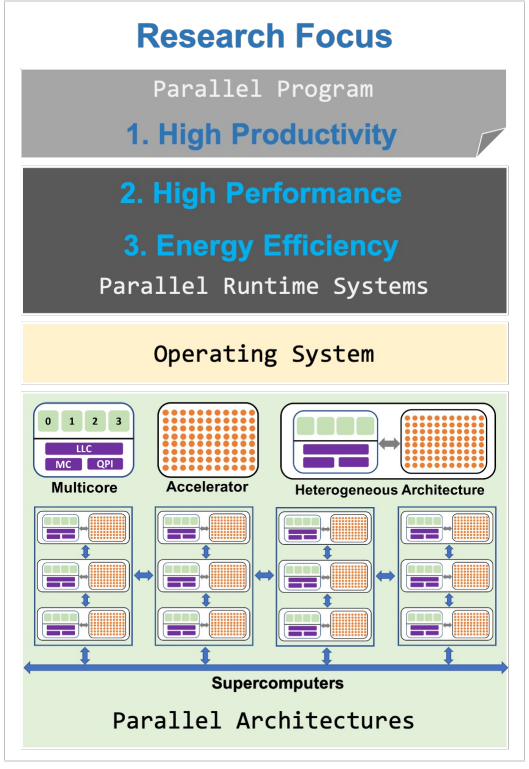
Lecture 01: Course Introduction

Vivek Kumar
Computer Science and Engineering
IIIT Delhi
vivekk@iiitd.ac.in

About me

- https://vivkumar.github.io/
- https://hipec.github.io/



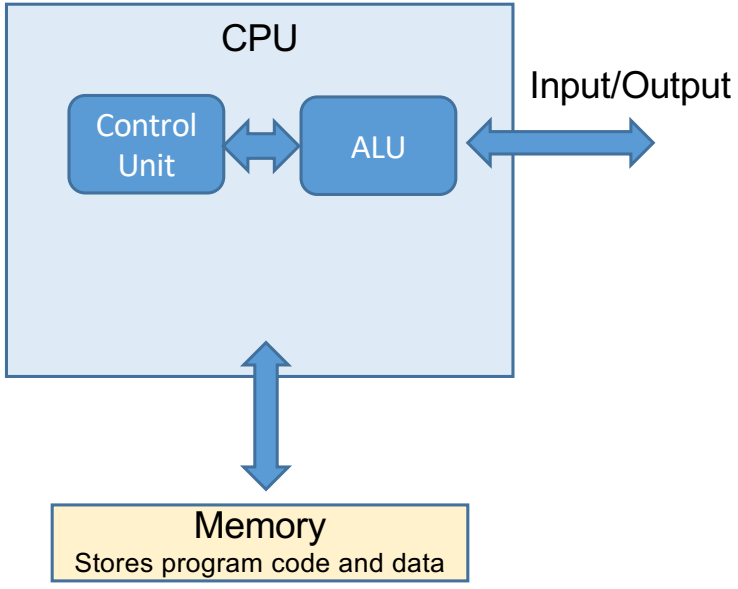
I spend my time at the HiPeC Lab or roaming in the majestic Himalayas..

Today's Lecture

- Modern computing architecture
- High-level overview of operating systems
- Roles of an OS
- Challenges in modern OS
- Course evaluation and logistics

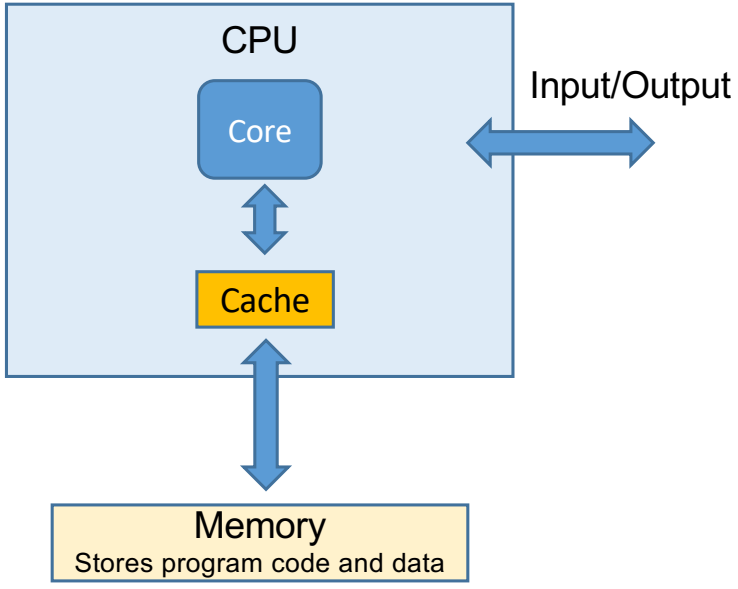
Let us first quickly try to understand the modern computer architecture

Von Neumann Architecture & Associated Issues (1/3)



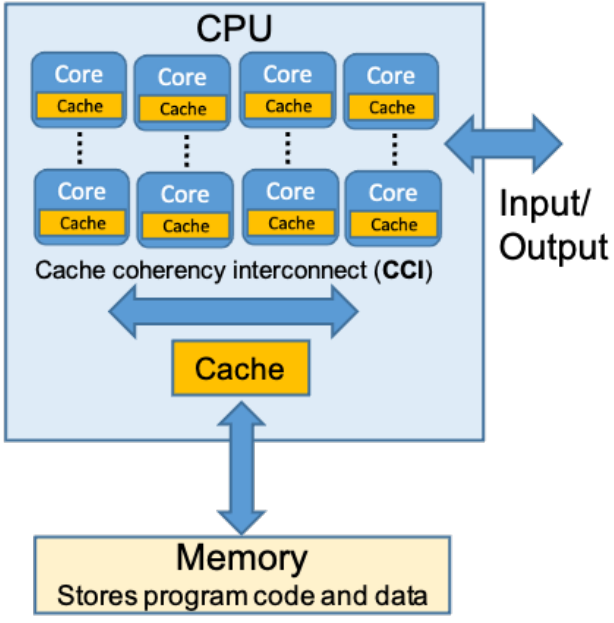
- John Von Neuman in 1945 came up with the architecture for computers that we even use today (albeit with several changes)

Von Neumann Architecture & Associated Issues (2/3)



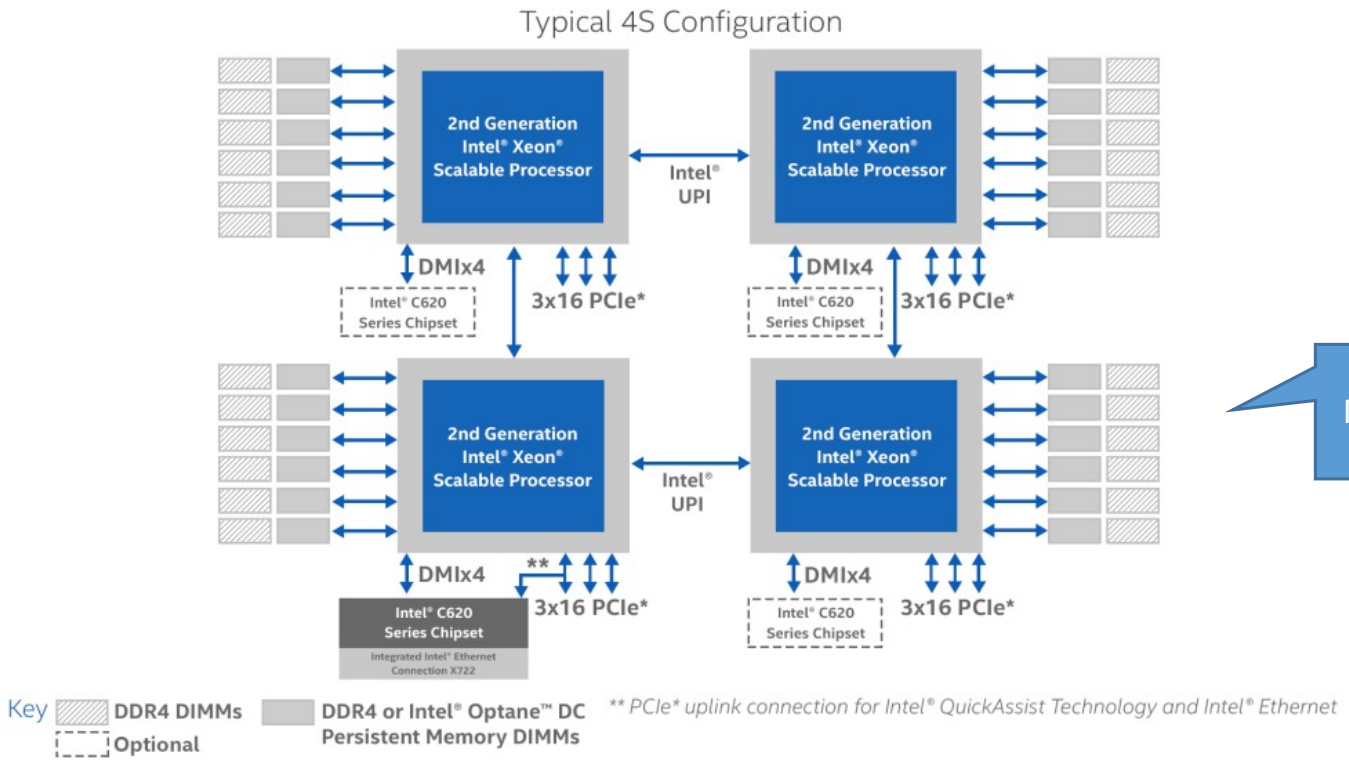
- Memory bottleneck
  - Problem
    - Access latency to memory quite high
      - High CPU stalls while fetching code and data
  - Solution
    - Add cache on the CPU chip to store frequently accessed memory

Von Neumann Architecture & Associated Issues (3/3)



- Performance bottleneck
  - Problem
    - Design issues with increasing the performance of single core processor
      - High heat dissipation
        - Even capable of melting the processor!
      - High power consumption
  - Solution (around 2004)
    - Add more cores to achieve better performance instead of increasing the performance of a single core
      - Still maintains the Moore's law
    - Add cache coherency interconnect (CCI) to fetch data on one core from the other core's cache instead of going all the way up to main memory

Latest Server Processors

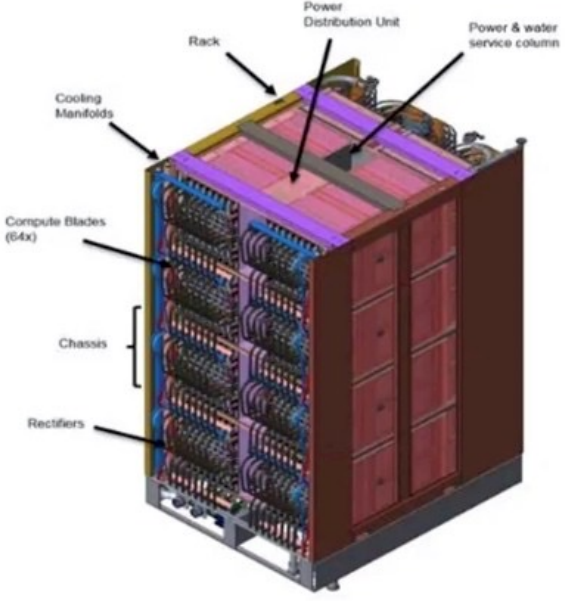


Non-Uniform Memory Access Architecture

Latest Supercomputer (May 2022)

System
- 2 EF Peak DP FLOPS
- 74 compute racks
- 29 MW Power Consumption
- 9,408 nodes
- 9.2 PB memory
  (4.6 PB HBM, 4.6 PB DDR4)
- Cray Slingshot network with dragonfly topology
- 37 PB Node Local Storage
- 716 PB Center-wide storage
- 4000 ft^2 foot print

Olympus rack
- 128 AMD nodes
- 8,000 lbs
- Supports 400 KW



AMD node
- 1 AMD "Trento" CPU
- 4 AMD MI250X GPUs
- 512 GiB DDR4 memory on CPU
- 512 GiB HBM2e total per node
  (128 GiB HBM per GPU)
- Coherent memory across the node
- 4 TB NVM
- GPUs & CPU fully connected with AMD Infinity Fabric
- 4 Cassini NICs, 100 GB/s network BW

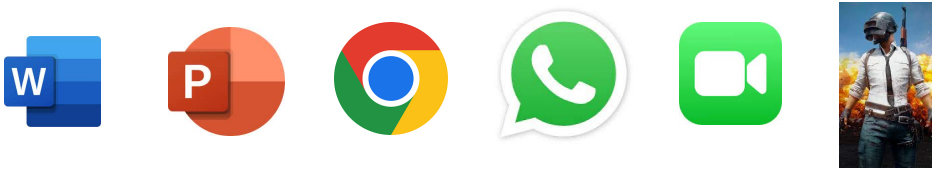
Compute blade
- 2 AMD nodes

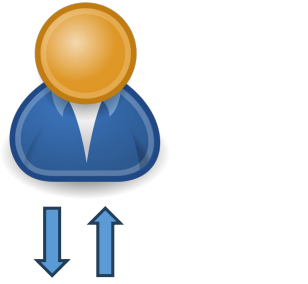
Source: https://www.nextplatform.com/wp-content/uploads/2022/03/oak-ridge-al-geist-frontier-specs.jpg

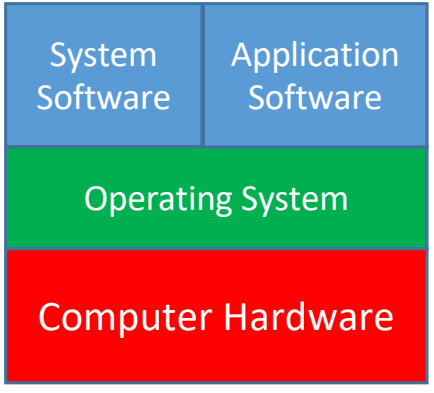
Today's Lecture

- Modern computing architecture
- High-level overview of operating systems
- Roles of an OS
- Challenges in modern OS
- Course evaluation and logistics

What is an Operating System?







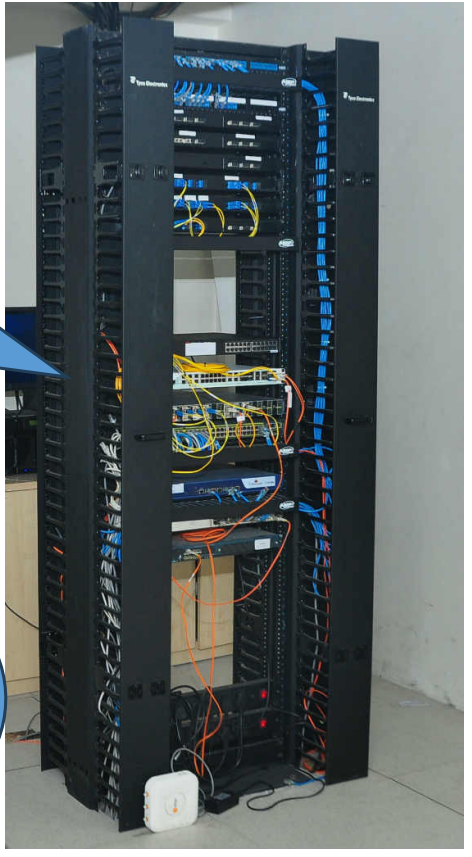
- OS is a piece of software whose job is the manage the computer's resources for its users and applications

Analogy: The Hardware

Analogy: Disk & DRAM

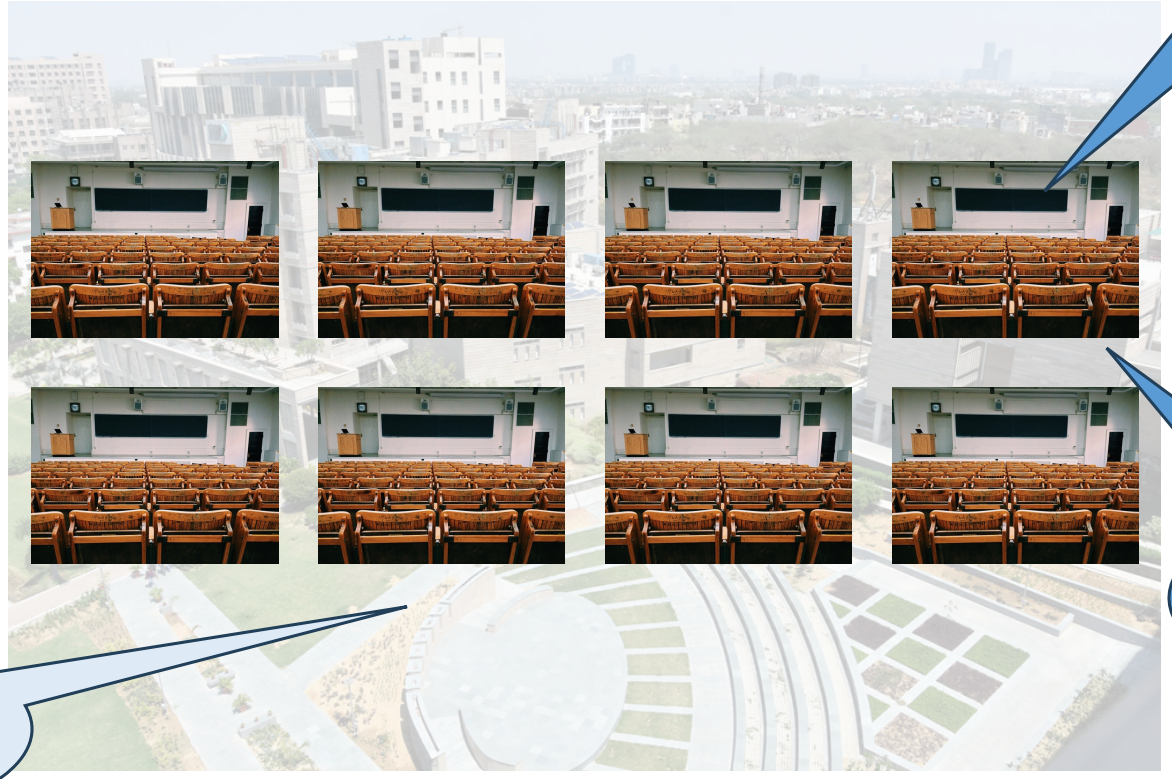
Library – persistent storage of data (Disk)

Data is read from the Disk (**Library**) and stored in central server (**DRAM**) in the form of **lecture slides**



Time required to fetch the teaching data from the library v/s central server during lectures ????

Analogy: The Multicore CPU



Board (Cache)

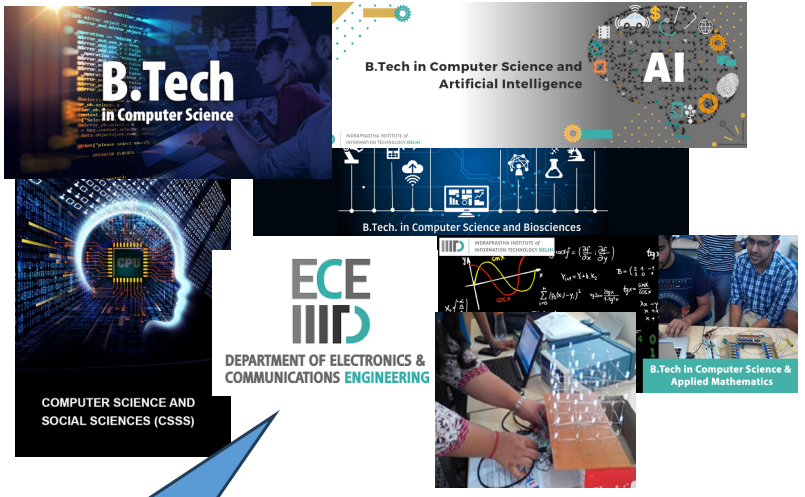
Classroom (Core)

Fixed number of Cores

Analogy: The Complete Hardware

Analogy: Operating System & Applications

OS: Faculties and Staff



Applications: Several BTech/MTech programs

Each **application** has several **Tasks** to complete (courses)

Major Goals of Operating Systems

1. Virtualization
2. Concurrency
3. Persistence
4. Protection and security

Analogy: Virtualization (CPU cores)

Scheduling: Time table is prepared for each semester for smooth running of each program

- The operating system gives an "**illusion**" to each application that the entire hardware is only for that application
  - So many applications can execute on the same hardware with limited resources (fixed number of cores, DRAM, and disk)

Analogy: Virtualization (CPU cores)

Every program has many courses, each with several lectures

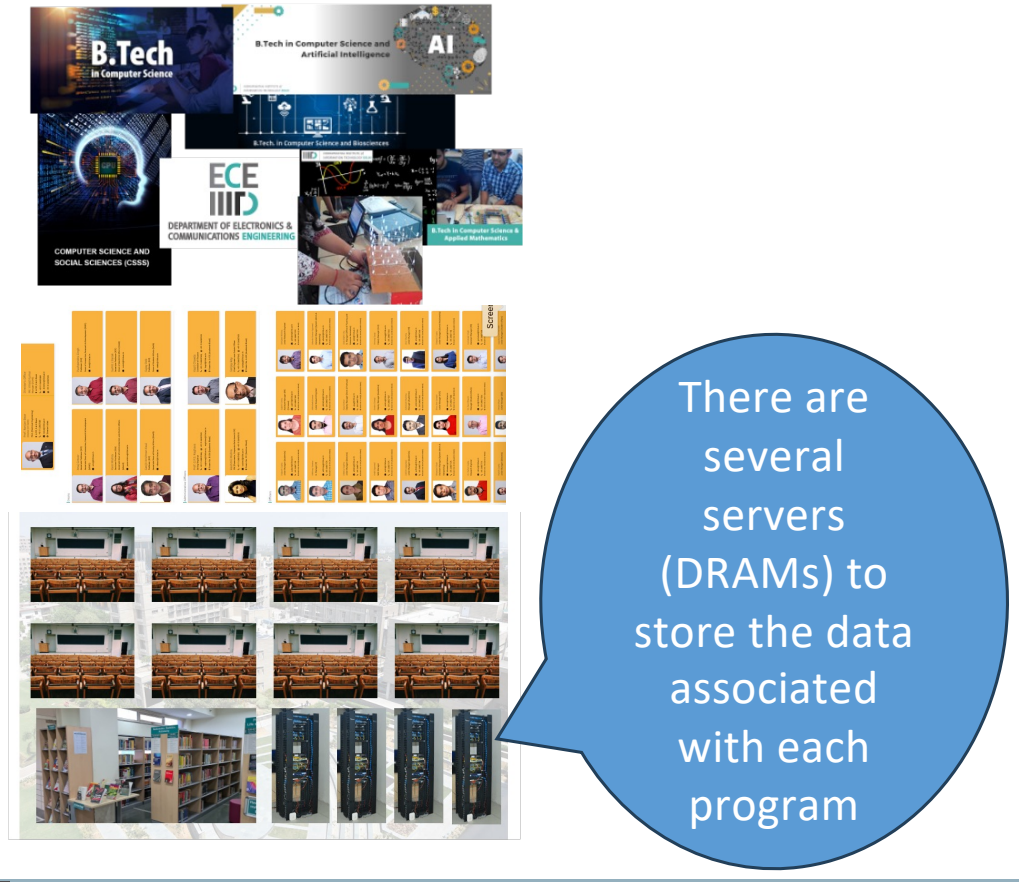
- Each application (**Process**) has some code and data associated with it (e.g., course and lecture materials)
  - Faculty prepares the lecture by fetching materials from library (**Disk**) and then preparing lecture material/slides saved on servers (**DRAM**)
  - Lectures are **Scheduled** in a classroom where the faculty uses the whiteboard (**Cache**) to teach the topic

Analogy: Virtualization (CPU cores)

Several lectures are taught using a fixed number of classrooms

- Only one **Task** can run on a CPU core at any given time
  1. Same classroom is used to run lectures from different courses
  2. During the lecture, students write down notes from the board (**cache**) onto their notebook (persistent storage)
  3. Next faculty enters the classroom for delivering a different lecture
  4. First, a recap of the topics discussed in the last lecture is carried out before starting the lecture
     - Reloading the context
- Above steps are a very high-level overview of **context switch between processes for time sharing of a CPU core**!

Analogy: Virtualization (Memory)



- Faculties and admin (OS) stores the memory of each program (courses, lectures, documents, etc.) in the servers (DRAMs)
- They gives an illusion that they have a dedicated memory space for each program
  - In the reality each program's data could be stored in pieces over several DRAMs (physical address)
  - However, this complexity is abstracted away by the O.S.

Analogy: Concurrency

Scheduling algorithm of IIIT-Delhi

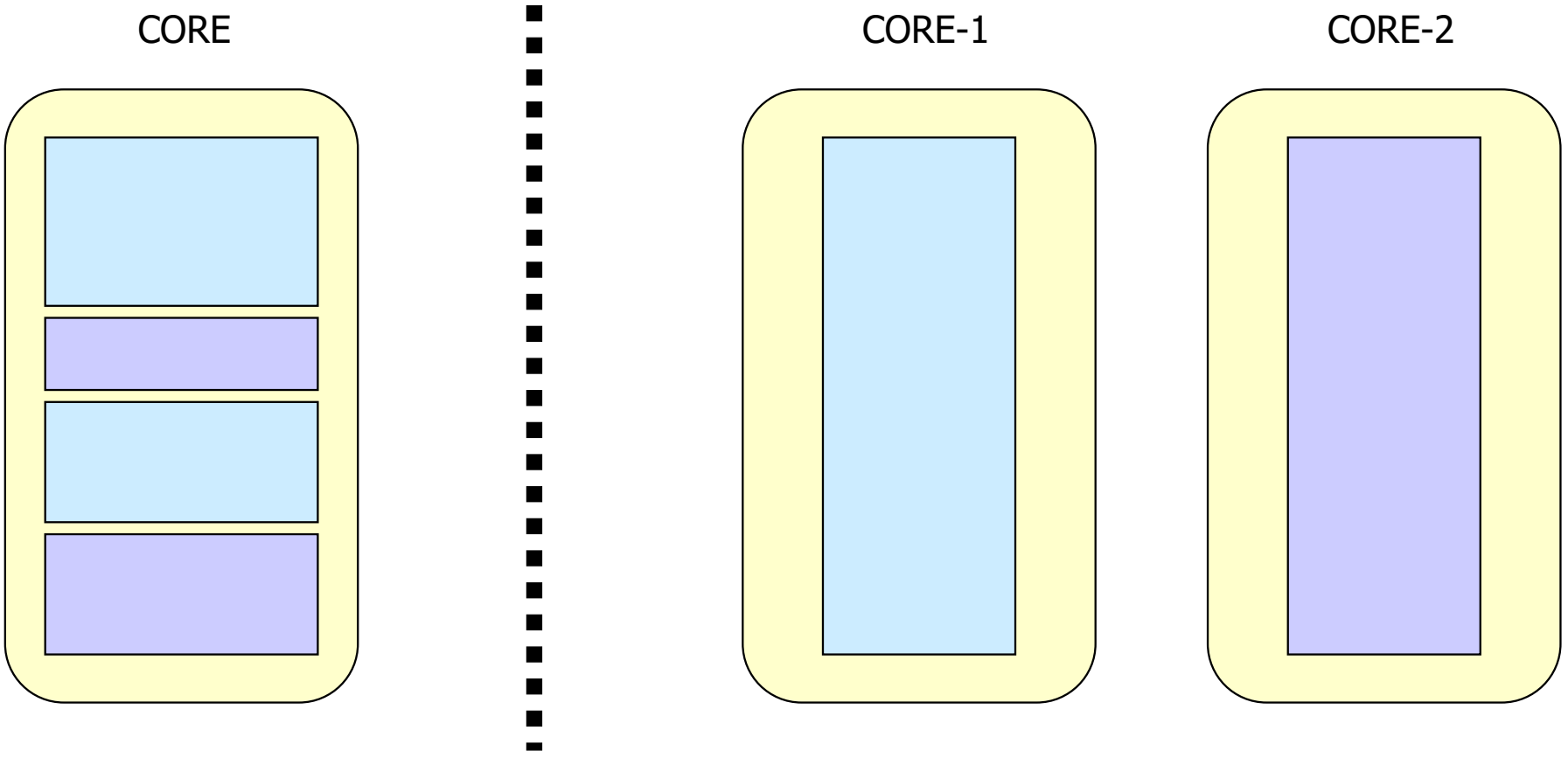
B.Tech. 3rd & 4th Year, M.Tech. 1st & 2nd year, Ph.D. - Monsoon Semester AY 2022-23

Day	8.30-9.00	9-9:30	9:30-10	10-10:30	10:30-11	11-11:30	11:30-12	12-12:30	12:30-1	1-1:30	1:30-2	2-2:30	2:30-3	3-3:30
Mon			CN (Sec A) CN (Sec B) BML WSI IDUDA NDM MLBA CMSMR MAD RA-II	C21 C102 C03 C13 C208 C22 A006 C209 C214 C215	Slot 4	AC DMG QMD ETB ME IAGA DFML BioP ADL	C03 C21 C216 C22 A007 C208 C209 C214 A006	Slot 2						DIP
Tue			CoMeG ML (A) ML (B) LST QM TML AERM DAVP WN NDMA	C03 C11 C21 C13 C22 C208 C215 C24 C213 C210	Slot 1	FOE CGAS IIA SDOS DSc CMOS SC FF ADC WARDI G&M	C02 A006 C213 A106 C210 C211 A007 C21 C12 C208 C209	Slot 5			CGS			TFW PRP CG
			CN (Sec A) CN (Sec B)	C21 C102	Slot 4	CA Chi	A006 C13	Slot 9						DIP GA

B.Tech. CSE program has several courses, some of which could be scheduled together in the same slot, but in different **classrooms**!

- A program could have several independent tasks (set of instructions) that could executed simultaneously over multicore CPUs
  - Multiple threads of execution
- However, some course has prerequisite courses that the students should complete first
  - Dependencies in multi-threaded execution

Concurrency vs. Parallelism



Analogy: Persistence

- At the end of each semester, academic department collect and permanently save the lecture slides, quiz sheets, assignments and project, and student's answer sheets for mid/end semester exams (for each courses)

Analogy: Protection (Internal) & Security (External)

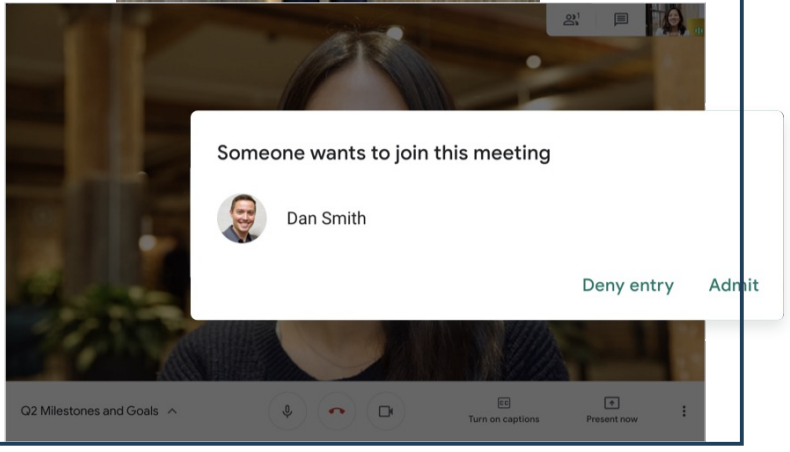
≠

ECE

ARTMENT OF ELECTRONICS & COMMUNICATIONS ENGINEERING

Course Name	Course Acronym	Course Code
Computer Architecture	CA	CSE511/ECE511

Each program can only access its own data (courses) unless there is some authorization obtained by the O.S



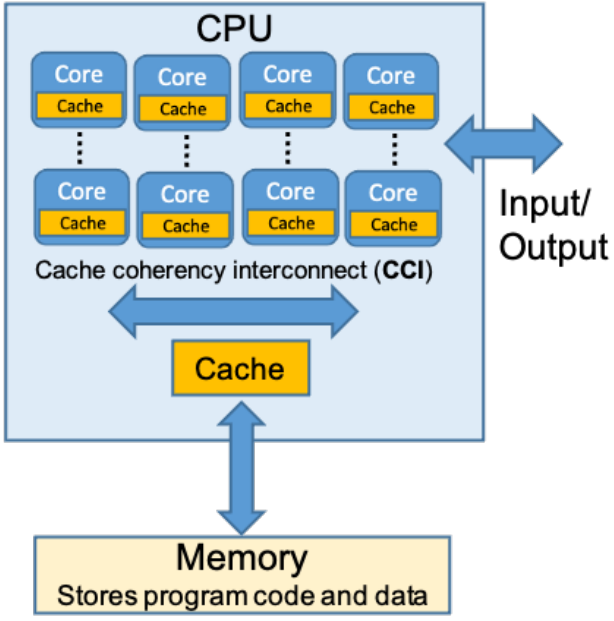
Today's Lecture

- Modern computing architecture
- High-level overview of operating systems
- Roles of an OS
- Challenges in modern OS
- Course evaluation and logistics

Roles of an Operating System

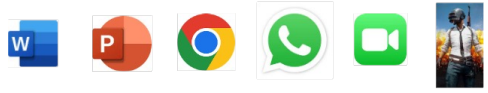
- Referee
  - Manages all the shared resources in the computer
- Illusionist
  - Each application thinks that the entire computer belong to itself
- Glue
  - Offers standard services to simplify application development and facilitate sharing

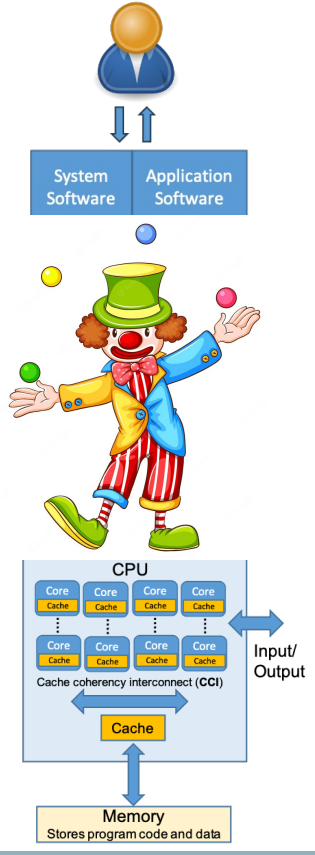
Roles of an Operating System (Example-1)



- Let us try to understand it from the perspective of a desktop/laptop

Roles of an Operating System (Example-1)



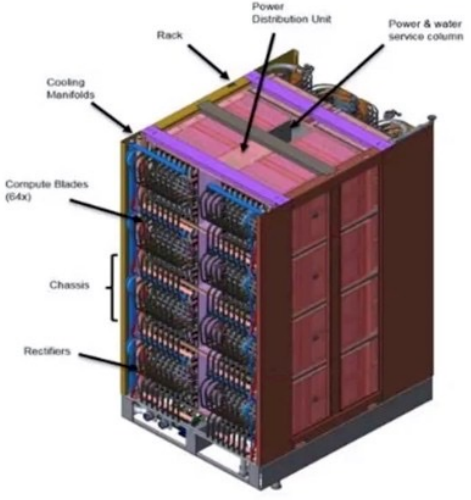


- **Referee**
  - Each application will be launching several tasks
    - Gaming app will require graphic rendering, network play support, taking player input via keyboard/mouse, etc.
    - Zoom will require tasks such as network connection, access to mike/camera, graphic rendering, etc.
  - As there are several cores, each task can run on a different core
    - But, there are only finite number of resources (cores, caches, input/output devices, memory etc.)
- **Illusionist**
  - Allows the application to have hundreds of tasks (several times more than available cores), infinite amount of memory (DRAMs are only in few GBs), etc.
- **Glue**
  - Standard APIs for network connection, accessing input/output devices, etc.

Roles of an Operating System (Example-2)

System
- 2 EF Peak DP FLOPS
- 74 compute racks
- 29 MW Power Consumption
- 9,408 nodes
- 9.2 PB memory (4.6 PB HBM, 4.6 PB DDR4)
- Cray Slingshot network with dragonfly topology
- 37 PB Node Local Storage
- 716 PB Center-wide storage
- 4000 ft^2 foot print

Olympus rack
- 128 AMD nodes
- 8,000 lbs
- Supports 400 KW



AMD node
- 1 AMD "Trento" CPU
- 4 AMD MI250X GPUs
- 512 GiB DDR4 memory on CPU
- 512 GiB HBM2e total per node (128 GiB HBM per GPU)
- Coherent memory across the node
- 4 TB NVM
- GPUs & CPU fully connected with AMD Infinity Fabric
- 4 Cassini NICs, 100 GB/s network BW

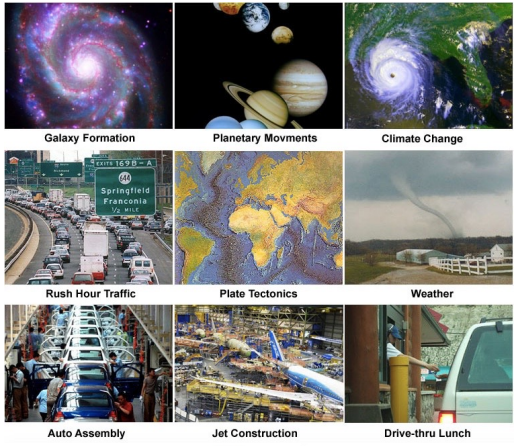
Compute blade
- 2 AMD nodes

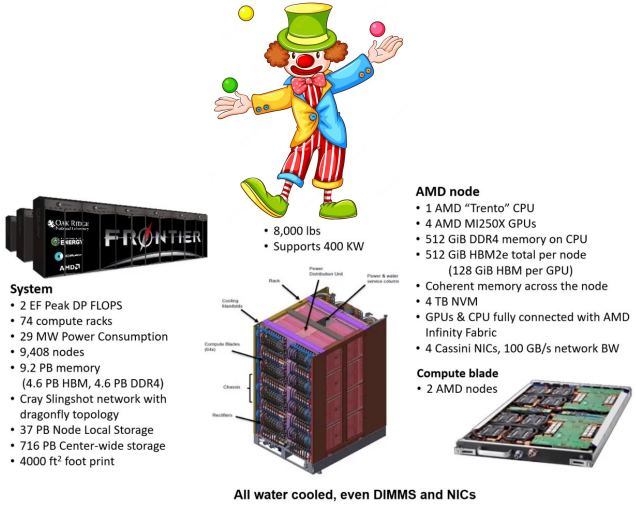
All water cooled, even DIMMS and NICs

- Let us now try to understand it from the perspective of a **supercomputer**

Picture source: https://www.nextplatform.com/wp-content/uploads/2022/03/oak-ridge-al-geist-frontier-specs.jpg

Roles of an Operating System (Example-2)



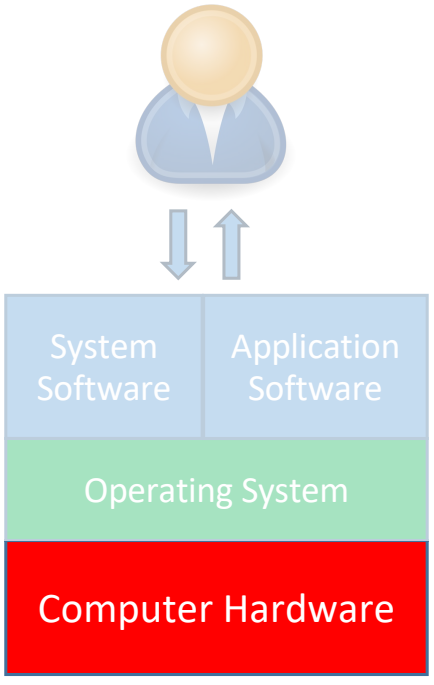


- **Referee**
  - Each application will be launching several tasks
    - Either computation, communication, or file handling (no camera, mic, etc.)
    - Application can wait in queue if it is requesting more cores than currently available
- **Illusionist**
  - Allows the application to interact with the supercomputer via a single computer node
    - Several computing nodes, each with its own OS
    - Application tasks distributed across all the nodes
    - Adapts to node failures (migrate tasks to another node)
- **Glue**
  - APIs for communication and accessing disjoint data (at different nodes) with low latency (e.g., Message Passing Interface APIs)

Today's Lecture

- Modern computing architecture
- High-level overview of operating systems
- Roles of an OS
- Challenges in modern OS
- Course evaluation and logistics

Operating System Specific to Hardware

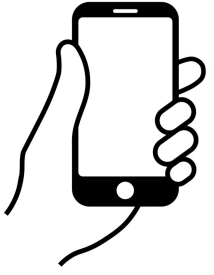


- Throughput oriented (manages several tasks)
- E.g., Windows, Linux, MacOS

Laptop/Desktop Operating System

- Lightweight
- Highly user friendly (GUI)
- Smaller memory footprint (more than RTOS)
- E.g., Andriod, iOS

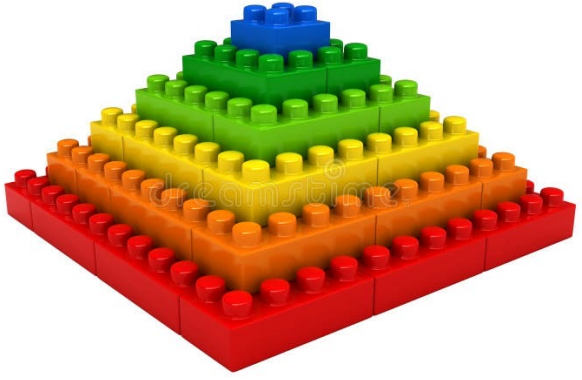
Mobile Operating System



- Time bound and limited tasks
- May work with small memory
- E.g., Nucleus RTOS

Real time Operating System

Operating System Design



Convenient to use?

Easily portable?

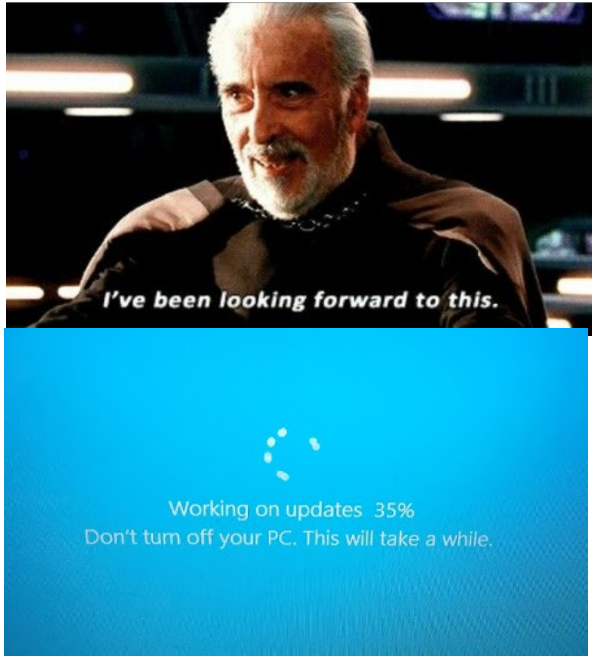
Reliable? Fast?

Debugging?

Major Challenges in Modern OS

- Dependability

Me: *restarts computer because it's lagging*
Windows Update:



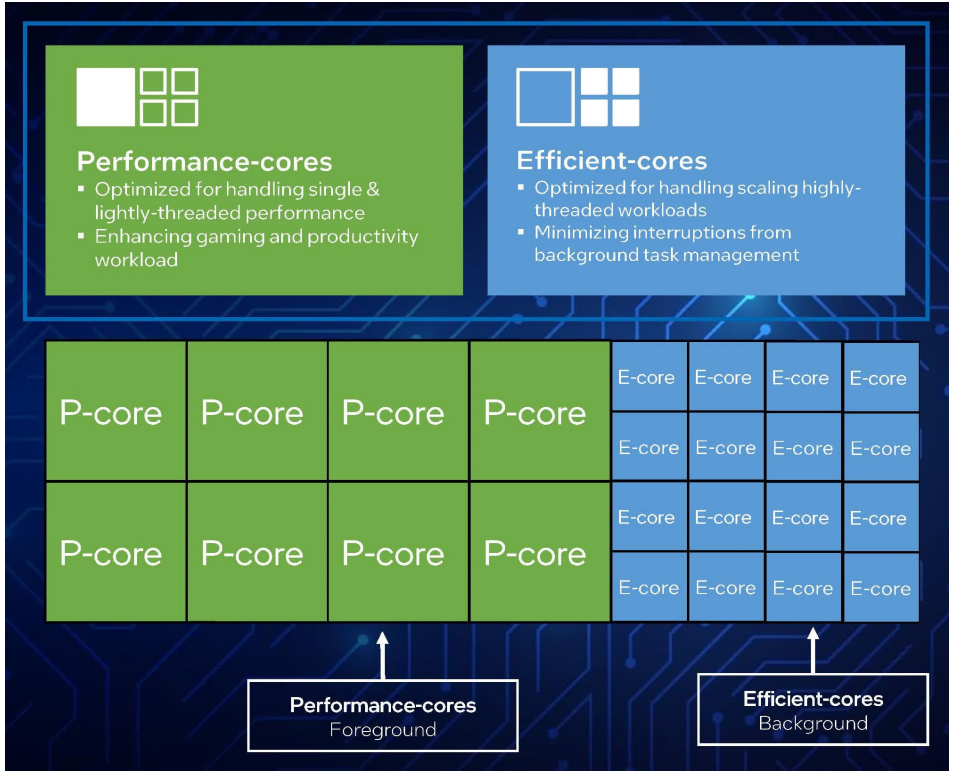
Predictable? Reliable?
(frequent bug fixes, updates…)

V/S

Predictable?
Reliable?

Major Challenges in Modern OS

- Performance and Energy

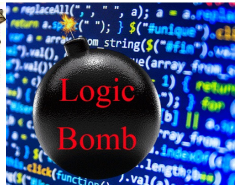


Which tasks should run on P-core and which ones on E-cores?

What should be the processor core frequency while running a particular task? High frequency can give better performance, but requires more energy, and vice-versa

Images source: https://www.techspot.com/news/94919-preview-core-i9-13900-engineering-sample-performance-looks.html

Major Challenges in Modern OS



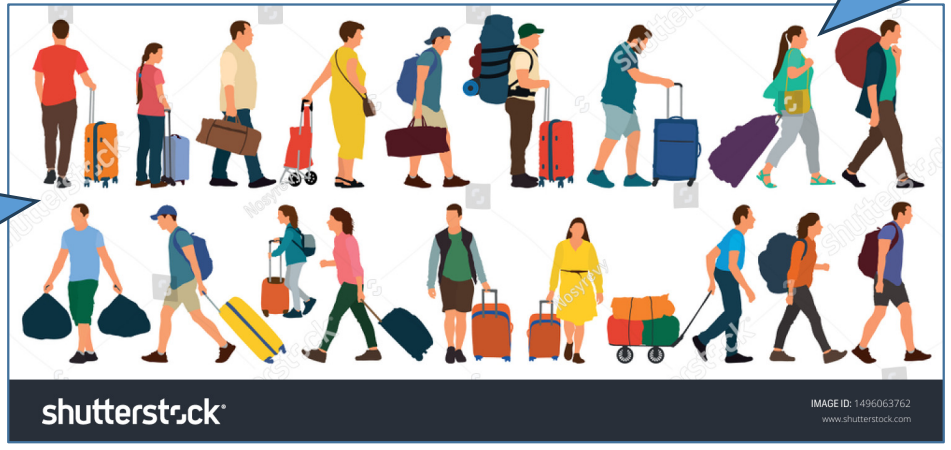
- Protection and Security

They are lacking **Security**

Let's unleash our arsenal

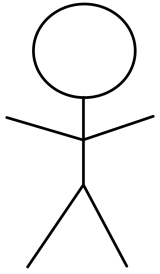
We are **Protected**

Each application can only access its own **baggage** (files, memory, CPU). They should not access another application's **baggage** unless they have proper authorization



Major Challenges in Modern OS

- Portability



I am quite **Portable**. I just need to change my shoes

I am quite **Portable too!** I just need to change my machine interfacing code in kernel

Today's Lecture

- Modern computing architecture
- High-level overview of operating systems
- Roles of an OS
- Challenges in modern OS
- Course evaluation and logistics

Course Evaluation (Section-A)

- Quiz: 10% (N-1 policy)
  - No surprise quizzes
  - Will be held during lecture hours (around 20mins duration)
  - **Both sections will have different quiz schedule/questions**
- Semester exams (same for both sections)
  - Mid-sem: 20%
  - End-sem: 30%
- Take home assignments 40%
  - **NO (N-1 policy) !!!**
  - **Differ across both sections**
  - **There is some surprise for you…!**

I might have a different lecture plan/content than you see currently on Techtree. However, the COs would remain the same

Important Information

1. Please **Don't** open-source assignment implementations after the course gets over
2. We will **Not** upload mid/end semester solution/rubric on Google Classroom
   - Although, we will discuss it in class
3. We will be using biometric machines for in-class attendance
   - Lecture recordings will not be provided

Course Prerequisites

- **C programming and debugging skills are must!**
  - You must have used C in your DSA course and in OS refresher module
  - If you don't know C then better start practicing so as you don't face issues with the course assignments
    - First one will be released early next week!

We will strictly follow the IIITD **plagiarism policy**. No excuses if you get caught in plagiarism

Course Logistics

ad7lxqu

CSE231: Operating System A Copy invite link

- Only Section-A students should register to above classroom
- Machines for assignments
  - You can use your laptop
    - Would require Linux OS (or any VM running Linux OS)
  - In case you have problems please feel free to first shoot an email to TF (Madiha Tariq), and then to me if your issues isn't resolved
- Course textbook is Operating Systems Three Easy Pieces (OSTEP)

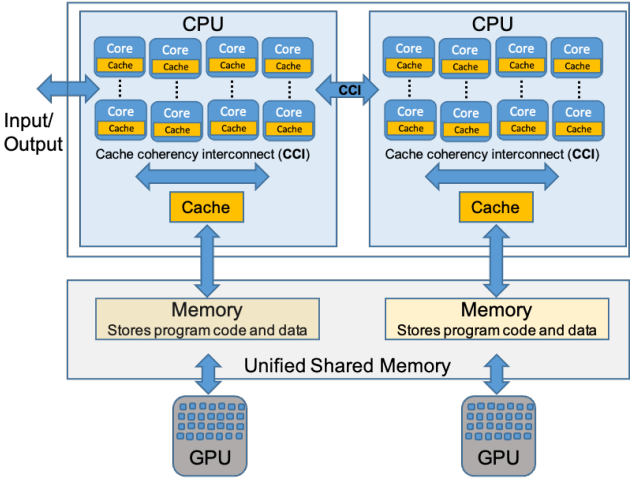
Next Lecture

- Source code to machine code

Lecture 02: Source Code to Machine Code

Vivek Kumar
Computer Science and Engineering
IIIT Delhi
vivekk@iiitd.ac.in

Last Lecture



Hardware: CPU (Classroom) + Cache (Whiteboard) + Disk (Library) + DRAM (Servers)

OS
(All employees)

System Software (Time table)

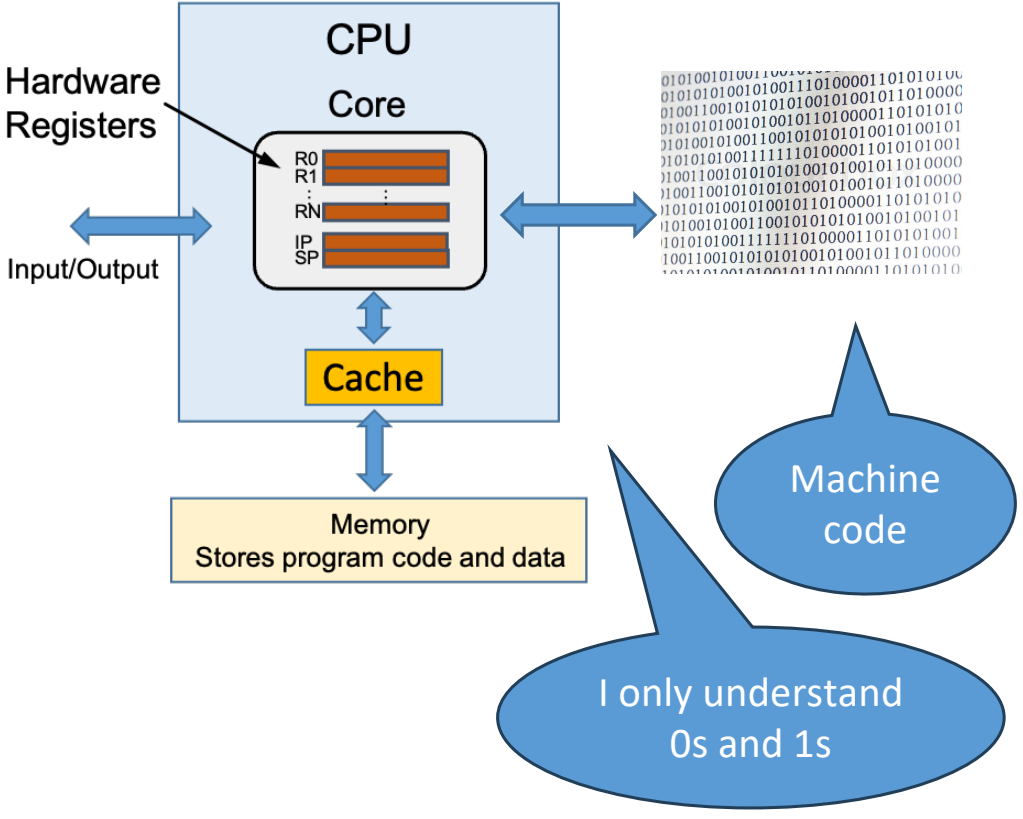
Application Software
(BTech programs)

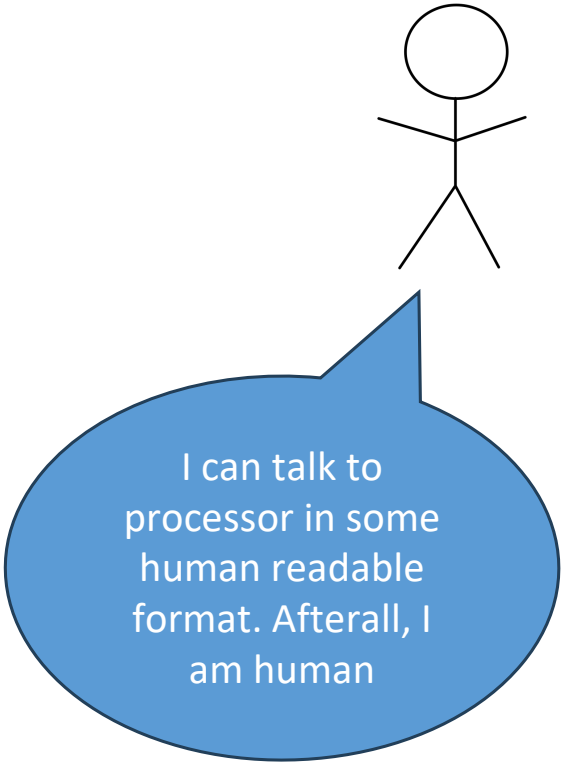
- Modern computer architecture are complex
- Major goals of OS are virtualization, concurrency, persistence, protection, and security
- OS plays the role of an illusionist, referee, and glue
- Challenges in modern OS are dependability, performance/energy, protection/security, and portability
- Course evaluation scheme

Today's Class

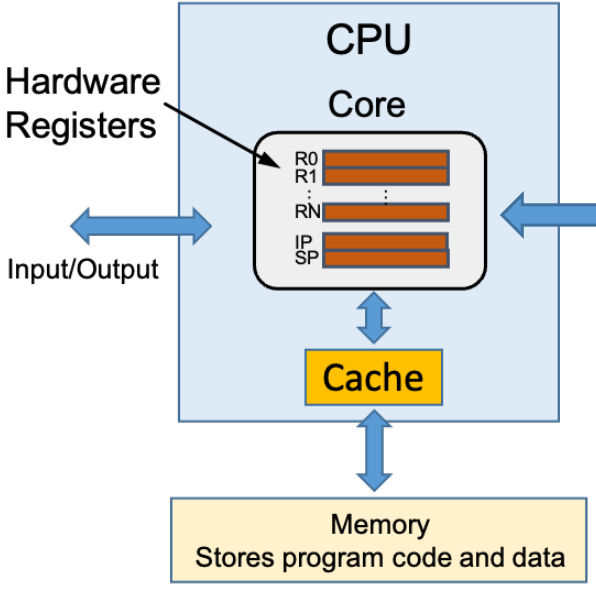
- Compilation steps
- Linking
- Introduction to Executable File Format (ELF)

Hardware and Software

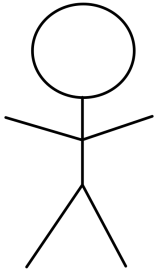




Hardware and Software



```
movl    8(%ebp), %eax
subl    $1, %eax
subl    $12, %esp
pushl   %eax
call    fib
addl    $16, %esp
movl    %eax, -16(%ebp)
movl    8(%ebp), %eax
subl    $2, %eax
subl    $12, %esp
pushl   %eax
call    fib
addl    $16, %esp
movl    %eax, -12(%ebp)
movl    -16(%ebp), %edx
movl    -12(%ebp), %eax
addl    %edx, %eax
```



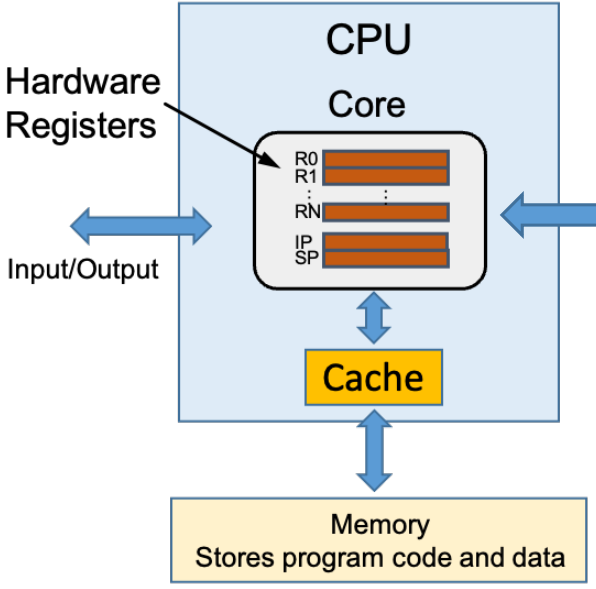
I only understand 0s and 1s

Machine code

Assembly code

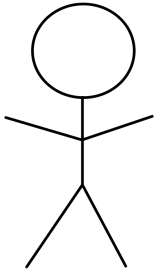
It's better.. but Afterall, I am human

Hardware and Software



```
movl    8(%ebp), %eax
subl    $1, %eax
subl    $12, %esp
pushl   %eax
call    fib
addl    $16, %esp
movl    %eax, -16(%ebp)
movl    8(%ebp), %eax
subl    $2, %eax
subl    $12, %esp
pushl   %eax
call    fib
addl    $16, %esp
movl    %eax, -12(%ebp)
movl    -16(%ebp), %edx
movl    -12(%ebp), %eax
addl    %edx, %eax
```

```
uint64_t fib(uint64_t n) {
  if (n < 2) {
    return n;
  } else {
    uint64_t x = fib(n-1);
    uint64_t y = fib(n-2);
    return (x + y);
  }
}
```



C-program

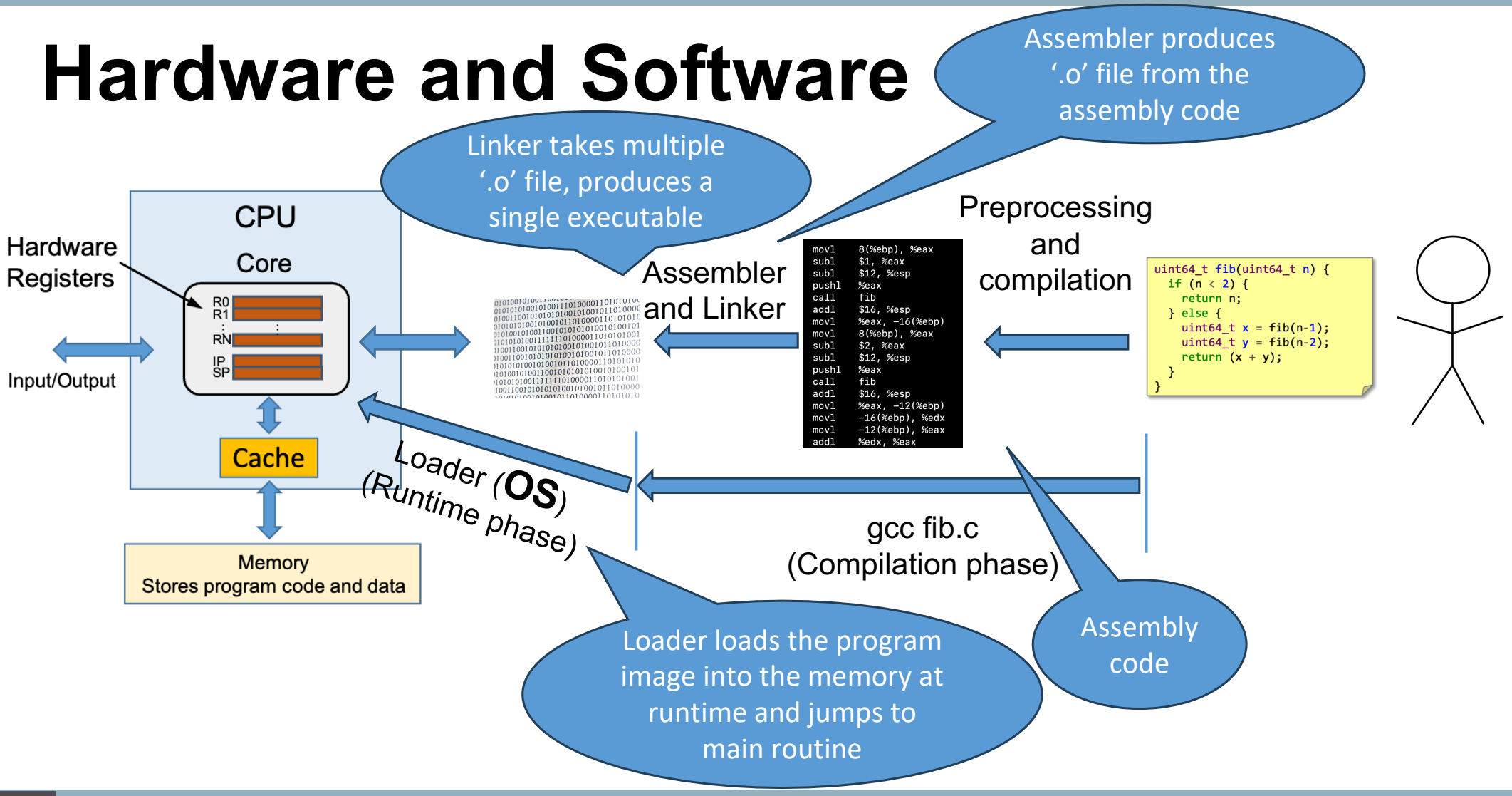
Machine code

Assembly code

I only understand 0s and 1s

Wonderful!

Hardware and Software

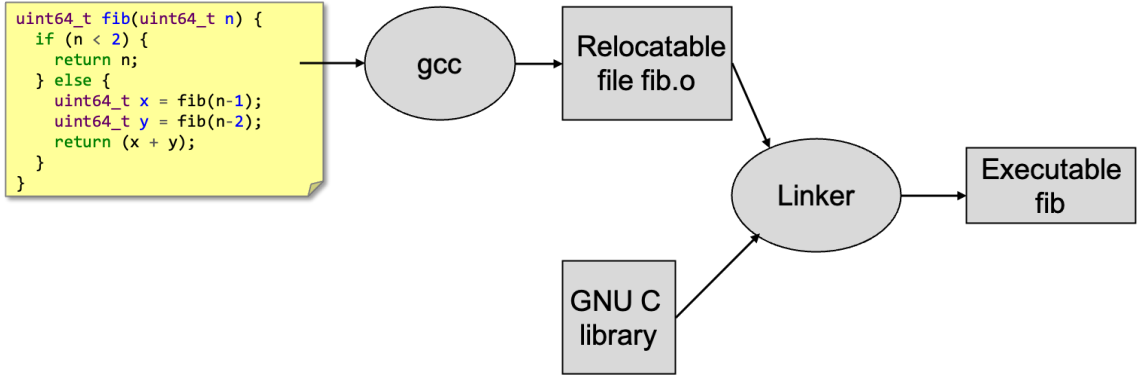


Today's Class

- Compilation steps
- Linking
- Introduction to Executable File Format (ELF)

Linking

```
$ gcc -o fib fib.c     // compiling + linking
$ file fib
...ELF 64-bit shared object...dynamically linked
$ gcc -c fib.c         // compiling
$ file fib.o           // relocatable file
fib.o
$ gcc -o fib fib.o     // linking
```

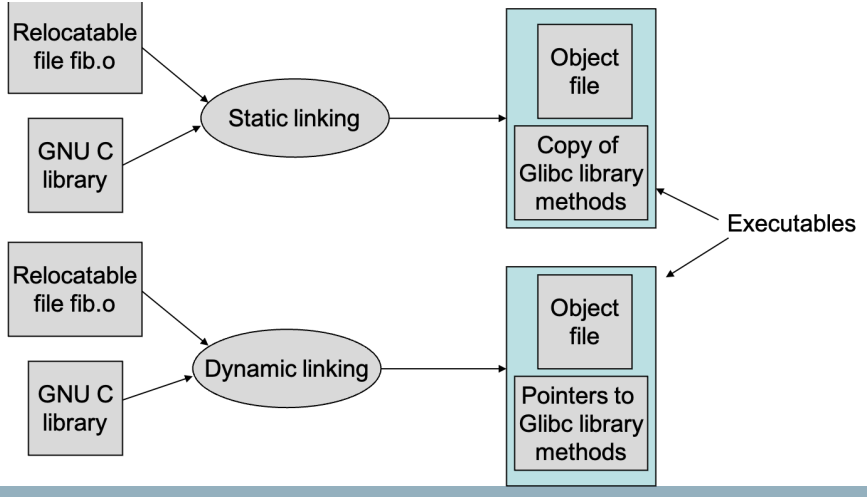


- Combines several relocatable files (.o) into a single executable
- Enable separate compilation of files
  - Changes in one file does not affect other files
    - Recompilation process is small

Static and Dynamic Linking

```
$ gcc -static -o fib fib.c
$ file fib
...ELF 64-bit executable...statically linked
```

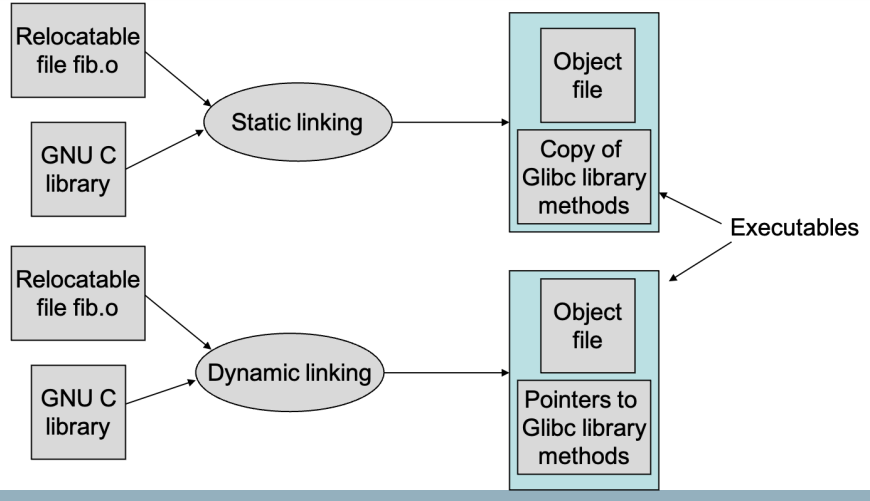
```
$ gcc -o fib fib.c
```



- Static linking
  - Each and every library modules referenced in the relocatable file is copied into the final executable
    - Static binding at compile time
- Dynamic linking
  - Final executable only contains references (pointers) to the library method instead of the copy of a library method
    - Binding with library done at runtime during execution

Static and Dynamic Linking

```
$ gcc -static -o fib fib.c
$ file fib
...ELF 64-bit executable...statically linked
$ ls -l fib
-rwxrwxr-x 1 vivek vivek 845304 Aug  6 10:26 fib
$ gcc -o fib fib.c
$ ls -l fib
-rwxrwxr-x 1 vivek vivek 8328 Aug  6 10:27 fib
```



- Static linking
  - Each and every library modules referenced in the relocatable file is copied into the final executable
    - Static binding at compile time
- Dynamic linking
  - Final executable only contains references (pointers) to the library method instead of the copy of a library method
    - Binding with library done at runtime during execution
- Which is better?

Dynamic Linking

- Advantages
  - Modular programming
    - If library requires changes/recompilation, the executables referring it does not need recompilation
  - Saves disk space
    - Executable just contain references
    - Multiple executables can access the same **.so** at runtime
- Disadvantages
  - May impact performance
    - Loading shared libraries at runtime and “glue” the references
    - What if the shared library changes or is missing at runtime?
    - *Reduction in locality due to NUMA (we will discuss it in future lectures)*

Today's Class

- Compilation steps
- Linking
- Introduction to Executable File Format (ELF)

Main Types of Object Files in Linux

- Relocatable
- Executable
- Shared object

Relocatable Object Files

- Output of assembler
- Symbols (functions and variables) are not bound to any specific address
  - Happens after linking
- Contains code and data suitable for linking with other object files to create the final executable or shared object

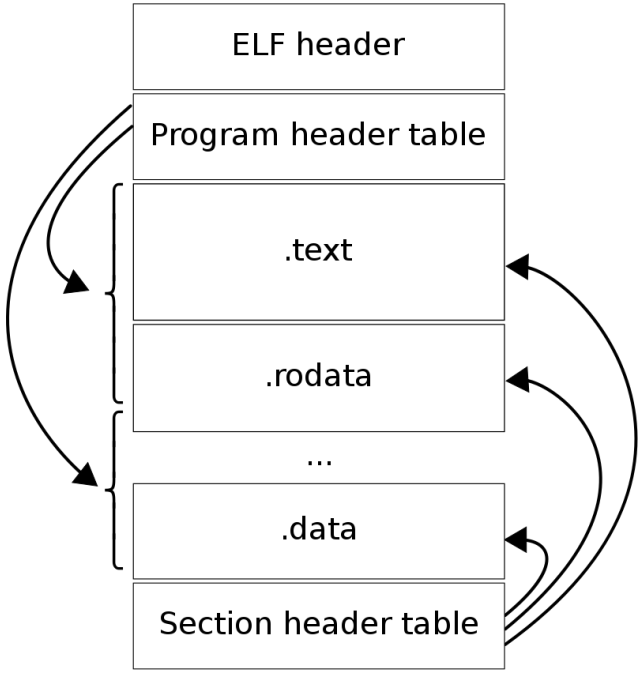
Executable Object Files

- Output of linker
- It is similar to the relocatable file, but all symbols (functions and variables) have addresses resolved
- Contains all the information for the OS loader to run the program

Executable and Linkable Format (ELF)

- Standard **format** for binary files on Linux based OS
  - Contains all the information required by the OS to run the program
  - Platform independent format
  - A standard format eases the process of dynamic linking, loading and runtime control on a program
- Microsoft Windows OS uses Portable Executable (PE)
- IBM AIX OS uses Extended Common Object File Format (X-COFF)

Executable and Linkable Format (ELF)



Note: For simplicity, we will only discuss ELF format for 32-bit object files

Image source: https://en.wikipedia.org/wiki/Executable_and_Linkable_Format

- ELF header
  - Provides a roadmap for the entire file organization
    - Always at offset zero of the object file
    - Provides entry point address for execution
- Two views of an ELF file
  - Linkable view (relocatable file)
  - Execution view (executable file, shared object)
- Linkable view
  - Section header table
    - One section header for each section
  - Sections
    - Contains data required for linking
    - Machine code, global variables, (initialized and un-initialized), symbol tables, line mapping between machine code and original C code, etc.
- Execution (or Loader) view
  - Program header table
    - One program header for each segment
  - Segments
    - Created by merging several sections
    - Contains information required for by the loader for execution
- Contiguous chunk of memory (ELF header, PHT, SHT, each section, each segment)

ELF Header

```
#define EI_NIDENT 16

typedef struct {
        unsigned char   e_ident[EI_NIDENT];
        Elf32_Half      e_type;
        Elf32_Half      e_machine;
        Elf32_Word      e_version;
        Elf32_Addr      e_entry;
        Elf32_Off       e_phoff;
        Elf32_Off       e_shoff;
        Elf32_Word      e_flags;
        Elf32_Half      e_ehsize;
        Elf32_Half      e_phentsize;
        Elf32_Half      e_phnum;
        Elf32_Half      e_shentsize;
        Elf32_Half      e_shnum;
        Elf32_Half      e_shstrndx;
} Elf32_Ehdr;
```

Identification for object file

Object file type

Entry point address for starting the executable

Program Header Table (PHT) file offset

Section Header Table (SHT) file offset

ELF header size in bytes (fixed)

Size in bytes of each entry in PHT

Total number of entries in PHT

Size in bytes of each entry in SHT

Total number of entries in SHT

Index of the section called as "String Table"

Image source: https://refspecs.linuxfoundation.org/elf/gabi4+/ch4.eheader.html

Entry Point?

```
vivek@possum:~/elf_os-l2$ readelf -h ./a.out
ELF Header:
  Magic:   7f 45 4c 46 01 01 01 00 00 00 00 00 00 00 00 00
  Class:                             ELF32
  Data:                              2's complement, little endian
  Version:                           1 (current)
  OS/ABI:                            UNIX - System V
  ABI Version:                       0
  Type:                              DYN (Shared object file)
  Machine:                           Intel 80386
  Version:                           0x1
  Entry point address:               0x3e0
  Start of program headers:          52 (bytes into file)
  Start of section headers:          6060 (bytes into file)
  Flags:                             0x0
  Size of this header:               52 (bytes)
  Size of program headers:           32 (bytes)
  Number of program headers:         9
  Size of section headers:           40 (bytes)
  Number of section headers:         29
  Section header string table index: 28
```

Entry point address

Entry Point?

```
vivek@possum:~/elf_os-l2$ readelf -h ./a.out
ELF Header:
  Magic:   7f 45 4c 46 01 01 01 00 00 00 00 00 00 00 00 00
  Class:                             ELF32
  Data:                              2's complement, little endian
  Version:                           1 (current)
  OS/ABI:                            UNIX - System V
  ABI Version:                       0
  Type:                              DYN (Shared object file)
  Machine:                           Intel 80386
  Version:                           0x1
  Entry point address:               0x3e0
  Start of program headers:          52 (bytes into file)
  Start of section headers:          6060 (bytes into file)
  Flags:                             0x0
  Size of this header:               52 (bytes)
  Size of program headers:           32 (bytes)
  Number of program headers:         9
  Size of section headers:           40 (bytes)
  Number of section headers:         29
  Section header string table index: 28
vivek@possum:~/elf_os-l2$ objdump -d ./a.out | grep -A16 3e0
00000 3e0 <_start>:
 3e0:   31 ed                   xor    %ebp,%ebp
 3e2:   5e                      pop    %esi
 3e3:   89 e1                   mov    %esp,%ecx
 3e5:   83 e4 f0                and    $0xfffffff0,%esp
 3e8:   50                      push   %eax
 3e9:   54                      push   %esp
 3ea:   52                      push   %edx
 3eb:   e8 22 00 00 00          call   412 <_start+0x32>
 3f0:   81 c3 e8 1b 00 00       add    $0x1be8,%ebx
 3f6:   8d 83 f8 e5 ff ff       lea    -0x1a08(%ebx),%eax
 3fc:   50                      push   %eax
 3fd:   8d 83 98 e5 ff ff       lea    -0x1a68(%ebx),%eax
 403:   50                      push   %eax
 404:   51                      push   %ecx
 405:   56                      push   %esi
 406:   ff b3 20 00 00 00       pushl  0x20(%ebx)
 40c:   e8 af ff ff ff          call   3c0 <__libc_start_main@plt>
```

Entry point address

Entry point actual method

Entry point address

_start calls __libc_start_main with the address of user main as a parameter (invoked from here)

ELF Section Header

```
typedef struct {
        Elf32_Word      sh_name;
        Elf32_Word      sh_type;
        Elf32_Word      sh_flags;
        Elf32_Addr      sh_addr;
        Elf32_Off       sh_offset;
        Elf32_Word      sh_size;
        Elf32_Word      sh_link;
        Elf32_Word      sh_info;
        Elf32_Word      sh_addralign;
        Elf32_Word      sh_entsize;
} Elf32_Shdr;
```

Offset (bytes) into the section "String Table" where the name of this section is stored

Section type (symbol table, string table, programmer defined data, etc.)

Runtime information for section, e.g., writable data, executable data, loadable

Address at which first byte of this section should reside at runtime (actual address in PHT for a.out)

Offset (bytes) of the starting byte in this section from the beginning of the object file

Size of the section (vary across sections)

Sections at Linking and Executable

```
vivek@possum:~/elf_os-12$ readelf -S hello.o
There are 17 section headers, starting at offset 0x3d4:

Section Headers:
  [Nr] Name              Type             Addr     Off    Size   ES Flg Lk Inf Al
  [ 0]                   NULL             00000000 000000 000000 00      0   0  0
  [ 1] .group            GROUP            00000000 000034 000008 04     14  16  4
  [ 2] .text             PROGBITS         00000000 00003c 000059 00  AX  0   0  1
  [ 3] .rel.text         REL              00000000 0002e4 000040 08   I 14   2  4
  [ 4] .data             PROGBITS         00000000 000098 000004 00  WA  0   0  4
  [ 5] .bss              NOBITS           00000000 00009c 000000 00  WA  0   0  1
  [ 6] .rodata           PROGBITS         00000000 00009c 000025 00   A  0   0  1
  [ 7] .data.rel.local   PROGBITS         00000000 0000c4 000004 00  WA  0   0  4
  [ 8] .rel.data.rel.loc REL              00000000 000324 000008 08   I 14   7  4
  [ 9] .text.__x86.get_p PROGBITS         00000000 0000c8 000004 00 AXG  0   0  1
  [10] .comment          PROGBITS         00000000 0000cc 00002a 01  MS  0   0  1
  [11] .note.GNU-stack   PROGBITS         00000000 0000f6 000000 00      0   0  1
  [12] .eh_frame         PROGBITS         00000000 0000f8 000060 00   A  0   0  4
  [13] .rel.eh_frame     REL              00000000 00032c 000010 08   I 14  12  4
  [14] .symtab           SYMTAB           00000000 000158 000130 10     15  12  4
  [15] .strtab           STRTAB           00000000 000288 00005b 00      0   0  1
  [16] .shstrtab         STRTAB           00000000 00033c 000096 00      0   0  1
Key to Flags:
  W (write), A (alloc), X (execute), M (merge), S (strings), I (info),
  L (link order), O (extra OS processing required), G (group), T (TLS),
  C (compressed), x (unknown), o (OS specific), E (exclude),
  p (processor specific)
```

```
vivek@possum:~/elf_os-12$ readelf -S hello
There are 29 section headers, starting at offset 0x17fc:

Section Headers:
  [Nr] Name              Type             Addr     Off    Size   ES Flg Lk Inf Al
  [ 0]                   NULL             00000000 000000 000000 00      0   0  0
  [ 1] .interp           PROGBITS         00000154 000154 000013 00   A  0   0  1
  [ 2] .note.ABI-tag     NOTE             00000168 000168 000020 00   A  0   0  4
  [ 3] .note.gnu.build-i NOTE             00000188 000188 000024 00   A  0   0  4
  [ 4] .gnu.hash         GNU_HASH         000001ac 0001ac 000020 04   A  5   0  4
  [ 5] .dynsym           DYNSYM           000001cc 0001cc 000080 10   A  6   1  4
  [ 6] .dynstr           STRTAB           0000024c 00024c 00009d 00   A  0   0  1
  [ 7] .gnu.version      VERSYM           000002ea 0002ea 000010 02   A  5   0  2
  [ 8] .gnu.version_r    VERNEED          000002fc 0002fc 000030 00   A  6   1  4
  [ 9] .rel.dyn          REL              0000032c 00032c 000048 08   A  5   0  4
  [10] .rel.plt          REL              00000374 000374 000010 08  AI  5  22  4
  [11] .init             PROGBITS         00000384 000384 000023 00  AX  0   0  4
  [12] .plt              PROGBITS         000003b0 0003b0 000030 04  AX  0   0 16
  [13] .plt.got          PROGBITS         000003e0 0003e0 000010 08  AX  0   0  8
  [14] .text             PROGBITS         000003f0 0003f0 000202 00  AX  0   0 16
  [15] .fini             PROGBITS         000005f4 0005f4 000014 00  AX  0   0  4
  [16] .rodata           PROGBITS         00000608 000608 00002d 00   A  0   0  4
  [17] .eh_frame_hdr     PROGBITS         00000638 000638 00003c 00   A  0   0  4
  [18] .eh_frame         PROGBITS         00000674 000674 0000fc 00   A  0   0  4
  [19] .init_array       INIT_ARRAY       00001ed8 000ed8 000004 04  WA  0   0  4
  [20] .fini_array       FINI_ARRAY       00001edc 000edc 000004 04  WA  0   0  4
  [21] .dynamic          DYNAMIC          00001ee0 000ee0 0000f8 08  WA  6   0  4
  [22] .got              PROGBITS         00001fd8 000fd8 000028 04  WA  0   0  4
  [23] .data             PROGBITS         00002000 001000 000010 00  WA  0   0  4
  [24] .bss              NOBITS           00002020 001010 400020 00  WA  0   0 32
  [25] .comment          PROGBITS         00000000 001010 000029 01  MS  0   0  1
  [26] .symtab           SYMTAB           00000000 00103c 000460 10     27  43  4
  [27] .strtab           STRTAB           00000000 00149c 000264 00      0   0  1
  [28] .shstrtab         STRTAB           00000000 001700 0000fc 00      0   0  1
Key to Flags:
  W (write), A (alloc), X (execute), M (merge), S (strings), I (info),
  L (link order), O (extra OS processing required), G (group), T (TLS),
  C (compressed), x (unknown), o (OS specific), E (exclude),
  p (processor specific)
```

Q: Why there are more number of sections in executable v/s relocatable file?

