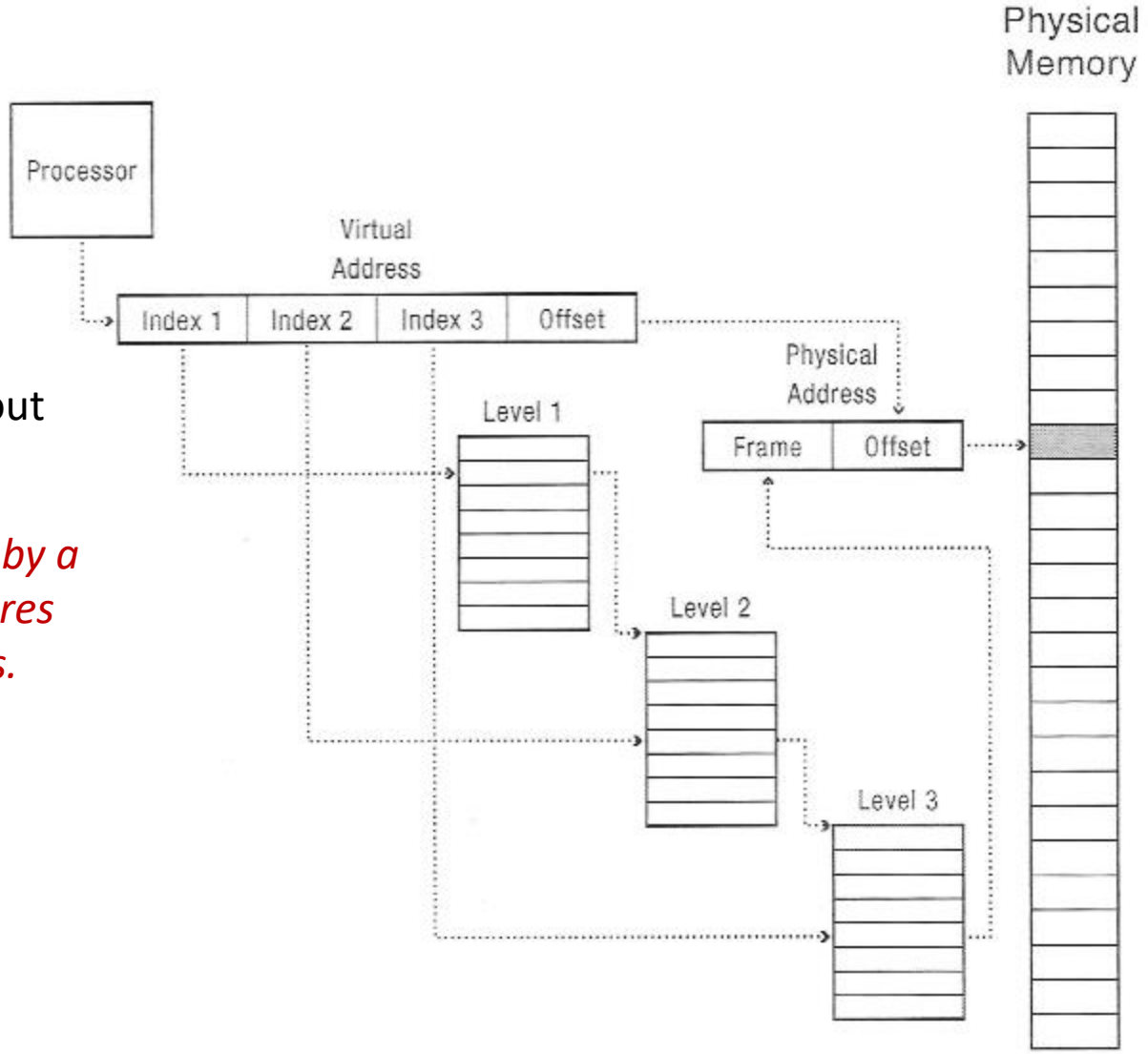


Image source: Anderson & Dahlin, *Operating Systems: Principles and Practice*, p. 398.

Multilevel paging

Pros

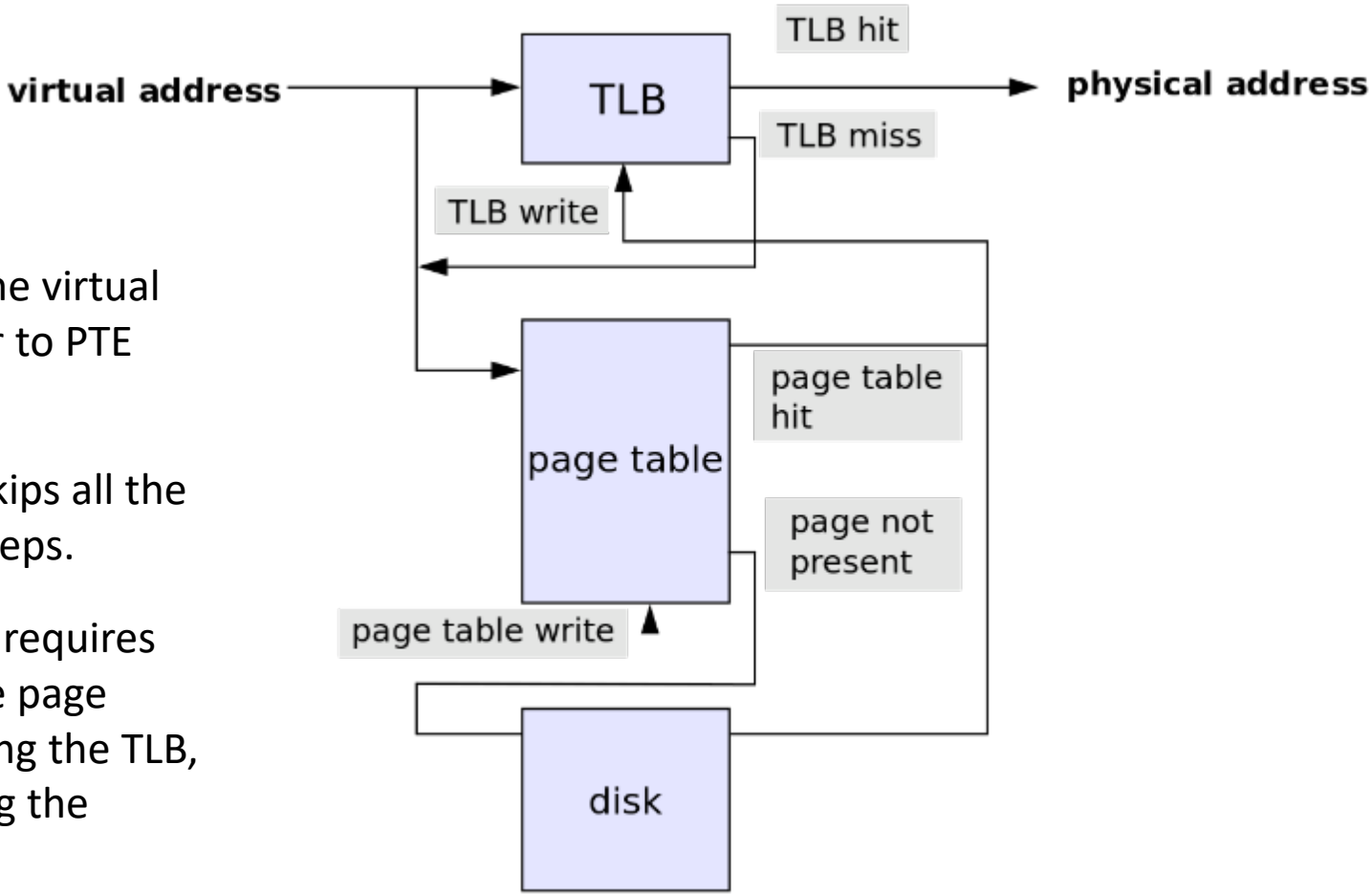
1. Simple memory allocation.
2. Flexible sharing.
3. Easy to grow address space.
4. Space-efficient representation of the page table.

