(Even) More Virtual Memory

ECE 469, Jan 30

Aravind Machiry

A gap among Architecture, Compiler and OS courses

main.c
math.c

compiler

main.o
math.o

linker?

a.out

loader

memory management

Load a.out to mem
Manage mem for proc

arch

Instruction execution

Example

Main.c:

```
extern float sin( );
main( )
{
  static float x, val;


  printf("Type number: ");
  scanf("%f", &x);
  val = sin(x);
  printf("Sine is %f", val);
}
```

Math.c:

```
float sin(float x)
{
  static float temp1, temp2, result;

  – Calculate Sine –

  return result ;
}
```

Example (cont)

- Main.c uses externally defined sin() and C library function calls
  - printf()
  - scanf()

- How does this program get compiled and linked?

Compiler

- Compiler: generates object file
  - Information is incomplete
  - Each file may refer to symbols defined in other files

Components of Object File

- Header
- Two segments
  - Code segment and data segment
  - OS adds empty heap/stack segment while loading
- Size and address of each segment
  - Address of a segment is the address where the segment begins.

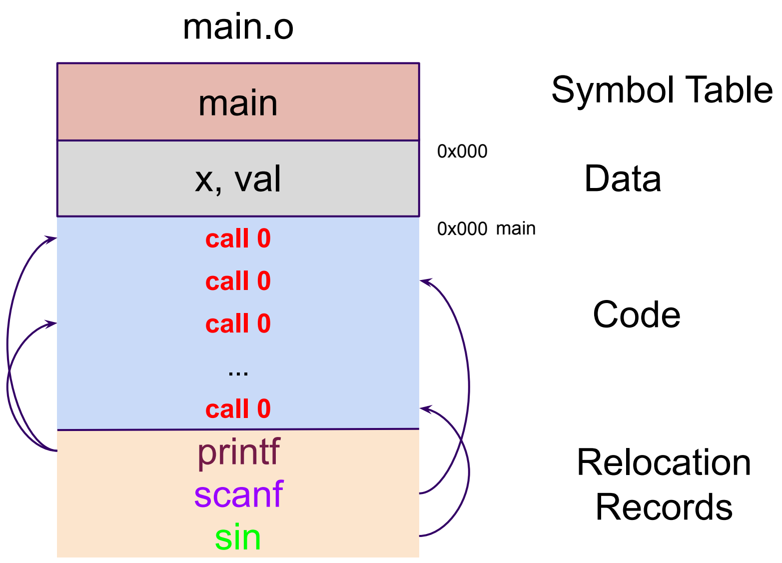
Components of Object File (cont)

- Symbol table
  - Information about stuff defined in this module
  - Used for getting from the name of a thing (subroutine/variable) to the thing itself
- Relocation information
  - Information about addresses in this module linker should fix
    - External references (e.g. lib call)
    - Internal references (e.g. absolute jumps)
- Additional information for debugger

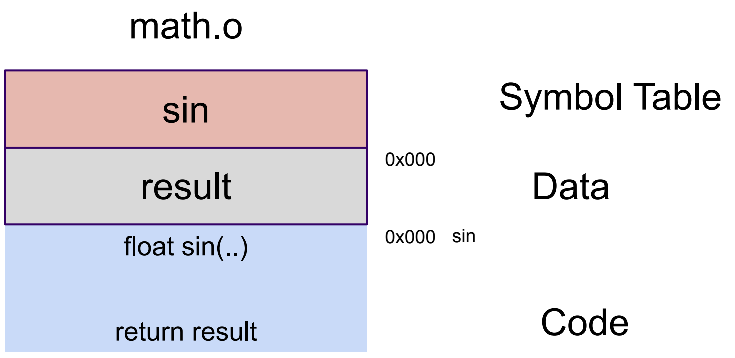
What could the compiler not do?

- Compiler does not know final memory layout
  - It assumes everything in .o starts at address zero
  - For each .o file, compiler puts information in the symbol table to tell the linker how to rearrange outside references safely/efficiently
    - For exported functions, absolute jumps, etc

Compiler: main.c



Compiler: math.c



Linker functionality

- Three functions of a linker
  - Collect all the pieces of a program
  - Figure out new memory organization
    - Combine like segments
    - Does the ordering matter? (spatial locality for cache)
  - Touch-up addresses

- The result is a runnable object file (e.g. a.out)

Linker – a closer look

- Linker can shuffle segments around at will, but cannot rearrange information within a segment

Linker requires at least two passes

- Pass 1: decide how to arrange memory
- Pass 2: address touch-up

Pass 1 – Segment Relocation

- Pass 1 assigns input segment locations to fill-up output segments
  - Read and adjust symbol table information
  - Read relocation info to see what additional stuff from libraries is required