A: To assist the dynamic loader, the linker adds several more sections in the executable than in the relocatable

Few Interesting Sections

```
vivek@possum:~/elf_os-l2$ readelf -S hello
There are 29 section headers, starting at offset 0x17fc:

Section Headers:
  [Nr] Name              Type             Addr     Off    Size   ES Flg Lk Inf Al
  [ 0]                   NULL             00000000 000000 000000 00      0   0  0
  [ 1] .interp           PROGBITS         00000154 000154 000013 00   A  0   0  1
  [ 2] .note.ABI-tag     NOTE             00000168 000168 000020 00   A  0   0  4
  [ 3] .note.gnu.build-i NOTE             00000188 000188 000024 00   A  0   0  4
  [ 4] .gnu.hash         GNU_HASH         000001ac 0001ac 000020 04   A  5   0  4
  [ 5] .dynsym           DYNSYM           000001cc 0001cc 000080 10   A  6   1  4
  [ 6] .dynstr           STRTAB           0000024c 00024c 00009d 00   A  0   0  1
  [ 7] .gnu.version      VERSYM           000002ea 0002ea 000010 02   A  5   0  2
  [ 8] .gnu.version_r    VERNEED          000002fc 0002fc 000030 00   A  6   1  4
  [ 9] .rel.dyn          REL              0000032c 00032c 000048 08   A  5   0  4
  [10] .rel.plt          REL              00000374 000374 000010 08  AI  5  22  4
  [11] .init             PROGBITS         00000384 000384 000023 00  AX  0   0  4
  [12] .plt              PROGBITS         000003b0 0003b0 000030 04  AX  0   0 16
  [13] .plt.got          PROGBITS         000003e0 0003e0 000010 08  AX  0   0  8
  [14] .text             PROGBITS         000003f0 0003f0 000202 00  AX  0   0 16
  [15] .fini             PROGBITS         000005f4 0005f4 000014 00  AX  0   0  4
  [16] .rodata           PROGBITS         00000608 000608 00002d 00   A  0   0  4
  [17] .eh_frame_hdr     PROGBITS         00000638 000638 00003c 00   A  0   0  4
  [18] .eh_frame         PROGBITS         00000674 000674 0000fc 00   A  0   0  4
  [19] .init_array       INIT_ARRAY       00001ed8 000ed8 000004 04  WA  0   0  4
  [20] .fini_array       FINI_ARRAY       00001edc 000edc 000004 04  WA  0   0  4
  [21] .dynamic          DYNAMIC          00001ee0 000ee0 0000f8 08  WA  6   0  4
  [22] .got              PROGBITS         00001fd8 000fd8 000028 04  WA  0   0  4
  [23] .data             PROGBITS         00002000 001000 000010 00  WA  0   0  4
  [24] .bss              NOBITS           00002020 001010 400020 00  WA  0   0 32
  [25] .comment          PROGBITS         00000000 001010 000029 01  MS  0   0  1
  [26] .symtab           SYMTAB           00000000 00103c 000460 10     27  43  4
  [27] .strtab           STRTAB           00000000 00149c 000264 00      0   0  1
  [28] .shstrtab         STRTAB           00000000 001700 0000fc 00      0   0  1
Key to Flags:
  W (write), A (alloc), X (execute), M (merge), S (strings), I (info),
  L (link order), O (extra OS processing required), G (group), T (TLS),
  C (compressed), x (unknown), o (OS specific), E (exclude),
  p (processor specific)
```

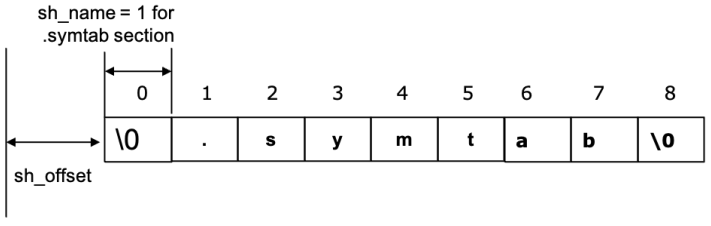
```
vivek@possum:~/elf_os-l2$ cat hello.c
#include<stdio.h>

int my_global1 = 1, my_global2[1024*1024];
char* str = "Updated value of my_global1";

int main() {
  my_global1 += 1;
  printf("%s = %d\n", str, my_global1);
  return 0;
}
```

- Available in both relocatable and executable
  - .shstrtab
  - .text
  - .bss
  - .data
  - .rodata
  - .symtab
  - .strtab
- Available only in executable
  - .interp
  - .dynsym
  - .dynstr

String Table Section (".shstrtab")

```
vivek@possum:~/elf_os-12$ readelf -p .shstrtab hello

String dump of section '.shstrtab':
  [     1]  .symtab
  [     9]  .strtab
  [    11]  .shstrtab
  [    1b]  .interp
  [    23]  .note.ABI-tag
  [    31]  .note.gnu.build-id
  [    44]  .gnu.hash
  [    4e]  .dynsym
  [    56]  .dynstr
  [    5e]  .gnu.version
  [    6b]  .gnu.version_r
  [    7a]  .rel.dyn
  [    83]  .rel.plt
  [    8c]  .init
  [    92]  .plt.got
  [    9b]  .text
  [    a1]  .fini
  [    a7]  .rodata
  [    af]  .eh_frame_hdr
  [    bd]  .eh_frame
  [    c7]  .init_array
  [    d3]  .fini_array
  [    df]  .dynamic
  [    e8]  .data
  [    ee]  .bss
  [    f3]  .comment
```

- As with other sections, this section is also a contiguous chunk of memory
- Stores the name of each sections
  - String (NULL terminated)



First column is the index (hex value) of the first character of that section name. First entry starts at index 1

Executable Instructions (“.text”)

```
vivek@possum:~/elf_os-12$ objdump -d hello.o | grep -A20 text
Disassembly of section .text:

00000000 <main>:
   0:   8d 4c 24 04             lea    0x4(%esp),%ecx
   4:   83 e4 f0                and    $0xfffffff0,%esp
   7:   ff 71 fc                pushl  -0x4(%ecx)
   a:   55                      push   %ebp
   b:   89 e5                   mov    %esp,%ebp
   d:   53                      push   %ebx
   e:   51                      push   %ecx
   f:   e8 fc ff ff ff          call   10 <main+0x10>
  14:   05 01 00 00 00          add    $0x1,%eax
  19:   8b 90 00 00 00 00       mov    0x0(%eax),%edx
  1f:   83 c2 01                add    $0x1,%edx
  22:   89 90 00 00 00 00       mov    %edx,0x0(%eax)
  28:   8b 88 00 00 00 00       mov    0x0(%eax),%ecx
  2e:   8b 90 00 00 00 00       mov    0x0(%eax),%edx
  34:   83 ec 04                sub    $0x4,%esp
  37:   51                      push   %ecx
  38:   52                      push   %edx
  39:   8d 90 1c 00 00 00       lea    0x1c(%eax),%edx
```

```
vivek@possum:~/elf_os-12$ objdump -d hello | grep -A20  text
Disassembly of section .text:

000003f0 <_start>:
 3f0:   31 ed                   xor    %ebp,%ebp
 3f2:   5e                      pop    %esi
 3f3:   89 e1                   mov    %esp,%ecx
 3f5:   83 e4 f0                and    $0xfffffff0,%esp
 3f8:   50                      push   %eax
 3f9:   54                      push   %esp
 3fa:   52                      push   %edx
 3fb:   e8 22 00 00 00          call   422 <_start+0x32>
 400:   81 c3 d8 1b 00 00       add    $0x1bd8,%ebx
 406:   8d 83 18 e6 ff ff       lea    -0x19e8(%ebx),%eax
 40c:   50                      push   %eax
 40d:   8d 83 b8 e5 ff ff       lea    -0x1a48(%ebx),%eax
 413:   50                      push   %eax
 414:   51                      push   %ecx
 415:   56                      push   %esi
 416:   ff b3 20 00 00 00       pushl  0x20(%ebx)
 41c:   e8 af ff ff ff          call   3d0 <__libc_start_main@
 421:   f4                      hlt
```

```
vivek@possum:~/elf_os-12$ readelf -S hello | grep 'text \| Name'
  [Nr] Name              Type            Addr     Off    Size   ES Flg Lk Inf Al
  [14] .text             PROGBITS        000003f0 0003f0 000202 00  AX  0   0 16
```

- Stores bytes corresponding to program execution (user code and the GNU C library supporting code)
- “AX” flags indicating loadable and executable section

Section ".rodata"

```
vivek@possum:~/elf_os-l2$ cat hello.c
#include<stdio.h>

int my_global1 = 1, my_global2[1024*1024];
char* str = "Updated value of my_global1";

int main() {
  my_global1 += 1;
  printf("%s = %d\n", str, my_global1);
  return 0;
}
```

```
vivek@possum:~/elf_os-l2$ readelf -p .rodata hello

String dump of section '.rodata':
  [     8]  Updated value of my_global1
  [    24]  %s = %d^J
```

- Stores all the string literals defined in the program
- Question
  - Would the content change across relocatable and executable?

Section ".data"

```
vivek@possum:~/elf_os-l2$ cat hello.c
#include<stdio.h>

int my_global1 = 1, my_global2[1024*1024];
char* str = "Updated value of my_global1";

int main() {
  my_global1 += 1;
  printf("%s = %d\n", str, my_global1);
  return 0;
}
```

```
vivek@possum:~/elf_os-l2$ readelf -S hello | grep '\.data \| Name'
  [Nr] Name              Type             Addr     Off    Size   ES Flg Lk Inf Al
  [23] .data             PROGBITS         00002000 001000 000010 00  WA  0   0  4
```

```
vivek@possum:~/elf_os-l2$ readelf -S hello.o | grep '\.data \| Name'
  [Nr] Name              Type             Addr     Off    Size   ES Flg Lk Inf Al
  [ 4] .data             PROGBITS         00000000 000098 000004 00  WA  0   0  4
```

- Contains the **initialized value** of all the global and static variables from the user code (not the variable name)
  - In our running example it is the **value** of the variable "my_global1"
    - Why not the value of "str"?
- "WA" flag indicates that the section content is to be loaded at runtime in memory and is writable
- PROGBITS indicating it contains information defined by the user code
- Where is the name of the global variable stored?

Section ".bss"

```
vivek@possum:~/elf_os-l2$ cat hello.c
#include<stdio.h>

int my_global1 = 1, my_global2[1024*1024];
char* str = "Updated value of my_global1";

int main() {
  my_global1 += 1;
  printf("%s = %d\n", str, my_global1);
  return 0;
}
```

Hex

```
vivek@possum:~/elf_os-l2$ readelf -S hello | grep '\.bss\| Name'
  [Nr] Name              Type             Addr     Off    Size   ES Flg Lk Inf Al
  [24] .bss              NOBITS           00002020 001010 400020 00  WA  0   0 32
```

- .bss (Block Started by Symbol) section indicates the total memory to be allocated at runtime for uninitialized global/static variables
  - In our running example it is the **value** of the variable "my_global2"
- "WA" flag indicates that the section content is to be loaded at runtime in memory and is writable
- NOBITS indicating it relates to the user code, but it doesn't occupy any space in the object file
- Where is the name of the global variable stored?

Sections ".symtab" and ".strtab"

```
vivek@possum:~/elf_os-l2$ readelf -p .symtab hello.o

String dump of section '.symtab':
  [     d4]
  [     da]  @
  [     f0]  #
  [     f8]  Y
  [    100]  (
  [    110]  >
  [    120]  T

vivek@possum:~/elf_os-l2$ readelf -p .strtab hello.o

String dump of section '.strtab':
  [      1]  hello.c
  [      9]  my_global1
  [     14]  my_global2
  [     1f]  str
  [     23]  main
  [     28]  __x86.get_pc_thunk.ax
  [     3e]  _GLOBAL_OFFSET_TABLE_
  [     54]  printf
```

- ".symtab" is like a directory to map the name of the symbols (present inside .strtab) with the actual content (value) of that symbol
  - E.g., for the symbol "my_global1", symbol table will contain an index into the string table ".strtab" where the name "my_global1" is stored. It will also contain an index to the section header holding the value of "my_global1"
    - What will be that section name?
  - ".symtab" only details about the symbols present during compile-time linking **only**
    - E.g., it will contain the symbol "printf"
      - Why?
- Sections ".dynsym" and ".dynstr" are similar to ".symtab" and ".strtab" with the only difference that they **only** refer to dynamically linked symbols (e.g., printf)

Section ".interp

- ".interp" stores the path to the interpreter (/lib/ld-linux.so) that contains a dynamic linker to bind the references to shared library functions to the actual definitions (code) in the shared library

Segments in Executable File

```
vivek@possum:~/elf_os-l2$ readelf -l hello

Elf file type is DYN (Shared object file)
Entry point 0x3f0
There are 9 program headers, starting at offset 52

Program Headers:
  Type           Offset   VirtAddr   PhysAddr   FileSiz MemSiz  Flg Align
  PHDR           0x000034 0x00000034 0x00000034 0x00120 0x00120 R   0x4
  INTERP         0x000154 0x00000154 0x00000154 0x00013 0x00013 R   0x1
      [Requesting program interpreter: /lib/ld-linux.so.2]
  LOAD           0x000000 0x00000000 0x00000000 0x00770 0x00770 R E 0x1000
  LOAD           0x000ed8 0x00001ed8 0x00001ed8 0x00138 0x400168 RW  0x1000
  DYNAMIC        0x000ee0 0x00001ee0 0x00001ee0 0x000f8 0x000f8 RW  0x4
  NOTE           0x000168 0x00000168 0x00000168 0x00044 0x00044 R   0x4
  GNU_EH_FRAME   0x000638 0x00000638 0x00000638 0x0003c 0x0003c R   0x4
  GNU_STACK      0x000000 0x00000000 0x00000000 0x00000 0x00000 RW  0x10
  GNU_RELRO      0x000ed8 0x00001ed8 0x00001ed8 0x00128 0x00128 R   0x1

 Section to Segment mapping:
  Segment Sections...
   00
   01     .interp
   02     .interp .note.ABI-tag .note.gnu.build-id .gnu.hash .dynsym .dynstr .
gnu.version .gnu.version_r .rel.dyn .rel.plt .init .plt .plt.got .text .fini .
rodata .eh_frame_hdr .eh_frame
   03     .init_array .fini_array .dynamic .got .data .bss
   04     .dynamic
   05     .note.ABI-tag .note.gnu.build-id
   06     .eh_frame_hdr
   07
```

- Segments are created by merging several sections and each segment contains a specific type of information required for by the loader for execution (i.e., all the sections inside a segment are related)
- Not available in relocatable file

Reading Materials

- Executable and Linkable Format
  - https://www.cs.cmu.edu/afs/cs/academic/class/15213-f00/docs/elf.pdf
  - https://man7.org/linux/man-pages/man5/elf.5.html

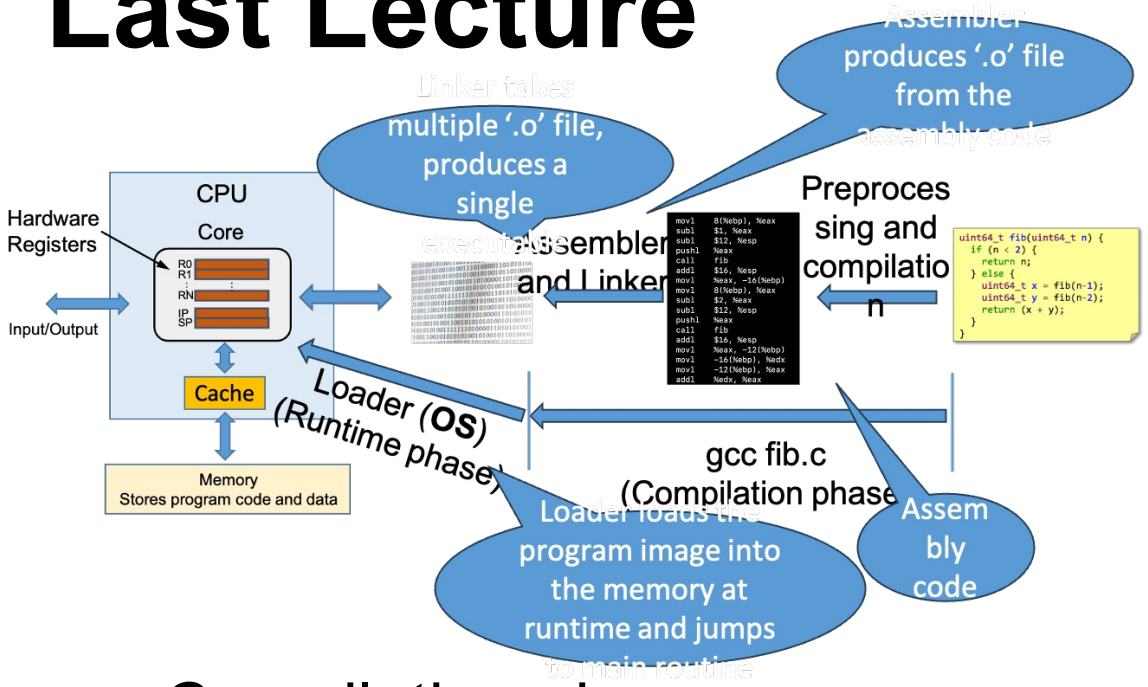
Next Lecture

- Dissecting an ELF file
  - It will be the topic for assignment-1
    - Will be released on 17th August

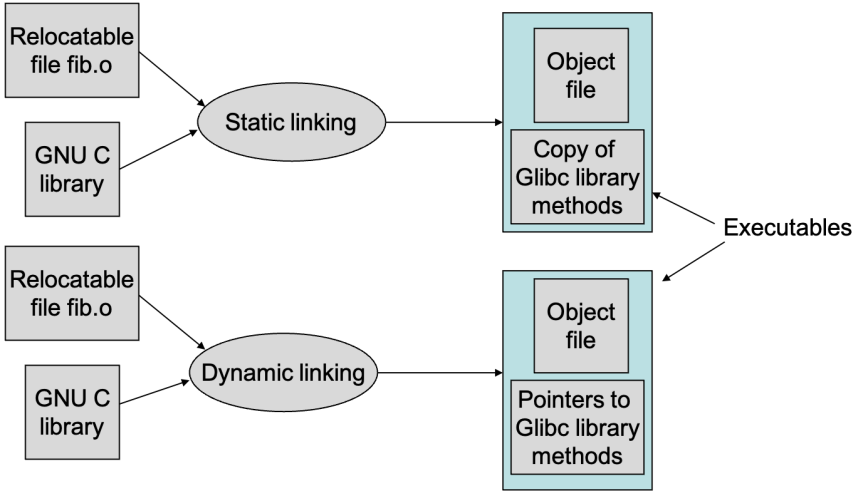
Lecture 03: Linking and Loading of an ELF File

Vivek Kumar
Computer Science and Engineering
IIIT Delhi
vivekk@iiitd.ac.in

Last Lecture




```
$ gcc -static -o fib fib.c
$ file fib
...ELF 64-bit executable...statically linked
$ ls -l fib
-rwxrwxr-x 1 vivek vivek 845304 Aug  6 10:26 fib
$ gcc -o fib fib.c
$ ls -l fib
-rwxrwxr-x 1 vivek vivek 8328 Aug  6 10:27 fib
```



- Compilation phases
- Static and dynamic linking
- Object files
  - Relocatable and executable files

Today's Class

- Revisiting Executable File Format (ELF)
- Dissecting the sections of an ELF
  - Symbol resolution
- Working of loader

Class Quiz (1/5)

- Match the following structures with their respective names and responsibilities

(A) Elf32_Phdr	(X) ELF header	(1) Information for Loader
(B) Elf32_Ehdr	(Y) ELF Section Header	(2) Stores roadmap of ELF
(C) Elf32_Shdr	(Z) ELF Program Header	(3) Information for Linker

Class Quiz (2/5)

- What is the meaning of "entrypoint" address?
  1. Offset of **SHDR** table from the beginning of ELF file
  2. Offset of **PHDR** table from the beginning of ELF file
  3. Address of "main" method
  4. Address of "_start" method

- Where it is stored inside the ELF binaries?
  1. Program header table
  2. Section header table
  3. ELF header
  4. Segments
  5. Sections

Class Quiz (3/5)

- Which of these are required at the compile time and runtime, respectively?
  1. ELF header
  2. PHDR
  3. SHDR
  4. Segments
  5. Sections

At Runtime

At Compile time

Class Quiz (4/5)

- Which of these sections appear in “executable” file and which ones in “relocatable” file?

1. .bss
2. .dynsym
3. .shstrtab
4. .data
5. .text
6. .dynstr
7. .symtab
8. .strtab

In Executable File

In Relocatable Files

Class Quiz (5/5)

- Match the location shown with their respective section name inside the ELF file

```
vivek@possum:~/elf_os-l2$ cat hello.c
#include<stdio.h>
int my_global1 = 1, my_global2[1024*1024];
char* str = "Updated value of my_global1";

int main() {
  my_global1 += 1;
  printf("%s = %d\n", str, my_global1);
  return 0;
}
```

A B C D E F

Symbol name only in this case

1. .bss
2. .strtab
3. .rodata
4. .text
5. .data

Today's Class

- Revisiting Executable File Format (ELF)
- Dissecting the sections of an ELF
  - Symbol resolution
- Working of loader

Portability with ELF

- Recall, one of the challenges of OS is to support portability
- Machine code is architecture dependent
- Having an architecture independent layout of object files helps in achieving portability
  - The interpreter (loader + dynamic linker) becomes portable, and could have same set of steps to load executables across different architectures (portability)
    - Easier to change and test the tools used for manipulating object files

Example: Reading ".shstrtab" in Plain C

Relocatable file fib.o

```
ELF Header:
  Magic:   7f 45 4c 46 01 01 01 00 00 00 00 00 00 00 00 00
  Class:                             ELF32
  Data:                              2's complement, little en
  Version:                           1 (current)
  OS/ABI:                            UNIX - System V
  ABI Version:                       0
  Type:                              REL (Relocatable file)
  Machine:                           Intel 80386
  Version:                           0x1
  Entry point address:               0x0
  Start of program headers:          0 (bytes into file)
  Start of section headers:          860 (bytes into file)
  Flags:                             0x0
  Size of this header:               52 (bytes)
  Size of program headers:           0 (bytes)
  Number of program headers:         0
  Size of section headers:           40 (bytes)
  Number of section headers:         14
  Section header_string table index: 13
```

.shstrtab

```
Section Headers:
  [Nr] Name              Type            Addr     Off    Size
  [ 0]                   NULL            00000000 000000 000000
  [ 1] .group            GROUP           00000000 000034 000008
  [ 2] .text             PROGBITS        00000000 00003c 00008f
  [ 3] .rel.text         REL             00000000 000290 000038
  [ 4] .data             PROGBITS        00000000 0000cb 000000
  [ 5] .bss              NOBITS          00000000 0000cb 000000
  [ 6] .text.__x86.get_p PROGBITS        00000000 0000cb 000004
  [ 7] .comment          PROGBITS        00000000 0000cf 00002a
  [ 8] .note.GNU-stack   PROGBITS        00000000 0000f9 000000
  [ 9] .eh_frame         PROGBITS        00000000 0000fc 000078
  [10] .rel.eh_frame     REL             00000000 0002c8 000018
  [11] .symtab           SYMTAB          00000000 000174 0000e0
  [12] .strtab           STRTAB          00000000 000254 00003c
  [13] .shstrtab         STRTAB          00000000 0002e0 00007a
```

```
#define EI_NIDENT 16

typedef struct {
        unsigned char   e_ident[EI_NIDENT];
        Elf32_Half      e_type;
        Elf32_Half      e_machine;
        Elf32_Word      e_version;
        Elf32_Addr      e_entry;
        Elf32_Off       e_phoff;
        Elf32_Off       e_shoff;
        Elf32_Word      e_flags;
        Elf32_Half      e_ehsize;
        Elf32_Half      e_phentsize;
        Elf32_Half      e_phnum;
        Elf32_Half      e_shentsize;
        Elf32_Half      e_shnum;
        Elf32_Half      e_shstrndx;
} Elf32_Ehdr;
```

```
typedef struct {
        Elf32_Word      sh_name;
        Elf32_Word      sh_type;
        Elf32_Word      sh_flags;
        Elf32_Addr      sh_addr;
        Elf32_Off       sh_offset;
        Elf32_Word      sh_size;
        Elf32_Word      sh_link;
        Elf32_Word      sh_info;
        Elf32_Word      sh_addralign;
        Elf32_Word      sh_entsize;
} Elf32_Shdr;
```

1. Open the a.out in O_RDONLY mode
2. Read the value of the following member in the Elf32_Ehdr
   a. e_shoff
   b. e_shentsize (fixed)
   c. e_shstrndx
3. Move to the offset e_shoff + e_shstrndx* e_shentsize
   a. Starting address of Elf32_Shdr for ".shstrtab"
4. Read the value of the following members in the Elf32_Shdr
   a. sh_offset
   b. sh_size
5. Move to the offset sh_offset from the start of the file a.out
   a. Content of the ".shstrtab" starts from this address
   b. Total content size "sh_size" bytes

Example: Reading Section Names in Plain C

Relocatable file fib.o

```
ELF Header:
  Magic:   7f 45 4c 46 01 01 01 00 00 00 00 00 00 00 00 00
  Class:                             ELF32
  Data:                              2's complement, little en
  Version:                           1 (current)
  OS/ABI:                            UNIX - System V
  ABI Version:                       0
  Type:                              REL (Relocatable file)
  Machine:                           Intel 80386
  Version:                           0x1
  Entry point address:               0x0
  Start of program headers:          0 (bytes into file)
  Start of section headers:          860 (bytes into file)
  Flags:                             0x0
  Size of this header:               52 (bytes)
  Size of program headers:           0 (bytes)
  Number of program headers:         0
  Size of section headers:           40 (bytes)
  Number of section headers:         14
  Section header_string table index: 13
```

.text

.bss

.shstrtab

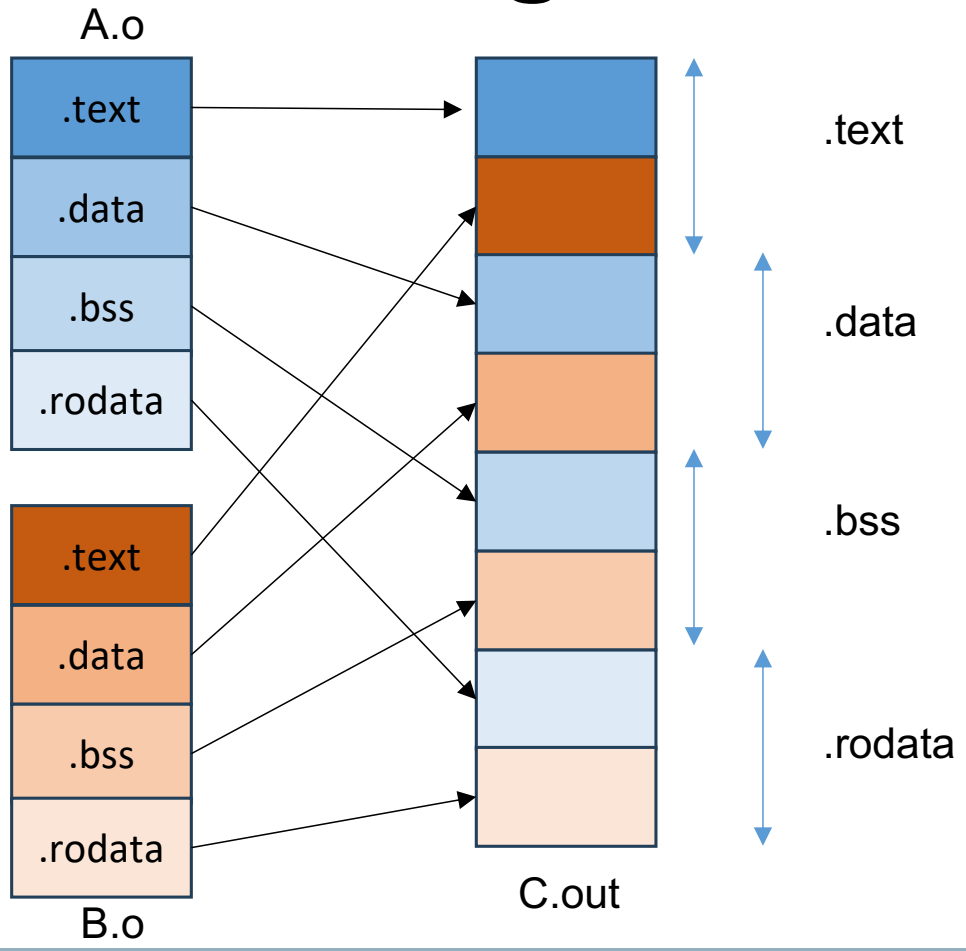
```
Section Headers:
  [Nr] Name              Type            Addr     Off    Size
  [ 0]                   NULL            00000000 000000 000000
  [ 1] .group            GROUP           00000000 000034 000008
  [ 2] .text             PROGBITS        00000000 00003c 00008f
  [ 3] .rel.text         REL             00000000 000290 000038
  [ 4] .data             PROGBITS        00000000 0000cb 000000
  [ 5] .bss              NOBITS          00000000 0000cb 000000
  [ 6] .text.__x86.get_p PROGBITS        00000000 0000cb 000004
  [ 7] .comment          PROGBITS        00000000 0000cf 00002a
  [ 8] .note.GNU-stack   PROGBITS        00000000 0000f9 000000
  [ 9] .eh_frame         PROGBITS        00000000 0000fc 000078
  [10] .rel.eh_frame     REL             00000000 0002c8 000018
  [11] .symtab           SYMTAB          00000000 000174 0000e0
  [12] .strtab           STRTAB          00000000 000254 00003c
  [13] .shstrtab         STRTAB          00000000 0002e0 00007a
```

```
#define EI_NIDENT 16

typedef struct {
        unsigned char   e_ident[EI_NIDENT];
        Elf32_Half      e_type;
        Elf32_Half      e_machine;
        Elf32_Word      e_version;
        Elf32_Addr      e_entry;
        Elf32_Off       e_phoff;
        Elf32_Off       e_shoff;
        Elf32_Word      e_flags;
        Elf32_Half      e_ehsize;
        Elf32_Half      e_phentsize;
        Elf32_Half      e_phnum;
        Elf32_Half      e_shentsize;
        Elf32_Half      e_shnum;
        Elf32_Half      e_shstrndx;
} Elf32_Ehdr;
```

```
typedef struct {
        Elf32_Word      sh_name;
        Elf32_Word      sh_type;
        Elf32_Word      sh_flags;
        Elf32_Addr      sh_addr;
        Elf32_Off       sh_offset;
        Elf32_Word      sh_size;
        Elf32_Word      sh_link;
        Elf32_Word      sh_info;
        Elf32_Word      sh_addralign;
        Elf32_Word      sh_entsize;
} Elf32_Shdr;
```

6. Iterate through all entries in the SHDR one by one
   - SHDR array starts at e_shoff
7. Find the value of sh_name in the SHDR entry
8. Name of this section will be inside the content of the section ".shstrtab and starting at the location sh_name
   - printf (%s) at that location prints the name

Linker: Merges the Sections



- Relocation is the process of merging the sections and resolving the symbol addresses

Linker: Relocation

```
[vivek@possum]$ cat fib.c
int get_number();

int fib(int n) {
  if(n<2) return n;
  else return fib(n-1)+fib(n-2);
}

int compute() {
  int n = get_number();
  return fib(n);
}
```

```
[vivek@possum]$ cat call-fib.c
int compute();
int get_number() {
  return 40;
}
int main() {
        int result = compute();
        return 0;
}
```

Compilation

fib.o

SYM Name	Type	Location
get_number	External	NULL
fib	Internal	Addr_A
compute	Internal	Addr_B

call-fib.o

SYM Name	Type	Location
compute	External	NULL
get_number	Internal	Addr_C

Resolve symbols by linking fib.o and call-fib.o together

a.out

SYM Name	Location
get_number	Addr_D
fib	Addr_E
compute	Addr_F

- Relocatable files contains section headers of type SHT_REL indicating the presence of relocation sections
- Relocation section contains an array of relocation entries, where each entry corresponds to a relocatable symbol in the object file
- Each relocation entry contains information about the name of the symbol (index into the string table section), type of relocation (internal or external), and position (address) of the symbol in the current file

Today's Class

- Revisiting Executable File Format (ELF)
- Dissecting the sections of an ELF
  - Symbol resolution
- Working of loader

Loader

- When an executable file is run, the operating system loads it into memory and start running it
- Loader uses the information stored inside the Program Header Table (PHT)
  - It doesn't require information stored in the Section Header Table (SHDR)
  - Linker combines multiple sections of similar type to create one segment
  - There are multiple segments in an executable file, and for each segment, there is one entry in the PHDR
  - Several sections can be removed from the executable without affecting program execution

Example: Loading Executable in Plain C

```
[vivek@possum]$ cat fib.c
int fib(int n) {
  if(n<2) return n;
  else return fib(n-1)+fib(n-2);
}

int main() {
        int val = fib(40);
        return val;
}
```

```
[vivek@possum]$ gcc -m32 -no-pie -nostdlib fib.c
/usr/bin/ld: warning: cannot find entry symbol _start; de
```

Why ??

Fix: rename "main" to _start"

- For simplicity, a) assume there are no APIs from the GNU C library, b) 32-bit compilation, and c) no use of global variables
- The loader has to simply iterate through the PHDR and identify the segment of PT_LOAD type
- Load this segment in memory and jump to [where?] for starting the execution

e_entrypoint address in EHDR

Example: Loading Executable in Plain C

```
[vivek@possum]$ readelf -h fib
ELF Header:
  Magic:   7f 45 4c 46 01 01 01 00 00 00 00 00 00
  Class:                             ELF32
  Data:                              2's compleme
  Version:                           1 (current)
  OS/ABI:                            UNIX - Syste
  ABI Version:                       0
  Type:                              EXEC (Execut
  Machine:                           Intel 80386
  Version:                           0x1
  Entry point address:               0x8048141
  Start of program headers:          52 (bytes in
  Start of section headers:          5200 (bytes
  Flags:                             0x0
  Size of this header:               52 (bytes)
  Size of program headers:           32 (bytes)
  Number of program headers:         5
  Size of section headers:           40 (bytes)
  Number of section headers:         15
  Section header string table index: 14
```

```
[vivek@possum]$ readelf -l fib

Elf file type is EXEC (Executable file)
Entry point 0x8048141
There are 5 program headers, starting at offset 52

Program Headers:
  Type           Offset   VirtAddr   PhysAddr   FileSiz MemSiz
  LOAD           0x000000 0x08048000 0x08048000 0x00200 0x00200
  LOAD           0x001000 0x0804a000 0x0804a000 0x0000c 0x0000c
  NOTE           0x0000d4 0x080480d4 0x080480d4 0x00024 0x00024
  GNU_EH_FRAME   0x00016c 0x0804816c 0x0804816c 0x00024 0x00024
  GNU_STACK      0x000000 0x00000000 0x00000000 0x00000 0x00000

 Section to Segment mapping:
  Segment Sections...
   00     .note.gnu.build-id .text .eh_frame_hdr .eh_frame
   01     .got.plt
   02     .note.gnu.build-id
   03     .eh_frame_hdr
   04
```

```
typedef struct {
        Elf32_Word      p_type;
        Elf32_Off       p_offset;
        Elf32_Addr      p_vaddr;
        Elf32_Addr      p_paddr;
        Elf32_Word      p_filesz;
        Elf32_Word      p_memsz;
        Elf32_Word      p_flags;
        Elf32_Word      p_align;
} Elf32_Phdr;
```

- Array of PHDR with total ”e_phnum” number of elements starts at ”e_phoff” bytes from the start of the file a.out
- Load the first segment whose ”p_type” is “PT_LOAD” in memory. This segment starts at “p_offset”
- Move to the virtual address ”e_entrypoint” inside this segment
- Typecast it to the ”_start” function pointer type and simply execute!

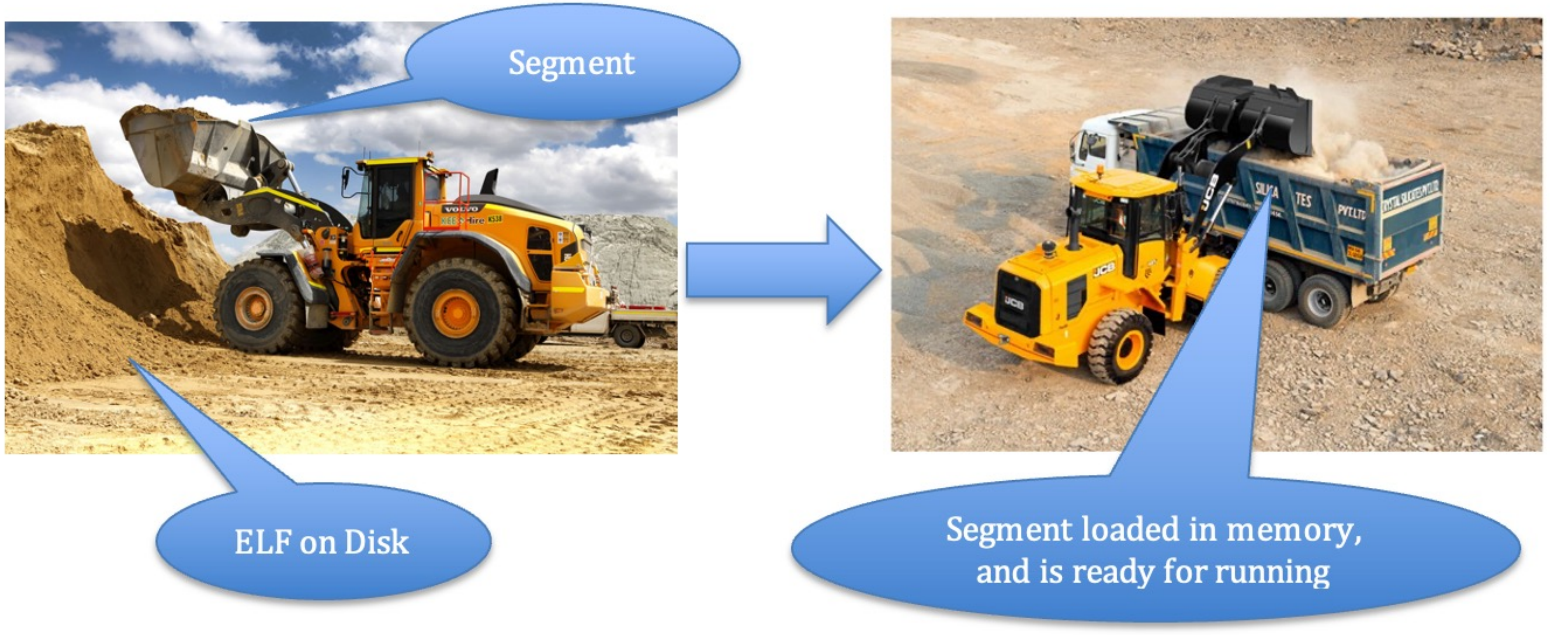
Loader in OS: Complete Steps

- When an executable file is run, the operating system loads it into memory and start it running as follows
  1. OS kernel reads the PHT and loads the parts specified in the segments of type “INTERP” and “LOAD”
  2. Control is transferred to the interpreter (/lib/ld-library.so)
  3. Dynamic Linker (DL) is invoked
  4. DL load the shared libraries needed by the executable and carry out the relocations
     - This information is available in “.dynamic” section
     - Connects symbolic references with their corresponding symbolic definitions (e.g., connecting the call to printf with the actual implementation of the printf in GNU C library)
  5. DL transfer the control to the address given by the symbol “_start” (and eventually to user main)

Assignment-1 (Section-A)

SimpleLoader: An ELF Loader in C from Scratch

Due by 11:59pm on 23rd August 2023
(Total **8%** weightage)
Instructor: Vivek Kumar



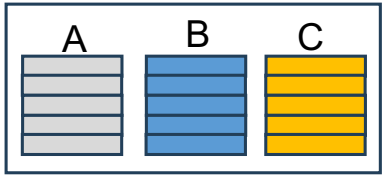
Reference Materials

- Executable and Linkable Format
  - https://man7.org/linux/man-pages/man5/elf.5.html
  - https://www.sco.com/developers/devspecs/gabi41.pdf
  - https://wiki.osdev.org/ELF_Tutorial

Lecture 04: Procedure Calling Convention

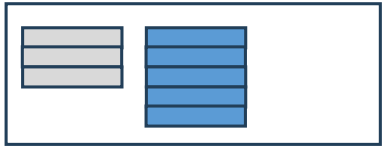
Vivek Kumar
Computer Science and Engineering
IIIT Delhi
vivekk@iiitd.ac.in

Re-motivating ELF



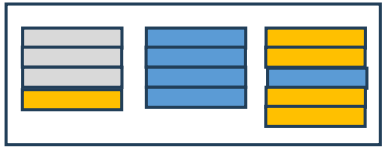
Assignment given to students to write one-line summary for five research papers in each of the 3-research areas A, B, and C. I expected them to submit the summary as shown here

Submission-1



However, student submitted their 1-line summaries in different style/combinations as shown. Imagine how difficult it will be for me to read their summary from 250+ students!

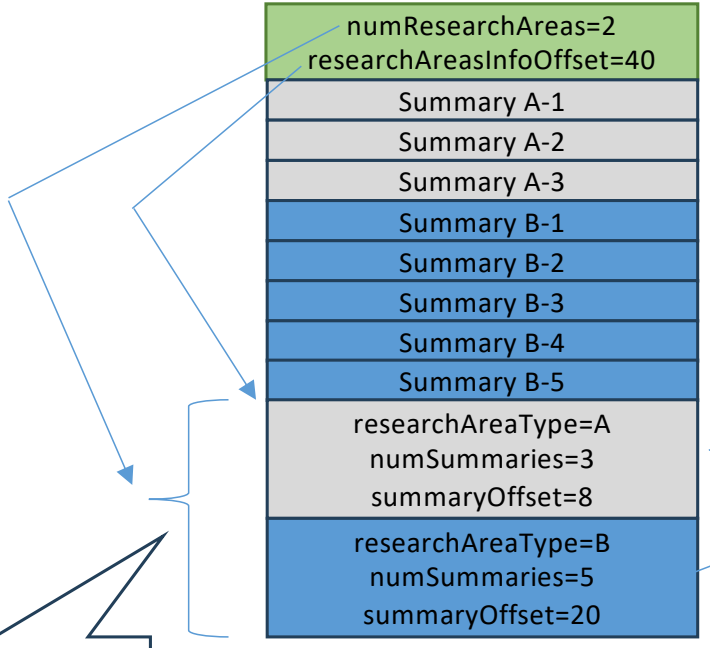
Submission-2



Submission-3

I can easily "load" the student "file" from any number of students in my "memory" if they follow a standard "format" to store information, e.g., as the one shown below for the **Submission-1** shown here

Student submission summary



One line summary for 3 research papers in category-A

One line summary for 5 research papers in category-B

Research paper summary table

© Vivek Kumar

Last Lecture

- Linker
- Loader

```
[vivek@possum]$ cat fib.c
int get_number();

int fib(int n) {
  if(n<2) return n;
  else return fib(n-1)+fib(n-2);
}

int compute() {
  int n = get_number();
  return fib(n);
}
```

```
[vivek@possum]$ cat call-fib.c
int compute();
int get_number() {
  return 40;
}
int main() {
        int result = compute();
        return 0;
}
```

Compilation

fib.o

SYM Name	Type	Location
get_number	External	NULL
fib	Internal	Addr_A
compute	Internal	Addr_B

call-fib.o

SYM Name	Type	Location
compute	External	NULL
get_number	Internal	Addr_C

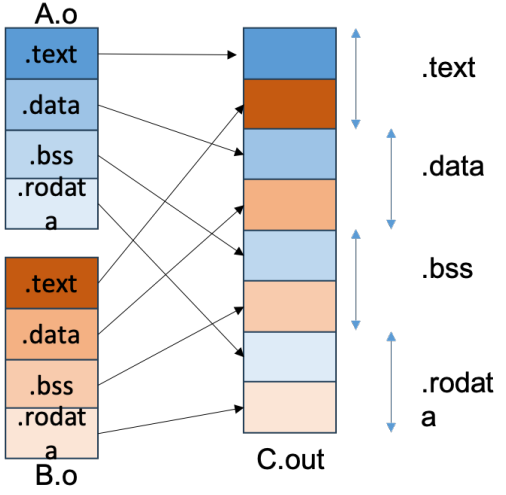
Resolve symbols by linking fib.o and call-fib.o together

a.out

SYM Name	Location
get_number	Addr_D
fib	Addr_E
compute	Addr_F

Loader in OS: Complete Steps

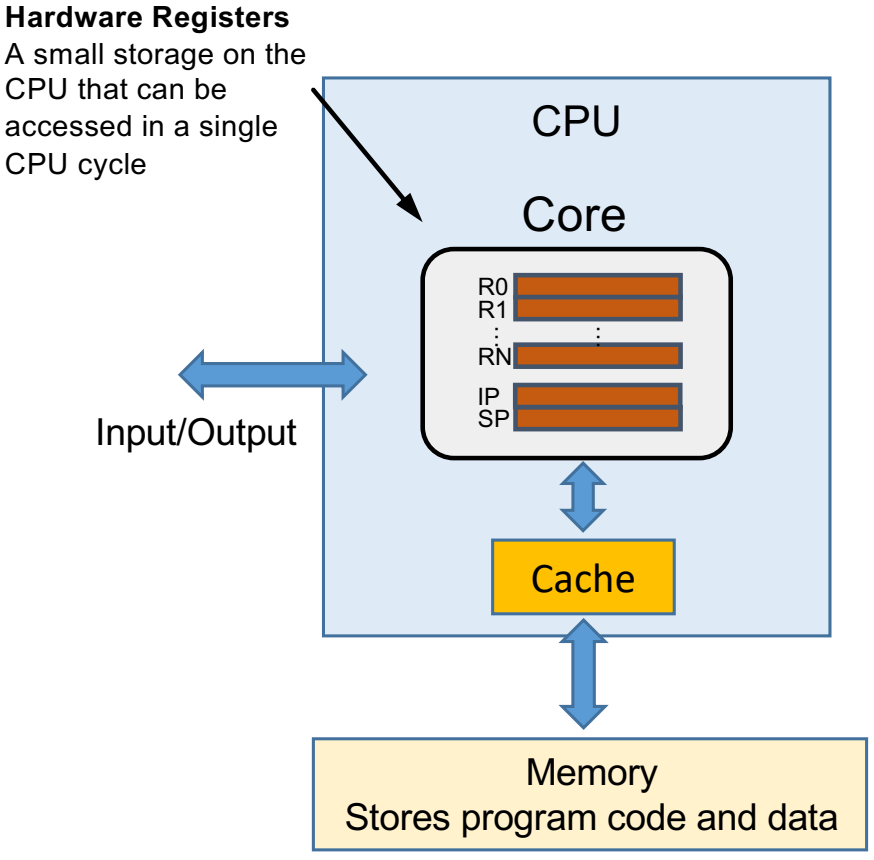
- When an executable file is run, the operating system loads it into memory and start it running as follows
  1. OS kernel reads the PHT and loads the parts specified in the segments of type “INTERP” and “LOAD”
  2. Control is transferred to the interpreter (/lib/ld-library.so)
  3. Dynamic Linker (DL) is invoked
  4. DL load the shared libraries needed by the executable and carry out the relocations
     - This information is available in “.dynamic” section
     - Connects symbolic references with their corresponding symbolic definitions (e.g., connecting the call to printf with the actual implementation of the printf in GNU C library)
  5. DL transfer the control to the address given by the symbol “_start” (and eventually to user main)



Today's Lecture

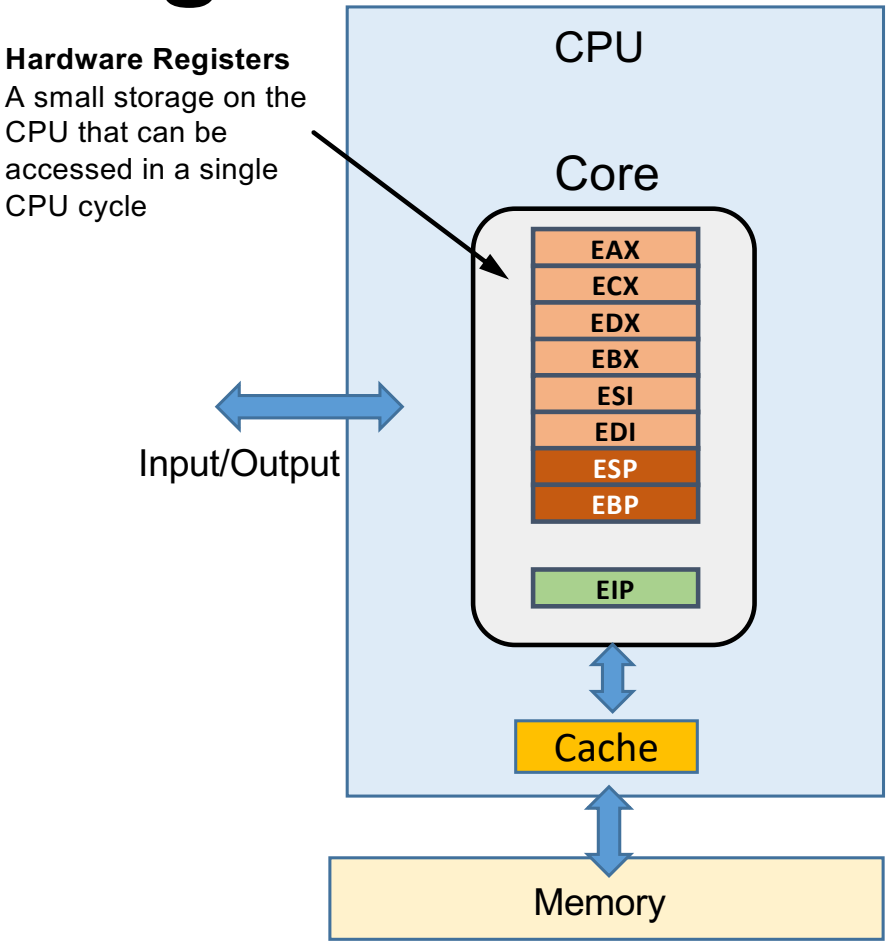
- Call stack
- Procedure calling convention

Instruction Set Architectures (ISA)



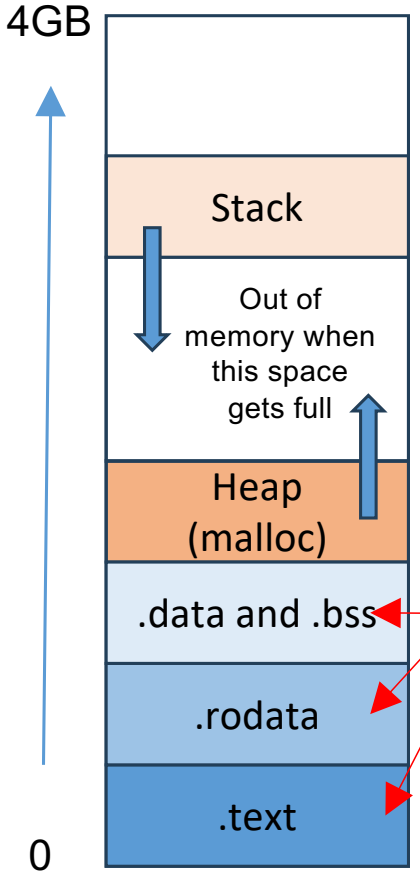
- Abstract model of a computer defining how CPU is controlled by the software
  - Hardware software interface (e.g., program counter, registers, memory, instructions that the CPU can execute, etc.)
  - Can be thought of a programmer's manual to run programs on a CPU
- Some of the widely used ISAs
  - X86 (or IA32, i386)
    - Developed by Intel
    - Widely used in desktop/laptop processors
  - x86-64 (or AMD64)
    - AMD Opteron was first x86-64 processor
    - 64-bit version of x86 ISA
  - ARM
    - Widely used on processors for mobile devices due to their reduced instruction set, which results in small die area and low power consumption

Registers in IA32



- There are total of eight data registers where six are general purpose and two are special purpose (ESP and EBP)
- EIP is a control register to hold the program counter

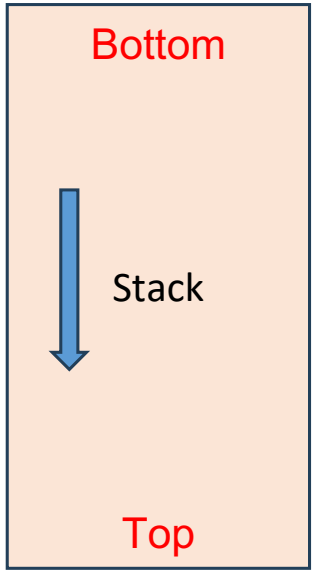
Memory Layout of Program in Execution




```
Program Headers:
  Type           Offset   VirtAddr   PhysAddr   FileSiz MemSiz  Flg Align
  PHDR           0x000034 0x00000034 0x00000034 0x00120 0x00120 R   0x4
  INTERP         0x000154 0x00000154 0x00000154 0x00013 0x00013 R   0x1
      [Requesting program interpreter: /lib/ld-linux.so.2]
  LOAD           0x000000 0x00000000 0x00000000 0x00770 0x00770 R E 0x1000
  LOAD           0x000ed8 0x00001ed8 0x00001ed8 0x00138 0x400168 RW  0x1000
  DYNAMIC        0x000ee0 0x00001ee0 0x00001ee0 0x000f8 0x000f8 RW  0x4
  NOTE           0x000168 0x00000168 0x00000168 0x00044 0x00044 R   0x4
  GNU_EH_FRAME   0x000638 0x00000638 0x00000638 0x0003c 0x0003c R   0x4
  GNU_STACK      0x000000 0x00000000 0x00000000 0x00000 0x00000 RW  0x10
  GNU_RELRO      0x000ed8 0x00001ed8 0x00001ed8 0x00128 0x00128 R   0x1

 Section to Segment mapping:
  Segment Sections...
   00
   01     .interp
   02     .interp .note.ABI-tag .note.gnu.build-id .gnu.hash .dynsym .dynstr
.gnu.version .gnu.version_r .rel.dyn .rel.plt .init .plt .plt.got .text .fini
 .rodata .eh_frame_hdr .eh_frame
   03     .init_array .fini_array .dynamic .got .data .bss
   04     .dynamic
   05     .note.ABI-tag .note.gnu.build-id
   06     .eh_frame_hdr
   07
```

- Total 2^{32} unique memory address on a 32-bit machine
- Stack grows downward whereas heap grows upward
  - Out of memory error generated when there is no more space left. You can easily get this runtime exception while running a Java program. Why?
- Each of the slots shown in the figure consists of several memory pages (4K)
- Stack is the place where the intermediate state of program execution is stored
- Which of the memory slots are Readable? Executable? Writable?

Upside Down Call Stack in IA32



- Stack is just a region of memory allocated by the OS for a running program to save the temporary variables inside a method and to support method invocation
  - Stack grows downward
- Used to hold temporary program state
  - Method call stack (callee and caller information)
    - Local variables
    - Function arguments
    - Return address
    - Temporary space
- This **convention** of saving and restoring program state across method calls is called procedure call linkage. We will now see how it happens on IA32

Callee and Caller

```
L1: main () {

L2:   foo();

L3: }
```

```
L4: foo () {

L5:   bar(100, 200);

L6: }
```

```
L7: bar(int x1, int x2) {

L8:   int b1=x1,b2=x2, b3;

L9:   b3 = baz();

L10: }
```

```
L11: int baz () {

L12:    int r = 10;
L13:    return r;

L14: }
```

- Call chain
  - main → foo → bar → baz
- Callee and Caller
  - main (caller) → foo (callee)
  - foo (caller) → bar (callee)
  - bar (caller) → baz (callee)

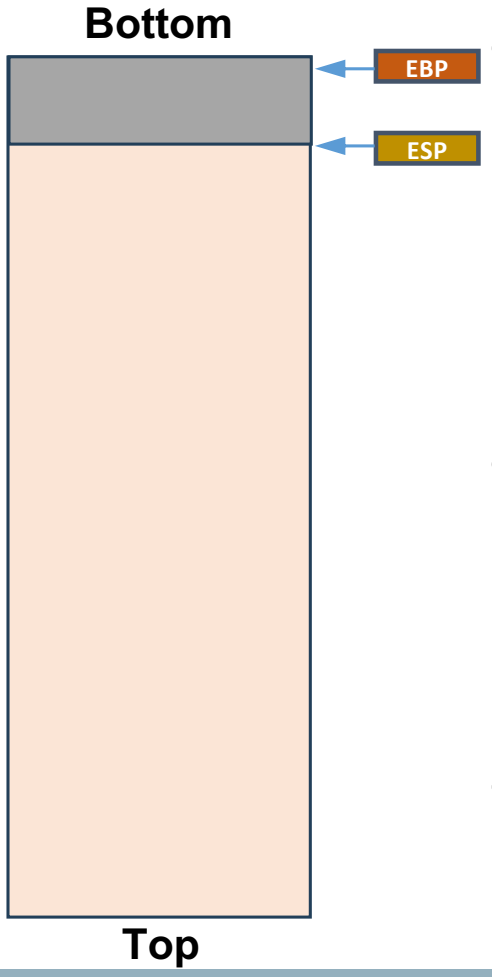
Stack Frames in IA32 (1/7)

```
L1: main () {

L2:   foo();

L3: }
```

```
L4: foo () {

L5:   bar(100, 200);

L6: }
```

```
L7: bar(int x1, int x2) {

L8:   int b1=x1,b2=x2, b3;

L9:   b3 = baz();

L10: }
```

```
L11: int baz () {

L12:    int r = 10;
L13:    return r;

L14: }
```



- EBP register points to bottom of the stack (first item pushed on the stack) and ESP register points to the top of the stack (last item pushed on stack)
- Area of the stack between the location pointed by EBP and ESP is called stack frames for that method
- The topmost frame corresponds to the currently executing method

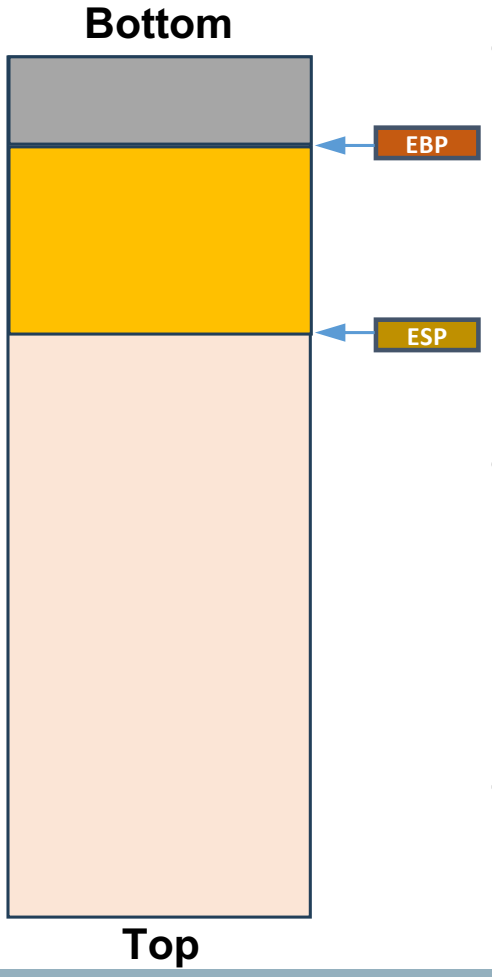
Stack Frames in IA32 (2/7)

```
L1: main () {

L2:   foo();

L3: }
```

```
L4: foo () {

L5:   bar(100, 200);

L6: }
```

```
L7: bar(int x1, int x2) {

L8:   int b1=x1,b2=x2, b3;

L9:   b3 = baz();

L10: }
```

```
L11: int baz () {

L12:    int r = 10;
L13:    return r;

L14: }
```



- EBP register points to bottom of the stack (first item pushed on the stack) and ESP register points to the top of the stack (last item pushed on stack)
- Area of the stack between the location pointed by EBP and ESP is called stack frames for that method
- The topmost frame corresponds to the currently executing method

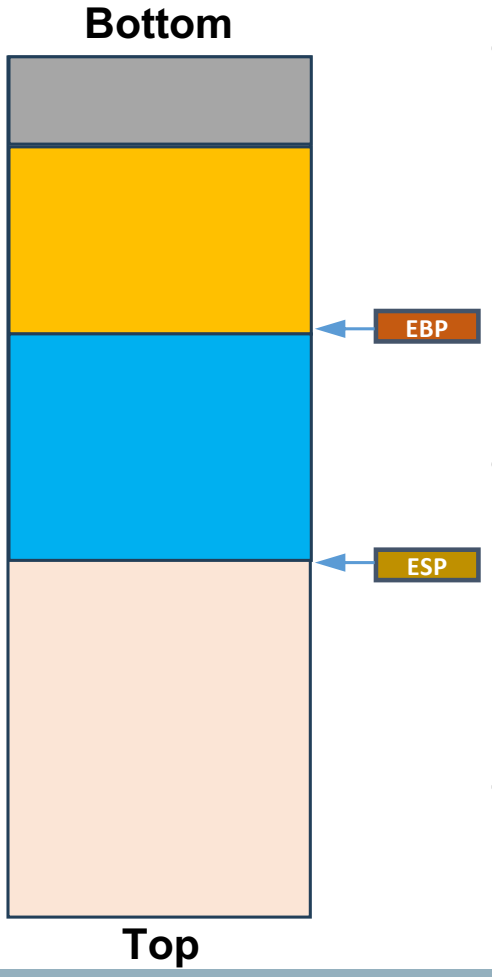
Stack Frames in IA32 (3/7)

```
L1: main () {

L2:   foo();

L3: }
```

```
L4: foo () {

L5:   bar(100, 200);

L6: }
```

```
L7: bar(int x1, int x2) {

L8:   int b1=x1,b2=x2, b3;

L9:   b3 = baz();

L10: }
```

```
L11: int baz () {

L12:    int r = 10;
L13:    return r;

L14: }
```



- EBP register points to bottom of the stack (first item pushed on the stack) and ESP register points to the top of the stack (last item pushed on stack)
- Area of the stack between the location pointed by EBP and ESP is called stack frames for that method
- The topmost frame corresponds to the currently executing method

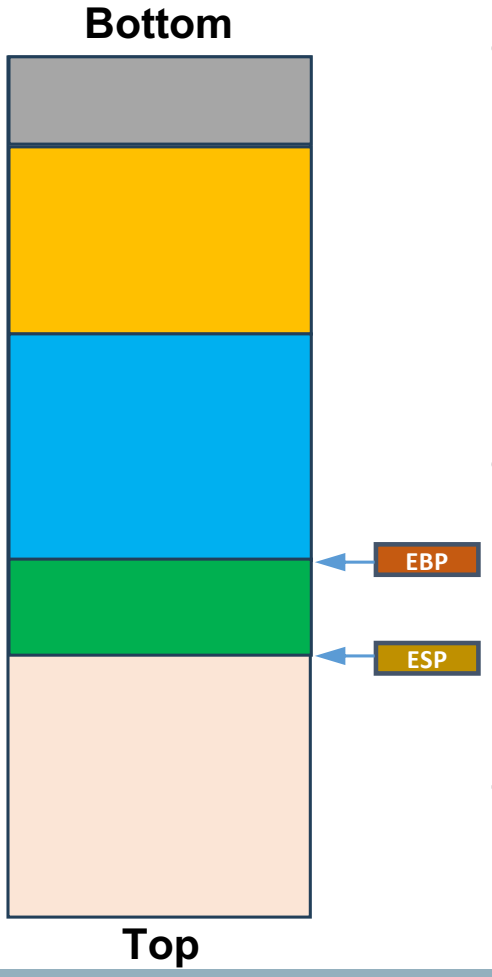
Stack Frames in IA32 (4/7)

```
L1: main () {

L2:   foo();

L3: }
```

```
L4: foo () {

L5:   bar(100, 200);

L6: }
```

```
L7: bar(int x1, int x2) {

L8:   int b1=x1,b2=x2, b3;

L9:   b3 = baz();

L10: }
```

```
L11: int baz () {

L12:    int r = 10;
L13:    return r;

L14: }
```



- EBP register points to bottom of the stack (first item pushed on the stack) and ESP register points to the top of the stack (last item pushed on stack)
- Area of the stack between the location pointed by EBP and ESP is called stack frames for that method
- The topmost frame corresponds to the currently executing method

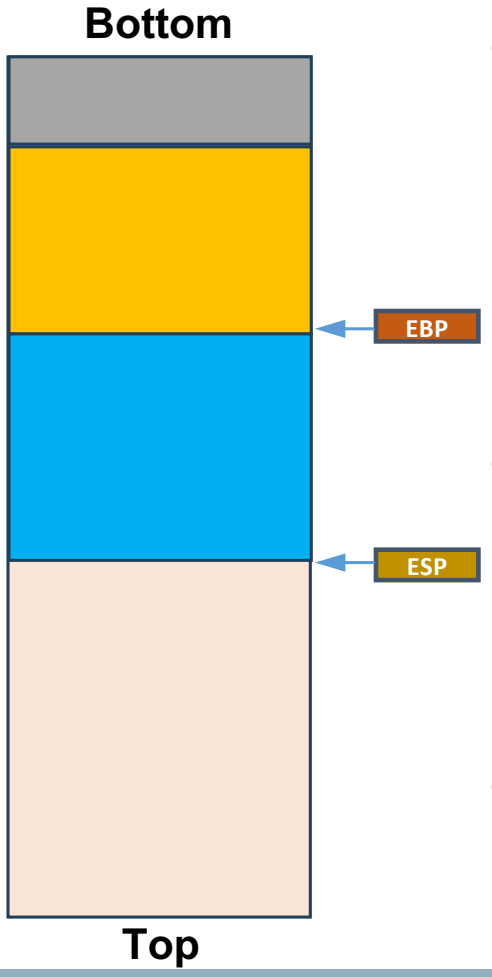
Stack Frames in IA32 (5/7)

```
L1: main () {

L2:   foo();

L3: }
```

```
L4: foo () {

L5:   bar(100, 200);

L6: }
```

```
L7: bar(int x1, int x2) {

L8:   int b1=x1,b2=x2, b3;

L9:   b3 = baz();

L10: }
```

```
L11: int baz () {

L12:    int r = 10;
L13:    return r;

L14: }
```



- EBP register points to bottom of the stack (first item pushed on the stack) and ESP register points to the top of the stack (last item pushed on stack)
- Area of the stack between the location pointed by EBP and ESP is called stack frames for that method
- The topmost frame corresponds to the currently executing method

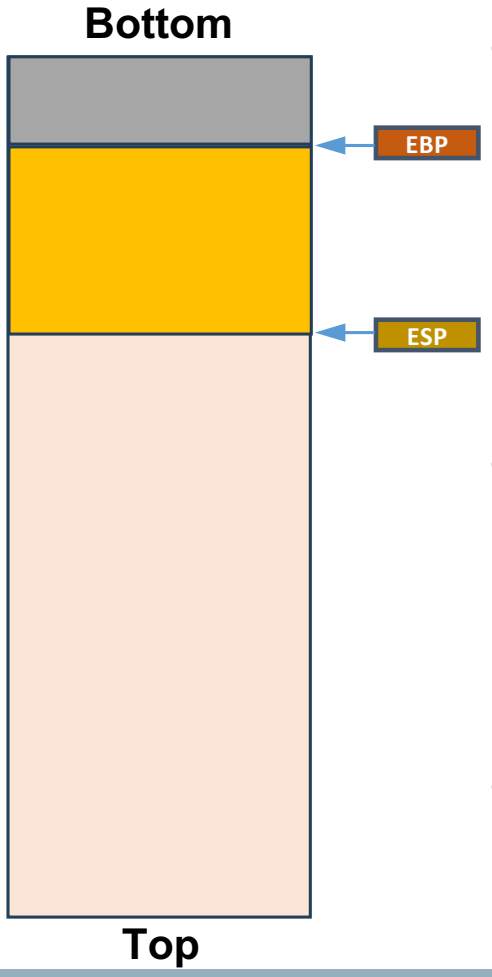
Stack Frames in IA32 (6/7)

```
L1: main () {

L2:   foo();

L3: }
```

```
L4: foo () {

L5:   bar(100, 200);

L6: }
```

```
L7: bar(int x1, int x2) {

L8:   int b1=x1,b2=x2, b3;

L9:   b3 = baz();

L10: }
```

```
L11: int baz () {

L12:    int r = 10;
L13:    return r;

L14: }
```



- EBP register points to bottom of the stack (first item pushed on the stack) and ESP register points to the top of the stack (last item pushed on stack)
- Area of the stack between the location pointed by EBP and ESP is called stack frames for that method
- The topmost frame corresponds to the currently executing method

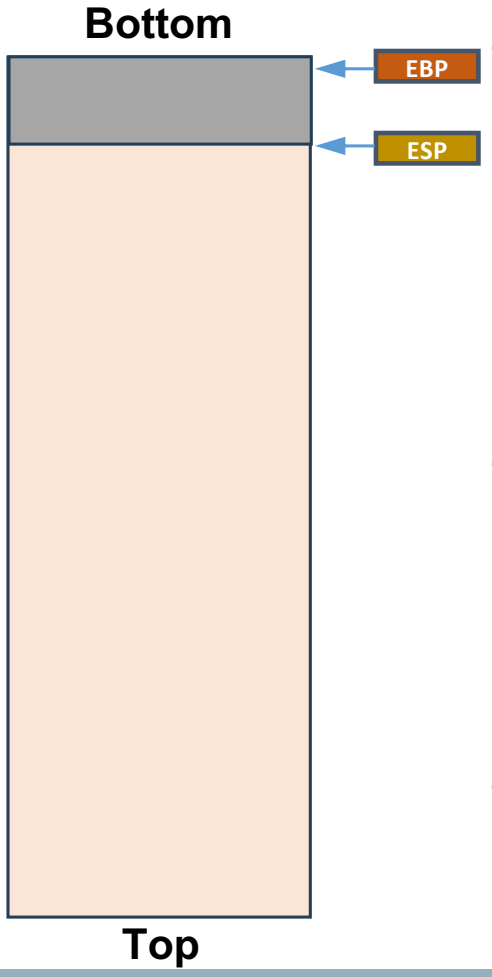
Stack Frames in IA32 (7/7)

```
L1: main () {

L2:   foo();

L3: }
```

```
L4: foo () {

L5:   bar(100, 200);

L6: }
```

```
L7: bar(int x1, int x2) {

L8:   int b1=x1,b2=x2, b3;

L9:   b3 = baz();

L10: }
```

```
L11: int baz () {

L12:    int r = 10;
L13:    return r;

L14: }
```



- EBP register points to bottom of the stack (first item pushed on the stack) and ESP register points to the top of the stack (last item pushed on stack)
- Area of the stack between the location pointed by EBP and ESP is called stack frames for that method
- The topmost frame corresponds to the currently executing method

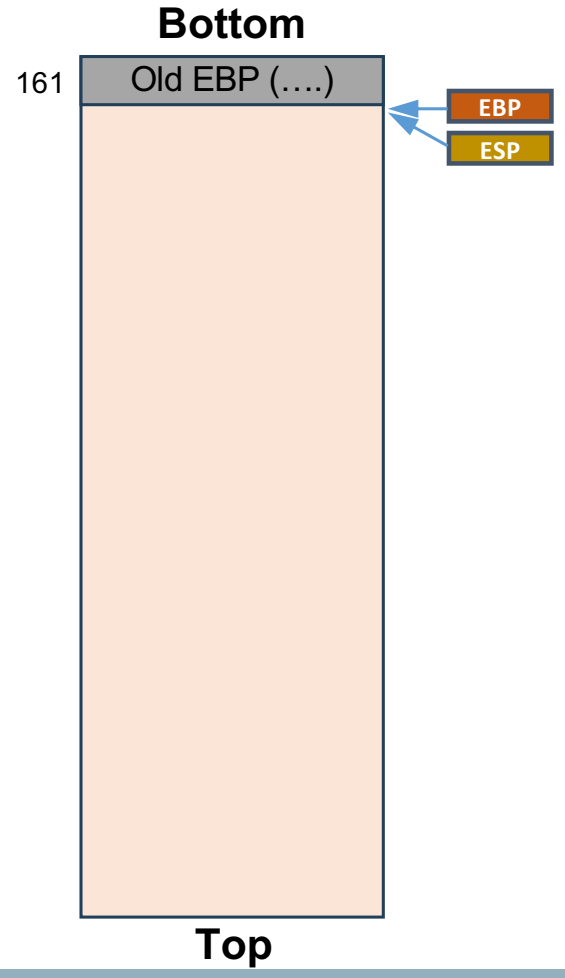
Calling Convention in X86 (1/14)

```
L1: main () {
      <save_regs>

L2:   foo();

L3: }
```

```
L4: foo () {

L5:   bar(100, 200);

L6: }
```

```
L7: bar(int x1, int x2) {

L8:   int b1=x1,b2=x2, b3;

L9:   b3 = baz();

L10: }
```

```
L11: int baz () {

L12:    int r = 10;
L13:    return r;

L14: }
```



- <save regs>
  - Current content of EBP register is saved in stack
    - Which method it corresponds to?
    - Carried out before every method call
    - Also known as function prologue
  - EBP now points to the slot containing old value of EBP
  - ESP also points to the same slot

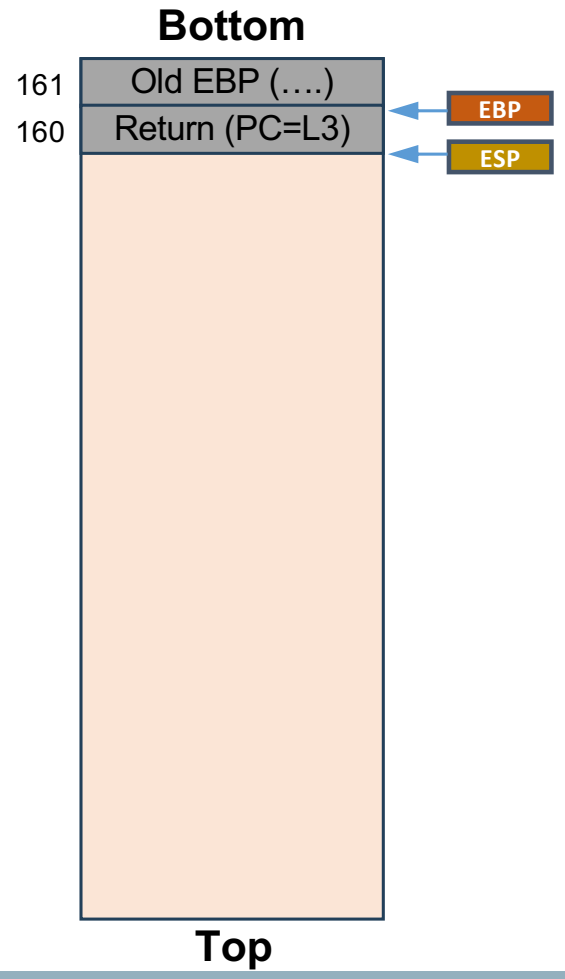
Calling Convention in X86 (2/14)

```
L1: main () {
        <save_regs>
        <save(PC)>
L2:     foo();

L3: }
```

```
L4: foo () {

L5:     bar(100, 200);

L6: }
```

```
L7: bar(int x1, int x2) {

L8:     int b1=x1,b2=x2, b3;

L9:     b3 = baz();

L10: }
```

```
L11: int baz () {

L12:    int r = 10;
L13:    return r;

L14: }
```



- <save (PC)>
  - Current content of EIP register is saved in stack
  - ESP is now pointing to newly added slot

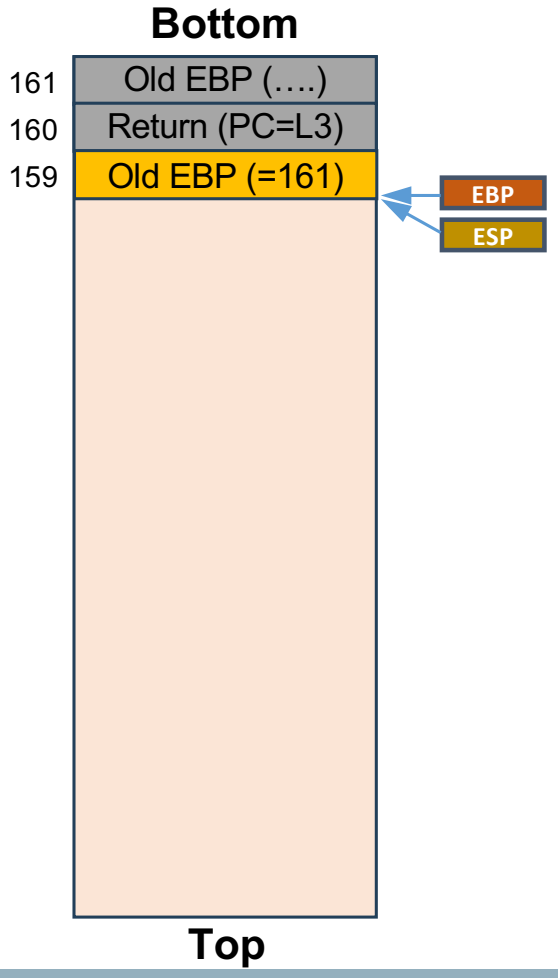
Calling Convention in X86 (3/14)

```
L1: main () {
        <save_regs>
        <save(PC)>
L2:     foo();

L3: }
```

```
L4: foo () {
        <save_regs>

L5:     bar(100, 200);

L6: }
```

```
L7: bar(int x1, int x2) {

L8:     int b1=x1,b2=x2, b3;

L9:     b3 = baz();

L10: }
```

```
L11: int baz () {

L12:     int r = 10;
L13:     return r;

L14: }
```



- <save regs>
  - Current content of EBP register is saved in stack
  - EBP now points to the slot containing old value of EBP
  - ESP also points to the same slot

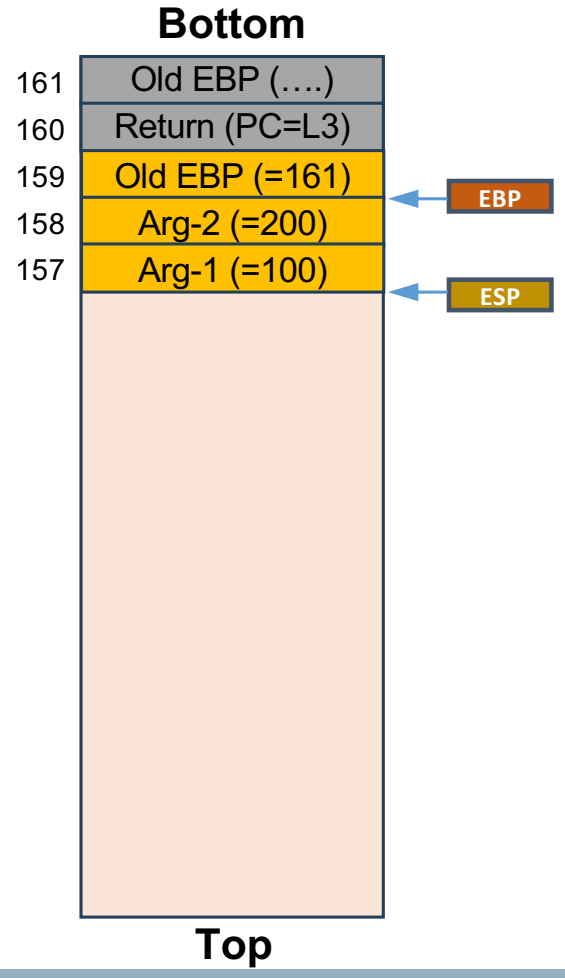
Calling Convention in X86 (4/14)

```
L1: main () {
        <save_regs>
        <save(PC)>
L2:     foo();

L3: }
```

```
L4: foo () {
        <save_regs>
        <alloc(8)>

L5:     bar(100, 200);

L6: }
```

```
L7: bar(int x1, int x2) {

L8:     int b1=x1,b2=x2, b3;

L9:     b3 = baz();

L10: }
```

```
L11: int baz () {

L12:     int r = 10;
L13:     return r;

L14: }
```



- <alloc(8)>
  - Two slots are added on the stack in the current stack frame for two arguments to the callee method
  - Arguments are stored in the reverse ordered on the stack
  - ESP is updated

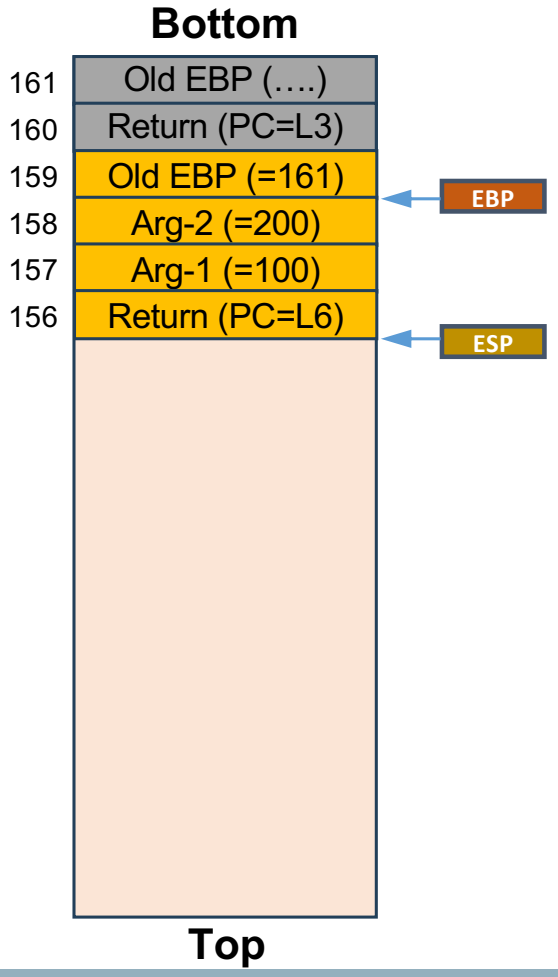
Calling Convention in X86 (5/14)

```
L1: main () {
        <save_regs>
        <save(PC)>
L2:     foo();

L3: }
```

```
L4: foo () {
        <save_regs>
        <alloc(8)>
        <save(PC)>
L5:     bar(100, 200);

L6: }
```

```
L7: bar(int x1, int x2) {

L8:     int b1=x1,b2=x2, b3;

L9:     b3 = baz();

L10: }
```

```
L11: int baz () {

L12:    int r = 10;
L13:    return r;

L14: }
```



- <save (PC)>
  - Current content of EIP register is saved in stack
  - ESP is now pointing to newly added slot

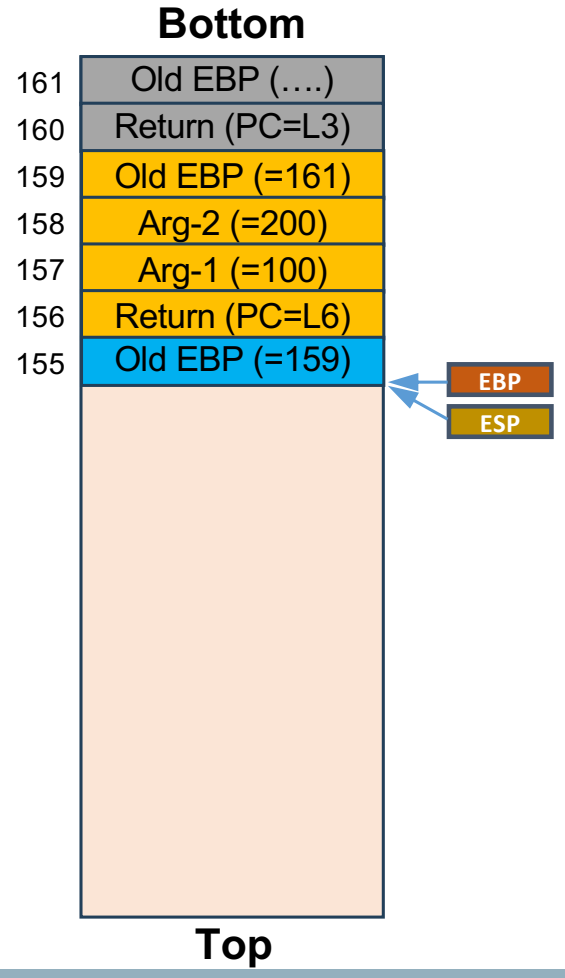
Calling Convention in X86 (6/14)

```
L1: main () {
        <save_regs>
        <save(PC)>
L2:     foo();

L3: }
```

```
L4: foo () {
        <save_regs>
        <alloc(8)>
        <save(PC)>
L5:     bar(100, 200);

L6: }
```

```
L7: bar(int x1, int x2) {
        <save_regs>

L8:     int b1=x1,b2=x2, b3;

L9:     b3 = baz();

L10: }
```

```
L11: int baz () {

L12:     int r = 10;
L13:     return r;

L14: }
```



- <save regs>
  - Current content of EBP register is saved in stack
  - EBP now points to the slot containing old value of EBP
  - ESP also points to the same slot

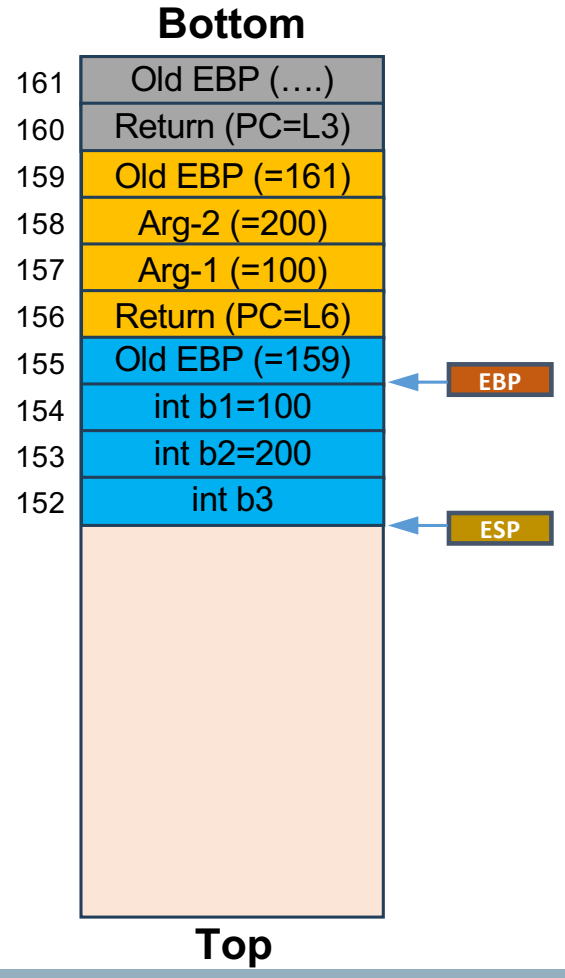
Calling Convention in X86 (7/14)

```
L1: main () {
        <save_regs>
        <save(PC)>
L2:     foo();

L3: }
```

```
L4: foo () {
        <save_regs>
        <alloc(8)>
        <save(PC)>
L5:     bar(100, 200);

L6: }
```

```
L7: bar(int x1, int x2) {
        <save_regs>
        <alloc(12)>
L8:     int b1=x1,b2=x2, b3;

L9:     b3 = baz();

L10: }
```

```
L11: int baz () {

L12:     int r = 10;
L13:     return r;

L14: }
```



- <alloc(12)>
  - Three slots are added on the stack in the current stack frame for the three local variables
  - Variables are stored in the same order on the stack as encountered
  - ESP is updated

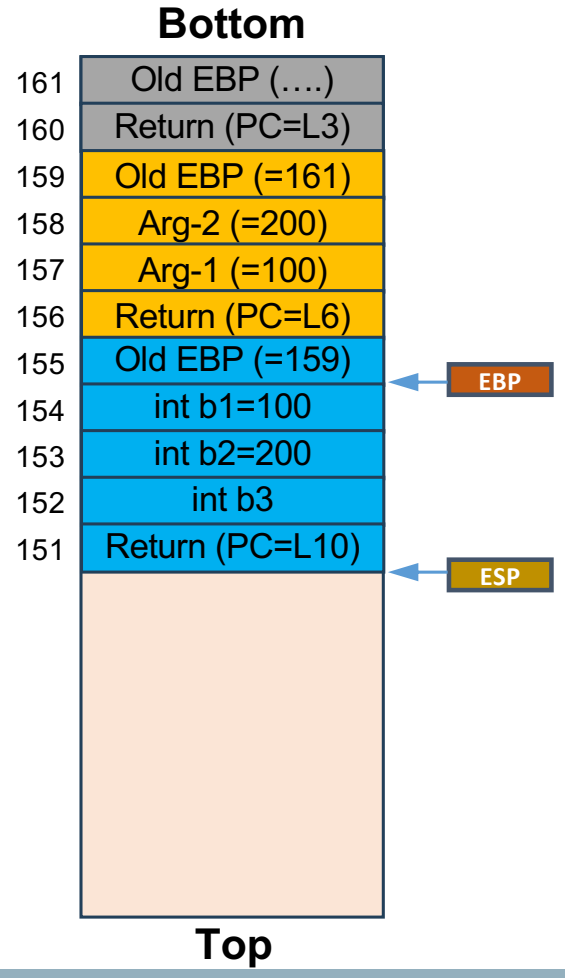
Calling Convention in X86 (8/14)

```
L1: main () {
        <save_regs>
        <save(PC)>
L2:     foo();

L3: }
```

```
L4: foo () {
        <save_regs>
        <alloc(8)>
        <save(PC)>
L5:     bar(100, 200);

L6: }
```

```
L7: bar(int x1, int x2) {
        <save_regs>
        <alloc(12)>
L8:     int b1=x1,b2=x2, b3;
        <save(PC)>
L9:     b3 = baz();

L10: }
```

```
L11: int baz () {

L12:    int r = 10;
L13:    return r;

L14: }
```



- <save (PC)>
  - Current content of EIP register is saved in stack
  - ESP is now pointing to newly added slot

