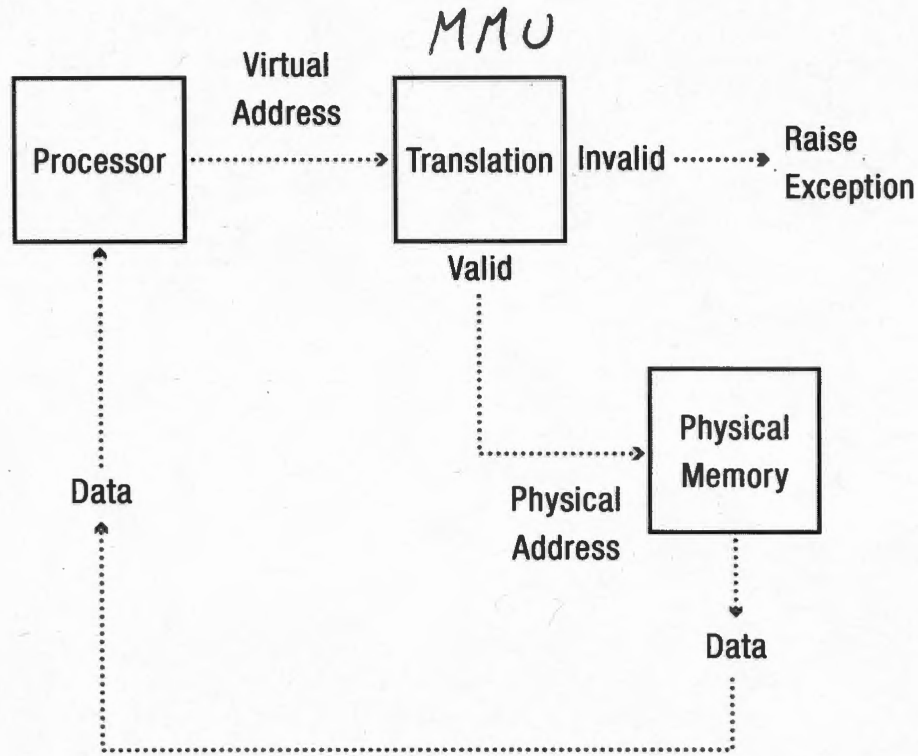
Please scan

Address Translation

Main Points

- Address Translation Concept
  - How do we convert a virtual address to a physical address?
- Flexible Address Translation
  - Segmentation
  - Paging
  - Multilevel translation
- Efficient Address Translation
  - Translation Lookaside Buffers
  - Virtually and physically addressed caches