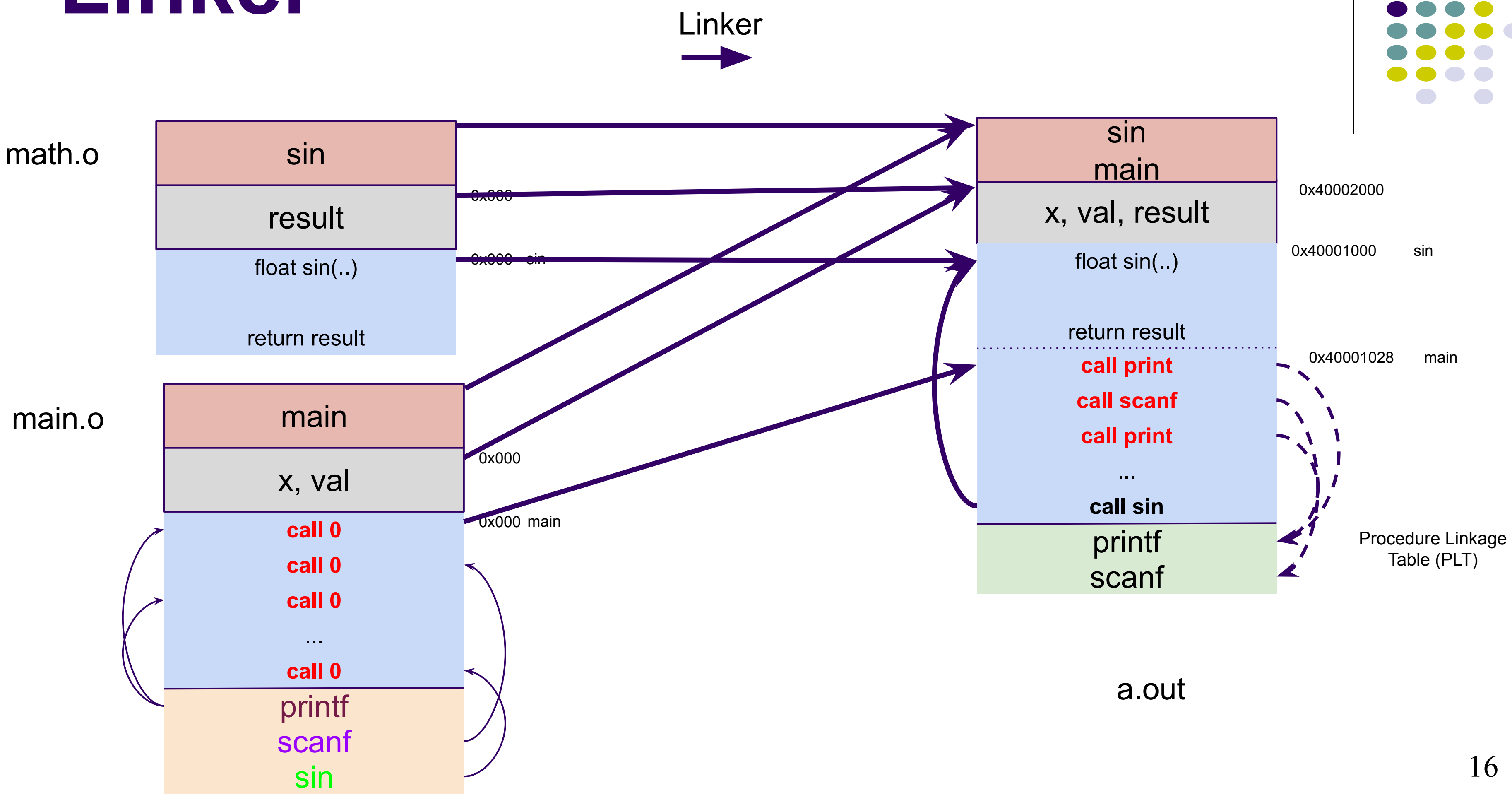
Pass 2 – Address translation

- In pass 2, linker reads segment and relocation information from files, fixes up addresses, and writes a new object file
- Relocation information is crucial for this part

Putting It Together

- Pass 1:
  - Read symbol table, relocation table
  - Rearrange segments, adjust symbol table
- Pass 2:
  - Read segments and relocation information
  - Touch-up addresses
  - Write new object file

Linker



Linker
math.o
sin
result
float sin(..)
return result
0x000
0x000 sin
main.o
main
x, val
call 0
call 0
call 0
...
call 0
printf
scanf
sin
0x000
0x000 main
sin
main
x, val, result
float sin(..)
return result
call print
call scanf
call print
...
call sin
printf
scanf
0x40002000
0x40001000 sin
0x40001028 main
Procedure Linkage Table (PLT)
a.out

Dynamic linking

- Static linking – each lib copied into each binary
- Dynamic linking:
  - Create wrapper code for library calls, a stub that finds lib code in memory, or loads it if it is not present
- Pros:
  - all procs can share copy (shared libraries)
    - Standard C library
  - live updates

Dynamic loading

- Program can call dynamic linker via
  - dlopen()
  - library is loaded at running time
- Pros:
  - More flexibility -- A running program can
    - create a new program
    - invoke the compiler
    - invoke the linker
    - load it!

Memory Usage Classification

- Memory required by a program can be used in various ways
- Some possible classifications
  - Role in programming language
  - Changeability
  - Address vs. data
  - Binding time

Role in Programming Language

- Instructions
  - Specify the operations to be performed on the operands
- Variables
  - Store the information that changes as program runs
- Constants
  - Used as operands but never change

Changeability

- Read-only
  - Example: code, constants
- Read and write
  - Example: Variables

Address vs. Data

- Need to distinguish between addresses and data
- Why?
  - Addresses need to be modified if the memory is re-arranged

Binding Time

- When is the space allocated?
  - Compile-time, link-time, or load-time
  - Static: arrangement determined once and for all
  - Dynamic: arrangement cannot be determined until runtime, and may change
    - malloc( ), free( )

Classification – summary

- Classifications overlap
  - Variables may be static or dynamic
  - Code may be read-only or read and write
    - Read-only: Solaris
    - Read and write: DOS
- So what is this all about?
- What does memory look like when a process is running?

Memory Layout

- Memory divided into segments
  - Code (called text in Unix terminology)
  - Data
  - Stack

- Why different segments?
  - To enforce classification
  - e.g. code and data treated differently at hardware level

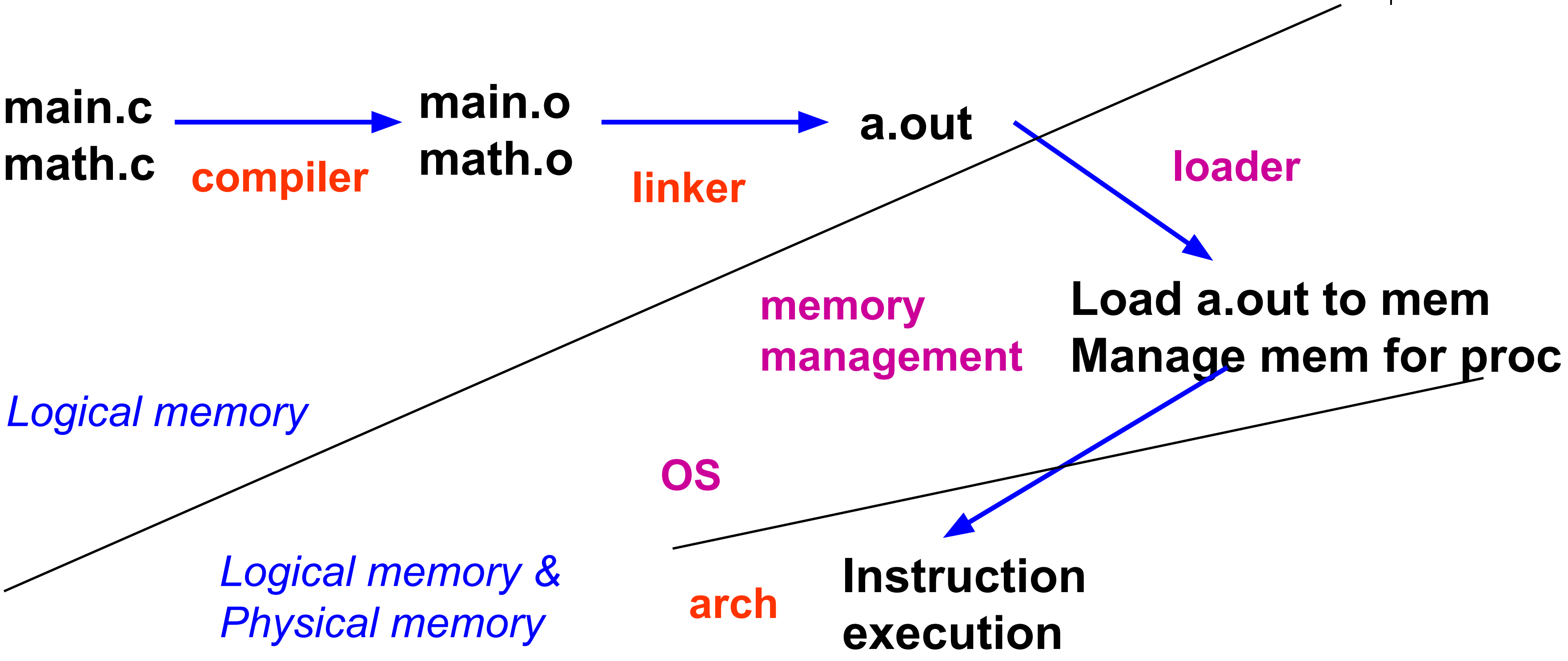
The big picture

- a.out needs address space for
  - text seg, data seg, and (hypothetical) heap, stack
- A running process needs phy. memory for
  - text seg, data seg, heap, stack
- But no way of knowing where in phy mem at
  - Programming time, compile time, linking time
- Best way out?
  - Make agreement to divide responsibility
    - Assume address starts at 0 at prog/compile/link time
    - OS needs to work hard at loading/runing time