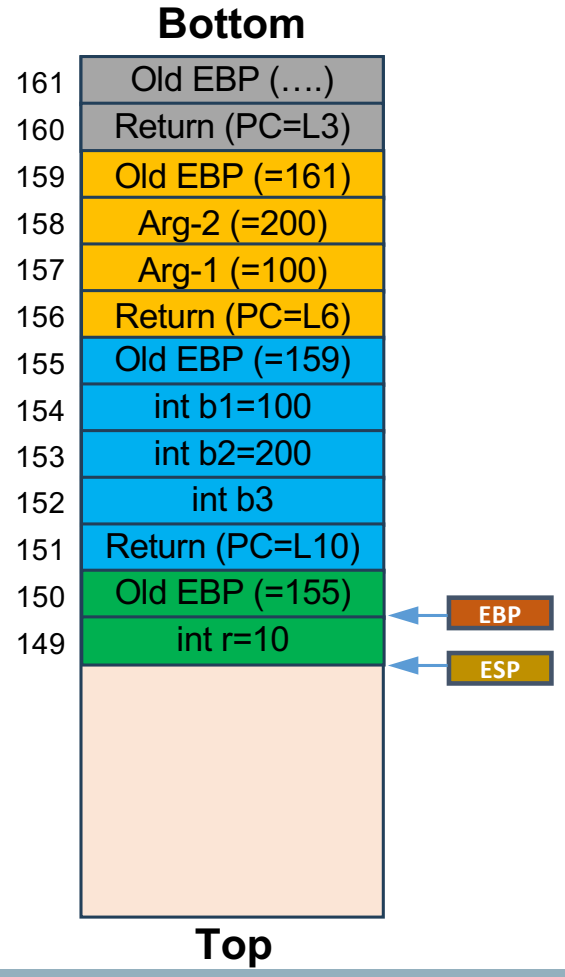
Calling Convention in X86 (9/14)

```
L1: main () {
        <save_regs>
        <save(PC)>
L2:     foo();

L3: }
```

```
L4: foo () {
        <save_regs>
        <alloc(8)>
        <save(PC)>
L5:     bar(100, 200);

L6: }
```

```
L7: bar(int x1, int x2) {
        <save_regs>
        <alloc(12)>
L8:     int b1=x1,b2=x2, b3;
        <save(PC)>
L9:     b3 = baz();

L10: }
```

```
L11: int baz () {
        <save_regs>
        <alloc(4)>
L12:    int r = 10;
L13:    return r;

L14: }
```



- <save regs>
  - Current content of EBP register is saved in stack
  - EBP now points to the slot containing old value of EBP
  - ESP also points to the same slot
- <alloc(4)
  - A slot is added on the stack in the current stack frame for the local variable

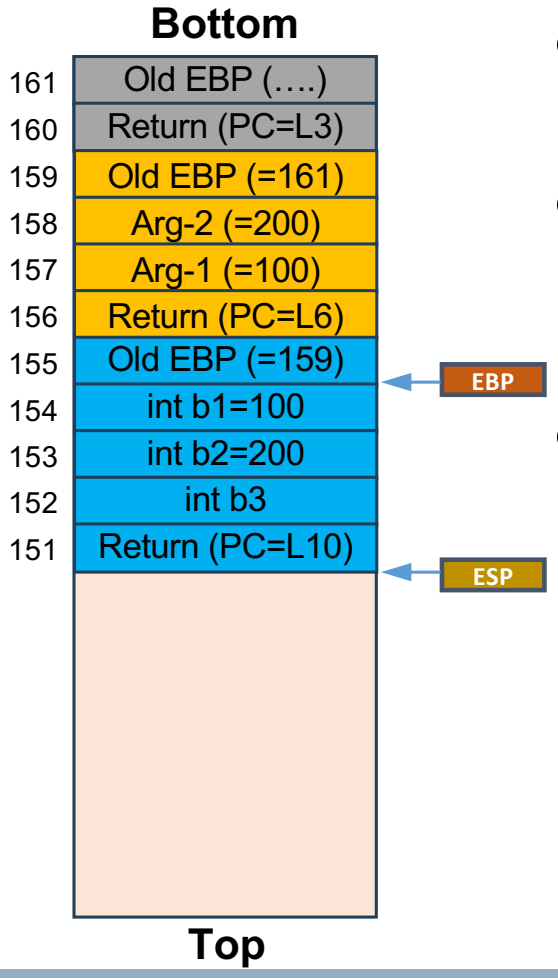
Calling Convention in X86 (10/14)

```
L1: main () {
        <save_regs>
        <save(PC)>
L2:     foo();

L3: }
```

```
L4: foo () {
        <save_regs>
        <alloc(8)>
        <save(PC)>
L5:     bar(100, 200);

L6: }
```

```
L7: bar(int x1, int x2) {
        <save_regs>
        <alloc(12)>
L8:     int b1=x1,b2=x2, b3;
        <save(PC)>
L9:     b3 = baz();

L10: }
```

```
L11: int baz () {
        <save_regs>
        <alloc(4)>
L12:    int r = 10;
L13:    return r;
        <dealloc(4)>
        <restore_regs>
L14: }
```



- Value returned is stored inside the slot 152 for “b3”
- <dealloc(4)
  - ESP is updated to point to the slot just before the slot(s) added for storing local variable(s)
- <restore_regs>
  - Old value of EBP is popped and stored in EBP
  - ESP adjusted to point to the last slot in the current stack frame
  - Carried out before exiting any method call
  - Also known as function epilogue

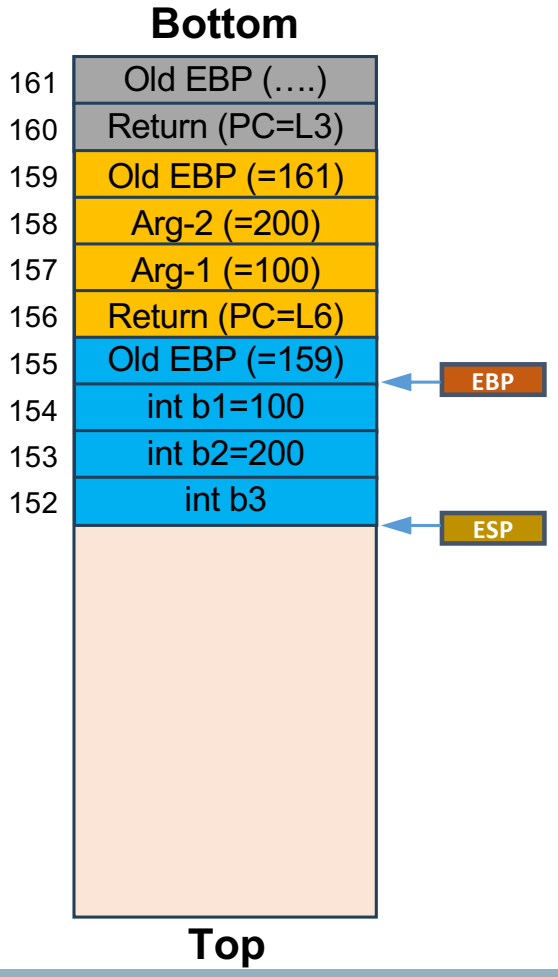
Calling Convention in X86 (11/14)

```
L1: main () {
        <save_regs>
        <save(PC)>
L2:     foo();

L3: }
```

```
L4: foo () {
        <save_regs>
        <alloc(8)>
        <save(PC)>
L5:     bar(100, 200);

L6: }
```

```
L7: bar(int x1, int x2) {
        <save_regs>
        <alloc(12)>
L8:     int b1=x1,b2=x2, b3;
        <save(PC)>
L9:     b3 = baz();
        <restore(PC)>

L10: }
```

```
L11: int baz () {
        <save_regs>
        <alloc(4)>
L12:    int r = 10;
L13:    return r;
        <dealloc(4)>
        <restore_regs>
L14: }
```



- <restore(PC)>
  - EIP register is now restored with the old value of EIP before the call went inside the callee

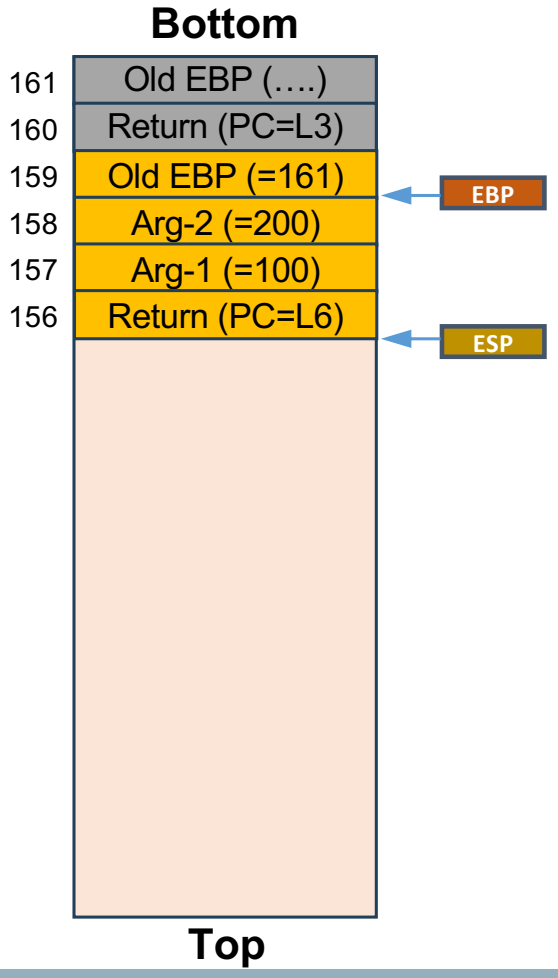
Calling Convention in X86 (12/14)

```
L1: main () {
        <save_regs>
        <save(PC)>
L2:     foo();

L3: }
```

```
L4: foo () {
        <save_regs>
        <alloc(8)>
        <save(PC)>
L5:     bar(100, 200);

L6: }
```

```
L7: bar(int x1, int x2) {
        <save_regs>
        <alloc(12)>
L8:     int b1=x1,b2=x2, b3;
        <save(PC)>
L9:     b3 = baz();
        <restore(PC)>
        <dealloc(12)>
        <restore_regs>
L10: }
```

```
L11: int baz () {
        <save_regs>
        <alloc(4)>
L12:    int r = 10;
L13:    return r;
        <dealloc(4)>
        <restore_regs>
L14: }
```



- <dealloc(12)>
  - ESP is updated to point to the slot just before the slot(s) added for storing local variable(s)
- <restore_regs>
  - Old value of EBP is popped and stored in EBP
  - ESP adjusted to point to the last slot in the current stack frame

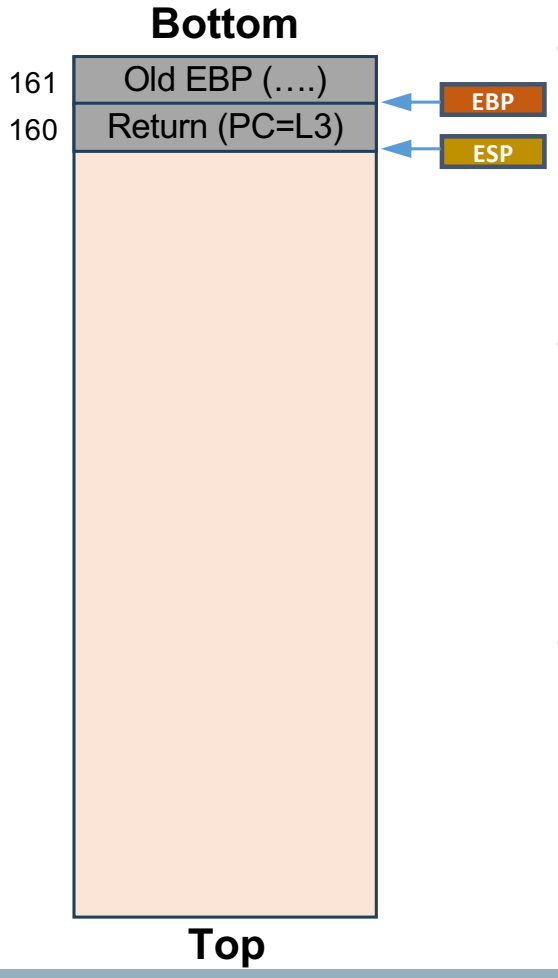
Calling Convention in X86 (13/14)

```
L1: main () {
      <save_regs>
      <save(PC)>
L2:   foo();

L3: }
```

```
L4: foo () {
      <save_regs>
      <alloc(8)>
      <save(PC)>
L5:   bar(100, 200);
      <restore(PC)>
      <dealloc(8)>
      <restore_regs>
L6: }
```

```
L7: bar(int x1, int x2) {
      <save_regs>
      <alloc(12)>
L8:   int b1=x1,b2=x2, b3;
      <save(PC)>
L9:   b3 = baz();
      <restore(PC)>
      <dealloc(12)>
      <restore_regs>
L10: }
```

```
L11: int baz () {
       <save_regs>
       <alloc(4)>
L12:   int r = 10;
L13:   return r;
       <dealloc(4)>
       <restore_regs>
L14: }
```



- <restore(PC)>
  - EIP register is now restored with the old value of EIP before the call went inside the callee
- <dealloc(8)>
  - ESP is updated to point to the slot just before the slot(s) added for storing local variable(s)
- <restore_regs>
  - Old value of EBP is popped and stored in EBP
  - ESP adjusted to point to the last slot in the current stack frame

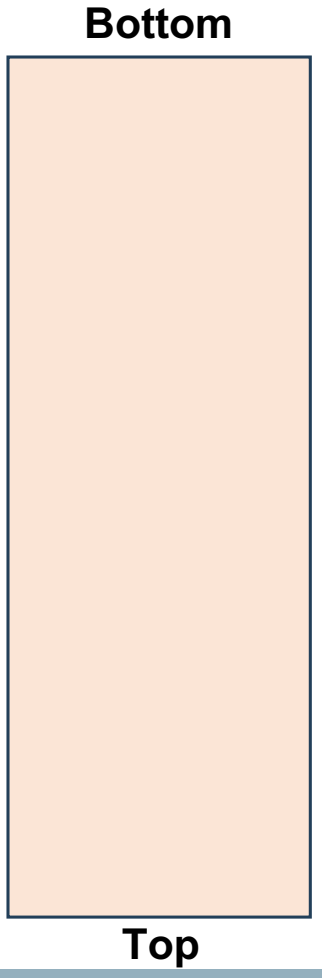
Calling Convention in X86 (14/14)

```
L1: main () {
        <save_regs>
        <save(PC)>
L2:     foo();
        <restore(PC)>
        <restore_regs>
L3: }
```

```
L4: foo () {
        <save_regs>
        <alloc(8)>
        <save(PC)>
L5:     bar(100, 200);
        <restore(PC)>
        <dealloc(8)>
        <restore_regs>
L6: }
```

```
L7: bar(int x1, int x2) {
        <save_regs>
        <alloc(12)>
L8:     int b1=x1,b2=x2, b3;
        <save(PC)>
L9:     b3 = baz();
        <restore(PC)>
        <dealloc(12)>
        <restore_regs>
L10: }
```

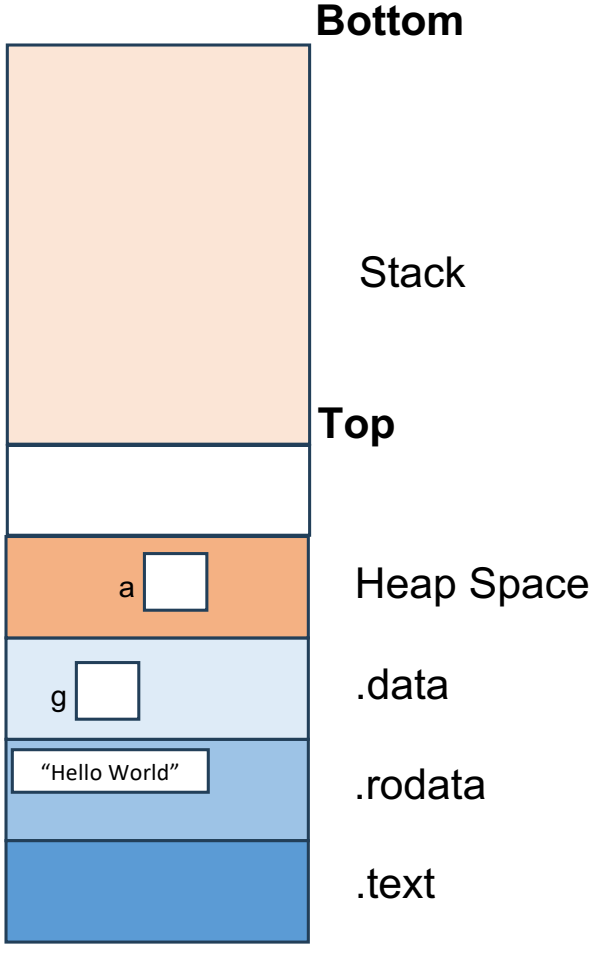
```
L11: int baz () {
        <save_regs>
        <alloc(4)>
L12:    int r = 10;
L13:    return r;
        <dealloc(4)>
        <restore_regs>
L14: }
```



- As the main method terminates, call is transferred to the caller that happens to be the libc main and then finally to the _start method

Program Execution: Class Exercise

```
L1: int g=0;

L2: void main() {
L3:     int *a = (int*) malloc(4);
L4:     char *b = “Hello World”;
L5:     foo(a);
L6:     g=*a;
L7: }

L8: void foo(int* a) {
L9:     *a = 20;
L10:}
```



Caller and Callee Save Registers

- Recall, there are six general purpose registers that can be used to store temporary data during a method execution
- If the caller is using these registers, it must save each of these 6 registers before transferring the call to the callee
- However, it could be possible that callee doesn't use all these registers. Hence, the saving and restoring of these 6 registers is divided across both caller and the callee
- If the caller method is using these data registers then it saves/restore only these 3 on its call stack before/after method call: **EAX**, **EDX**, and **ECX**
- If the callee method wants to use these 3 registers then it saves/restore the current value on its call stack at entry/exit: **EBX**, **ESI**, and **EDI**

x86-64 Calling Convention

- Different than IA32 (x86)
- There are 16 general purpose 64-bit registers compared to 8, 32-bit registers on x86. Hence, less dependence on stack
  - As example, the arguments to callee and temporary variables can be stored on registers
  - If more data than registers then fall back to x86 way of storing on stack
- Execution is much faster due to less dependence on stack

Next Lecture

- The process abstraction
- Quiz-1 during lecture hours

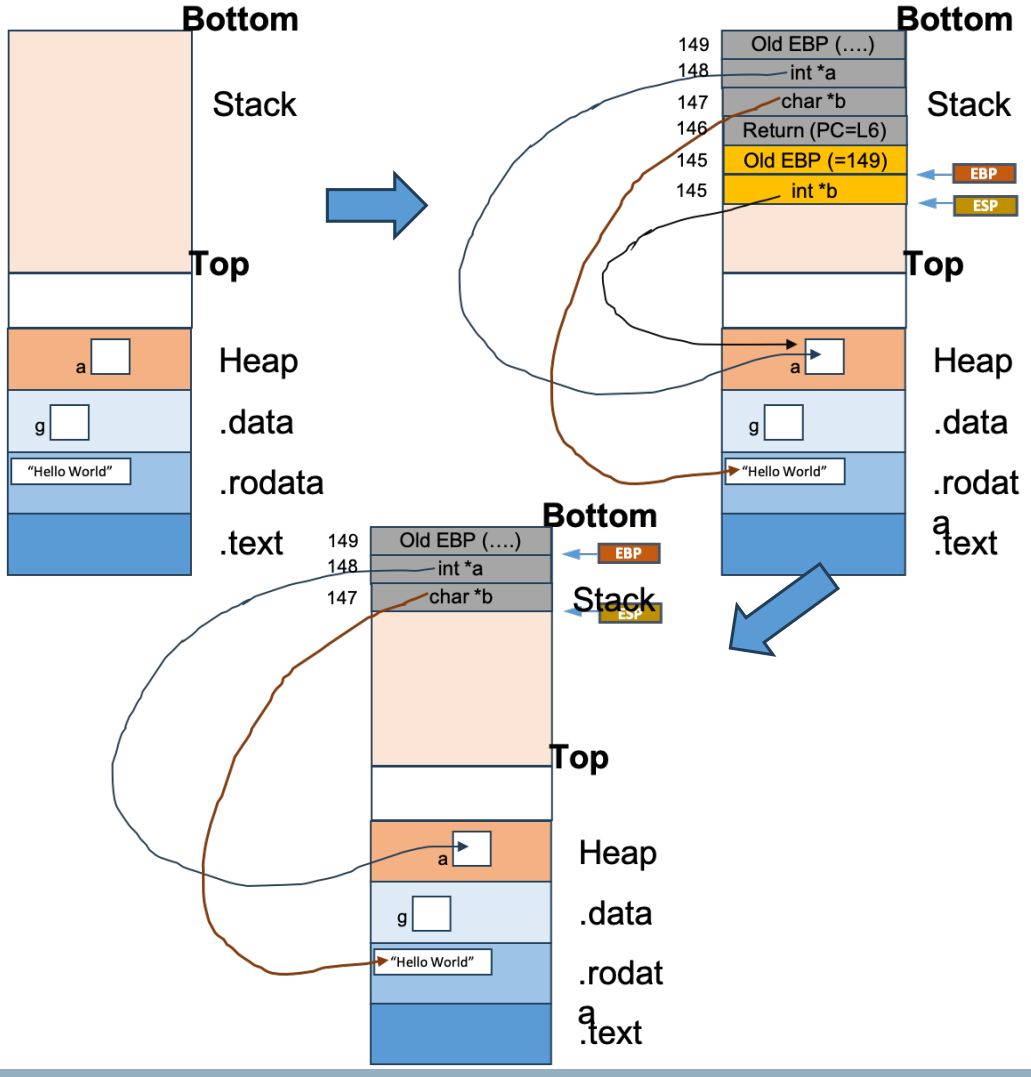
Lecture 05: The Process Abstraction

Vivek Kumar
Computer Science and Engineering
IIIT Delhi
vivekk@iiitd.ac.in

Last Lecture

```
L1: int g=0;

L2: void main() {
L3:     int *a = (int*) malloc(4);
L4:     char *b = "Hello World";
L5:     foo(a);
L6:     g=*a;
L7: }

L8: void foo(int* b) {
L9:     *b = 20;
L10:}
```

```
Program Headers:
  Type           Offset   VirtAddr   PhysAddr   FileSiz MemSiz  Flg Align
  PHDR           0x000034 0x08048034 0x08048034 0x00120 0x00120 R   0x4
  INTERP         0x000154 0x08048154 0x08048154 0x00013 0x00013 R   0x1
      [Requesting program interpreter: /lib/ld-linux.so.2]
  LOAD           0x000000 0x08048000 0x08048000 0x006a8 0x006a8 R E 0x1000
  LOAD           0x000f0c 0x08049f0c 0x08049f0c 0x00110 0x00118 RW  0x1000
  DYNAMIC        0x000f14 0x08049f14 0x08049f14 0x000e8 0x000e8 RW  0x4
  NOTE           0x000168 0x08048168 0x08048168 0x00044 0x00044 R   0x4
  GNU_EH_FRAME   0x00052c 0x0804852c 0x0804852c 0x0004c 0x0004c R   0x4
  GNU_STACK      0x000000 0x00000000 0x00000000 0x00000 0x00000 RW  0x10
  GNU_RELRO      0x000f0c 0x08049f0c 0x08049f0c 0x000f4 0x000f4 R   0x1

 Section to Segment mapping:
  Segment Sections...
   00
   01     .interp
   02     .interp .note.ABI-tag .note.gnu.build-id .gnu.hash .dynsym .dynstr
 .gnu.version .gnu.version_r .rel.dyn .rel.plt .init .plt .plt.got .text .fi
ni .rodata .eh_frame_hdr .eh_frame
   03     .init_array .fini_array .dynamic .got .got.plt .data .bss
   04     .dynamic
```



Today's Class

- The process abstraction
- Quiz-1 (Lectures 02-04)

All Executables are ELF

```
iiitd@possum:~$ file /bin/cat
/bin/cat: ELF 64-bit LSB shared object, x86-64, version 1 (SYSV), dynamicall
y linked, interpreter /lib64/ld-linux-x86-64.so.2, for GNU/Linux 3.2.0, Buil
dID[sha1]=747e524bc20d33ce25ed4aea108e3025e5c3b78f, stripped
iiitd@possum:~$ file /bin/ls
/bin/ls: ELF 64-bit LSB shared object, x86-64, version 1 (SYSV), dynamically
 linked, interpreter /lib64/ld-linux-x86-64.so.2, for GNU/Linux 3.2.0, Build
ID[sha1]=9567f9a28e66f4d7ec4baf31cfbf68d0410f0ae6, stripped
iiitd@possum:~$ file /bin/mkdir
/bin/mkdir: ELF 64-bit LSB shared object, x86-64, version 1 (SYSV), dynamica
lly linked, interpreter /lib64/ld-linux-x86-64.so.2, for GNU/Linux 3.2.0, Bu
ildID[sha1]=6c825e92c5eae304845d070ed34749495d67c566, stripped
```

The Process

```
iiitd@possum:~$ vi fib.c
iiitd@possum:~$ gcc fib.c
iiitd@possum:~$ ./a.out
Fib(40) = 102334155
```

PID	USER	PRI	NI	VIRT	RES	SHR	S	CPU%	MEM%	TIME+	Command
2034	iiitd	20	0	4384	820	756	R	100.	0.0	0:17.48	./a.out

- Whenever we type and enter a command on shell, the command prompt is returned back only after that command has completed its execution
  - How this execution happens?
- A program under execution is called a process
  - Its again all about abstraction. First we discussed about ELF to have an abstraction and presenting a unified view of all object files for the loader. Now, we are going one level above where the OS needs to have an abstract representation of any kind of program under execution
  - Process is a collection of resources

There are Several Processes

```
iiitd@possum:~$ for i in {1..20}
> do
> ./a.out &
> done
[1] 2062
[2] 2063
[3] 2064
[4] 2065
[5] 2066
[6] 2067
[7] 2068
[8] 2069
[9] 2070
[10] 2071
[11] 2072
[12] 2073
[13] 2074
[14] 2075
[15] 2076
[16] 2077
[17] 2078
[18] 2079
[19] 2080
[20] 2081
```

```
1  ||||||||||||||||||||||||||||||||100.0%    Tasks: 104, 202 thr; 4
2  ||||||||||||||||||||||||||||||||100.0%    Load average:      2.36
3  ||||||||||||||||||||||||||||||||100.0%    Uptime: 00:09:59
4  ||||||||||||||||||||||||||||||||100.0%
Mem |||||||||||||                 470M/3.75G
Swp                                0K/2.00G

 PID USER      PRI  NI  VIRT   RES   SHR S CPU% MEM%   TIME+  Command
2074 iiitd      20   0  4384   804   740 R 20.5  0.0  0:06.97 ./a.out
2076 iiitd      20   0  4384   768   704 R 20.5  0.0  0:06.96 ./a.out
2080 iiitd      20   0  4384   792   728 R 20.5  0.0  0:06.96 ./a.out
2068 iiitd      20   0  4384   712   648 R 20.5  0.0  0:06.96 ./a.out
2066 iiitd      20   0  4384   820   756 R 19.9  0.0  0:06.97 ./a.out
2064 iiitd      20   0  4384   708   648 R 19.9  0.0  0:06.97 ./a.out
2072 iiitd      20   0  4384   756   692 R 19.9  0.0  0:06.96 ./a.out
2063 iiitd      20   0  4384   800   740 R 19.9  0.0  0:06.96 ./a.out
2069 iiitd      20   0  4384   720   656 R 19.9  0.0  0:06.96 ./a.out
2078 iiitd      20   0  4384   864   800 R 19.9  0.0  0:06.98 ./a.out
2070 iiitd      20   0  4384   804   740 R 19.9  0.0  0:06.98 ./a.out
2075 iiitd      20   0  4384   752   692 R 19.9  0.0  0:06.96 ./a.out
2073 iiitd      20   0  4384   764   700 R 19.9  0.0  0:06.98 ./a.out
2065 iiitd      20   0  4384   804   740 R 19.9  0.0  0:06.98 ./a.out
2077 iiitd      20   0  4384   712   648 R 19.9  0.0  0:06.93 ./a.out
2071 iiitd      20   0  4384   752   692 R 19.9  0.0  0:06.94 ./a.out
2081 iiitd      20   0  4384   756   692 R 19.9  0.0  0:06.93 ./a.out
2079 iiitd      20   0  4384   720   656 R 19.9  0.0  0:06.94 ./a.out
2067 iiitd      20   0  4384   716   656 R 19.9  0.0  0:06.93 ./a.out
2062 iiitd      20   0  4384   800   740 R 19.9  0.0  0:06.94 ./a.out
2021 iiitd      20   0 33784  4536  3612 R  0.0  0.1  0:00.68 htop
```

```
iiitd@possum:~$ lscpu
Architecture:          x86_64
CPU op-mode(s):        32-bit, 64-bit
Byte Order:            Little Endian
CPU(s):                4
On-line CPU(s) list:   0-3
Thread(s) per core:    1
Core(s) per socket:    4
```

- Note the "R"(running) state of the program in "S"(state) column in htop output
- There are only four CPUs in this system, but how is the OS able to run 20 a.out simultaneously?
- Virtualization of the CPU!
  - Any limit?

```
iiitd@possum:~$ cat /proc/sys/kernel/pid_max
32768
```

Process Keep Changing its State

```
iiitd@possum:~$ ./a.out
Enter input..
40
Fib(40) = 102334155
Enter input..
50
```

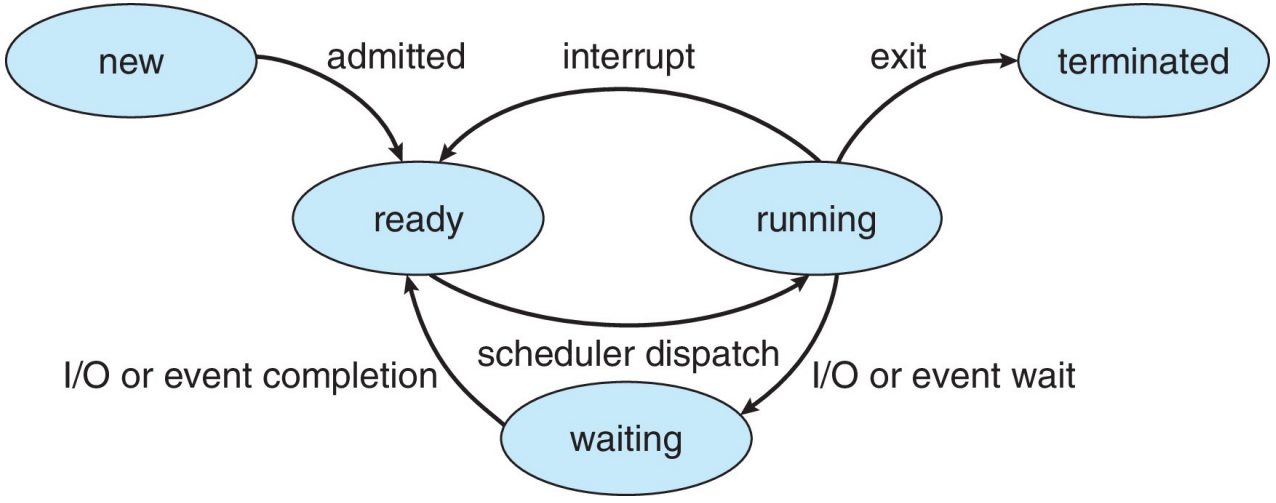
PID	USER	PRI	NI	VIRT	RES	SHR	S	CPU%	MEM%	TIME+	Command
2327	iiitd	20	0	4516	760	700	S	0.0	0.0	1:15.50	./a.out

PID	USER	PRI	NI	VIRT	RES	SHR	S	CPU%	MEM%	TIME+	Command
2327	iiitd	20	0	4516	760	700	R	99.5	0.0	1:21.13	./a.out

PID	USER	PRI	NI	VIRT	RES	SHR	S	CPU%	MEM%	TIME+	Command
2327	iiitd	20	0	4516	760	700	S	0.0	0.0	1:15.50	./a.out

- There are two distinct phases in the above program
  - Asking the user for the input
    - The process representing the a.out is moved into “S” state
      - It got kicked out of the CPU
  - Calculating the Fibonacci number of that input
    - The process representing the a.out is moved into “R” state
      - It got its CPU back!
  - Why not simply leave the process occupying the CPU?

Process States



- As a process executes, it changes state
  - **New**: The process is being created
  - **Running**: Instructions are being executed
  - **Waiting**: The process is waiting for some event to occur
  - **Ready**: The process is waiting to be assigned to a processor
  - **Terminated**: The process has finished execution

Source: https://os-book.com/OS10/slide-dir/index.html

Process State: Example

Time	$Process_0$	$Process_1$	Notes
1	Running	Ready	
2	Running	Ready	
3	Running	Ready	$Process_0$ initiates I/O
4	Blocked	Running	$Process_0$ is blocked,
5	Blocked	Running	so $Process_1$ runs
6	Blocked	Running	
7	Ready	Running	I/O done
8	Ready	Running	$Process_1$ now done
9	Running	–	
10	Running	–	$Process_0$ now done

Figure 4.4: **Tracing Process State: CPU and I/O**

What Constitutes a Process

- There are so many processes at any given time
  - Each process has its state
  - A running process will be using some CPU registers
    - We push and pop the registers on call stack during the course of execution
  - Process requires memory for execution (stack and heap)
  - Process have several open file handles
- For process management, the OS uses a data-structure called as “process descriptor” or “process control block” to keep track of every process’s progress and usage of the computer’s available resources

Process Descriptor (a.k.a PCB)

- Kernel maintains detail on each process inside a structure of type **task_struct**
- You can see it yourself
  - https://raw.githubusercontent.com/torvalds/linux/master/include/linux/sched.h

```
struct task_struct {
    unsigned int                    __state;
    void                            *stack;
    int                             prio;
    int                             nr_cpus_allowed;
    struct mm_struct                *mm;
    struct mm_struct                *active_mm;
    pid_t                           pid;
    struct list_head                children;
    struct list_head                sibling;
    struct task_struct              *group_leader;
    /*
     * executable name, excluding path.
     *
     * - normally initialized setup_new_exec()
     * - access it with [gs]et_task_comm()
     * - lock it with task_lock()
     */
    char                            comm[TASK_COMM_LEN];

    struct nameidata                *nameidata;
    /* Filesystem information: */
    struct fs_struct                *fs;

    /* Open file information: */
    struct files_struct             *files;
```

- Extracts from the struct task_struct
  - There are many more members!
- Registers are saved on stack
- Linked list of PCB to dynamically add a new process information

```
iiitd@possum:~$ ps -u iiitd --forest
  PID TTY          TIME CMD
23005 ?        00:00:00 sshd
23006 pts/1    00:00:00  \_ bash
23062 pts/1    00:00:01      \_ a.out
22906 ?        00:00:00 sshd
22908 pts/0    00:00:00  \_ bash
23063 pts/0    00:00:00      \_ ps
22807 ?        00:00:00 systemd
22808 ?        00:00:00  \_ (sd-pam)
```

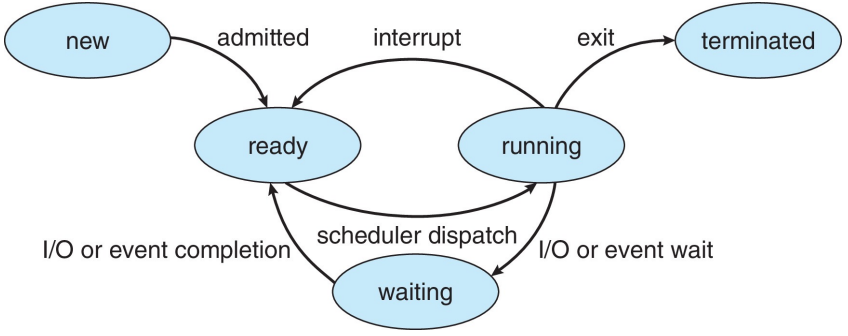
Next Lecture

- Process creation and termination

Lecture 06: Process Creation

Vivek Kumar
Computer Science and Engineering
IIIT Delhi
vivekk@iiitd.ac.in

Last Lecture



Time	$Process_0$	$Process_1$	Notes
1	Running	Ready	
2	Running	Ready	
3	Running	Ready	$Process_0$ initiates I/O
4	Blocked	Running	$Process_0$ is blocked,
5	Blocked	Running	so $Process_1$ runs
6	Blocked	Running	
7	Ready	Running	I/O done
8	Ready	Running	$Process_1$ now done
9	Running	–	
10	Running	–	$Process_0$ now done

Figure 4.4: **Tracing Process State: CPU and I/O**

```
struct task_struct {
    unsigned int                    __state;
    void                            *stack;
    int                             prio;
    int                             nr_cpus_allowed;
    struct mm_struct                *mm;
    struct mm_struct                *active_mm;
    pid_t                           pid;
    struct list_head                children;
    struct list_head                sibling;
    struct task_struct              *group_leader;
    /*
     * executable name, excluding path.
     *
     * - normally initialized setup_new_exec()
     * - access it with [gs]et_task_comm()
     * - lock it with task_lock()
     */
    char                            comm[TASK_COMM_LEN];

    struct nameidata                *nameidata;
    /* Filesystem information: */
    struct fs_struct                *fs;

    /* Open file information: */
    struct files_struct             *files;
```

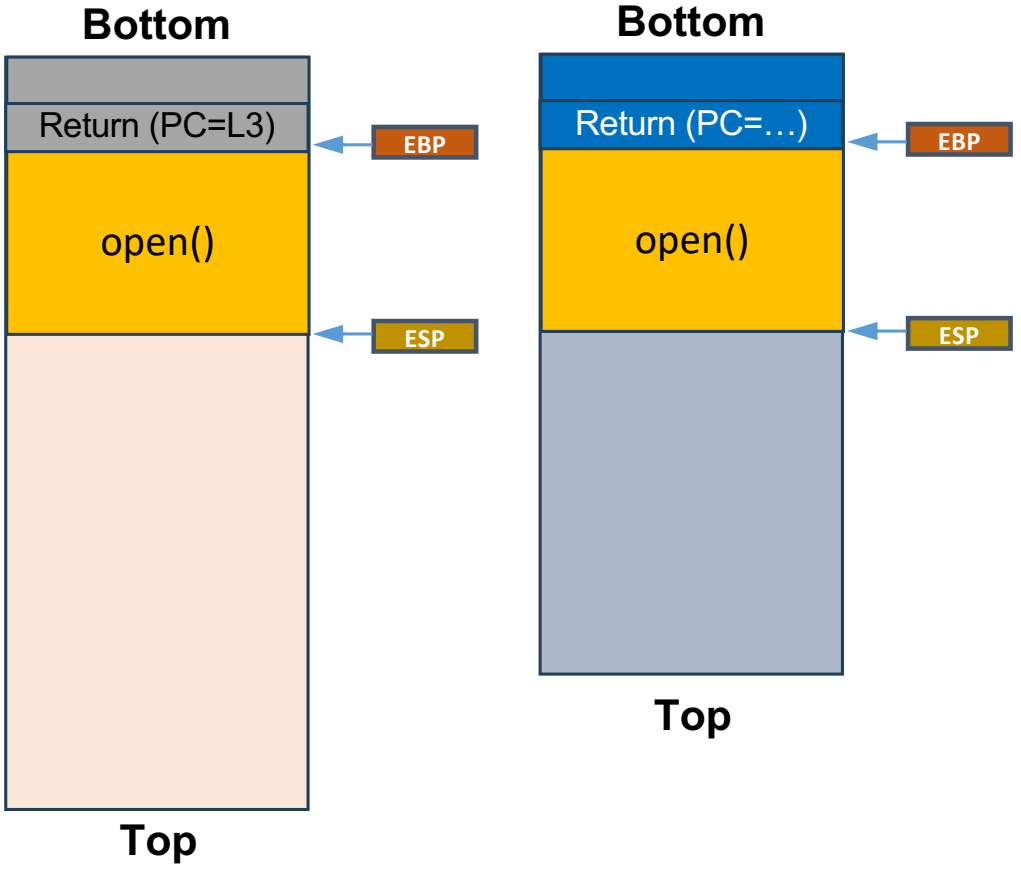
- Process is a program in execution
- Processes have a life cycle
- OS has a datastructure (PCB) to store the details of a process

Today's Class

- Unix architecture
- System calls
- Process creation and termination

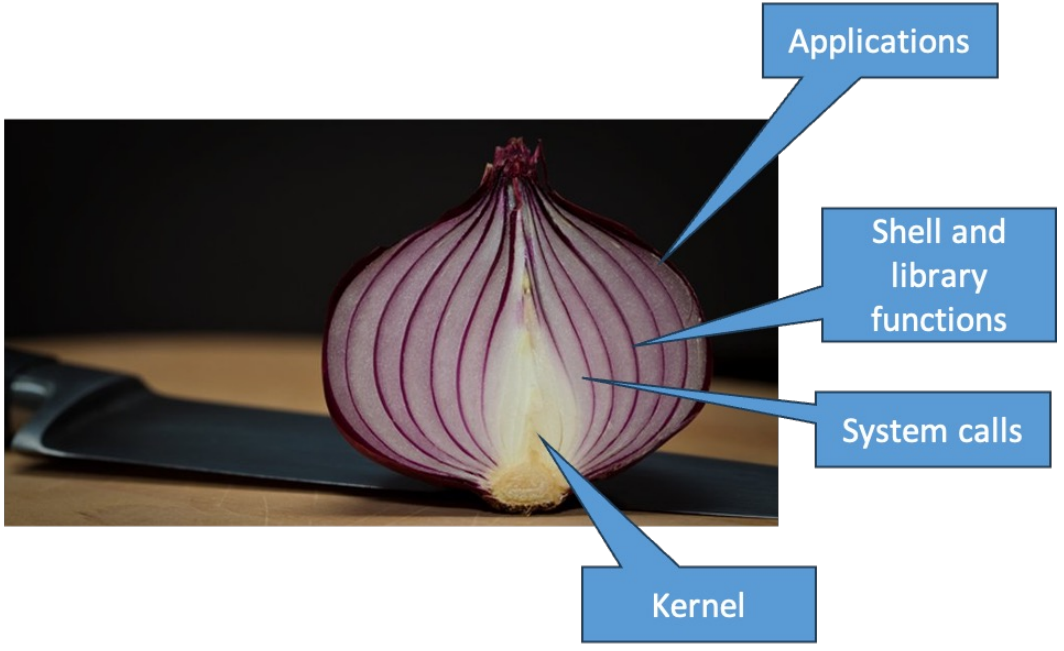
Return Address on Stack can be Modified

```
L1: main () {
      <save_regs>
      <save(PC)>
L2:   open();
      <restore(PC)>
      <restore_regs>
L3: }
```



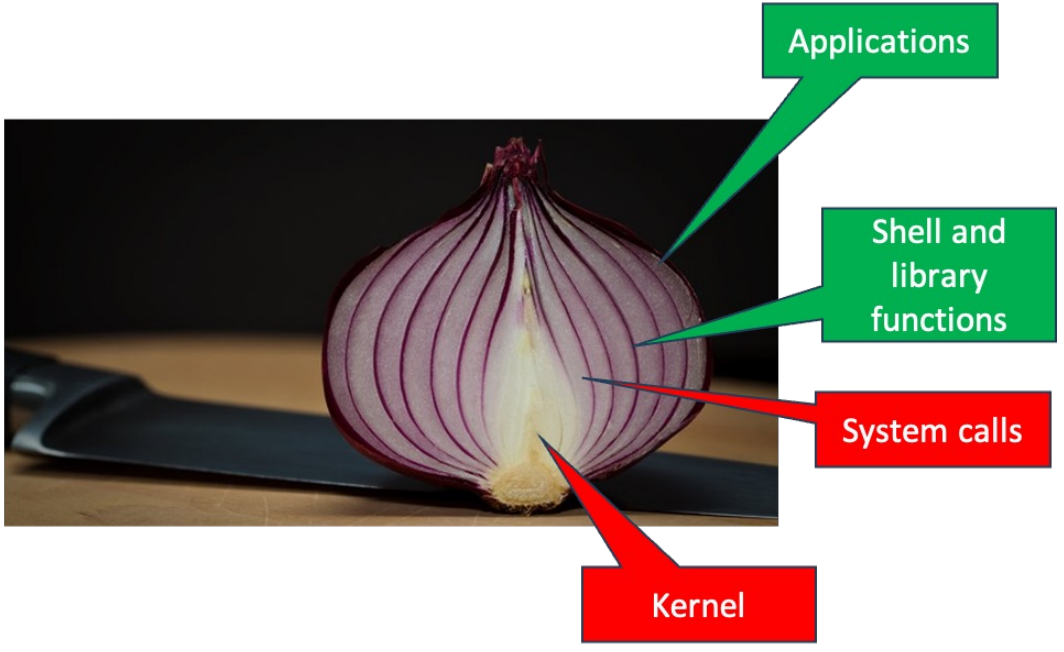
- Issues
  1. If **open**() is a regular method call then it's machine code won't be able to have a foolproof check if the calling process has the permission to access this file
     - The loader can simply patch the machine code with the call to open() at the runtime
  2. The main() called **open**(), but then **open** could be **forced** to return to some malicious process call stack

Unix Architecture



- A major goal of OS is to support portability across various architecture
- A layered approach helps in achieving this goal as the innermost layer is only one that has to interact with the hardware
  - This innermost layer is called as OS **kernel**
  - It is relatively small, controls the hardware resources, and provides an environment under which programs can run
- System call is the interface to the kernel (e.g., read, open, etc.)
- Library functions are are built on top of system calls that applications can use
  - E.g., C-library function malloc uses sbrk system call, open(), etc.
- Shell is a command line interpreter that reads user input and execute commands
  - E.g., bash, csh, etc.

Protection Rings



- Protection rings are hardware supported mechanisms used by the OS for protection of data and functionality
  - E.g., x86 supports four protection levels (0-3)
  - Level-0 is called as **kernel mode** (supervisor mode) and Level-3 is **user mode**
- Unix-like OS execute user applications, shell, and library functions in the user mode whereas kernel/syscalls in the kernel mode

Today's Class

- Unix architecture
- System calls
- Process creation and termination

Interrupts: Gateway to the Kernel Mode

- Interrupts are signals to the CPU that something special has to happen
  - INT instruction on x86 to generate an interrupt
  - Interrupts are handled only by the kernel
- Three kinds of interrupts
  - Exceptions
    - Attempt to access invalid memory address, divide by zero, etc.
      - Covered in next lecture
  - **System calls (software interrupt)** – today's focus
  - Hardware interrupt
    - Signal generated by some hardware device. E.g., disk can generate an interrupt when a block of memory has been read and is ready to be retrieved
- As per Unix terminology, "interrupts" refer to the hardware interrupts, whereas "trap" refers to software interrupts (e.g., some special type of exceptions and all **system calls**)

Steps for Making a System Call (1/5)

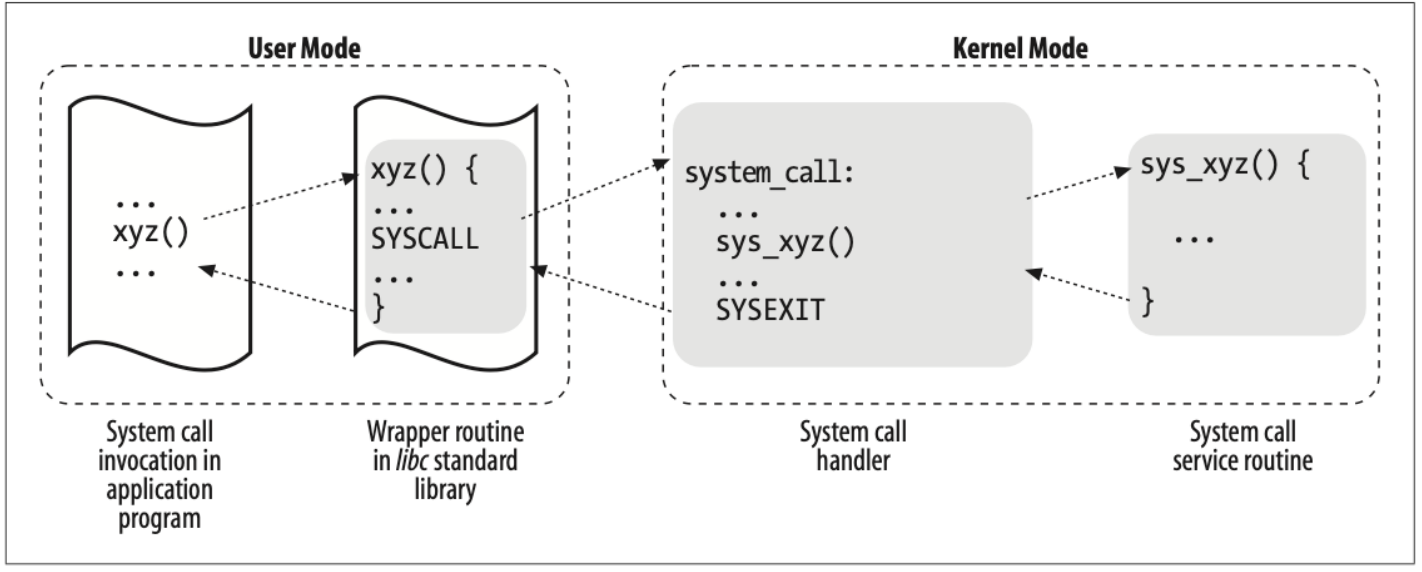
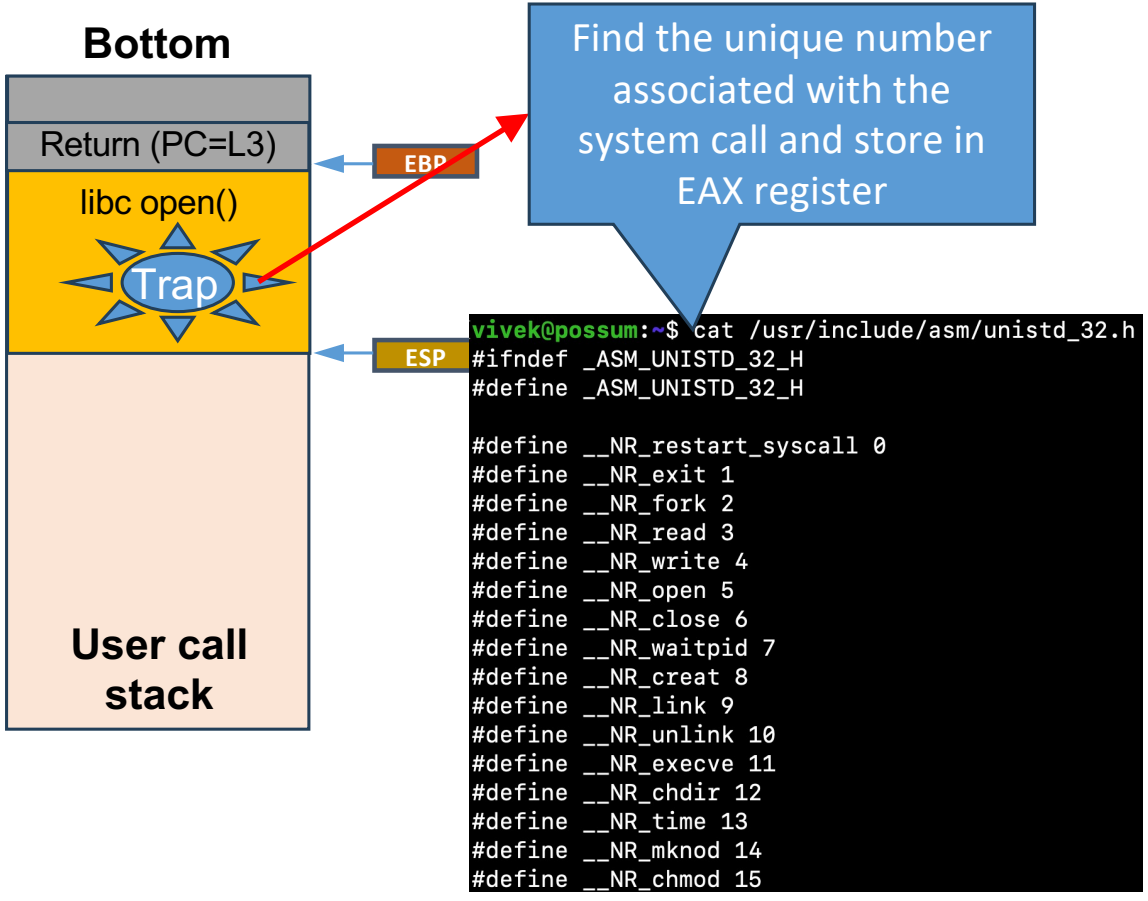


Figure 10-1. Invoking a system call

- **At a high level** the steps are as follows:
  1. Save registers in kernel mode stack
  2. Handle the system call by invoking a corresponding system call service routine
  3. Restore the registers and switch back into user mode

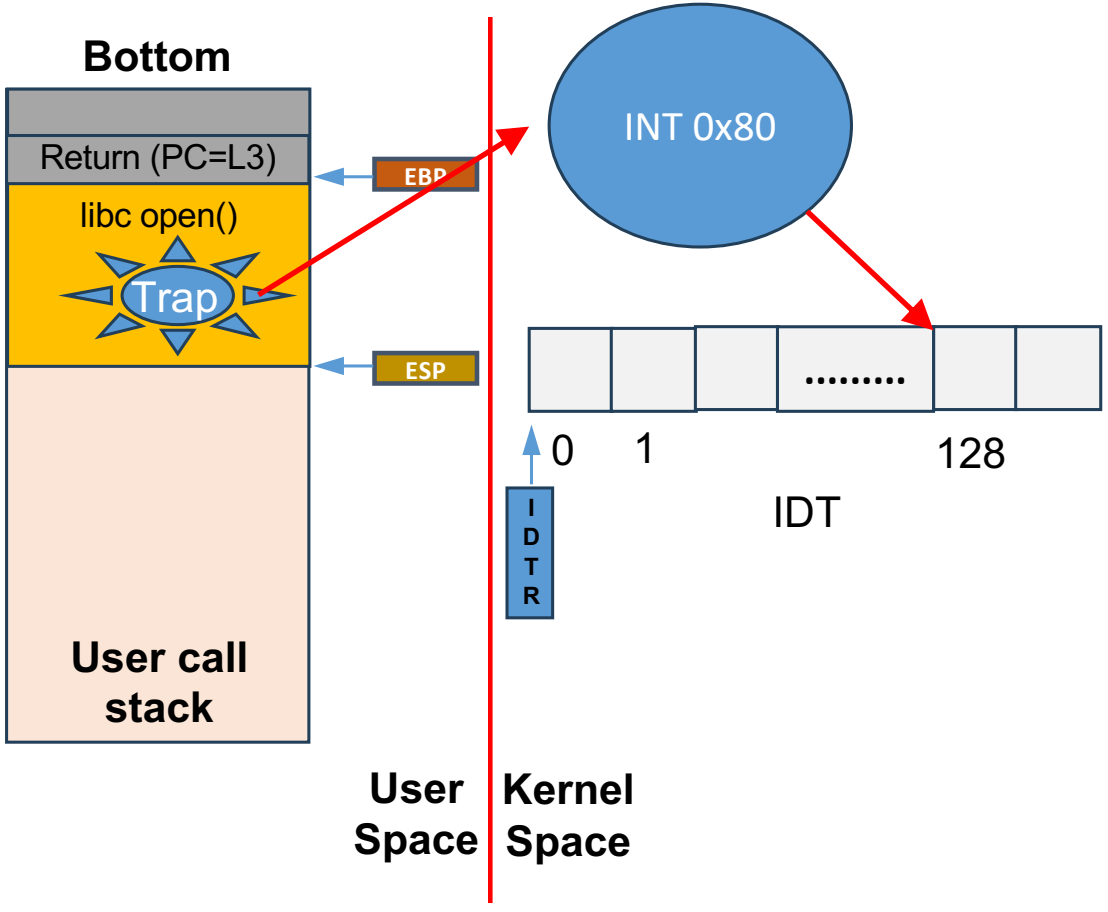
Figure source: from the book "Understanding the Linux Kernel"

Steps for Making a System Call (2/5)



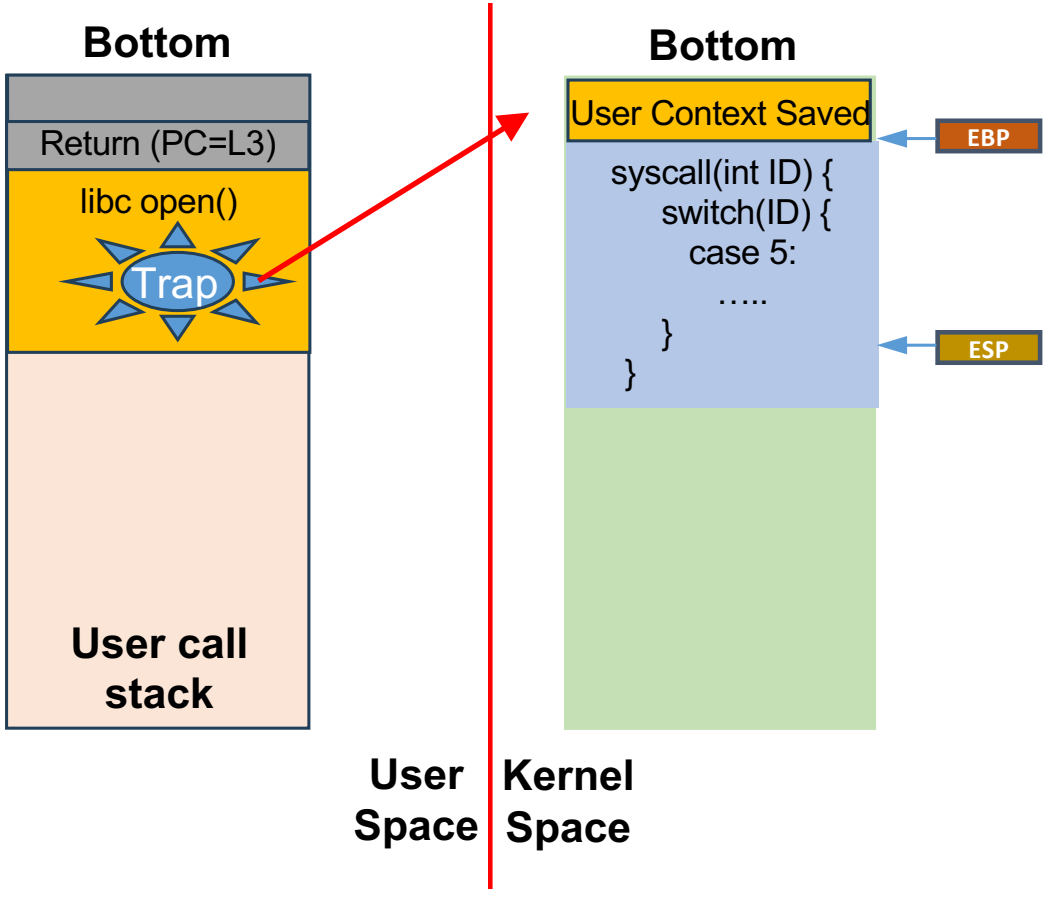
- Each system call has a unique number associated that is declared inside the file unistd.h
- Store this number in the EAX register
- Process is still executing inside user space

Steps for Making a System Call (3/5)



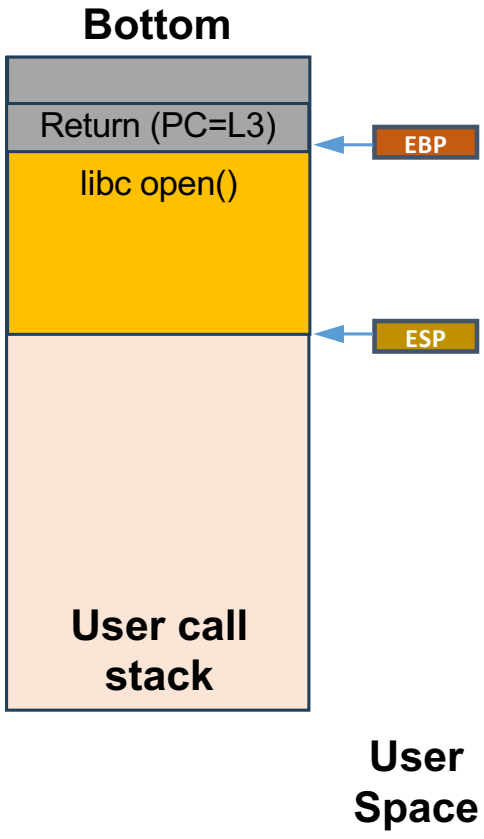
- Kernel sets up Interrupt Descriptor Table (IDT) at boot time whose address is stored in IDT register (IDTR). Total 256 entries in IDT on x86
- Each IDT index has the address of some interrupt handler
- 128th entry associated with the system call handler

Steps for Making a System Call (4/5)



- Switch to kernel stack
  - What should be the first step?
- Save registers on kernel stack. These registers are currently holding details to return back to user stack
- Look up the syscall number from EAX and process the syscall

Steps for Making a System Call (5/5)

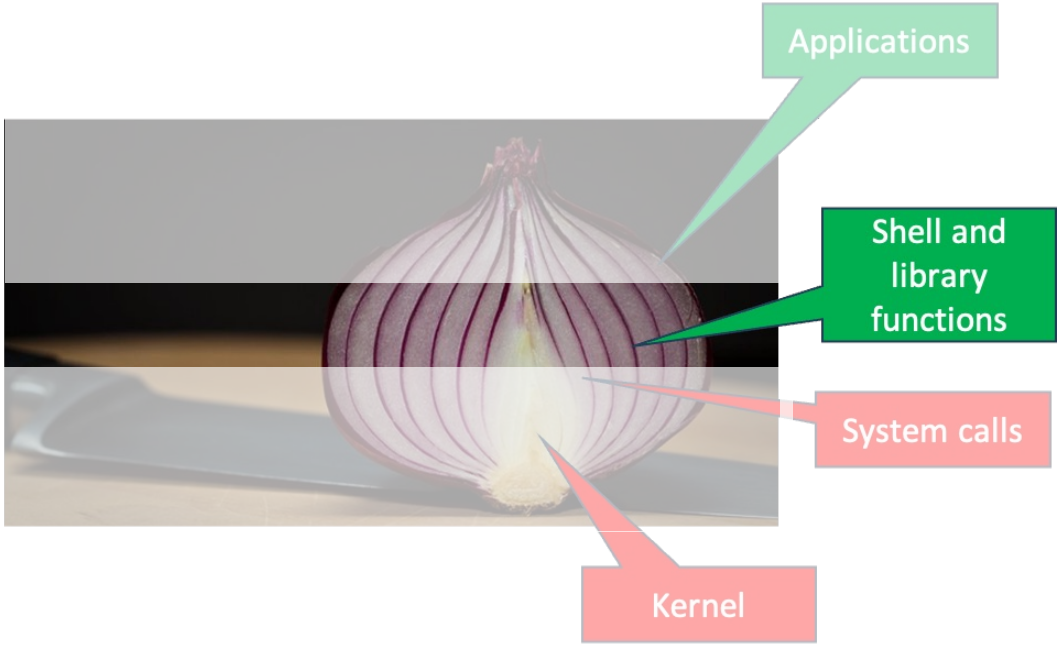


- Execute the IRET instruction that first pops the user registers from kernel stack and stores them back into respective registers
  - Return back to user call stack in the user mode
    - Protection level-3
- Note that latest architectures (both Intel and AMD) uses a special assembly instruction (sysenter) to avoid cache misses for accessing IDT
  - Faster execution of syscalls

Today's Class

- Unix architecture
- System calls
- Process creation and termination

The Shell



- It is the first user process created by the OS after the bootup
- It runs in the user mode but it can create more processes by using system call
- Its main job is to execute user commands
  - Recall how we launched ./fib executable in previous lecture

Shell Pseudocode (1/2)

```
void shell_loop() {
    int status;
    do {
        printf("iiitd@possum:~$ ");
        char* command = read_user_input();
        status = launch(command);
    } while(status);
}
```

```
iiitd@possum:~$ vi fib.c
iiitd@possum:~$ gcc fib.c
iiitd@possum:~$ ./a.out
Fib(40) = 102334155
```

- Shell runs in an infinite loop and reads the user input to execute
- Should cease execution if it was unable to execute user command

Shell Pseudocode (2/2)

```
int launch (char *command) {
    int status;
    status = create_process_and_run(command);
    return status;
}
```

```
iiitd@possum:~$ vi fib.c
iiitd@possum:~$ gcc fib.c
iiitd@possum:~$ ./a.out
Fib(40) = 102334155
```

- The launch method accepts the user input (command name along with arguments to it)
- It will create a new process that would execute the user command and return execution status

A Process's Life Lessons

1. Processes can have children
2. Children should be obedient to their parent
3. Parent must follow the steps for good parenting
4. Children should not run their family business

Creating Child Processes (1/3)

```
int create_process_and_run(char* command) {
    int status = fork();
    if(status < 0) {
        printf("Something bad happened\n");
    } else if(status == 0) {
        printf("I am the child process\n");
    } else {
        printf("I am the parent Shell\n");
    }
    ....
    return 0;
}
```

- fork is a system call used for creating a new process
- Called once, but returns twice!
  - Return value in child process is zero, whereas child's process PID is returned in parent process
- It creates a replica of the parent process
  - Both the child and parent process are going to execute the same code with a minute difference
  - Copy-on-Write (COW) – Initially, both parent and child process have read-only access to parent's address space. Whichever process attempts a write on a memory page in parent's address space, it would get a copy of that page (lazy copy)
    - What about opened file descriptors?

Creating Child Processes (2/3)

```
int create_process_and_run(char* command) {
    int status = fork();
    if(status < 0) {
        printf("Something bad happened\n");
    } else if(status == 0) {
        printf("I am the child process\n");
    } else {
        printf("I am the parent Shell\n");
    }
    ....
    return 0;
}
```

- Which of the two printfs would get printed first?
- The output is non-deterministic as the OS can decide on its own which one of the child or parent process should be in the "running" queue
  - Imagine there is single CPU
  - Will be discussed in details in later lectures on process scheduling

Creating Child Processes (3/3)

```
int global=0;
int create_process_and_run(char* command) {
    int status = fork();
    if(status < 0) {
        printf("Something bad happened\n");
    } else if(status == 0) {
        printf("I am the child process\n");
        global++;
    } else {
        printf("I am the parent Shell\n");
        sleep(2)
    }
    printf("Global value = %d\n",global);
    ....
    return 0;
}
```

- What value of the global variable will be printed?
- Although, the child is replica of the parent process, it has its own address space (heap, call stack, etc.) and registers
  - "Replica" here means both child and parent will run the exact same executable a.out immediately after calling fork (unless child and parent path are made separate as shown – if statement)
- Although we have made the parent to sleep for 2 seconds, it is not guaranteed that this duration is adequate for the child to move into running queue and complete its execution
- Inter-process communication is required for the updated global value to be seen by the parent
  - Next lecture!

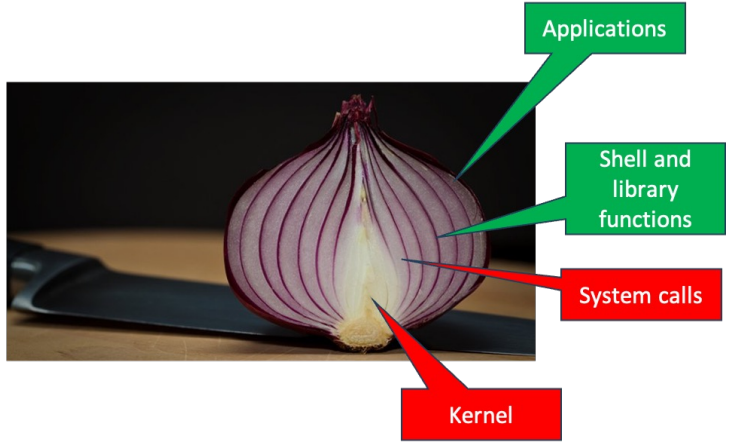
Next Lecture

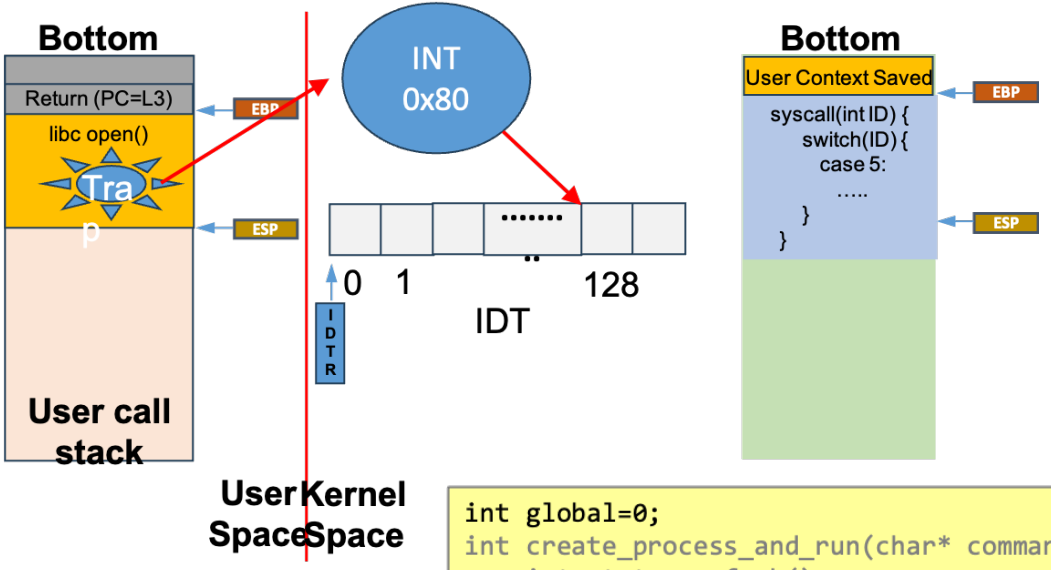
- Process life lessons (contd.) and inter process communication
- Assignment-2 will be released later this week
  - Having it now will give you ample time to prepare for mid semester exams as you might get flooded with assignments from other courses later
  - Rest of the assignments will be after the mid semester exam

Lecture 07: Process Termination and Inter Process Communication

Vivek Kumar
Computer Science and Engineering
IIIT Delhi
vivekk@iiitd.ac.in

Last Lecture





- Protection rings (kernel and User mode) in Unix-like OS
- Interrupts and system call
- Process creation
  - Copy on write when forked

```
int global=0;
int create_process_and_run(char* command) {
    int status = fork();
    if(status < 0) {
        printf("Something bad happened\n");
    } else if(status == 0) {
        printf("I am the child process\n");
        global++;
    } else {
        printf("I am the parent Shell\n");

    }
    printf("Global value = %d\n",global);
    ....
    return 0;
}
```

Today's Class

- Process's life lessons (contd.)
- Inter-process communication
  - Signals
  - Pipes

A Process's Life Lessons (contd.)

1. Processes can have children
2. Children should be obedient to their parent
3. Parent must follow the steps for good parenting
4. Children should not run their family business

The Obedient Child

```
int create_process_and_run(char* command) {
    int status = fork();
    if(status < 0) {
        printf("Something bad happened\n");
        exit(0);
    } else if(status == 0) {
        printf("I am the child process\n");
    } else {
        printf("I am the parent Shell\n");
    }
    ....
    return 0;
}
```

- **exit** syscall allows to send a specific termination code (exit status) from a child process to the parent upon termination
  - "signal" is sent to the parent (inter-process communication)
  - In case of abnormal termination of child, the exit status is generated and send by the kernel
- exit carries out process cleanup – reclaiming memory, flushes buffers, closing fds, etc.
- But how the parent can get the exit status (next slide)?
  - Remember child is not going to return to the parent just like a callee method returns to a caller method

The Act of Good Parenting

```
int create_process_and_run(char* command) {
    int status = fork();
    if(status < 0) {
        printf("Something bad happened\n");
        exit(0);
    } else if(status == 0) {
        printf("I am the child (%d)\n",getpid());
    } else {
        int ret;
        int pid = wait(&ret);
        if(WIFEXITED(ret)) {
          printf("%d Exit =%d\n",pid,WEXITSTATUS(ret));
        } else {
          printf("Abnormal termination of %d\n",pid);
        }
        printf("I am the parent Shell\n");
    }
    return 0;
}
```

- **wait** and **waitpid** allows the parent process to block until the child process terminates
  - **wait** will block only for the first child, whereas **waitpid** can be used for a specific child
  - Returns the child's PID
  - Used for retrieving exit status from child
- Will there be deterministic execution of printfs from parent and child processes (notice there is no sleep)?
- Good parents **wait** for their children to avoid making them **zombies** (terminated child's exit code remaining in the process table)
- However, **orphaned** children outliving their parent's lifetime are adopted by the mother-of-all-processes (**init**)

An Act of Kindness From a Bad Parent

```
int create_process_and_run(char* command) {
    int status = fork();
    if(status < 0) {
        printf("Something bad happened\n");
        exit(0);
    } else if(status == 0) {
        int status2 = fork();
        if(status2<0) printf("Kindness failed\n");
        else if (status2 == 0) {
           printf("Child will not live like Zombie\n");
        } else {
            _exit(0);
        }
    } else {
        printf("I am the parent Shell\n");
    }
    return 0;
}
```

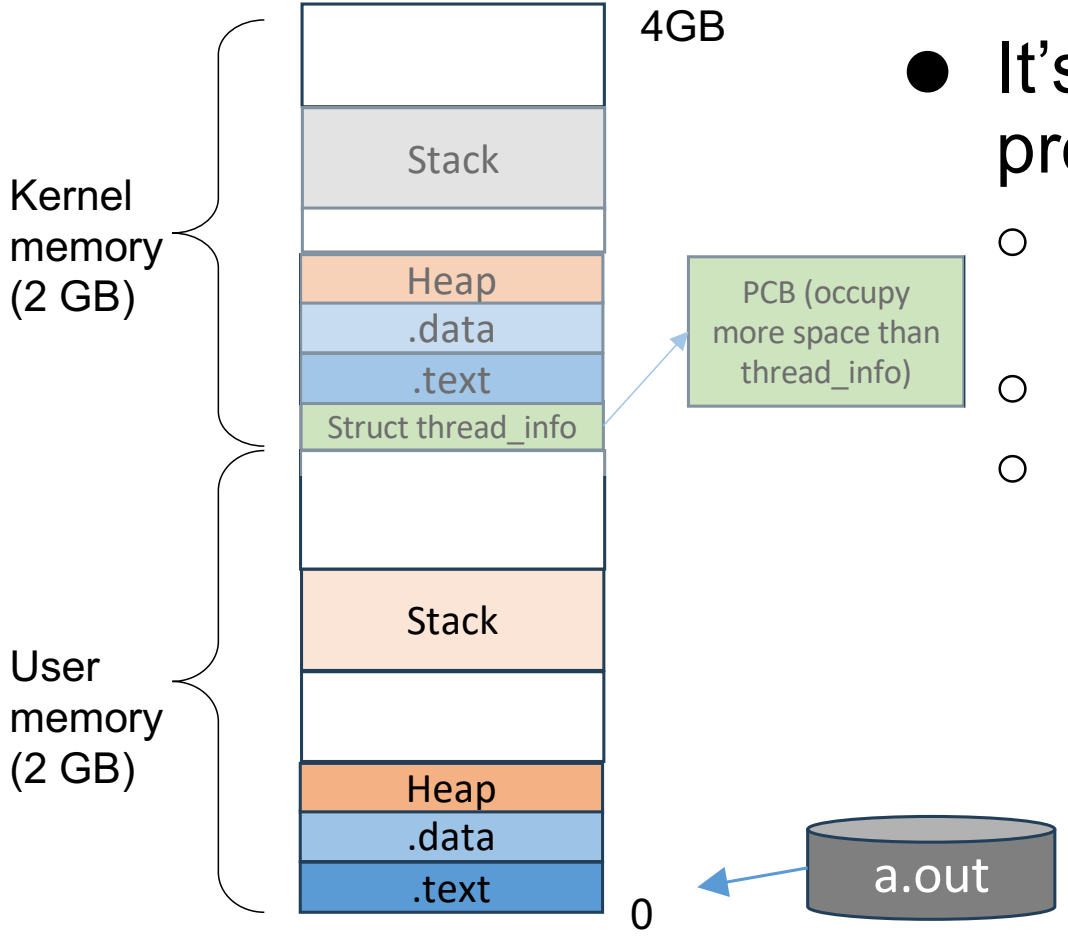
- There is a really bad parent who don't have a patience for good parenting
  - Get grandchildren and let them do the real work instead of the immediate children who will suffer a premature death
- The immediate child's only job is to get an offspring and die a quick death
  - The child should call **_exit** to ensure the offspring can inherit it's resources *(some kindness)*
- The mother-of-all process (**init**) will kindly adopt this process as her own child by issuing a **wait** call
  - No more Zombie!

Child Should Not Run Family Business

```
int create_process_and_run(char* command) {
    int status = fork();
    if(status < 0) {
        printf("Something bad happened\n");
        exit(0);
    } else if(status == 0) {
        printf("I am the child process\n");
        char* args[2] = {"./fib", "40"};
        execv(args[0], args);
        printf("I should never print\n");
    } else {
        printf("I am the parent Shell\n");
    }
    ....
    return 0;
}
```

- Main goal for creating a child process is to let it live its own free life without depending on its parent
  - The child won't let go off the parent's property until its forced to call exec
- An exec calls the OS loader internally that loads the ELF file with its command line argument as specified in the argument list
- There are seven different versions of exec which are collectively referred as exec function

exec Behind the Curtain

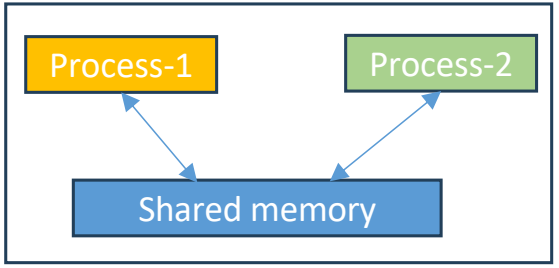


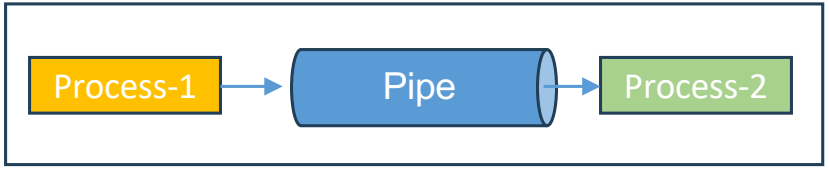
- It's only job is to construct the process's address space
  - Create the kernel part of the address space (2-pages)
  - Read ELF file from the disk
  - Create the user part of the address space lazily
    - E.g., space (pages) for .data will be allocated only after some global variable is accessed during program execution from the .text segment

Today's Class

- Process's life lessons (contd.)
- Inter-process communication
  - Signals
  - Pipes

Inter-Process Communication





- Recall, forked process calls an exec after which it has its own address space that is not shared with the parent
- IPC mechanisms to share information between processes

Points to Ponder

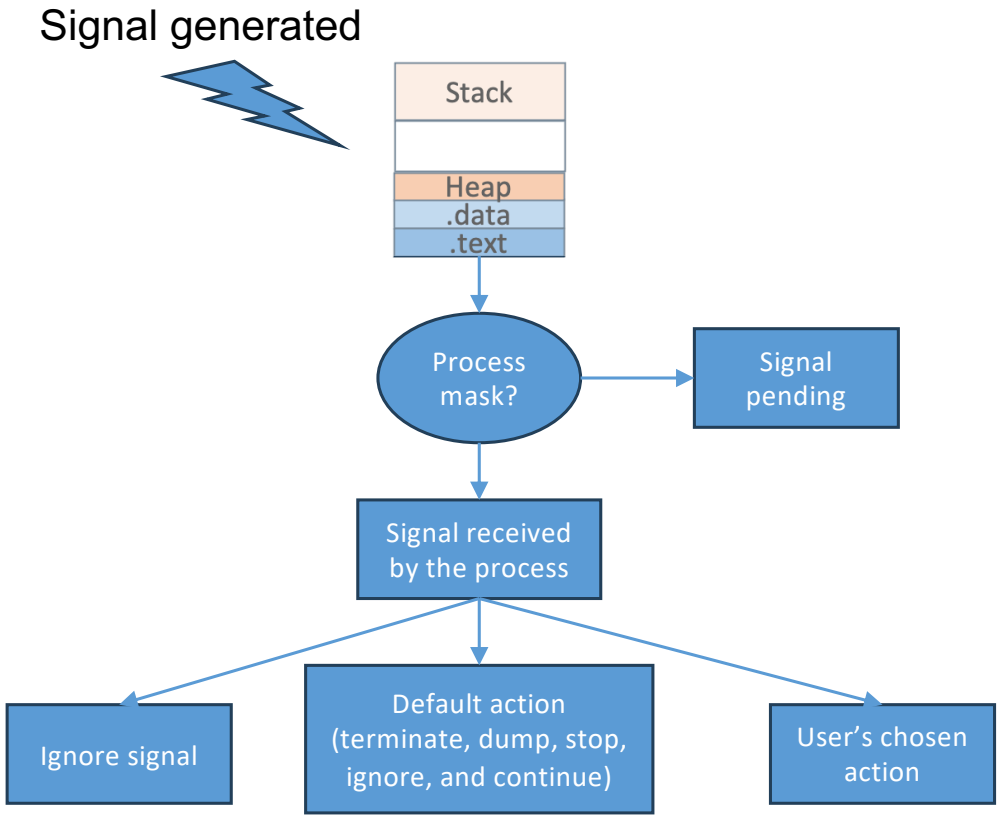
```
vivek@possum:~/os23$ vi infinite-fib.c
vivek@possum:~/os23$ gcc infinite-fib.c
vivek@possum:~/os23$ ./a.out
Input i:
20
Fib(20) = 6765
Input i:
^C
vivek@possum:~/os23$ ./a.out
Input i:
20
Fib(20) = 6765
Input i:
30
Fib(30) = 832040
Input i:
^Z
[1]+  Stopped                 ./a.out
vivek@possum:~/os23$ fg
./a.out
23
Fib(23) = 28657
Input i:
```

- What happens when we press the following key combination on a shell that is running some program
  - Ctrl-c
    - Process simply terminates even though it was executing some instruction inside the .text segment
  - Ctrl-z
    - Process moves to background (you can get it back to foreground using “fg”)
  - How come these actions are happening even though the program did not have any code to demonstrate the above mentioned behavior
- How does a child process that died inform about its fate to its parent?
- What happens when a program attempts to access an invalid memory address?

Signals – A Limited Form of IPC

- Signals are software interrupts
  - Technique used by the OS to communicate with a process
- Every signal has a name that has a unique number assigned
  - Ctrl-c and Ctrl-z generates hardware interrupt from the keyboard that is handled by the OS by sending SIGINT (2) and SIGSTOP (20) signal, respectively, to the running process
- Each signal type has a default handler
  - A program can install its own handler for signals of any type
    - Except SIGKILL (default action is to exit the process) and SIGSTOP (default action is to suspend the process)
- States
  - Generated – due to some event
  - Delivered – received by the process
  - Pending – to be delivered
  - Blocked – process has blocked the signal
  - Caught – action associated with signal has been taken by the process

Signal Delivery and Handling



- Process can use sigprocmask function to make changes to its signal mask (member in task_struct) if it wants to temporarily block a set of signals (not all signals can be blocked, e.g., SIGKILL and SIGSTOP)
- User can create his own signal handler function (we will see soon)
- Signal name has prefix "SIG". Some of the signals that you can easily come across are:
  - SIGINT (ctrl-c), SIGQUIT (ctrl-\), SIGSTOP (ctrl-z), SIGTRAP (breakpoint), SIGSEV (segmentation fault), SIGTERM (process termination), SIGCHLD (child stopped/terminated), SIGFPE (floating point exception), etc.