Address Translation Concept



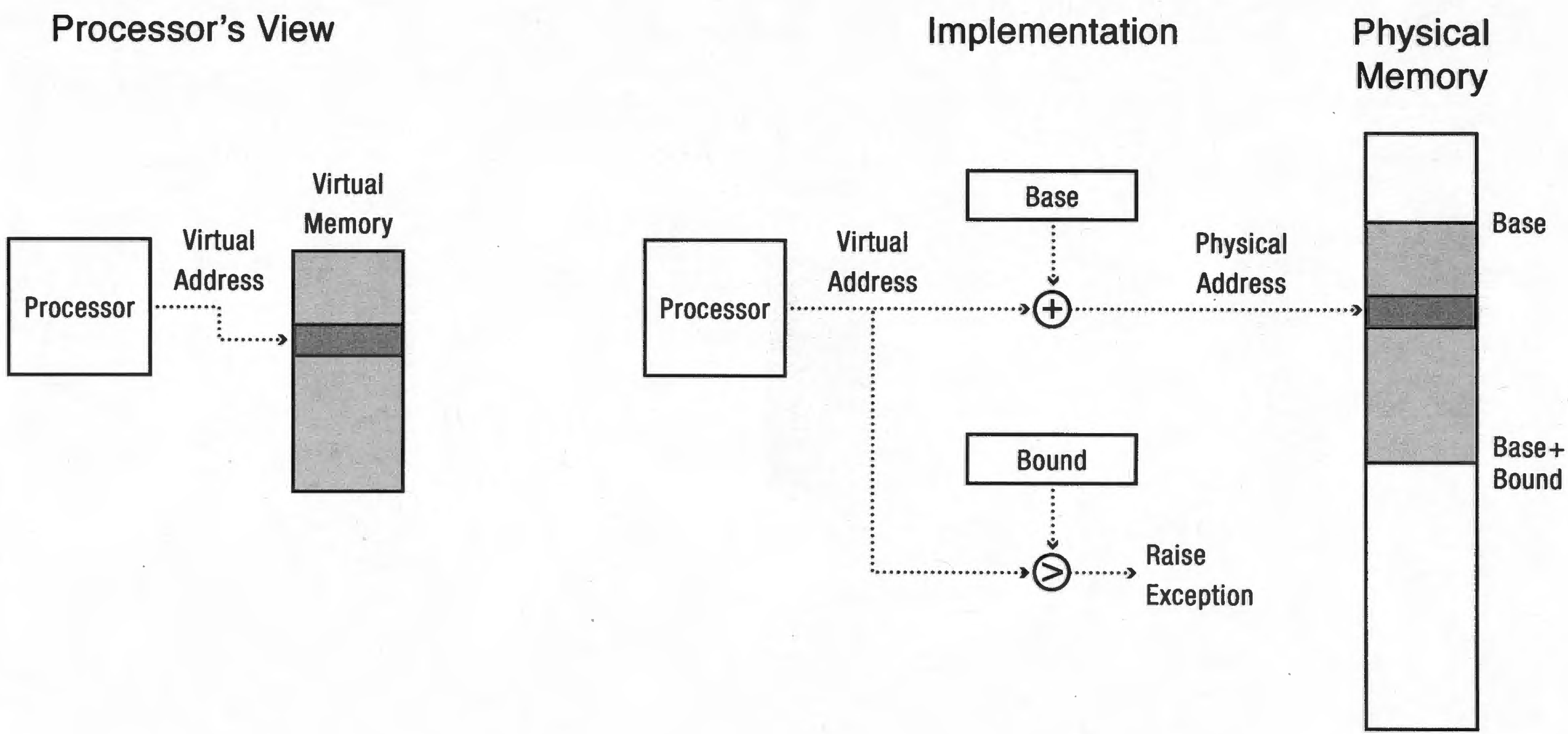
Address Translation Goals

- Memory protection
  - Isolate process to its only memory
  - Prevent virus from re-writing machine instructions
- Memory sharing
  - Shared libraries, interprocess communication
- Sparse addresses
  - Dynamically allocated regions: heaps, stacks, mmap
- Efficiency
  - Reduce fragmentation and copying
  - Runtime lookup cost and TLB hit rate
  - Translation table size
- Portability

Bonus Feature

- What if the kernel can regain control whenever a program reads or writes a particular virtual memory location?
- Examples:
  - Copy on write
  - Zero on reference
  - Fill on demand
  - Demand paging
  - Memory mapped files
  - …

Virtually Addressed Base and Bounds



Virtually Addressed Base and Bounds

- Pros?
  - Simple
  - Fast (2 registers, adder, comparator)
  - Safe
  - Can relocate in physical memory without changing process
- Cons?
  - Can't keep program from accidentally overwriting its own code
  - Can't share code/data with other processes
  - Can't grow stack/heap as needed