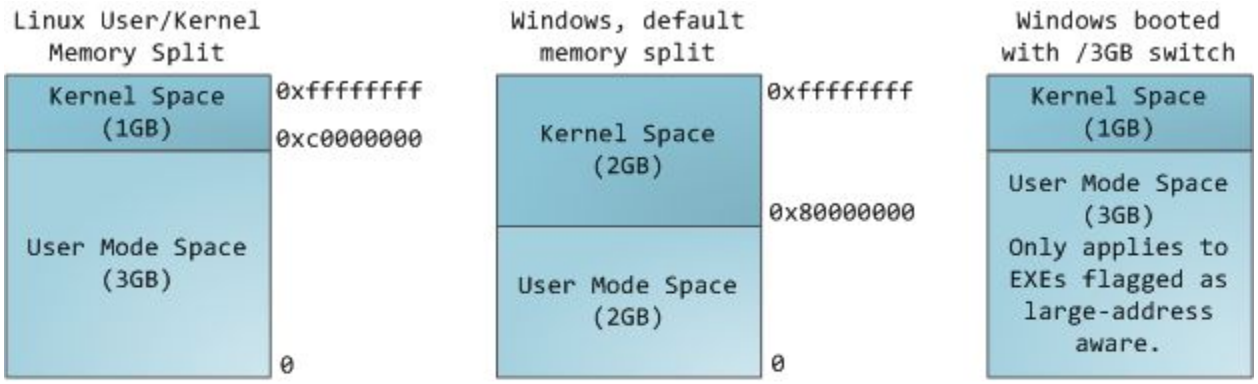
Big picture (cont)

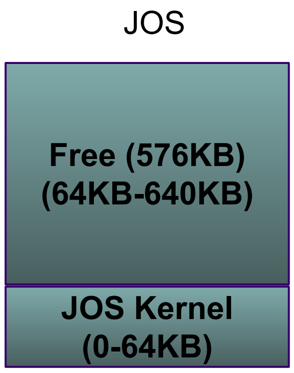
- OS deals with physical memory
  - Loading
  - Sharing physical memory between processes
  - Dynamic memory allocation

Connecting the dots



Process memory map

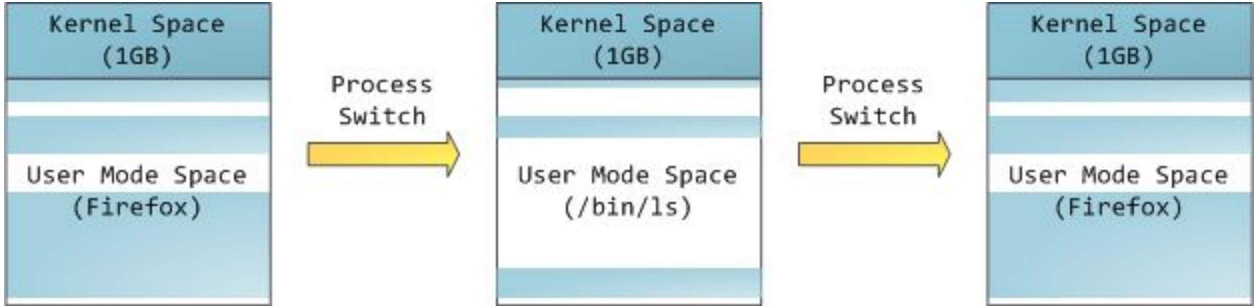




On ubuntu (check kernel map): sudo cat /proc/iomem

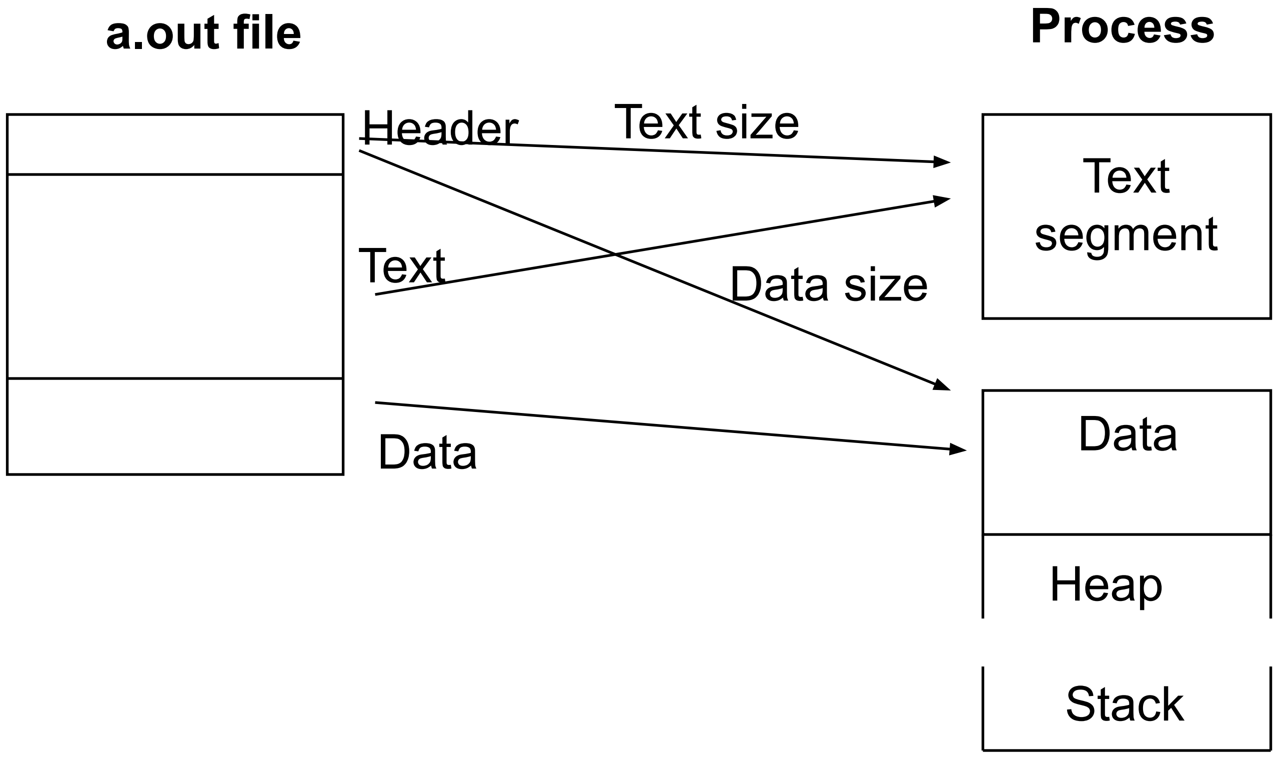
Copied from: https://manybutfinite.com/post/anatomy-of-a-program-in-memory/