Catching the Signal

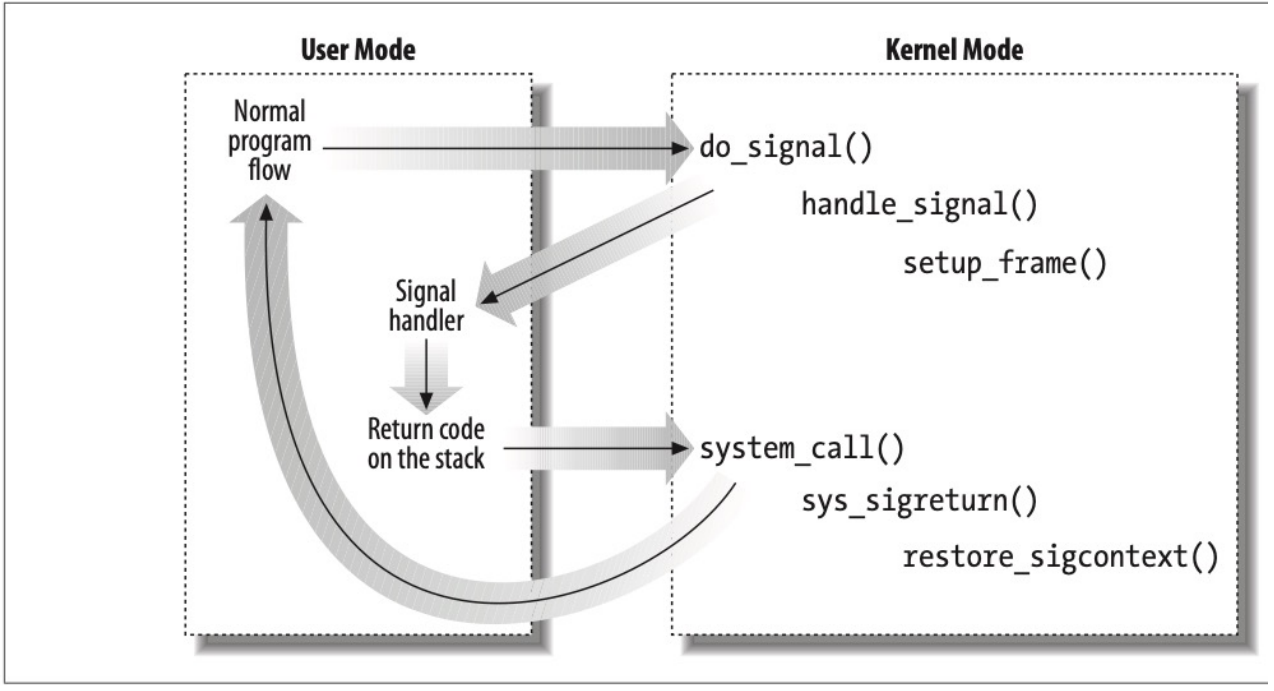


Figure 11-2. Catching a signal

- In the absence of user signal handler, signal is handled in kernel mode, otherwise it is handled in the user mode
- Process switches into kernel mode after receiving a non-blocking signal
- Kernel handle the signal by first setting up stack frame on user stack (save context)
- Control is passed to user stack and user signal handler is executed
- Control is returned back into kernel mode which restores the user stack to its original state to resume the execution of the program

Figure source: from the book "Understanding the Linux Kernel"

Programming Signals

```
static void my_handler(int signum) {
  static int counter = 0;
  if(signum == SIGINT) {
    char buff1[23] = "\nCaught SIGINT signal\n";
    write(STDOUT_FILENO, buff1, 23);
    if(++counter >= 2) {
      char buff2[20] = "Cannot handle more\n";
      write(STDOUT_FILENO, buff2, 20);
      exit(0);
    }
  } else if (signum == SIGCHLD) {
    char buff1[23] = "Caught SIGCHLD signal\n";
    write(STDOUT_FILENO, buff1, 23);
  }
}
```

```
int main() {
  struct sigaction sig;
  memset(&sig, 0, sizeof(sig));
  sig.sa_handler = my_handler;
  sigaction(SIGINT, &sig, NULL);
  sigaction(SIGCHLD, &sig, NULL);
  int n;
  while(1) {
    printf("Input i: \n");
    scanf("%d",&n);
    if(fork()==0) {
      printf("Fib(%d) = %d\n",n,fib(n));
      exit(0);
    }
    else wait(NULL);
  }
  return 0;
}
```

Why Fib(20) is calculated twice?

```
vivek@possum:~/os23$ ./a.out
Input i:
10
Fib(10) = 55
Caught SIGCHLD signal
Input i:
20
Fib(20) = 6765
Caught SIGCHLD signal
Input i:
^C
Caught SIGINT signal
Fib(20) = 6765
Caught SIGCHLD signal
Input i:
^C
Caught SIGINT signal
Cannot handle more
```

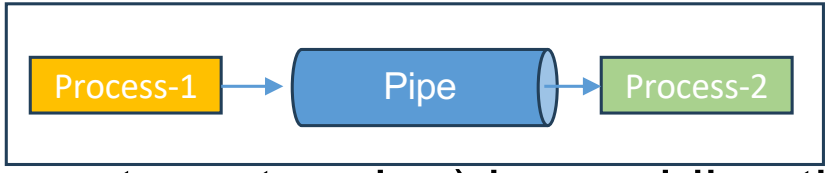
- sigaction is used to change the default action associated with a signal
  - Except SIGKILL and SIGSTOP
- User can assign his own signal handler method that would get invoked when the assigned signal is received by the process
- Only asynchronous safe functions should be called inside signal handler (e.g., never call printf)
  - *https://man7.org/linux/man-pages/man7/signal-safety.7.html*

Today's Class

- Process's life lessons (contd.)
- Inter-process communication
  - Signals
  - Pipes

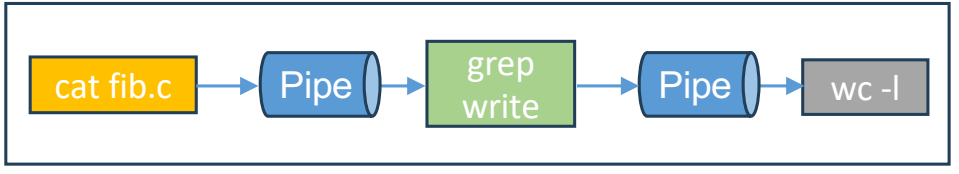
IPC Using Pipes

Processes **must** be running on the **SAME** machine



- Pipes (analogous to water pipe) is a unidirectional stream of data flowing from a source process to a destination process
  - Kernel buffer exposed to processes as a pair of file descriptors (readable end and writable end). Default buffer size on latest kernel is 16 pages (16x4096 bytes)
  - Data delivered in the same order as sent
  - Uses blocking IO and the writer process will block if the pipe is full
- We use it frequently on Shell
  - Better than using temporary files as pipes automatically clean up themselves unlike files that must be explicitly removed using the “rm” command on Shell

```
vivek@possum:~/os23$ cat fib.c | wc -c
1788
vivek@possum:~/os23$ cat fib.c | wc -l
77
vivek@possum:~/os23$ cat fib.c | grep write | wc -l
10
```



Programming With pipe

```
int main() {
    int fd[2], status;
    pipe(fd);
    if(fork() == 0) {
        /* Child process */
        close(fd[0]);
        char buff[] = “Hello my dear good Parent”;
        sleep(2);
        write(fd[1], buff, sizeof(buff));
        exit(0);
    }
    /* Parent process */
    close(fd[1]);
    char buff[100];
    read(fd[0], buff, sizeof(buff));
    printf(“My obedient child says: %s\n”, buff);
    wait(NULL);
    return 0;
}
```

- pipe expects an int array of size two only for a readable and writable file descriptor
- Reader and writer processes must close the pipe end they are not going to use
  - It is very important as fork duplicates the open file descriptors in the parent process
- What would happen if the child sleeps before writing to the pipe, as the parent has already issued the read?

Programming With pipe and dup

```
int main() {
    int fd[2], status;
    pipe(fd);
    if(fork() == 0) {
        /* Child process */
        close(fd[0]);
        dup2(fd[1], STDOUT_FILENO);
        char buff[] = "Hello my dear good Parent";
        printf("%s", buff);
        exit(0);
    }
    /* Parent process */
    close(fd[1]);
    dup2(fd[0], STDIN_FILENO);
    char buff[100];
    read(fd[0], buff, sizeof(buff));
    printf("My obedient child says: %s\n", buff);
    wait(NULL);
    return 0;
}
```

- dup2 can be used to duplicate a file descriptor
  - E.g., duplicate one of the end of the pipe as STDOUT or STDIN
    - Duplicating to STDOUT will cause printf to print to the pipe instead of the STDOUT
- Used by the Shell when we pipe the output of one command to another command

Next Lecture

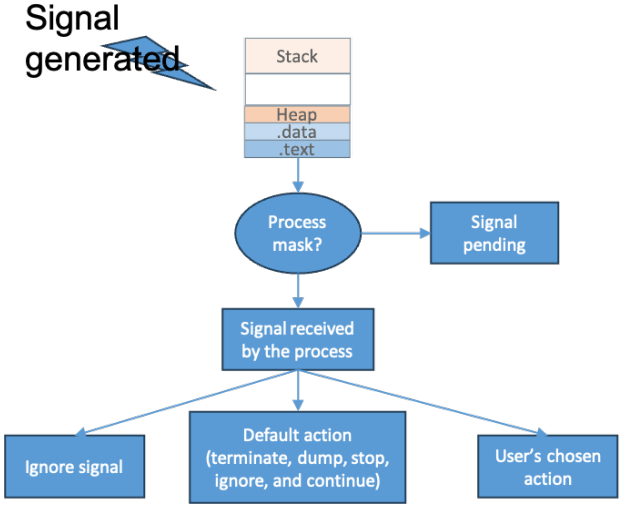
- Inter-process communication (contd.)
- Assignment-2 will be released on Monday (04/09)

Lecture 08: IPC in Shared Memory

Vivek Kumar
Computer Science and Engineering
IIIT Delhi
vivekk@iiitd.ac.in

Last Lecture

```
int create_process_and_run(char* command) {
    int status = fork();
    if(status < 0) {
        printf("Something bad happened\n");
        exit(0);
    } else if(status == 0) {
        printf("I am the child process\n");
        char* args[2] = {"./fib", "40"};
        execv(args[0], args);
        printf("I should never print\n");
    } else {
        printf("I am the parent Shell\n");
    }
    ....
    return 0;
}
```




```
int main() {
    int fd[2], status;
    pipe(fd);
    if(fork() == 0) {
        /* Child process */
        close(fd[0]);
        dup2(fd[1], STDOUT_FILENO);
        char buff[] = "Hello my dear good Parent";
        printf("%s", buff);
        exit(0);
    }
    /* Parent process */
    close(fd[1]);
    dup2(fd[0], STDIN_FILENO);
    char buff[100];
    read(fd[0], buff, sizeof(buff));
    printf("My obedient child says: %s\n", buff);
    wait(NULL);
    return 0;
}
```

- Child process should call **exit** system call before terminating to pass its exit status to the parent process
  - Parent can retrieve it by calling **wait**
- Zombies and orphan processes
- The exec system call
- IPC using signaling
- IPC using pipe and dup

Today's Class

- Shared memory
  - Introduction to parallel computing
  - Semaphores for process synchronization
- Producer consumer problem

Array Sum Program

```
int main() {
    int A[SIZE]; //initialized
    int sum=0;
    for(int i=0; i<SIZE; i++) sum += A[i];
    return 0;
}
```

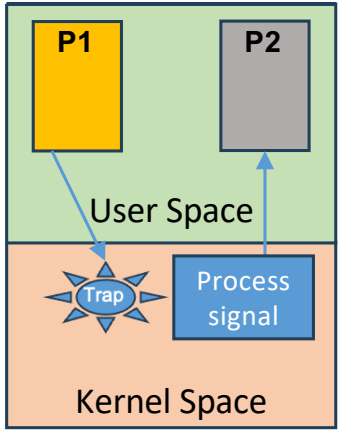
- A simple program to calculate the sum of all the elements inside a huge array
- How to shorten the execution time for calculating the sum for a very big array?

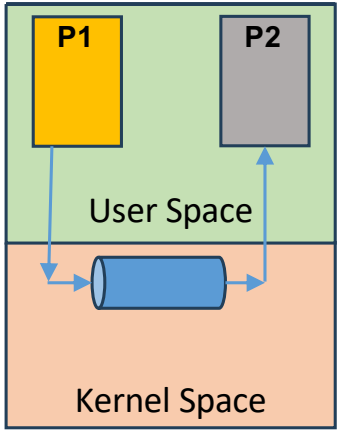
Array Sum Program

```
int main() {
    int A[SIZE]; //initialized
    int sum=0;
    if(fork() == 0) {
        for(int i=SIZE/2; i<SIZE; i++) sum += A[i];
        exit(0);
    }
    for(int i=0; i<SIZE/2; i++) sum += A[i];
    wait(NULL);
    return 0;
}
```

- We can fork a child and let the parent and child calculate the sum of left and right half of the array, respectively
- The problem(s)
  - How the child and parent would communicate their partial results to get the total sum?
    - Can the child pass the partial sum as its exit code?
      - exit API accepts only “int” type parameter. We cannot pass the sum of “double” type array, etc.
  - Can we setup a pipe between the parent and child?
    - Yes, but what if there are several children to improve the execution time further? We will have to use multiple pipes and keep track of them (closing fds differently at child v/s parent, etc.)
  - What if its some other kind of program, such as vector addition or matrix multiplication?
    - Three arrays (or matrices) would require accesses by child and parent

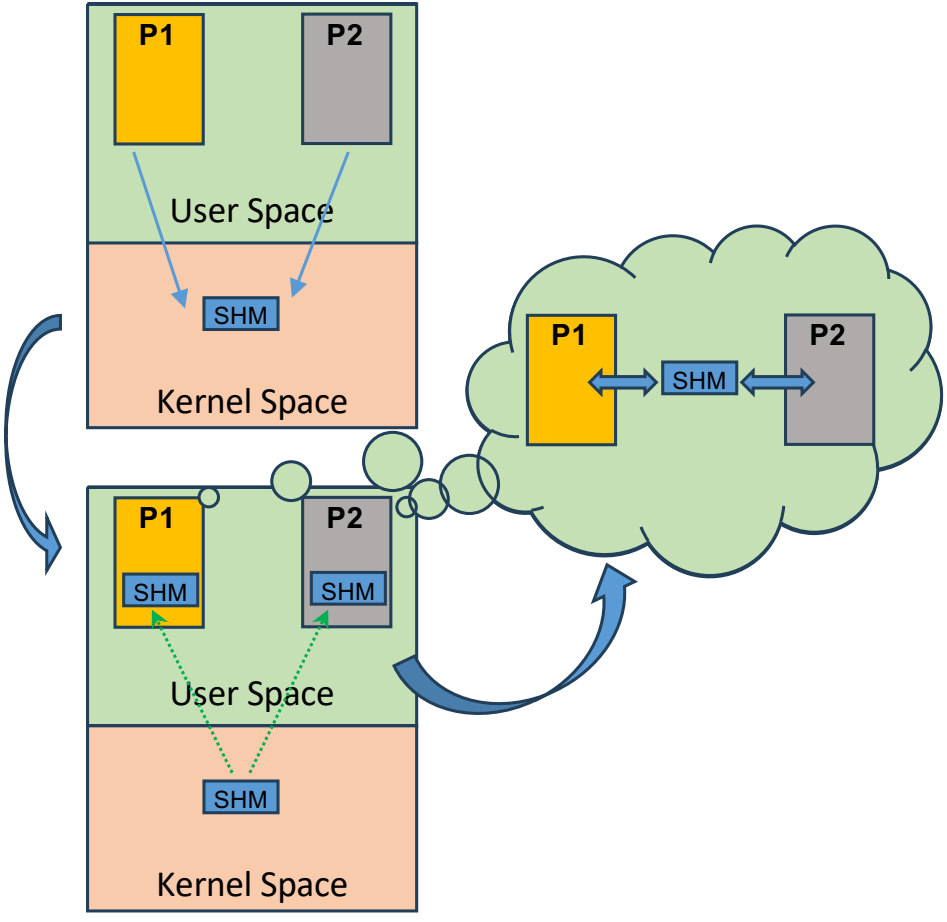
IPC Using Signals and Pipes





- Processes running inside a single machine can use simple IPC techniques such as signaling, pipes, shared memory, etc.
- Issues with signaling
  - Limited form of IPC
  - Requires help from OS for every signaling instances
  - Transition between user space and kernel space (overheads)
- Issues with pipes
  - Limited buffer size (although significantly better than signaling)
  - Use of system calls for every communication
    - Transition between user space and kernel space (overheads)
  - One way communication
  - Blocking communication

IPC Using POSIX Shared Memory



- Shared memory is the fastest form of IPC in which processes can asynchronously write to a shared region of memory **without** the need for switching to kernel space
  - User specified memory size
- Access to the shared memory is achieved as follows
  1. Create and open a new shared memory object using shm_open system call in the parent process
  2. Set the desired size for the newly created shared memory region using ftruncate
  3. Map the shared memory region into the process's address space using mmap (parent process calls mmap that gets shared in child's address space after fork)
  4. Both child and the parent processes un-maps the shared memory from their address space upon completion by using munmap system call, followed by closing the file descriptor for the shared memory object (close system call)
  5. Parent deletes the shared memory object by using shm_unlink
- Can be used for parallel programming!

Array Sum Program

```
int main() {

    return 0;
}
```

```
typedef struct shm_t {
    int A[SIZE];
    int sum;

} shm_t;
```

- We will create a shared memory segment of our own type shm_t that contains the array and a variable sum to store the partial result

Array Sum Program

```
int main() {
    shm_t* shm = setup();

    cleanup();
    return 0;
}
```

1) shm_open (...)
2) ftruncate (...)
3) shm = mmap (...)

1) munmap (shm, ...)
2) close (...)
3) shm_unlink (...)

```
typedef struct shm_t {
    int A[SIZE];
    int sum;

} shm_t;
```

- The three steps for setting up a shared memory segment are:
  - Creating a shared memory object (shm_open)
  - Assigning a size for the shared memory object (ftruncate)
  - Mapping the shared memory into the process's address space (mmap)
- The three steps in cleanup are:
  - Removing the mapping of shared memory from process's address space
  - Closing the file descriptor for the shared memory object
  - Deleting the shared memory object

Array Sum Program

```
int main() {
    shm_t* shm = setup();


    for(int i=0; i<NPROCS; i++) {
        if(fork()==0) {
```

1) munmap (shm, ...)
2) close (...)
3) exit(0)

```
            cleanup_and_exit();
        }
    }
    for(int i=0; i<NPROCS; i++) wait(NULL);
    cleanup();
    return 0;
}
```

```
typedef struct shm_t {
    int A[SIZE];
    int sum;

} shm_t;
```

- Parent process will fork children that gets a copy of the parent's address space (inherits the file descriptor and shared memory object from the parent)
- Child process also carries out cleanup similar to parent except for calling **shm_unlink** that is called only by the parent
  - Missing this step will fail the **shm_open** next time (unless you change the name of the memory region)
- Child must also call **exit** in their cleanup call
  - **Why?**

Array Sum Program

```
int main() {
    shm_t* shm = setup();

    int chunks = SIZE/NPROCS;
    for(int i=0; i<NPROCS; i++) {
        if(fork()==0) {
            int local=0;
            int start = i*chunks;
            int end = start+chunk;
            for(int i=start; i<end; i++) local += shm->array[i];

            shm->sum += local;

            cleanup_and_exit();
        }
    }
    for(int i=0; i<NPROCS; i++) wait(NULL);
    cleanup();
    return 0;
}
```

```
typedef struct shm_t {
    int A[SIZE];
    int sum;

} shm_t;
```

- Children update the global **sum** in the shared memory region
  - Is is correct?
    - **Race condition!**
    - All children are updating the same shared global variable sum

Race Condition

```
shm->sum += local;
```

- Final value of the sum will not be correct as there is a race between reading and writing the counter "sum" by multiple processes
- The line of code shown above is called as critical section
  - Critical section is a line **or** block of code that access shared modifiable data or resource that should be operated on by only one process at a time
- We will revisit this topic during lectures on concurrency, but for now we will use semaphores to solve the problem of race condition by using mutual exclusion
  - Mutual exclusion is a property that ensures that there is no race condition by executing the critical section by a single process only at any given time

Semaphores

Value of Semaphore	Process-T1	Process-T2
1		
1	call sem_wait ()	
0	sem_wait () returns	
0	(crit sect)	
0	call sem_post ()	
1	sem_post () returns	

Val	Process-T1	State	Process-T2	State
1		Run		Ready
1	call sem_wait ()	Run		Ready
0	sem_wait () returns	Run		Ready
0	(crit sect begin)	Run		Ready
0	*Interrupt; Switch→T1*	Ready		Run
0		Ready	call sem_wait ()	Run
-1		Ready	decr sem	Run
-1		Ready	(sem<0)→sleep	Sleep
-1		Run	*Switch→T0*	Sleep
-1	(crit sect end)	Run		Sleep
-1	call sem_post ()	Run		Sleep
0	incr sem	Run		Sleep
0	wake(T1)	Run		Ready
0	sem_post () returns	Run		Ready
0	*Interrupt; Switch→T1*	Ready		Run
0		Ready	sem_wait () returns	Run
0		Ready	(crit sect)	Run
0		Ready	call sem_post ()	Run
1		Ready	sem_post () returns	Run

- A semaphore is an object with an integer **value** can be manipulate with:
  - **sem_wait**
    - value = value **-1**
    - Blocking if value < 0, otherwise non-blocking
  - **sem_post**
    - Non-blocking API
    - value = value **+1**
    - If there are one more processes blocked inside sem_wait then wake **any one** of them
  - Helps in achieving mutual exclusion over a critical section

Source: OSTEP Book

Array Sum Program

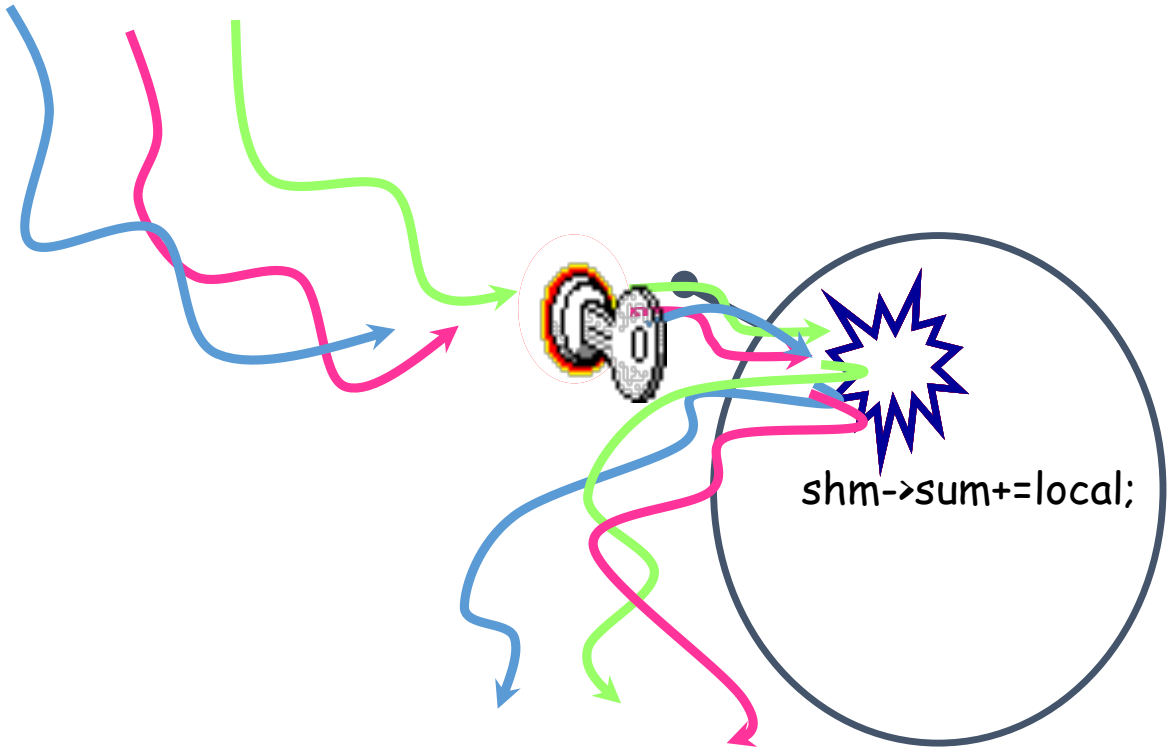
```
int main() {
    shm_t* shm = setup();
    sem_init(&shm->mutex, 1, 1);
    int chunks = SIZE/NPROCS;
    for(int i=0; i<NPROCS; i++) {
        if(fork()==0) {
            int local=0;
            int start = i*chunks;
            int end = start+chunk;
            for(int i=start; i<end; i++) local += shm->array[i];
            sem_wait(&shm->mutex);
            shm->sum += local;
            sem_post(&shm->mutex);
            cleanup_and_exit();
        }
    }
    for(int i=0; i<NPROCS; i++) wait(NULL);
    cleanup();
    return 0;
}
```

```
typedef struct shm_t {
    int A[SIZE];
    int sum;
    sem_t mutex;
} shm_t;
```

1) munmap (shm, ...)
2) close (...)
3) shm_unlink (...)
4) **sem_destroy**

- We used a binary semaphore to synchronize the accesses on the sum variable
- Semaphore helped in achieving mutual exclusion over the critical section
  - No more race condition!

Visualizing Mutual Exclusion



- Only one process can get the “key” to enter the critical section
- Rest all processes will be queued to get the key

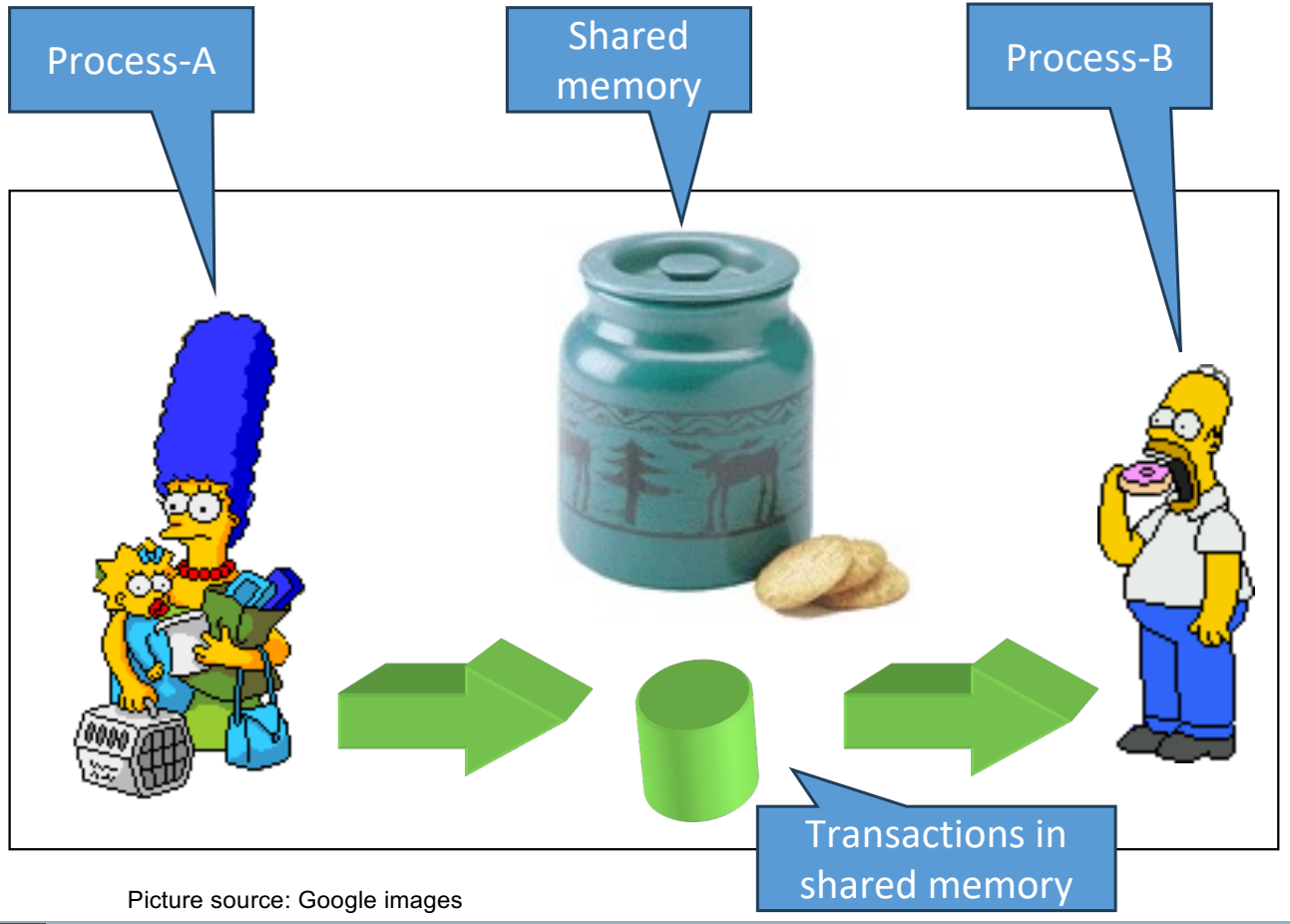
Measuring the Parallel Performance?

- Speedup is used to denote the performance improvement by using multiple processes
  - Speedup = $T_{serial}/T_{parallel}$

Today's Class

- Shared memory
  - Introduction to parallel computing
  - Semaphores for process synchronization
- Producer consumer problem

The Producer Consumer Problem



Picture source: Google images

- The Simpsons story
  - Homer is fond of eating cookies every time
  - Marge is super busy with her baby and so cannot bake a lot of cookies once
  - The cookie jar is the place to store cookies shared between Marge and Homer
  - Homer picks a cookies from cookie jar if available and then waits for Marge to prepare one for him
  - Marge waits for Homer to eat the cookie from cookie jar and prepares one if no cookie is available
- We need to synchronize between transactions, for example, the producer-consumer scenario described above (or bounded buffer problem)

The Producer Consumer Problem

```
vivek@possum:~/os23$ ./a.out
Marge bake Cookie-1
Homer ate Cookie-1
Marge bake Cookie-2
Homer ate Cookie-2
Marge bake Cookie-3
Homer ate Cookie-3
Marge bake Cookie-4
Homer ate Cookie-4
Marge bake Cookie-5
Homer ate Cookie-5
```

- The Simpsons story
  - Homer is fond of eating cookies every time
  - Marge is super busy with her baby and so cannot bake a lot of cookies once
  - The cookie jar is the place to store cookies shared between Marge and Homer
  - Homer picks a cookies from cookie jar if available and then waits for Marge to prepare one for him
  - Marge waits for Homer to eat the cookie from cookie jar and prepares one if no cookie is available
- We need to synchronize between transactions, for example, the producer-consumer scenario described above (or bounded buffer)

The Producer Consumer Problem

```
typedef struct cookiejar_t {
    int cookie;
    int empty;
} cookiejar_t;

cookiejar_t* cookiejar;

int main() {
    cookiejar = setup();
    cookiejar->empty=1;
    if(fork() == 0) { homer(); }
    if(fork() == 0) { marge(); }
    wait(NULL); // wait for Homer process
    wait(NULL); // wait for Marge process
    cleanup();
    return 0;
}
```

1) shm_open (...)
2) ftruncate (...)
3) mmap (...)

1) munmap (...)
2) close (...)
3) shm_unlink (...)

- We will create a shared memory segment of the type cookiejar_t that contains the “cookie” number and a flag “empty” to indicate the availability of a cookie in the cookie jar
- Setup and cleanup of shared memory region works exactly as described earlier in Slide #8

The Producer Consumer Problem

```
void homer() {
    for(int i=0; i<5; i++) {
        while(cookiejar->empty) {
            /*Loop endlessly*/
        }
        printf("Homer ate Cookie-%d\n", cookiejar->cookie);
        cookiejar->empty = 1;
    }
    cleanup_and_exit();
}
```

```
void marge() {
    for(int i=0; i<5; i++) {
        while(!cookiejar->empty) {
            /*Loop endlessly*/
        }
        printf("Marge bake Cookie-%d\n", ++cookiejar->cookie);
        cookiejar->empty = 0;
    }
    cleanup_and_exit();
}
```

```
vivek@possum:~/os23$ ./a.out
Marge bake Cookie-1
^C
vivek@possum:~/os23$ ./a.out
Marge bake Cookie-1
^C
vivek@possum:~/os23$ ./a.out
Marge bake Cookie-1
^C
```

- I tried running the program several times but Homer was simply not eating any cookie despite the fact that Marge had in fact baked one for him..
  - What went wrong?

Compiler Ruined the Show!

```
void homer() {
    for(int i=0; i<5; i++) {
        while(true) {
            /*Loop endlessly*/
        }
        printf("Homer ate Cookie-%d\n", cookiejar->cookie);
        cookiejar->empty = 1;
    }
    cleanup_and_exit();
}
```

```
void marge() {
    for(int i=0; i<5; i++) {
        while(false) {
            /*Loop endlessly*/
        }
        printf("Marge bake Cookie-%d\n", ++cookiejar->cookie);
        cookiejar->empty = 0;
    }
    cleanup_and_exit();
}
```

- Compiler did not know that the value of variable "**empty**" is being changed by another process
- It noticed during compilation that main() first sets empty=1 and then calls the methods homer() and marge() where the variable **empty** was simply being tested inside while condition
- It acts smart and carries out optimization by simply replacing the variable "**empty**" with 1 at all places
- Fix?
  - Declare the variables "empty" as **volatile**

The Producer Consumer Problem

```
void homer() {
    for(int i=0; i<5; i++) {
        while(cookiejar->empty) {
            /*Loop endlessly*/
        }
        printf("Homer ate Cookie-%d\n", cookiejar->cookie);
        cookiejar->empty = 1;
    }
    cleanup_and_exit();
}
```

```
void marge() {
    for(int i=0; i<5; i++) {
        while(!cookiejar->empty) {
            /*Loop endlessly*/
        }
        printf("Marge bake Cookie-%d\n", ++cookiejar->cookie);
        cookiejar->empty = 0;
    }
    cleanup_and_exit();
}
```

```
vivek@possum:~/os23$ ./a.out
Marge bake Cookie-1
Homer ate Cookie-1
Marge bake Cookie-2
Homer ate Cookie-2
Marge bake Cookie-3
Homer ate Cookie-3
Marge bake Cookie-4
Homer ate Cookie-4
Marge bake Cookie-5
Homer ate Cookie-5
```

- **volatile** seems to have fixed the problem?
  - It is an incorrect solution
  - What happens when several processes are racing for updating the variable "empty"?
    - **Race condition!**

Semaphores to the Rescue!

```
typedef struct cookiejar_t {
    int cookie;
    sem_t jar_empty;
    sem_t jar_full;
} cookiejar_t;

cookiejar_t* cookiejar;

int main() {
    cookiejar = setup();
    cookiejar->empty=1;
    sem_init(&cookiejar->jar_empty, 1, 1);
    sem_init(&cookiejar->jar_full, 1, 0);
    if(fork() == 0) { homer(); }
    if(fork() == 0) { marge(); }
    wait(NULL); // wait for Homer process
    wait(NULL); // wait for Marge process
    sem_destroy(&cookiejar->jar_empty);
    sem_destroy(&cookiejar->jar_full);
    cleanup();
    return 0;
}
```

- We will declare two semaphore variables inside the shared memory region so as they can be shared by both Marge and Homer processes
- Main process will call the **init** and **destroy** APIs on these semaphore objects

Semaphores to the Rescue!

```
void homer() {
    for(int i=0; i<5; i++) {
        sem_wait(&cookiejar->jar_full);
        printf("Homer ate Cookie-%d\n", cookiejar->cookie);
        sem_post(&cookiejar->jar_empty);
    }
    cleanup_and_exit();
}
```

```
void marge() {
    for(int i=0; i<5; i++) {
        sem_wait(&cookiejar->jar_empty);
        printf("Marge bake Cookie-%d\n", ++cookiejar->cookie);
        sem_post(&cookiejar->jar_full);
    }
    cleanup_and_exit();
}
```

```
vivek@possum:~/os23$ ./a.out
Marge bake Cookie-1
Homer ate Cookie-1
Marge bake Cookie-2
Homer ate Cookie-2
Marge bake Cookie-3
Homer ate Cookie-3
Marge bake Cookie-4
Homer ate Cookie-4
Marge bake Cookie-5
Homer ate Cookie-5
```

- At the start, the value of semaphores **jar_full** and **jar_empty** was set to **0** and **1**, respectively
- Homer process upon creation will block in sem_wait as the value of jar_full was initially 0 (decremented to -1)
- Marge upon activation will decrement jar_empty to 0 and will not block. It will bake a cookie, increment the jar_full (now its zero), and wake up Homer from sem_wait. Finally, it will block inside sem_wait after decrementing jar_empty (now its -1)
- Homer will awake from sem_wait, eat the cookie, increment the jar_empty semaphore (now its 0), and wake up Marge from sem_wait. Finally, it will block inside sem_wait after decrementing jar_full (now its -1)
- And the cookie business continues for five times..
- We will revisit the topic of mutual exclusion during lectures on concurrency (multithreading), where we will discuss it in depth

Next Lecture

- IPC in distributed memory
  - Last remaining topic in IPC
- Quiz-2
  - Syllabus: Lectures 05-08

Lecture 09: Inter-Process Communication in Distributed Memory

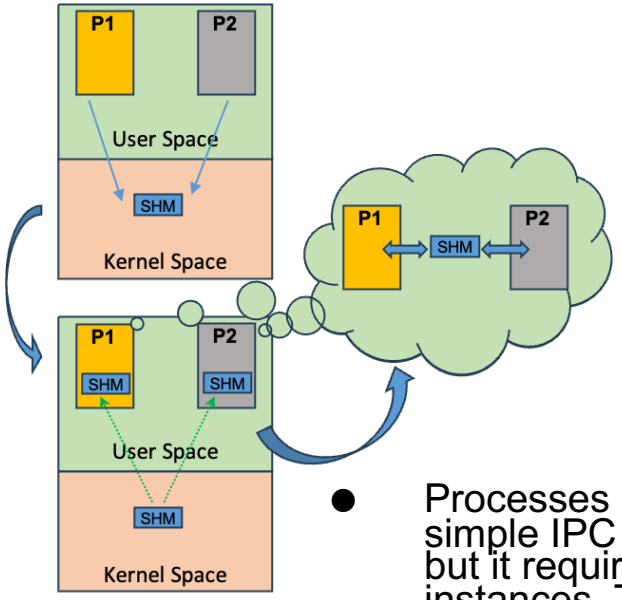
Vivek Kumar

Computer Science and Engineering

IIIT Delhi

vivekk@iiitd.ac.in

Last Lecture




```
int main() {
    shm_t* shm = setup();
    sem_init(&shm->mutex, 1, 1);
    int chunks = SIZE/NPROCS;
    for(int i=0; i<NPROCS; i++) {
        if(fork()==0) {
            int local=0;
            int start = i*chunks;
            int end = start+chunk;
            for(int i=start; i<end; i++) local += shm->array[i];
            sem_wait(&shm->mutex);
            shm->sum += local;
            sem_post(&shm->mutex);
            cleanup_and_exit();
        }
    }
    for(int i=0; i<NPROCS; i++) wait(NULL);
    cleanup();
    return 0;
}
```

```
typedef struct shm_t {
    int A[SIZE];
    int sum;
    sem_t mutex;
} shm_t;
```

- Processes running inside a single machine can use simple IPC techniques such as signaling or pipes, but it requires help from OS for every signaling instances. There is always a transition between user space and kernel space (overheads)
- Shared memory is the fastest form of IPC in which processes can asynchronously write to a shared region of memory without the need for switching to kernel space
- A semaphore is an object with an integer **value** that is used for achieving mutual exclusion over a critical section

```
void homer() {
    for(int i=0; i<5; i++) {
        sem_wait(&cookiejar->jar_full);
        printf("Homer ate Cookie-%d\n", cookiejar->cookie);
        sem_post(&cookiejar->jar_empty);
    }
    cleanup_and_exit();
}
```

```
void marge() {
    for(int i=0; i<5; i++) {
        sem_wait(&cookiejar->jar_empty);
        printf("Marge bake Cookie-%d\n", ++cookiejar->cookie);
        sem_post(&cookiejar->jar_full);
    }
    cleanup_and_exit();
}
```

Today's Class

- IPC in distributed memory
  - Message Passing Interface
- Quiz-2

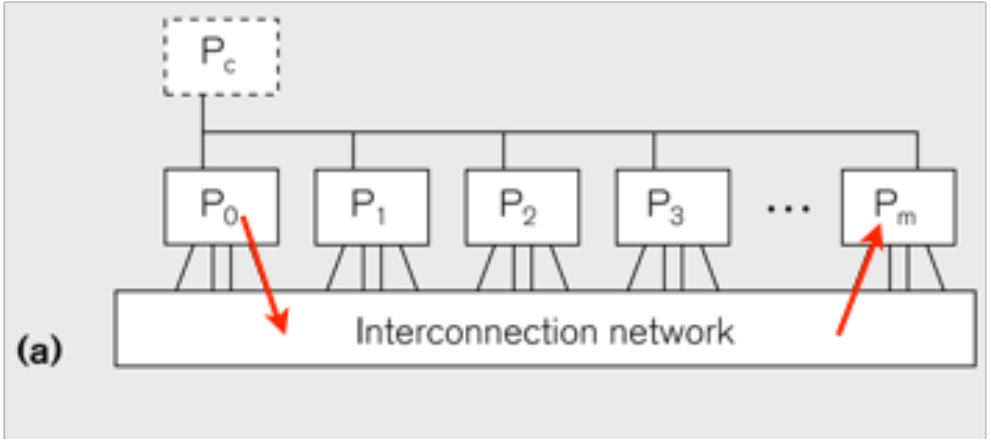
Distributed Memory

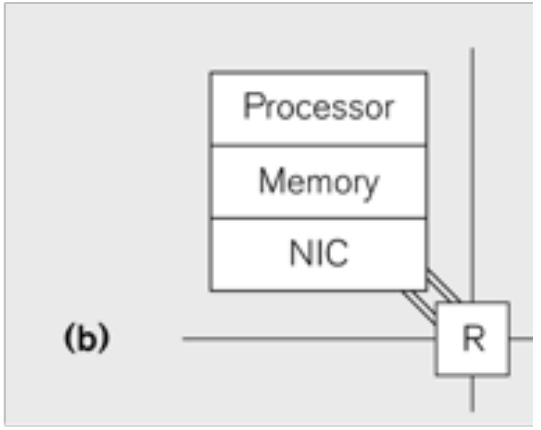
Picture source: https://cs.boisestate.edu/~amit/research/beowulf/cluster-small.jpg

- Several machines (e.g., workstations) connected together in a cluster with some interconnect (e.g., ethernet)
  - Easy to scale computing resources. E.g., ten desktops each having a 10-core processor would result in 100-core processing power in a cluster (a.k.a. Beowulf cluster)
  - Supercomputers are similar but with high-end processors and interconnects
  - Grid computing is also similar, but the machines are not inside a single room but can be in different countries/location connected with internet (it is a type of distributed computing instead of cluster computing)
- One big computation can be divided equally into N number of sub-computations that can run on N different processes
  - 1x1 mapping between a process and a core in the cluster

Organization of a Distributed Memory Multiprocessor

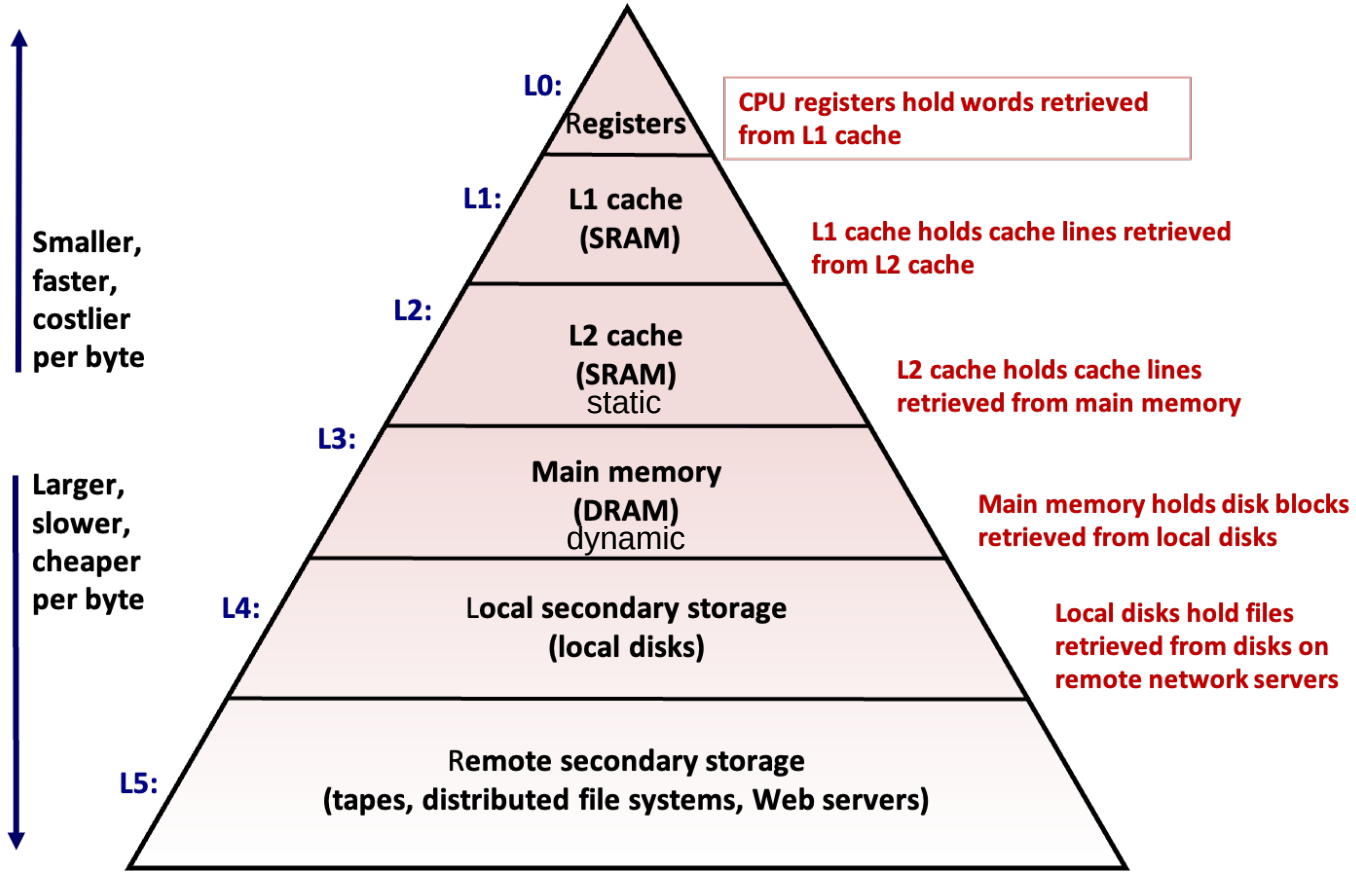
==> Processors communicate by sending messages via an interconnect





- Figure-a
  - Host processor (Pc) connected to a cluster of processor nodes (P0 … Pm)
  - Processors P0 … Pm communicate via an interconnection network
- Figure-b
  - Each processor node consists of a processor, memory, and a Network Interface Card (NIC) connected to a router (R) in the interconnect

Distributed Memory



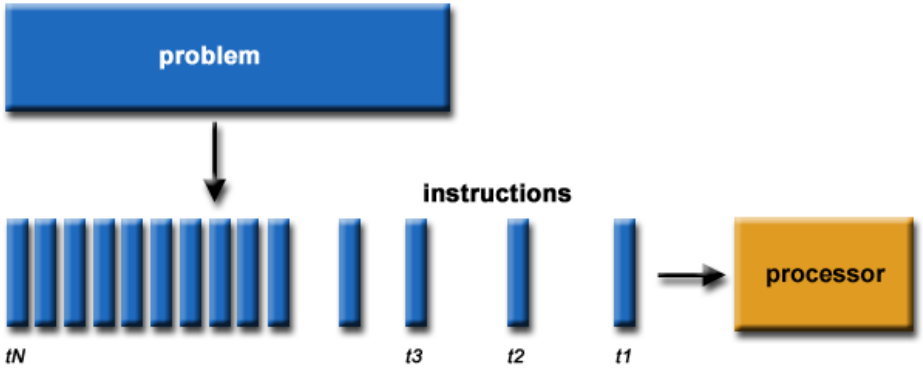
Picture source: http://www.cs.cmu.edu/afs/cs/academic/class/15213-f10/www/lectures/09-memory-hierarchy.pptx

- Distributed memory access latency slowest in the memory hierarchy
- Each process has access to the memory on the local system as well as the memory on the remote machines
- Processes communicate with each other using inter-process communication

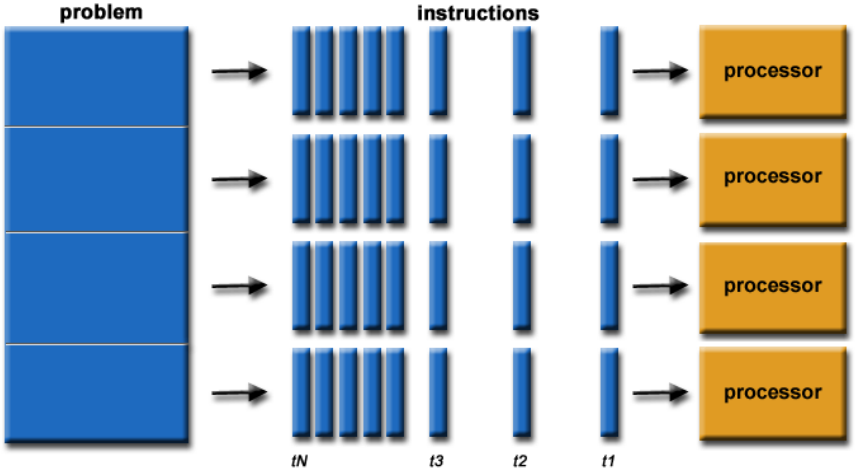
IPC in Distributed Memory

- Processes communicate in distributed memory by exchanging messages with each other
  - For example, client-server model that uses sockets based communication
- In this course we will only **introduce** a simple to use runtime library for exchanging messages between processes
  - Message Passing Interface (MPI) that provides a set of standard and portable APIs for **parallel computing** in distributed memory (as well as between the processes running in a shared memory)

Introduction to Parallel Computing

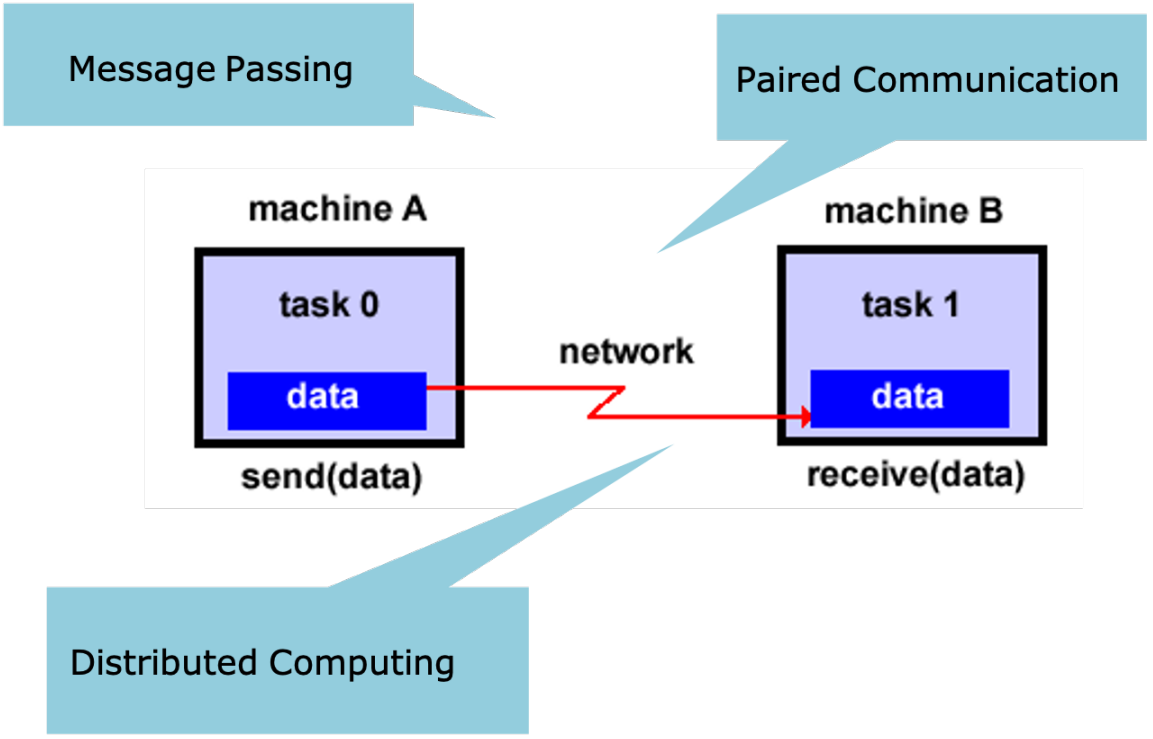


Serial (sequential) computing



Parallel computing

Message Passing Interface (MPI)



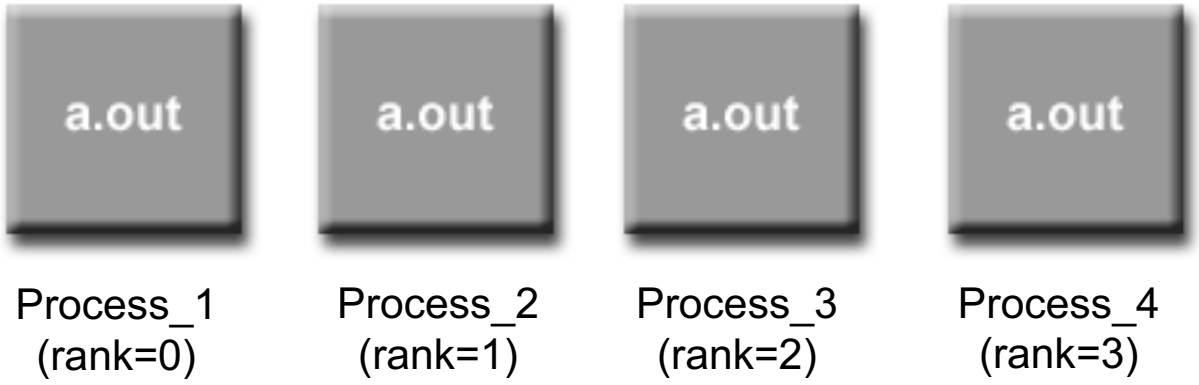
- The logical view of a machine supporting the message-passing paradigm consists of *p processes*, each with its own exclusive address space, that are capable of executing on different nodes in a distributed-memory multiprocessor
  1. Each data element must belong to one of the partitions of the space; hence, data must be explicitly partitioned and placed.
  2. All interactions (read-only or read/write) require cooperation of two processes - the process that has the data and the process that wants to access the data.
- In this loosely synchronous model, processes synchronize infrequently to perform interactions. Between these interactions, they execute completely asynchronously.

Single Program Multiple Data (SPMD)

```
void array_sum() {
    int A[SIZE]; //initialized
    int sum=0;
    if(fork() == 0) {
        printf("I am the child process\n");
        for(int i=SIZE/2; i<SIZE; i++) sum += A[i];
        exit(0);
    }
    printf("I am the parent process\n");
    for(int i=0; i<SIZE/2; i++) sum += A[i];
    wait(NULL);
    return 0;
}
```

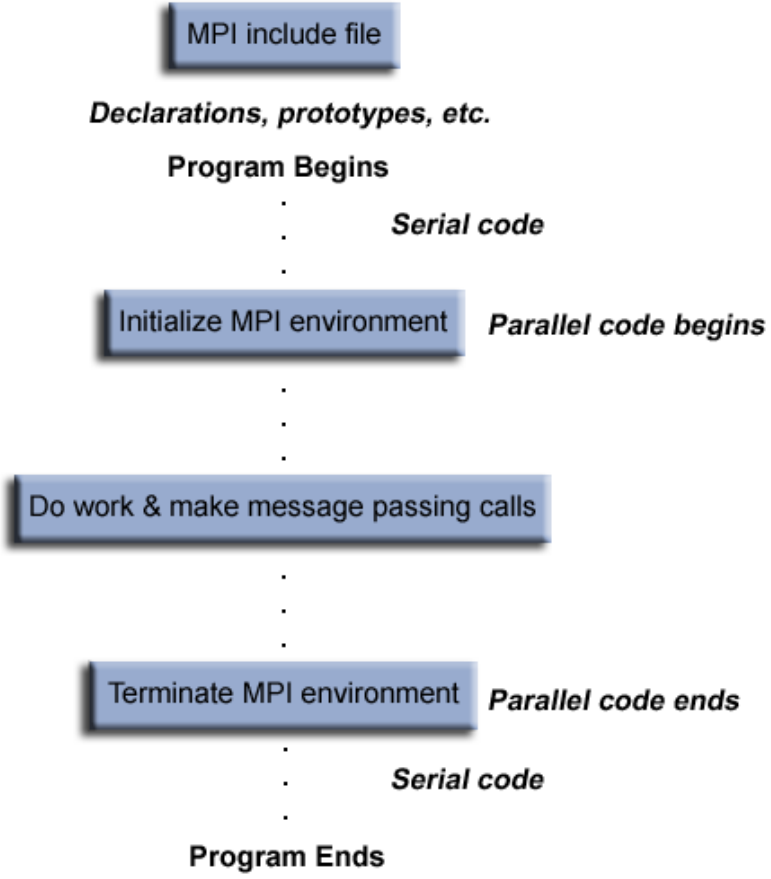
- **S**ingle **P**rogram executed by both the parent and child processes but with different halves of the same array (**M**ultiple **D**ata)
  - Parent calculate the sum of first half of the arrays whereas the child calculate the sum of the second half
- The problem(s)
  - How the child and parent would communicate their partial results to get the total sum?
  - What if we have a huge array size that could be faster to compute with multiple processes, some of which would be running on a shared memory whereas the rest on distributed memory?
    - Imagine the difficulty in writing such a program

SPMD Using the MPI



- Run the same program on P processing elements (PEs)
- Use the “rank” … an ID ranging from 0 to (P-1) … to determine what computation is performed on what data by a given PE
- Different PEs can follow different paths through the same code

General MPI Program Structure



- Write one program that will execute on each process
- Each of the N processes will have a unique rank (0 – N-1)
- Processes will communicate via messages for non-local data accesses

The Hello World Program in MPI (1/3)

```
// the header file containing MPI APIs
#include <mpi.h>
int main(int argc, char **argv) {
    // Initialize the MPI runtime
    MPI_Init(&argc, &argv);





    MPI_Finalize();
    return 0;
}
```

- MPI_Init and MPI_Finalize APIs are used for initializing and cleanup the MPI environment (runtime)

The Hello World Program in MPI (2/3)

```
// the header file containing MPI APIs
#include <mpi.h>
int main(int argc, char **argv) {
    // Initialize the MPI runtime
    MPI_Init(&argc, &argv);
    int rank, nprocs;
    // Get the total number of processes in MPI_COMM_WORLD
    MPI_Comm_size(MPI_COMM_WORLD, &nprocs);
    // Get the rank of this process in MPI_COMM_WORLD




    MPI_Finalize();
    return 0;
}
```

- Compile the program as "**mpicc program.c**" (first install MPI using **sudo apt-get install mpich**)
- Run the program as "**mpirun -np 4 ./a.out**" where 4 is the total number of MPI processes to be created
- MPI_COMM_WORLD is the **name** of the group that contains the processes created by the user using mpirun command (e.g., 4 above)
  - Total number of processes in the group (MPI_COMM_WORLD) can be enquired using the MPI_Comm_size API

The Hello World Program in MPI (3/3)

```
// the header file containing MPI APIs
#include <mpi.h>
int main(int argc, char **argv) {
    // Initialize the MPI runtime
    MPI_Init(&argc, &argv);
    int rank, nprocs;
    // Get the total number of processes in MPI_COMM_WORLD
    MPI_Comm_size(MPI_COMM_WORLD, &nprocs);
    // Get the rank of this process in MPI_COMM_WORLD
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    printf("My rank is %d in world of size %d\n", rank, nprocs);
    // Terminate the MPI runtime
    MPI_Finalize();
    return 0;
}
```

There is no user data being exchanged in this program. We will see message exchange in the next slides. Each process will simply print their rank and total number of processes invoked by the user

- Each of the processes will get a unique rank (id) in the group MPI_COMM_WORLD that can be enquired using the MPI_Comm_rank API
- MPI rank can be used to decide what will execute on which process

Parallel Array Sum Using MPI (1/4)

```
int main(int argc, char **argv) {
    int rank=0, nproc=4;
    MPI_Init(&argc, &argv);
    // 1. Get to know your world
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &nproc);

    MPI_Finalize();
}
```

- These four APIs are must in any parallel program being written using MPI
- There are more than 120 APIs in MPI, but we will only restrict our discussion to only six APIs

Parallel Array Sum Using MPI (2/4)

```
int main(int argc, char **argv) {
    int rank=0, nproc=4;
    MPI_Init(&argc, &argv);
    // 1. Get to know your world
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &nproc);
    int array[SIZE]; // initialized and assume (SIZE % nproc = 0)
    // 2. calculate local sum
    int my_sum = 0, chunk = SIZE/nproc;
    for (int i=rank*chunk; i<(chunk+1)*rank; i++) my_sum += array[i];



    MPI_Finalize();
}
```

- The array initialization is happening for each of the MPI process. Hence, each process has the exactly same array, but a separate copy in their own address space
  - In reality, the array could be read from disk in parallel by each processes (chunk size)
- For simplicity, we are assuming the array can be divided into equal sized chunk depending on the total number of MPI processes
- Each MPI process calculate the sum of its own chunk
- The variable “my_sum” is local to each process

Parallel Array Sum Using MPI (3/4)

```
int main(int argc, char **argv) {
    int rank=0, nproc=4;
    MPI_Init(&argc, &argv);
    // 1. Get to know your world
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &nproc);
    int array[SIZE]; // initialized and assume (SIZE % nproc = 0)
    // 2. calculate local sum
    int my_sum = 0, chunk = SIZE/nproc;
    for (int i=rank*chunk; i<(chunk+1)*rank; i++) my_sum += array[i];
    // 3. All non-root processes send result to root processes (rank=0)
    if(rank > 0) {
        MPI_Send(&my_sum, 1, MPI_INT, 0, 0, MPI_COMM_WORLD);
    }
    else { // executed only at rank=0





    }
    MPI_Finalize();
}
```

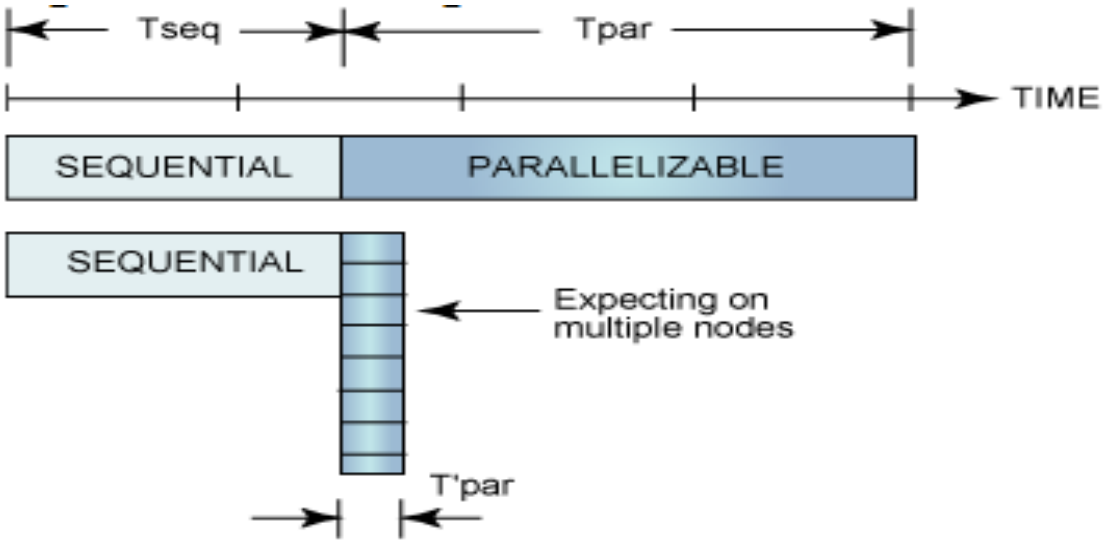
- Non-root rank use MPI_Send API to send the value of their local variable my_sum
- Parameters to MPI_Send are
  1. Address of the variable (or array) to be send
  2. Total count of data being sent (it will be the size of array when being sent)
  3. Platform independent name of the data type that MPI can understand (MPI_INT, MPI_FLOAT, MPI_DOUBLE, MPI_CHAR, etc.)
  4. Rank of the receiver process (0 in this case)
  5. Tag (you can ignore it for now and simply pass 0 at all places)
  6. The group name MPI_COMM_WORLD

Parallel Array Sum Using MPI (4/4)

```
int main(int argc, char **argv) {
    int rank=0, nproc=4;
    MPI_Init(&argc, &argv);
    // 1. Get to know your world
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &nproc);
    int array[SIZE]; // initialized and assume (SIZE % nproc = 0)
    // 2. calculate local sum
    int my_sum = 0, chunk = SIZE/nproc;
    for (int i=rank*chunk; i<(chunk+1)*rank; i++) my_sum += array[i];
    // 3. All non-root processes send result to root processes (rank=0)
    if(rank > 0) {
        MPI_Send(&my_sum, 1, MPI_INT, 0, 0, MPI_COMM_WORLD);
    }
    else { // executed only at rank=0
        int total_sum = my_sum, tmp;
        for(int src=1; src<nproc; src++) {
          MPI_Recv(&tmp, 1, MPI_INT, src, 0, MPI_COMM_WORLD, NULL);
          total_sum += tmp;
        }
    }
    MPI_Finalize();
}
```

- Root rank will use (nproc-1) number of MPI_Recv API to receive the message being sent by (nproc-1) number of processes
- Parameters to MPI_Recv are exactly same as in MPI_Recv with few minor changes
  - The first parameter is the buffer used to store the message receive from the destination. It must be of the same size and datatype as in the sender side
  - The fourth parameter is the rank of the sender
  - You can ignore the last parameter and simply pass it as NULL

Amdahl's Law



- If 50% of your application is parallel and 50% is serial, you can't get more than a factor of 2 speedup, no matter how many processors it runs on
  - $T_{Total} = T_{seq} + T_{par}$
  - With infinite processors (or cores), $T_{par} = 0$ (theoretically) implying $T_{Total} = T_{seq}$

Reading Material

- Tutorial on MPI by LLNL
  - https://hpc-tutorials.llnl.gov/mpi/

Next Lecture

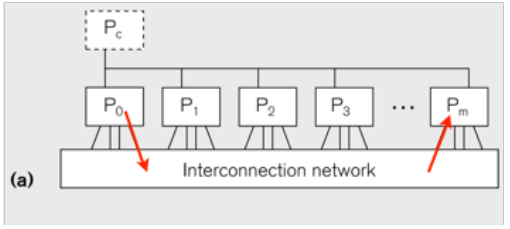
- Introduction to process scheduling

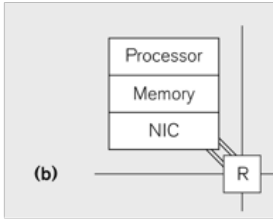
Lecture 10: Process Scheduler

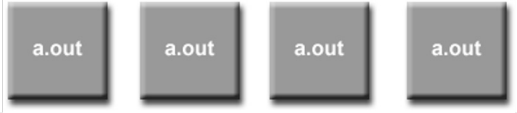
Vivek Kumar
Computer Science and Engineering
IIIT Delhi
vivekk@iiitd.ac.in

Last Lecture

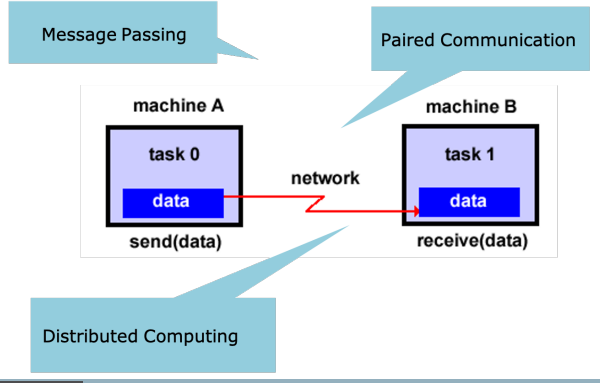
==> Processors communicate by sending messages via an interconnect







Process_ 1 (rank=0) | Process_ 2 (rank=1) | Process_ 3 (rank=2) | Process_ 4 (rank=3)




```
int main(int argc, char **argv) {
    int rank=0, nproc=4;
    MPI_Init(&argc, &argv);
    // 1. Get to know your world
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &nproc);
    int array[SIZE]; // initialized and assume (SIZE % nproc = 0)
    // 2. calculate local sum
    int my_sum = 0, chunk = SIZE/nproc;
    for (int i=rank*chunk; i<(chunk+1)*rank; i++) my_sum += array[i];
    // 3. All non-root processes send result to root processes (rank=0)
    if(rank > 0) {
        MPI_Send(&my_sum, 1, MPI_INT, 0, 0, MPI_COMM_WORLD);
    }
    else { // executed only at rank=0
        int total_sum = my_sum, tmp;
        for(int src=1; src<nproc; src++) {
          MPI_Recv(&tmp, 1, MPI_INT, src, 0, MPI_COMM_WORLD, NULL);
          total_sum += tmp;
        }
    }
    MPI_Finalize();
}
```

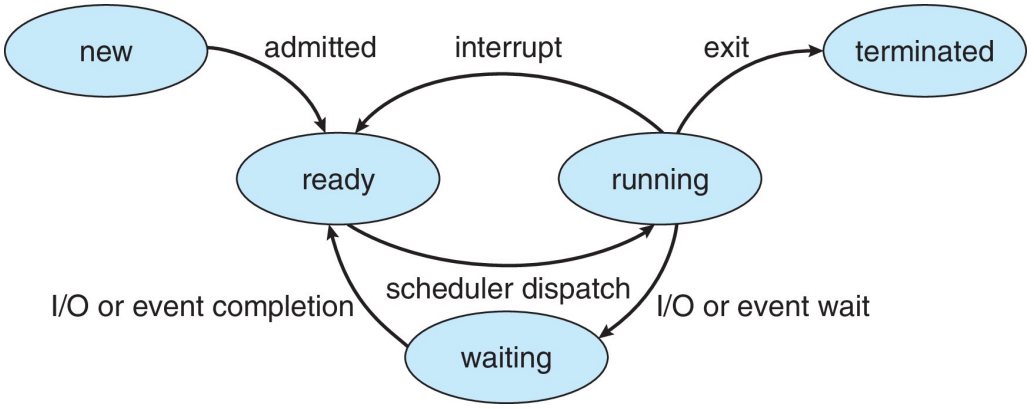
- Several machines (e.g., workstations) connected together in a cluster with some interconnect
- One big computation can be divided equally into N number of sub-computations that can run on N different processes
- Processes use MPI to communicate in distributed memory by exchanging messages with each other
- Amdhal's law: if 50% of your application is parallel and 50% is serial, you can't get more than a factor of 2 speedup, no matter how many processors it runs on

Today's Class

- Introduction to process scheduling
  - Context switch
- Short discussion on student feedback

In this course, we will not have any assignment involving carrying out changes inside any operating systems. Still, to understand some concepts, we would revisit well-known OS, such as xv6 OS – as in today's lecture!

What is the Process State?



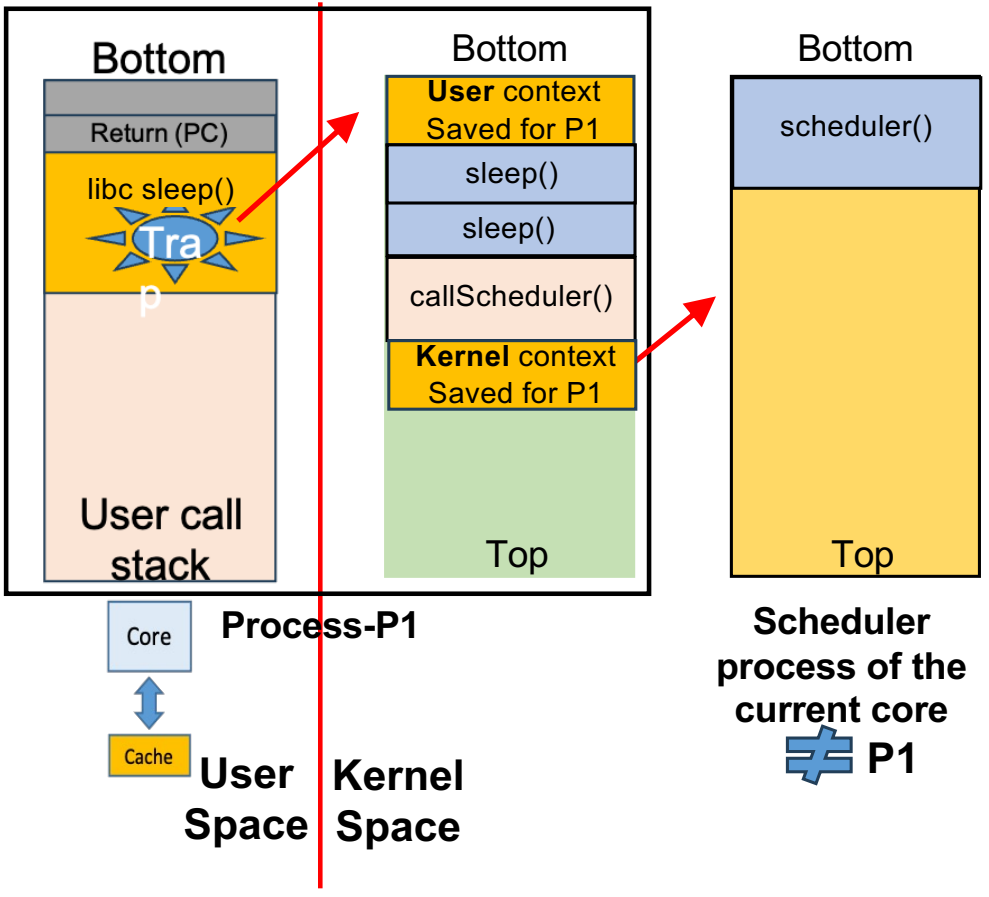
Assume infinite number of CPU cores

1. Writer process trying to write on pipe that is currently full
2. Sleep
3. Wait
4. libc scanf waiting for user input
5. Recursive Fibonacci number calculation
6. Process inside sem_wait with negative semaphore
7. Process inside sem_post

How OS Can Juggle Between Processes?

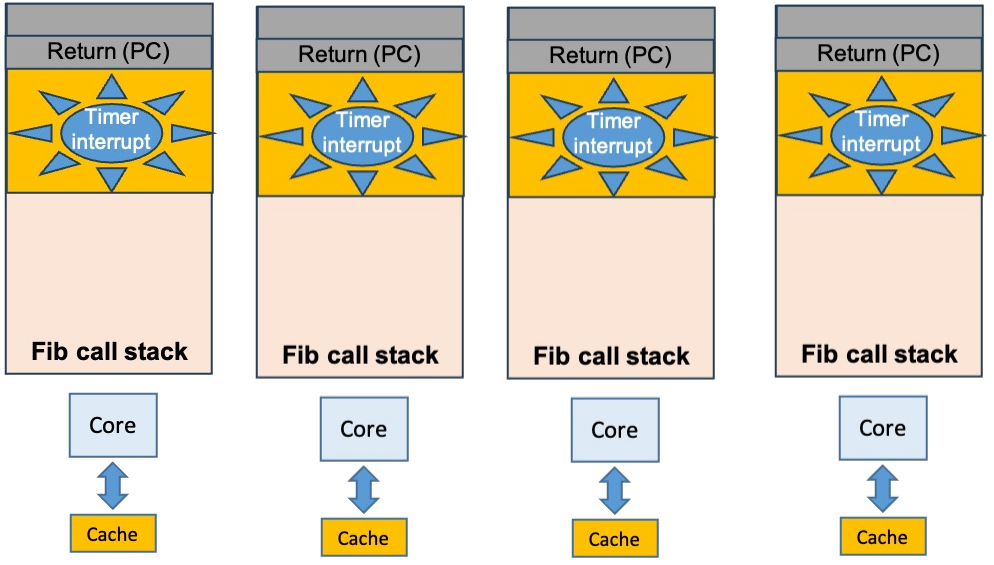
- Case-1: Whenever a process is waiting for some resource
  - It does not make any sense to let a blocked process running on the CPU (e.g., waiting for IO, sem_wait, reading from an empty pipe, etc.)
    - Even if the CPUs are idle
  - OS can safely move the process from the running to waiting queue
    - Does not affect the performance as if CPUs are idle, the process can be brought back into the running queue as soon as it unblocks
- Case-2: If a process is not blocking for some resource
  - Timer interrupt is generated by the CPU
    - It is a hardware generated interrupt at a fixed interval where the interval duration depends upon the processor architecture
  - It is used to bring another process into the running queue when the currently executing process is not blocking

Case-1: Process is Going to Block



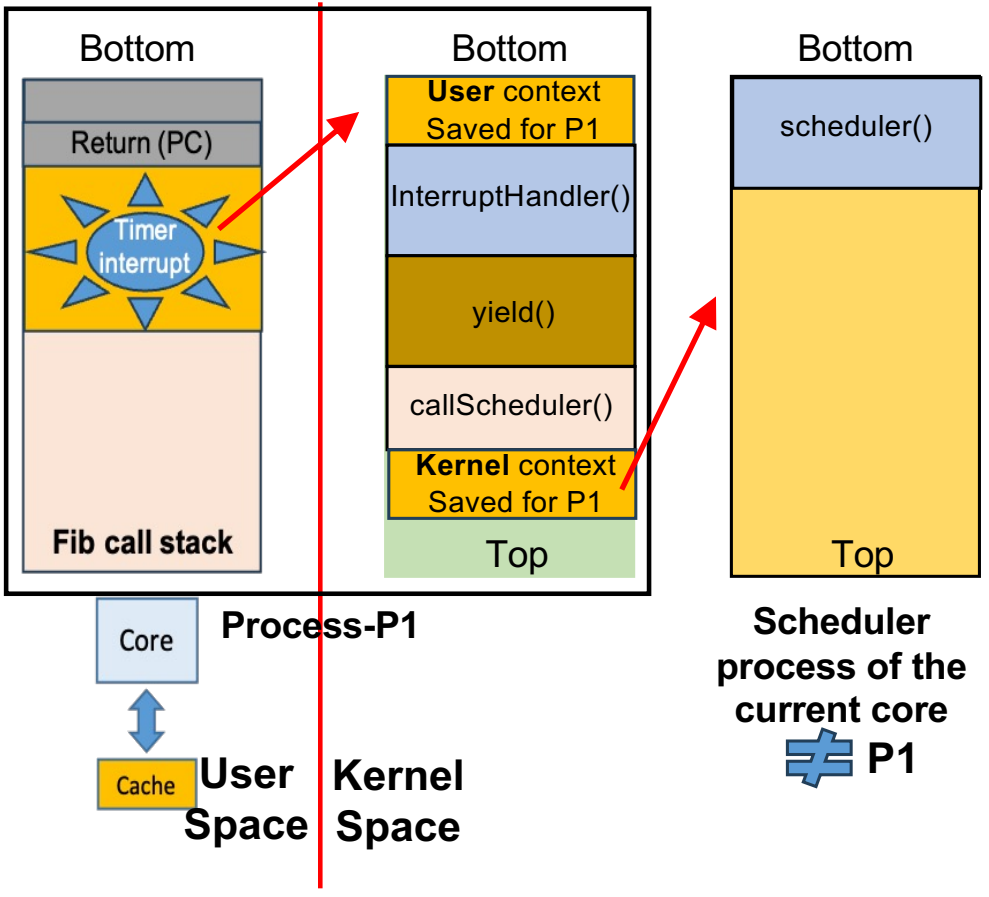
- The user process P1 calls a blocking API (e.g., **sleep** on an empty pipe)
- P1 switches its execution from the user stack to its kernel stack (each user process has its own kernel stack)
  - User context is saved on the bottom of the kernel stack and interrupt handler is invoked
  - The blocking system call will eventually call some sleep implementation
- The sleep will invoke a call to the CPU scheduler, which will: a) save the kernel context of the user process at the top of the kernel stack, b) load the context of the CPU scheduler from the corresponding stack, and c) jump to the CPU scheduler stack where it will call the schedule() routine
- Each CPU has its own CPU scheduler process that runs on a separate kernel stack than that of the process it was originally executing
  - Why?

Case-2: Process is Not Going to Block



- There could be several Fibonacci processes being displayed in **running** state from htop on a 4-core processor
  - However, only four of them are **actually** in running state any given time (1-1 mapping between a process and a core)
- Timer interrupt (IDT index 32 on x86) is generated by the hardware after every fixed intervals

Case-2: Process is Not Going to Block



- Timer interrupt moves the process execution from user stack to its kernel stack (each user process has its own kernel stack)
- User context is saved on the bottom of the kernel stack and interrupt handler is invoked
- The handler will call yield() method as it was a timer interrupt
- The **yield** will invoke a call to the CPU scheduler, which will: a) save the kernel context of the user process at the top of the kernel stack, b) load the context of the CPU scheduler from the corresponding stack, and c) jump to the CPU scheduler stack where it will call the schedule() routine
- Each CPU has its own CPU scheduler process that runs on a separate kernel stack than that of the process it was originally executing

The Context Switch – swtch(OldCtx, NewCtx)

- Typical steps for calling a context switch on x86
  1. Load the parameters passed to the context switching routine in registers, as it is currently on the kernel stack of the user process
     - Parameters would be the context of current process that has to be saved, and the context of the scheduler process that has to be restored
     - These parameters would no longer be accessible as the ESP will not be pointing to the old stack after context switch
  2. Save context of the user process
     - **a)** **Save** the callee save registers (EBX, ESI, and EDI), EBP, and ESP on the "**top**" of user process's kernel stack
     - b) No need to explicitly save EIP as it was already saved by x86 procedure calling convention (see lecture 04 slides)
  3. Load context of the scheduler process
     - **a)** **Load** the callee save registers (EBX, ESI, and EDI), EBP, and ESP from the "**top**" of scheduler process stack
     - b) Load EIP from scheduler stack and jump to that instruction address

The Juggling API: `scheduler`

- **Basic questions about the design**
  - How many scheduler processes should be there in an OS?
  - How many copies of process table should be there in the OS?
    - Process table is an array of PCB
  - Can there be any race on the process table accesses?
    - How to resolve?

Who Calls the scheduler

```
kernel_main() {
    setup();
    init_all_cpus();
    start_init_process(); // on CPU-0
    for(int cpu=1; cpu<numCPUs; cpu++) {
        launch(cpu, scheduler);
    }
    scheduler(); // on CPU-0
}
```

- OS is booted with the help of another program called bootloader
  - Reads the kernel ELF executable
  - Loads the program segments
  - Jump to the entry point (kernel_main)
  - Call never returns to bootloader (similar to exec)
- Kernel main
  - Complete the basic setup (initializing CPUs, memory, disk, registers, etc.)
  - Launch the first user process
  - Launch one scheduler process on each CPU that will eventually pick a process from the process table and start executing it until it gives up the CPU (see case-1 and case-2 in previous slides)

The Juggling API: `scheduler` Pseudocode

```
void scheduler() {
    while(true) {

        foreach(Process p:                    (process_table)) {




            swtch(scheduler_process, p);
            // p is done for now..

        }

    }
}
```

- Scheduler job is to simply pick a process from the process table
- Context switch of the scheduler process with the process picked from the process table
- Execute the process from it was previously stopped
- When the control will be passed back to scheduler from this process?

The Juggling API: `scheduler` Pseudocode

```
void scheduler() {
    while(true) {

        foreach(Process p:                    (process_table)) {
            if(p->state != READY) {
                continue;
            }
            p->state = RUNNING;

            swtch(scheduler_process, p);
            // p is done for now..

        }

    }
}
```

- The scheduler should pay attention that the process picked up from the process table is in READY state (should not be in WAITING state)
- It should change the state of this process from READY to RUNNING
- When control is passed back to the scheduler from this process, its state would have changed to either READY or WAITING

The Juggling API: `scheduler` Pseudocode

```
void scheduler() {
    while(true) {
        lock(process_table);
        foreach(Process p:                    (process_table)) {
            if(p->state != READY) {
                continue;
            }
            p->state = RUNNING;
            unlock(process_table);
            swtch(scheduler_process, p);
            // p is done for now..
            lock(process_table);
        }
        unlock(process_table);
    }
}
```

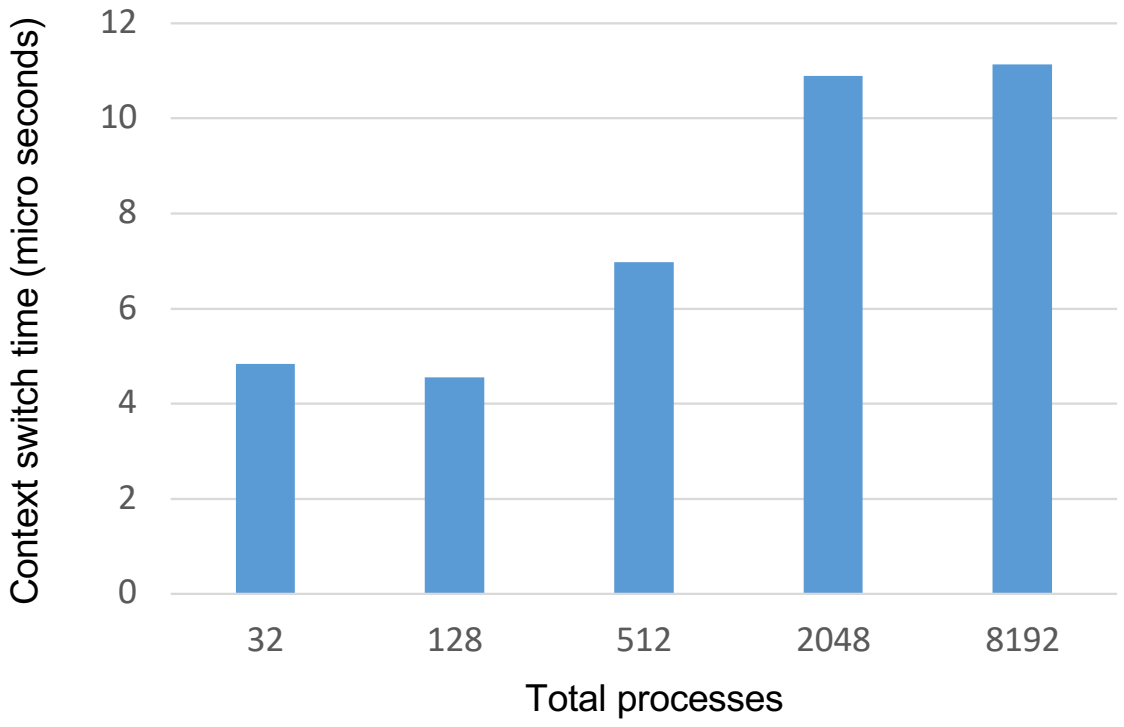
- There is a single process table on a system irrespective of the number of CPUs
- Proper locking mechanism must be ensured to avoid any race condition on the process table by the scheduler processes running at other CPUs
- Modern OS may use scalable implementations of locking to improve performance

The Juggling API: `scheduler` Pseudocode

```
void scheduler() {
    while(true) {
        lock(process_table);
        foreach(Process p: scheduling_algorithm(process_table)) {
            if(p->state != READY) {
                continue;
            }
            p->state = RUNNING;
            unlock(process_table);
            swtch(scheduler_process, p);
            // p is done for now..
            lock(process_table);
        }
        unlock(process_table);
    }
}
```

- Process scheduling algorithm plays an important role in the design of operating system
- Different algorithms are chosen according to the need

Context Switch: Cost?



