CS 422/522 Design & Implementation
of Operating Systems

Lecture 4: Memory Management & The Programming Interface

Zhong Shao
Dept. of Computer Science
Yale University

Acknowledgement: some slides are taken from previous versions of the CS422/522 lectures taught by Prof. Bryan Ford and Dr. David Wolinsky, and also from the official set of slides accompanying the OSPP textbook by Anderson and Dahlin.

This lecture

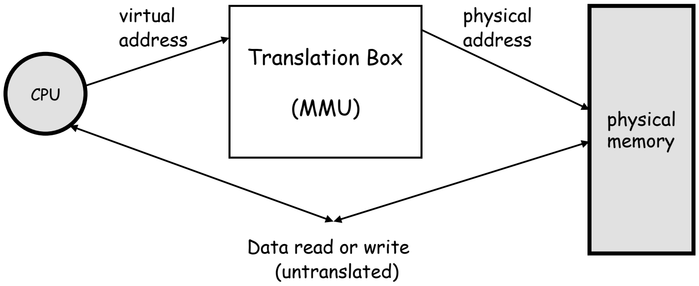
To support multiprogramming, we need "Protection"

- Kernel vs. user mode
- What is an address space?
- How to implement it?

Physical memory	**Abstraction: virtual memory**
No protection	Each program isolated from all others and from the OS
Limited size	Illusion of "infinite" memory
Sharing visible to programs	Transparent --- can't tell if memory is shared

The big picture

- To support multiprogramming with protection, we need:
  - dual mode operations
  - translation between virtual address space and physical memory
- How to implement the translation?

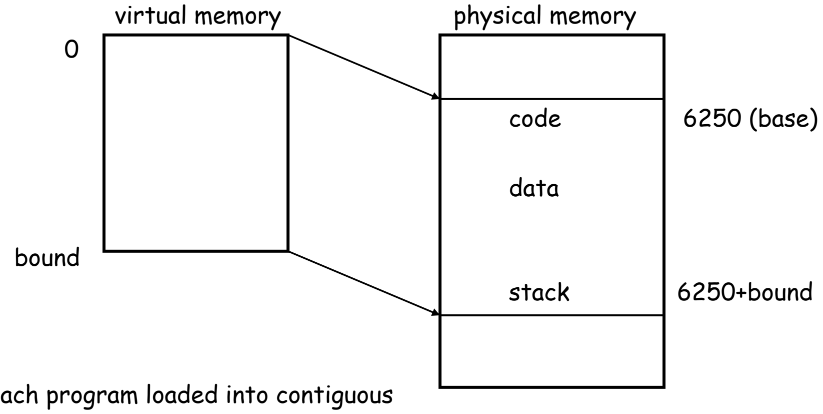


Address translation

- Goals
  - implicit translation on every memory reference
  - should be very fast
  - protected from user' s faults

- Options
  - Base and Bounds
  - Segmentation
  - Paging
  - Multilevel translation
  - Paged page tables

Base and Bounds



virtual memory

physical memory

0

bound

code

data

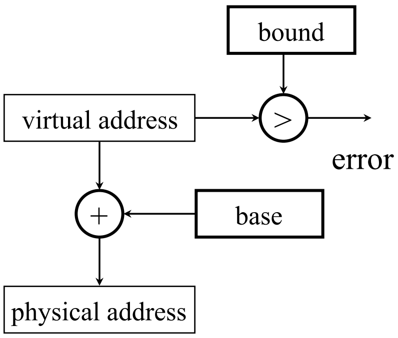
stack

6250 (base)

6250+bound

Each program loaded into contiguous
regions of physical memory.
Hardware cost: 2 registers, adder, comparator.

Base and Bounds (cont’ d)



bound

virtual address

>

error

+

base

physical address

- Built in Cray-1
- A program can only access physical memory in **[base, base+bound]**
- On a context switch: save/restore base, bound registers
- Pros: Simple
- Cons: fragmentation; hard to share (code but not data and stack); complex memory allocation