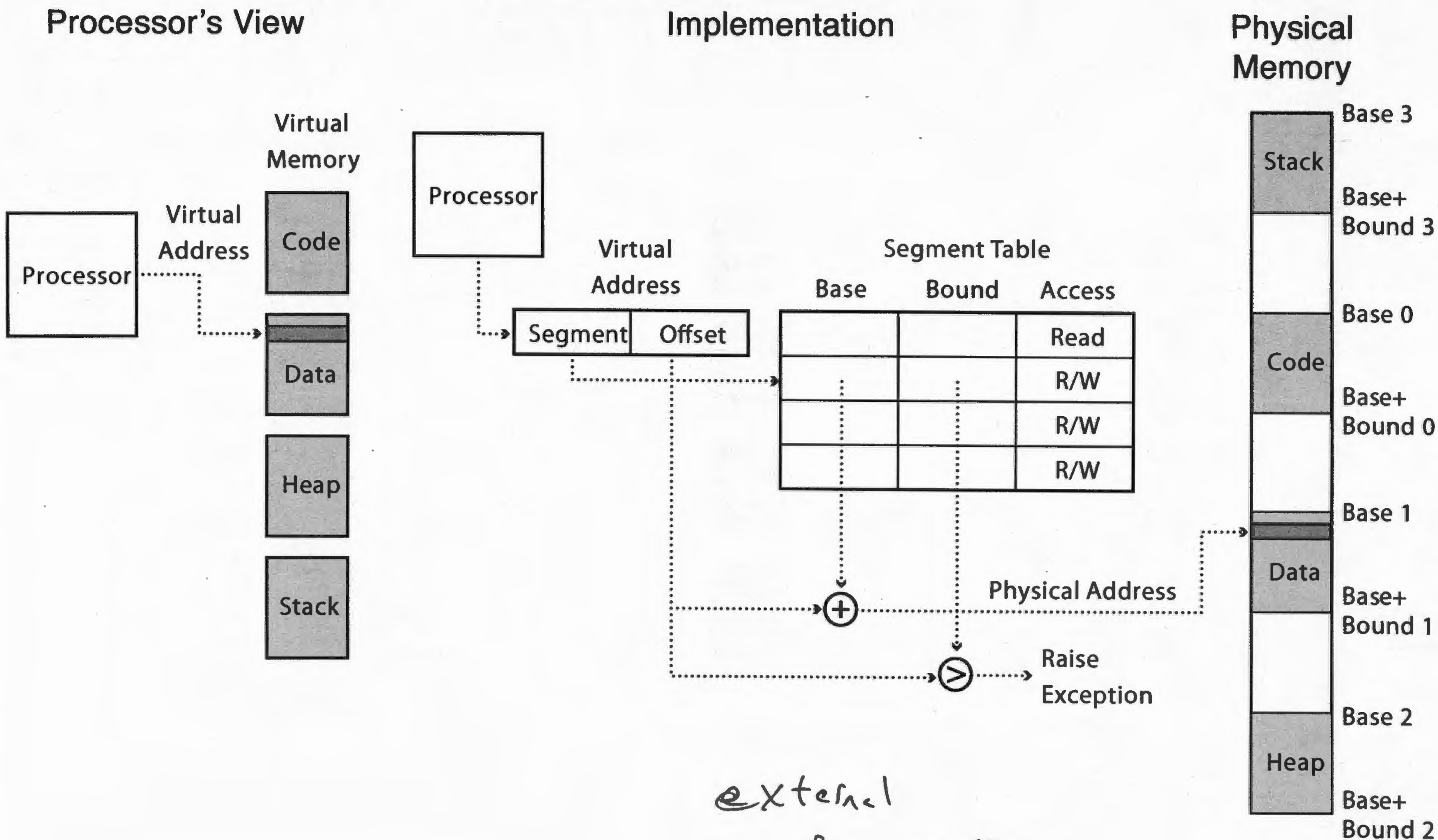
Process Regions or Segments

- Every process has logical regions or segments
  - Contiguous region of process memory
- Code, data, heap, stack, dynamic library (code, data), memory mapped files, …
- Each with its own
  - protection: read-only, read-write, execute-only
  - sharing: code vs. data
  - access pattern: code vs. mmap file

Segmentation

- Segment is a contiguous region of *virtual* memory
- Each process has a segment table (in hardware)
  - Entry in table = segment
- Segment can be located anywhere in physical memory
  - Each segment has: start, length, access permission
- Processes can share segments
  - Same start, length, same/different access permissions

Segmentation



external fragmentation

seg registers 2 bit segment #
offset 12 bit offset

	Segment start	length
code	0x4000	0x700
data	0	0x500
heap	-	-
stack	0x2000	0x1000

Virtual Memory

main: 0:240	store #108, r2
0:244	store pc+8, r31
0:248	jump 360
0:24c	
...	
strlen: 0:360	loadbyte (r2), r3
...	...
0:420	jump (r31)
...	
x: 1:108	a b c \0
...	