- Context switch overhead measured on an AMD EPYC 32-core processor running Ubuntu 18.04.3 LTS
  - Data generated using lmbench benchmark (./lat_ctx –s 0 32 128 512 2048 8192)
- Overheads
  - Timer interrupt latency
  - Saving restoring context
  - Process scheduling
  - Loss in cache locality

Reference Material

- https://pdos.csail.mit.edu/6.828/2018/xv6/book-rev11.pdf
  - You may read the chapter-5 just to have a high-level understanding of scheduling and context switch in a simple OS
  - You can avoid referring to the xv6 codebase as we will not have any assignment on reading/modifying xv6 code

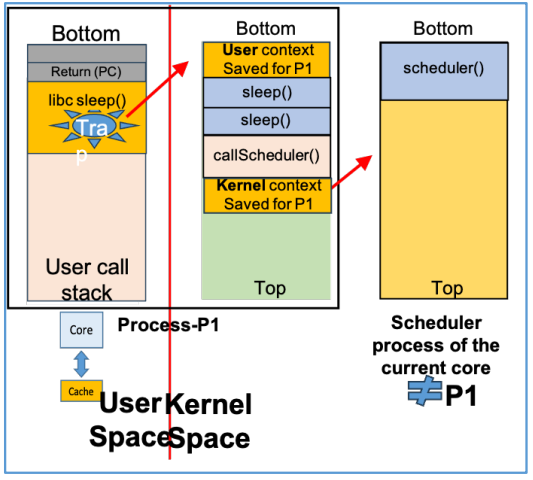
Next Lecture (Tuesday)

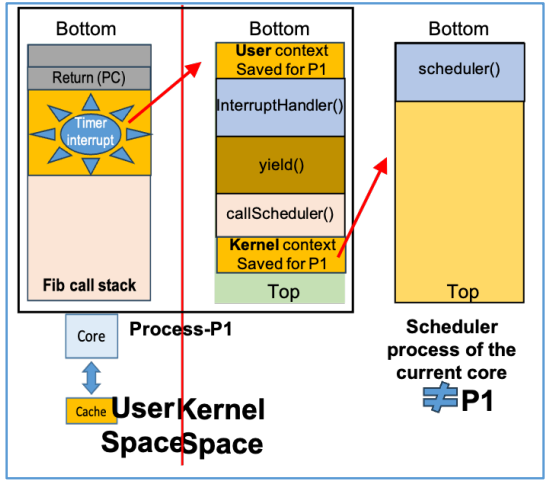
- Process scheduling policies
- Extra lecture in the tutorial slot next week (Wednesday)

Lecture 11: Scheduling Policies (Part-1)

Vivek Kumar
Computer Science and Engineering
IIIT Delhi
vivekk@iiitd.ac.in

Last Lecture





- Juggling with two different types of processes
  - CPU bound
  - IO bound / blocking APIs
- Context switch swtch(OldCtx, NewCtx)
  - Load parameter in registers
  - Save user context (oldCtx) from the registers into top of user process's kernel stack
  - Load scheduler process's context into registers from the top of scheduler process stack
  - Load EIP from scheduler stack and resume execution of the **scheduler**

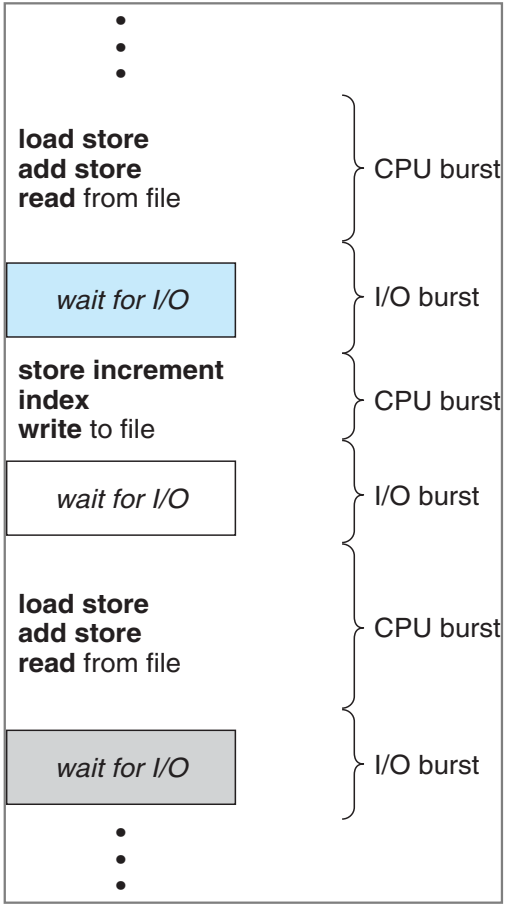
```
void scheduler() {
    while(true) {
        lock(process_table);
        foreach(Process p: scheduling_algorithm(process_table)) {
            if(p->state != READY) {
                continue;
            }
            p->state = RUNNING;
            unlock(process_table);
            swtch(scheduler_process, p);
            // p is done for now..
            lock(process_table);
        }
        unlock(process_table);
    }
}
```

Today's Class

- First In First Out / First Come First Serve
- Shortest Job First
- Shortest Time to Completion First
- Round Robin

Acknowledgements: Today's lecture content derived from the CS162 course offered at University of California, Berkeley

CPU and I/O Bursts



- Process execution model: programs alternate between bursts of CPU and I/O
  - Program typically uses the CPU for some period of time, then does I/O, then uses CPU again
  - Each scheduling algorithm is about which process to give to the CPU for use by its next CPU burst
  - With time slicing, process may be forced to give up CPU before finishing current CPU burst

Key Scheduling Goals

- **Response Time**
  - What user sees: time to echo a keystroke in editor
- **Throughput**
  - Total operations per second
    - Affected by the overheads of context switching
- **Fairness**
  - Conflicts with the response time
- **Starvation**
  - Due to improper resource allocation policy

First-Come, First-Served (FCFS)

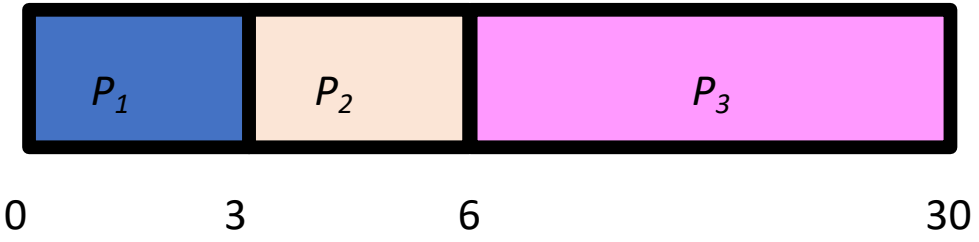
Run tasks in order of arrival.

Run task until completion (or blocks on IO).
No preemption

Also called FIFO

First-Come, First-Served (FCFS)

Process	Burst Time
P_1	3
P_2	3
P_3	24

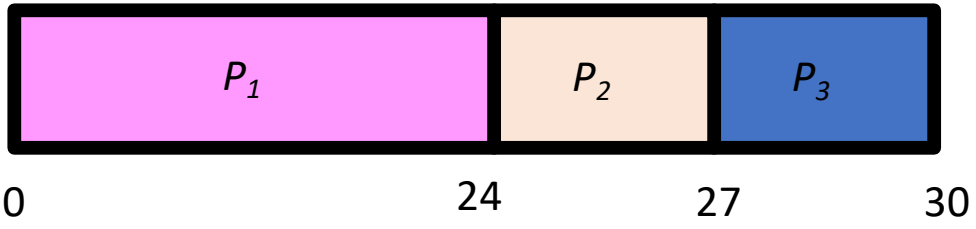


What is the average completion time? $\left(\frac{3+6+30}{3} = 13\right)$

What is the average waiting time? $\left(\frac{0+3+6}{3} = 3\right)$

First-Come, First-Served (FCFS)

Process	Burst Time
P_3	24
P_2	3
P_1	3

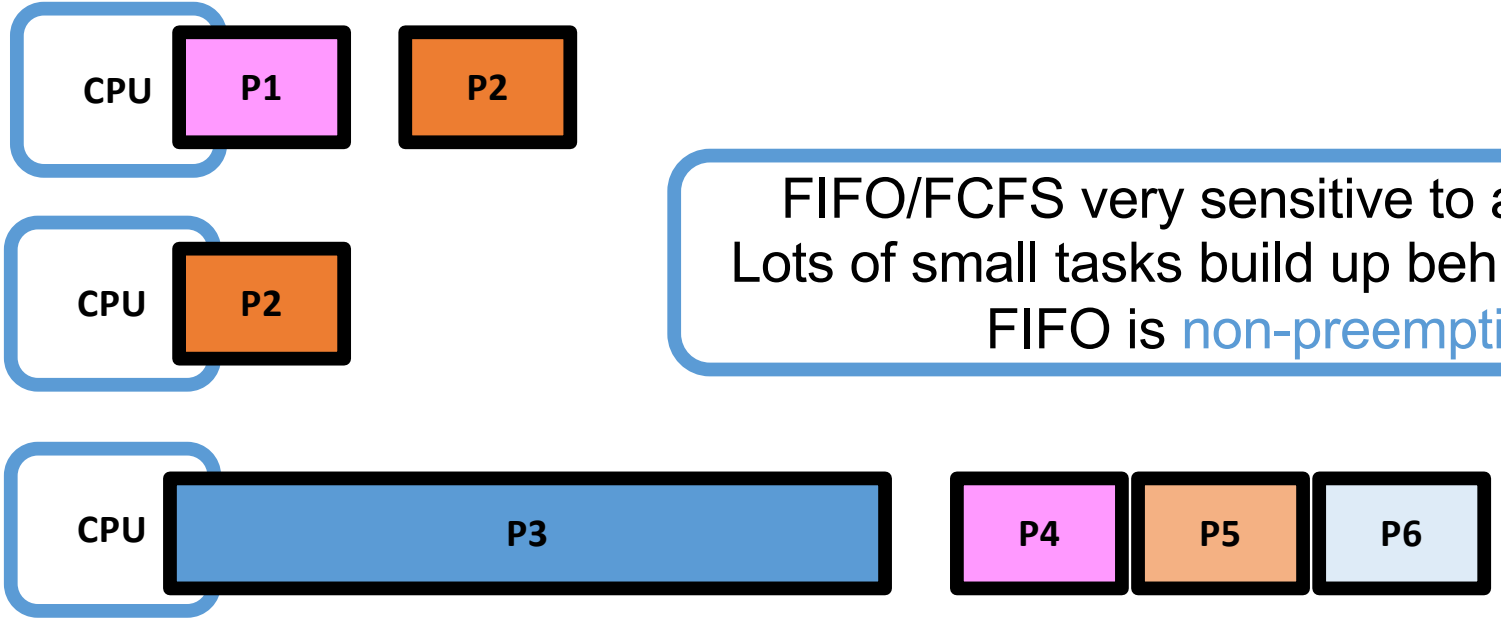


What is the average completion time? $\left(\frac{24+27+30}{3} = 27\right)$

What is the average waiting time? $\left(\frac{0+24+27}{3} = 17\right)$

Can there be Convoy Effect in FCFS?

Convoy effect: Short process stuck behind long process



FIFO/FCFS very sensitive to arrival order
Lots of small tasks build up behind long tasks
FIFO is non-preemptible

Can FIFO lead to starvation?

Today's Class

- First In First Out / First Come First Serve
- Shortest Job First
- Shortest Time to Completion First
- Round Robin

Acknowledgements: Today's lecture content derived from the CS162 course offered at University of California, Berkeley

Shortest Job First

How can we minimize average completion time?

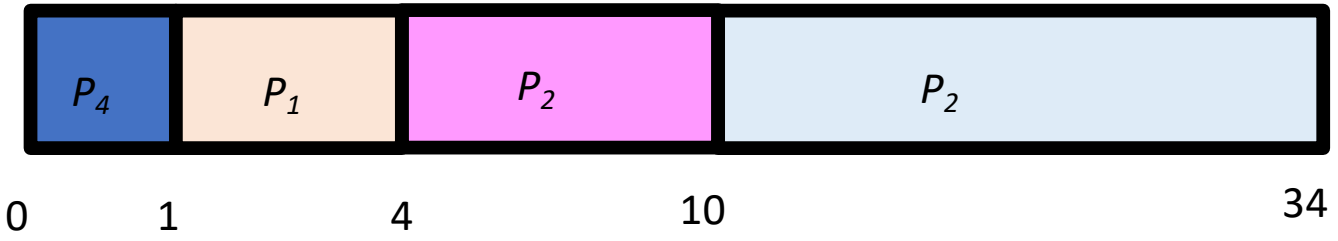
By scheduling jobs in order of

estimated completion time

Crooks CS162 © UCB Fall 2022

Shortest Job First

Process	Burst Time
P_1	3
P_2	6
P_3	24
P_4	1

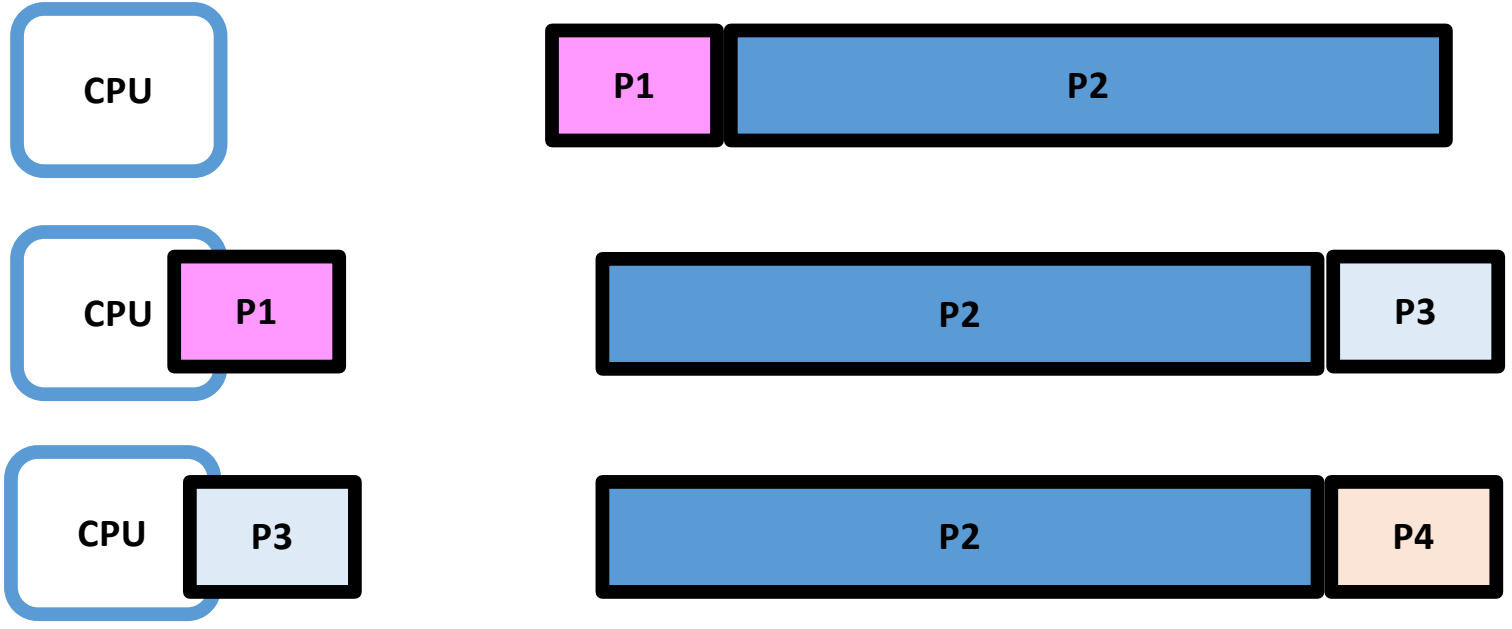


What is the average completion time?

$$\left(\frac{1+4+10+34}{4} = 12.25\right)$$

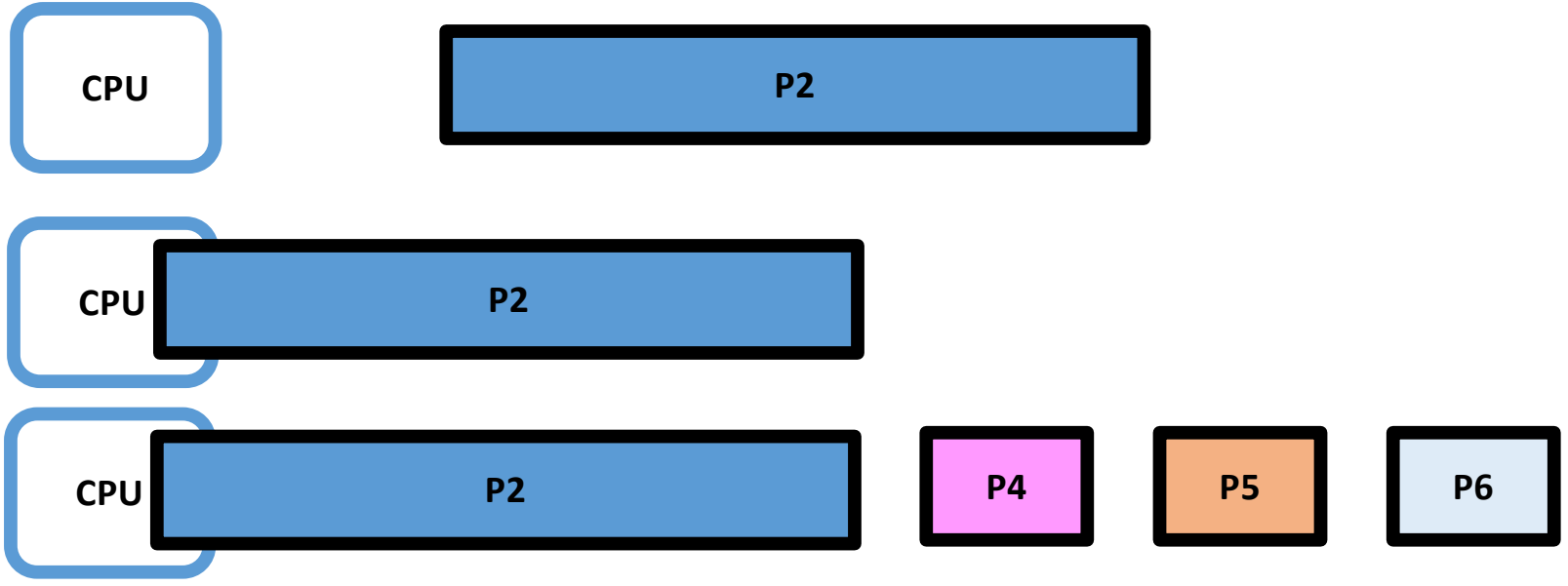
Can SJF Lead to Starvation?

Any scheduling policy that always favours a fixed property for scheduling leads starvation



Can there be Convoy Effect in SJF?

Any non-preemptible scheduling policy suffers from convoy effect



Today's Class

- First In First Out / First Come First Serve
- Shortest Job First
- Shortest Time to Completion First
- Round Robin

Acknowledgements: Today's lecture content derived from the CS162 course offered at University of California, Berkeley

Shortest Time to Completion First (STCF)

Introduce the notion of preemption

A running task can be de-scheduled before completion.

STCF

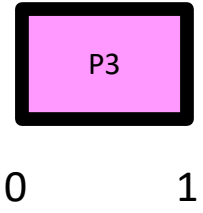
Schedule the task with the least amount of time left

Crooks CS162 © UCB Fall 2022

Shortest Time to Completion First (STCF)

Schedule the task with the least amount of time left

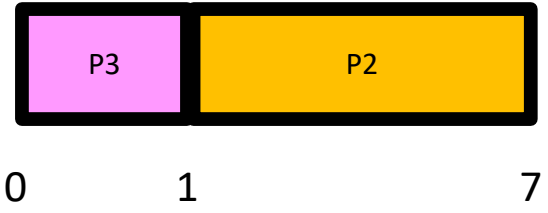
Process	Burst Time (left)	Arrival Time
P_1	3	*10*
P_2	6	*1*
P_3	24	*0*
P_4	16	*18*



Shortest Time to Completion First (STCF)

Schedule the task with the least amount of time left

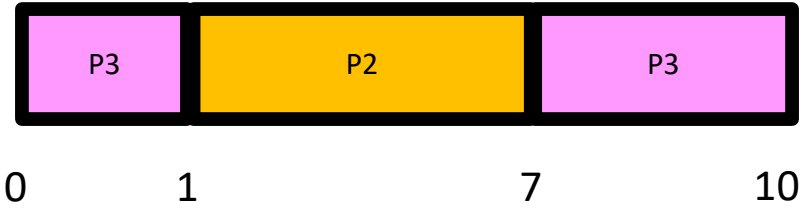
Process	Burst Time (left)	Arrival Time
P_1	3	*10*
P_2	6	*1*
P_3	23	*0*
P_4	16	*18*



Shortest Time to Completion First (STCF)

Schedule the task with the least amount of time left

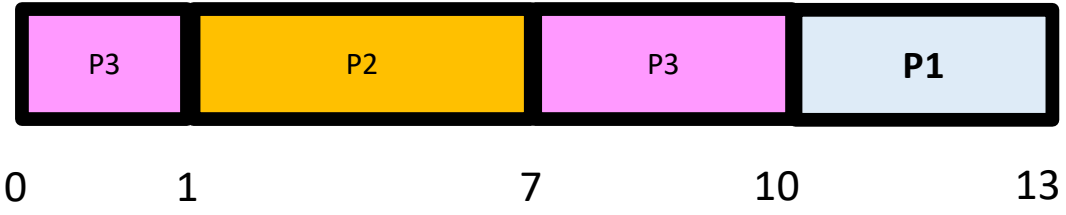
Process	Burst Time (left)	Arrival Time
P_1	3	*10*
P_2	0	*1*
P_3	23	*0*
P_4	16	*20*



Shortest Time to Completion First (STCF)

Schedule the task with the least amount of time left

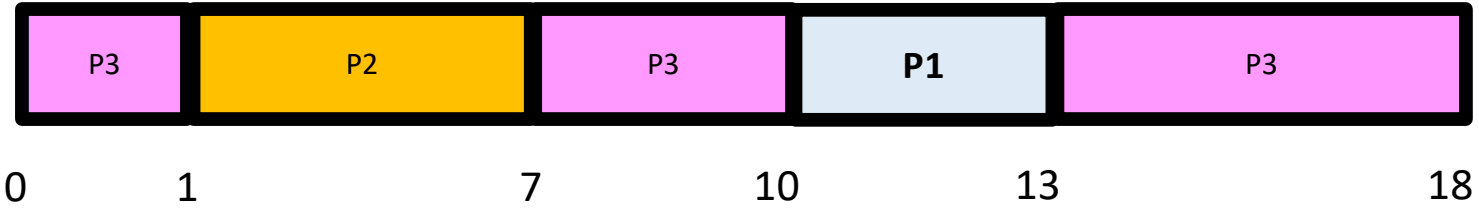
Process	Burst Time (left)	Arrival Time
P_1	3	*10*
P_2	0	*1*
P_3	20	*0*
P_4	16	*18*



Shortest Time to Completion First (STCF)

Schedule the task with the least amount of time left

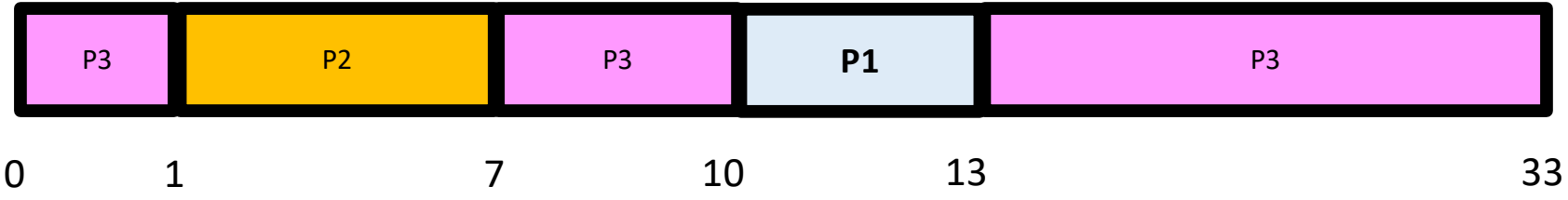
Process	Burst Time (left)	Arrival Time
P_1	0	*10*
P_2	0	*1*
P_3	15	*0*
P_4	16	*18*



Shortest Time to Completion First (STCF)

Schedule the task with the least amount of time left

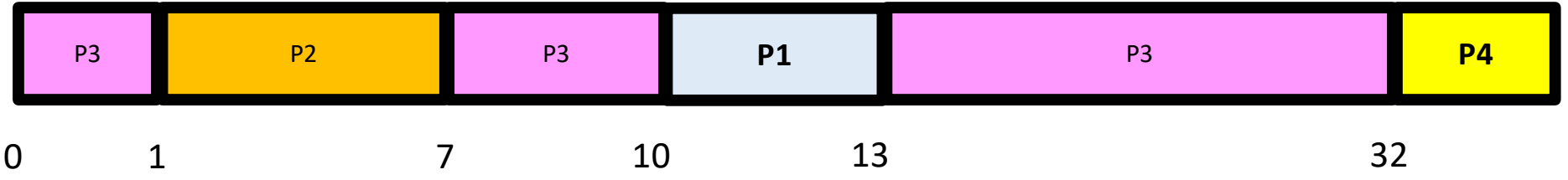
Process	Burst Time (left)	Arrival Time
P_1	0	*10*
P_2	0	*1*
P_3	0	*0*
P_4	15	*18*



Shortest Time to Completion First (STCF)

Schedule the task with the least amount of time left

Process	Burst Time (left)	Arrival Time
P_1	0	*10*
P_2	0	*1*
P_3	0	*0*
P_4	15	*18*



Are we done?

Can STCF lead to starvation?

Yes

Any scheduling policy that always favours a fixed property for scheduling leads starvation

No change!

Crooks CS162 © UCB Fall 2022

Are we done?

Is STCF subject to the convoy effect?

No!

STCF is a preemptible policy

Summary

If jobs arrive simultaneously

Always

Property	FCFS	SJF	STCF
Optimize Average Completion Time	✗	✓	✓
Prevent Starvation	✓	✗	✗
Prevent Convoy Effect	✗	✗	✓
Psychic Skills **NOT** Needed	✓	✗	✗

Preemptible

Process execution time to be known

Any scheduling policy that always favors a fixed property for scheduling leads to starvation

Can we design a non-psychic, starvation-free scheduler with good response time?

Today's Class

- First In First Out / First Come First Serve
- Shortest Job First
- Shortest Time to Completion First
- Round Robin

Acknowledgements: Today's lecture content derived from the CS162 course offered at University of California, Berkeley

Round-Robin Scheduling

RR runs a job for a **time slice**

(a **scheduling quantum**)

Once time slice over,

Switch to next job in ready queue.

=> Called time-slicing

Crooks CS162 © UCB Fall 2022

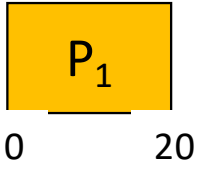
RR with Time Quantum = 20

Process	Burst Time
P_1	53
P_2	8
P_3	68
P_4	24

Crooks CS162 © UCB Fall 2022

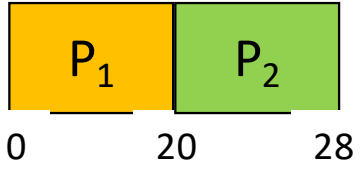
RR with Time Quantum = 20

Process	Burst Time
P_1	53 => 33
P_2	8
P_3	68
P_4	24



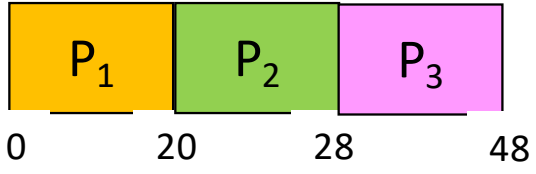
RR with Time Quantum = 20

Process	Burst Time
P_1	33
P_2	8 => 0
P_3	68
P_4	24



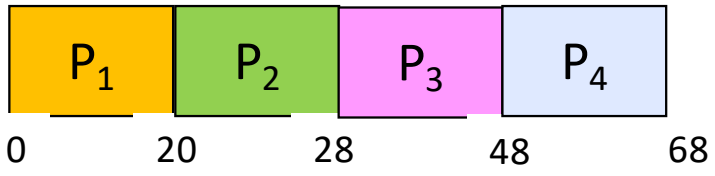
RR with Time Quantum = 20

Process	Burst Time
P_1	33
P_2	0
P_3	68 => 48
P_4	24



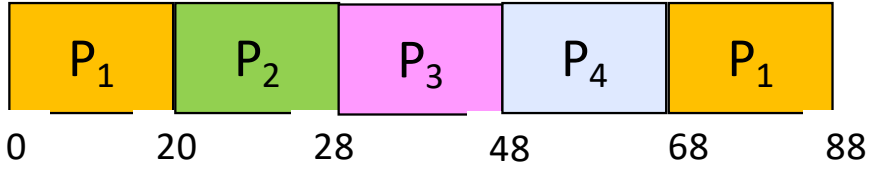
RR with Time Quantum = 20

Process	Burst Time
P_1	33
P_2	0
P_3	48
P_4	24 => 4



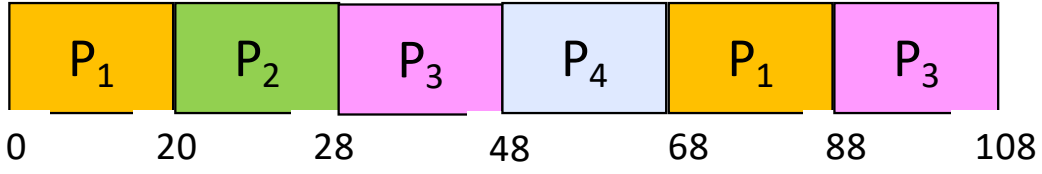
RR with Time Quantum = 20

Process	Burst Time
P_1	33 => 13
P_2	0
P_3	48
P_4	4



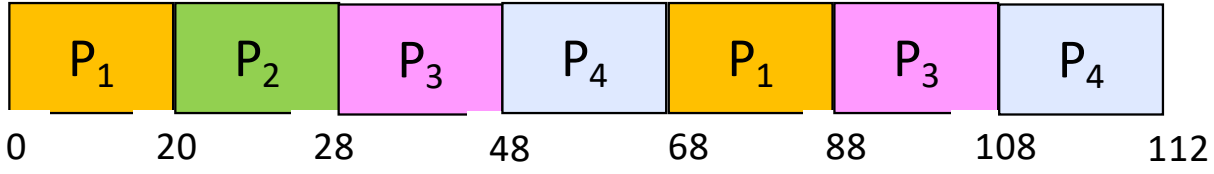
RR with Time Quantum = 20

Process	Burst Time
P_1	13
P_2	0
P_3	48 => 28
P_4	4



RR with Time Quantum = 20

Process	Burst Time
P_1	13
P_2	0
P_3	28
P_4	4 => 0



RR with Time Quantum = 20

Waiting time

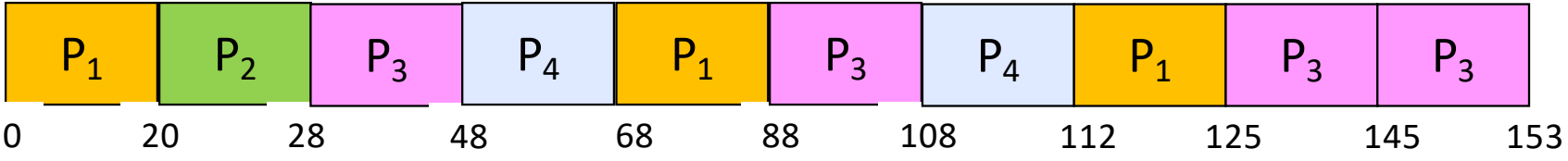
- P_1= 0 + (68-20)+(112-88)=72
- P_2=(20-0)=20
- P_3=(28-0)+(88-48)+(125-108)+0=85
- P_4=(48-0)+(108-68)=88

Average waiting time

$$\left(\frac{72+20+85+88}{4} = 66.25\right)$$

Average completion time

$$\left(\frac{125+28+153+112}{4} = 104.25\right)$$

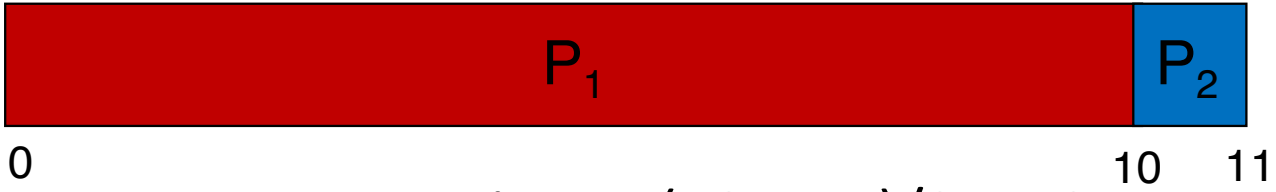


RR Quantum

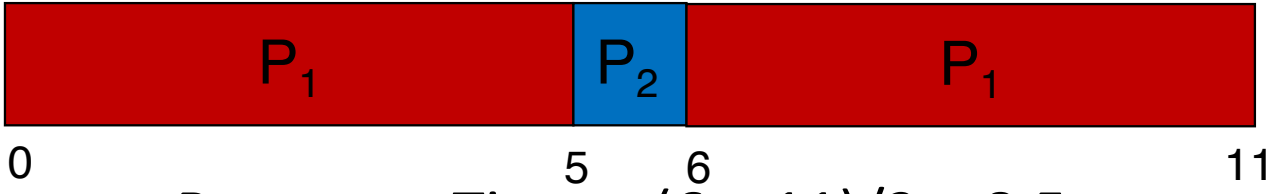
- Assume there is no context switching overhead
- What happens to average response time when we *decrease Q*?

Decrease Completion Time

- P_1: Burst Length 10
- P_2: Burst Length 1
- Q = 10



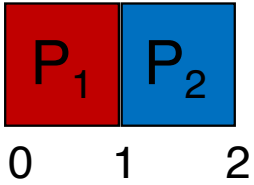
- Average Response Time = (10 + 11)/2 = 10.5
- Q = 5



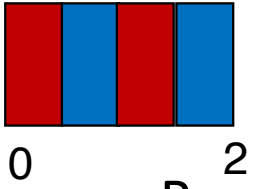
- Average Response Time = (6 + 11)/2 = 8.5

Increase Response Time

- P_1: Burst Length 1
- P_2: Burst Length 1
- $Q = 1$



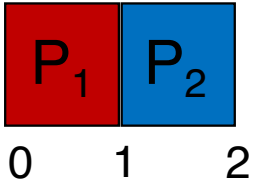
- Average Response Time = (1 + 2)/2 = 1.5
- $Q = 0.5$



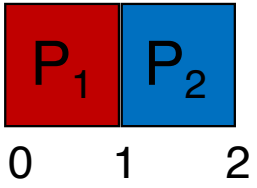
- Average Response Time = (1.5 + 2)/2 = 1.75

Same Response Time

- P_1: Burst Length 1
- P_2: Burst Length 1
- Q = 10



- Average Response Time = (1 + 2)/2 = 1.5
- Q = 1



- Average Response Time = (1 + 2)/2 = 1.5

Switching is not free!

Small scheduling quantas lead to

frequent context switches

- Context switch overhead
- Trash cache-state

q must be large with respect to context switch,

otherwise overhead is too high

Crooks CS162 © UCB Fall 2022

Are we done?

Can RR lead to starvation?

No

Crooks CS162 © UCB Fall 2022

Are we done?

Can RR suffer from convoy effect?

No

Only run a time-slice at a time

Crooks CS162 © UCB Fall 2022

Summary

Property	FCFS	SJF	STCF	RR
Optimize Average **Completion** Time	✗	✓	✓	✗
Optimize Average **Waiting** Time	✗	✗	✗	✓
Prevent Starvation	✓	✗	✗	✓
Prevent Convoy Effect	✗	✗	✓	✓
Psychic Skills **NOT** Needed	✓	✗	✗	✓

High in case of long running processes

Doesn't favor any fixed property

Due to time slicing

Crooks CS162 © UCB Fall 2022

Next Lecture (Tomorrow 8.30am)

- Scheduling policies (Part-2)

Lecture 12: Scheduling Policies (Part-2)

Vivek Kumar
Computer Science and Engineering
IIIT Delhi
vivekk@iiitd.ac.in

Last Lecture

- Shortest Job First
- Shortest Time to Completion First
- Round Robin

Property	FCFS	SJF	STCF	RR
Optimize Average **Completion** Time	✗	✓	✓	✗
Optimize Average **Waiting** Time	✗	✗	✗	✓
Prevent Starvation	✓	✗	✗	✓
Prevent Convoy Effect	✗	✗	✓	✓
Psychic Skills **NOT** Needed	✓	✗	✗	✓

Today's Class

- Multi-level feedback scheduling
- Linux Completely Fair Scheduler (CFS)

Acknowledgements: Today's lecture content derived from CS162 course at University of California, Berkeley

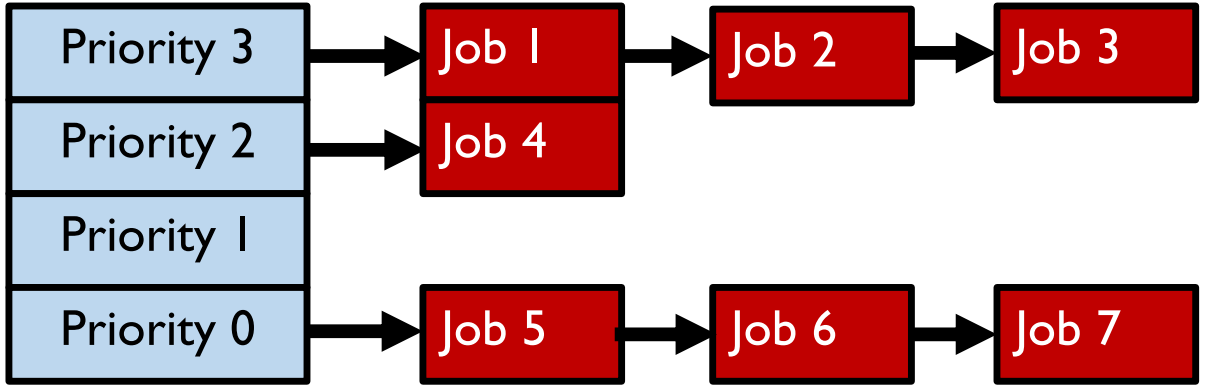
What we Want?

It cannot be achieved without having psychic skills

Would it be true for IO/interactive jobs having short CPU bursts?

Property	STCF	RR
Optimize Average **Completion** Time	✓	✗
Optimize Average **Waiting** Time	✗	✓
Prevent Starvation	✗	✓
Prevent Convoy Effect	✓	✓
Psychic Skills **NOT** Needed	✗	✓

Approximation of CPU Bursts using Priority Scheduling



- Priority scheduling always run the **ready** process (or job) with *highest priority*
- Systems may try to set priorities according to some **policy goal**
  - Example: Give IO/interactive jobs higher priority than long calculation
- How to achieve fairness and avoid starvation?
  - Elevate priority of threads that don't get CPU time

Multi-level Feedback Queue

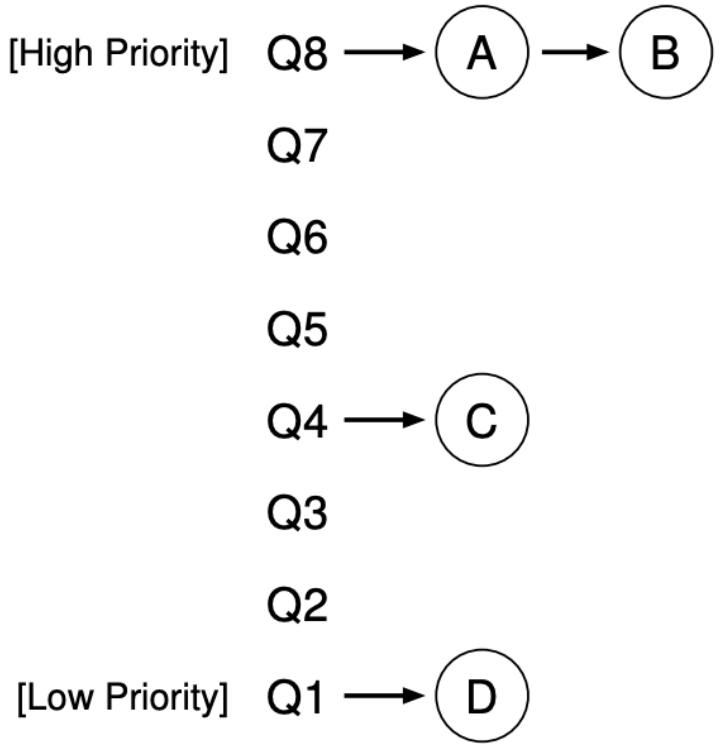


Figure 8.1: **MLFQ Example**

- Create distinct **queues for ready processes**, each assigned a different **priority level**
- All processes belong to any one queue at any given time
- Processes can move between queues
- Use priorities to decide from which queue process should be picked next
- Individual queues run RR with increasing time slice

Crooks CS162 © UCB Fall 2022

MLFQ (Naïve Version)

Rule 1

If Priority(A) > Priority(B) (different queues)

A runs (B doesn't).

Rule 2

If Priority(A) = Priority(B), A & B run in RR.

Key question:

How do you set the priorities?

Vary the priority of a job based on its *observed behavior*

Crooks CS162 © UCB Fall 2022

Learning behavior

Rule 3

When a job enters the system, it is placed at the highest priority (the topmost queue).

Rule 4a

If a job uses up an entire time slice while running, its priority is *reduced* (i.e., it moves down one queue).

Crooks CS162 © UCB Fall 2022

Learning behavior

P_1

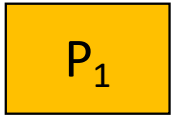
Long running CPU-bound

q= 10 ms

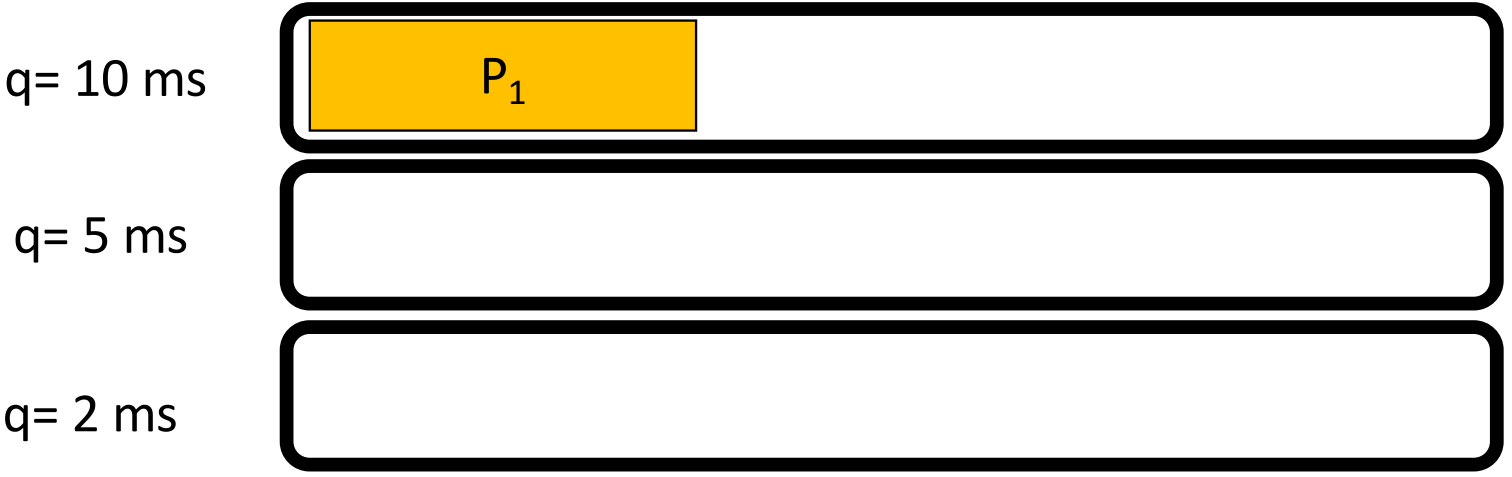
q= 5 ms

q= 2 ms

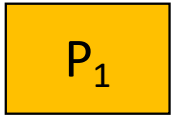
Learning behavior



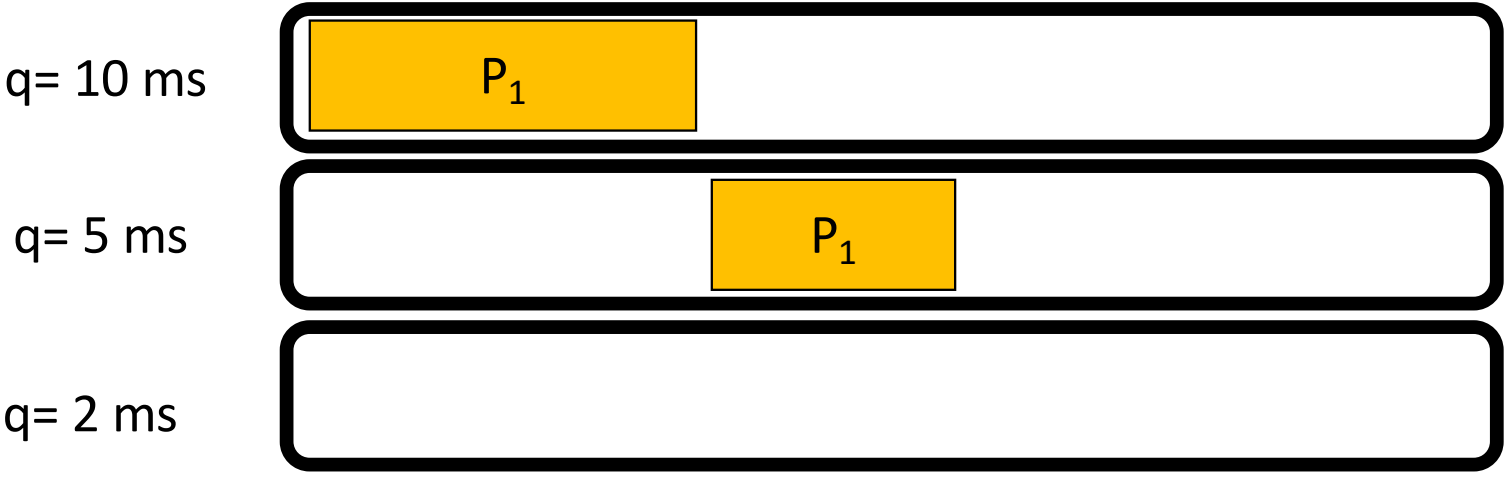
Long running CPU-bound



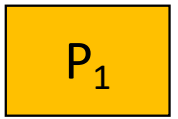
Learning behavior



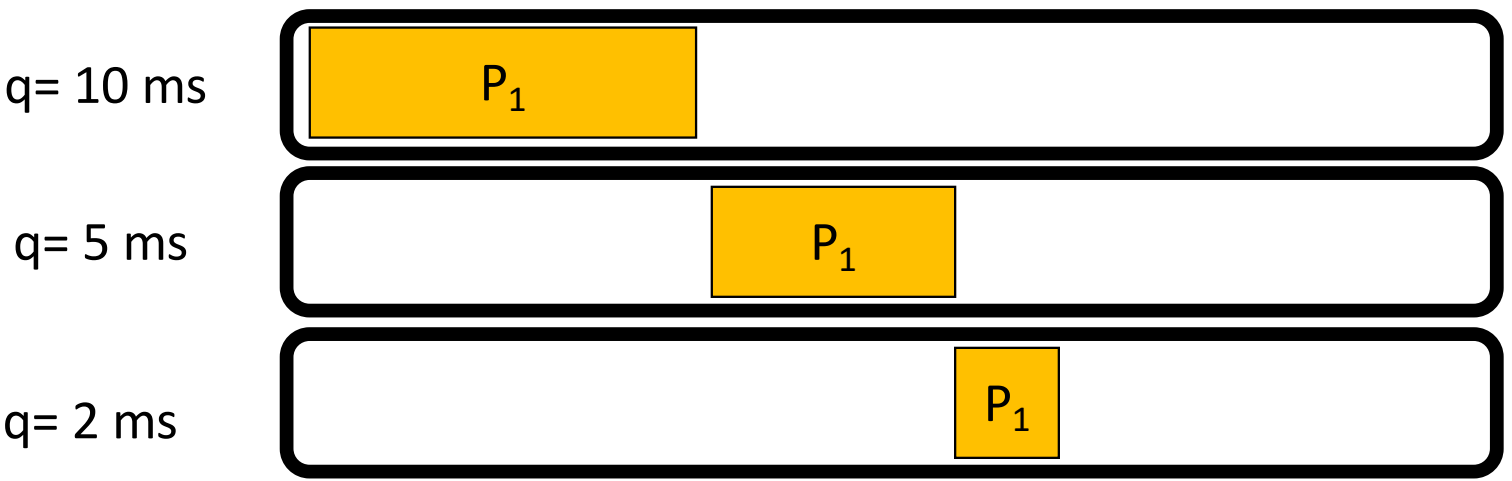
Long running CPU-bound



Learning behavior



Long running CPU-bound



Learning behavior

Rule 3

When a job enters the system, it is placed at the highest priority (the topmost queue).

Rule 4a

If a job uses up an entire time slice while running, its priority is *reduced* (i.e., it moves down one queue).

Rule 4b

If a job gives up the CPU before the time slice is up, it stays at the *same* priority level.

Crooks CS162 © UCB Fall 2022

Learning behavior

Where do IO-bound/interactive jobs end up?

a) High Priority Queue b) Low Priority Queue

Ideally in high priority, but if it finishes up the time slice then moved to a lower priority queue

Learning behavior

P_1

Long running CPU-bound

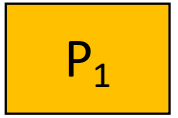
q= 10 ms

q= 5 ms

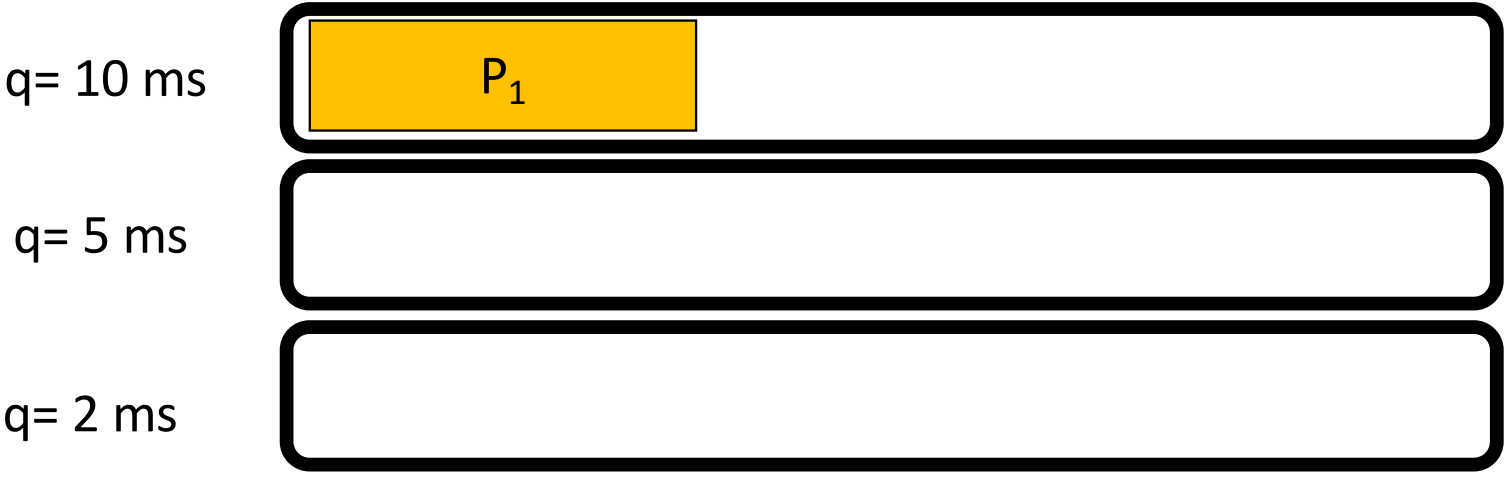
q= 2 ms

Schedule

Learning behavior

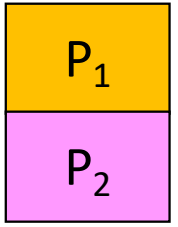


Long running CPU-bound



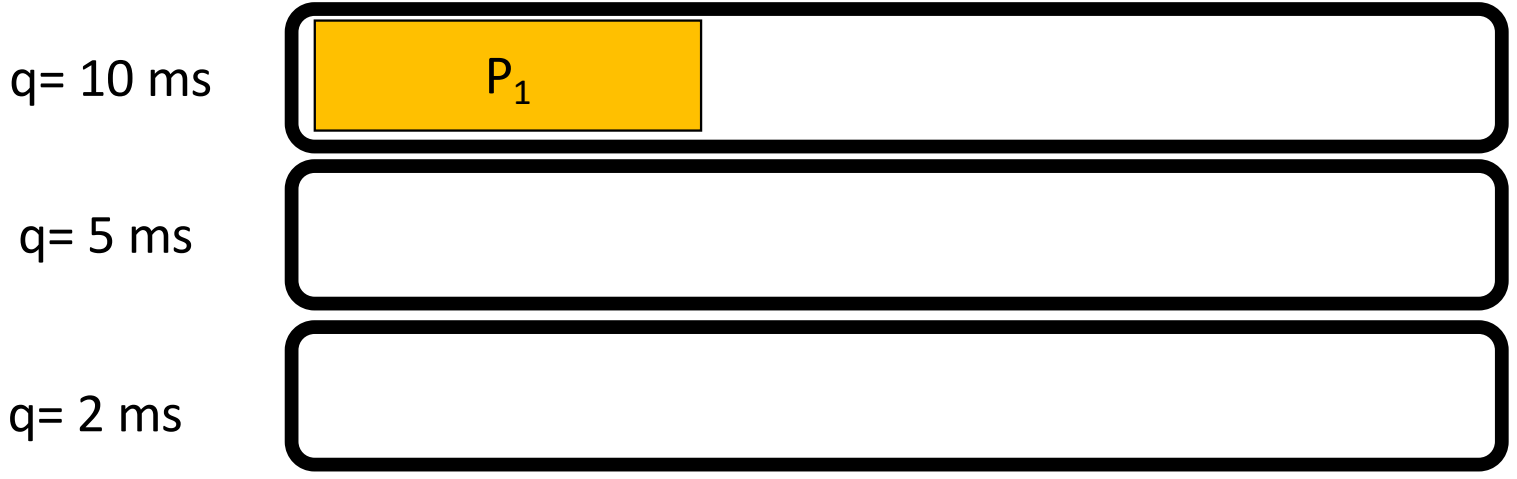
Schedule

Learning behavior



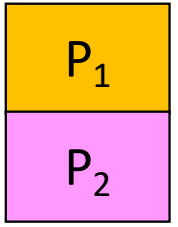
Long running CPU-bound

Computes for 1 ms and then IO for 1 ms



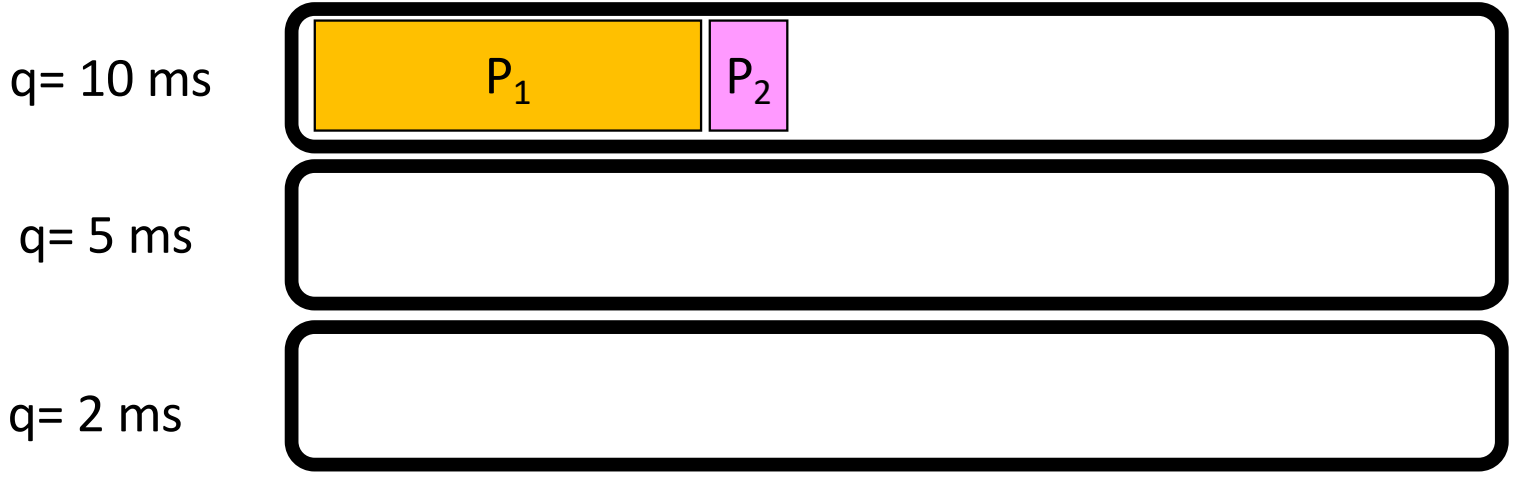
Schedule

Learning behavior



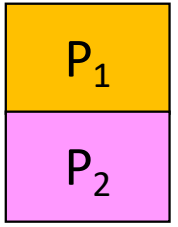
Long running CPU-bound

Computes for 1 ms and then IO for 1 ms



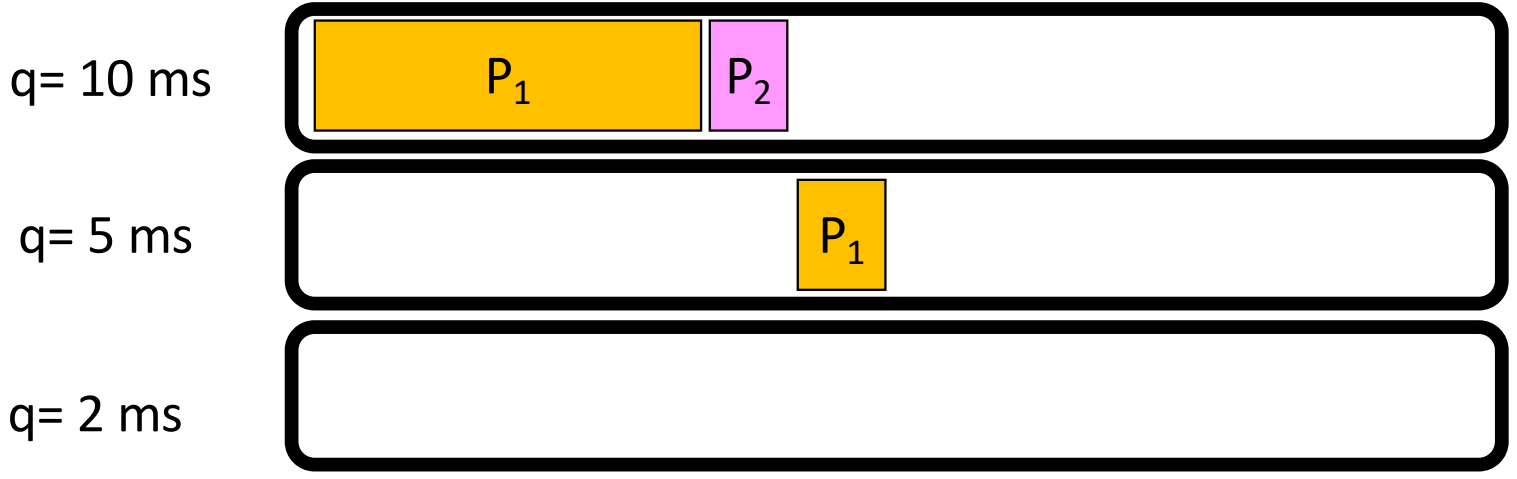
Schedule

Learning behavior



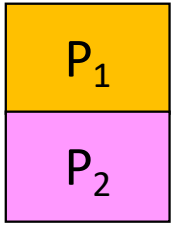
Long running CPU-bound

Computes for 1 ms and then IO for 1 ms



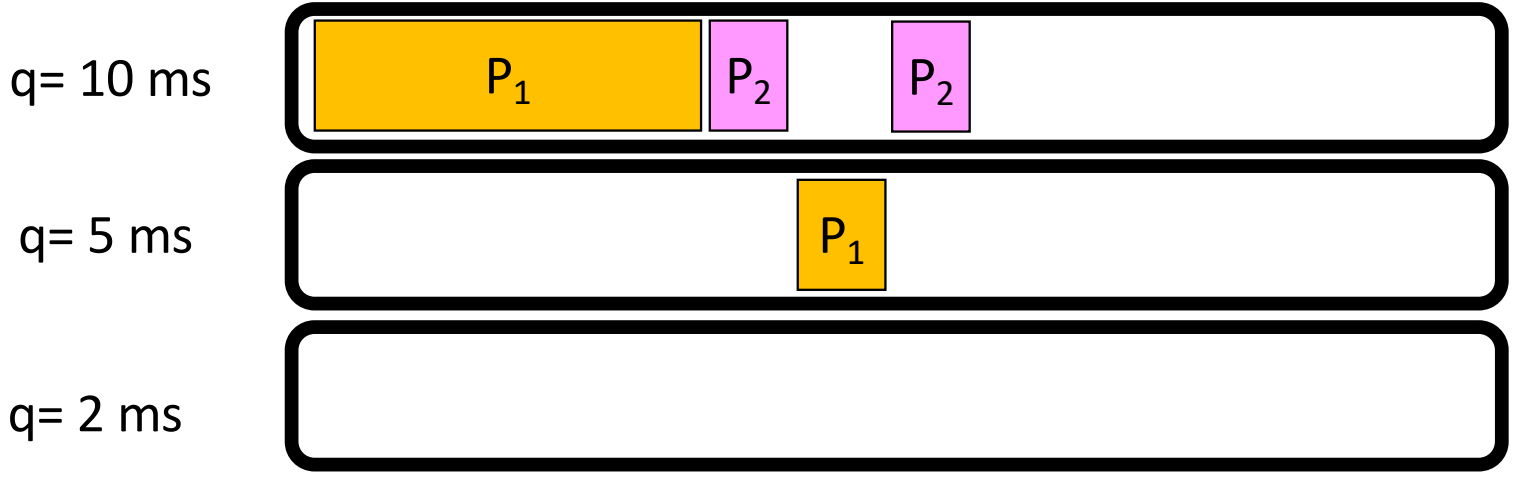
Schedule

Learning behavior



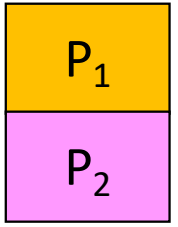
Long running CPU-bound

Computes for 1 ms and then IO for 1 ms



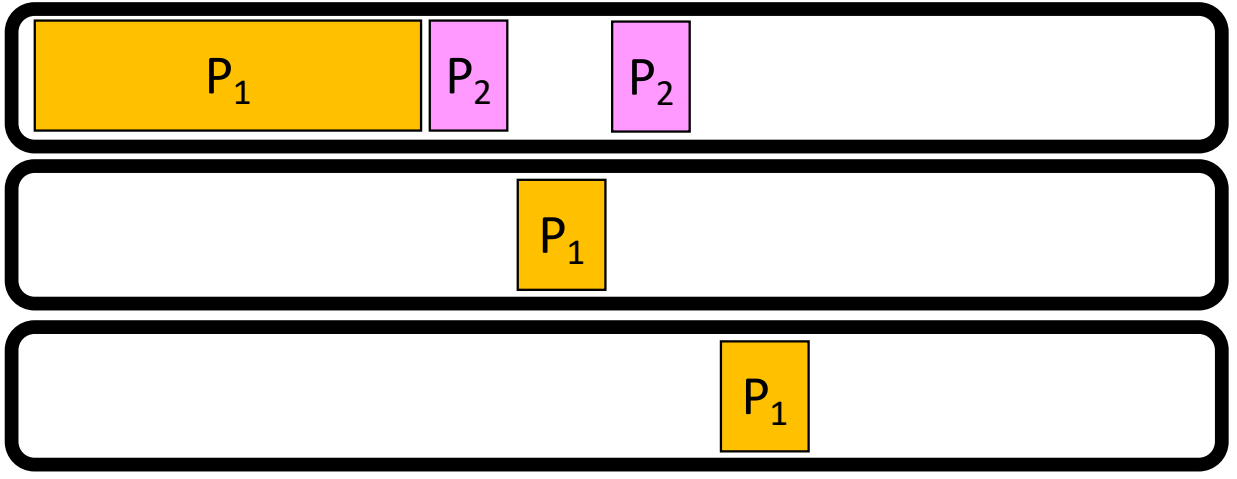
Schedule

Learning behavior



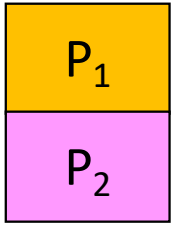
Long running CPU-bound

Computes for 1 ms and then IO for 1 ms



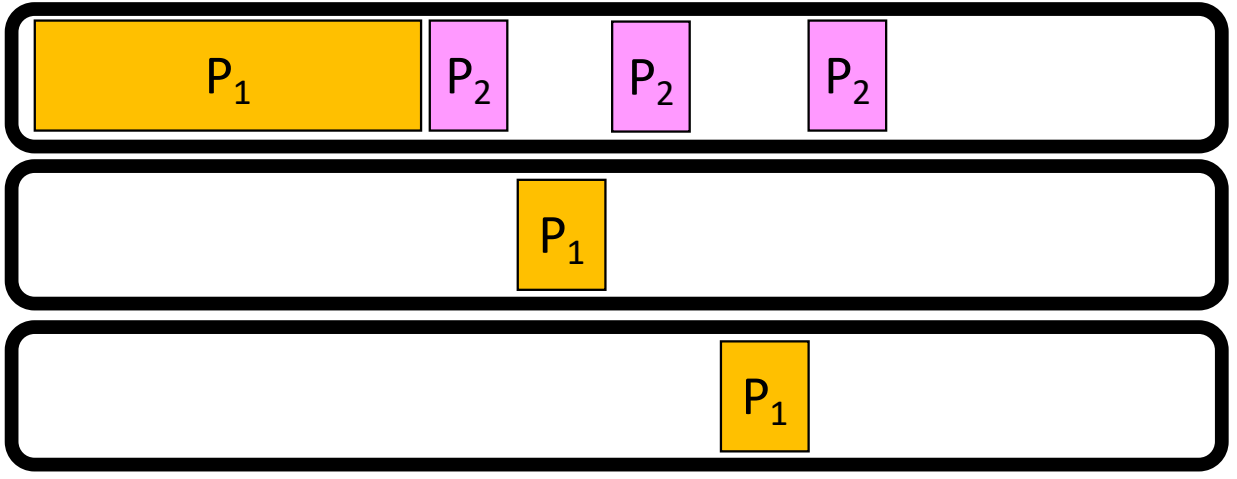
Schedule

Learning behavior



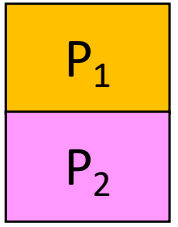
Long running CPU-bound

Computes for 1 ms and then IO for 1 ms



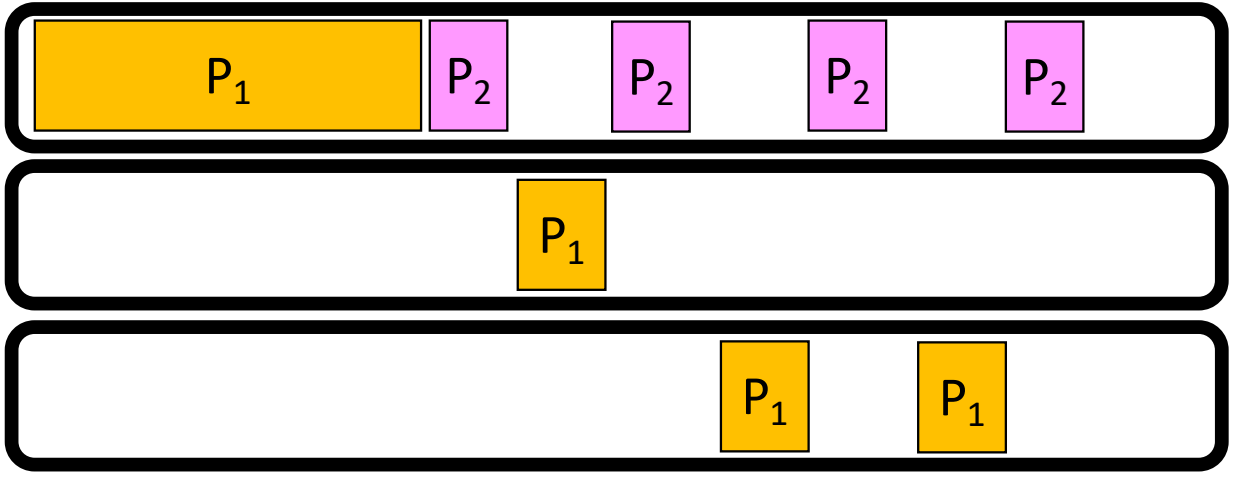
Schedule

Learning behavior



Long running CPU-bound

Computes for 1 ms and then IO for 1 ms



Schedule

MLFQ (Naïve Version): Pseudocode

```
foreach(queue : PrioritizedReadyQueues) { // RR
    foreach(job : queue) { // RR
        int timeslice = queue.getSlice();
        // Execute job for timeslice associated with this queue
        // If job used full timeslice then push job in
        // ready queue with one lower priority level
    }
}
```


Crooks CS162 © UCB Fall 2022

Are we done?

Can MLQF can be gamed?

Intentionally insert IO request just before time quanta to stay on queue.

MLQF is subject to starvation?

Yes!

Crooks CS162 © UCB Fall 2022

Are we done?

MLQF can be gamed:

Intentionally insert IO request just before time quanta to stay on queue.

Rule 4

Once a job uses up its time allotment at a given levels (regardless of how many times gave up CPU), reduce priority

MLQF is subject to starvation:

Systematically prioritize higher-priority queues

Rule 5

After some time period S, move all jobs in system to the topmost queue.

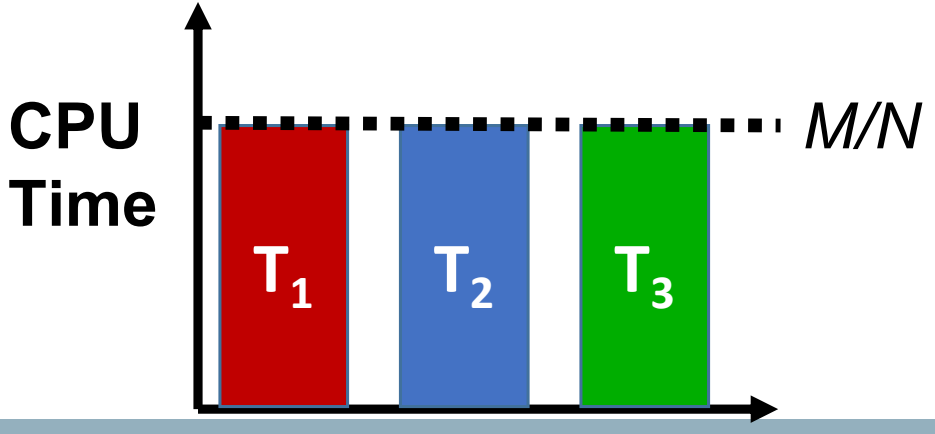
Today's Class

- Multi-level feedback scheduling
- Linux Completely Fair Scheduler (CFS)

Acknowledgements: Today's lecture content derived from CS162 course at University of California, Berkeley

Linux Completely Fair Scheduler

- Goal: Each process gets an equal share of CPU
- Attempts to divide the CPU time fairly (equally) among all the processes
  - If the latency to resume a scheduler is M units of time, N processes will get M/N time slice of CPU



Linux CFS

- **Constraint 1**: *Scheduling Latency*

- Target Latency: 20ms, 4 Processes
  - Each process gets 5ms time slice
- Target Latency: 20 ms, 200 Processes
  - Each process gets 0.1ms time slice
  - Recall Round-Robin: Huge context switching overhead

Crooks CS162 © UCB Fall 2022

Linux CFS

- **Constraint 2**: *Minimum Granularity for scheduling latency*

- Target Latency 20ms, Minimum Granularity 1ms, 100 processes
  - Each process gets 1ms time slice

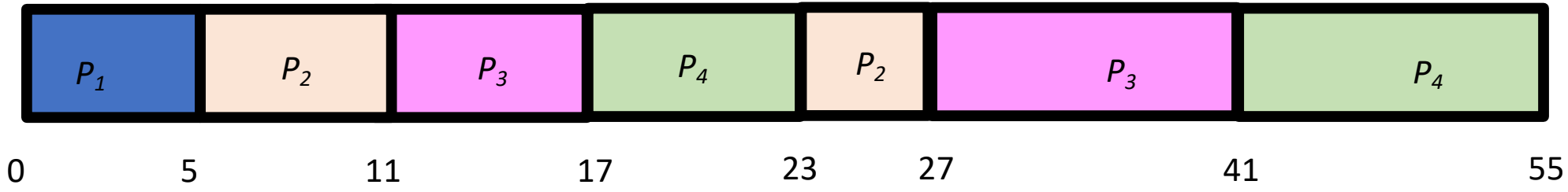
Linux CFS

- Which process to pick from the ready queue?
- **Constraint 3**: *Pick the process from the ready queue with minimum vruntime*
  - Uses virtual runtime (vruntime) variable in PCB to keep track of time a process has executed on the CPU, and it is updated at every **context switch**
    - vruntime += t * weight based on process priority

Linux CFS

Process	Burst Time
P_1	5
P_2	10
P_3	20
P_4	20

- Scheduling latency is **20ms**, minimum granularity is 1ms, and weight=1
- Time slice in beginning = 20/4 = 5ms
- Time slice after P1 terminates = 20/3 ~ 6ms
- Time slice after P2 terminates = 20/2 = 10ms



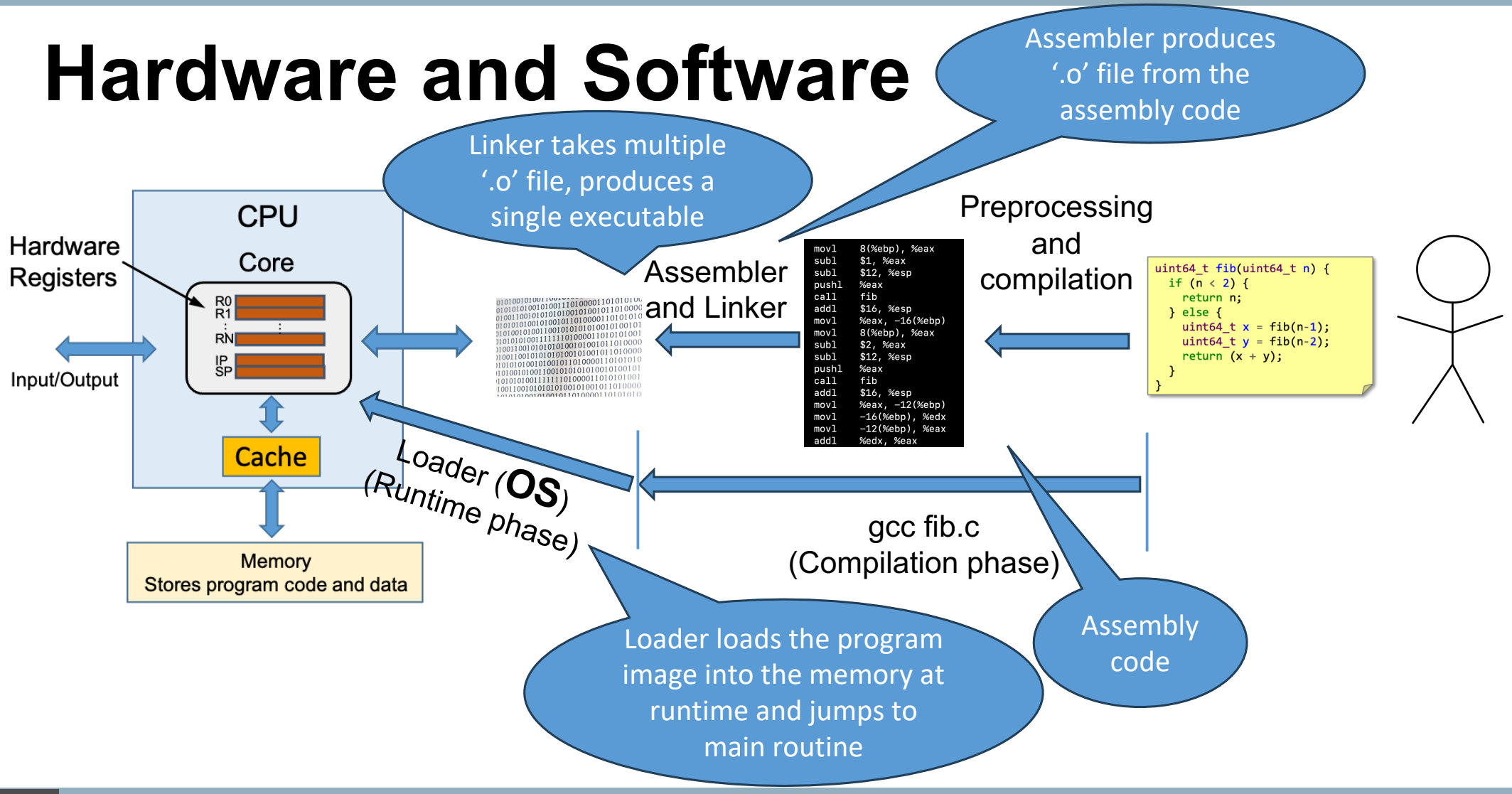
Next Lecture (Tomorrow)

- Mid semester review

Lecture 13: Mid-Semester Review

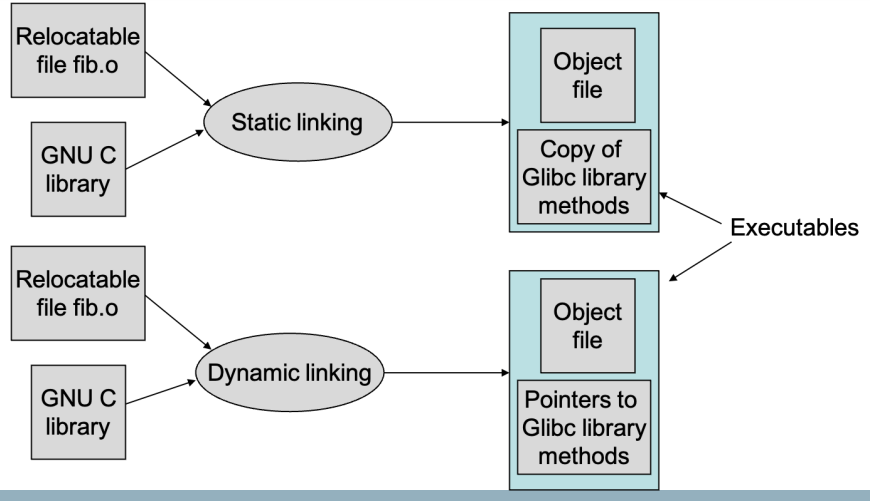
Vivek Kumar
Computer Science and Engineering
IIIT Delhi
vivekk@iiitd.ac.in

Hardware and Software



Static and Dynamic Linking

```
$ gcc -static -o fib fib.c
$ file fib
...ELF 64-bit executable...statically linked
$ ls -l fib
-rwxrwxr-x 1 vivek vivek 845304 Aug  6 10:26 fib
$ gcc -o fib fib.c
$ ls -l fib
-rwxrwxr-x 1 vivek vivek 8328 Aug  6 10:27 fib
```



- Static linking
  - Each and every library modules referenced in the relocatable file is copied into the final executable
    - Static binding at compile time
- Dynamic linking
  - Final executable only contains references (pointers) to the library method instead of the copy of a library method
    - Binding with library done at runtime during execution

Executable and Linkable Format (ELF)

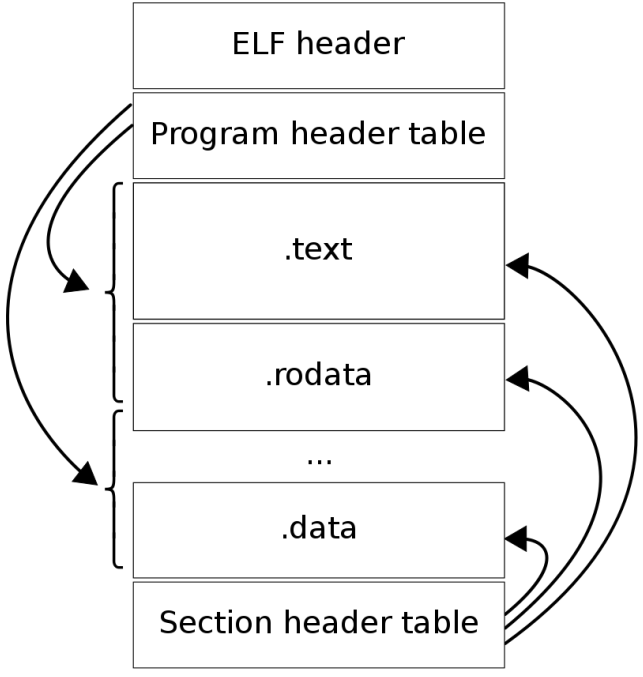


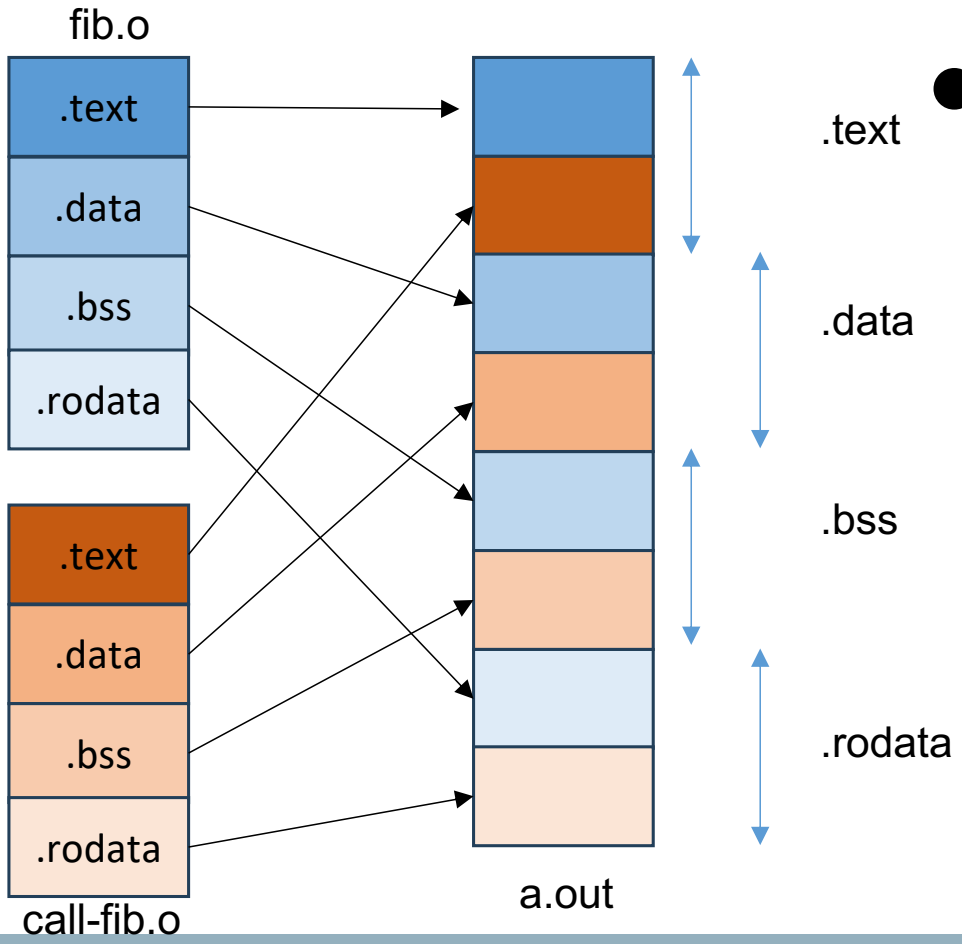
Image source: https://en.wikipedia.org/wiki/Executable_and_Linkable_Format

- ELF header
  - Provides a roadmap for the entire file organization
    - Always at offset zero of the object file
    - Provides entry point address for execution
- Two views of an ELF file
  - Linkable view (relocatable file)
  - Execution view (executable file, shared object)
- Linkable view
  - Section header table
    - One section header for each section
  - Sections
    - Contains data required for linking
    - Machine code, global variables, (initialized and un-initialized), symbol tables, line mapping between machine code and original C code, etc.
- Execution (or Loader) view
  - Program header table
    - One program header for each segment
  - Segments
    - Created by merging several sections
    - Contains information required for by the loader for execution
- Contiguous chunk of memory (ELF header, PHT, SHT, each section, each segment)

Linker: Merges the Sections

```
[vivek@possum]$ cat fib.c
int get_number();

int fib(int n) {
  if(n<2) return n;
  else return fib(n-1)+fib(n-2);
}

int compute() {
  int n = get_number();
  return fib(n);
}
```

```
[vivek@possum]$ cat call-fib.c
int compute();
int get_number() {
  return 40;
}
int main() {
        int result = compute();
        return 0;
}
```