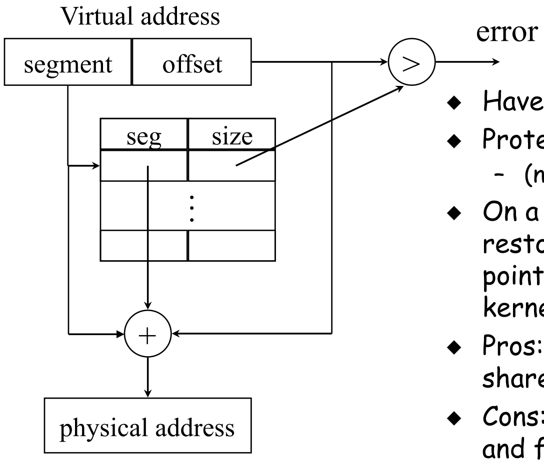
Segmentation

- Motivation
  - separate the virtual address space into several segments so that we can share some of them if necessary
- A segment is a region of logically contiguous memory
- Main idea: generalize base and bounds by allowing a table of base&bound pairs

(assume 2 bit segment ID, 12 bit segment offset)

virtual segment #	physical segment start	segment size
code (00)	0x4000	0x700
data (01)	0	0x500
- (10)	0	0
stack (11)	0x2000	0x1000

Segmentation (cont' d)



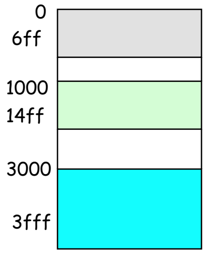
- Have a table of (seg, size)
- Protection: each entry has
  - (nil,read,write)
- On a context switch: save/ restore the table or a pointer to the table in kernel memory
- Pros: efficient, easy to share
- Cons: complex management and fragmentation within a segment

Segmentation example

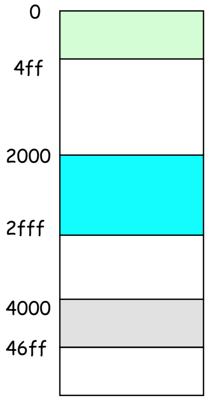
(assume 2 bit segment ID, 12 bit segment offset)

v-segment #	p-segment start	segment size
code (00)	0x4000	0x700
data (01)	0	0x500
- (10)	0	0
stack (11)	0x2000	0x1000

virtual memory



physical memory



Segmentation example (cont' d)

Virtual memory for `strlen(x)`

```
Main: 240          store 1108, r2
      244          store pc+8, r31
      248          jump 360
      24c
       …
strlen: 360        loadbyte (r2), r3
       …
       420         jump (r31)
      …

   x: 1108         a b c \0
      …
```

physical memory for `strlen(x)`

```
x:    108             a b c \0
       …

Main: 4240            store 1108, r2
      4244            store pc+8, r31
      4248            jump 360
      424c
       …
strlen: 4360          loadbyte (r2), r3
       …
       4420           jump (r31)
      …
```

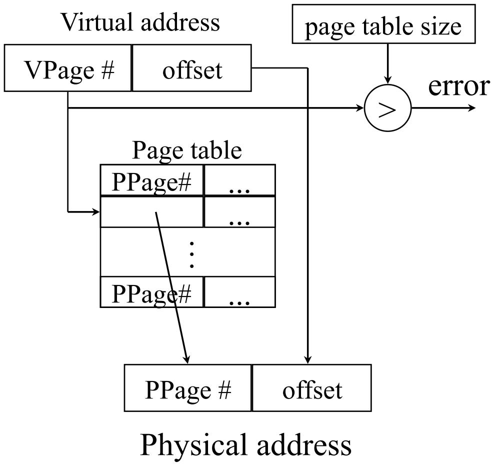
Paging

- Motivations
  - both branch bounds and segmentation still require fancy memory management (e.g., first fit, best fit, re-shuffling to coalesce free fragments if no single free space is big enough for a new segment)
  - can we find something simple and easy
- Solution
  - allocate physical memory in terms of fixed size chunks of memory, or **pages**.
  - Simpler because it allows use of a bitmap

    00111110000001100 --- each bit represents one page of physical memory

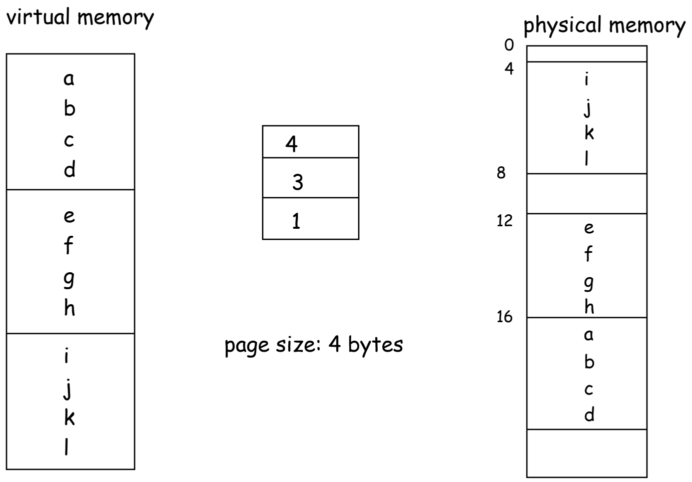
    1 means allocated, 0 means unallocated