Cons

1. Two or more extra lookups per memory reference.

What could be done to solve this?
We can cache the translations in hardware.

Translation lookaside buffer



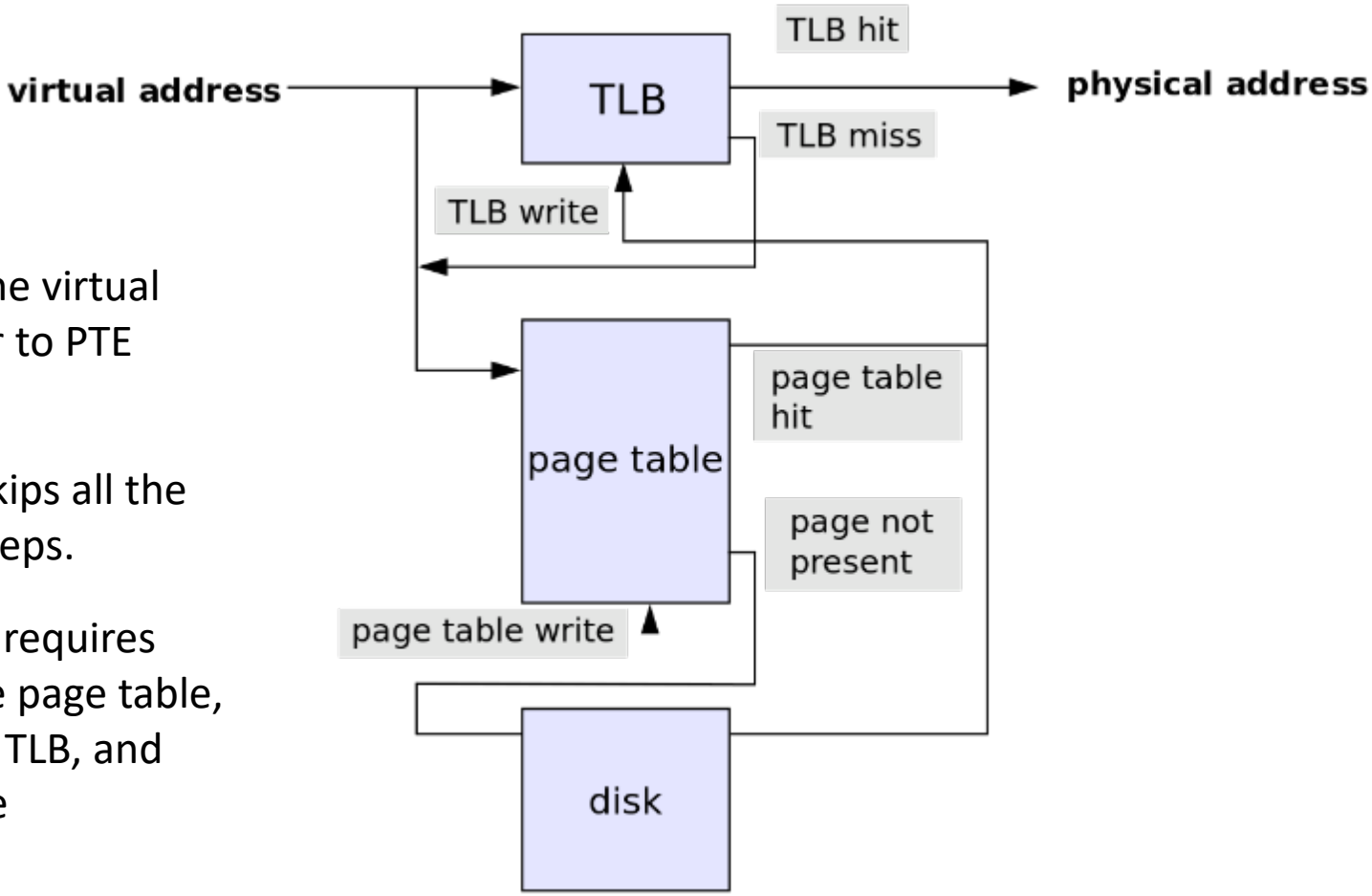
TLB caches the virtual page number to PTE mapping.

A *cache hit* skips all the translation steps.

A *cache miss* requires searching the page table, updating the TLB, and restarting the instruction.

Image source: Wikipedia, *"Page table"*.

Translation lookaside buffer



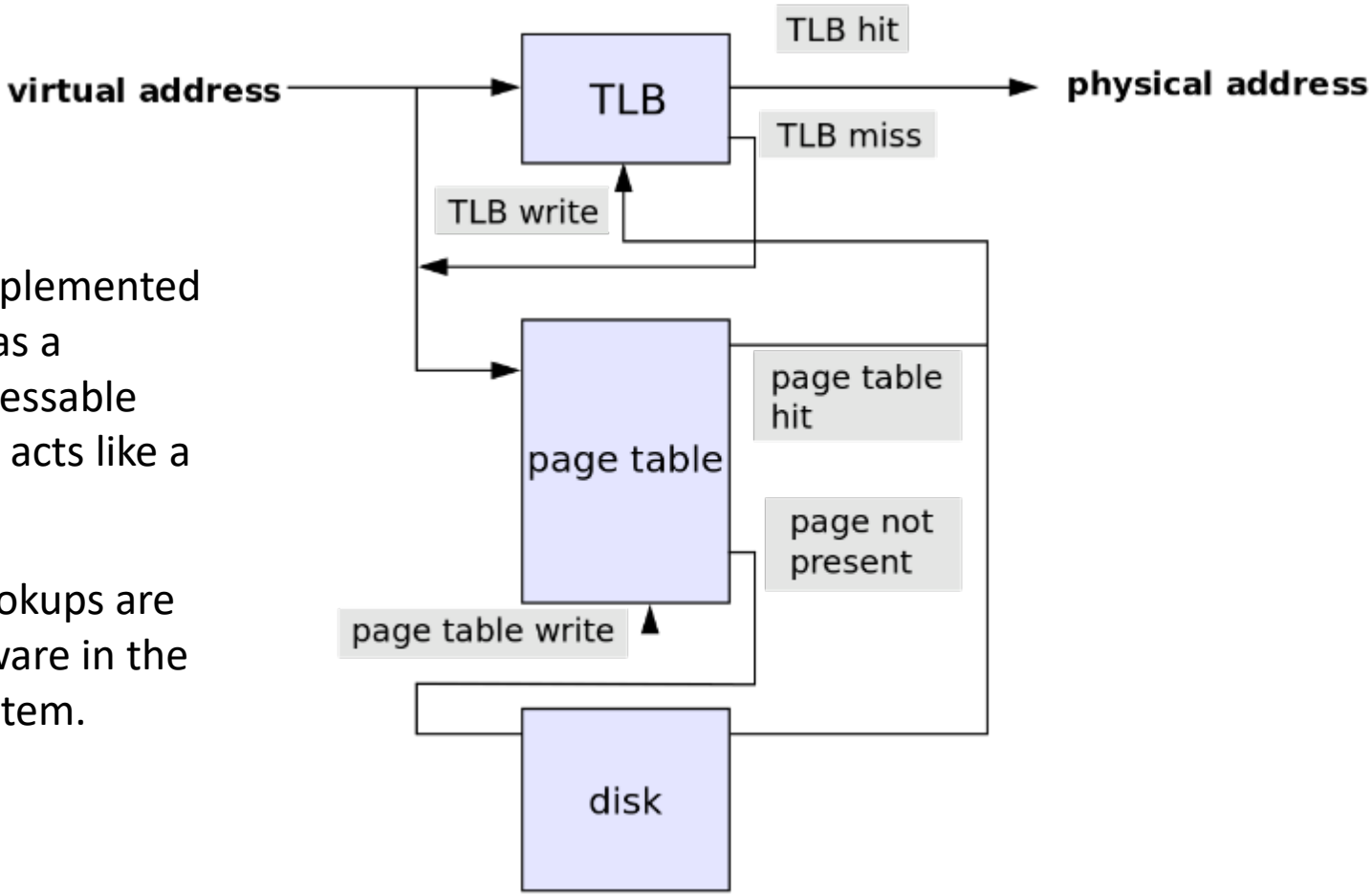
TLB caches the virtual page number to PTE mapping.