Physical Memory

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)
...	

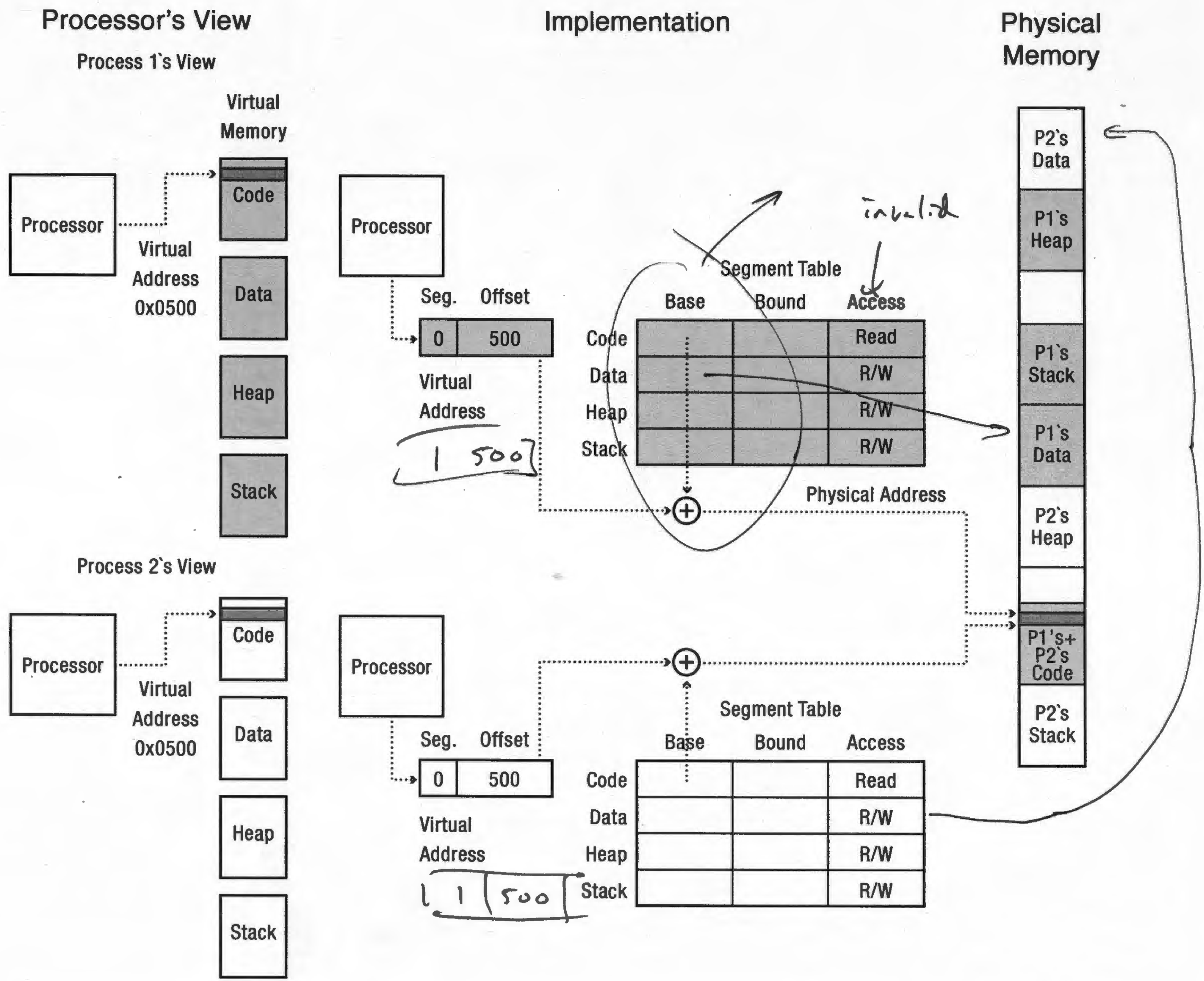
3: 240 graphics instr.