Easier context-switch



Copied from: https://manybutfinite.com/post/anatomy-of-a-program-in-memory/

Loading



a.out file
Process
Header
Text size
Text
Data size
Data
Text segment
Data
Heap
Stack

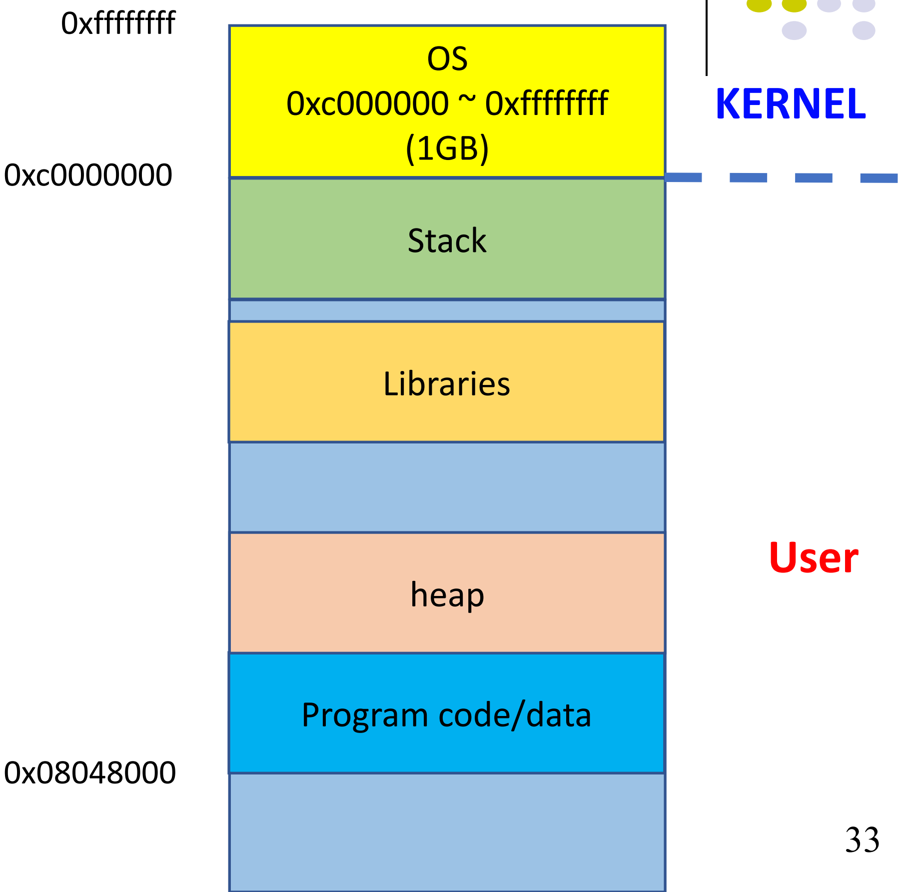
Dynamic memory allocation during program execution

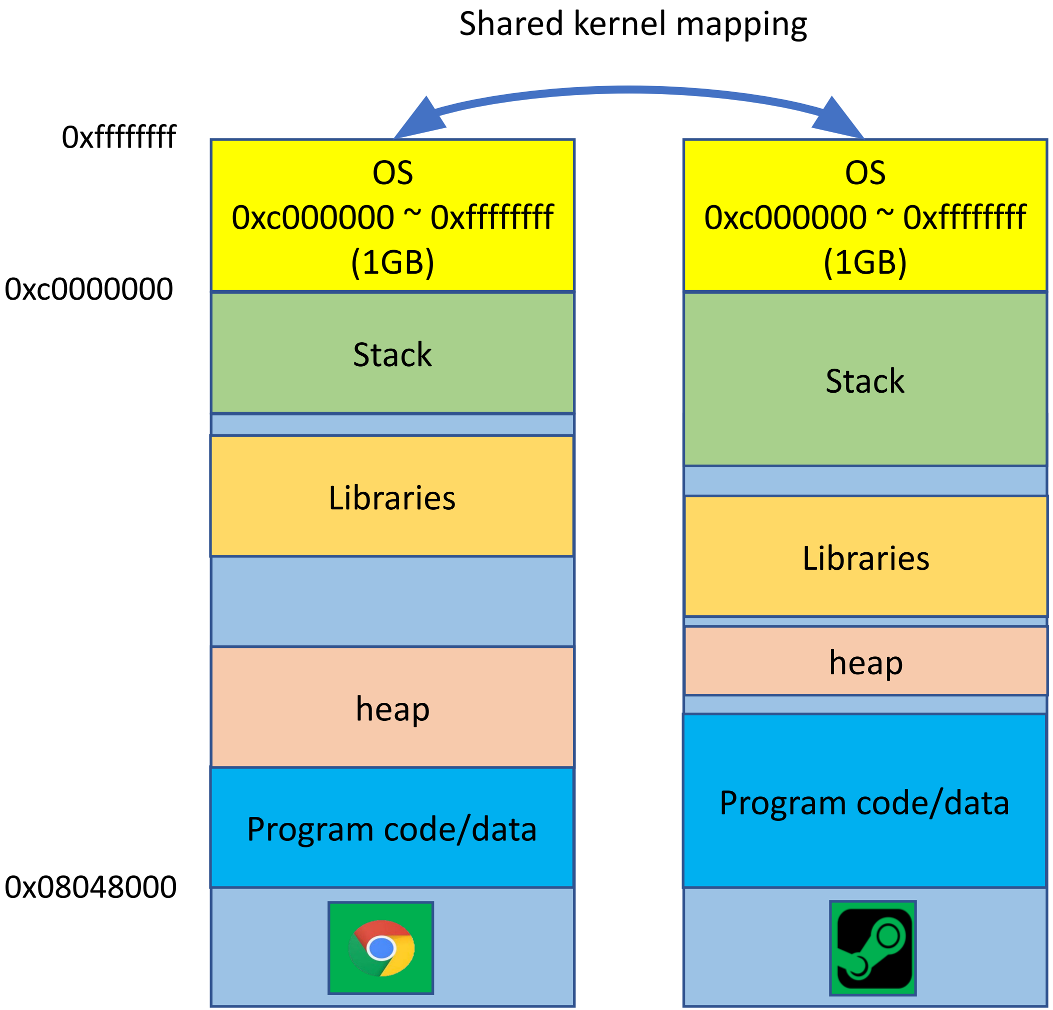
- Stack: for procedure calls
- Heap: for malloc()
- Both dynamically growing/shrinking

- Assumption for now:
  - Heap and stack are fixed size
  - OS has to worry about loading 4 segments per process:
    - Text
    - Data
    - Heap
    - stack

Process Virtual Memory Layout

- OS allocates a separate virtual memory space to each process
- Transparency
  - Do not have to worry about a system's memory usage status
- Isolation
  - Others can't access my virtual memory space





Why kernel is mapped in all user processes?