Paging (cont’ d)



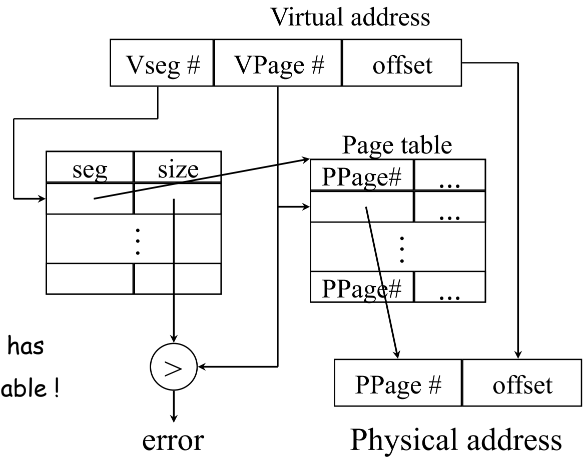
- Use a page table to translate
- Various bits in each entry
- Context switch: similar to the segmentation scheme
- What should be the page size?
- Pros: simple allocation, easy to share
- Cons: big page table and cannot deal with internal fragmentation easily

Paging example



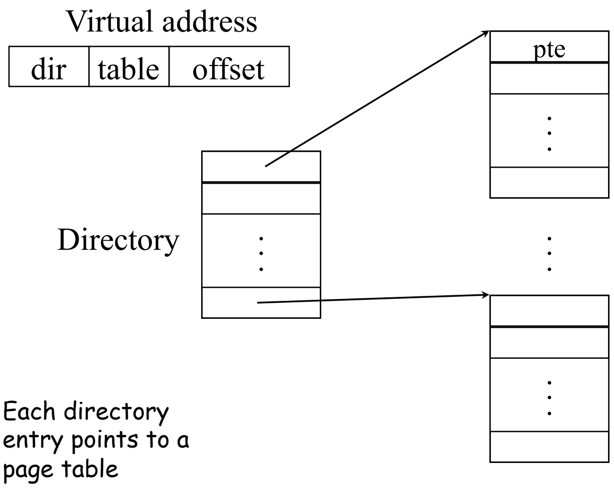
page size: 4 bytes

Segmentation with paging



Each segment has
its own page table !

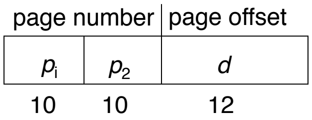
Two-level paging



Each directory entry points to a page table

Two-level paging example

- A logical address (on 32-bit machine with 4K page size) is divided into:
  - a page number consisting of 20 bits.
  - a page offset consisting of 12 bits.
- Since the page table is paged, the page number is further divided into:
  - a 10-bit page number.
  - a 10-bit page offset.
- Thus, a logical address is as follows:

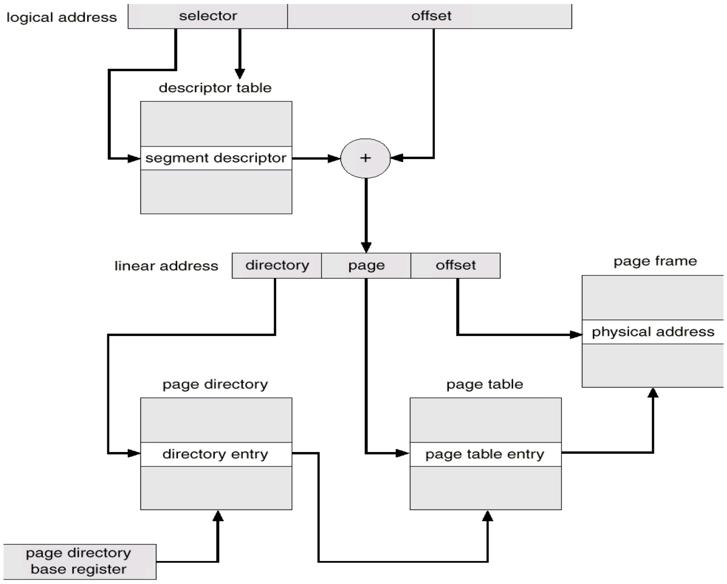


where p_i is an index into the outer page table, and p_2 is the displacement within the page of the outer page table.