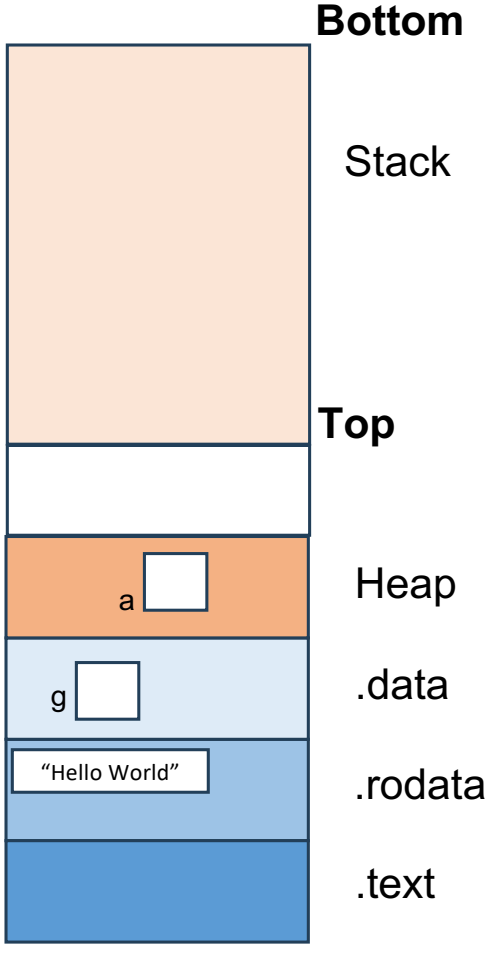
- Relocation is the process of merging the sections and resolving the symbol addresses

Loader in OS: Complete Steps

- When an executable file is run, the operating system loads it into memory and start it running as follows
  1. OS kernel reads the PHT and loads the parts specified in the segments of type “INTERP” and “LOAD”
  2. Control is transferred to the interpreter (/lib/ld-library.so)
  3. Dynamic Linker (DL) is invoked
  4. DL load the shared libraries needed by the executable and carry out the relocations
     - This information is available in “.dynamic” section
     - Connects symbolic references with their corresponding symbolic definitions (e.g., connecting the call to printf with the actual implementation of the printf in GNU C library)
  5. DL transfer the control to the address given by the symbol “_start” (and eventually to user main)

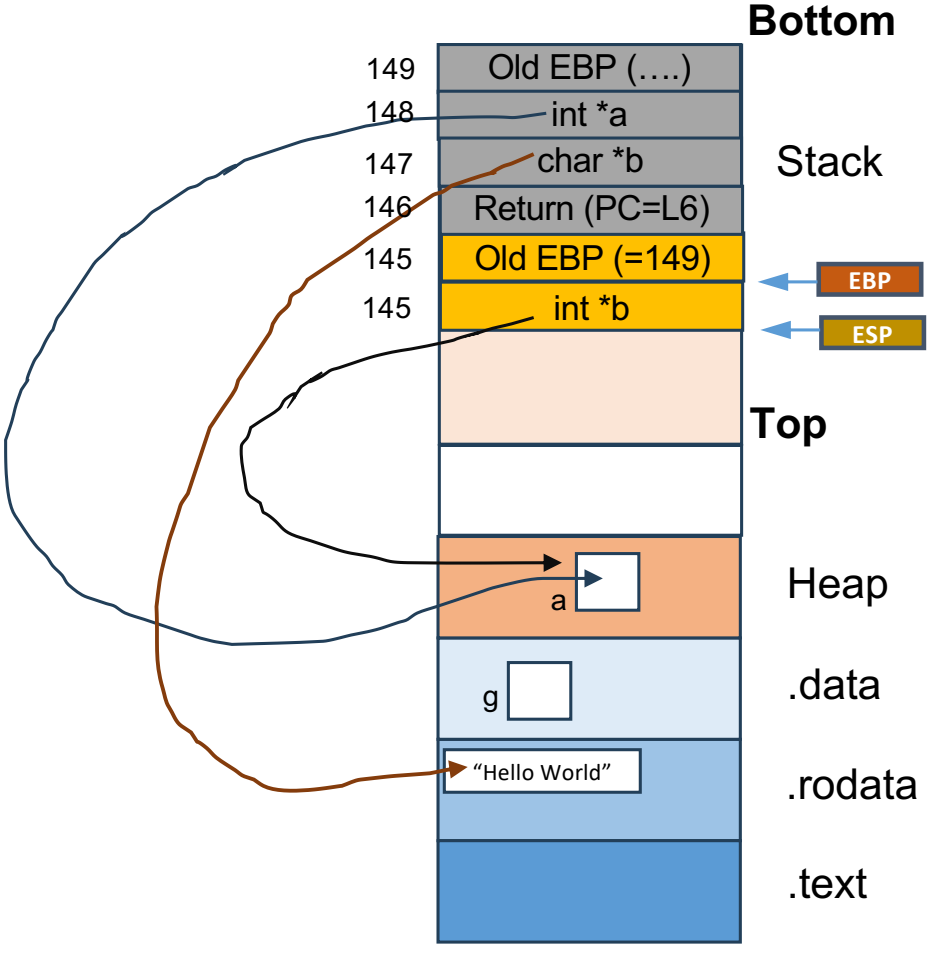
Program Execution: Class Exercise

```
L1: int g=0;

L2: void main() {
L3:     int *a = (int*) malloc(4);
L4:     char *b = “Hello World”;
L5:     foo(a);
L6:     g=*a;
L7: }

L8: void foo(int* b) {
L9:     *b = 20;
L10:}
```



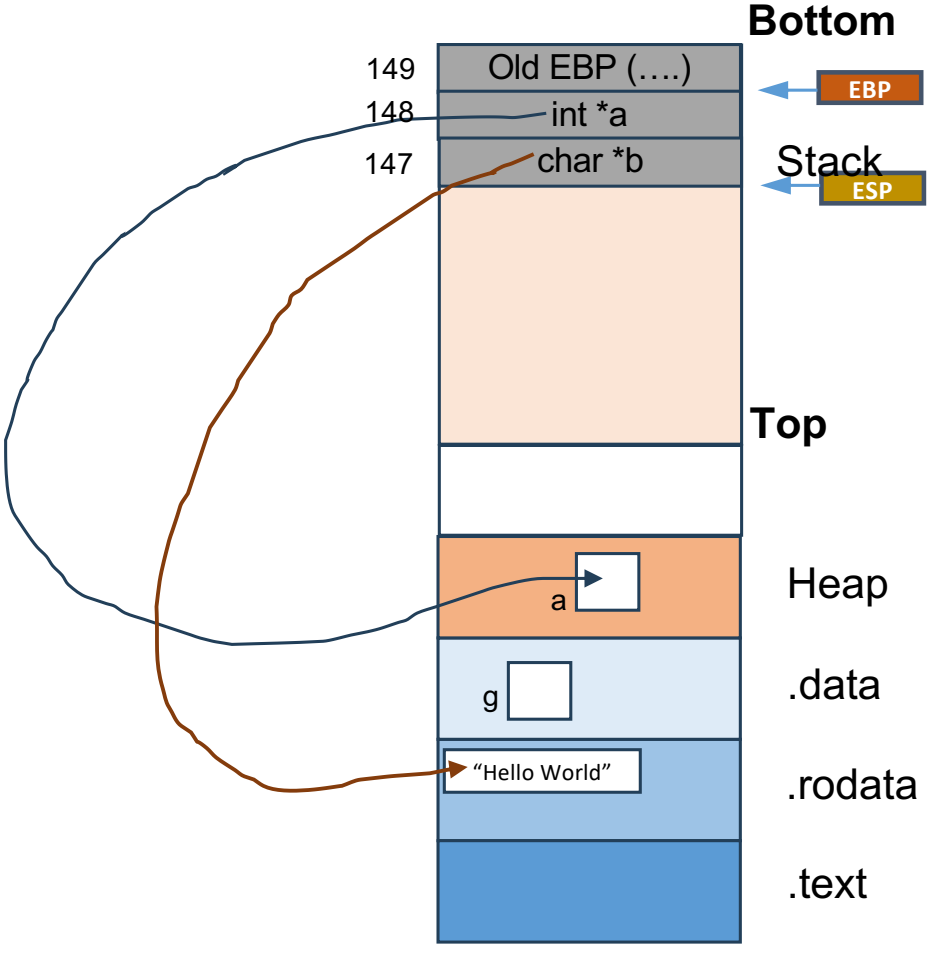
Program Execution: Class Exercise

```
L1: int g=0;

L2: void main() {
L3:     int *a = (int*) malloc(4);
L4:     char *b = "Hello World";
L5:     foo(a);
L6:     g=*a;
L7: }

L8: void foo(int* b) {
L9:     *b = 20;
L10:}
```

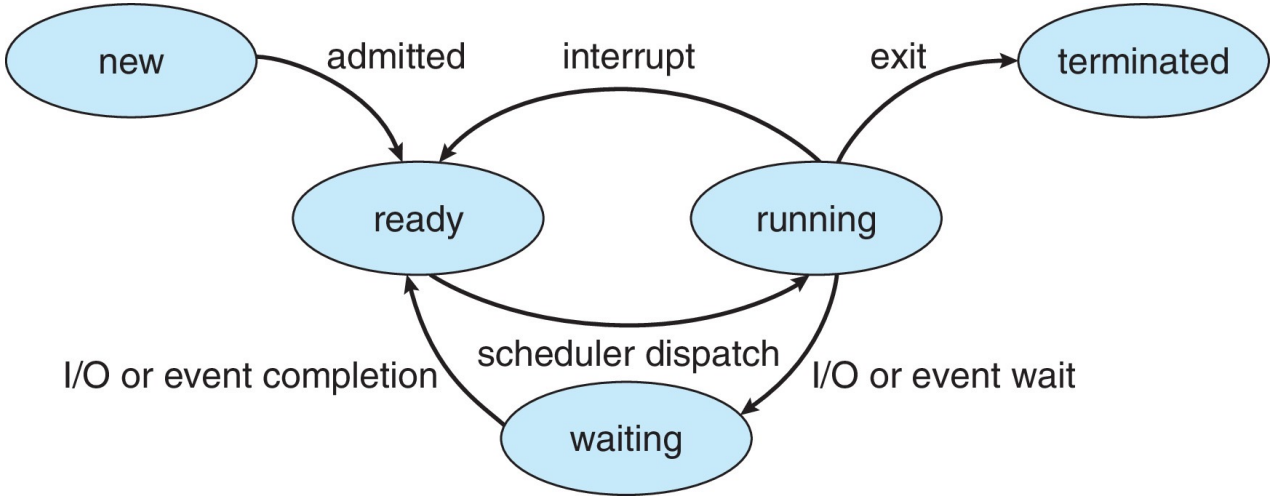


Program Execution: Class Exercise

```
L1: int g=0;

L2: void main() {
L3:     int *a = (int*) malloc(4);
L4:     char *b = "Hello World";
L5:     foo(a);
L6:     g=*a;
L7: }

L8: void foo(int* b) {
L9:     *b = 20;
L10:}
```



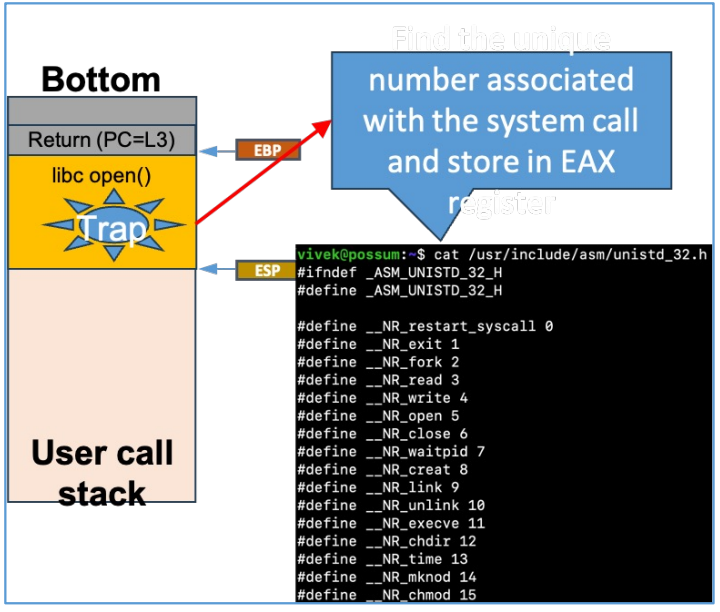
Process States

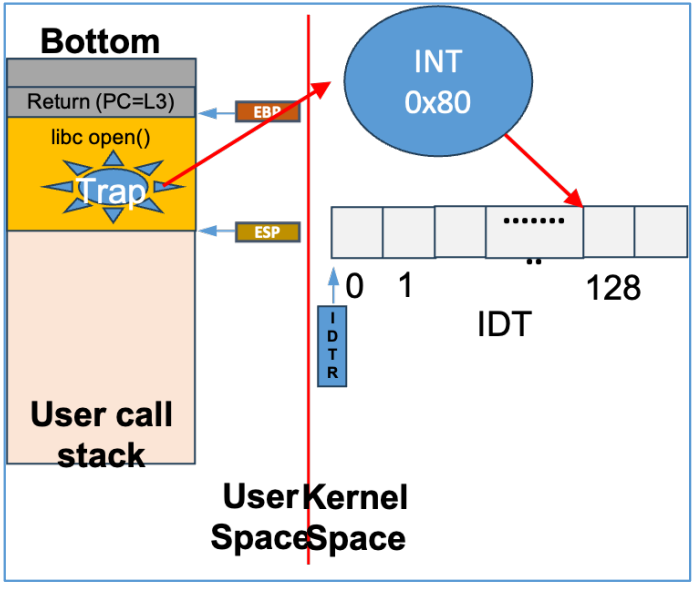


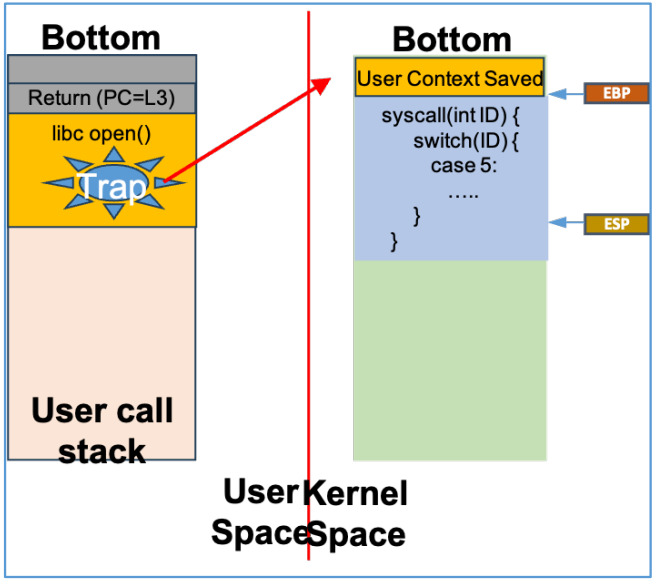
- As a process executes, it changes state
  - **New**: The process is being created
  - **Running**: Instructions are being executed
  - **Waiting**: The process is waiting for some event to occur
  - **Ready**: The process is waiting to be assigned to a processor
  - **Terminated**: The process has finished execution

Source: https://os-book.com/OS10/slide-dir/index.html

Steps for Making a System Call





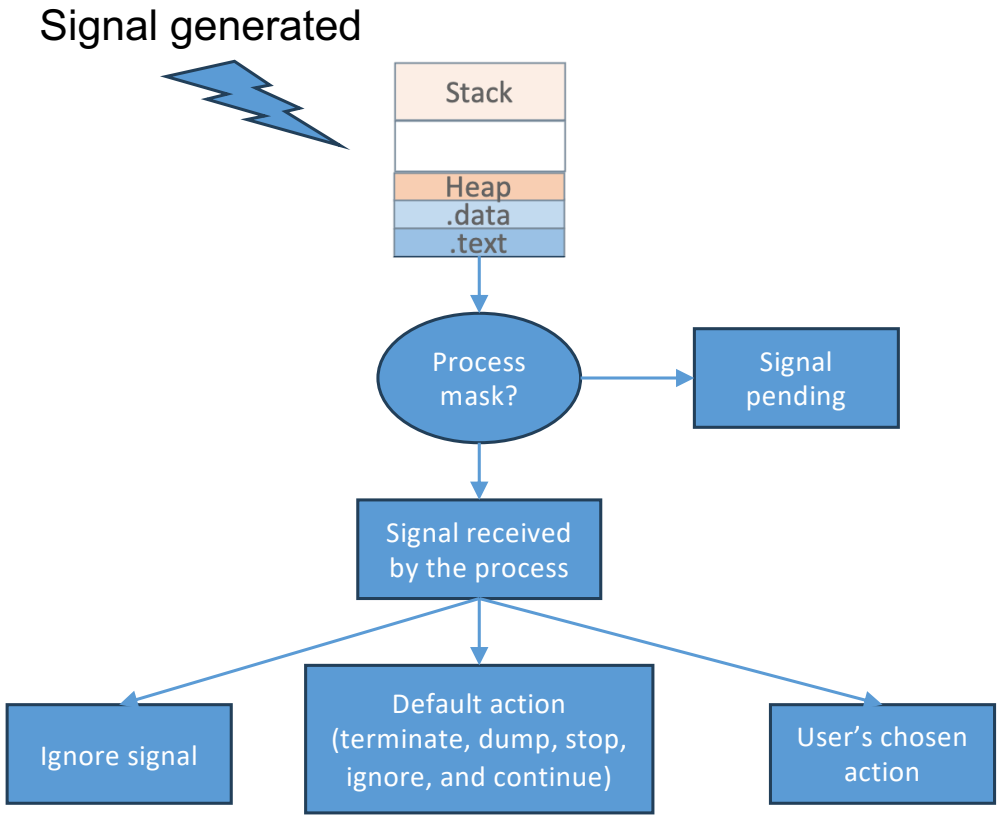


A Process's Life Lessons (contd.)

```
int main() {
    if(fork() == 0) {
        printf("I am the child process\n");
        char* args[2] = {"./fib", "40"};
        execv(args[0], args);
        printf("I should never print\n");
    }
    wait(NULL);
    return 0;
}
```

1. Processes can have children
   - fork()
2. Children should be obedient to their parent
   - exit()
3. Parent must follow the steps for good parenting
   - wait() or waitpid()
4. Children should not run their family business
   - exec()

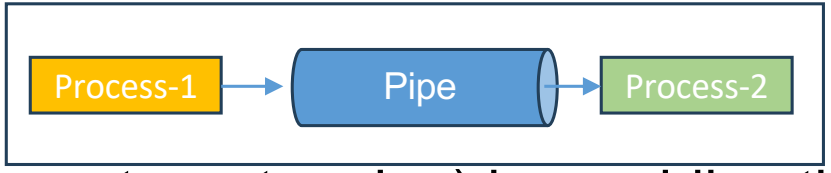
Signal Delivery and Handling



- Process can use sigprocmask function to make changes to its signal mask (member in task_struct) if it wants to temporarily block a set of signals (not all signals can be blocked, e.g., SIGKILL and SIGSTOP)
- User can create his own signal handler function
- Signal name has prefix "SIG". Some of the signals that you can easily come across are:
  - SIGINT (ctrl-c), SIGQUIT (ctrl-\), SIGSTOP (ctrl-z), SIGTRAP (breakpoint), SIGSEV (segmentation fault), SIGTERM (process termination), SIGCHLD (child stopped/terminated), SIGFPE (floating point exception), etc.

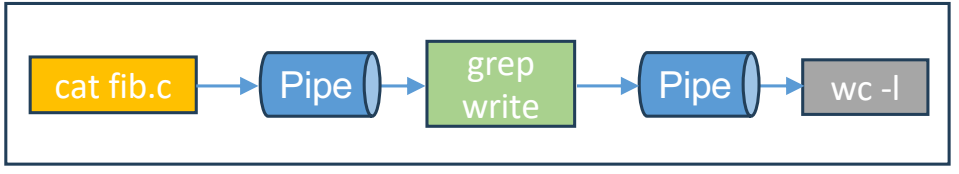
IPC Using Pipes

Processes **must** be running on the **SAME** machine

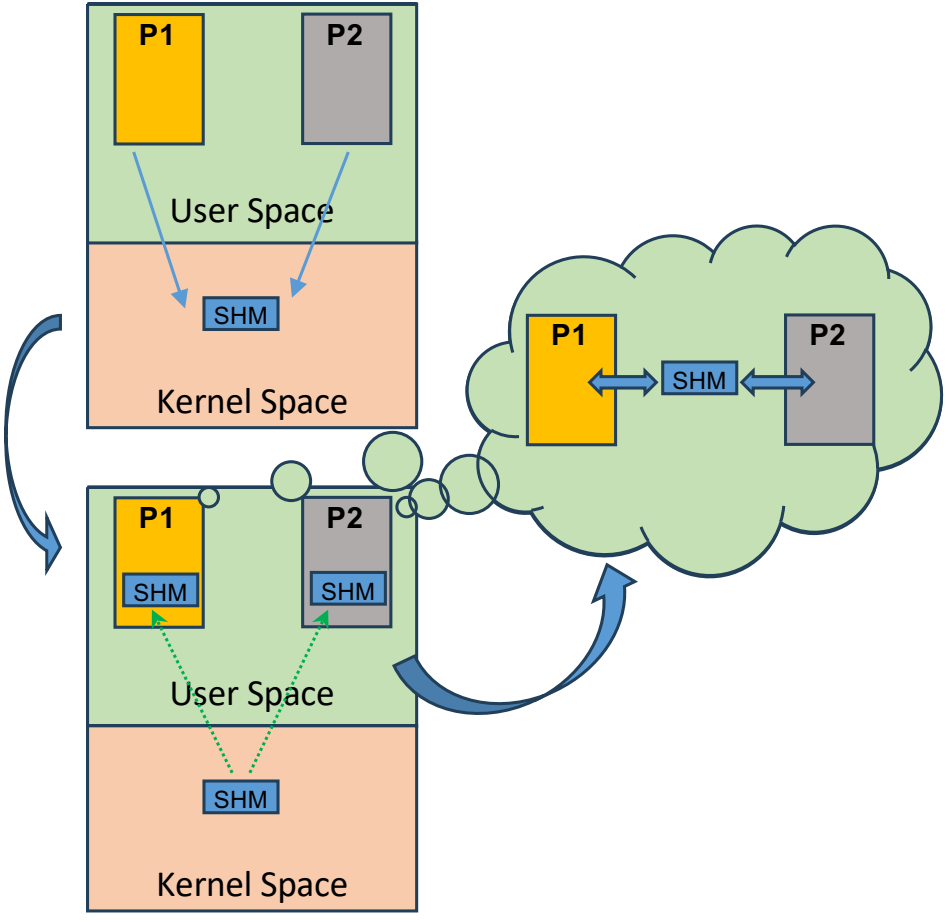


- Pipes (analogous to water pipe) is a unidirectional stream of data flowing from a source process to a destination process
  - Kernel buffer exposed to processes as a pair of file descriptors (readable end and writable end). Default buffer size on latest kernel is 16 pages (16x4096 bytes)
  - Data delivered in the same order as sent
  - Uses blocking IO and the writer process will block if the pipe is full
- We use it frequently on Shell
  - Better than using temporary files as pipes automatically clean up themselves unlike files that must be explicitly removed using the “rm” command on Shell

```
vivek@possum:~/os23$ cat fib.c | wc -c
1788
vivek@possum:~/os23$ cat fib.c | wc -l
77
vivek@possum:~/os23$ cat fib.c | grep write | wc -l
10
```



IPC Using POSIX Shared Memory



- Shared memory is the fastest form of IPC in which processes can asynchronously write to a shared region of memory **without** the need for switching to kernel space
  - User specified memory size
- Access to the shared memory is achieved as follows
  1. Create and open a new shared memory object using shm_open system call in the parent process
  2. Set the desired size for the newly created shared memory region using ftruncate
  3. Map the shared memory region into the process's address space using mmap (parent process calls mmap that gets shared in child's address space after fork)
  4. Both child and the parent processes un-maps the shared memory from their address space upon completion by using munmap system call, followed by closing the file descriptor for the shared memory object (close system call)
  5. Parent deletes the shared memory object by using shm_unlink
- Can be used for parallel programming!

IPC in Shared Memory

1) shm_open (...)
2) ftruncate (...)
3) shm = mmap (...)

```
int main() {
    shm_t* shm = setup();
    sem_init(&shm->mutex, 1, 1);
    int chunks = SIZE/NPROCS;
    for(int i=0; i<NPROCS; i++) {
        if(fork()==0) {
            int local=0;
            int start = i*chunks;
            int end = start+chunk;
            for(int i=start; i<end; i++) local += shm->array[i];
            sem_wait(&shm->mutex);
            shm->sum += local;
            sem_post(&shm->mutex);
            cleanup_and_exit();
        }
    }
    for(int i=0; i<NPROCS; i++) wait(NULL);
    cleanup();
    return 0;
}
```

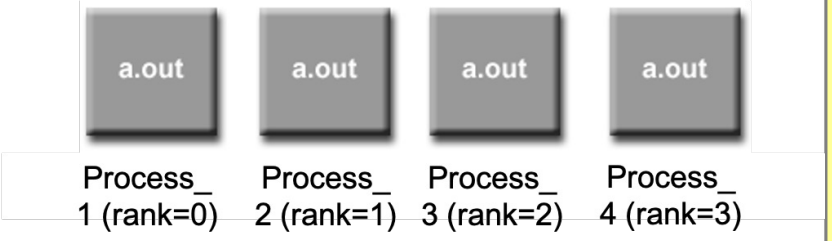
```
typedef struct shm_t {
    int A[SIZE];
    int sum;
    sem_t mutex;
} shm_t;
```

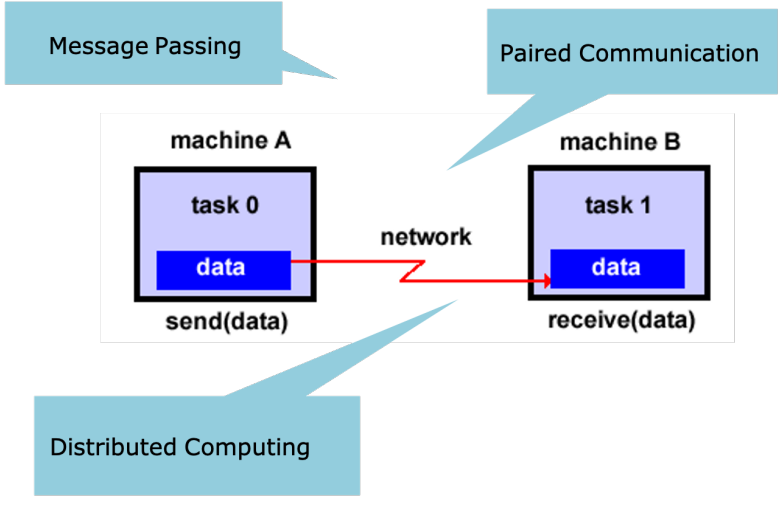
1) munmap (shm, ...)
2) close (...)
3) exit(0)

1) munmap (shm, ...)
2) close (...)
3) shm_unlink (...)
4) sem_destroy

Value of Semaphore	Process-T1	Process-T2
1		
1	call sem_wait ()	
0	sem_wait () returns	
0	(crit sect)	
0	call sem_post ()	
1	sem_post () returns	

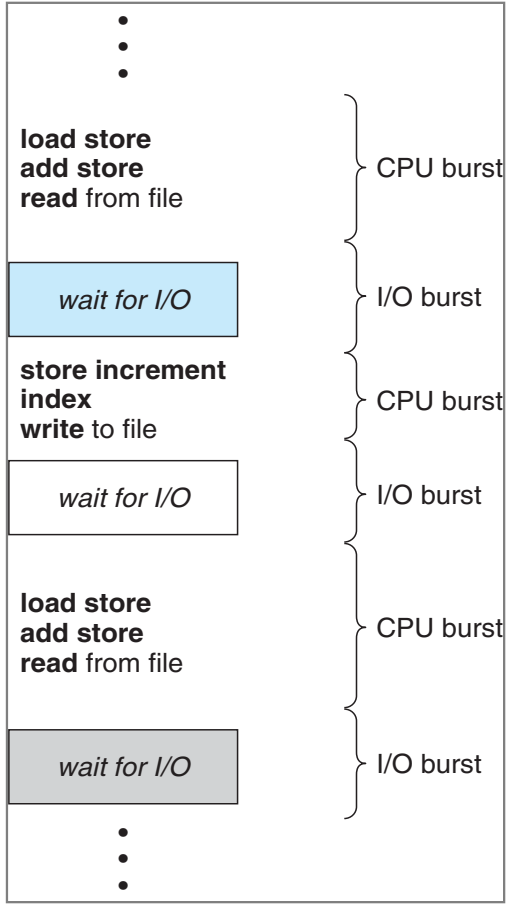
IPC in Distributed Memory






```
int main(int argc, char **argv) {
    int rank=0, nproc=4;
    MPI_Init(&argc, &argv);
    // 1. Get to know your world
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &nproc);
    int array[SIZE]; // initialized and assume (SIZE % nproc = 0)
    // 2. calculate local sum
    int my_sum = 0, chunk = SIZE/nproc;
    for (int i=rank*chunk; i<(chunk+1)*rank; i++) my_sum += array[i];
    // 3. All non-root processes send result to root processes (rank=0)
    if(rank > 0) {
        MPI_Send(&my_sum, 1, MPI_INT, 0, 0, MPI_COMM_WORLD);
    }
    else { // executed only at rank=0
        int total_sum = my_sum, tmp;
        for(int src=1; src<nproc; src++) {
          MPI_Recv(&tmp, 1, MPI_INT, src, 0, MPI_COMM_WORLD, NULL);
          total_sum += tmp;
        }
    }
    MPI_Finalize();
}
```

Process Scheduling



Property	FCFS	SJF	STCF	RR
Optimize Average **Completion** Time	✗	✓	✓	✗
Optimize Average **Waiting** Time	✗	✗	✗	✓
Prevent Starvation	✓	✗	✗	✓
Prevent Convoy Effect	✗	✗	✓	✓
Psychic Skills **NOT** Needed	✓	✗	✗	✓

Where are we as of now

- **CSE231 Post Conditions**
  1. Students are able to create a Unix shell with complete clarity about process creation and process execution
  2. Students are able to write multi- threaded applications with synchronization primitives and ability to analyze effects of concurrency on process execution and correctness
  3. Students are able to analyze the impact of OS concepts, e.g. virtual memory, concurrency, on program execution and ability to fine-tune the program to run efficiently on a given OS
  4. Students are able to demonstrate deeper understanding of the Unix-like OSes and kernel programming

All the best for your exam !!

- Exam syllabus
  - Lectures **02-11**
- Exam schedule
  - Date: Thursday (28th Sept.)
  - Time: 3:00pm-4:00pm
  - Venue: C102

Lecture 14: Dynamic Memory Allocation

Vivek Kumar
Computer Science and Engineering
IIIT Delhi
vivekk@iiitd.ac.in

Today's Class

- Dynamic memory allocations
- Fragmentation
- Allocation using implicit list
- Allocation using buddy allocator

Memory Allocations

- Static size, static allocation
  - Global and static variables uses stack.
  - Linker allocates final addresses
  - Executable stores these allocated addresses.

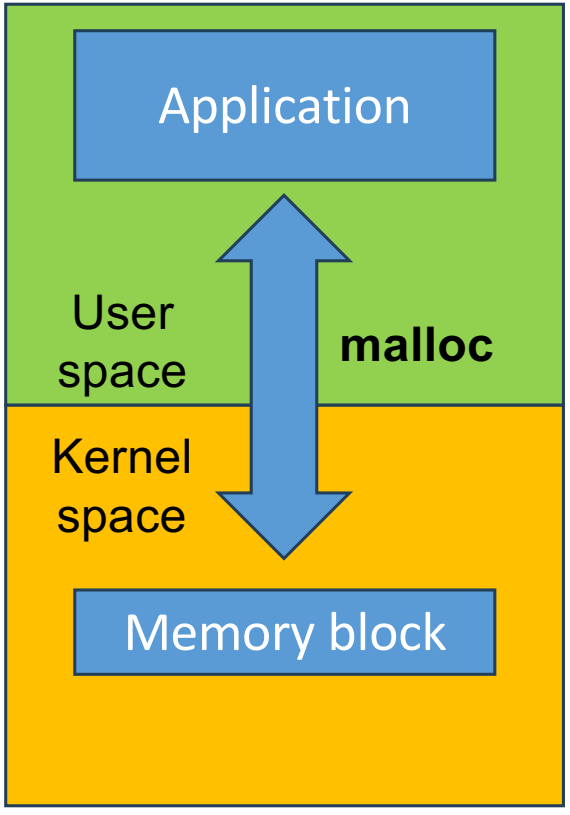
- Static size, dynamic allocation
  - Local variables
  - Compiler directs heap allocation

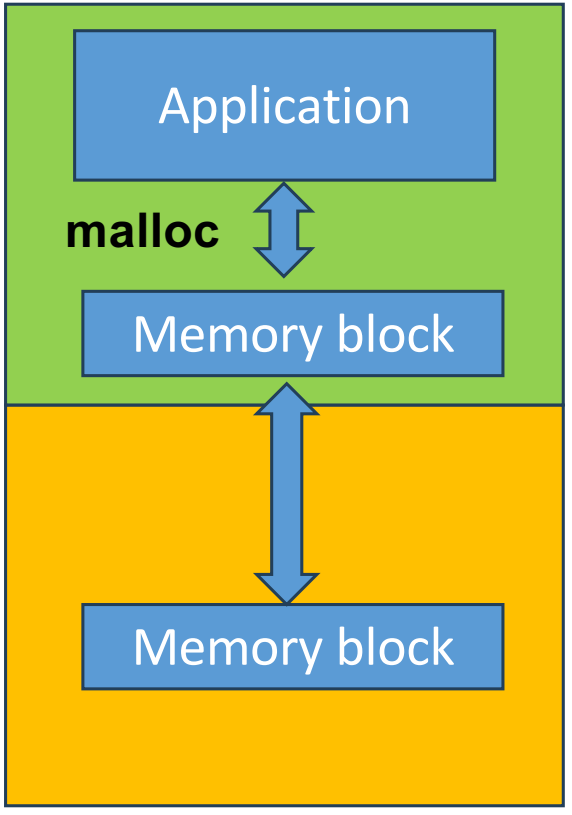
- Dynamic size, dynamic allocation
  - Programmer controlled
  - **Allocated in the heap – how?**

Dynamic Memory Allocation

- Explicit vs. implicit memory allocator
  - Explicit: application allocates and *frees* space
    - e.g., malloc and free in C
  - Implicit: application allocates, but does not free space
    - e.g., garbage collection in Java or Python
- Allocation
  - In both cases the memory allocator provides an abstraction of memory as a set of blocks
  - Gives out free memory blocks to application

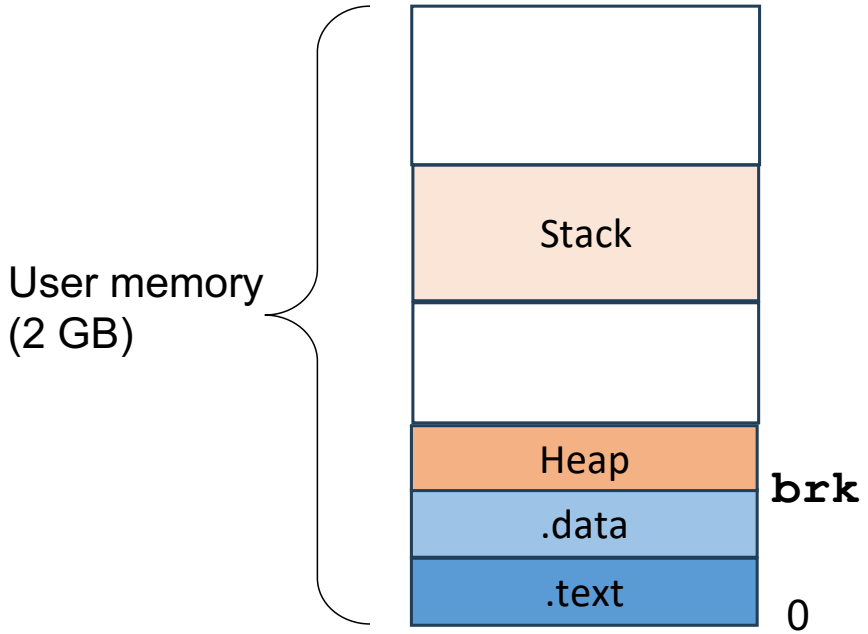
Which is Better and Why?





- Case-1
  - Switch from user space to kernel space at every call to malloc
  - Huge overheads in case of several calls to malloc
- Case-2
  - A big chunk of memory is allocated by the OS and mapped into process's address space
  - Calls to malloc then returns memory allocated inside the process address space
  - Switch from user space to kernel space is significantly reduced

Process Memory Image



- `void *sbrk(intptr_t incr)`
  - Easiest to use function for allocators to request additional heap memory from the OS
  - `brk` initially set to the end of the data section
  - Calls to `sbrk` increment `brk` by `incr` bytes (new memory is zero filled)
  - The `incr` can be negative, in which case the amount of allocated space is decreased

```
       brk() sets the end of the data segment to the value specified by addr, when that value is reasonable, the sys-
       tem has enough memory, and the process does not exceed its maximum data size (see setrlimit(2)).

       sbrk() increments the program's data space by increment bytes.  Calling sbrk() with an increment of 0  can  be
       used to find the current location of the program break.

RETURN VALUE
       On success, brk() returns zero.  On error, -1 is returned, and errno is set to ENOMEM.

       On  success,  sbrk()  returns  the  previous program break.  (If the break was increased, then this value is a
       pointer to the start of the newly allocated memory).  On error, (void *) -1 is returned, and errno is  set  to
       ENOMEM.
```

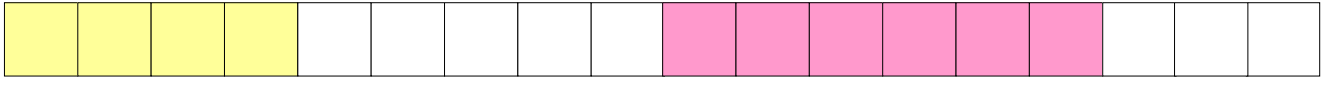
Dynamic Memory Allocation Examples

```
p1 = malloc(4*sizeof(char))
```

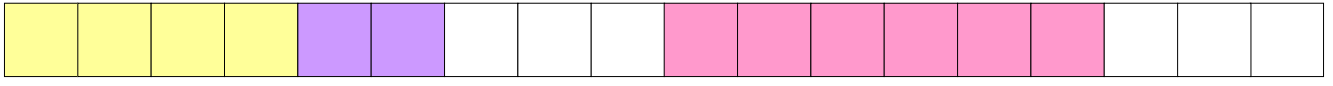
```
p2 = malloc(5*sizeof(char))
```

```
p3 = malloc(6*sizeof(char))
```

```
free(p2)
```



```
p4 = malloc(2*sizeof(char))
```



Goals of Good `malloc/free`

- Good performance for malloc and free
  - Ideally should take constant time (not always possible)
  - Should certainly not take linear time in the number of blocks
- Good space utilization
  - User allocated structures should use most of the heap
  - Want to minimize "**fragmentation**"

Today's Class

- Dynamic memory allocations
- Fragmentation
- Allocation using implicit list
- Allocation using buddy allocator

Fragmentation

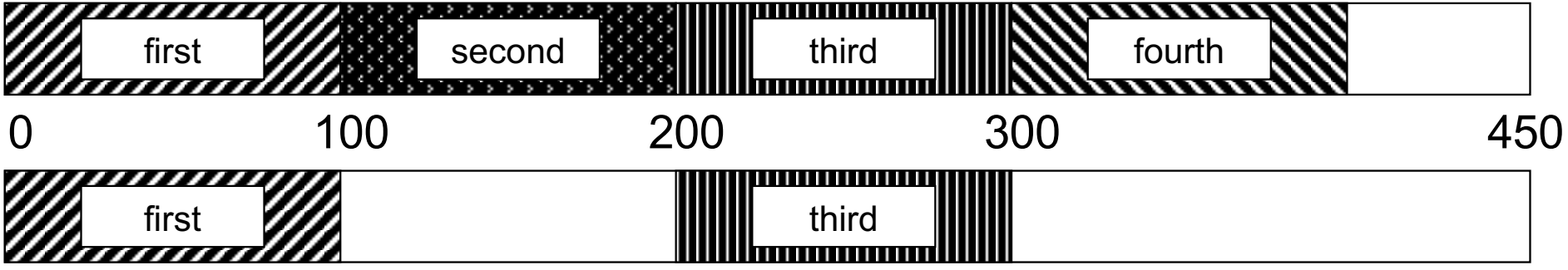
- Segments of memory can become unusable due to the result of allocation scheme
- Two types of fragmentation
  - External fragmentation
    - Memory remains unallocated
    - Variable allocation sizes
  - Internal fragmentation
    - Memory is allocated but unused
    - Fixed allocation sizes

External Fragmentation

- Imagine calling a series of *malloc* and *free*

```
char* first = malloc(100);
char* second = malloc(100);
char* third = malloc(100);
char* fourth = malloc(100);
free(second);
free(fourth);
char* problem = malloc(200);
```

- 250 free bytes of memory, only 150 contiguous
  - unable to satisfy final malloc request

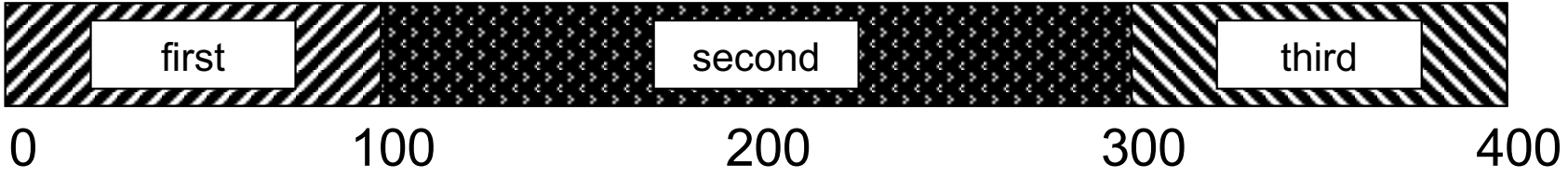


Internal Fragmentation

- Imagine calling a series of *malloc*
  - Assume allocation unit is 100 bytes

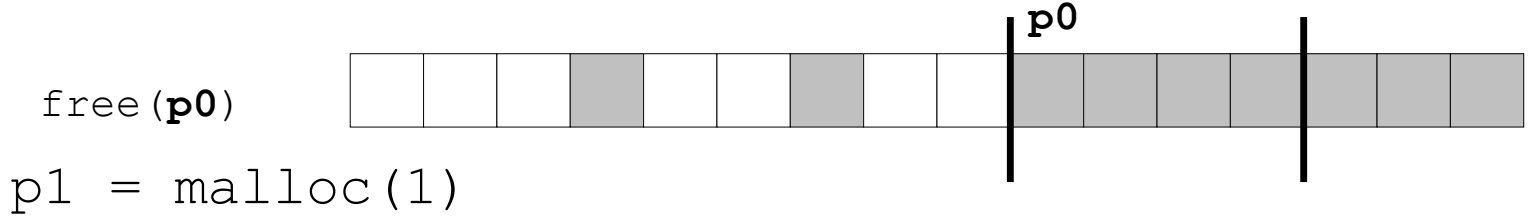
```
char* first = malloc(90);
char* second = malloc(120);
char* third = malloc(10);
char* problem = malloc(50);
```

- All of memory has been allocated but only a fraction of it is used (220 bytes)
  - Unable to handle final memory request



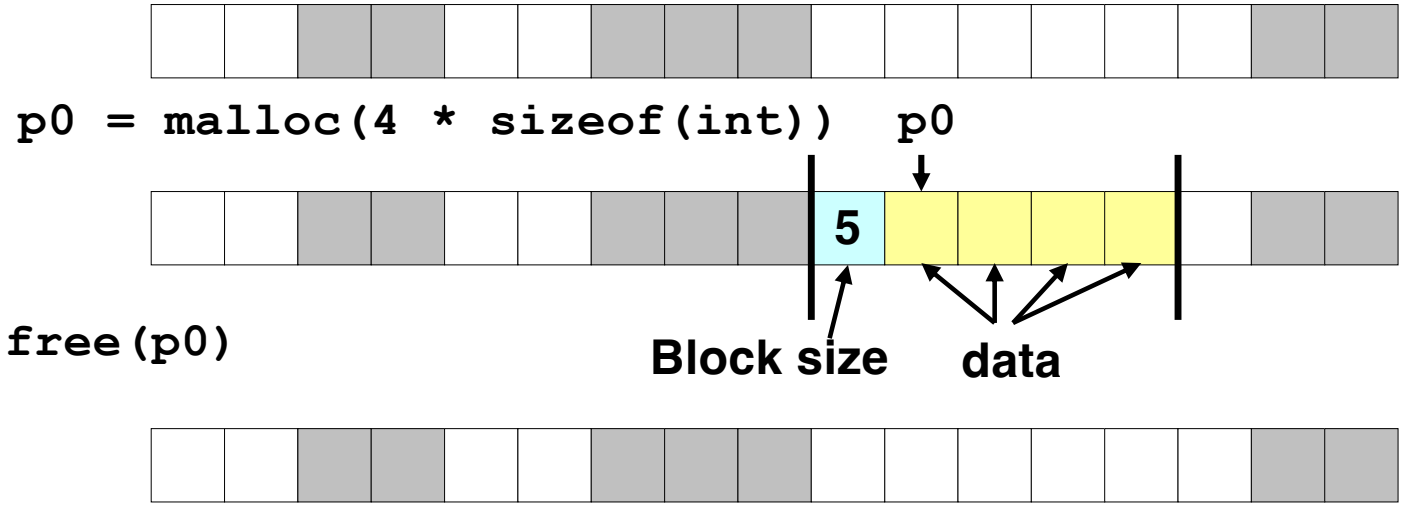
Dynamic Memory Allocation Challenges

- How do we know how much memory to free just given a pointer?
- How do we keep track of the free blocks?
- What do we do with the extra space when allocating a structure that is smaller than the free block it is placed in?
- How do we pick a block to use for allocation -- many might fit?
- How do we reinsert freed block?



Knowing How Much to Free

- Standard method
  - Keep the length of a block in the memory slot preceding the block
    - This slot is often called the header field or header
  - Requires a slot for every allocated block



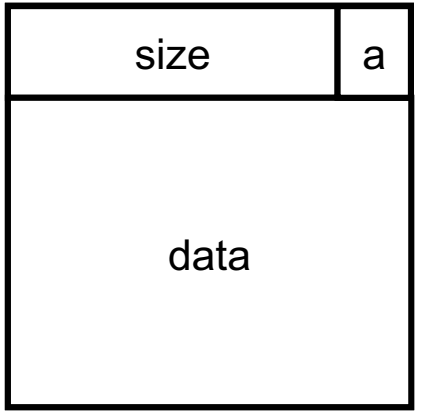
Today's Class

- Dynamic memory allocations
- Fragmentation
- Allocation using implicit list
- Allocation using buddy allocator

Implicit List: Keeping Track of Memory Blocks

- Need to identify whether each block is free or allocated
  - Block header at the start of each block of memory
    - Header is a structure with size and status

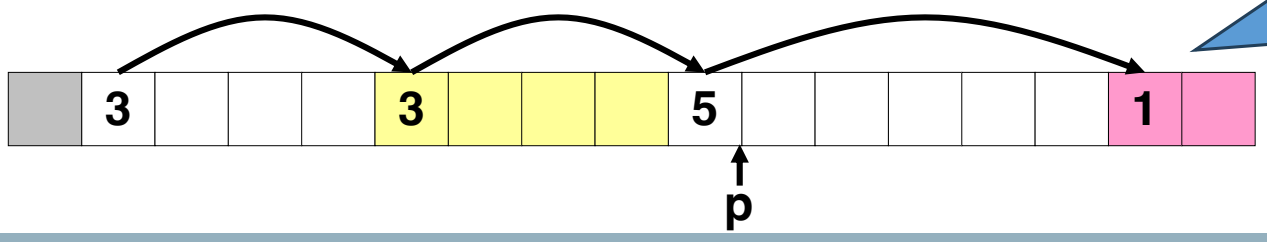
Format of allocated and free blocks



a = 1: allocated block
a = 0: free block

size: block size

data: application data (allocated blocks only)



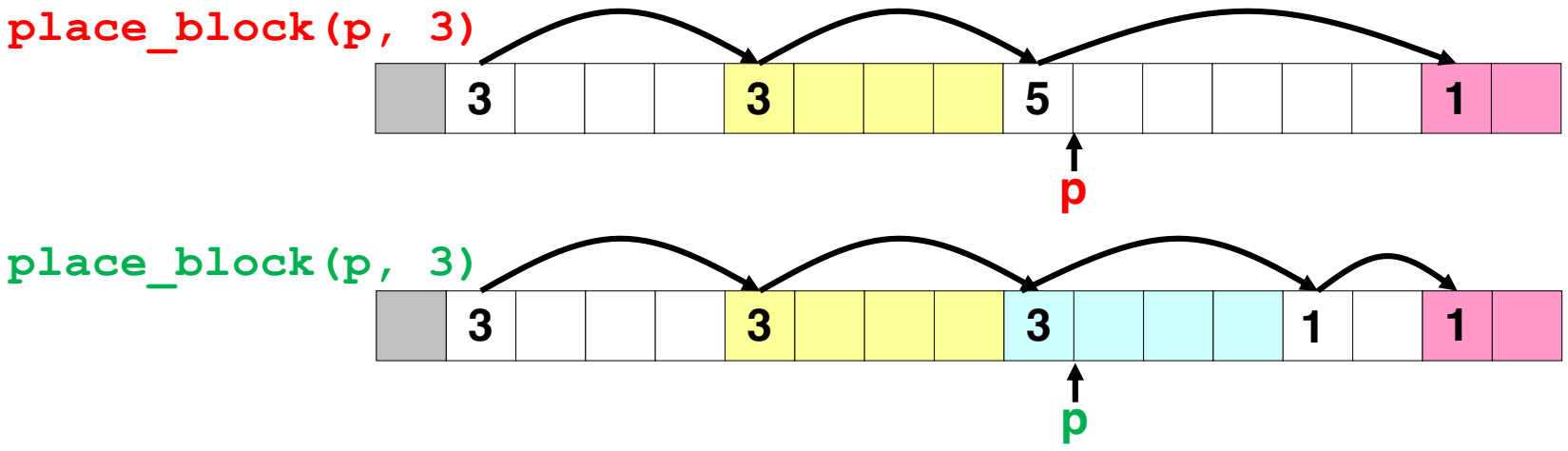
Blocks of memory (both free or used) are connected to each other as linked list

Implicit List: **Finding a Free Block**

- First fit:
  - Search list from beginning, choose first free block that fits
    - May miss a block with closest size that fits
  - Can take linear time in total number of blocks (allocated/free)
- Next fit:
  - Like first-fit, but search list from end of previous search
  - Research suggests that fragmentation is worse
    - May miss a block with closest size that fits
- **Best fit**:
  - Choose the free block with the closest size that fits
    - Requires complete search of the list
  - Keeps fragments small – usually helps fragmentation
  - Will typically run slower than first-fit

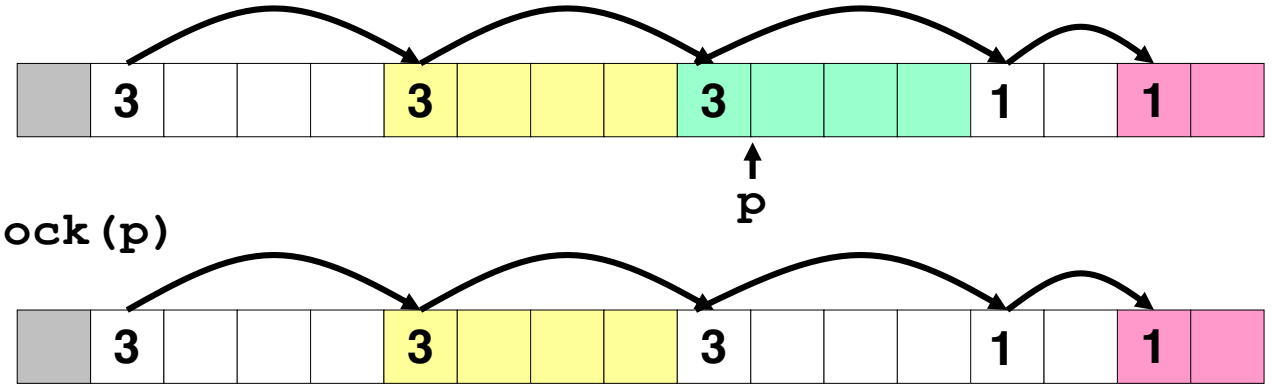
Implicit List: **Allocating in Free Block**

- Allocating in a free block
  - Since allocated space might be smaller than free space, we might want to split the block (Splitting)



Implicit List: **Freeing a Block**

- Simplest implementation:
  - Only need to clear allocated flag
  - But can lead to “false fragmentation”

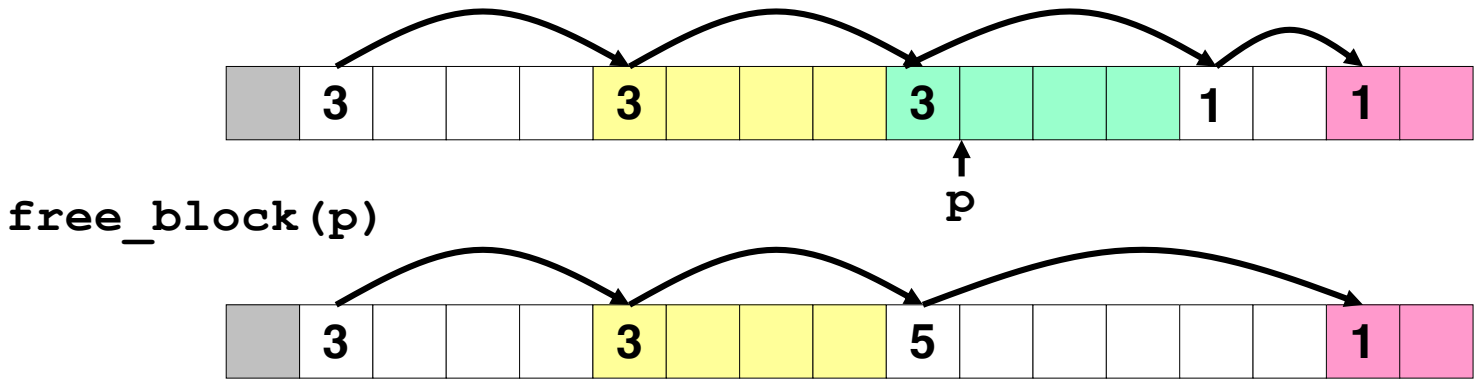


`malloc(4 * sizeof(int))` **Oops!**

  - There is enough free space, but the allocator won’t be able to find it!

Implicit List: **Coalescing**

- Join (coalesce) with next and/or previous block **if free**
  - Coalescing with **next** block

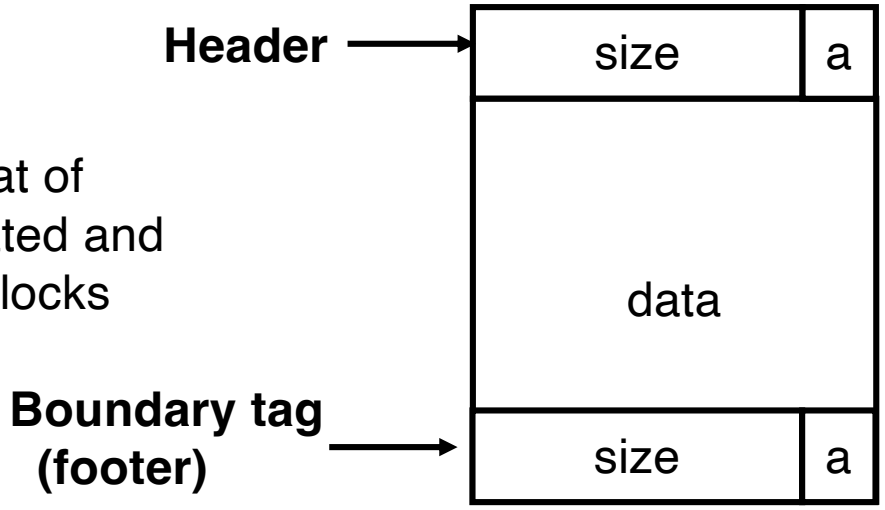


  - But how do we coalesce with previous block?

Implicit List: **Bidirectional Coalescing**

- Boundary tags [Knuth73]
  - Replicate header at end of block
  - Allows us to traverse the "list" backwards, but requires extra space

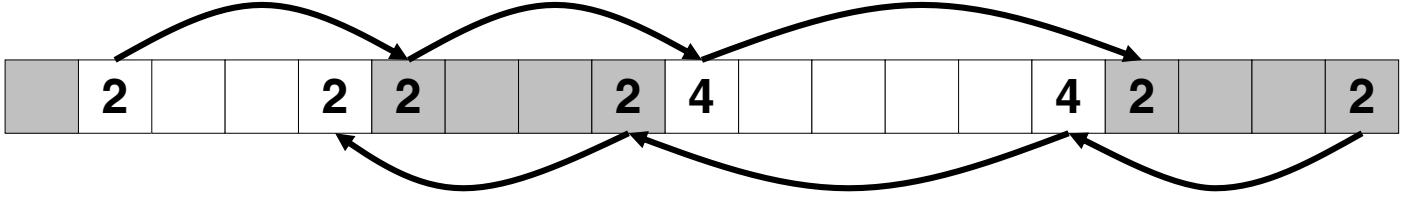
Format of allocated and free blocks



a = 1: allocated block
a = 0: free block

size: total block size

application data (allocated blocks only)



Today's Class

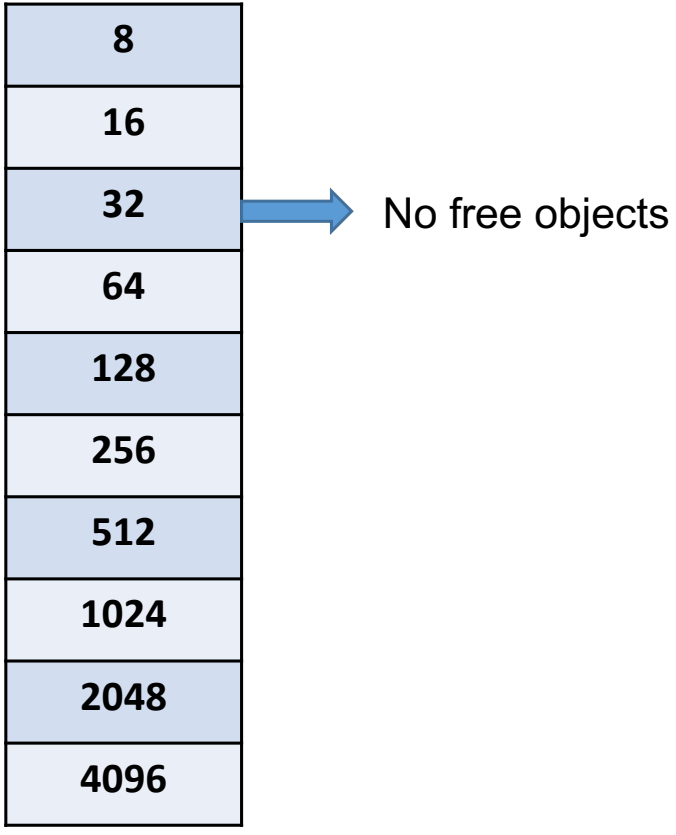
- Dynamic memory allocations
- Fragmentation
- Allocation using implicit list
- Allocation using buddy allocator

Buddy Allocator

8
16
32
64
128
256
512
1024
2048
4096

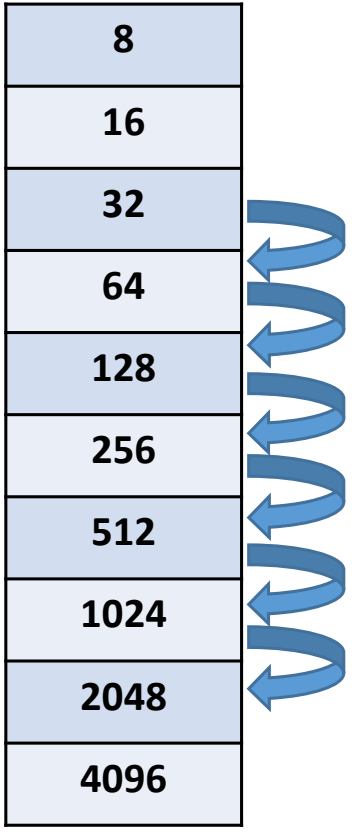
- Implicit list
  - Memory usage depends on placement policy
    - First fit, next fit, or best fit
  - Can lead to fragmentation
- **Buddy allocator**
  - It maintains free buckets of memory objects of different allocation sizes (2^k)
  - Initially all buckets are empty
  - No fragmentation

Buddy Allocator



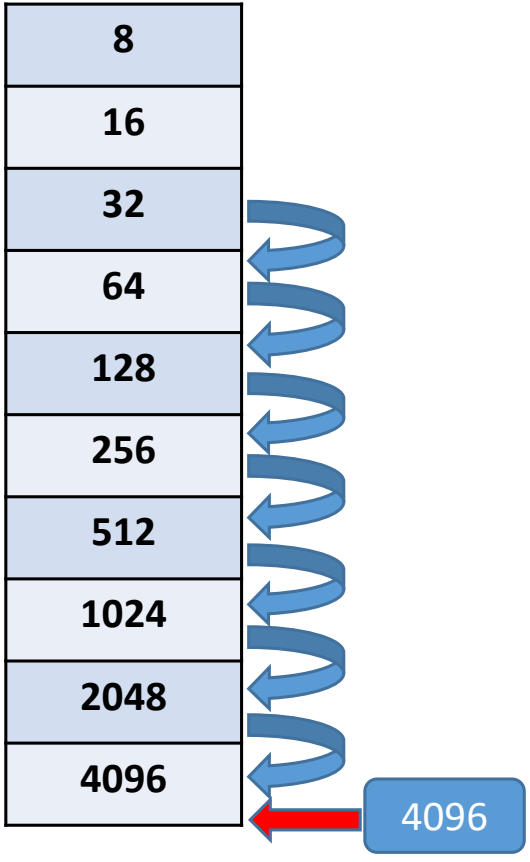
- A= malloc(24)
  - Rounded size = 32
    - Yes, little bit of fragmentation, but let's ignore that..
  - No free objects in that bucket

Buddy Allocator



- Allocator recursively calls itself to allocate an object that is twice the size of the bucket size
- Call would eventually lead to bucket 4096

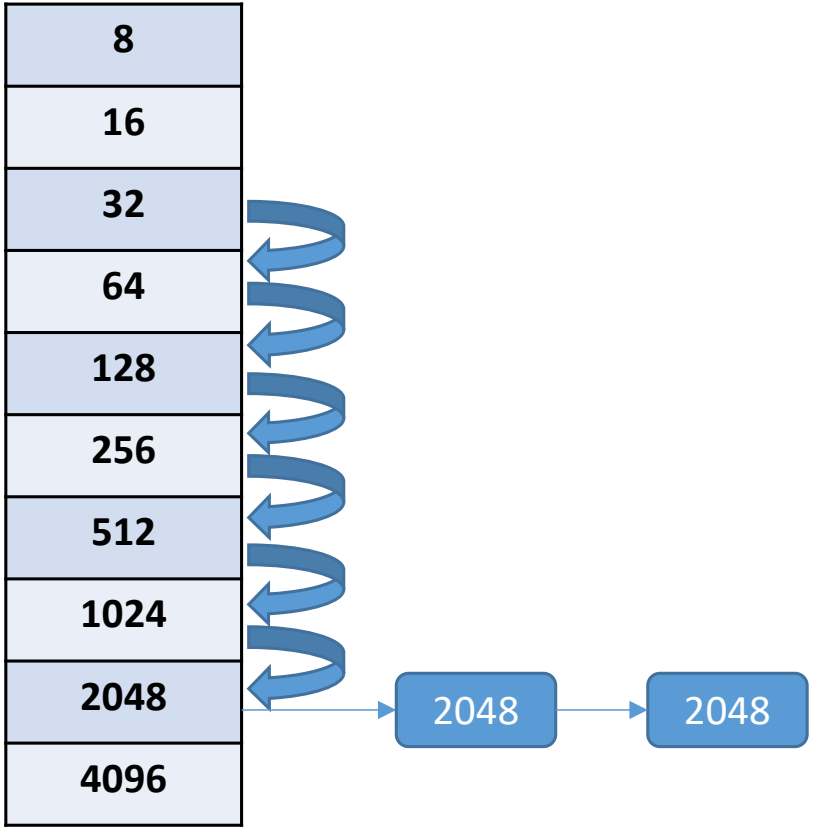
Buddy Allocator



No free objects. Hence, allocate 4Kb from RAM

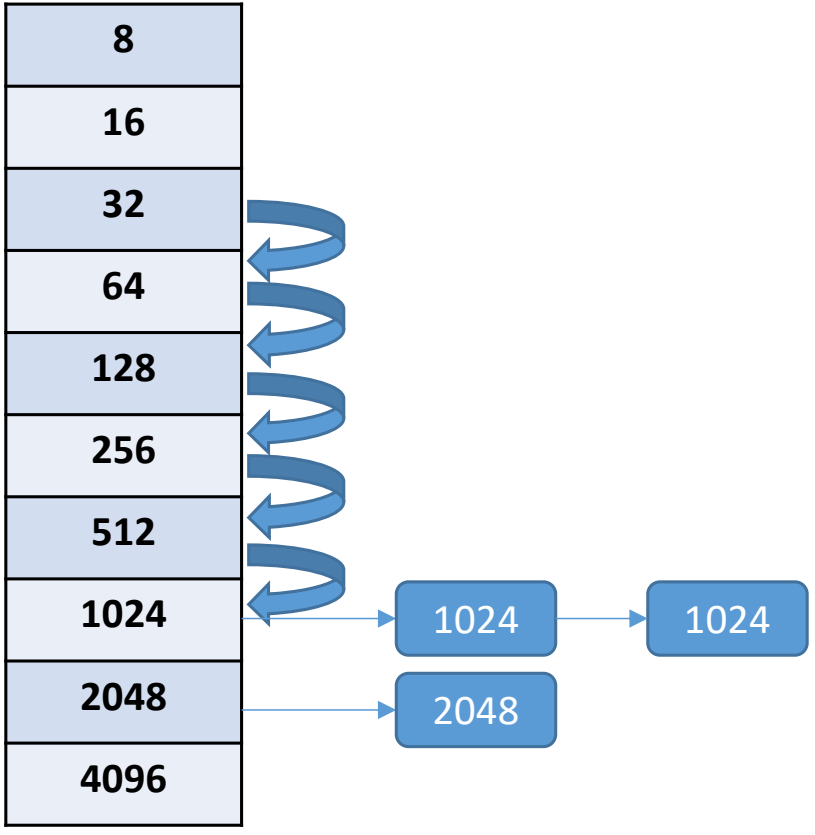
- There is no free object in the bucket size 4096
  - Request OS to allocate a chunk of size 4096 bytes from the RAM and return to its caller
    - System call
    - 4096 bytes is also called as page size in Linux (will be covered in Lecture 17)

Buddy Allocator



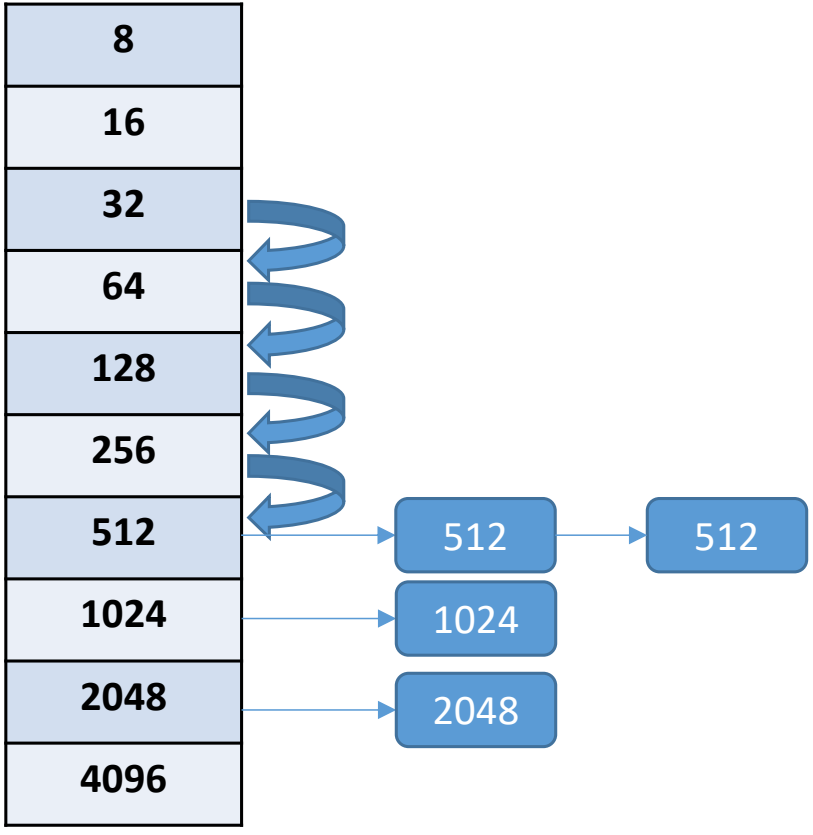
- Split the 4096 object into two halves (two 2048 objects), add into the parent bucket
  - Bucket 2048

Buddy Allocator



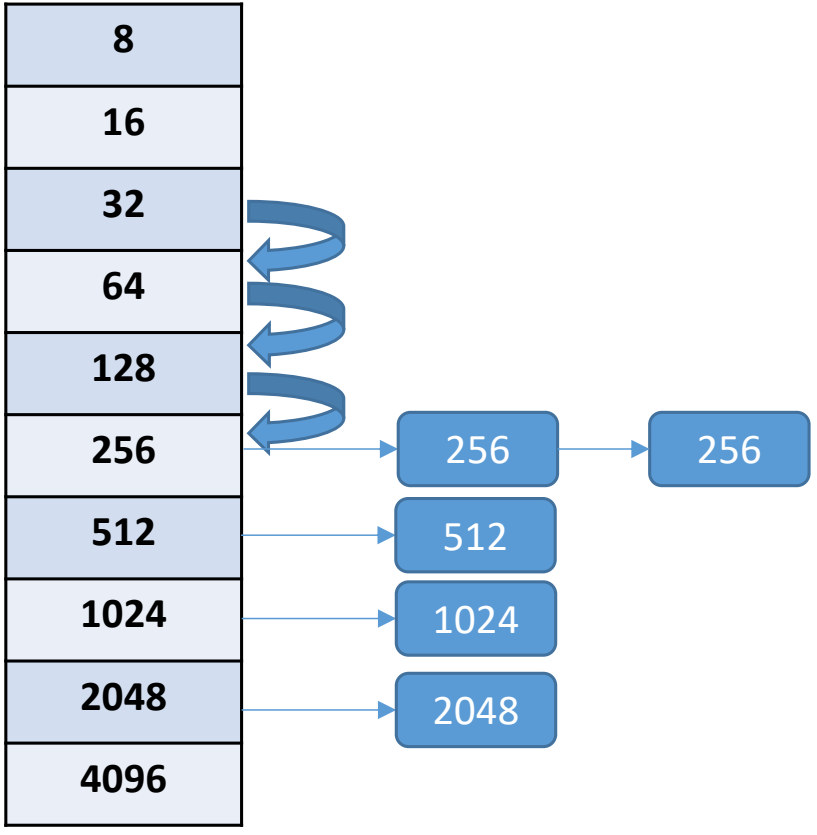
- One of the 2048 object is promoted to the bucket 1024 where it is split into two objects of size 1024 each

Buddy Allocator



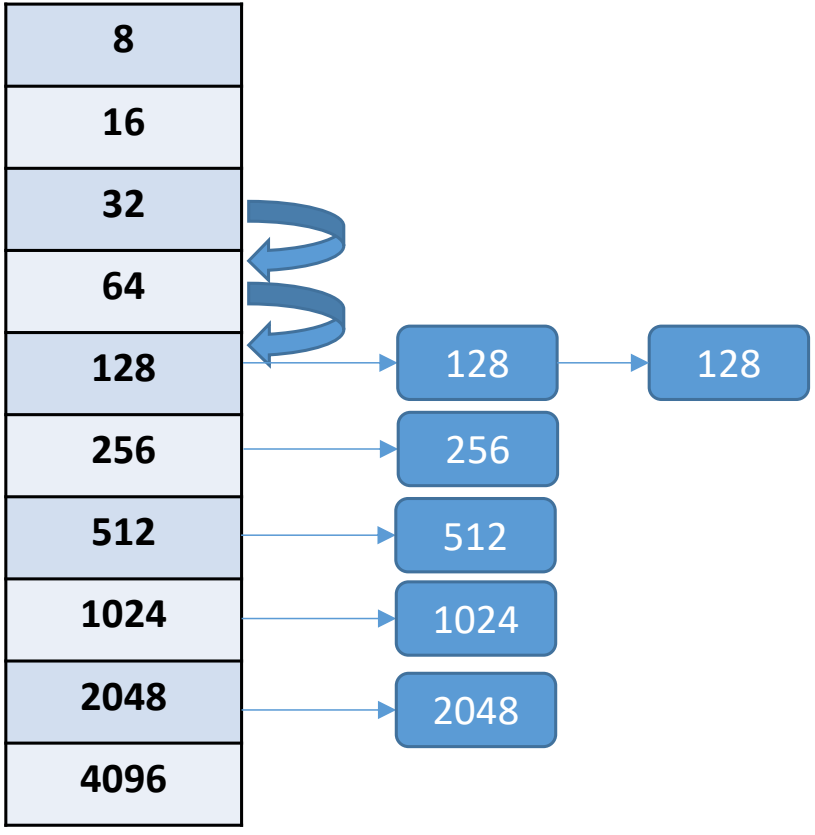
- One of the 1024 object is promoted to the bucket 512 where it is split into two objects of size 512 each

Buddy Allocator



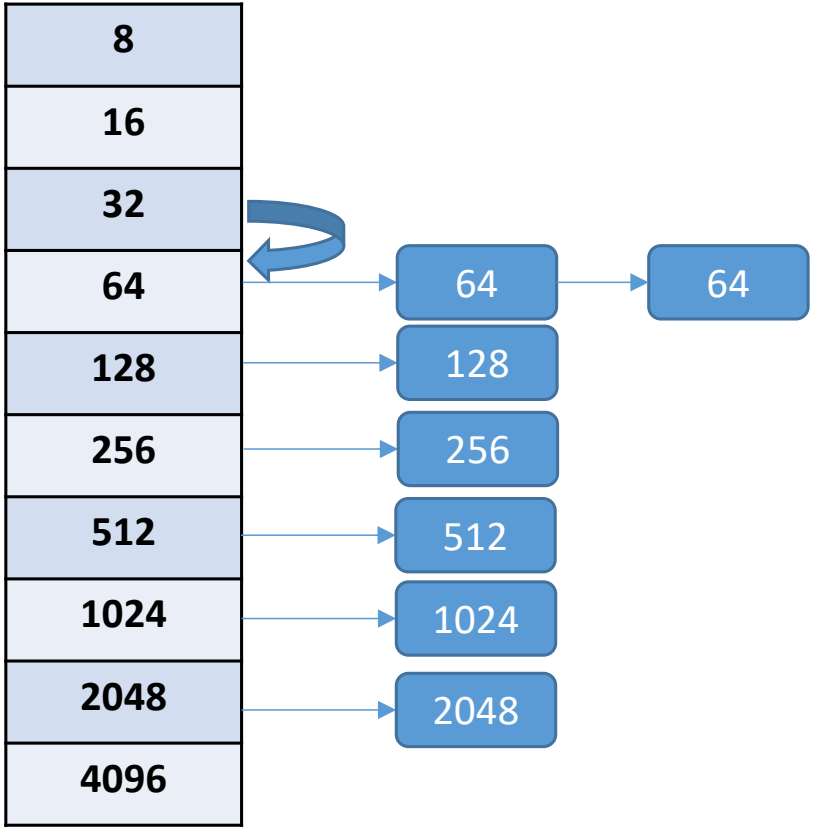
- One of the 512 object is promoted to the bucket 256 where it is split into two objects of size 256 each

Buddy Allocator



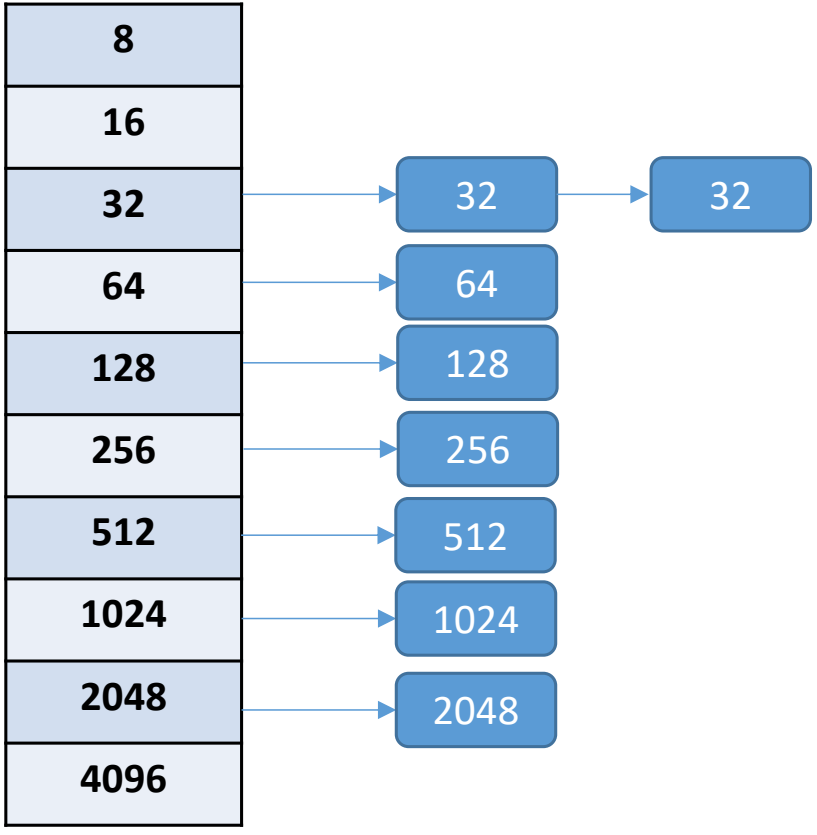
- One of the 256 object is promoted to the bucket 128 where it is split into two objects of size 128 each

Buddy Allocator



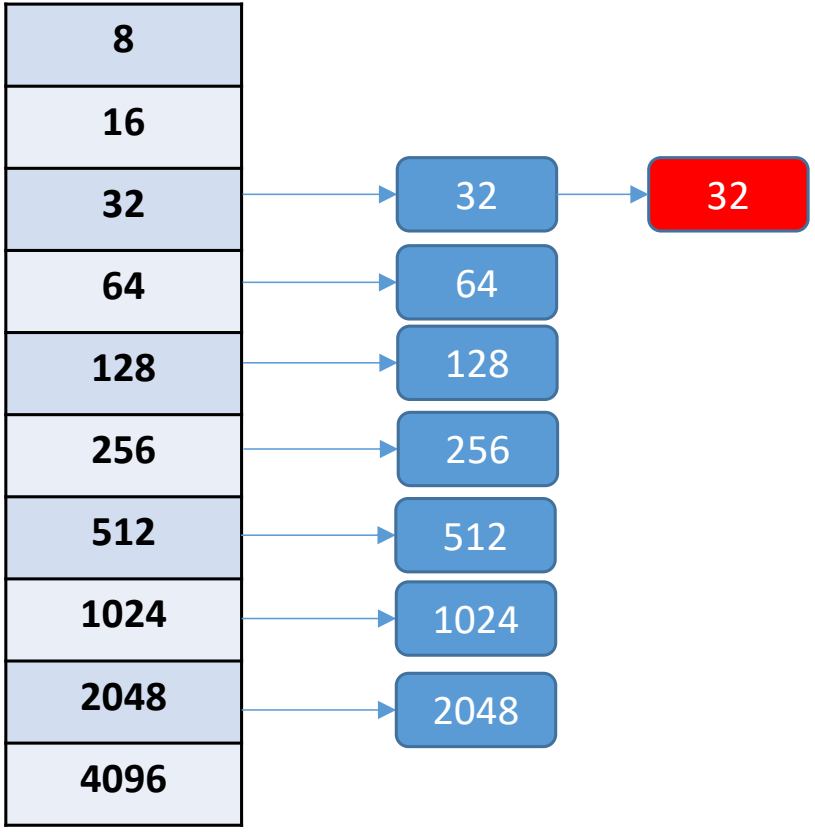
- One of the 128 object is promoted to the bucket 64 where it is split into two objects of size 64 each

Buddy Allocator



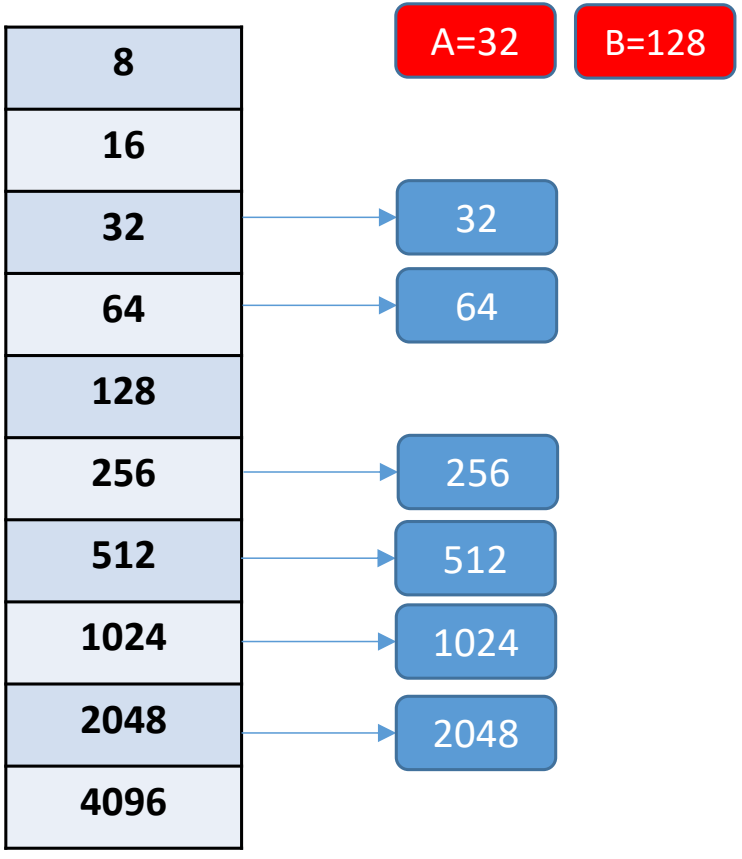
- One of the 64 object is promoted to the bucket 32 where it is split into two objects of size 32 each

Buddy Allocator



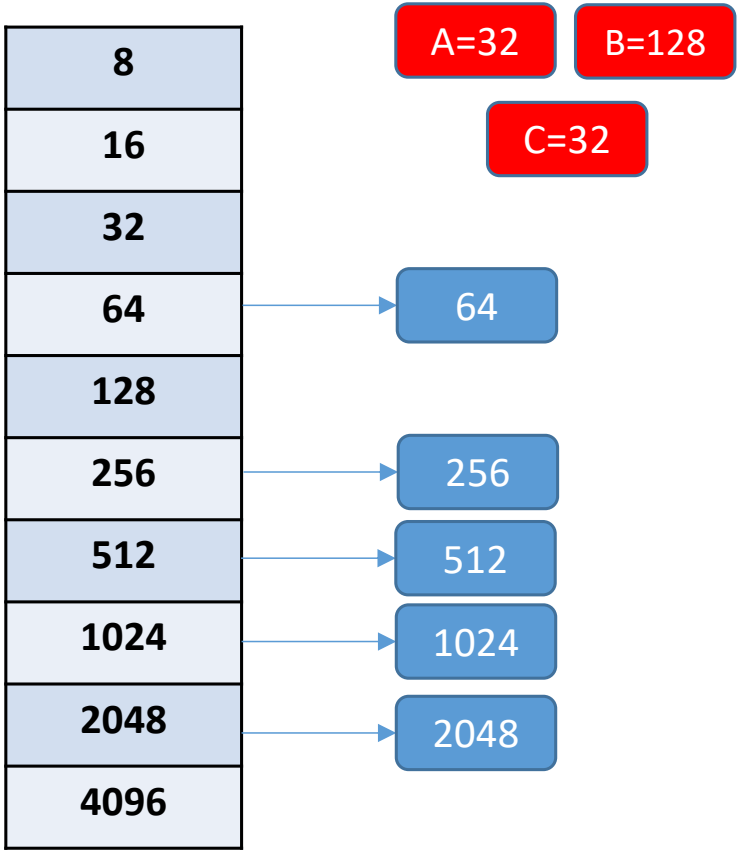
- One of the 32-byte object in bucket 32 will now be returned to the malloc(32) call

Buddy Allocator



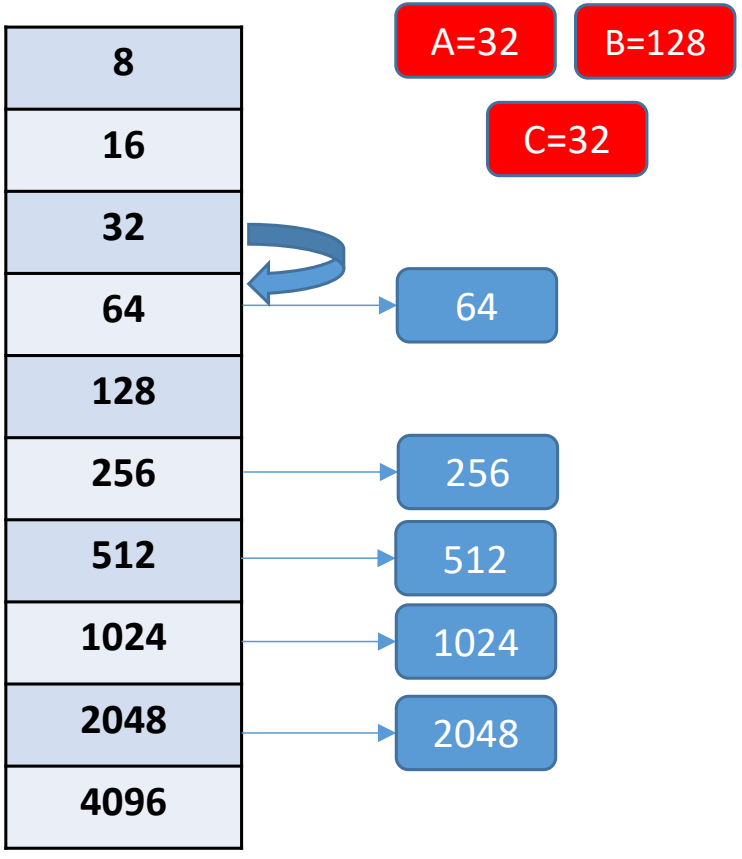
- B = malloc(128)
  - One free object already in that bucket 128
  - Return that free object from the bucket to the malloc(128)
  - Now no more free object in bucket 128