Easier context-switch

Memory Maps on x64 machine

```
machiry@machiry-home:~$ cat /proc/self/maps | tail
7fe7bcb0a000-7fe7bcb0b000 r--p 00000000 103:02 35785028                  /usr/lib/x86_64-linux-gnu/ld-2.31.so
7fe7bcb0b000-7fe7bcb2e000 r-xp 00001000 103:02 35785028                  /usr/lib/x86_64-linux-gnu/ld-2.31.so
7fe7bcb2e000-7fe7bcb36000 r--p 00024000 103:02 35785028                  /usr/lib/x86_64-linux-gnu/ld-2.31.so
7fe7bcb37000-7fe7bcb38000 r--p 0002c000 103:02 35785028                  /usr/lib/x86_64-linux-gnu/ld-2.31.so
7fe7bcb38000-7fe7bcb39000 rw-p 0002d000 103:02 35785028                  /usr/lib/x86_64-linux-gnu/ld-2.31.so
7fe7bcb39000-7fe7bcb3a000 rw-p 00000000 00:00 0
7ffda7458000-7ffda7479000 rw-p 00000000 00:00 0                          [stack]
7ffda7584000-7ffda7588000 r--p 00000000 00:00 0                          [vvar]
7ffda7588000-7ffda758a000 r-xp 00000000 00:00 0                          [vdso]
ffffffffff600000-ffffffffff601000 --xp 00000000 00:00 0                  [vsyscall]
```

Memory Maps on x64 machine

```
machiry@machiry-home:~$ cat /proc/self/maps | tail
7fe7bcb0a000-7fe7bcb0b000 r--p 00000000 103:02 35785028                  /usr/lib/x86_64-linux-gnu/ld-2.31.so
7fe7bcb0b000-7fe7bcb2e000 r-xp 00001000 103:02 35785028                  /usr/lib/x86_64-linux-gnu/ld-2.31.so
7fe7bcb2e000-7fe7bcb36000 r--p 00024000 103:02 35785028                  /usr/lib/x86_64-linux-gnu/ld-2.31.so
7fe7bcb37000-7fe7bcb38000 r--p 0002c000 103:02 35785028                  /usr/lib/x86_64-linux-gnu/ld-2.31.so
7fe7bcb38000-7fe7bcb39000 rw-p 0002d000 103:02 35785028                  /usr/lib/x86_64-linux-gnu/ld-2.31.so
7fe7bcb39000-7fe7bcb3a000 rw-p 00000000 00:00 0
7ffda7458000-7ffda7479000 rw-p 00000000 00:00 0                          [stack]
7ffda7584000-7ffda7588000 r--p 00000000 00:00 0                          [vvar]
7ffda7588000-7ffda758a000 r-xp 00000000 00:00 0                          [vdso]
ffffffffff600000-ffffffffff601000 --xp 00000000 00:00 0                  [vsyscall]
```

```
machiry@machiry-home:~$ cat /proc/5668/maps | tail
7f300561e000-7f3005641000 r-xp 00001000 103:02 35785028                  /usr/lib/x86_64-linux-gnu/ld-2.31.so
7f3005641000-7f3005649000 r--p 00024000 103:02 35785028                  /usr/lib/x86_64-linux-gnu/ld-2.31.so
7f3005649000-7f300564a000 r--s 00000000 00:36 204                        /run/user/1000/dconf/user
7f300564a000-7f300564b000 r--p 0002c000 103:02 35785028                  /usr/lib/x86_64-linux-gnu/ld-2.31.so
7f300564b000-7f300564c000 rw-p 0002d000 103:02 35785028                  /usr/lib/x86_64-linux-gnu/ld-2.31.so
7f300564c000-7f300564d000 rw-p 00000000 00:00 0
7ffc023b3000-7ffc023d4000 rw-p 00000000 00:00 0                          [stack]
7ffc023ea000-7ffc023ee000 r--p 00000000 00:00 0                          [vvar]
7ffc023ee000-7ffc023f0000 r-xp 00000000 00:00 0                          [vdso]
ffffffffff600000-ffffffffff601000 --xp 00000000 00:00 0                  [vsyscall]
```

How does OS ensure a user process does not access kernel memory?

- OS needs to ensure that a user process cannot access (read/write) kernel (or OS memory)?
  - Why?
    - Hint: Security!
      - Remember: sudo!?

How does OS ensure a user process does not access kernel memory?

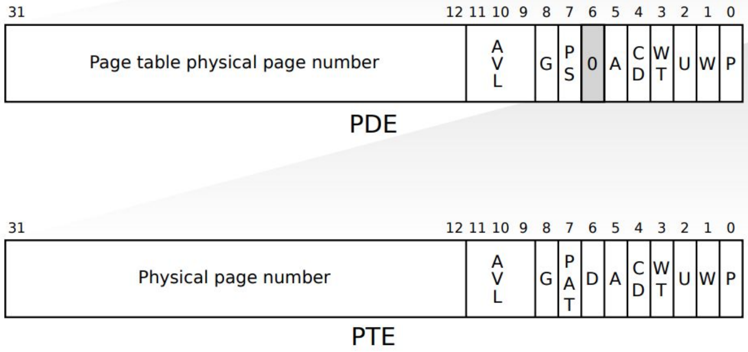
- OS needs to ensure that a user process cannot access (read/write) kernel (or OS memory)?
  - Why?
    - Hint: Security!
      - Remember: sudo!?

- Permissions bits in Page directories and Page Tables!!

Page Directory / Table Entry (PDE/PTE)

- Top 20 bits: physical page number
  - Physical page number of a page table (PDE)
  - Physical page number of the requested virtual address (PTE)

- Lower 12 bits: some flags
  - Permission
  - Etc.

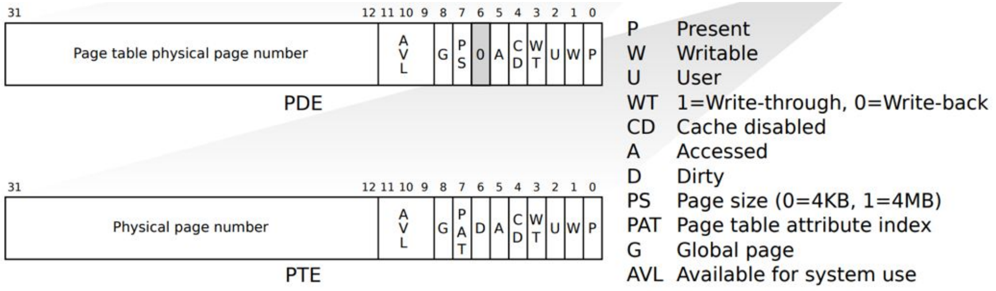


P Present
W Writable
U User
WT 1=Write-through, 0=Write-back
CD Cache disabled
A Accessed
D Dirty
PS Page size (0=4KB, 1=4MB)
PAT Page table attribute index
G Global page
AVL Available for system use