Segmentation with paging – Intel 386

- As shown in the following diagram, the Intel 386 uses segmentation with paging for memory management with a two-level paging scheme.

Intel 30386 address translation



logical address

selector

offset

descriptor table

segment descriptor

+

linear address

directory

page

offset

page frame

physical address

page directory

directory entry

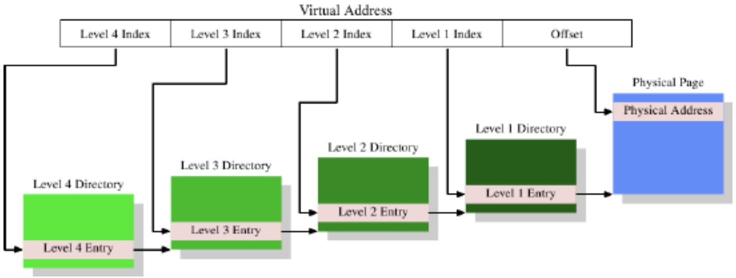
page table

page table entry

page directory base register

How many PTEs do we need ?

- Worst case for 32-bit address machine
  - # of processes × 2^{20} (if page size is 4096 bytes)
- What about 64-bit address machine?
  - # of processes × 2^{52}



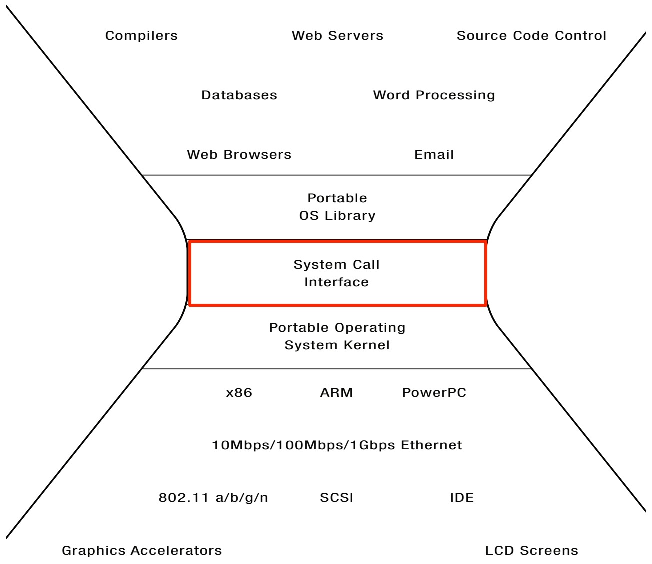
Summary: virtual memory mapping

- What?
  - separate the programmer's view of memory from the system's view
- How?
  - translate every memory operation using table (page table, segment table).
  - Speed: cache frequently used translations
- Result?
  - each user has a private address space
  - programs run independently of actual physical memory addresses used, and actual memory size
  - protection: check that they only access their own memory

Summary (cont'd)

- Goal: multiprogramming with protection + illusion of "infinite" memory
- Today's lecture so far:
  - HW-based approach for protection: dual mode operation + address space
  - Address translation: virtual address -> physical address
- Future topics
  - how to make address translation faster? use cache (TLB)
  - demand paged virtual memory
- The rest of today's lecture:
  - The programming interface

The programming interface



Compilers
Web Servers
Source Code Control
Databases
Word Processing
Web Browsers
Email

Portable
OS Library

System Call
Interface

Portable Operating
System Kernel

x86
ARM
PowerPC
10Mbps/100Mbps/1Gbps Ethernet
802.11 a/b/g/n
SCSI
IDE
Graphics Accelerators
LCD Screens