Buddy Allocator



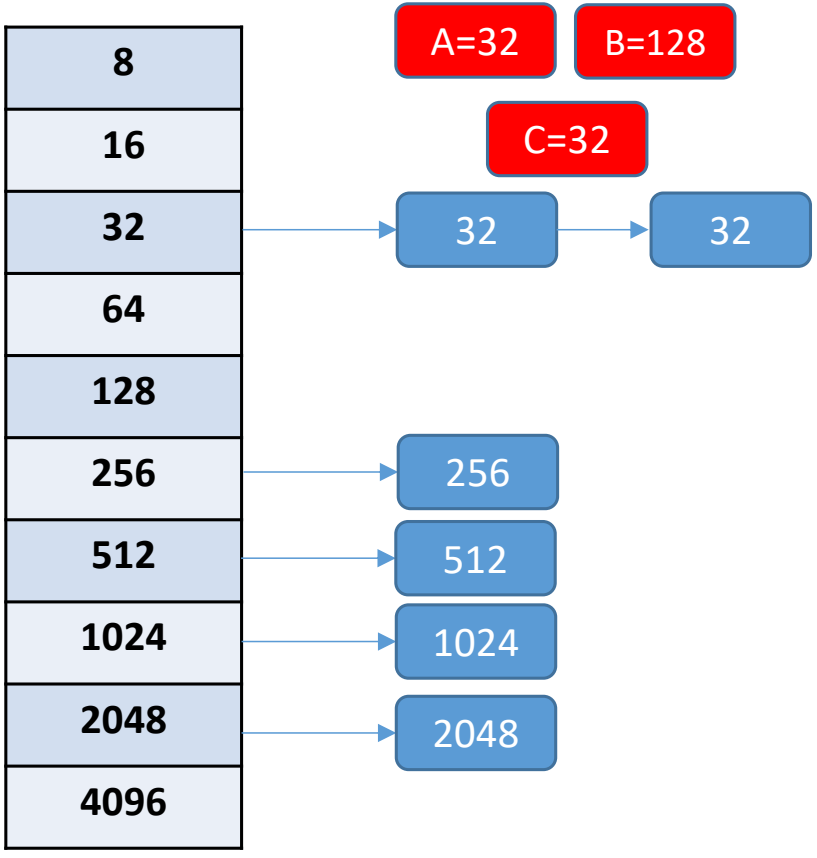
- C = malloc(32)
  - One free object already in that bucket
  - Return that free object from the bucket to the malloc(32)
  - Now no more free object in bucket 32

Buddy Allocator



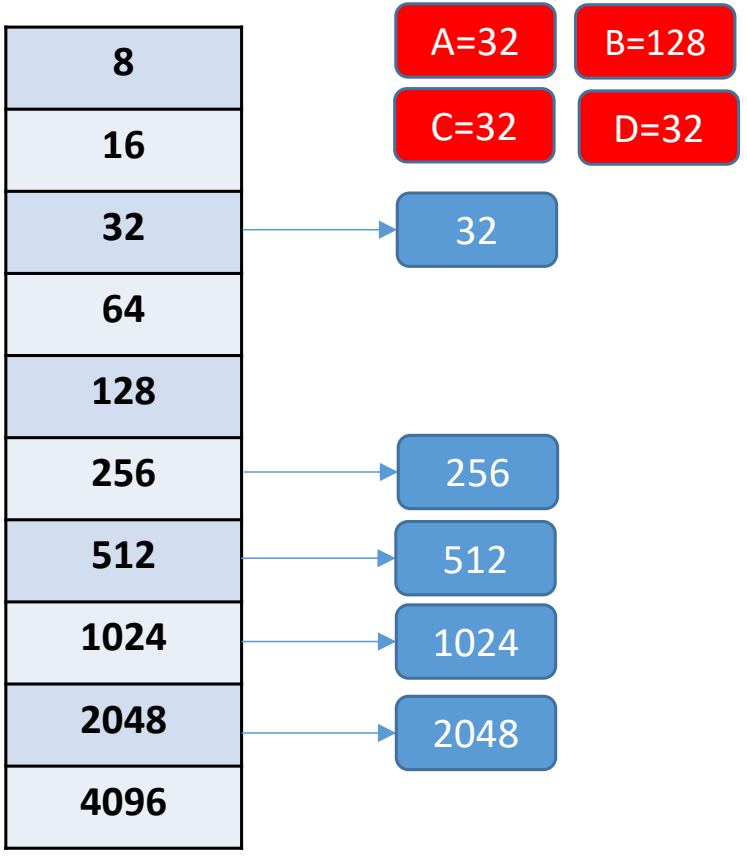
- D = malloc(32)
  - No more free object exists in the bucket 32
  - Fetch one object from the buddy bucket 64

Buddy Allocator



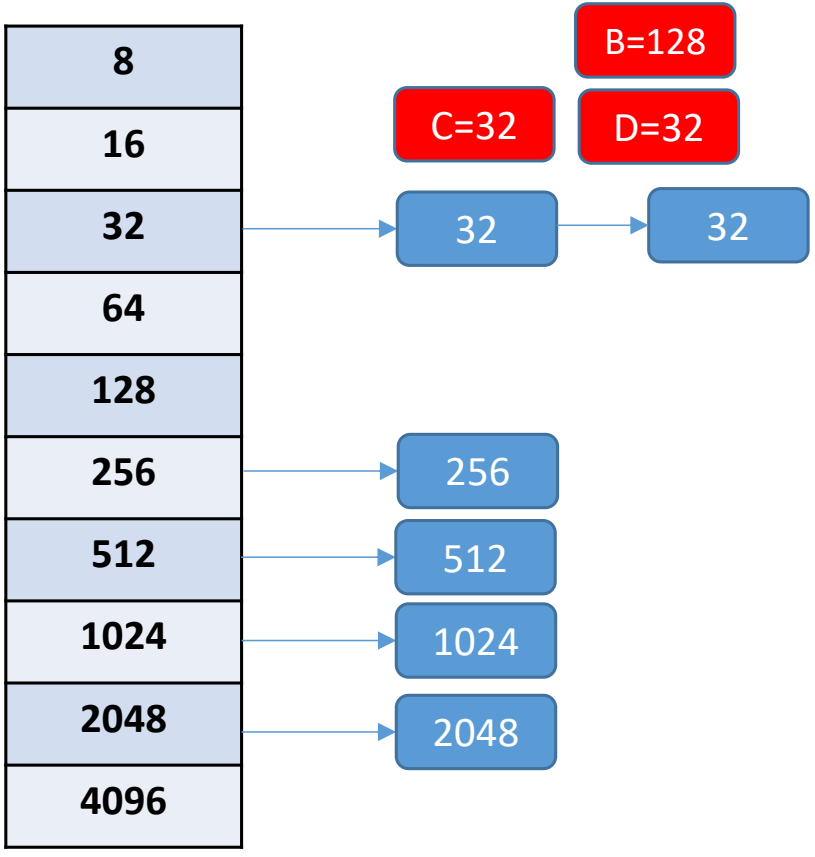
- Object of 64 bytes received from buddy bucket 64 will be split into two 32 byte free objects and added into the bucket 32

Buddy Allocator



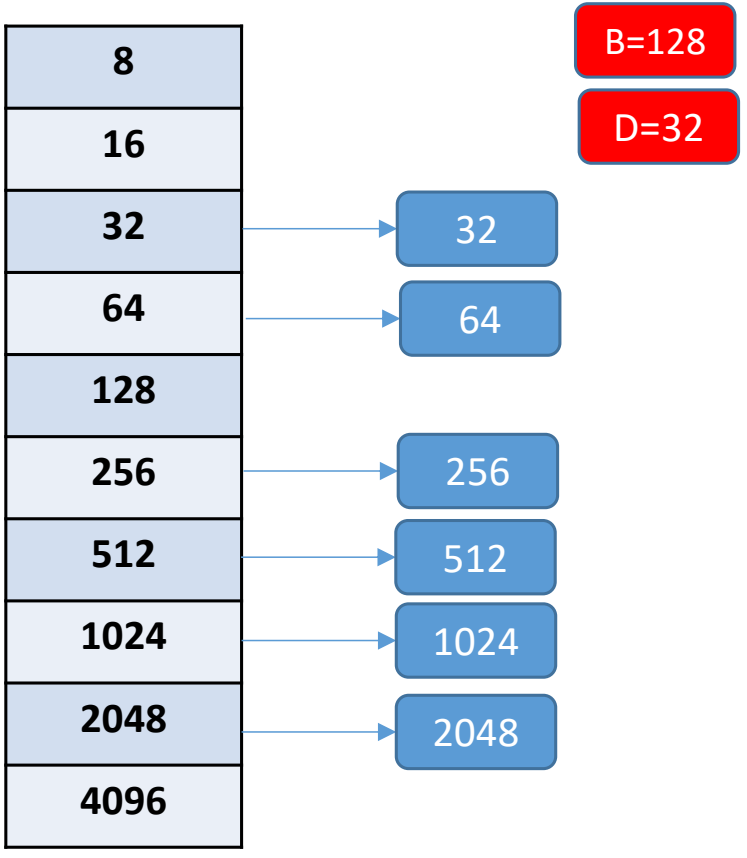
- One free object from bucket 32 returned to malloc(32) call

Buddy Allocator



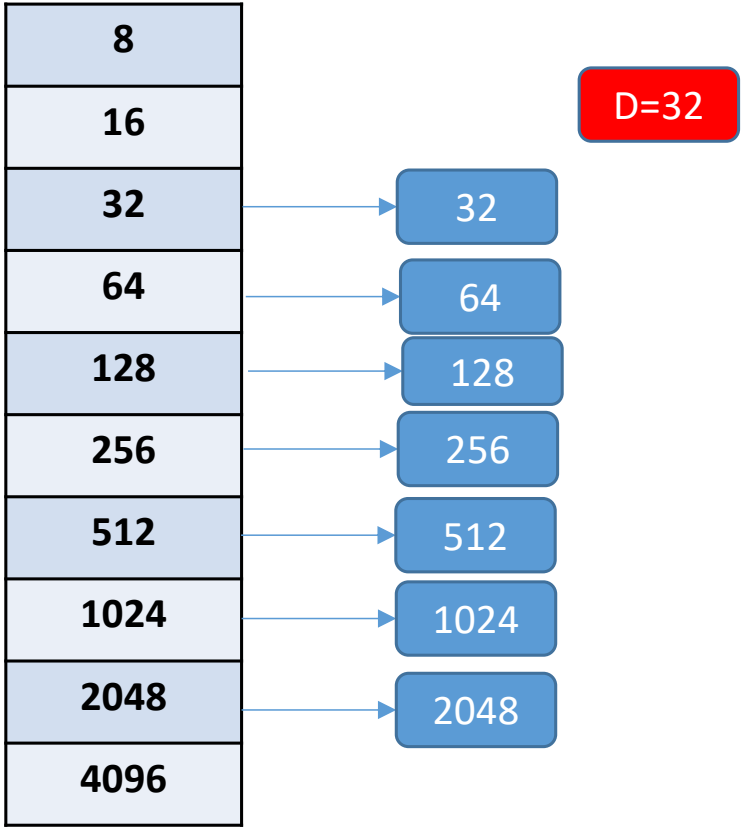
- free(A)
  - Append to the free object list in bucket 32

Buddy Allocator



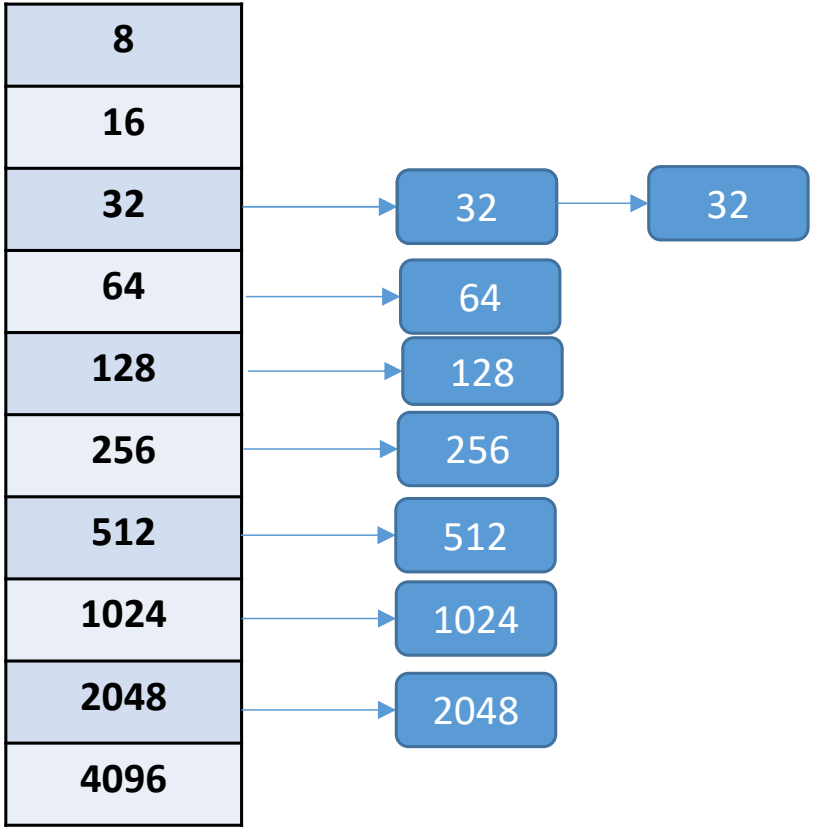
- free(C)
  - One more memory object appended to the free list in bucket 32 which results in a total of 3 free objects in that bucket
  - Two of the free objects are coalesced (merged) and promoted to bucket 64
- If merging and promotion doesn't happen then it will result in unnecessary calls to OS for 4Kb objects
  - System call invocation, thereby leading to overheads!

Buddy Allocator



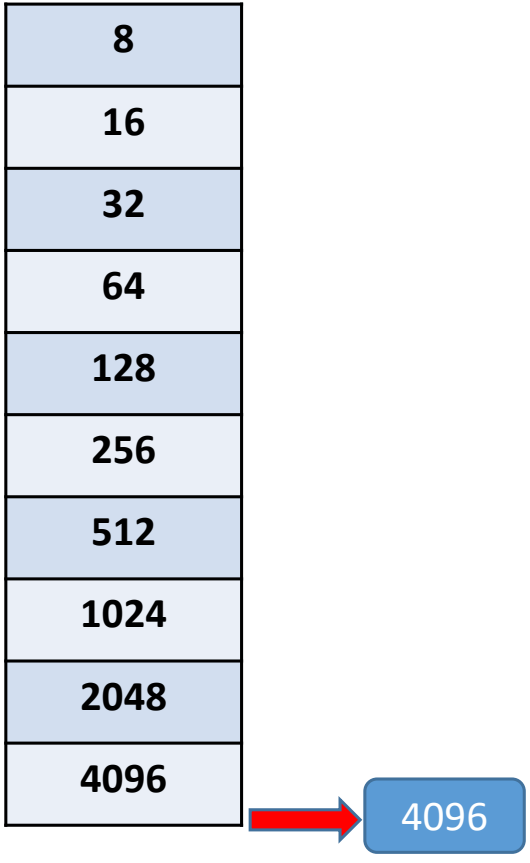
- free(B)
  - Append to the bucket 128

Buddy Allocator



- free(D)
  - Append the memory object back to the bucket (32)
- Finally, during process termination, all free objects are coalesced back into 4096 bucket and returned to the OS

Buddy Allocator



- All free objects are coalesced back into 4096 bucket and returned to the OS at process termination
  - What would happen if memory not returned to OS?
- How to handle allocations greater than 4096 bytes?
  - The size is rounded up to the nearest multiples of 4096 and allocated directly from OS

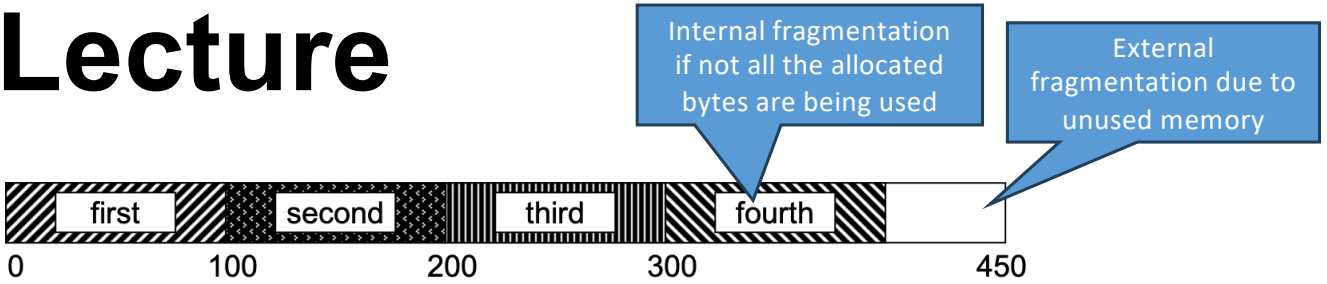
Next Lecture

- Virtual memory

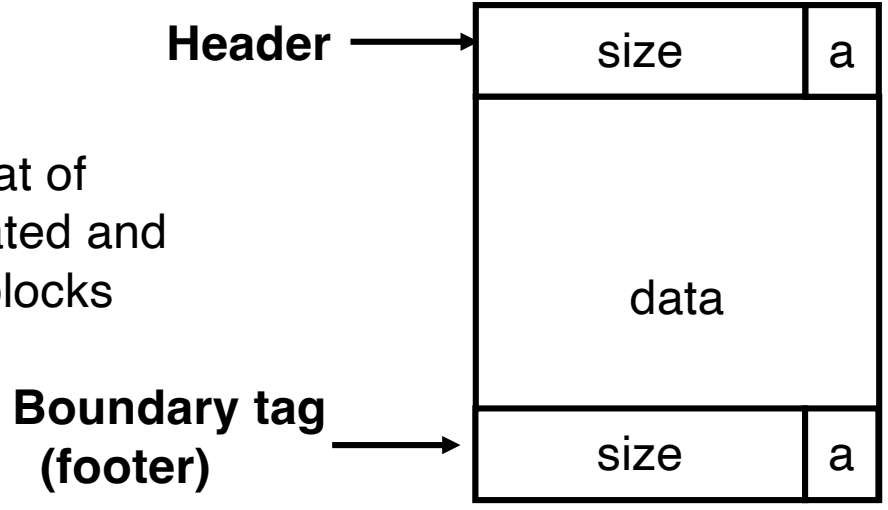
Lecture 15: Virtual Memory

Vivek Kumar
Computer Science and Engineering
IIIT Delhi
vivekk@iiitd.ac.in

Last Lecture



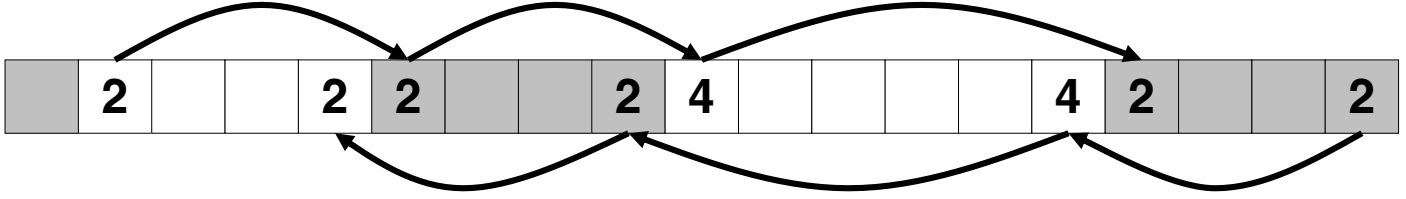
Format of allocated and free blocks



a = 1: allocated block
a = 0: free block

size: total block size

application data (allocated blocks only)

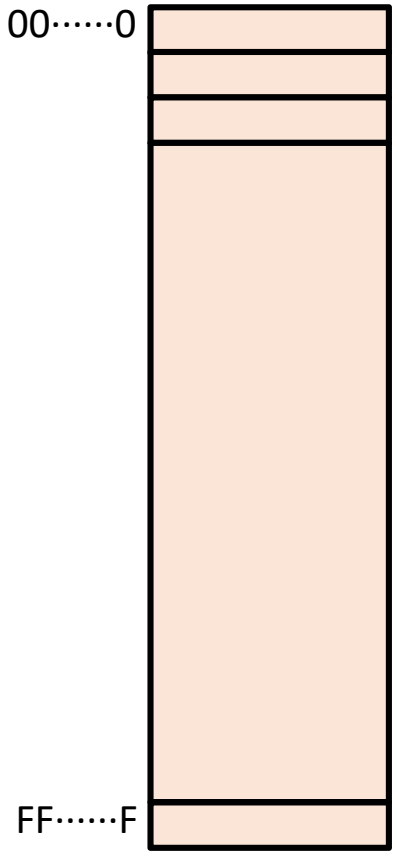


- Fragmentation
  - Internal and external
- Implicit list
  - Memory header
  - Memory blocks as linked list
  - Splitting on allocation
  - Coalescing on free

Today's Class

- **Introduction to virtual memory**
- Mapping virtual to physical memory
- Dynamic relocation using base/bound registers

Virtual Memory

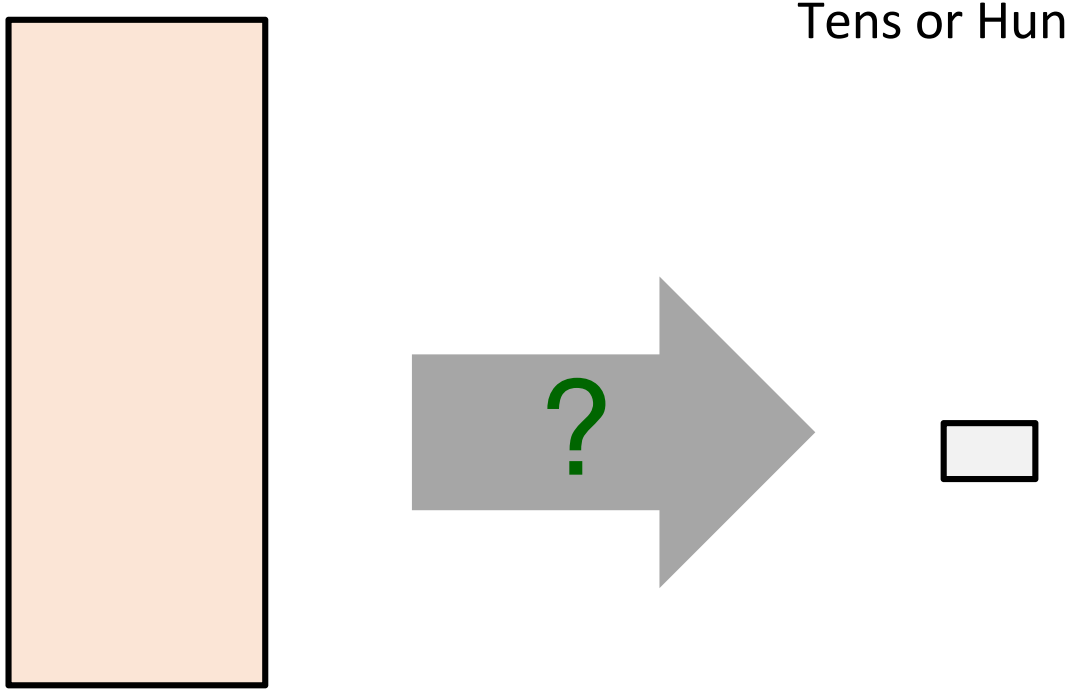


- Programs refer to memory using virtual memory addresses
  - `int* ptr = malloc(4);`
  - Conceptually very large array of bytes
  - Each byte has its own address
  - Operating system provides address space private to particular "process"
- Compiler and run-time system allocate VM
  - Where different program objects should be stored
  - All allocation within single virtual address space

Problem 1: How Does Everything Fit?

64-bit addresses:
16 Exabyte (16 billion GB!)

Physical main memory:
Tens or Hundreds of Gigabytes



And there are many processes ….

Problem 2: Memory Management

Physical main memory

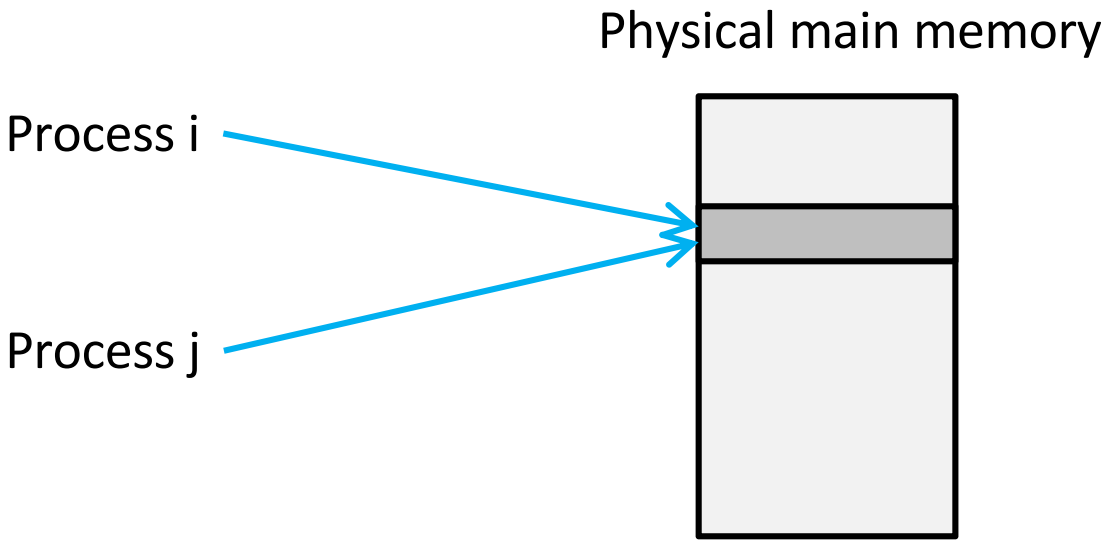
Process 1
Process 2
Process 3
…
Process n

X

stack
heap
`.text`
`.data`
…

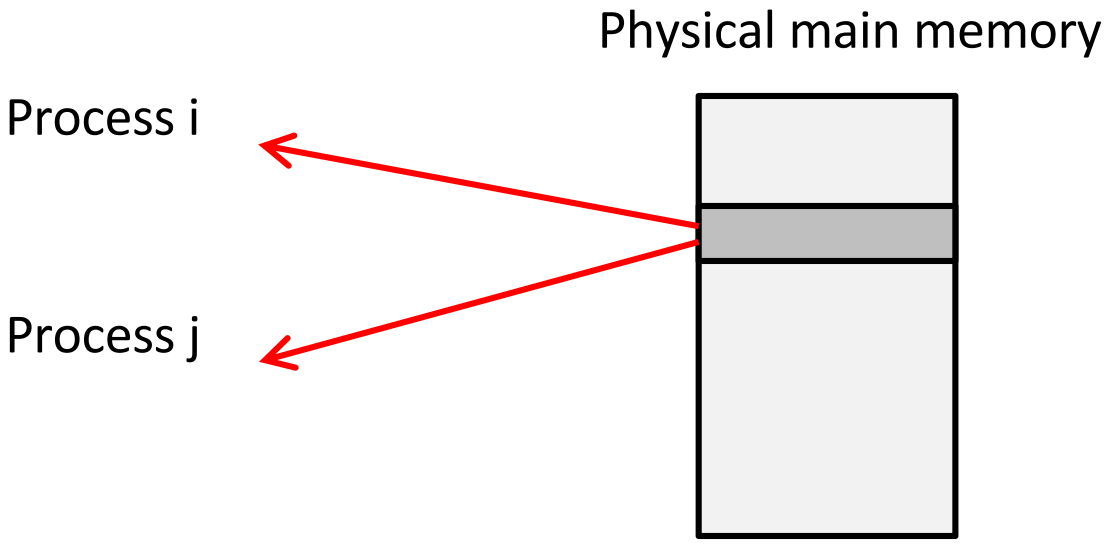
What goes where? There are several processes, but physical memory is limited

Problem 3: How To Protect?



- How to prevent Process-1 accessing the memory owned by Process-2 (unless shared)?

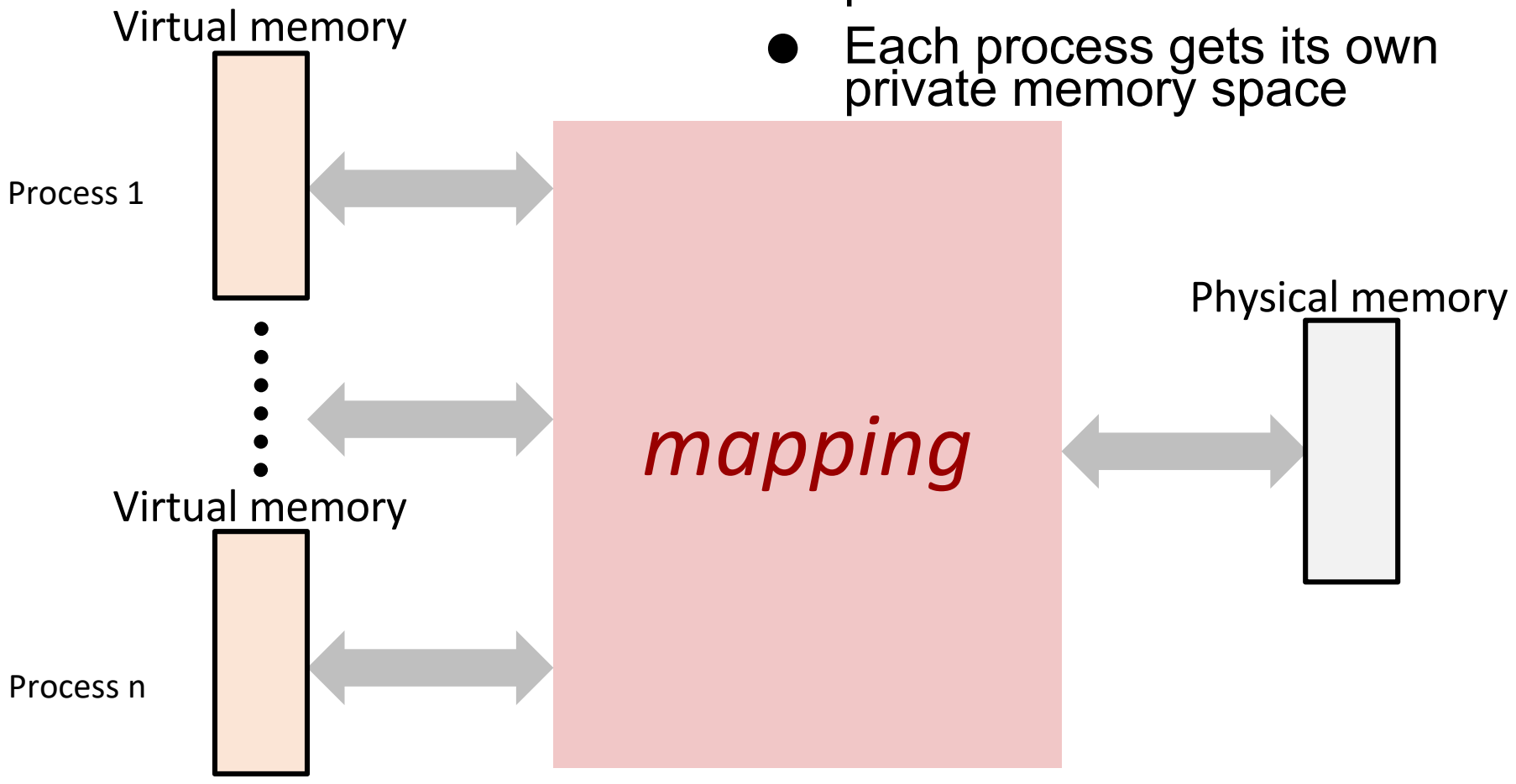
Problem 4: How To Share?



- How to share memory between two processes?

Solution: Indirection

- Using some mapping technique can solves all the previous problems
- Each process gets its own private memory space



Address Spaces

- Using some mapping technique to translate the address seen by the user to the actual address on the RAM gives two different views of addresses
  - Virtual address space (as seen by the process)
  - Physical address space (as seen by the OS depending on DRAM size)

- Virtual address space: Set of $N = 2^n$ virtual addresses
  $\{0, 1, 2, 3, \ldots, N-1\}$

- Physical address space: Set of $M = 2^m$ physical addresses
  $\{0, 1, 2, 3, \ldots, M-1\}$

Virtual Memory (VM)

- Efficient use of limited main memory (RAM)
  - Use RAM as a cache for parts of a virtual address space
    - Some non-cached parts stored on disk
  - Keep only active areas of virtual address space in RAM
    - Transfer data back and forth to disk as needed
- Simplifies memory management for programmers
  - Each process gets a full private address space
- Isolates address spaces
  - One process can't interfere with another's memory
    - Because they operate in different address spaces

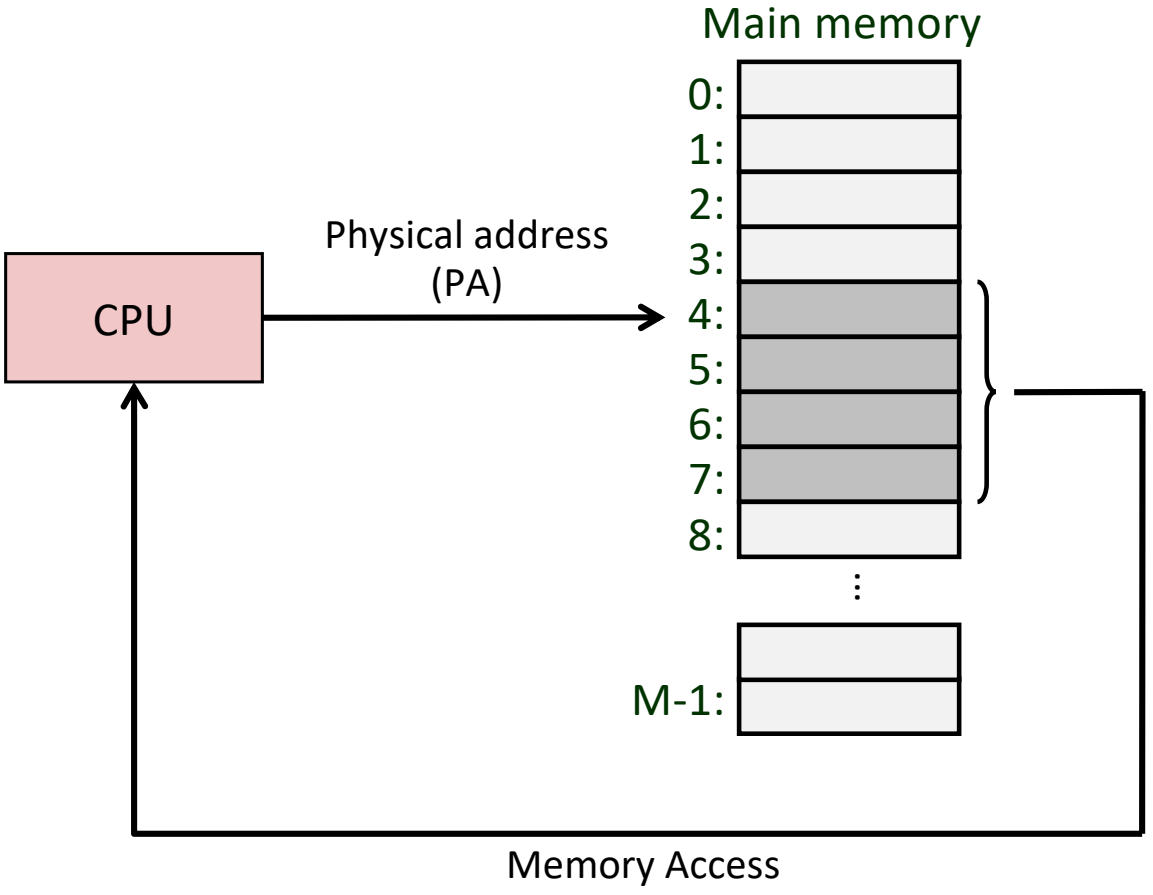
Today's Class

- Introduction to virtual memory
- **Mapping virtual to physical memory**
- Dynamic relocation using base/bound registers

Mapping Virtual to Physical Address

- Approaches
  1. Virtual address is same as the physical address

A System Using Physical Addressing

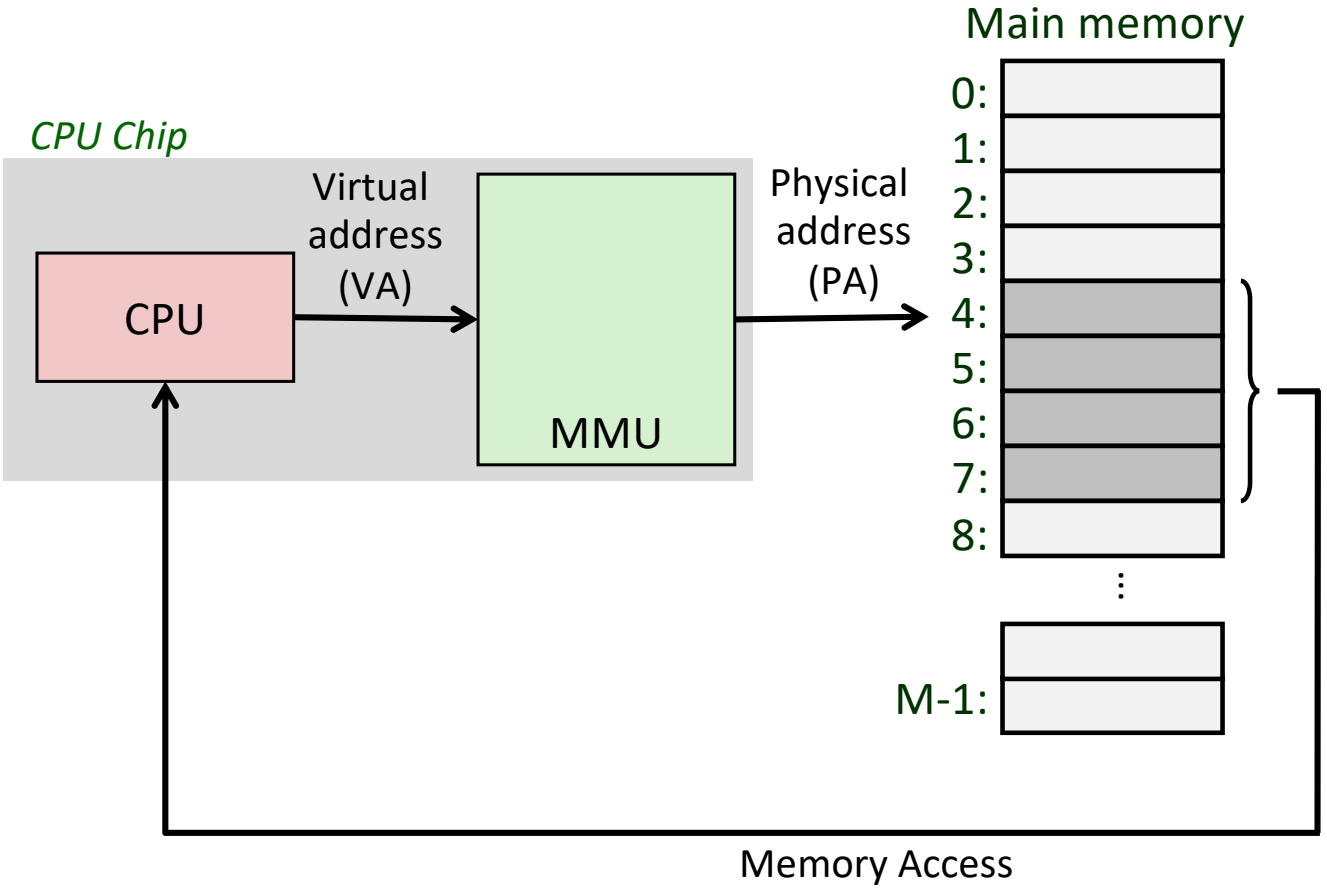


- Used in “simple” systems like embedded microcontrollers in devices like cars, elevators, and digital picture frames
  - Lowers production cost of the hardware (limited support of MMU)

Mapping Virtual to Physical Address

- Approaches
  1. Virtual address is same as the physical address
  2. Virtual address is not the same as physical address

A System Using Virtual Addressing



- Used in all modern servers, desktops, laptops, tablets, and cell phones
  - Why we cannot have VA same as PA in this case?

A System Using Virtual Addressing

```
vivek@possum:~/os23$ readelf -a ./fib
ELF Header:
  Magic:   7f 45 4c 46 02 01 01 00 00 00 00 00 00 00 00 00
  Class:                             ELF64
  Data:                              2's complement, little
  Version:                           1 (current)
  OS/ABI:                            UNIX - System V
  ABI Version:                       0
  Type:                              DYN (Position-Independe
  Machine:                           Advanced Micro Devices
  Version:                           0x1
  Entry point address:               0x1080
  Start of program headers:          64 (bytes into file)
  Start of section headers:          14056 (bytes into file)
  Flags:                             0x0
  Size of this header:               64 (bytes)
  Size of program headers:           56 (bytes)
  Number of program headers:         13
  Size of section headers:           64 (bytes)
  Number of section headers:         31
  Section header string table index: 30
```

- Can two processes have same virtual addresses?

  $./fib &

  $./fib &

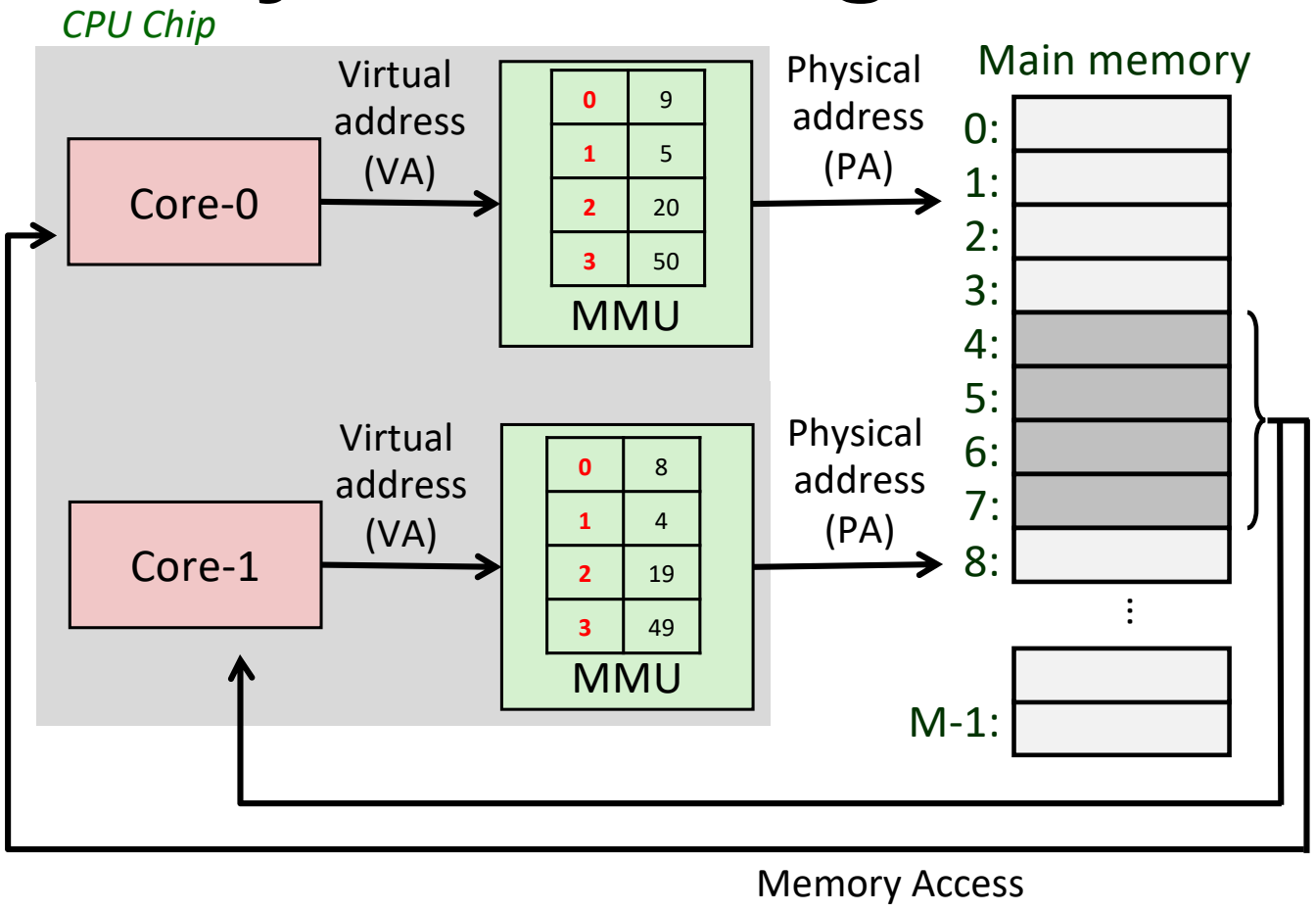
  $./fib &

  $./fib &

  $./fib &

  $./fib &

A System Using MMU Table



- Can two processes have same virtual addresses?
  - Yes, as long as the VA-PA mapping is different for both processes, and they are reflected in the **MMU table** during scheduling

A System Using MMU Table

```
void scheduler() {
    while(true) {
        lock(process_table);
        foreach(Process p: scheduling_algorithm(process_table)) {
            if(p->state != READY) {
                continue;
            }
            p->state = RUNNING;
            unlock(process_table);
            swtch_mmu_table(scheduler_process, p);
            swtch(scheduler_process, p);
            // p is done for now..
            lock(process_table);
        }
        unlock(process_table);
    }
}
```

- Can two processes have same virtual addresses?
  - Yes, as long as the VA-PA mapping is different for both processes, and they are reflected in the MMU table during scheduling
- How to support?
  - Save/restore the MMU table during context switching of the processes at each core

Issues with MMU Table Based Approach

- A process is not just going to have a few virtual addresses
  - It would be accessing several virtual addresses throughout its execution
- Not possible to store the entire VA-PA mapping inside per-process MMU table
  - Huge overhead in the operating system
- How about assigning a contiguous chunk of physical memory to each process?
  - **Any issues you can foresee?**
  - Nonetheless, let us see how to support it

Today's Class

- Introduction to virtual memory
- Mapping virtual to physical memory
- **Dynamic relocation using base/bound registers**

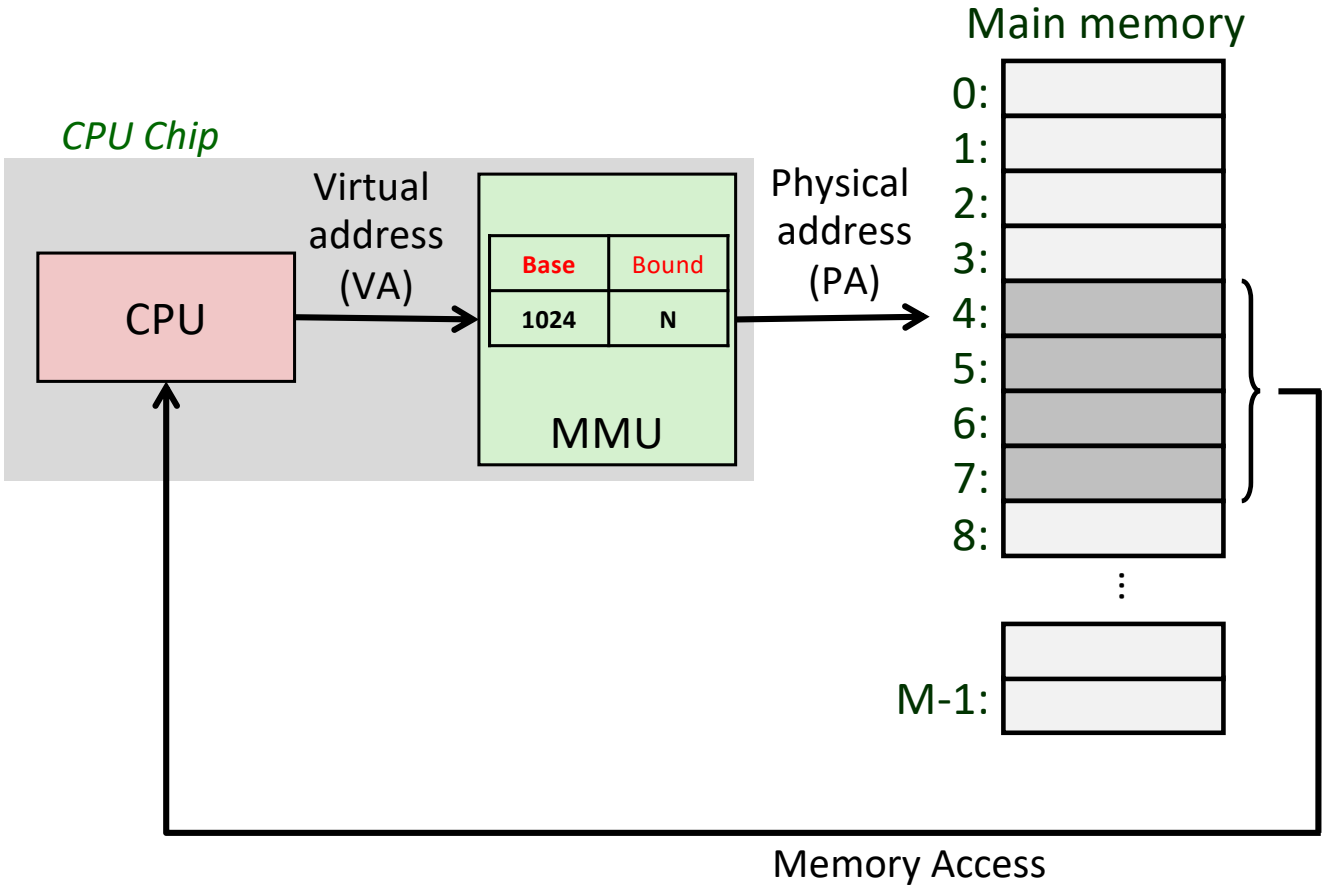
Dynamic Relocation

- "Relocation" means fixing an "indirection"
- Recall the "relocation" from the linker/loader lecture
  - In case of linker/loader, the linker created an "indirection" by just providing the pointers to the machine code inside the shared libraries
  - Loader used dynamic relocation during execution time by patching the pointers to the exact location of the machine code (e.g., printf's actual implementation in glibc)
- Similar approach of dynamic relocation can be used by the OS to decide the physical memory mapping for each processes

Mapping Virtual to Physical Address

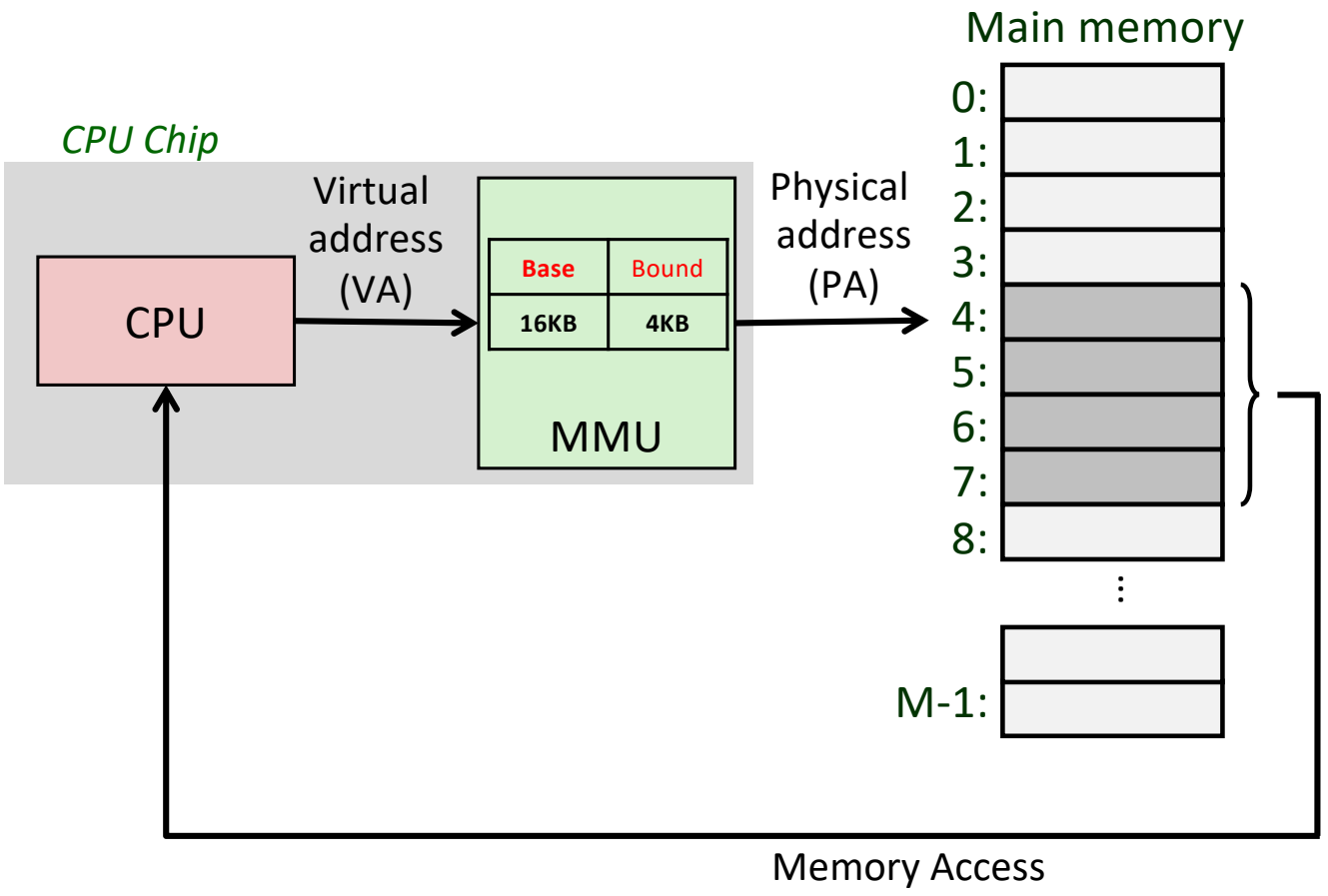
- Approaches
  1. Virtual address is same as the physical address
  2. Virtual address is not the same as physical address
     a) Using base & bound registers for VA to PA mapping

Approach-1: Using Base/Bound Registers



- Assume, during "exec", the loader somehow knows the exact memory requirement of the process (e.g., **N** bytes)
- OS will allocate a **contiguous** chunk of physical memory (a.k.a. **segment**) of **N** bytes for this process starting at some address. E.g., **1024**
- MMU will now have a pair of registers called as base (=**1024**) and bound (=**N** bytes)

Example: Base and Bound



- Each process will have a unique **base** value, but bound (**limit**) may be same for some processes (e.g., launching several fib executable)
- During execution, the value of base and bound can be loaded into respective registers to help in address translation from VA to PA
- Example for an address translation: A process with 4KB address space placed at index 16KB in physical memory

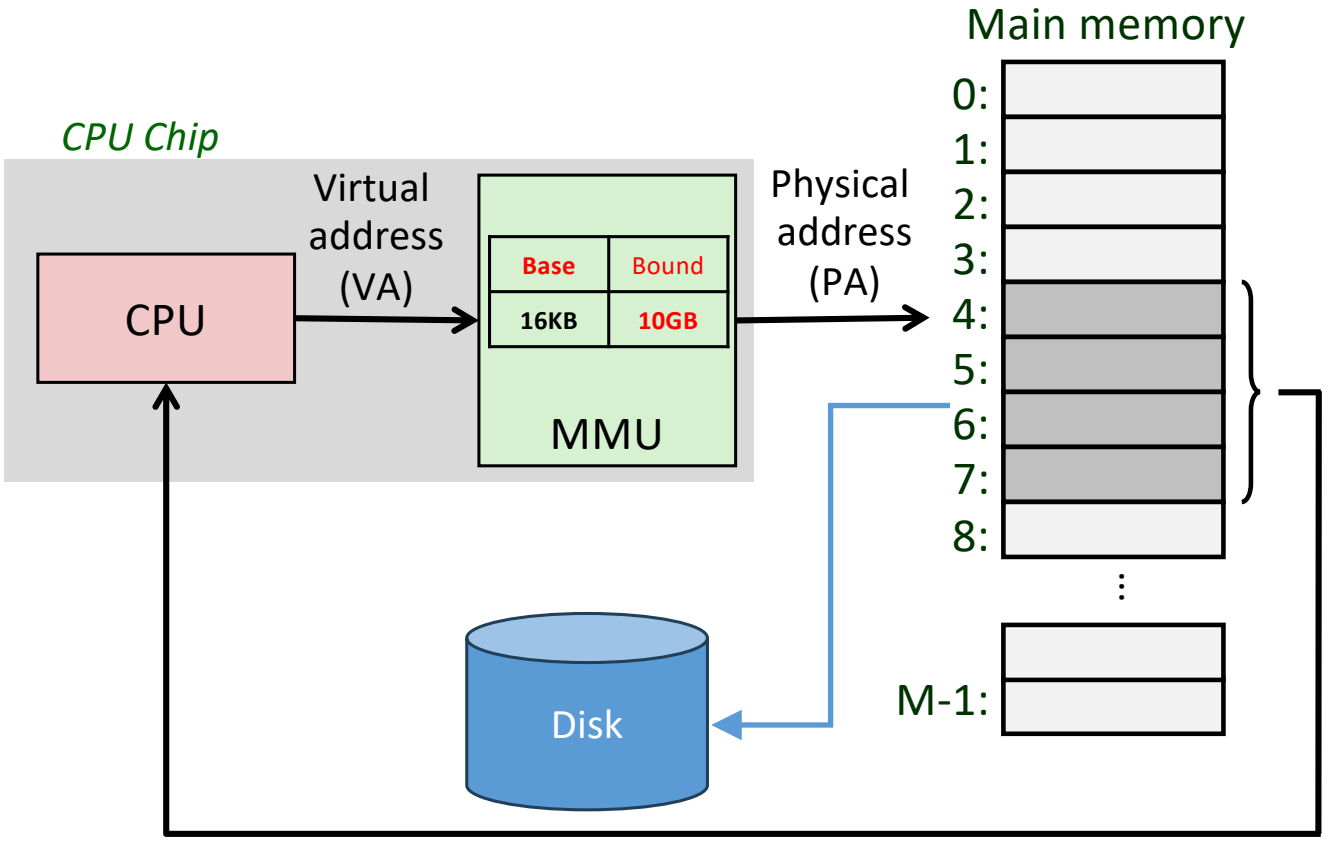
Virtual Address		Physical Address
0	→	16 KB
1 KB	→	17 KB
3000	→	19384
4400	→	*Fault (out of bounds)*

A System Using Base/Bound Registers

```
void scheduler() {
    while(true) {
        lock(process_table);
        foreach(Process p: scheduling_algorithm(process_table))
{
            if(p->state != READY) {
                continue;
            }
            p->state = RUNNING;
            unlock(process_table);
            swtch_mmu_bbRegisters(scheduler_process, p);
            swtch(scheduler_process, p);
            // p is done for now..
            lock(process_table);
        }
        unlock(process_table);
    }
}
```

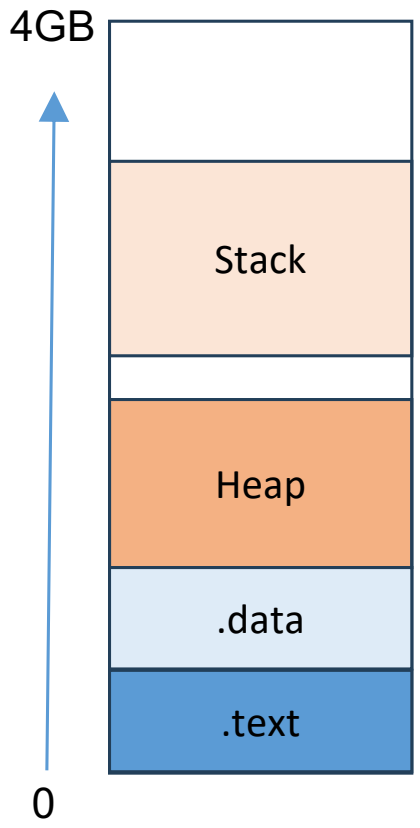
- swtch_mmu_bbRegisters (p1, p2)
  - Simply save the content of base/bound registers of currently executing P1 process into its PCB, and load the base/bound registers from P2's PCB

Reusing Same Physical Addresses



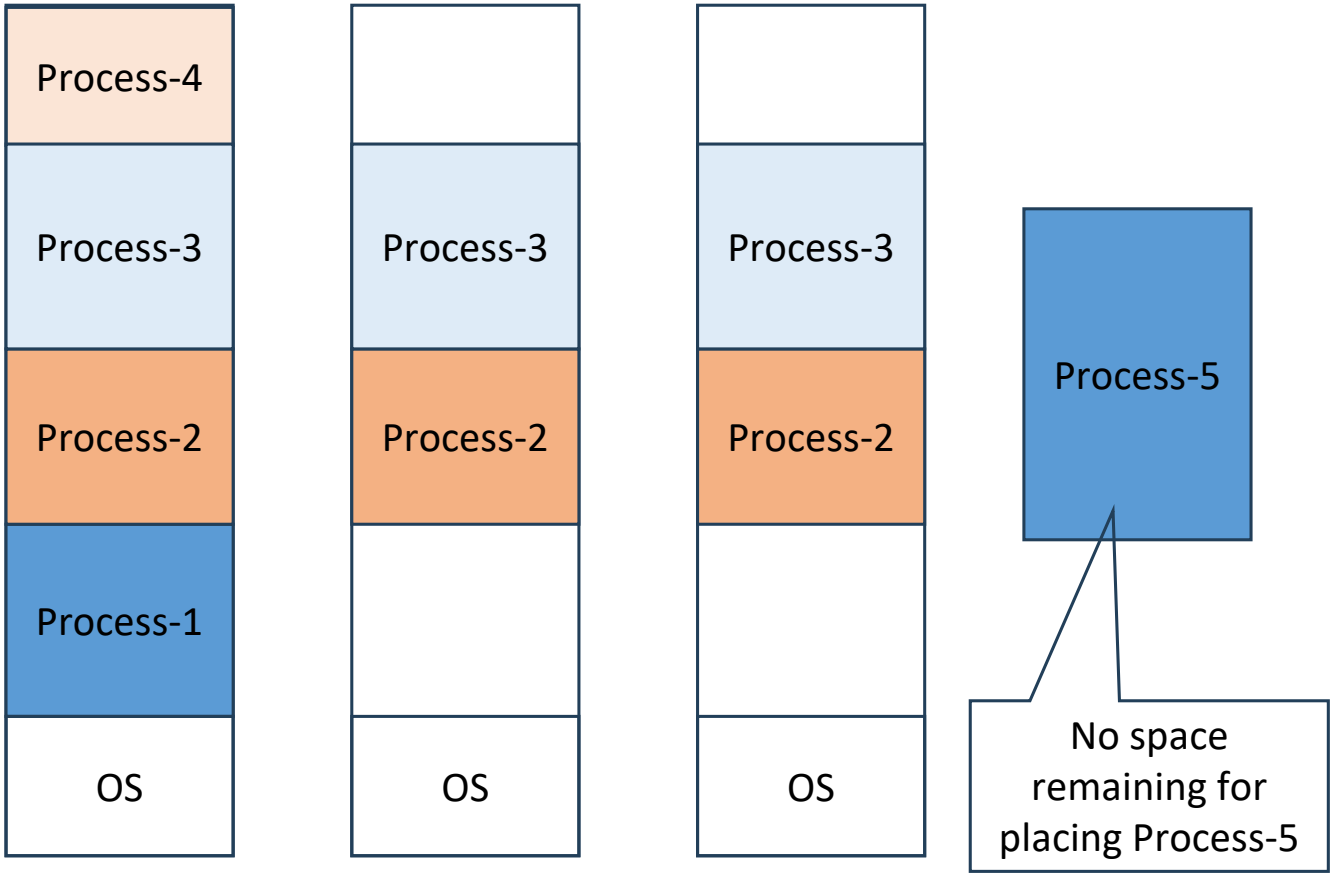
- Let us consider a situation where only one process P1 is running on a single-core processor and requires a huge address space that took away a major chunk of DRAM
- How to run another process P2 on this processor as it requires huge amount of address space?
  - Solution: Both could have overlapping physical address space
  - Swap out the physical address space of the running process to the disk during context switch
  - Huge overhead, but doable!

Fragmentation with Base/Bound (1/2)



- Large gap in address space between stack and heap
- Range of memory is unused, but still allocated to process
- Our approach-1 will lead to fragmentation
  - Internal or external fragmentation in this case?

Fragmentation with Base/Bound (2/2)



- There would also be situation that no contiguous free space is left for the address space of a new process
  - What type of fragmentation is this?
- Any other issues?
  - Can we create shared memory between processes?

Base/Bound: Summary

- Simple approach
  - Finding space for a new process's address space in memory
  - OS must *save and restore* the base-and-bounds pair at context switches
  - OS must provide exception handlers to be invoked if a process tries to access memory outside its bounds
  - OS will be able to move an address space of a stopped process from one location in memory to another
  - Reclaiming all of its memory at process termination
- Impractical to implement
  - Mandates OS to allocate contiguous chunk of physical address space to each process

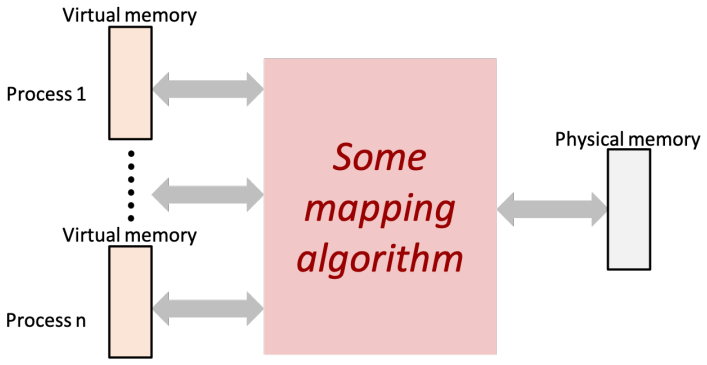
Next Lecture

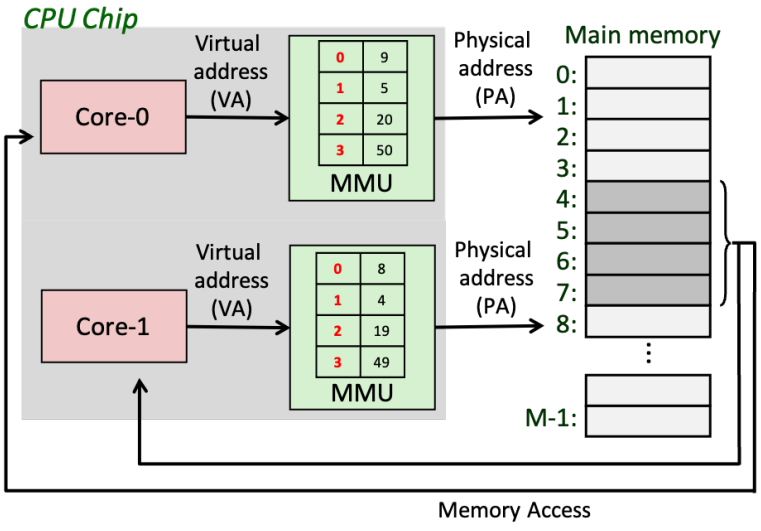
- Segmentation

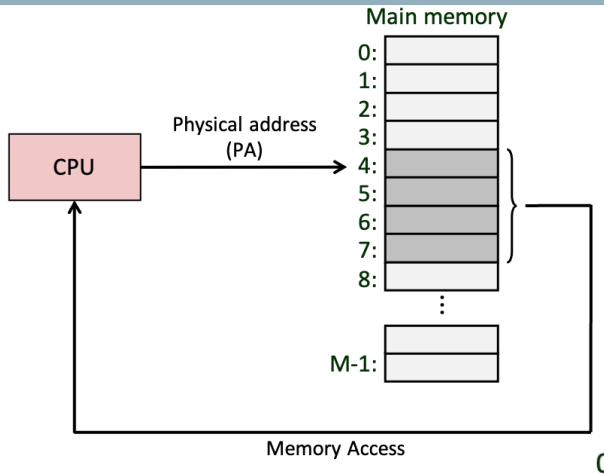
Lecture 16: Segmentation

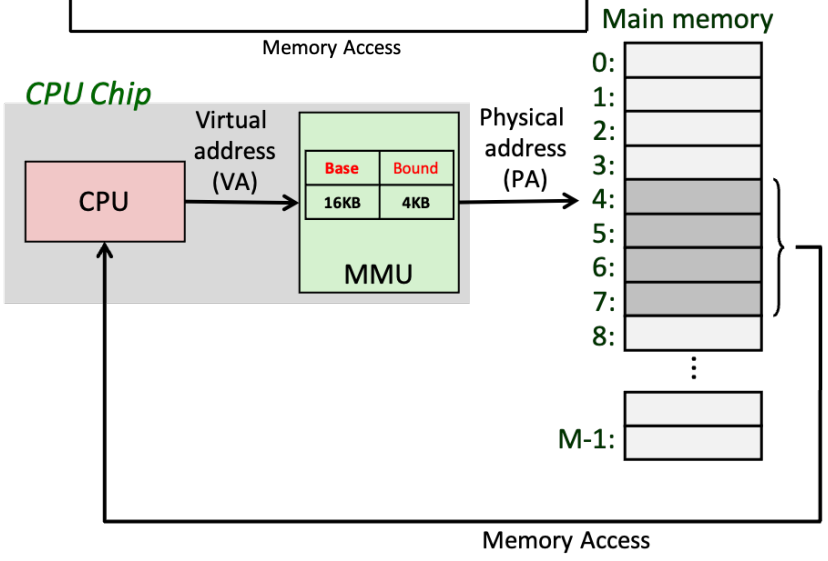
Vivek Kumar
Computer Science and Engineering
IIIT Delhi
vivekk@iiitd.ac.in

Last Lecture









Virtual Address		Physical Address
0	→	16 KB
1 KB	→	17 KB
3000	→	19384
4400	→	*Fault (out of bounds)*

- We need some kind of mapper between VM-PM as VM (virtual memory) too big than available PM (physical memory)
  - One-to-one mapping
  - Using MMU table
  - Dynamic relocation using base/bound registers
    - Prone to fragmentation

Today's Class

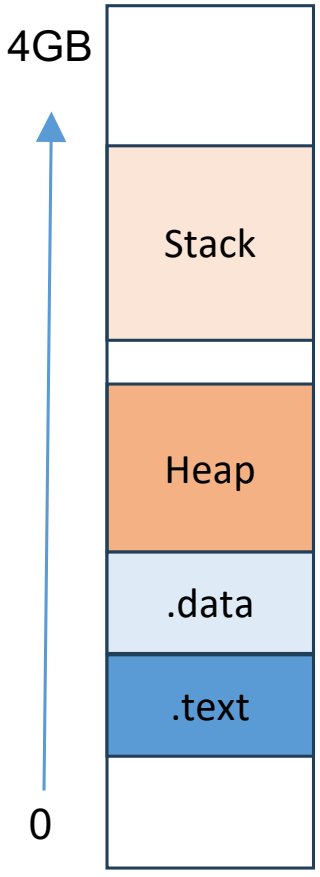
- Segmentation
- Segment registers
- Global Descriptor Table (GDT)

Loader View

```
vivek@possum:~/os23$ readelf -l fib

Elf file type is DYN (Position-Independent Executable file)
Entry point 0x1080
There are 11 program headers, starting at offset 52

Program Headers:
  Type           Offset   VirtAddr   PhysAddr   FileSiz MemSiz  Flg Align
  PHDR           0x000034 0x00000034 0x00000034 0x00160 0x00160 R   0x4
  INTERP         0x000194 0x00000194 0x00000194 0x00013 0x00013 R   0x1
      [Requesting program interpreter: /lib/ld-linux.so.2]
  LOAD           0x000000 0x00000000 0x00000000 0x003f8 0x003f8 R   0x1000
  LOAD           0x001000 0x00001000 0x00001000 0x00278 0x00278 R E 0x1000
  LOAD           0x002000 0x00002000 0x00002000 0x00134 0x00134 R   0x1000
  LOAD           0x002ed4 0x00003ed4 0x00003ed4 0x00134 0x0013c RW  0x1000
  DYNAMIC        0x002edc 0x00003edc 0x00003edc 0x000f8 0x000f8 RW  0x4
  NOTE           0x0001a8 0x000001a8 0x000001a8 0x00044 0x00044 R   0x4
  GNU_EH_FRAME   0x002020 0x00002020 0x00002020 0x0003c 0x0003c R   0x4
  GNU_STACK      0x000000 0x00000000 0x00000000 0x00000 0x00000 RW  0x10
  GNU_RELRO      0x002ed4 0x00003ed4 0x00003ed4 0x0012c 0x0012c R   0x1

 Section to Segment mapping:
  Segment Sections...
   00
   01     .interp
   02     .interp .note.gnu.build-id .note.ABI-tag .gnu.hash .dynsym .dynst
   03     .init .plt .plt.got .text .fini
   04     .rodata .eh_frame_hdr .eh_frame
   05     .init_array .fini_array .dynamic .got .data .bss
   06     .dynamic
   07     .note.gnu.build-id .note.ABI-tag
   08     .eh_frame_hdr
   09
   10     .init_array .fini_array .dynamic .got
```

- PHDR index 3
  - Starting virtual address is 4096 and total size 632
- PHDR index 4
  - Starting virtual address is 8192 and total size 308
- PHDR index 5
  - Starting virtual address is 16084
- Non-contiguous memory layout across segments (inter-segment)
- Contiguous memory within segments (intra-segment)

ELF Segment Allocations

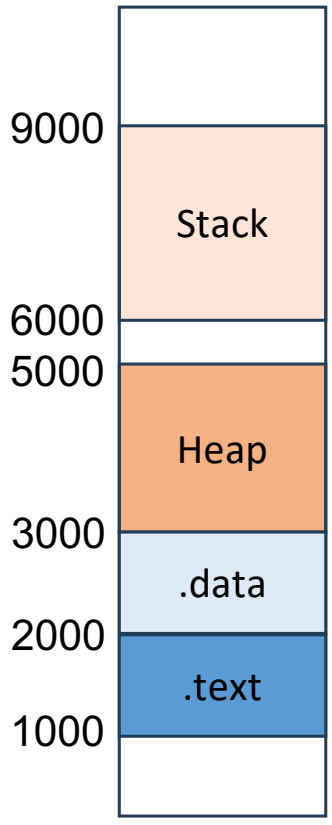


- Exact size for stack and heap size cannot be known upfront
  - Hello world program
    - Minimal sized stack and heap
  - Recursive Fibonacci program
    - Large stack but minimal sized heap
  - Recursive merge sort program
    - Large stack and large heap
- Observations
  - Each segment (stack, heap, data and text) has different memory requirements

Mapping Virtual to Physical Address

- Approaches
  1. Virtual address is same as the physical address
  2. Virtual address is not the same as physical address
     a) Using base & bound for VA to PA mapping
     b) Using segmentation for VA to PA mapping

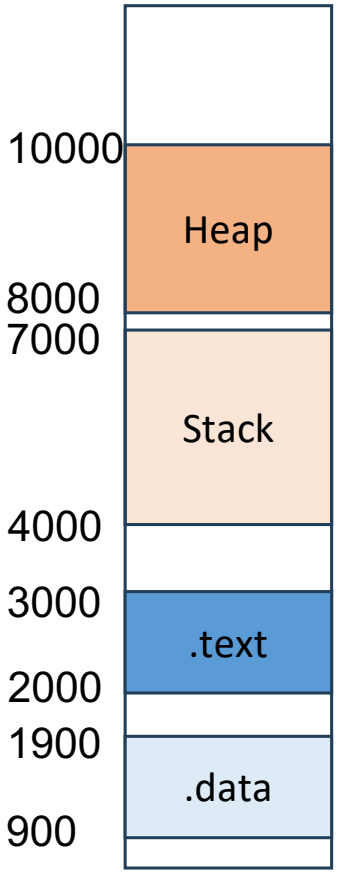
Approach-2: Segment Wise Base/Bound



- Each segment can have its own ("personal") base and bound values
  - **Segmentation**
- Each virtual address can have few bits to specify the segment they belong (e.g., 2 bits to specify 4 different segments), and remaining bits to specify the offset into that segment

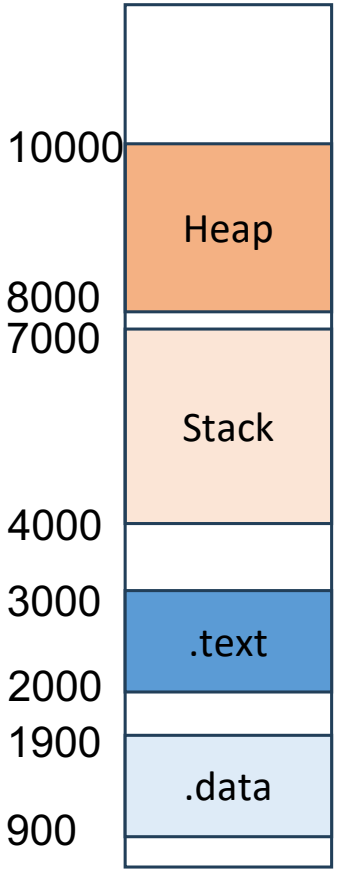
- Example: How to access virtual address 250 (binary: **11111010**)?
  1. Find the segment from the segment selector (e.g., data for first two bits)
  2. Find the offset (111010) = 58
  3. Find the **base** for this segment (Base=2000)
  4. Check if offset within **bounds** (Bound=1000)
  5. Physical address = 2058

Non-Contiguous Inter-Segment Memory



- As the base and bound is separate for each segment, now it is not a requirement to have a contiguous chunk of physical address space for the process's address space
  - Better space management and reduction in fragmentation
- Example: How to access virtual address 250 (binary: **11111010**)?
  1. Find the segment from the segment selector (e.g., data for first two bits)
  2. Find the offset (111010) = 58
  3. Find the **base** for this segment (Base=900)
  4. Check if offset within **bounds** (Bound=1000)
  5. Physical address = **958**

Accessing Segment's Base/Bound

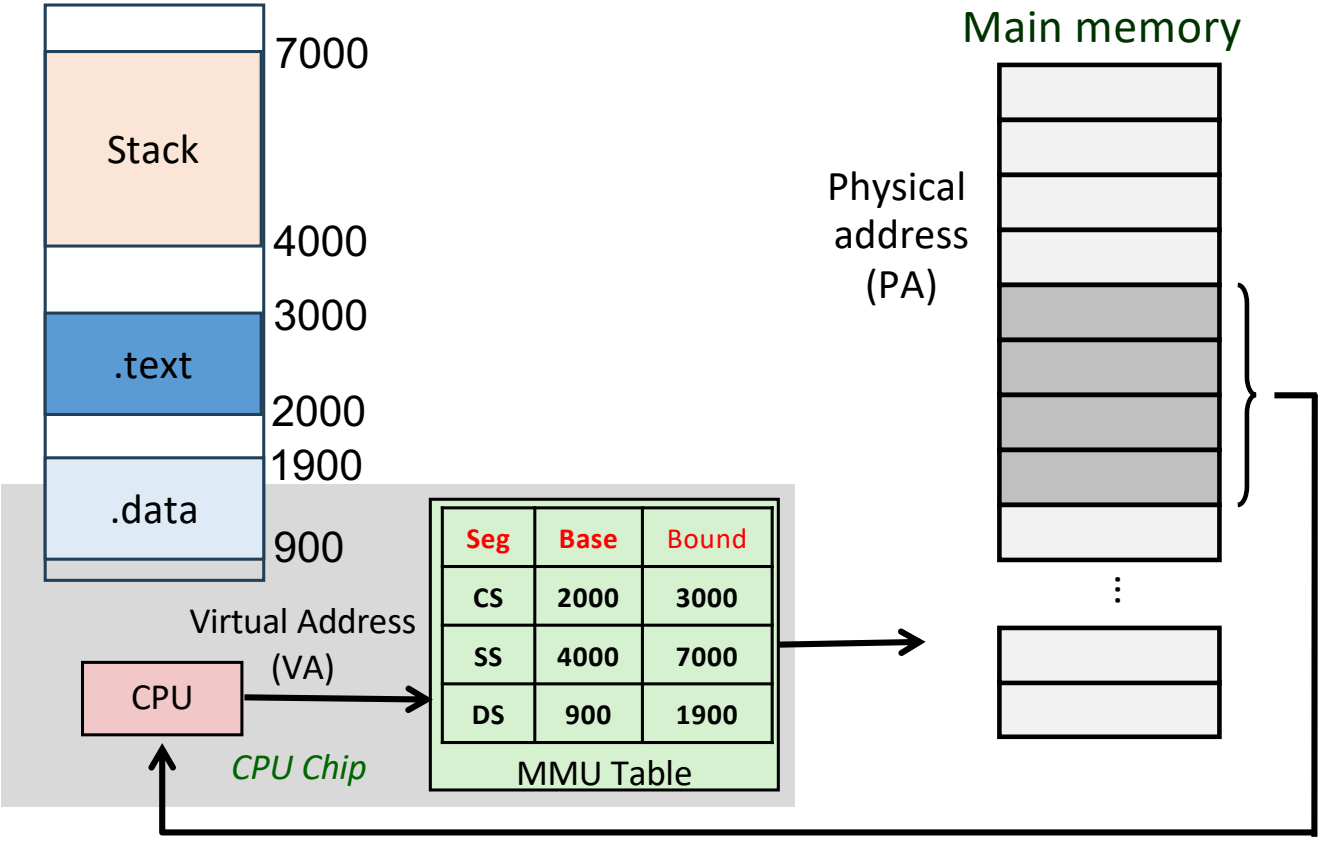


- How to access base and bounds of individual segments?
- Can we store the segments and their base/bound values in a MMU table?
  - Yes, but it would require **iterating** through the MMU table entries [O(n)]
  - Any simple technique?
    - Hardware support by providing dedicated registers on each core

Hardware Support for Segmentation

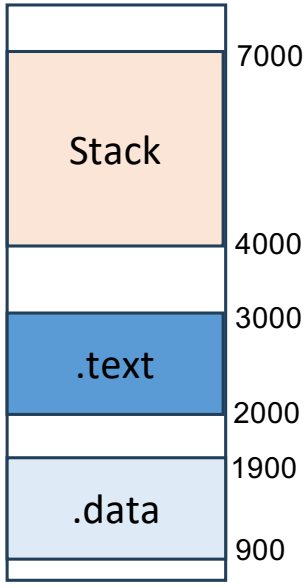
- x86 platform supports six segment registers per CPU
  - **CS (Code Segment)**
  - **SS (Stack Segment)**
  - **DS (Data Segment)**
  - ES, FG, and GS
    - General purpose
- Reduced address translation time as single CPU cycle required to access registers
- A program could have thousands of segments, but at any given time it can only use a maximum of 6 segment registers on x86

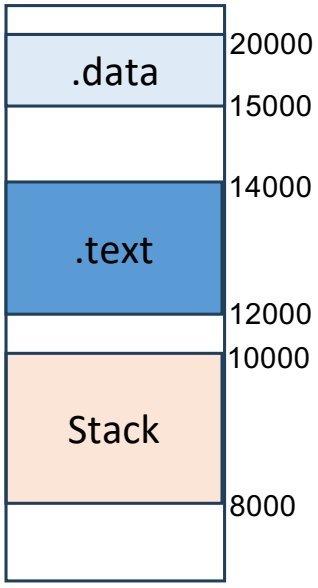
Segment Registers



- MMU table can now have entries for each segment registers and their corresponding base and bound values
- **Hardware provides special support for "MMU table"**

Global Descriptor Table (GDT)





CPU-0 | CS | SS | DS

Base	Bound	Type	Prv
2000	3000	RE	U
4000	7000	RW	U
900	1900	RW	U

GDT (CPU-1)

CPU-1 | CS | SS | DS

Base	Bound	Type	Prv
12000	14000	RE	U
8000	10000	RW	U
15000	20000	RW	U

GDT (CPU-2)

- Similar to IDT, there is also a GDT stored in memory (DRAM)
- One GDT for every CPU in the processor (x86)
- Each CPU has a dedicated register (GDTR) that holds the starting address of this GDT
  - Even IDTR is also per CPU on x86
- Each entries in GDT has a base-bound pair, privilege mode for this physical memory segment (kernel vs user space), memory access type (read/write/executable), etc.
- Still, too many operations for each virtual memory accesses
  1. Load segment selector
  2. Load GDT offset by looking into segment register
  3. Load GDTR and add above offset to get index
  4. Load values stored at index
- How to directly access segment's base/bound?
  - Add more hardware support!

Caching of Base/Bound Per Register

Visible Part	Hidden Part	
Segment Selector	Base Address, Limit, Access Information	CS
		SS
		DS
		ES
		FS
		GS

- Each segment register has a visible and a hidden part
  - Visible part is programable and is called as segment selector, which is an index into the GDT for that particular segment register
  - The hidden part is non-programable and it stores the base, bound (a.k.a limit) and access information for each segment register (cache)
    - No need to access GDT until a context switch is done (Why?)
    - No extra CPU cycles wasted for accessing GDT in VA to PA translation

A System Using Segmentation

```
void scheduler() {
    while(true) {
        lock(process_table);
        foreach(Process p: scheduling_algorithm(process_table)) {
            if(p->state != READY) {
                continue;
            }
            p->state = RUNNING;
            unlock(process_table);
            swtch_segment_registers(scheduler_process, p);
            swtch(scheduler_process, p);
            // p is done for now..
            lock(process_table);
        }
        unlock(process_table);
    }
}
```

- Segment selector (index into GDT) for all the segment registers are reloaded during context switch
- It also automatically updates the values stored inside the hidden part of each segment register (see previous slide)

Points to Ponder

- Can user process update GDTR to point to a new GDT?
  - No, it can only be done at the boot time by the OS running in privileged mode
- As GDT is stored in memory, can some rouge process can easily change the values stored inside the GDT?
  - No, GDT is stored in physical memory location that is accessible only to the OS
  - Updates to GDT entries are only carried out by OS in privileged mode (e.g., allocating process address space during exec)
- As GDT is stored in memory, can some rouge process access memory segments related to the OS code?
  - No, recall that each entry inside the GDT also has a privilege information stored
  - OS specific memory is stored in the kernel space whereas user application specific memory is stored in user space
- Can a user process update its segment register?
  - Yes, there are only 6 segment registers but a process could have thousands of segments, and it might want to switch to a different segment for which it will update its segment register (except CS, why?)