Permission Flags

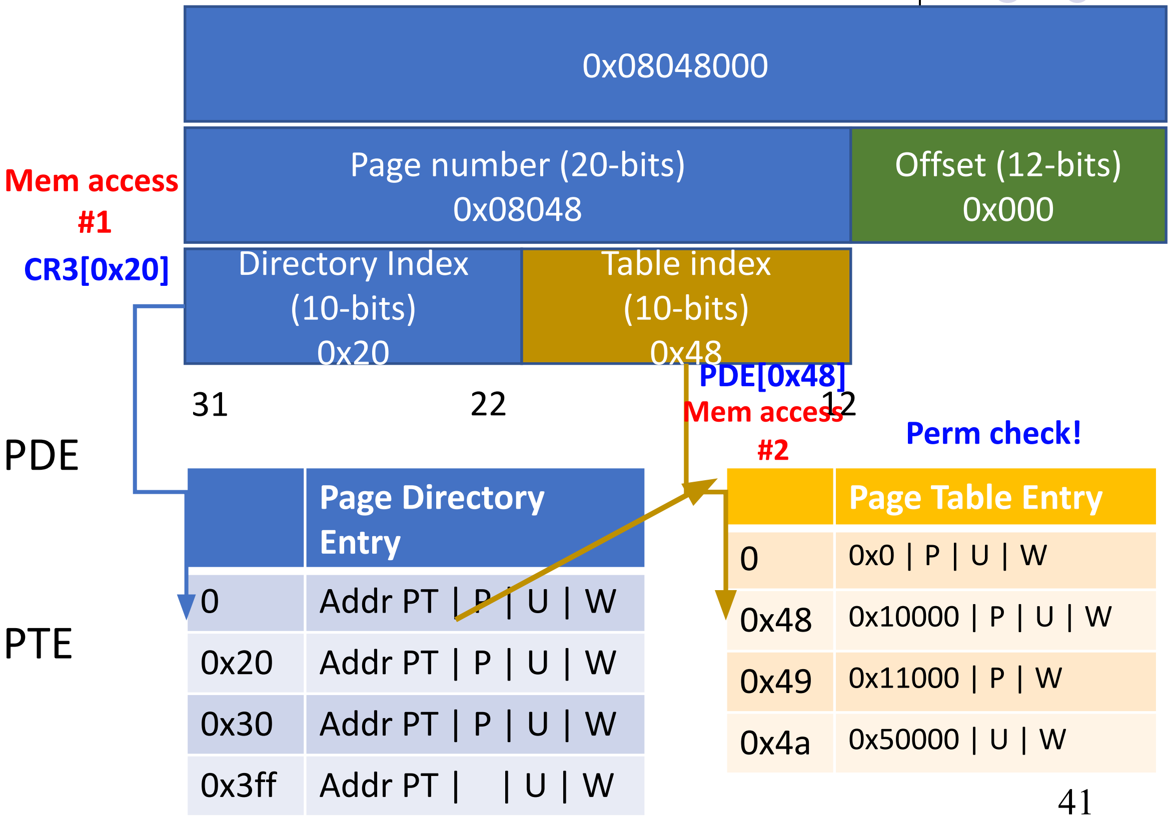
- PTE_P (PRESENT)
  - 0: invalid entry
  - 1: valid entry
- PTE_W (WRITABLE)
  - 0: read only
  - 1: writable
- PTE_U (USER)
  - 0: kernel (only ring 0 can access)
  - 1: user (accessible by ring 3)

	Page Table Entry	
0	Addr PT	
0x48	0x10000 << 12 | PTE_U | PTE_W	Invalid
0x49	0x11000 << 12 | PTE_P | PTE_W	Kernel, writable
0x4a	0x50000 << 12 | PTE_P | PTE_U	User, read-only



When CPU Checks Permission Bits?

- In address translation
- 1. Virtual address
- 2. PDE = CR3[PDX]
    - Checks permission bits in PDE
- 3. PTE = PDE[PTX]
    - Checks permission bits in PTE



When CPU Checks Permission Bits?

- A virtual memory address is inaccessible if PDE disallows the access
- A virtual memory address is inaccessible if PTE disallows the access
- Both **PDE and PTE should allow the access**…

PDE/PTE Permission Examples 0

- Virtual address 0x01020304
- PDE: PTE_P | PTE_W | PTE_U
- PTE: PTE_P | PTE_W | PTE_U
- **Can user (ring 3) access it? Is it writable?**

- PTE_P (PRESENT)
  - 0: invalid entry
  - 1: valid entry
- PTE_W (WRITABLE)
  - 0: read only
  - 1: writable
- PTE_U (USER)
  - 0: kernel (only ring 0 can access)
  - 1: user (accessible by ring 3)

PDE/PTE Permission Examples 0

- Virtual address 0x01020304
- PDE: PTE_P | PTE_W | PTE_U
- PTE: PTE_P | PTE_W | PTE_U
- **Can user (ring 3) access it? Is it writable?**
  - Valid, accessible by ring 3, and writable

- PTE_P (PRESENT)
  - 0: invalid entry
  - 1: valid entry
- PTE_W (WRITABLE)
  - 0: read only
  - 1: writable
- PTE_U (USER)
  - 0: kernel (only ring 0 can access)
  - 1: user (accessible by ring 3)

PDE/PTE Permission Examples 1

- Virtual address 0x01020304
- PDE: PTE_P | PTE_W | PTE_U
- PTE: PTE_P | PTE_U
- **Can user (ring 3) access it? Is it writable?**

- PTE_P (PRESENT)
  - 0: invalid entry
  - 1: valid entry
- PTE_W (WRITABLE)
  - 0: read only
  - 1: writable
- PTE_U (USER)
  - 0: kernel (only ring 0 can access)
  - 1: user (accessible by ring 3)

PDE/PTE Permission Examples 1

- Virtual address 0x01020304
- PDE: PTE_P | PTE_W | PTE_U
- PTE: PTE_P | PTE_U
- **Can user (ring 3) access it? Is it writable?**
  - Valid, accessible by ring 3, but not writable

- PTE_P (PRESENT)
  - 0: invalid entry
  - 1: valid entry
- PTE_W (WRITABLE)
  - 0: read only
  - 1: writable
- PTE_U (USER)
  - 0: kernel (only ring 0 can access)
  - 1: user (accessible by ring 3)

PDE/PTE Permission Examples 2

- Virtual address 0x01020304
- PDE: PTE_P | PTE_U
- PTE: PTE_P | PTE_W | PTE_U
- **Can user (ring 3) access it? Is it writable?**

- PTE_P (PRESENT)
  - 0: invalid entry
  - 1: valid entry
- PTE_W (WRITABLE)
  - 0: read only
  - 1: writable
- PTE_U (USER)
  - 0: kernel (only ring 0 can access)
  - 1: user (accessible by ring 3)

PDE/PTE Permission Examples 2

- Virtual address 0x01020304
- PDE: PTE_P | PTE_U
- PTE: PTE_P | PTE_W | PTE_U
- **Can user (ring 3) access it? Is it writable?**
  - Valid, accessible by ring 3, but not writable