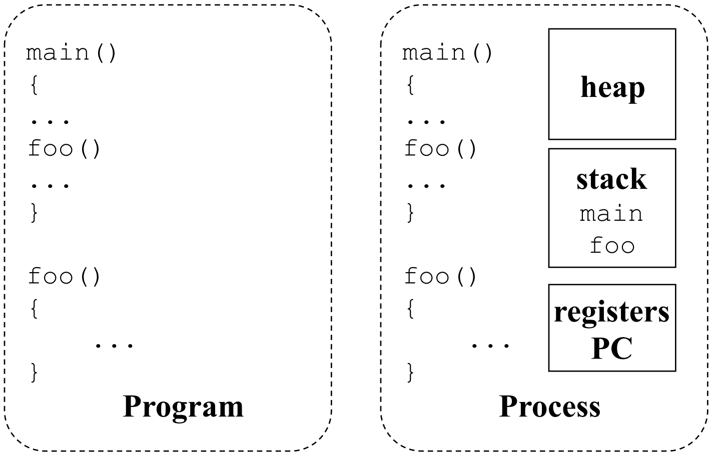
Abstraction: process & file system

- **Problem**
  - Multiple CPU cores, many I/O devices and lots of interrupts
  - Users feel they have machine to themselves
- **Answer**
  - Decompose hard problems into simple ones
  - Deal with one at a time
  - **Process is such a unit (reflecting something dynamic)**
  - **File system is another high-level abstraction (for "data")**
- **Future**
  - How processes differ from threads? What is a process really?
  - Generalizing "processes" to "containers" & "virtual machines"

Simplest process

- Sequential execution
  - No concurrency inside a process
  - Everything happens sequentially
  - Some coordination may be required
- Process state
  - Registers
  - Main memory
  - I/O devices
    - File system
    - Communication ports

Program vs. process



```
main()
{
...
foo()
...
}

foo()
{
    ...
}
```

Program

```
main()
{
...
foo()
...
}

foo()
{
    ...
}
```

heap

stack
`main`
`foo`

registers
PC

Process

Program vs. process (cont'd)

- Process > program
  - Program is just part of process state
  - Example: many users can run the same program (but different processes)
- Process < program
  - A program can invoke more than one process
  - Example: cc starts up cpp, cc1, cc2, as, ld (each are programs themselves)

Process control block (PCB)

- Process management info
  - State
    - Ready: ready to run
    - Running: currently running
    - Blocked: waiting for resources
  - Registers, EFLAGS, and other CPU state
  - Stack, code and data segment
  - Parents, etc
- Memory management info
  - Segments, page table, stats, etc
- I/O and file management
  - Communication ports, directories, file descriptors, etc.
- How OS takes care of processes
  - Resource allocation and process state transition

Primitives of processes

- Creation and termination
  - Exec, Fork, Wait, Kill
- Signals
  - Action, Return, Handler
- Operations
  - Block, Yield
- Synchronization
  - We will talk about this later

Make a process

- Creation
  - Load code and data into memory
  - Create an empty call stack
  - Initialize state to same as after a process switch
  - Make the process ready to run

- Clone
  - Stop current process and save state
  - Make copy of current code, data, stack and OS state
  - Make the process ready to run

UNIX process management

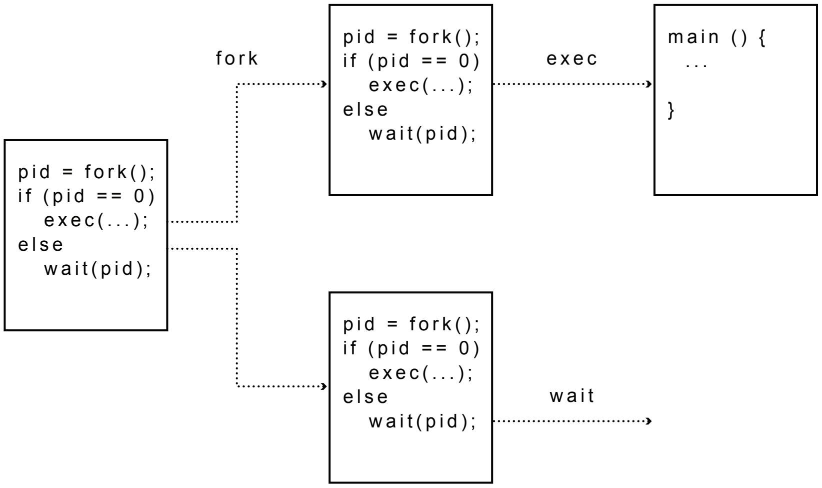
- UNIX fork – system call to create a copy of the current process, and start it running
  - No arguments!