A *cache hit* skips all the translation steps.

A *cache miss* requires searching the page table, updating the TLB, and restarting the instruction.

Image source: Wikipedia, *"Page table"*.

Translation lookaside buffer



The TLB is implemented in hardware as a content-addressable memory that acts like a map in C++.

Page table lookups are done in software in the operating system.

Image source: Wikipedia, *"Page table"*.

Unix process creation

Why first copy the process only to overwrite it?

Even if it makes for a simpler application programming interface (API), isn't it still expensive and wasteful?

No, because the operating system uses a virtual memory technique called copy-on-write.

Avoiding work on fork

Copying entire address space is expensive

Instead, Linux/Unix uses copy-on-write.

Maintains a reference count for each physical page.

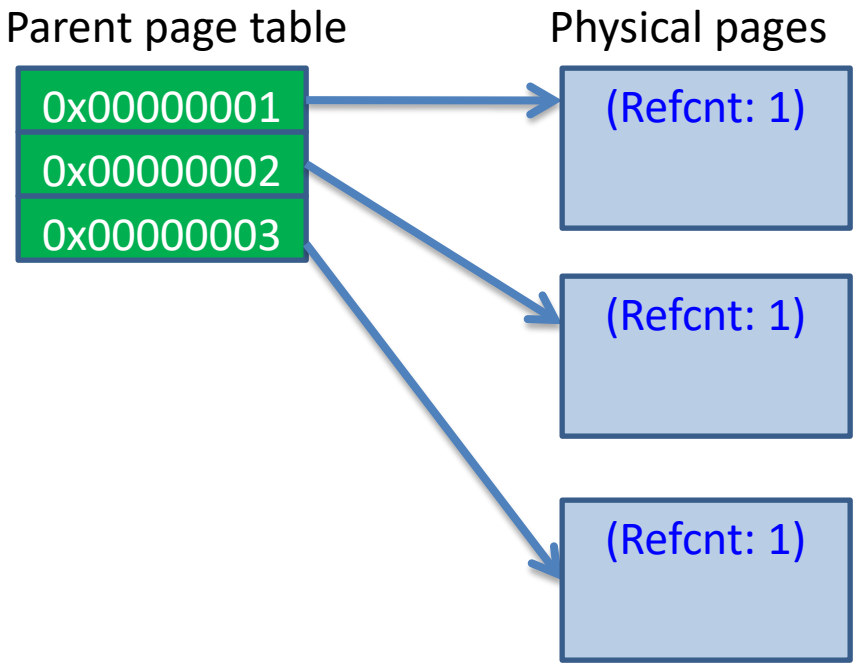
On `fork( )`, copy only the page table of parent.

- Increment reference count by one.

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

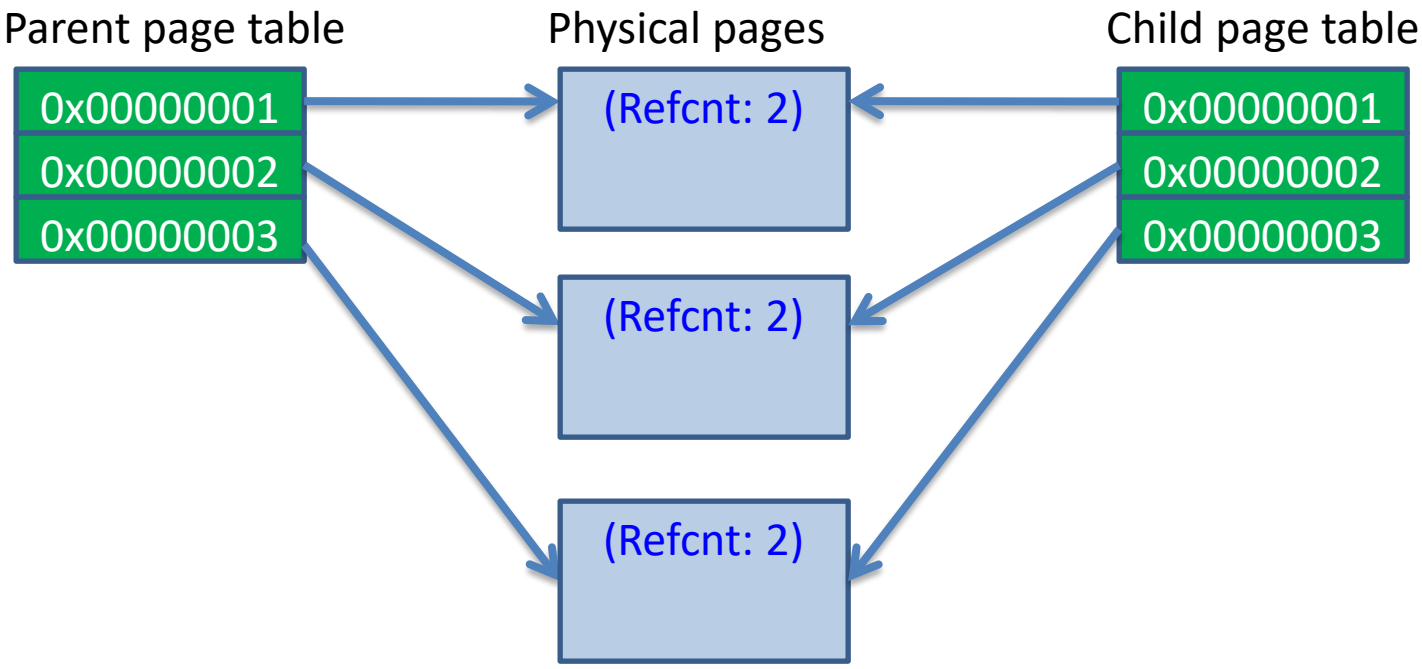
- Make a copy of the page with refcnt of one.
- Modify PTE of modifier to point to new page.
- Decrement reference count of old page.