Question

- With segmentation, what is saved/restored on a process context switch?

x86: global descriptor table, registers

segment table

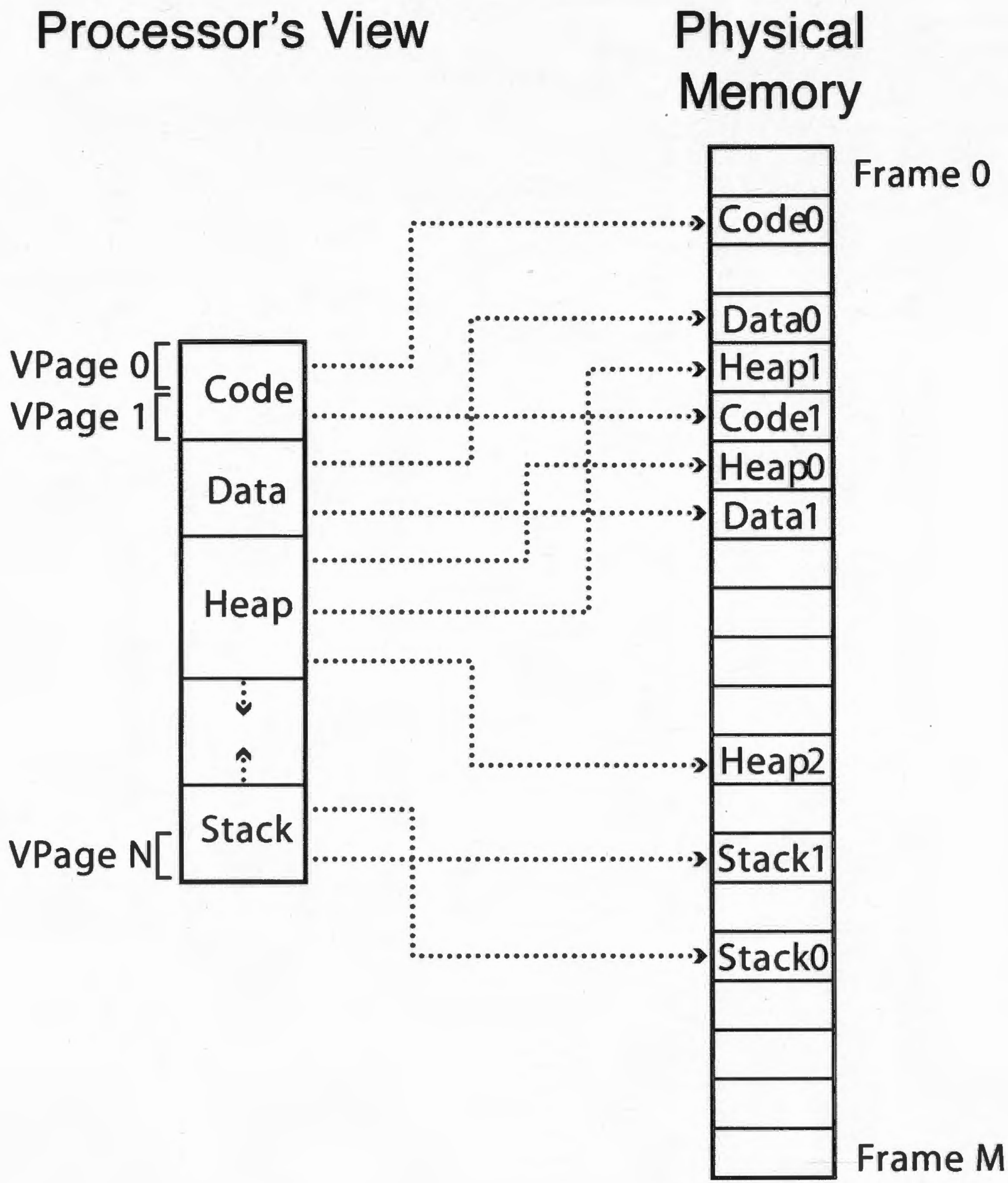

Processor's View
Implementation
Physical Memory
Process 1's View
Virtual Memory
Processor
Virtual Address 0x0500
Code
Data
Heap
Stack
Processor
Seg. Offset
0 500
Virtual Address
Segment Table
Base Bound Access
Code Read
Data R/W
Heap R/W
Stack R/W
Physical Address
P2's Data
P1's Heap
P1's Stack
P1's Data
P2's Heap
P1's+ P2's Code
P2's Stack
Process 2's View
Processor
Virtual Address 0x0500
Code
Data
Heap
Stack
Processor
Seg. Offset
0 500
Virtual Address
Segment Table
Base Bound Access
Code Read
Data R/W
Heap R/W
Stack R/W