- UNIX exec – system call to change the program being run by the current process

- UNIX wait – system call to wait for a process to finish

- UNIX signal – system call to send a notification to another process

UNIX process management



```
pid = fork();
if (pid == 0)
   exec(...);
else
   wait(pid);
```

```
pid = fork();
if (pid == 0)
   exec(...);
else
   wait(pid);
```

```
main () {
  ...

}
```

```
pid = fork();
if (pid == 0)
   exec(...);
else
   wait(pid);
```

Question: What does this code print?

```
int child_pid = fork();

if (child_pid == 0) {          // I'm the child process
   printf("I am process #%d\n", getpid());
   return 0;
} else {                       // I'm the parent process
   printf("I am parent of process #%d\n", child_pid);
   return 0;
}
```

Implementing UNIX fork & exec

- Steps to implement UNIX fork
  - Create and initialize the process control block (PCB) in the kernel
  - Create a new address space
  - Initialize the address space with a copy of the entire contents of the address space of the parent
  - Inherit the execution context of the parent (e.g., any open files)
  - Inform the scheduler that the new process is ready to run
- Steps to implement UNIX exec
  - Load the program into the current address space
  - Copy arguments into memory in the address space
  - Initialize the hardware context to start execution at ``start''

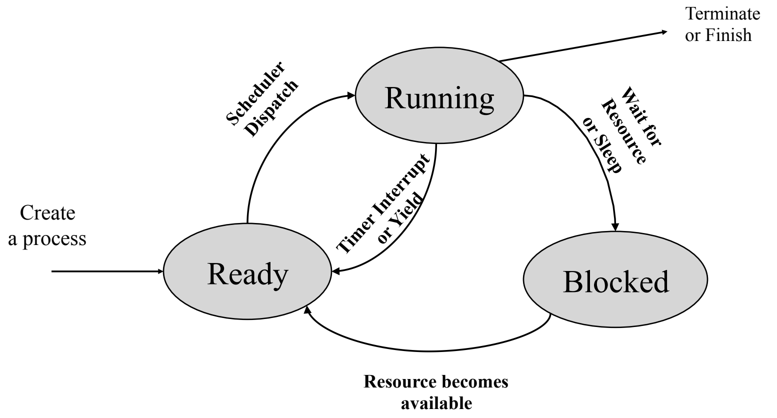
Process context switch

- Save a context (everything that a process may damage)
  - All registers (general purpose and floating point)
  - All co-processor state
  - Save all memory to disk?
  - What about cache and TLB stuff?

Very machine dependent !

- Start a context
  - Does the reverse
- Challenges
  - OS code must save state without changing any state
  - How to run without touching any registers?
    - CISC machines have a special instruction to save and restore all registers on stack
    - RISC: reserve registers for kernel or have way to carefully save one and then continue

Process state transition



Running: executing now
Ready: waiting for CPU
Blocked: waiting for I/O or lock

Which ready process to pick?

0 ready processes: run idle loop
1 ready process: easy!
> 1: what to do?

- FIFO?
  - put threads on back of list, pull them off from front
  - (nachos does this: schedule.cc)
- Pick random? (could result in starvation)
- Priority?
  - give some threads a better shot at the CPU

Scheduling policies

- Scheduling issues
  - fairness: don't starve process
  - prioritize: more important first
  - deadlines: must do by time 'x' (car brakes)
  - optimization: some schedules >> faster than others
- No universal policy:
  - many variables, can't maximize them all
  - conflicting goals
    - **more important jobs vs starving others**
    - **I want my job to run first, you want yours.**
- Given some policy, how to get control ?

How to get control?

- Traps: events generated by current process
  - system calls
  - errors (illegal instructions)
  - page faults
- Interrupts: events external to the process
  - I/O interrupt
  - timer interrupt (every 100 milliseconds or so)
- Process perspective:
  - explicit: process yields processor to another
  - implicit: causes an expensive blocking event, gets switched

UNIX I/O --- a key innovation ("files")

- Uniformity
  - All operations on all files, devices use the same set of system calls: open, close, read, write
- Open before use
  - Open returns a handle (file descriptor) for use in later calls on the file
- Byte-oriented
- Kernel-buffered reads/writes
- Explicit close
  - To garbage collect the open file descriptor
- Pipes (for interprocess communication → a kernel buffer with two file descriptors, one for reading, one for writing)

UNIX file system interface

- UNIX file open is a Swiss Army knife:
  - Open the file, return file descriptor
  - Options:
    - if file doesn't exist, return an error
    - If file doesn't exist, create file and open it
    - If file does exist, return an error
    - If file does exist, open file
    - If file exists but isn't empty, nix it then open
    - If file exists but isn't empty, return an error
    - …