Copy-on-write: Example



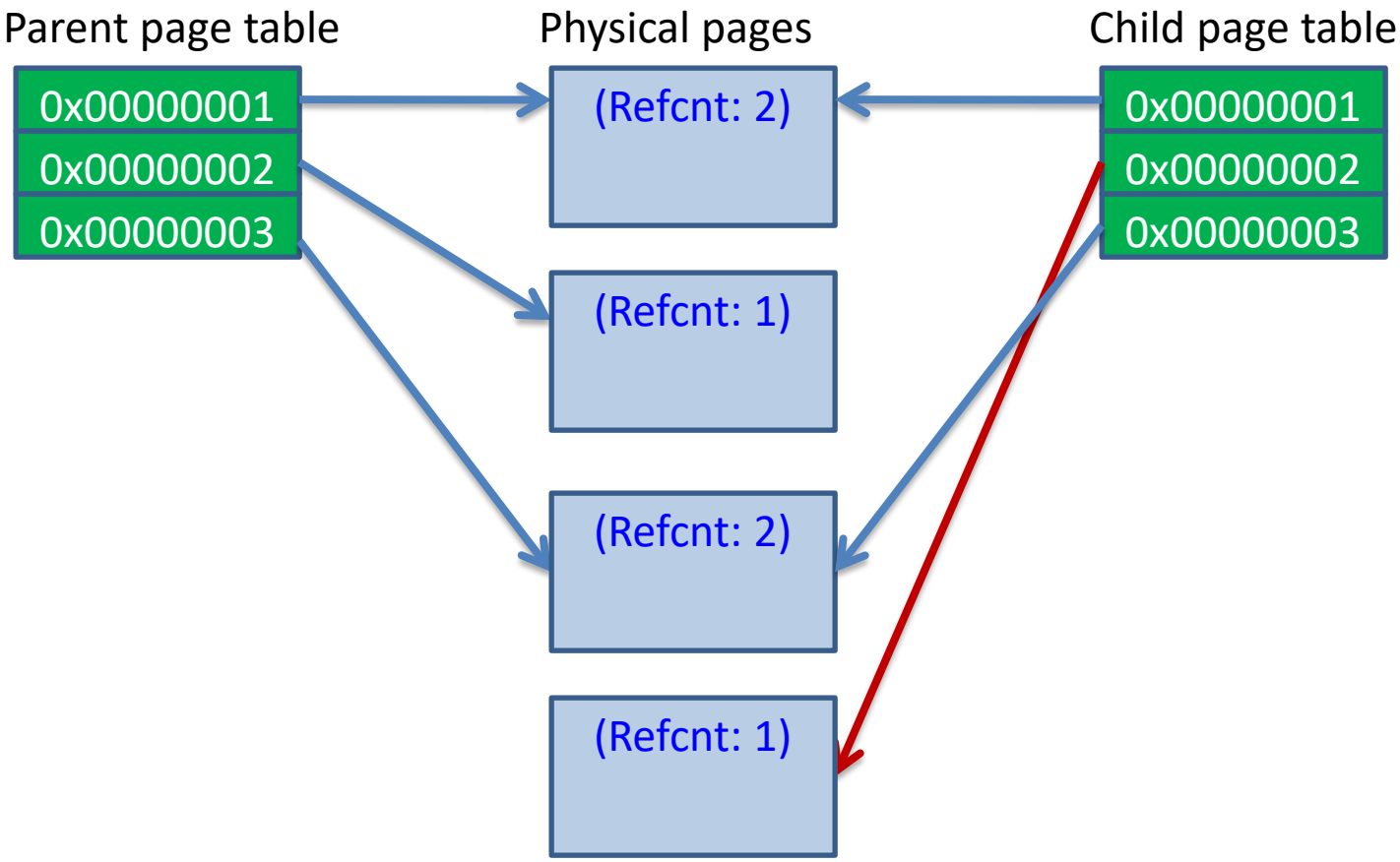
Parent about to `fork( )`.

Copy-on-write: Example



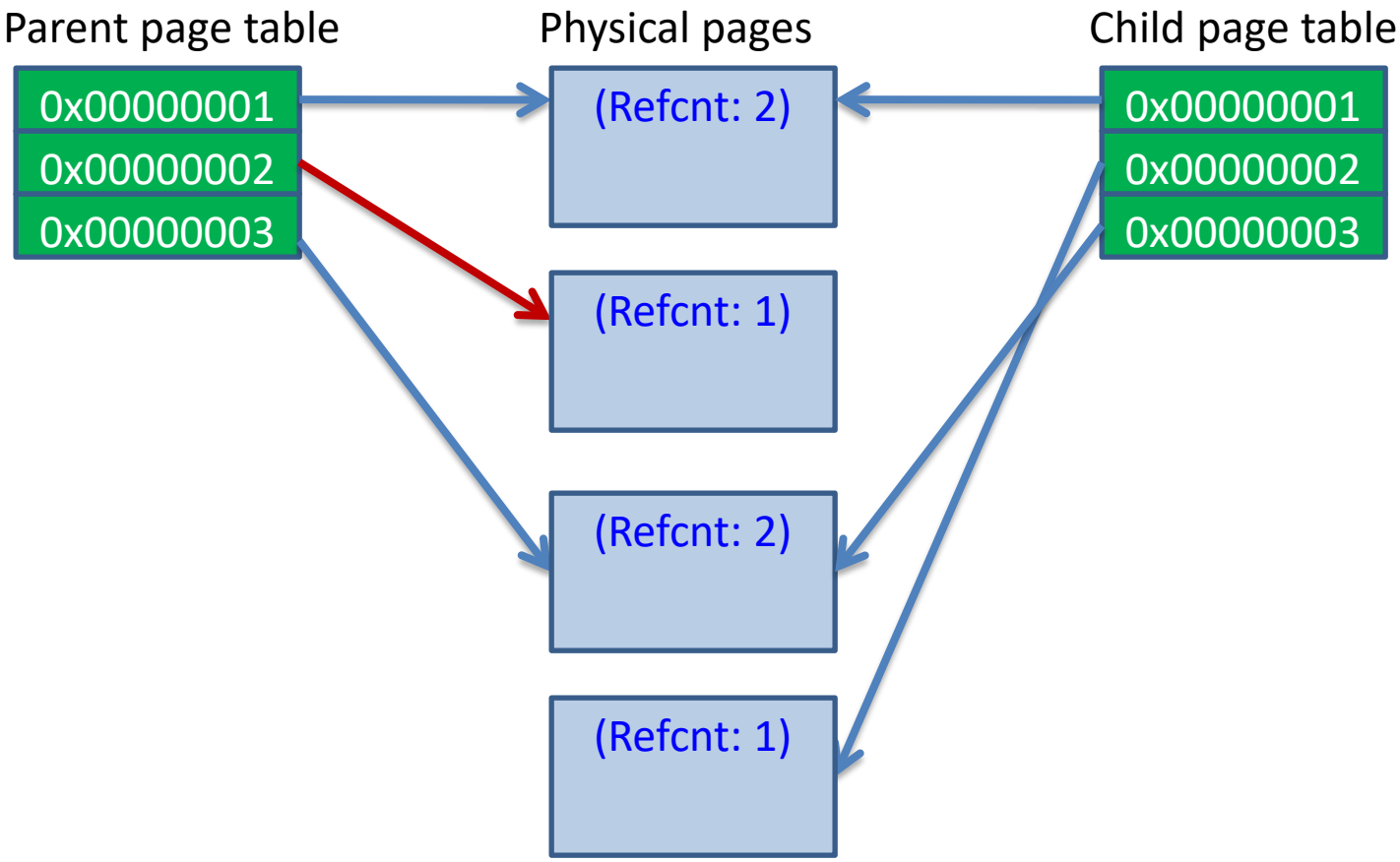
Copy-on-write of parent address space.

Copy-on-write: Example



Child modifying page 2 causes a copy to be made.