- PTE_P (PRESENT)
  - 0: invalid entry
  - 1: valid entry
- PTE_W (WRITABLE)
  - 0: read only
  - 1: writable
- PTE_U (USER)
  - 0: kernel (only ring 0 can access)
  - 1: user (accessible by ring 3)

PDE/PTE Permission Examples 3

- Virtual address 0x01020304
- PDE: PTE_P | PTE_W | PTE_U
- PTE: PTE_P
- **Can user (ring 3) access it? Is it writable?**

- PTE_P (PRESENT)
  - 0: invalid entry
  - 1: valid entry
- PTE_W (WRITABLE)
  - 0: read only
  - 1: writable
- PTE_U (USER)
  - 0: kernel (only ring 0 can access)
  - 1: user (accessible by ring 3)

PDE/PTE Permission Examples 3

- Virtual address 0x01020304
- PDE: PTE_P | PTE_W | PTE_U
- PTE: PTE_P
- **Can user (ring 3) access it? Is it writable?**
  - valid, inaccessible by ring3, not writable

- PTE_P (PRESENT)
  - 0: invalid entry
  - 1: valid entry
- PTE_W (WRITABLE)
  - 0: read only
  - 1: writable
- PTE_U (USER)
  - 0: kernel (only ring 0 can access)
  - 1: user (accessible by ring 3)

PDE/PTE Permission Examples 4

- Virtual address 0x01020304
- PDE: PTE_P | PTE_W
- PTE: PTE_P | PTE_U
- **Can user (ring 3) access it? Is it writable?**

- PTE_P (PRESENT)
  - 0: invalid entry
  - 1: valid entry
- PTE_W (WRITABLE)
  - 0: read only
  - 1: writable
- PTE_U (USER)
  - 0: kernel (only ring 0 can access)
  - 1: user (accessible by ring 3)

PDE/PTE Permission Examples 4

- Virtual address 0x01020304
- PDE: PTE_P | PTE_W
- PTE: PTE_P | PTE_U
- **Can user (ring 3) access it? Is it writable?**
  - valid, inaccessible by ring3, not writable

- PTE_P (PRESENT)
  - 0: invalid entry
  - 1: valid entry
- PTE_W (WRITABLE)
  - 0: read only
  - 1: writable
- PTE_U (USER)
  - 0: kernel (only ring 0 can access)
  - 1: user (accessible by ring 3)

PDE/PTE Permission Examples 5

- Virtual address 0x01020304
- PDE: PTE_P | PTE_U
- PTE: PTE_U
- **Can user (ring 3) access it? Is it writable?**

- PTE_P (PRESENT)
  - 0: invalid entry
  - 1: valid entry
- PTE_W (WRITABLE)
  - 0: read only
  - 1: writable
- PTE_U (USER)
  - 0: kernel (only ring 0 can access)
  - 1: user (accessible by ring 3)

PDE/PTE Permission Examples 5

- Virtual address 0x01020304
- PDE: PTE_P | PTE_U
- PTE: PTE_U
- **Can user (ring 3) access it? Is it writable?**
  - invalid

- PTE_P (PRESENT)
  - 0: invalid entry
  - 1: valid entry
- PTE_W (WRITABLE)
  - 0: read only
  - 1: writable
- PTE_U (USER)
  - 0: kernel (only ring 0 can access)
  - 1: user (accessible by ring 3)

PDE/PTE Permission Examples 6

- Virtual address 0x01020304
- PDE: PTE_U
- PTE: PTE_P | PTE_U
- **Can user (ring 3) access it? Is it writable?**

- PTE_P (PRESENT)
  - 0: invalid entry
  - 1: valid entry
- PTE_W (WRITABLE)
  - 0: read only
  - 1: writable
- PTE_U (USER)
  - 0: kernel (only ring 0 can access)
  - 1: user (accessible by ring 3)

PDE/PTE Permission Examples 6

- Virtual address 0x01020304
- PDE: PTE_U
- PTE: PTE_P | PTE_U
- **Can user (ring 3) access it? Is it writable?**
  - invalid

- PTE_P (PRESENT)
  - 0: invalid entry
  - 1: valid entry
- PTE_W (WRITABLE)
  - 0: read only
  - 1: writable
- PTE_U (USER)
  - 0: kernel (only ring 0 can access)
  - 1: user (accessible by ring 3)

Valid permission bits..

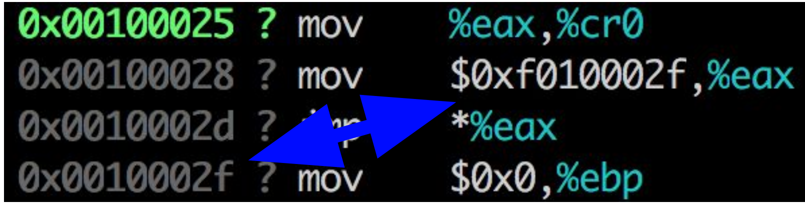
- Kernel: R, User: --
  - PTE_P
- Kernel: R, User: R
  - PTE_P | PTE_U
- Kernel: RW, User: RW
  - PTE_P | PTE_U | PTE_W

Cannot have permissions such as …

- Kernel: RW, User: R
  - PTE_P | PTE_W | PTE_U -> User RW…
  - PTE_P | PTE _W -> User --
- Kernel: R, User: RW
  - PTE_P | PTE_U | PTE_W -> Kernel RW…
  - PTE_P | PTE_U   -> User R…
- Kernel: --, User: RW
  - PTE_P | PTE_U | PTE_W -> Kernel RW…

Flexibility of virtual memory!

- Virtual to physical address mapping is in N-to-1 relation
  - N number of virtual addresses could be mapped to 1 physical address
- E.g., for a physical address 0x100000
  - JOS maps VA 0x100000 to PA 0x100000
  - JOS maps VA 0xf0100000 to PA 0x100000
- Why?
  - EIP before enabling paging: 0x100025
  - EIP after enabling paging: 0x100028
  - Then jumps to 0xf010002f



Sharing a Physical Page!

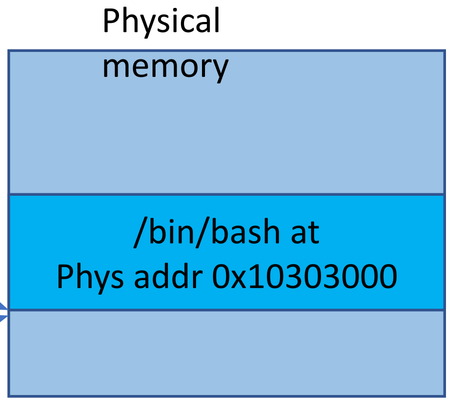
- Example: Loading of the same program

Process 0, runs /bin/bash, loads at virt addr 0x35555000

	Page Table Entry
0	…
0x155	0x10303 | FLAG

Process 1, runs /bin/bash, loads at virt addr 0x43132000

	Page Table Entry
0	…
0x132	0x10303 | FLAG



2 or more mappings to 0x10303000 is possible!