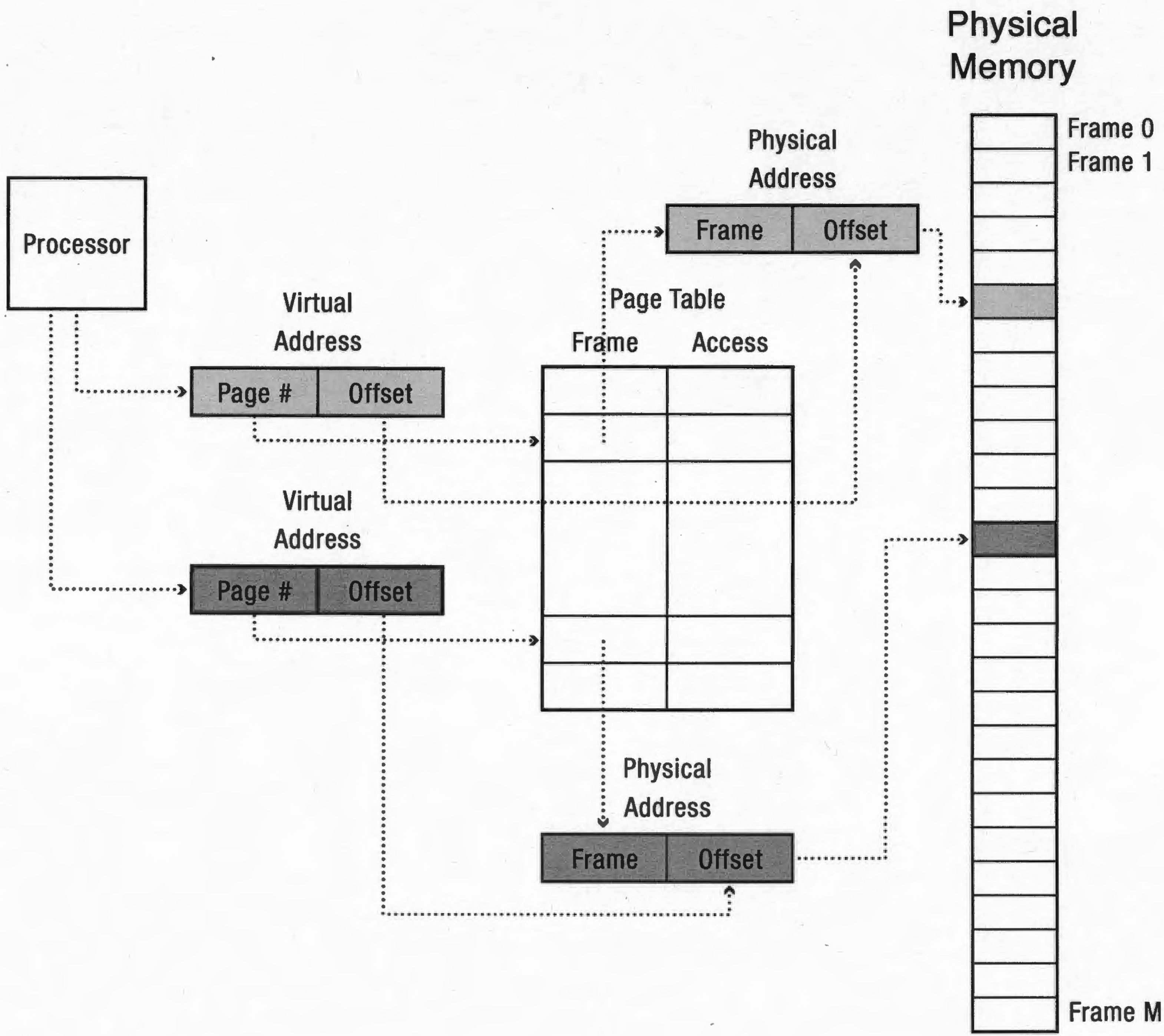
Segmentation

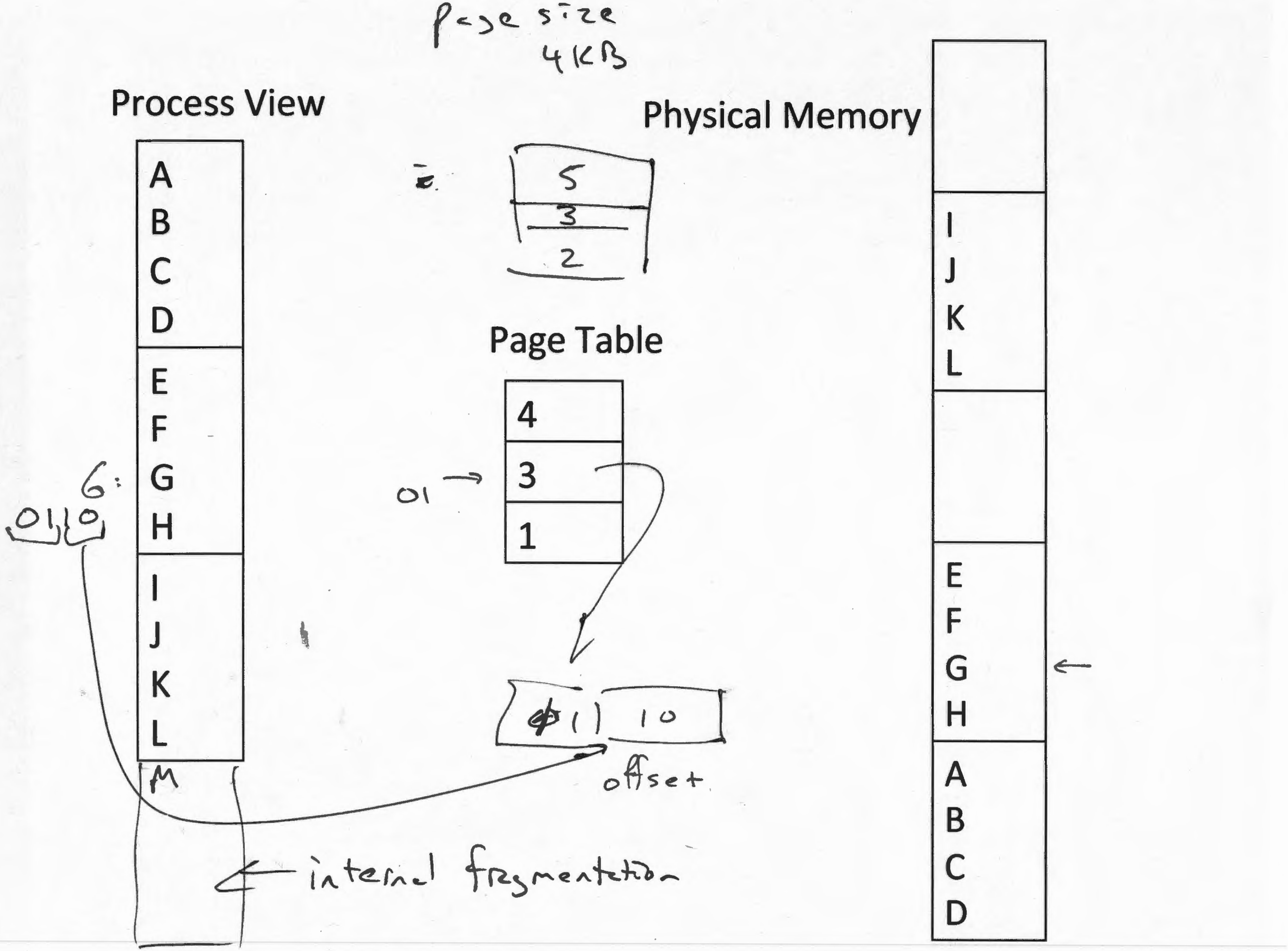
- Pros?
  - Can share code/data segments between processes
  - Can protect code segment from being overwritten
- Cons? Complex memory management
  - Need to find chunk of a particular size
  - May need to rearrange memory to make room for new segment or growing segment (e.g., sbrk)
  - External fragmentation: wasted space between chunks