Copy-on-write: Example



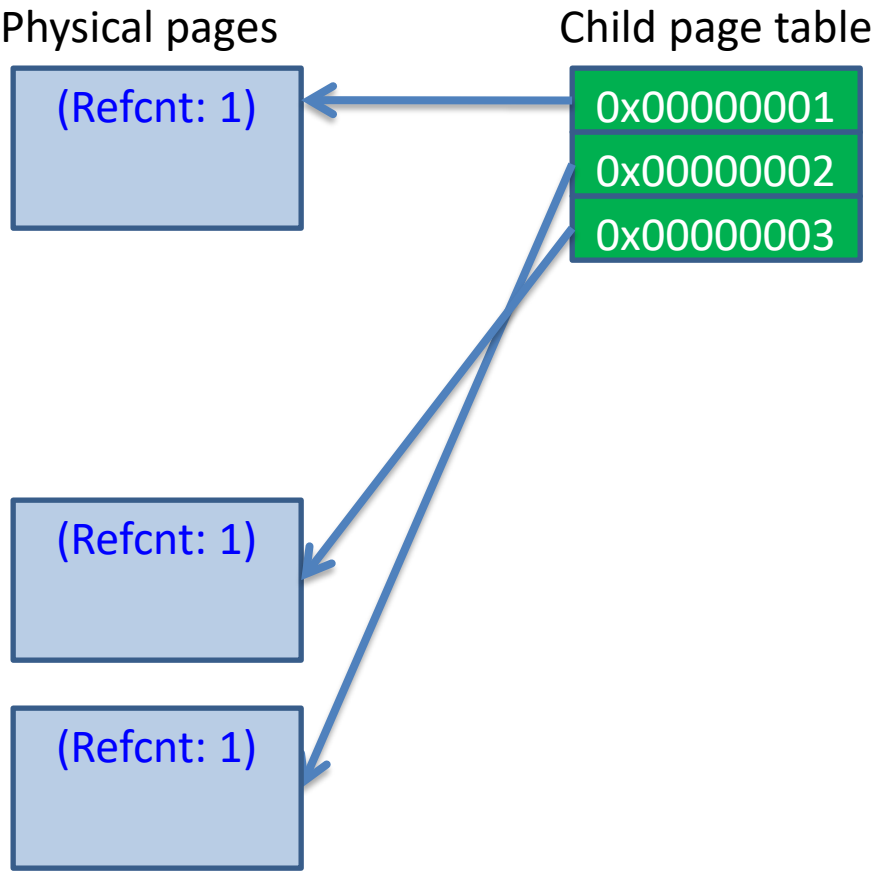
Parent modifying page 2 does not require copying.

Copy-on-write: Example

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.
- Similar to loading memory paged to disk.

Agenda

1. Course details.
2. Processes.
3. Virtual memory.
4. Threads.
5. Locks.
6. Producer-consumer relationships.
7. Multi-reader/single writer locks.

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:

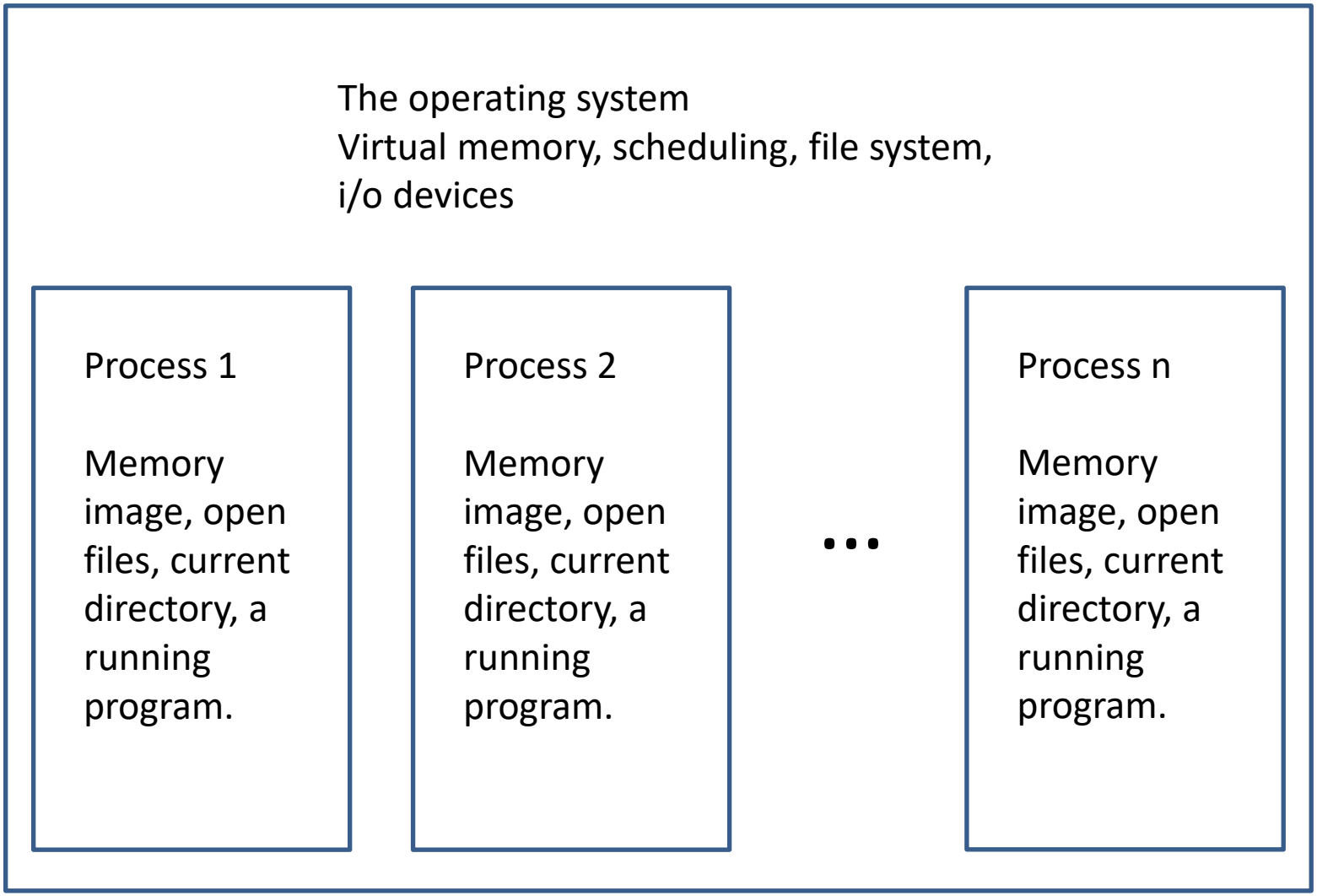
1. Very low cost to spawn.
2. Only a scheduler entry is created.
3. Everything else is shared.
4. No protection between threads.

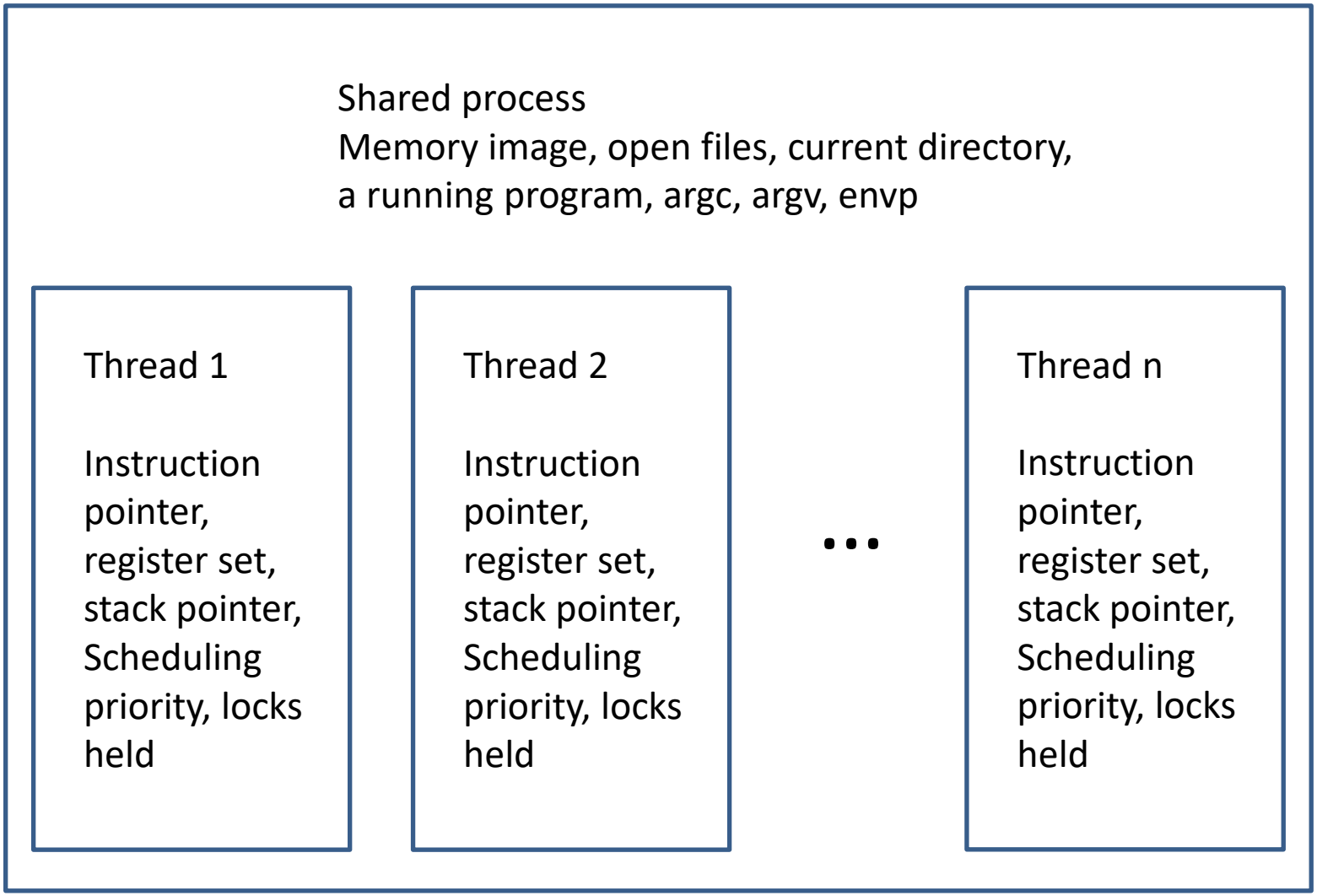
What is a thread?