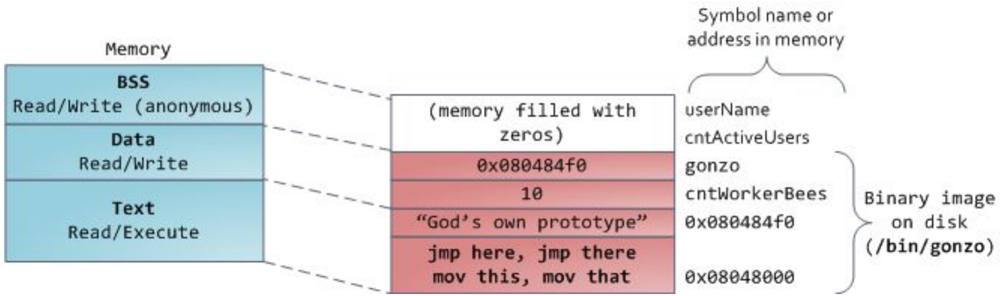
Allocating Virtual Memory

- Static allocation is inefficient:
  - Why don’t we just allocate entire virtual address space to a process?
  - Inefficient: The process may not access entire virtual address space

- Solution: Dynamic, Request based

Dynamic space allocation

- OS allocates space (valid PTE entries in page table) as dictated by the program binary and start running the process.



Dynamic space allocation

- When a process tries to access memory that is not allocated i.e., there is no corresponding valid PTE, then, OS kills the process.
  - E.g., Segmentation Fault!

- A process needs to **explicitly request OS** to allocate additional space (and create valid PTEs).
  - brk **system call** (We will cover this later)

Dynamic space allocation

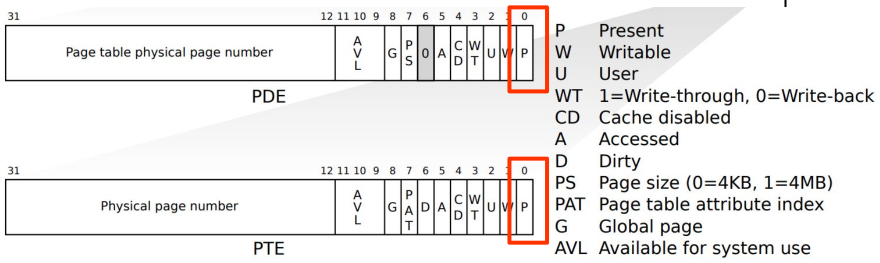
- We use malloc and free, which actually use brk system call internally

- brk only allocates virtual memory! not physical memory!

Dynamic space allocation: Example

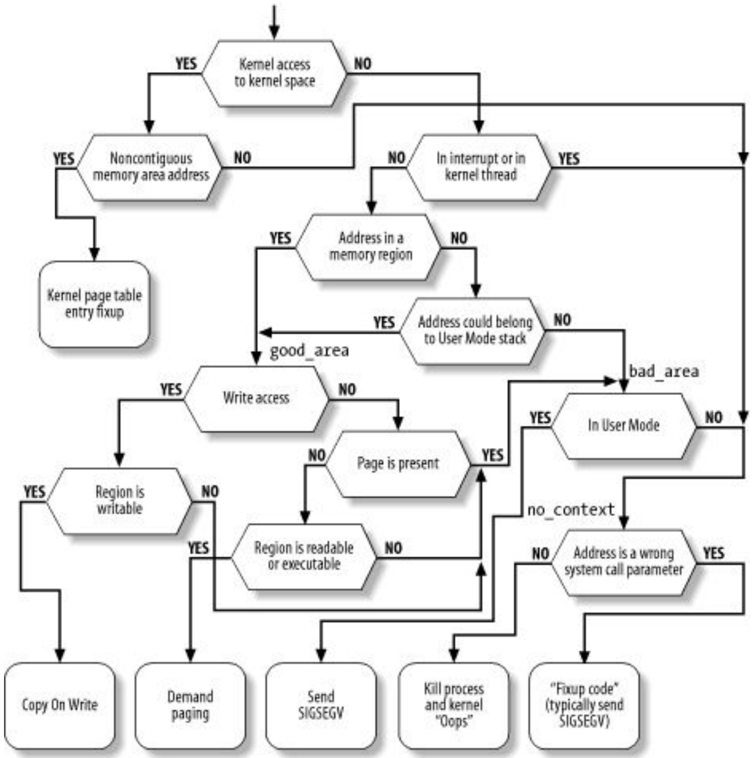
```
#include<stdio.h>
#include<stdlib.h>
int main() {
	char p;
	int *i = (int *)malloc(16*1024*1024*1024);
	printf("We just requested 16GB of virtual memory!!\n");
	printf("Check memory now, using htop!\n");
	printf("You will be surprised to see your memory usage.\nPress any key to exit.\n");
	scanf("%c", &p);
	return 0;

}
```

What happens when we call malloc?



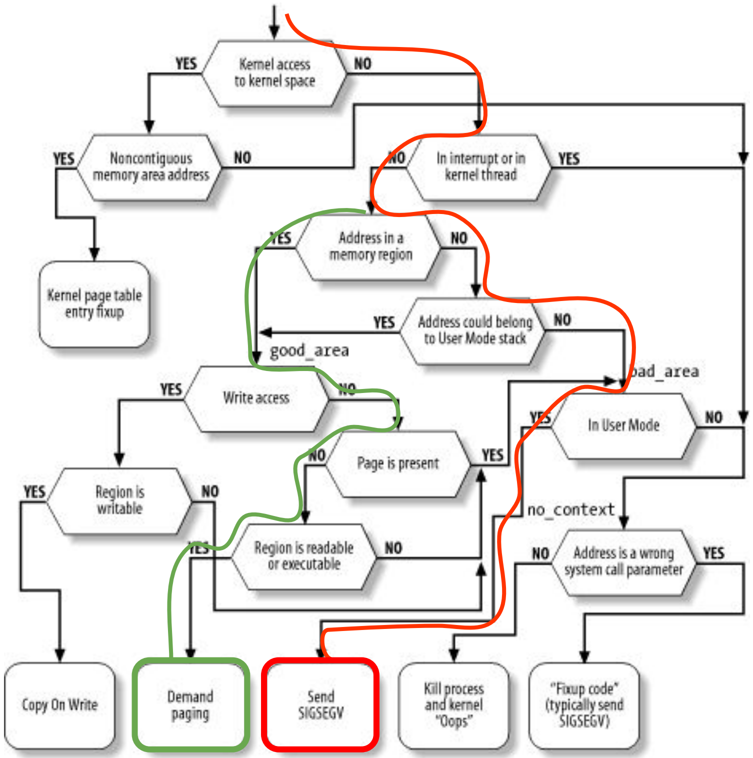
- Before malloc()?
  - No PTEs
- After malloc()?
  - **PDE/PTE updated but present bit not-set**
- Upon first access?
  - Assign physical page (and page table) and set the valid bit.

Handling Page faults in Kernel



Handling Page faults in Kernel

- Accessing unallocated memory: **Demand Paging**



- Accessing Unassigned Virtual Memory: **Segmentation Fault**