Paged Translation

- Manage memory in fixed size units, or pages
- Finding a free page is easy
  - Bitmap allocation: 0011111100000001100
  - Each bit represents one physical page frame
- Each process has its own page table
  - Stored in physical memory
  - Hardware registers
    - pointer to page table start
    - page table length


Processor's View
Physical Memory
Frame 0
Code0
Data0
Heap1
Code1
Heap0
Data1
Heap2
Stack1
Stack0
Frame M
VPage 0
VPage 1
VPage N
Code
Data
Heap
Stack


Physical Memory
Frame 0
Frame 1
Frame M
Processor
Virtual Address
Page #
Offset
Virtual Address
Page #
Offset
Page Table
Frame
Access
Physical Address
Frame
Offset
Physical Address
Frame
Offset


page size
4KB
Process View
A
B
C
D
E
F
G
H
I
J
K
L
M
6:
0110
internal fragmentation
5
3
2
Page Table
4
3
1
01
ɸ1
10
offset
Physical Memory
I
J
K
L
E
F
G
H
A
B
C
D

Paging Questions

- With paging, what is saved/restored on a process context switch?
  - Pointer to page table, size of page table
  - Page table itself is in main memory
- What if page size is very small?
- What if page size is very large?
  - Internal fragmentation: if we don't need all of the space inside a fixed size chunk

Paging and Sharing

- Can we use page tables to share memory between processes?

- Set page tables to point to same page frame
- Need *core map*
  - Array of information about each physical page frame
  - Set of processes pointing to that page frame
  - When reference count goes to zero, can reclaim!

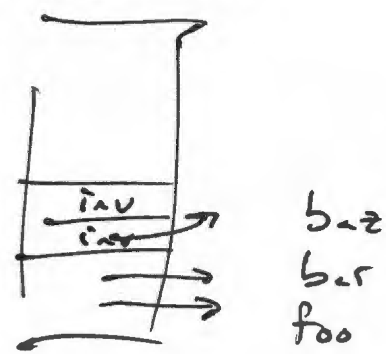
Question

- How big a user stack should I allocate?
- What if some programs need a large stack and others need a small one?

Expand Stack on Reference

- When program references memory beyond end of stack
  - Page fault into OS kernel
  - Kernel allocates some additional memory
    - How much?
  - Remember to zero the memory to avoid accidentally leaking information!
  - Modify page table
  - Resume process

```
baz () {
    int x[100000];
}
```



UNIX fork seems inefficient

- Makes a complete copy of process
- Throw copy away on exec
- Do we need to make the copy?
  - One solution: change the syscall interface!

Copy on Write

- Paging allows an efficient fork
  - Copy page table of parent into child
  - Mark all pages (in new/old page tables) as read-only
  - Start child process; restart parent
  - Trap into kernel on write (in child or parent)
  - Copy page
  - Mark ~~both~~ one as writeable, others as readonly except if last one ...
  - Resume execution

Reference counting