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 or locks 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

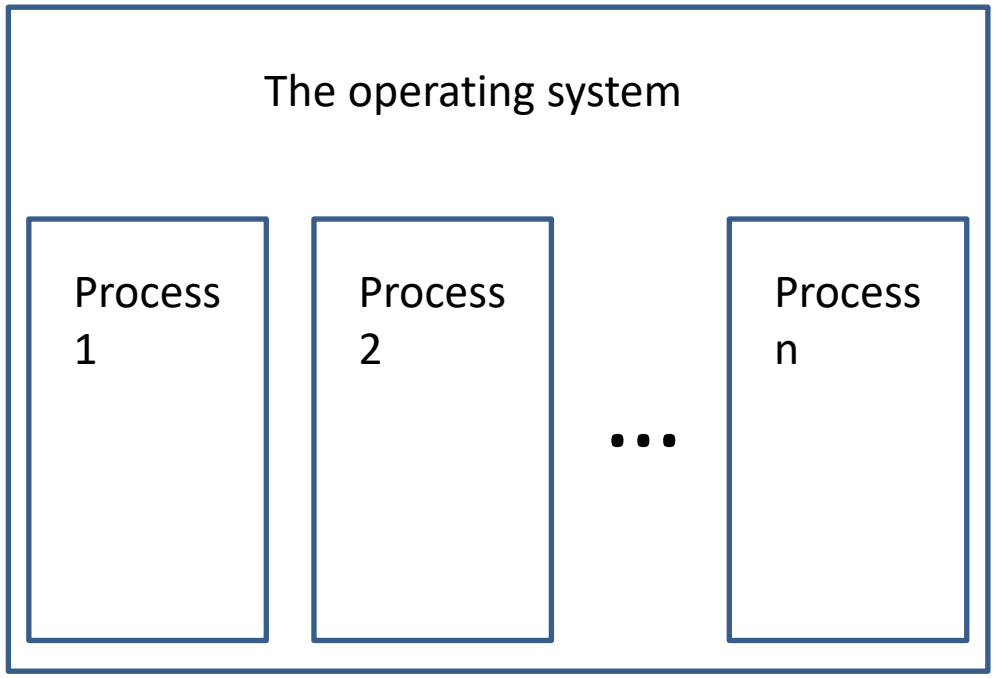
Processes provide concurrency between applications.

High startup costs.

One-way inheritance.

Lots of “firewalling.”

Errant apps can’t scribble on others.



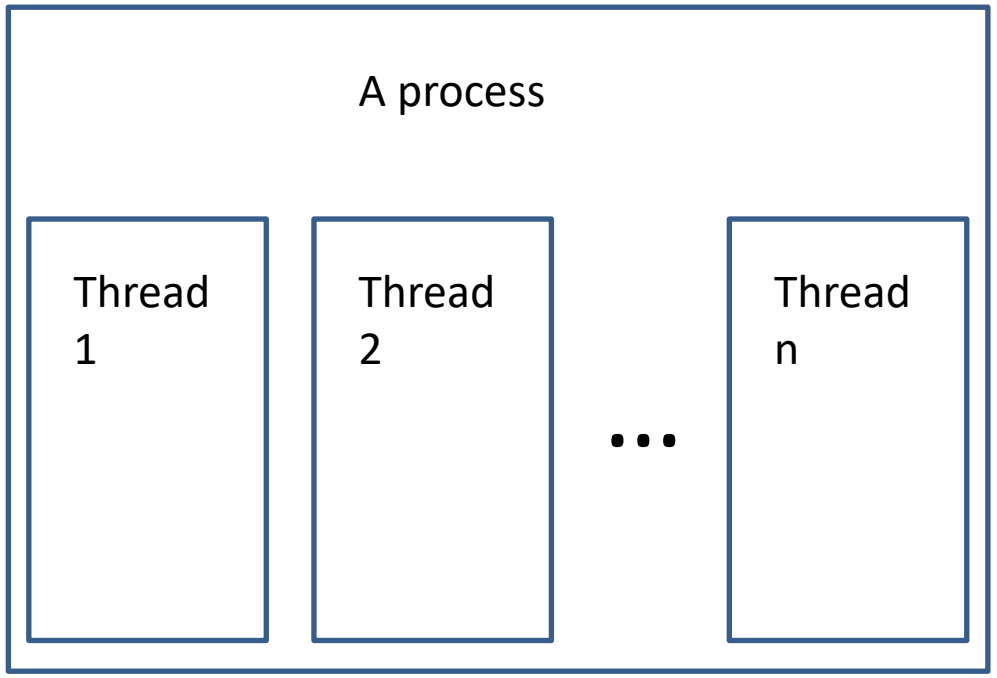
Threads provide concurrency *within* an application:

Very low cost to spawn.

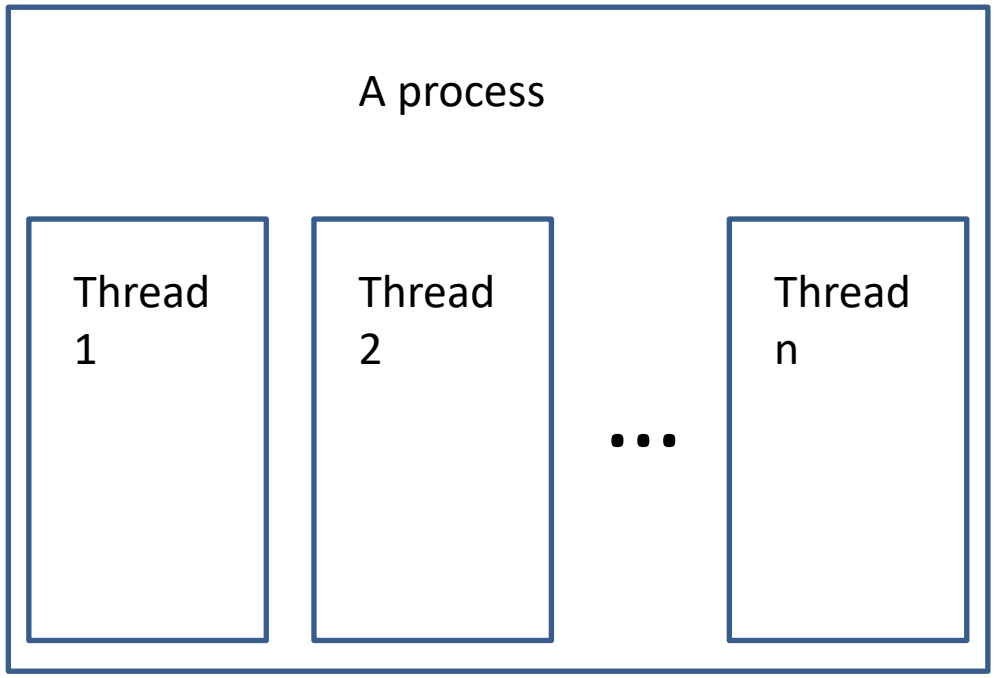
Only a scheduler entry is created.

Everything else is shared.

No protection between threads.

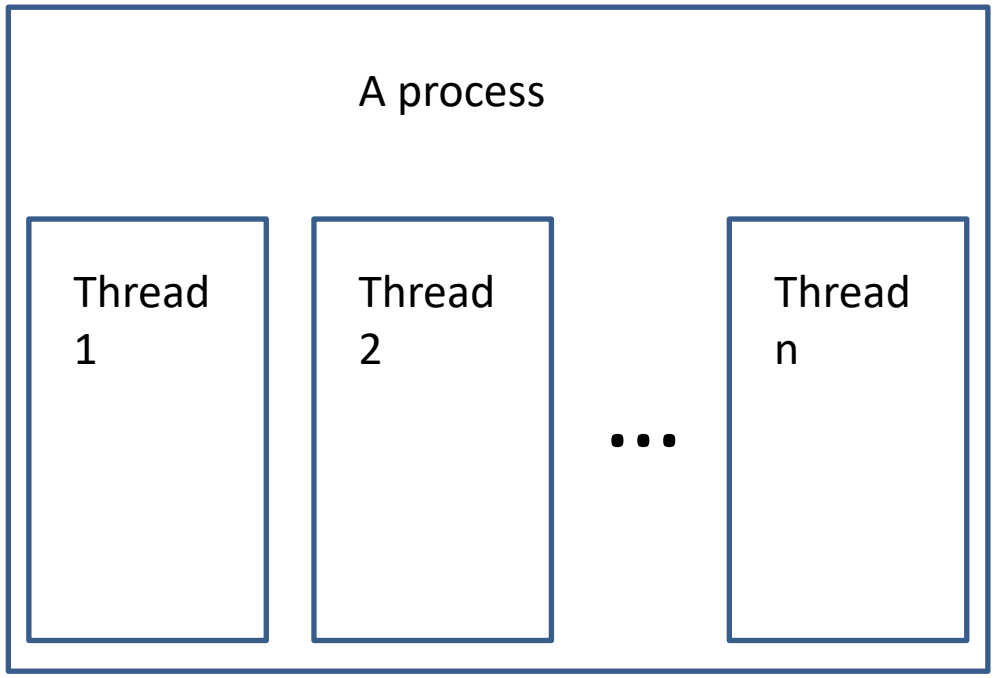


A thread is simple flow of control that can be separately scheduled.



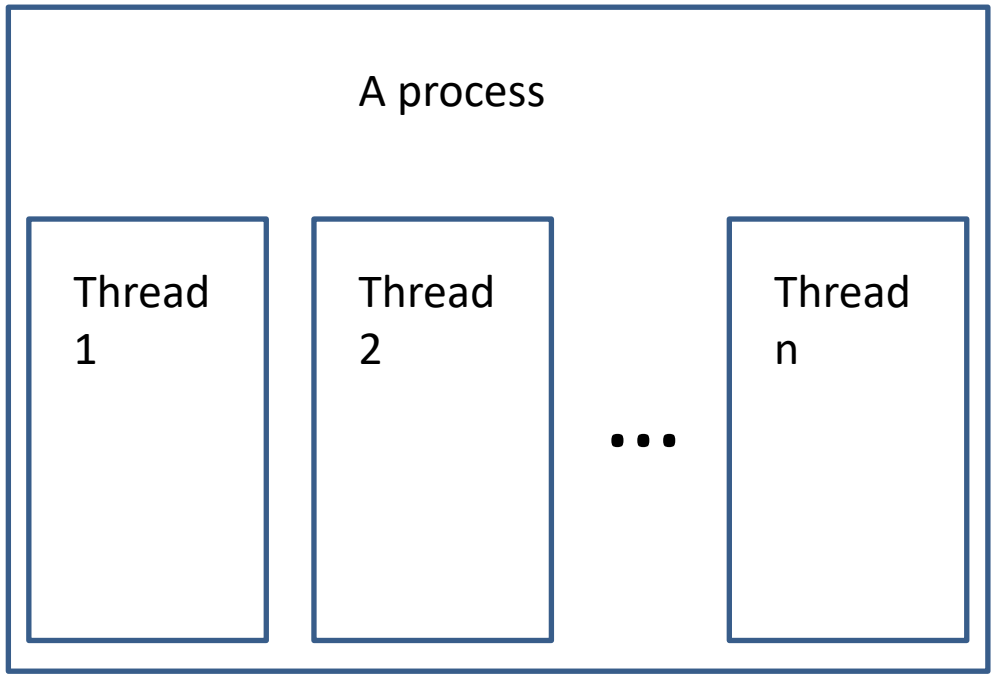
A thread's "state" consists of:

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:

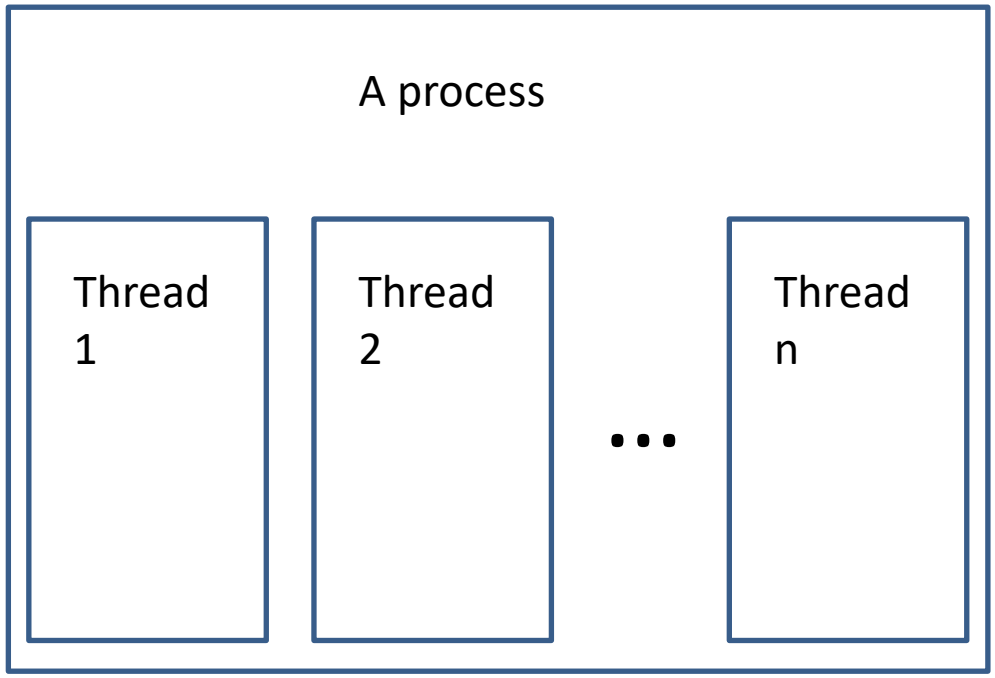
1. Memory (instructions and data),
2. Open handles to files, processes, pipes, etc.,
3. Current directory, and
4. Environment variables.



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.

```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);

int pthread_detach(pthread_t thread);
```

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().

```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);

int pthread_detach(pthread_t thread);
```

The pthread_join() function waits for the thread specified by thread to terminate and releases any resources still held. If the thread has already terminated, then pthread_join() returns immediately.

```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);

int pthread_detach(pthread_t thread);
```

pthread_detach() function marks the thread identified by thread as detached. When a detached thread terminates, its resources are automatically released back to the system without the need for another thread to join with the terminated thread.

```
$ 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
$
```

```
// 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. Processes.
3. Virtual memory.
4. Threads.
5. Locks.
6. Producer-consumer relationships.
7. Multi-reader/single writer locks.

Locks

1. All the threads share the same memory space. If one makes a change, it's instantly see by the others.
2. Must avoid scribbling on data being read by another thread.
3. If a data structure is being updated, other threads should not be reading it until the update is finished.
4. They need a protocol for locking the data, establishing ownership.

Locks

The problem is called *Mutual Exclusion*, meaning that if any is allowed, the rest are excluded.

On Linux, the mechanism we'll use is the pthread mutex.

Atomic operations

Before we can reason at all about cooperating threads, we must know that some operation is *atomic.*

1. It's indivisible. It happens in its entirety or not at all.
2. No events from other threads can occur in between when it starts and when it finishes.

Atomic operations

On most computers:

1. Memory load and store are atomic.
2. Many other instructions, e.g., double precision floating point, are not atomic.

Need an atomic operation to build bigger atomic operations.

Example

Assume i is a global shared variable.

Thread A

```
i = 0;
while ( i < 10 )
   i++;
print "A finished";
```

Thread B

```
i = 0;
while ( i > -10 )
    i--;
print "B finished";
```

Which thread will exit its while loop first?

Is the thread that exits the while first guaranteed to print first?

Is it guaranteed that anything will print?

Debugging Multi-Threaded Programs

Challenging due to non-deterministic interleaving.

Heisenbug: a bug that occurs non-deterministically.

All possible interleavings must be correct.

Synchronization

Objective:

Constrain interleavings between threads such that all possible interleavings produce a correct result.

Trivial solution:

Run each until it finishes before starting the next but that defeats the purpose of threads.

Challenge:

Constrain thread executions as little as possible.

Insight:

Some events are independent → order is irrelevant.

Other events are dependent → order matters.

```c
#include <pthread.h>

int pthread_mutex_init(pthread_mutex_t *restrict mutex,
      const pthread_mutexattr_t *restrict attr);

pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;

int pthread_mutex_destroy(pthread_mutex_t *mutex);
```

pthread_mutex_init( ) initializes the mutex to an unlocked state. If *attr* is NULL, the default attributes are used

pthread_mutex_destroy( ) destroys the mutex object referenced by *mutex*.

In cases where default mutex attributes are appropriate, the macro PTHREAD_MUTEX_INITIALIZER can be used to initialize mutexes that are statically allocated.

```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( ), and pthread_mutex_unlock( ) lock and unlock a mutex.

In the main thread:

```
pthread_mutex_t lock;

pthread_mutex_init( &lock, nullptr );

// Create lots of threads.

pthread_create( ... );



// Wait for them to finish.

pthread_join( ... );

pthread_mutex_destroy( &lock );
```

Within each child thread:

```
pthread_mutex_lock( &lock );

// Read or write the shared
data.

pthread_mutex_unlock( &lock );
```

Semaphore

A related problem is that the threads need to signal each other when events happen, e.g., when one has input for the other.

This done with a semaphore in Linux.

```c
#include <semaphore.h>

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

sem_init( ) initializes an unnamed semaphore.

```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, waking up anyone that’s waiting.

In the main thread:

```
sem_t available;

sem_init( &available, 0, 0 );

// Create lots of threads.

pthread_create( ... );



// Wait for them to finish.

pthread_join( ... );

sem_destroy( &available );
```

Within each child thread:

```
// To signal that data is
available.

sem_post( &available );


// To wait for data.

sem_wait( &available );
```

Agenda

1. Course details.
2. Processes.
3. Virtual memory.
4. Threads.
5. Locks.
6. Producer-consumer relationships.
7. Multi-reader/single writer locks.

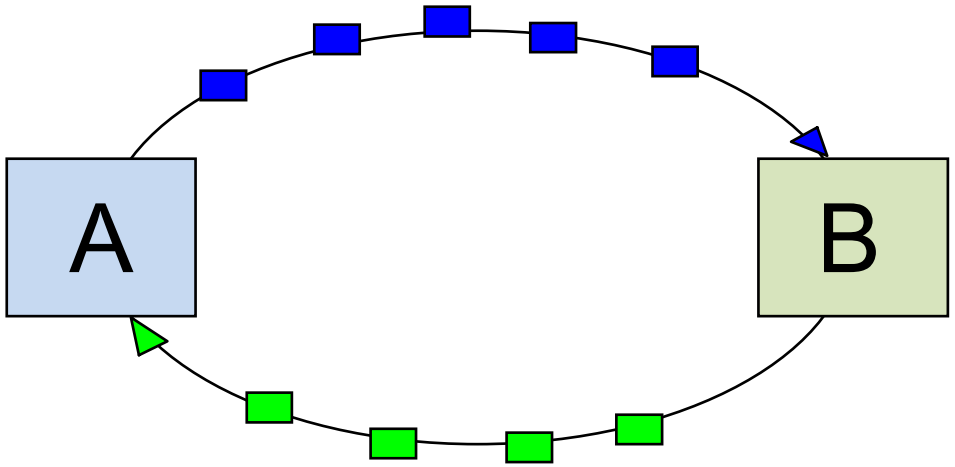
Producer-Consumer relationships

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.

```
$ 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
$
```

The basic idea:

1. Create two lists of empty and full buffers and start with 5 empty buffers.
2. Spawn a child thread to fill buffers, reading from stdin.
3. In the main thread, copy those buffers to stdout.
4. Anytime a thread wants to examine a list, it must lock it.
5. When the reader has data for the writer, it should signal it with a semaphore.

```
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();
   }
```

```
// 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 )
      {
      }
   };
```

```
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();
   }
```

Agenda

1. Course details.
2. Processes.
3. Virtual memory.
4. Threads.
5. Locks.
6. Producer-consumer relationships.
7. Multi-reader/single writer locks.

Basic problem

1. You have object in memory you would like to share between threads.
2. It doesn't change very often but it gets read a lot.
3. Lots of threads can share access that object if they're all just reading it.
4. But if any thread wants to write to it, it must lock out all the other threads while it does that.

Multiple reader / single writer problem

I'm going to ask you to try solving it.

It does show up on whiteboard interviews.

1. We'll assume some simple mutex and signaling facilities mapped to the OS.
2. I've give you a chance to invent your own solution. (Sorry, no autograder.)
3. I'll show you some known solutions but not today (and no, it won't be on the midterm.)

```
// Assume some basic Mutex and Signal mechanisms provided
// by the OS that we might wrapper as follows.

class Mutex
   {
   private:
      // ... a handle from the OS

   public:
      // Simple mutual exclusion.
      Take( );
      Release( );

      Mutex( );
      ~Mutex( );
   };
```

```
class Signal
   {
   private:
      // ... a handle from the OS
   public:
      // Simple signaling mechanism that can be set (true)
      // reset (false).

      // Wait for the signal to be set.
      // If it's not set, you sleep until it is.
      // Waiting for signal to be set does not reset it.
      // Other waiting threads will also wake up so long
      // as the signal remains set.

      void Wait( );

      // Set or reset it.
      void Set( );
      void Reset( );

      Signal( bool initialState = false );
      ~Signal( );
   };
```

```
// Here is the multiple reader / single writer interface
// to be built.

class SharedReader
   {
   public:
      virtual void ReadLock( ) = 0;
      virtual void ReleaseReadLock( ) = 0;
      virtual void WriteLock( ) = 0;
      virtual void ReleaseWriteLock( ) = 0;
   };
```