Segmentation in Linux

- Linux uses segmentation in a very limited form on modern x86 64-bit processors (but not on x86 32-bit processors)
  - Simplifies the memory management
  - Portability across non-x86 architectures
- Each segment register has exact same base and bound (effectively disabling segmentation)
  - Base is 0 and bound is 2^{32} - 1
  - Privilege is set as per kernel or user accessible segment
- How is then VA to PA mapping carried out?
  - Paging!

Segmentation Summary

- Advantages
  - Limited hardware support (GDTR, 6 segments, etc.)
  - Avoids internal segmentation (exact amount of memory allocated)
- Disadvantages
  - Difficult to increase the memory size once allocated if there is no contiguous chunk of unused physical memory of the new size
  - Favors external fragmentation due to allocation of variable sized memory chunk for each segments

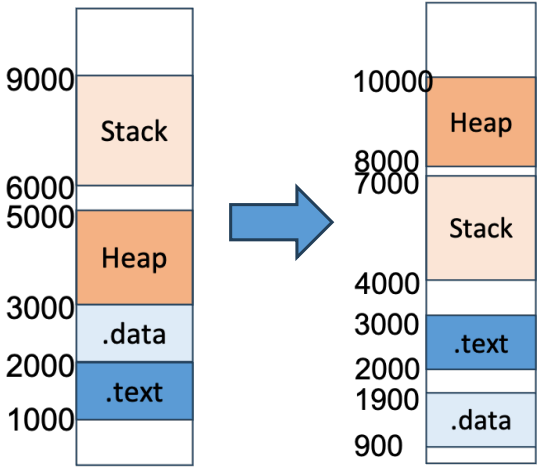
Next Lecture

- Introduction to paging

Lecture 17: Introduction to Paging

Vivek Kumar
Computer Science and Engineering
IIIT Delhi
vivekk@iiitd.ac.in

Last Lecture



Visible Part	Hidden Part	
GDT index	Base Address, Limit, Access Information	CS
		SS
		DS
		ES
		FS
		GS

Segment Selector	Offset

- Segment wise base/bound enables non-contiguous inter-segment memory
- VA has segment selector bits to identify the segment and an offset into that segment
- Base and bound for each segment stored in GDT
  - Address of GDT stored in GDTR
- x86 platform supports six segment registers per CPU
  - VA segment selector helps in choosing segment register
  - Each segment register contains the offset into GDT
  - Index into GDT is sum of GDT base address inside GDTR and above offset
  - Offset inside VA added to the segment's base address obtained from GDT
  - Base/bound is cached inside segment register's hidden part
- Segment registers must be saved/restored at context switch

Base	Bound	Type	Prv
2000	3000	RE	U
4000	7000	RW	U
900	1900	RW	U

Global Descriptor Table (GDT)

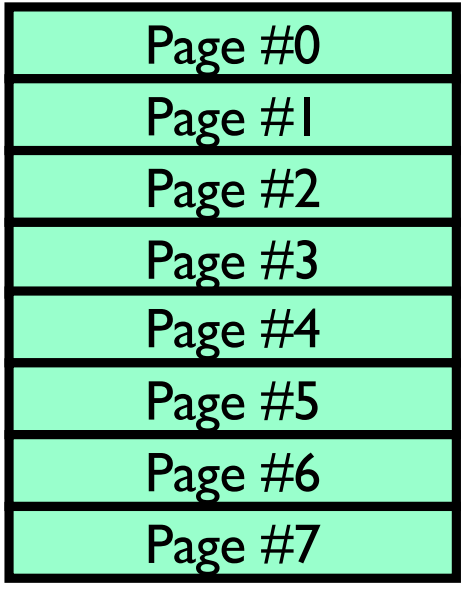
Today's Class

- Introduction to paging

Mapping Virtual to Physical Address

- Approaches
  1. Virtual address is same as the physical address
  2. Virtual address is not the same as physical address
     a) Using base & bound for VA to PA mapping
     b) Using segmentation for VA to PA mapping
     c) Paging
        a) Using Page Table (PT) in RAM and having a base register for PT

Paging



Physical Memory

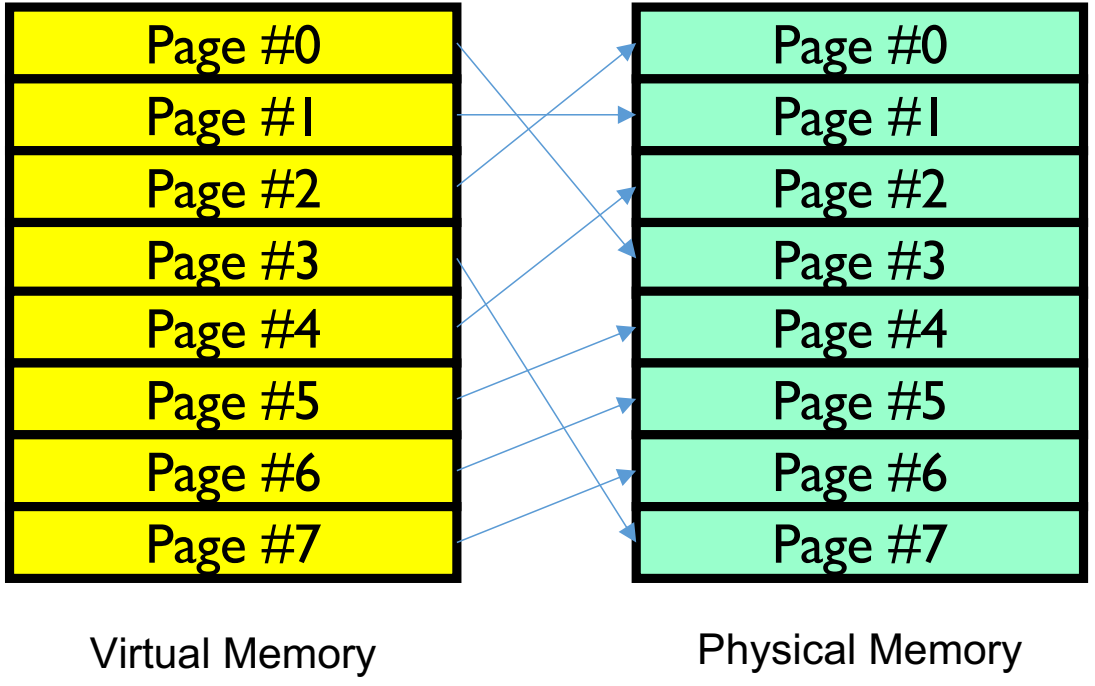
- Segmentation is easy to implement but it has major drawbacks because segments could have variable sized memory allocation
  - Can lead to **external** fragmentation
  - To allocate space for a new process, segments of some other process has to be swapped to the disk (huge overhead)
- Paging solves both the above issues by dividing the available memory into small and fixed size blocks (pages)
  - The memory can then be viewed as an array of fixed sized slots
  - For example, with a page size of 8 bytes, we can have an array of 8 pages on a DRAM whose size is 64 bytes

Total Number of Pages

- Default page size on Linux is 4096 bytes (4KB)
- Pages on physical memory depends on DRAM size
  - Assume DRAM size is 64GB, then total number of pages in **physical memory** will be 64GB/4KB, i.e., 16 x 10^6 pages
- Pages in virtual memory is fixed as per the addressing mode
  - Total number of pages in **32-bit** addressing mode is 2^{32}/4KB, i.e., 2^{20}
  - Total number of pages in **64-bit** addressing mode is 2^{64}/4KB, i.e., 2^{52}

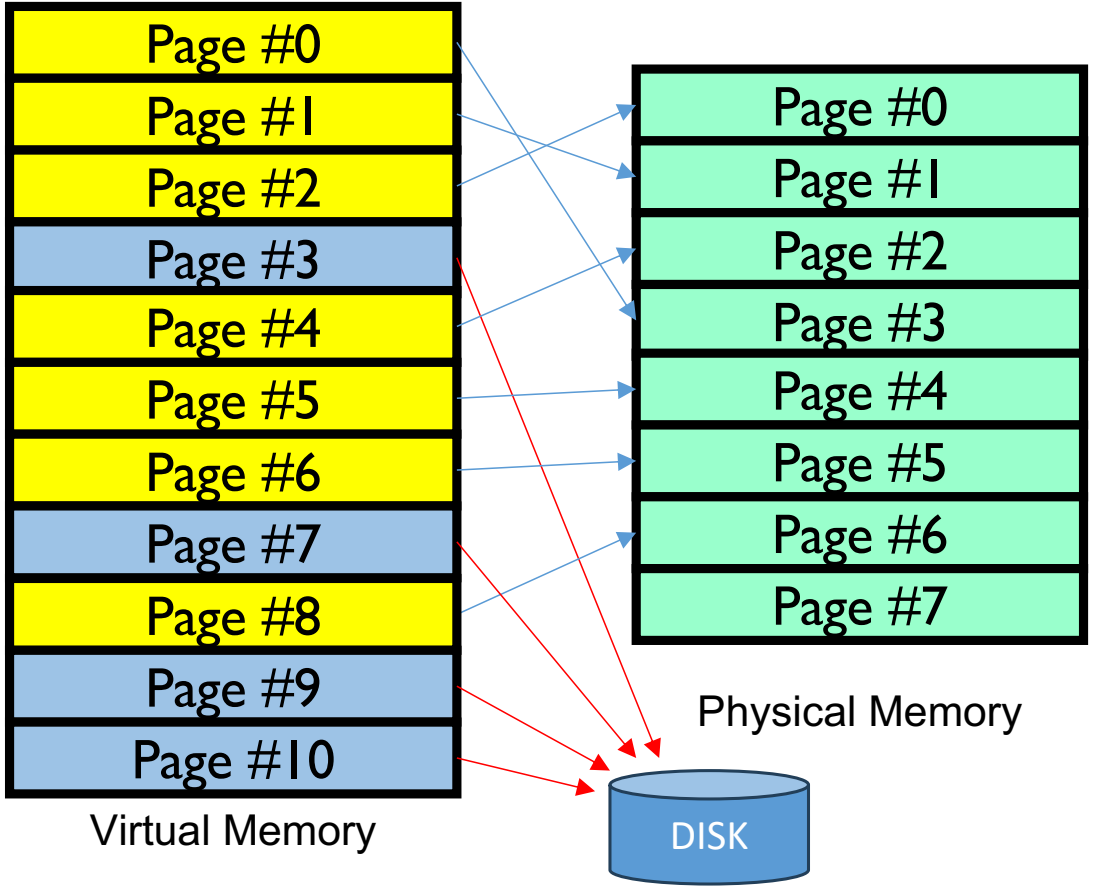
If you think of a 32-bit address space as the size of a tennis court, a 64-bit address space is about the size of Europe(!) -- OSTEP Book

Paging



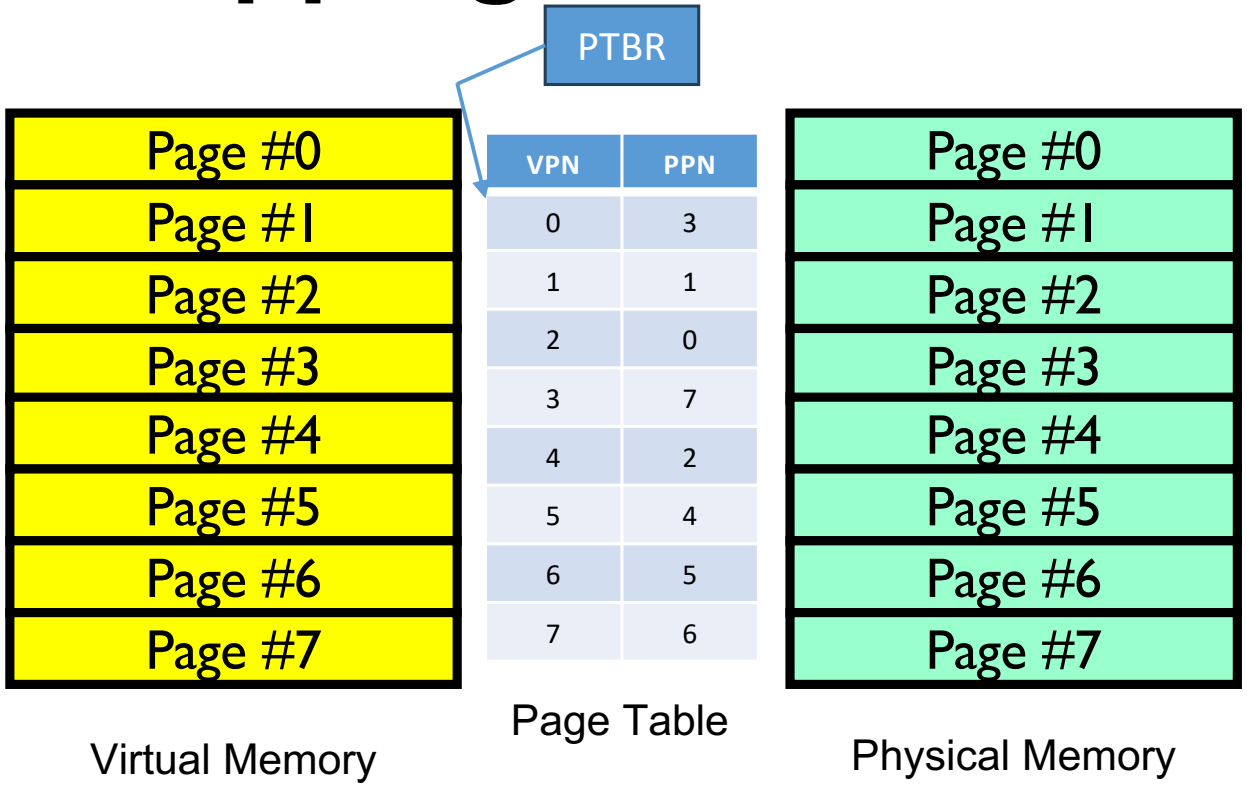
- Process's segments would still be of variable size, but now instead of a contiguous chunk of memory, they could have **non-contiguous** mapping with physical pages (both inter/intra-segment)
- How about internal fragmentation?
  - Yes, if a segment requires 5000 bytes of memory, it would have 2 page allocations where 4092 bytes would forever remain free in the second page

Paging



- Swapping out to disk can still happen, but where would the overheads will be minimal?
  - Segmentation Or paging?
- Lesser overheads in paging as compared to segmentation
  - Instead of swapping out an entire segment, only unused pages of program segment(s) will be swapped out
- Issue still remaining
  - How to map Virtual Page Number (VPN) to Physical Page Number (PPN)?

Mapping VPN to PPN



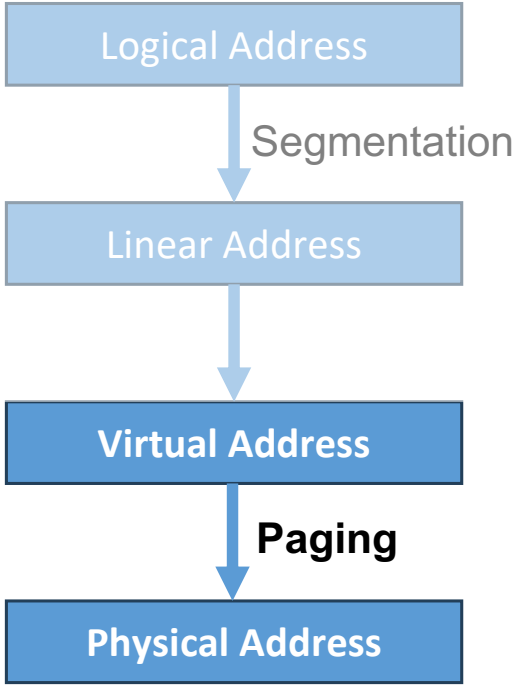
- Mapping of VPN to PPN is carried out using a data structure called as Page Table
- It is stored in DRAM
- Each CPU has a Page Table Base Register (PTBR) that holds the base address and Page Table Length Register (PTLR) that holds the size of the page table

A System Using Paging

```
void scheduler() {
    while(true) {
        lock(process_table);
        foreach(Process p: scheduling_algorithm(process_table)) {
            if(p->state != READY) {
                continue;
            }
            p->state = RUNNING;
            unlock(process_table);
            swtch_pagetable_base_registers(scheduler_process, p);
            swtch(scheduler_process, p);
            // p is done for now..
            lock(process_table);
        }
        unlock(process_table);
    }
}
```

- Page table resides in memory
- Process's PCB has a field to store PTBR
- PTBR of a running process is saved during context switch

Segmentation Along with Paging

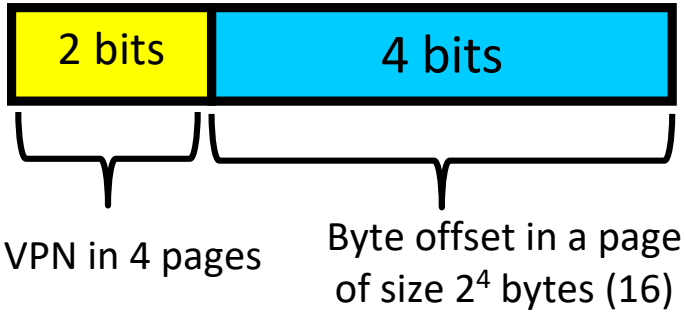


- Segmentation cannot be disabled in x86 processors, but address translation is complex using the segmentation-based approach
  - Hence, to simplify the address translation, segment registers are set to have the same base (0) and bound (2^{32}-1) values (Lecture #16)
- The address visible to a programmer is actually called as Logical address, as it has an associated segment selector (Lecture #16)
  - Segmentation converts this logical address to linear address, which is then converted into virtual address to support paging
- However, as the base and bound of each segment registers maps to entire 2^{32} memory size (4GB), logical address will be the same as virtual address
- Hence, we will call the address visible to the programmer as a virtual address in all the remaining slides/lectures

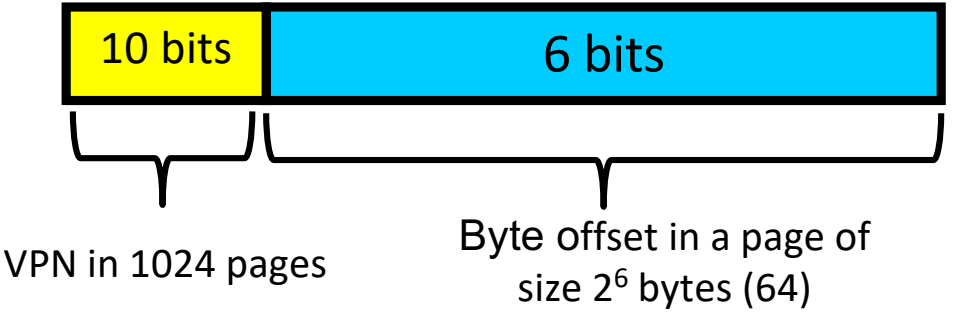
Encoding VPN inside Virtual Address

- How many VAs can reside in a 4KB page size?
  - 4096 bytes/4 bytes = 1024 virtual addresses
- How to find the VPN for a given VA?
  - Encode (prefix) the VPN inside each VA (**next slide**)
- How many bits required to represent a VPN for pages of N bytes in a virtual memory of M bytes (total M/N pages)?
  - M=64 (2^6) and N=16
    - Pages = 4 and bits required to identify 4 pages are 2 (2^2=4)
  - M=4GB (2^{32}) and N=4KB
    - Pages = $2^{32}/4096 = 2^{20}$ and bits required to identify 2^{20} pages are 20

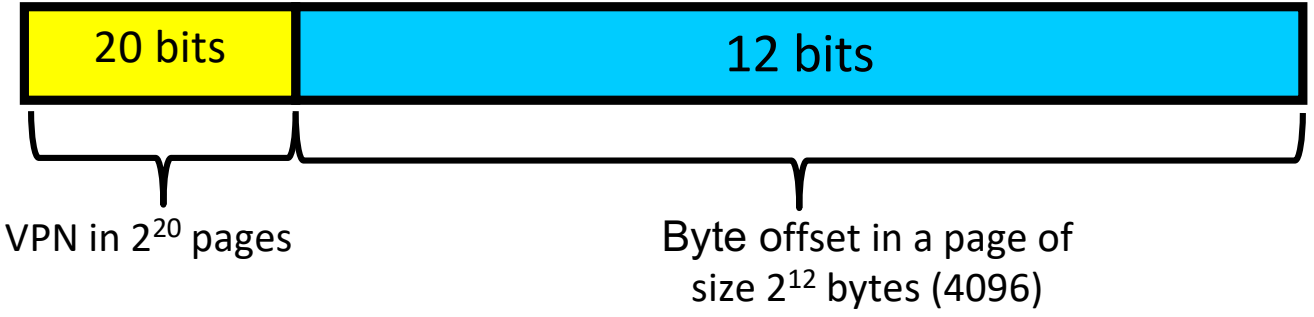
Two Components of a Virtual Address



6-bit addressing with page size of 16 bytes

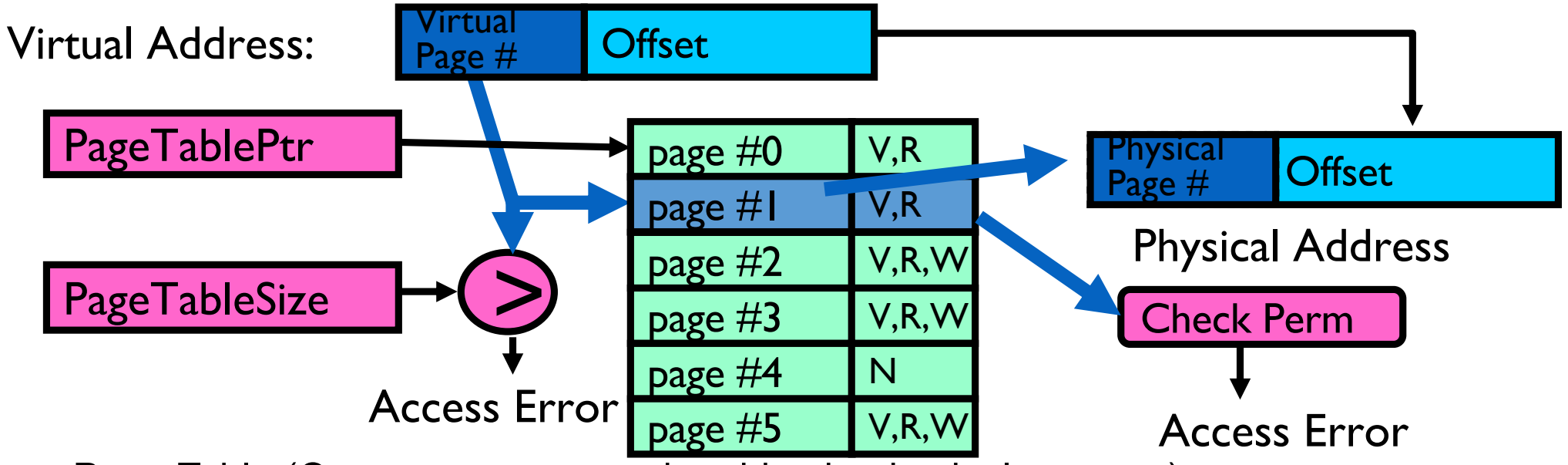


16-bit addressing with page size of 64 bytes



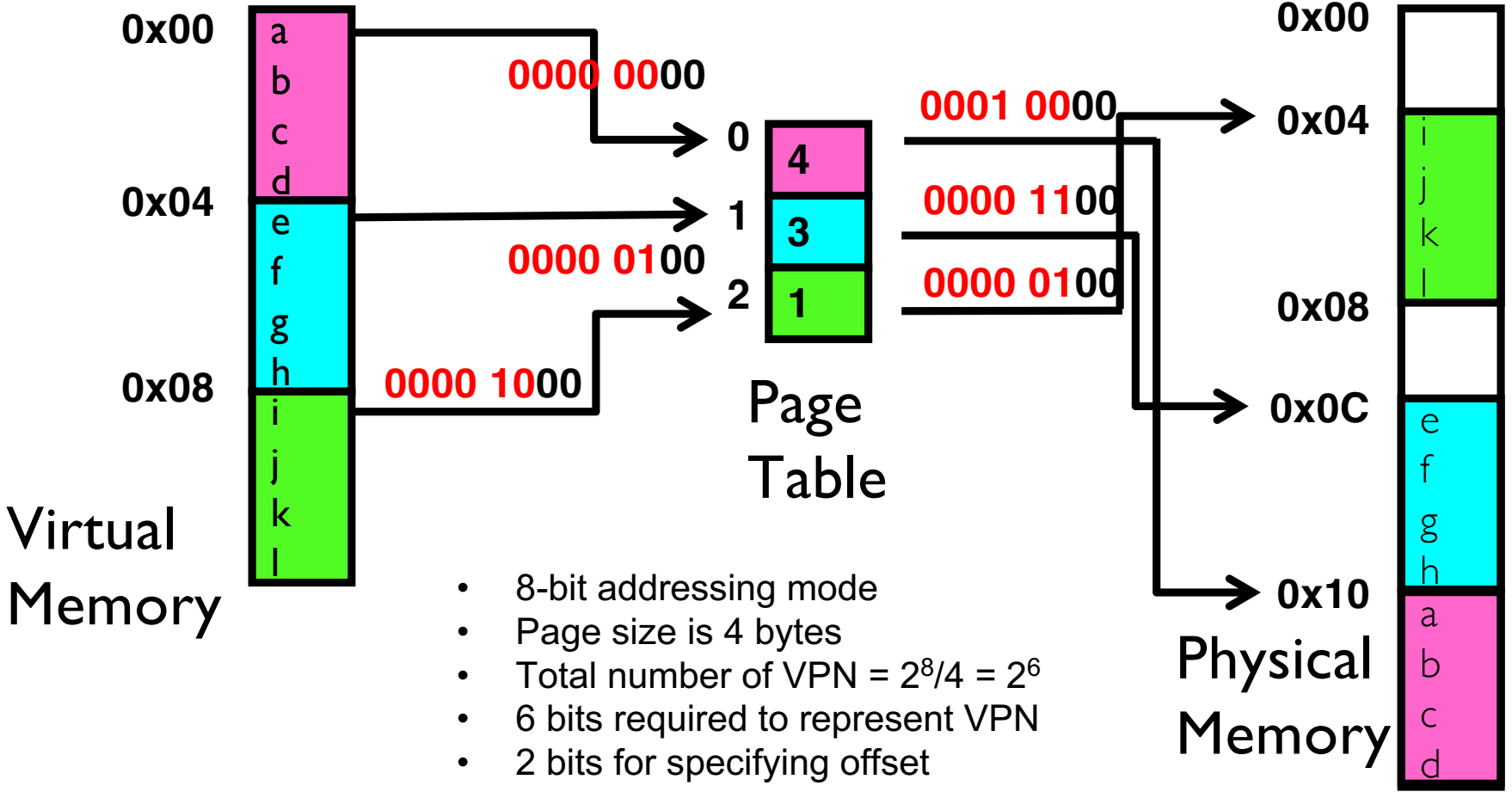
32-bit addressing with page size of 4KB

How to Implement Simple Paging?



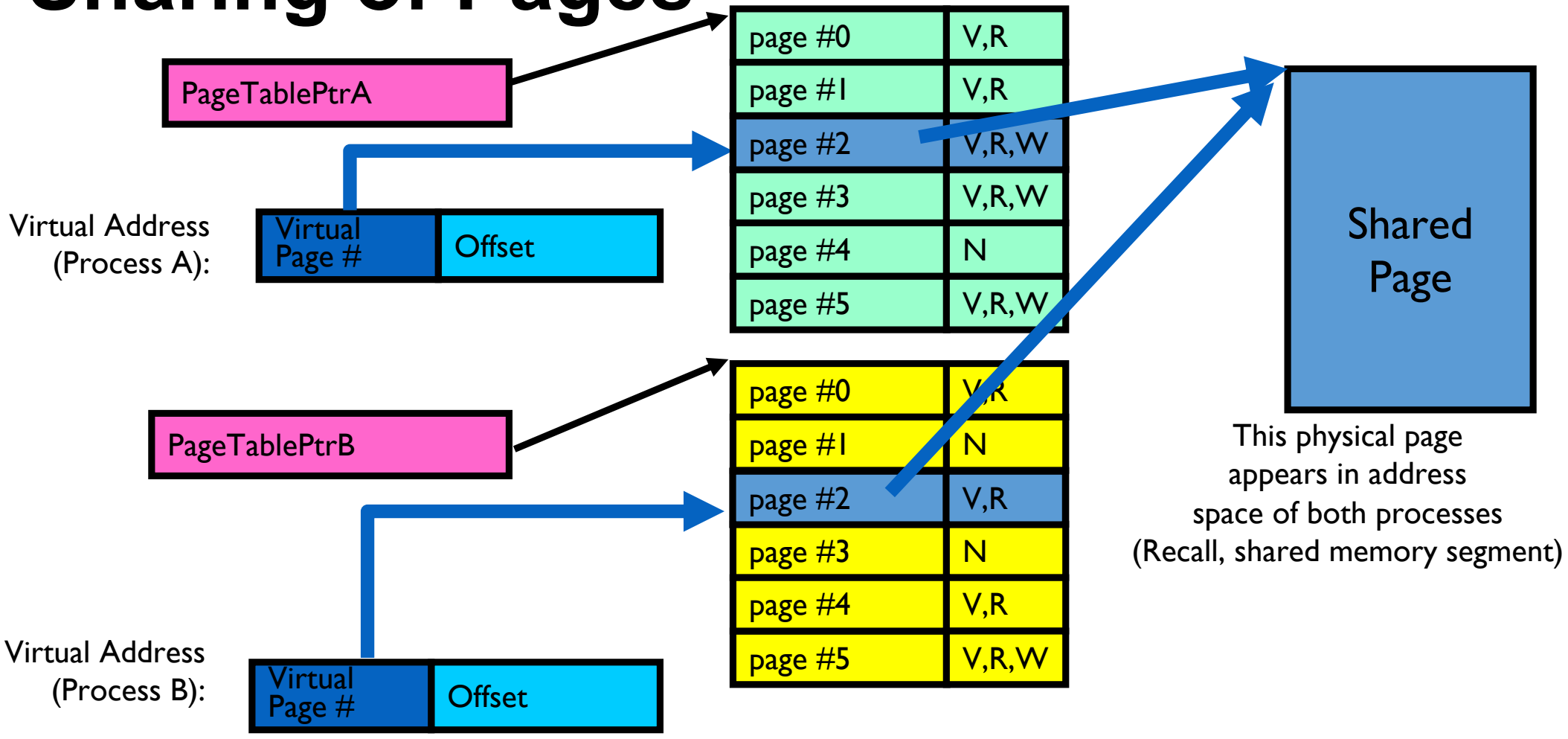
- Page Table (One per process and resides in physical memory)
  - Contains physical page and permission for each virtual page (e.g. **V**alid bits, **R**ead, **W**rite, etc)
- Virtual address mapping
  - Offset from Virtual address copied to Physical Address
  - Virtual page # is all remaining bits (Physical page # copied from table into physical address)
  - Check Page Table bounds and permissions

A Simple Page Table in Action



- 8-bit addressing mode
- Page size is 4 bytes
- Total number of VPN = $2^8/4 = 2^6$
- 6 bits required to represent VPN
- 2 bits for specifying offset

Sharing of Pages



Physical Page Allocation in Linux

```
char * array = (char*) malloc(sizeof(char)*1024*1024*1024);
```

```
  0 |                          1.3%      4                          0.0%
  1 ||||||||                  24.2%      5                          0.0%
  2                            0.0%      6 |                        0.6%
  3 ||                         2.6%      7 |                        0.6%
Mem ||||||||||||||||      963M/7.67G    Tasks: 183, 694 thr; 1 running
Swp |||||||               429M/2.00G    Load average:      0.59 0.60
```

- The above program allocates 1GB memory using malloc, but output of **htop** command doesn't show 1GB memory usage!
  - I did not see any change to that memory even after the above program finished, which implies its default memory usage in my system
    - What is happening?

Physical Page Allocation in Linux

```
char * array = (char*) malloc(sizeof(char)*1024*1024*1024);
memset(array, 0x00, sizeof(char)*1024*1024*1024);
```

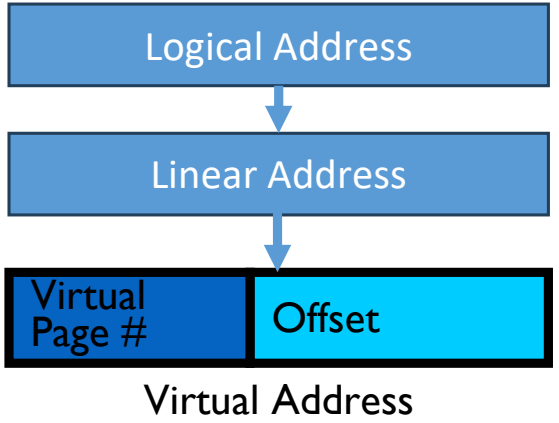
```
  0 |                          0.7%      4                                0.0%
  1 |                          0.7%      5                                0.0%
  2 |||||||||||               33.3%      6 |                              0.7%
  3 ||                         2.0%      7                                0.0%
Mem |||||||||||||||||||| 1.94G/7.67G    Tasks: 184, 694 thr; 1 running
Swp |||||||              429M/2.00G    Load average:      0.55 0.59
```

- Just writing to that allocated space changes the output of **htop** command to show the 1.94GB memory usage (1GB from the program and 0.94GB of default memory)
  - What just happened now?

Physical Page Allocation in Linux

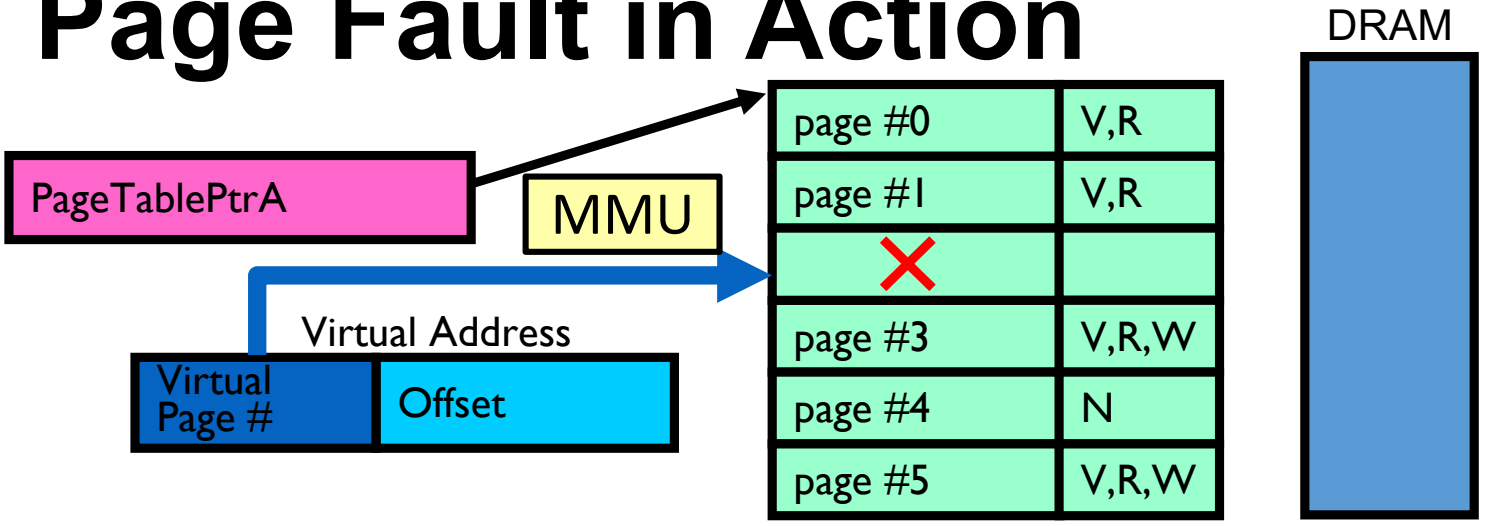
- Linux uses lazy memory allocation policy to save memory space (RAM is precious!)
  - Not just for malloc, but even while loading program segments
  - Physical memory for the segments are not allocated until accessed
    - E.g., if a program has a .data segment, but the user doesn't accesses the global variables (residing in .data segment) then it doesn't make sense to allocate memory for .data segment upfront (load time)
- So, how the physical memory is allocated?
  - During the program execution, when the program attempts to **write** to that memory
    - Linux uses a mechanism known as **Page Fault** for lazy page allocation

Page Fault in Action



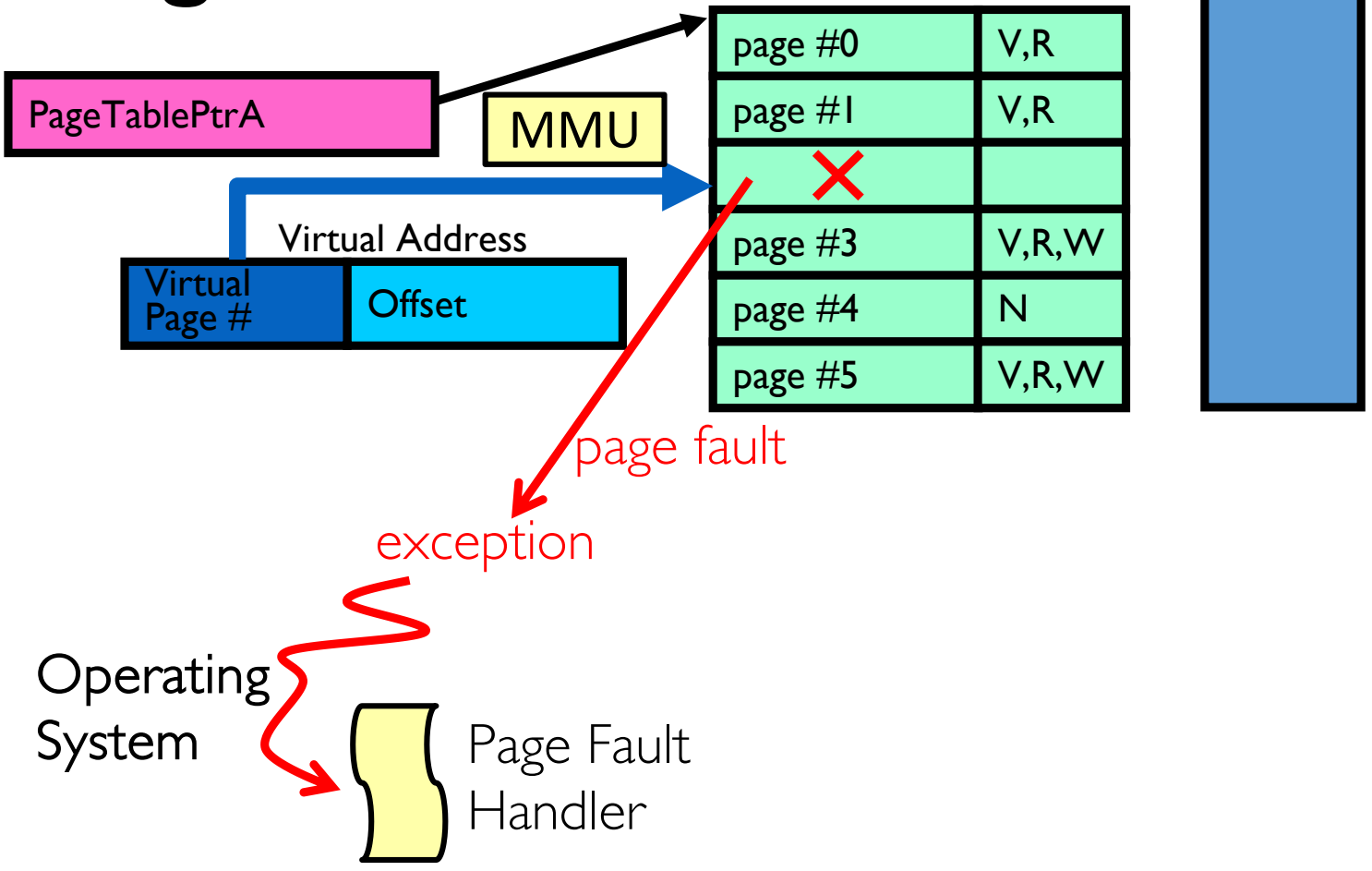
- Program attempts to access an address
- Segmentation converts this logical address to linear address, which is then converted into virtual address to support paging

Page Fault in Action



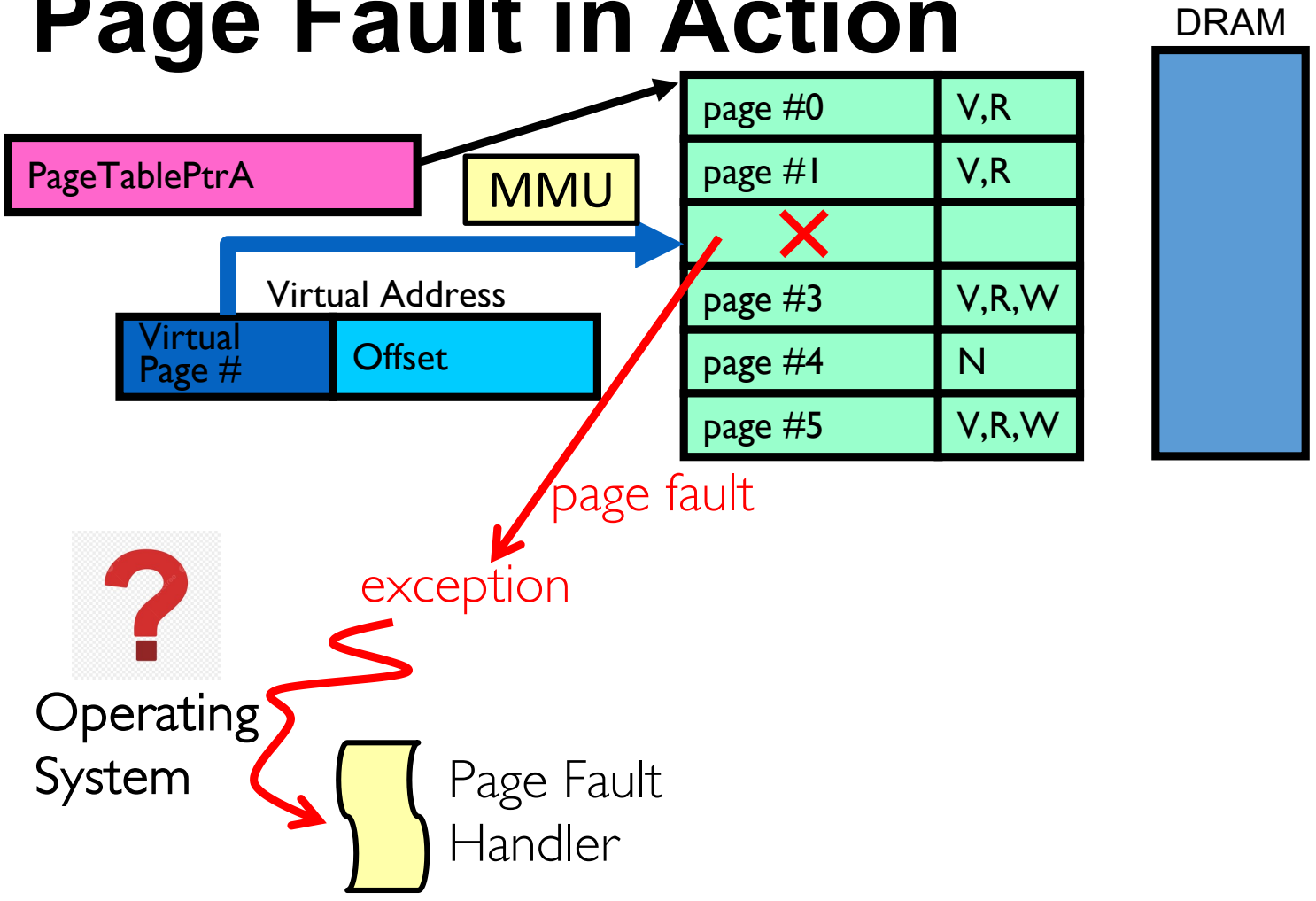
- This address is valid (i.e., it points to a valid heap allocation)
- However, the page table doesn't have any entry for this VA to PA mapping

Page Fault in Action



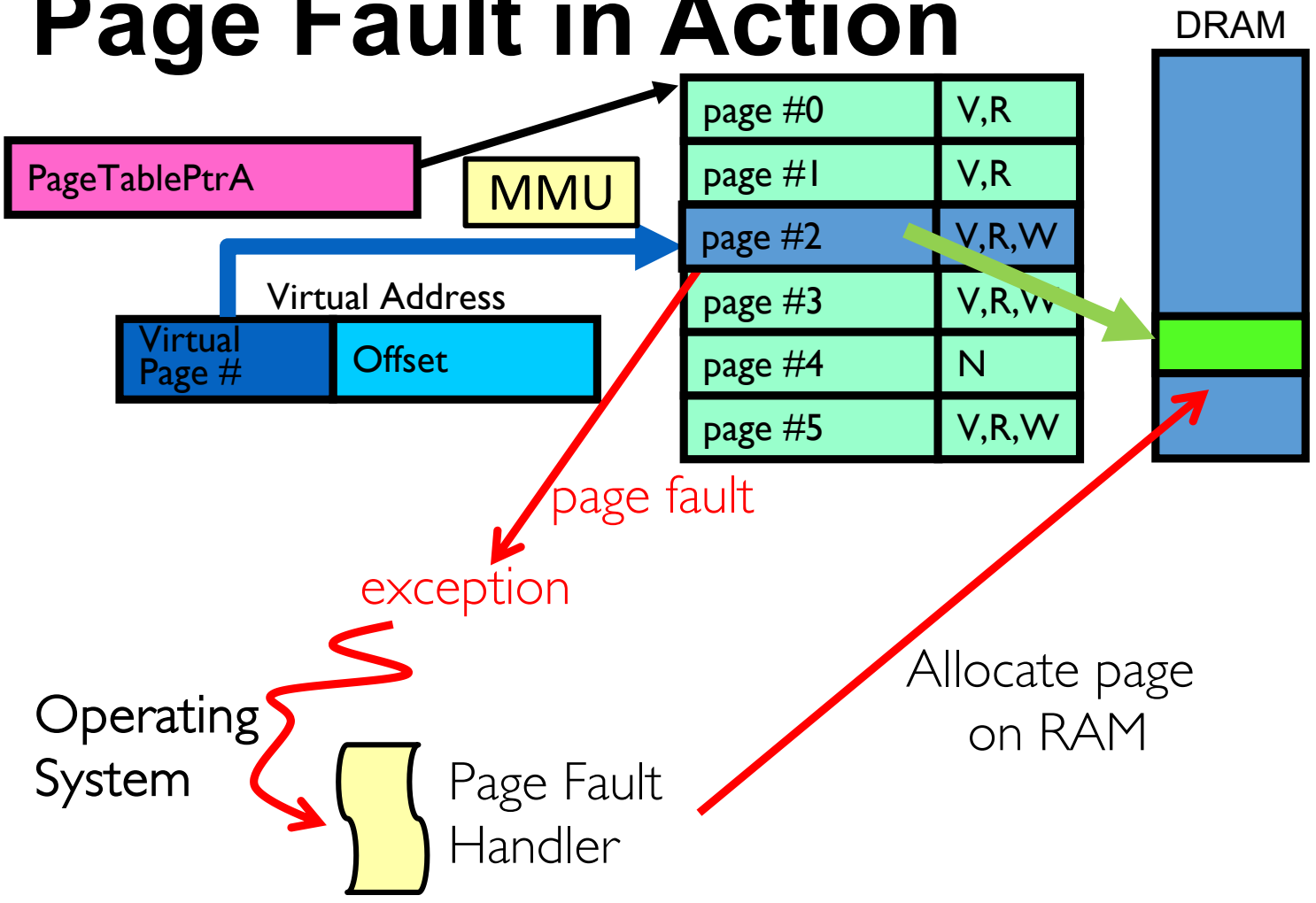
- Program state is saved and Interrupt 14 is generated by the MMU indicating page fault exception
  - What if the program only attempted to read the VA?
    - No! Page fault will be generated only upon a write access
      - Only in case of dynamic memory allocation

Page Fault in Action



- OS will decide what action is to be taken
  - It could be that the page was allocated but swapped out to disk (i.e., demand paging, we will see in future lecture)
- OS finds out that no allocation was ever made

Page Fault in Action



- OS will allocate page(s) on DRAM
- Page table entry is updated
- Program execution (state) is restored from the same location that caused the page fault

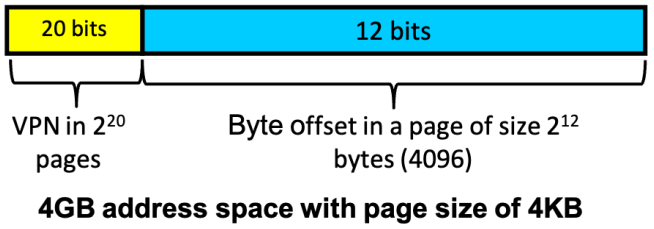
Next Lecture

- Translation Lookaside Buffer (TLB)
- Quiz-3 during next lecture
  - Syllabus: Lectures 14-17
- Assignment-4 will be released on 25th with 9-days of deadline
  - Extension of assignment-1
  - Weightage of 6% and one week deadline

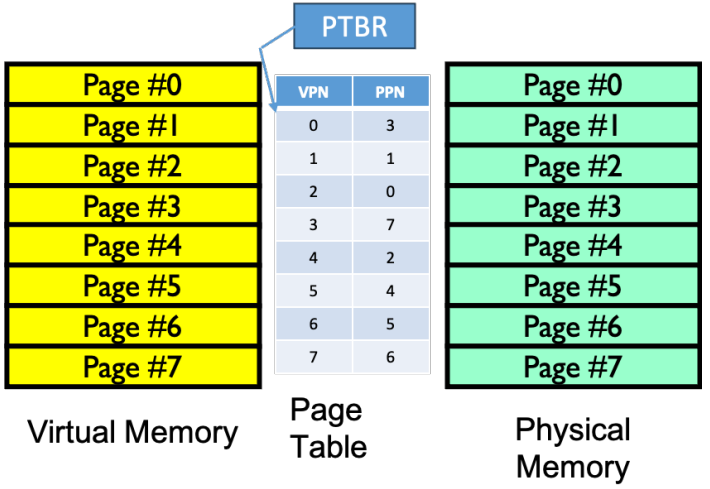
Lecture 18: Translation Lookaside Buffer (TLB)

Vivek Kumar
Computer Science and Engineering
IIIT Delhi
vivekk@iiitd.ac.in

Last Lecture

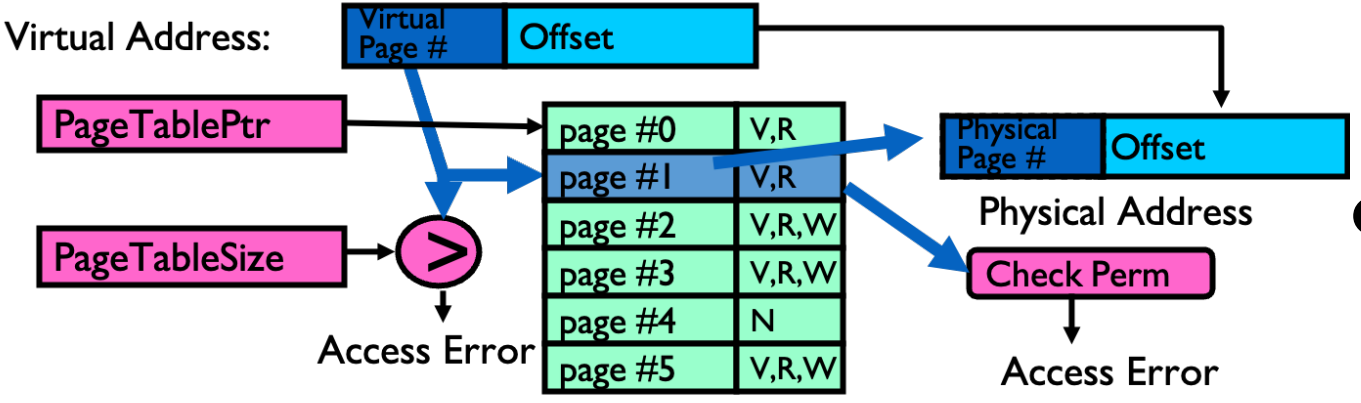


4GB address space with page size of 4KB



- Paging divides the available memory into small and fixed size blocks (4KB in Linux)
- Each VA or PA belongs to a VP or PP, respectively, and the corresponding numbers are VPN and PPN, respectively
- Linux uses a mechanism known as **Page Fault** for lazy page allocation

How to Implement Simple Paging?



Today's Class

- TLB
- Locality
- Quiz-3

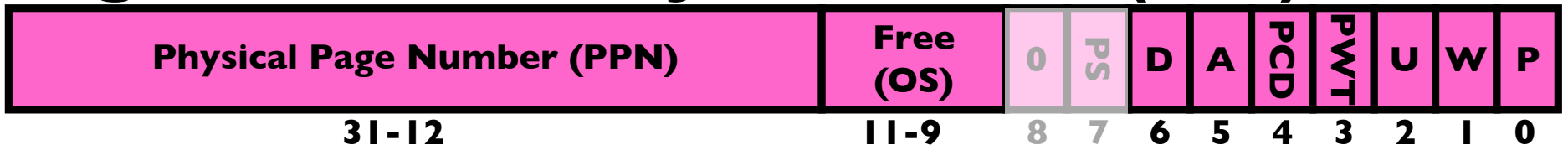
Mapping Virtual to Physical Address

- Approaches
  1. Virtual address is same as the physical address
  2. Virtual address is not the same as physical address
     a) Using base & bound for VA to PA mapping
     b) Using segmentation for VA to PA mapping
     c) Paging
        a) Using Page Table (PT) in RAM and having a base register for PT
        b) Using Page Table (PT) in RAM and having a base register for PT as well as a TLB register

Page Table Entry in 32-bit (1/2)

- Recall, a page table is a mapping between VPN to PPN
- What kind of data structures we can use for implementing a page table?
  - Arrays?
    - VPN will be the index in an array of PPN
  - Hash table?
    - VPN will be the key with the value as PPN
- Although we have been discussing PPN just as a number, it contains much more information than just the page number

Page Table Entry in 32-bit (2/2)



- **P:** Present Bit (set to 1 for 4KB page)
- **W:** Is this page writable**?**
- **U:** Usermode v/s kernel mode access to this page?
- ***PWT:*** *Related to caching of the associated page*
- ***PCD:*** *Related to caching of the associated page*
- **A:** Set by hardware when page is **first** accessed
- **D:** Page marked **dirty** whenever modified by application
- Bits 7 & 8: *Related to caching of the associated page*
- **PPN: Physical** Page Number (or frame number) associated with this PTE entry

Page Table Entry (PTE) in x86 for 4KB pages

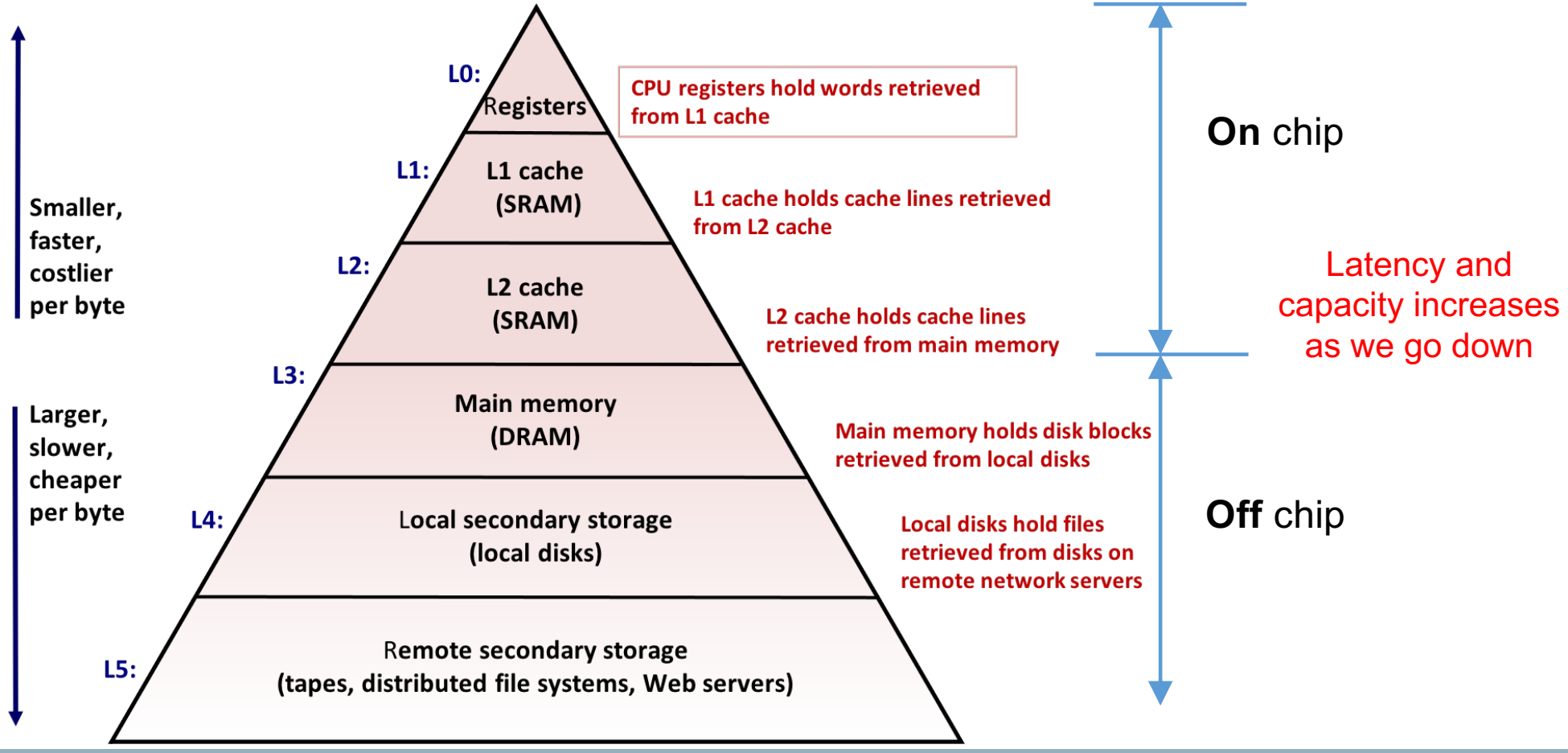
Overheads with the Page Table

Physical Page Number (PPN)	Free (OS)	0	PS	D	A	PCD	PWT	U	W	P
31-12	11-9	8	7	6	5	4	3	2	1	0

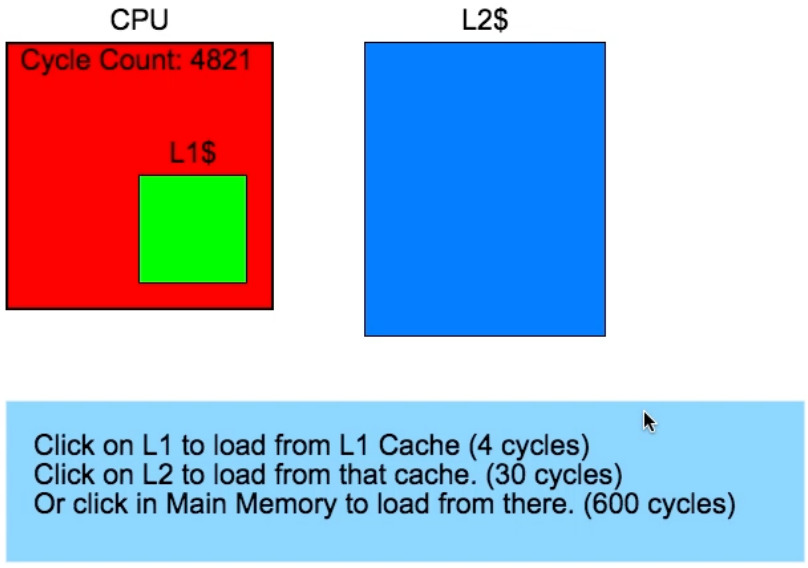
Discussion for today

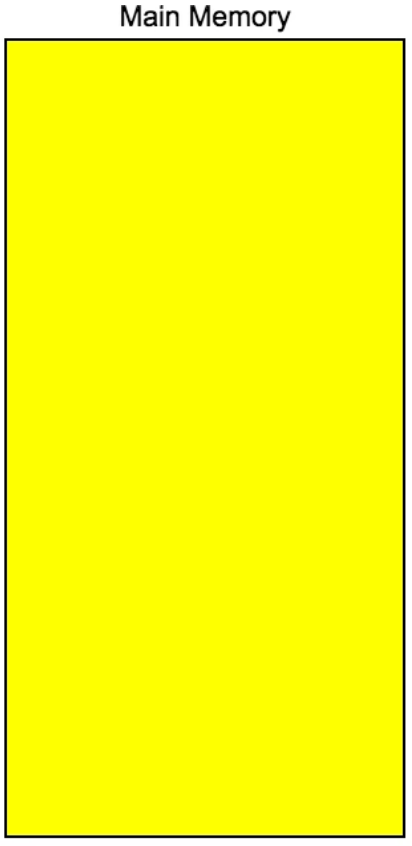
- **Time overhead**
  - Accessing any virtual address require one extra memory access
    - Step-1 is to fetch the page table base address from register (PTBR)
    - Step-2 is to index into the PTE and fetch the physical address
- Space overhead
  - Next lecture

Detour: Memory Hierarchy



Detour: How Bad is Latency as we go Down?

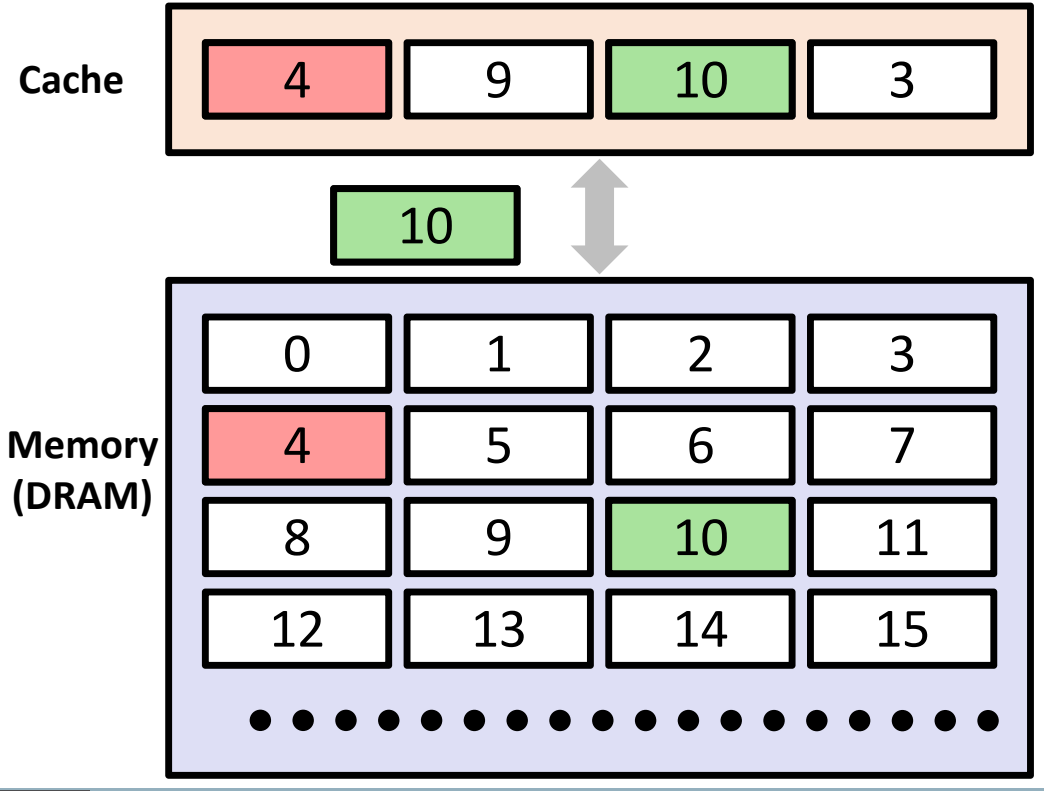




- Another analogy
  - Normalizing with L1 latency, and assuming one seconds is equal to 4 cycles
    - L1 = one second
    - L2 = 7.5 seconds
    - Main memory = 2.5 minutes
    - Hard drive = in several days!

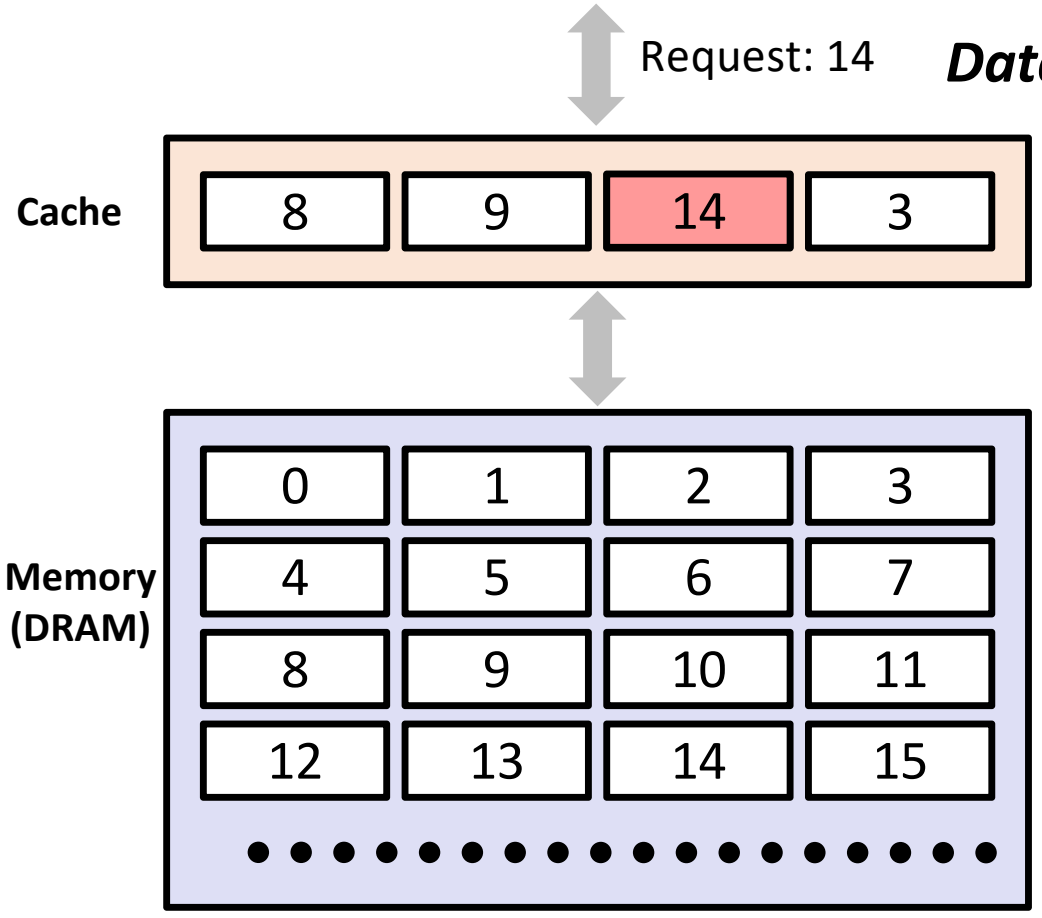
Animation source: https://overbyte.com.au/misc/Lesson3/CacheFun.html

Detour: General Cache Concepts



- Cache: A smaller, faster storage device that acts as a staging area for a subset of the data in a larger, slower device
  - Temporal and spatial locality

Detour: General Cache Concepts



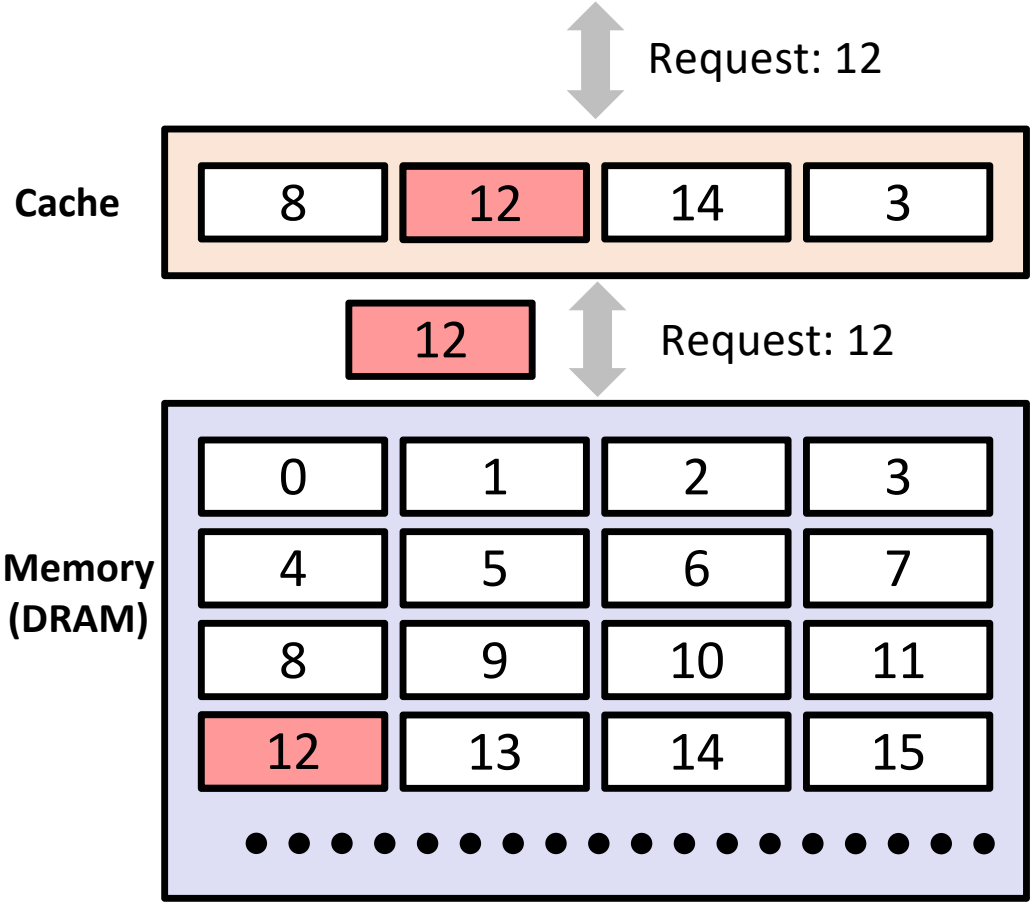
Data needed: 14

Data is in cache:

Hit!

- Cache hit
  - Data is already in the cache

Detour: General Cache Concepts



Data needed: 12

Data is not in cache:
Miss!

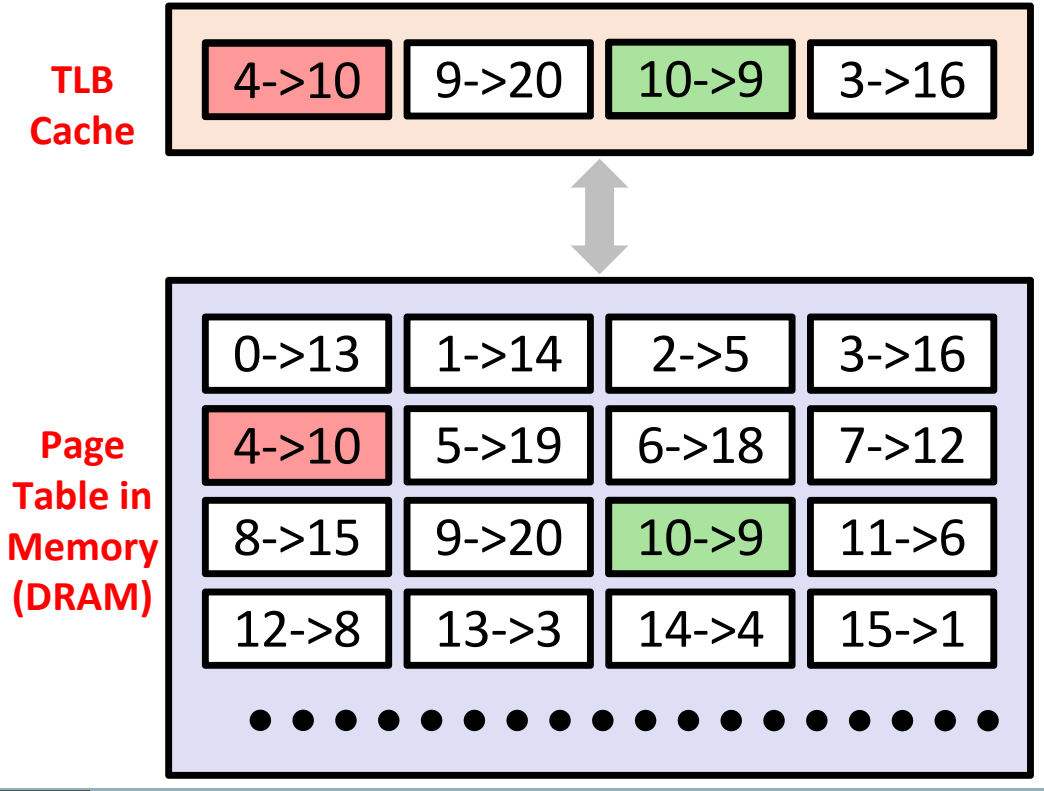
Data is fetched from *memory*

Data is stored in cache
By evicting some old data

So, How to Solve Slow Access of PT?

- Provide a hardware cache per CPU for saving frequently accessed PTE entries (different than the CPU cache L1, L2, etc.)
  - Translation Lookaside Buffer (TLB) cache that is a part of MMU
- Save most recent VPN->PPN mappings on TLB (plus some more info, but not PT entry)

TLB Hit and Miss

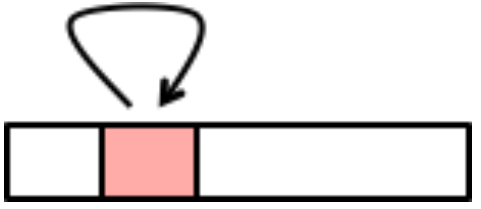


- TLB hit
  - If entry already in TLB then have the physical address without reading the page table
  - VPN(4)->PPN(10) is already on TLB. Hence, no need to access PT
- TLB miss
  - VPN(10)->PPN(9) mapping is not found on TBL
  - Locate PT using PTBR
  - Access page table to find the mapping and save the mapping on cache by evicting an earlier entry

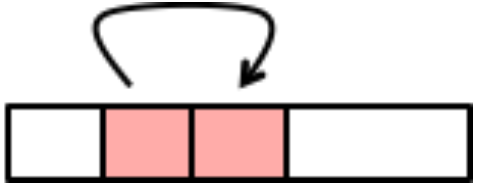
Locality Helps in Caching

- Principal of Locality
  - Empirical observation: Processors tend to access same set or nearby memory locations repetitively over a short period of time

- Temporal locality:
  - Recently referenced items are likely to be referenced again in the near future



- Spatial locality:
  - Items with nearby addresses tend to be referenced close together in time



Locality Helps in Caching

- Caching helps only if application supports **locality**
  - Let us understand the concept of **locality** using a simple application called as **iterative averaging**
  - Iterative averaging is the process of updating an array to so that each index becomes the average of the indices one before and one after it
  - After repeating this for many iterations, the array may converge to one set of numbers
    - Repetitive pattern in several scientific applications

Iterative Averaging

```
//all elements zeroed
float A[SIZE+2], B[SIZE+2];

void iterative_averaging() {
  A[SIZE+1] = B[SIZE+1] = 1;
  for (int iter=0; iter<ITERATIONS; iter++) {
    for (int j=1; j<=SIZE; j++) {
      B[j] = (A[j-1] + A[j+1])/2.0;
    }
    double* temp = B;
    B = A;
    A = temp;
  }
}
```

https://classes.engineering.wustl.edu/cse231/core/index.php/Iterative_Averaging

- For SIZE=9, the array A will eventually converge with values as 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9

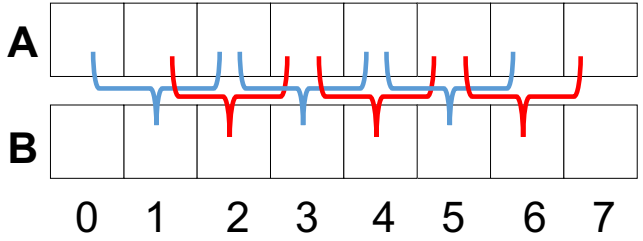
Spot the Locality

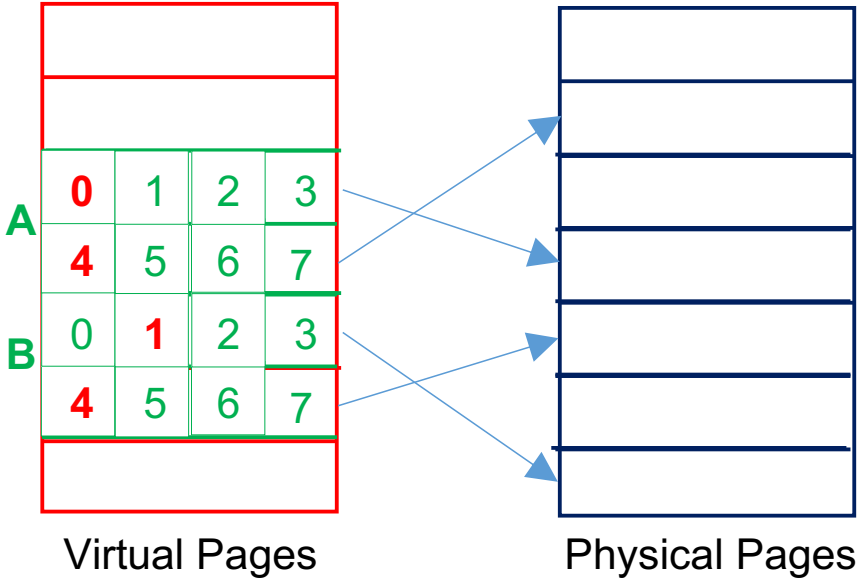
```
//all elements zeroed
float A[SIZE+2], B[SIZE+2];

void iterative_averaging() {
  A[SIZE+1] = B[SIZE+1] = 1;
  for (int iter=0; iter<ITERATIONS; iter++) {
    for (int j=1; j<=SIZE; j++) {
      B[j] = (A[j-1] + A[j+1])/2.0;
    }
    double* temp = B;
    B = A;
    A = temp;
  }
}
```

https://classes.engineering.wustl.edu/cse231/core/index.php/Iterative_Averaging

- Let us only consider the locality in array accesses
- Spatial locality
  - Inner for-loop access adjacent data
    - Reference array elements in succession (stride-1 reference pattern)
- Temporal locality
  - Outer for-loop
    - Accessing each arrays elements repeatedly

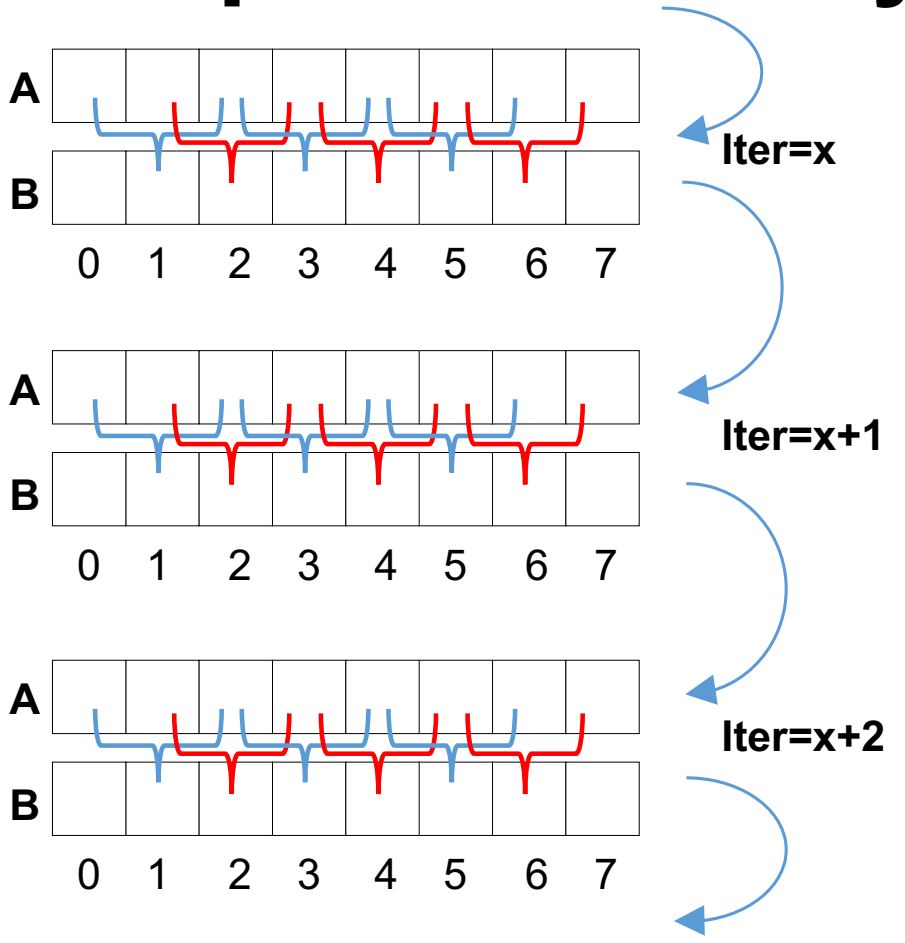
Spatial Locality





- Assumptions
  - The only memory access required are for accessing elements of arrays A and B
  - SIZE=6
  - Page size (PS) of 16 bytes
    - A requires 2 pages
    - B requires 2 pages
  - **TLB can hold 4 entries**
- Inside each iteration of outer loop
  - B is accessed **6** times and A is accessed **12** times (spatial locality)
  - Total memory accesses = 12+6=**18**
  - TLB misses
    - **2+2 = 4**
  - TLB hits
    - **10+4 = 14**

Temporal Locality

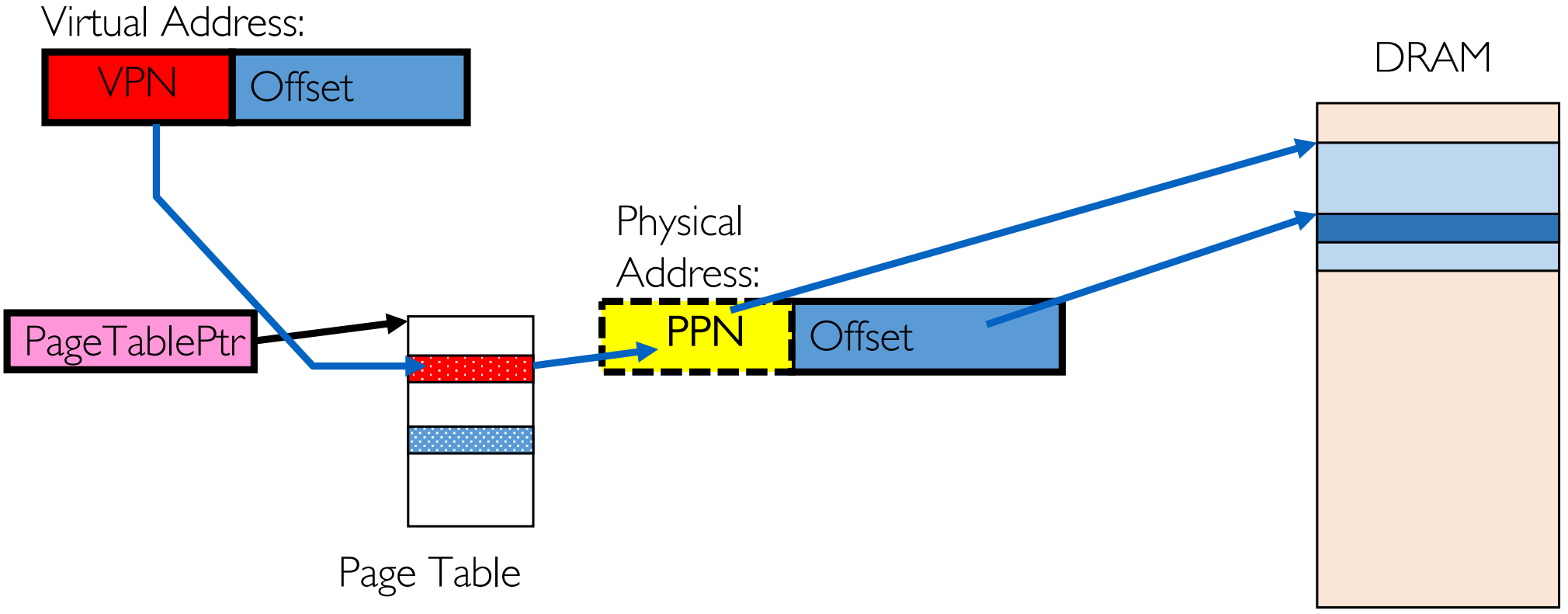


- Same indices are repeatedly accessed across the iterations of the outer for-loop
  - **Temporal locality**
- Total TLB misses for N number of iterations of outer for-loop?
  - 4 only!
- Total page table accesses with TLB?
  - 4 only!
- Total page table access without TLB?
  - 18N

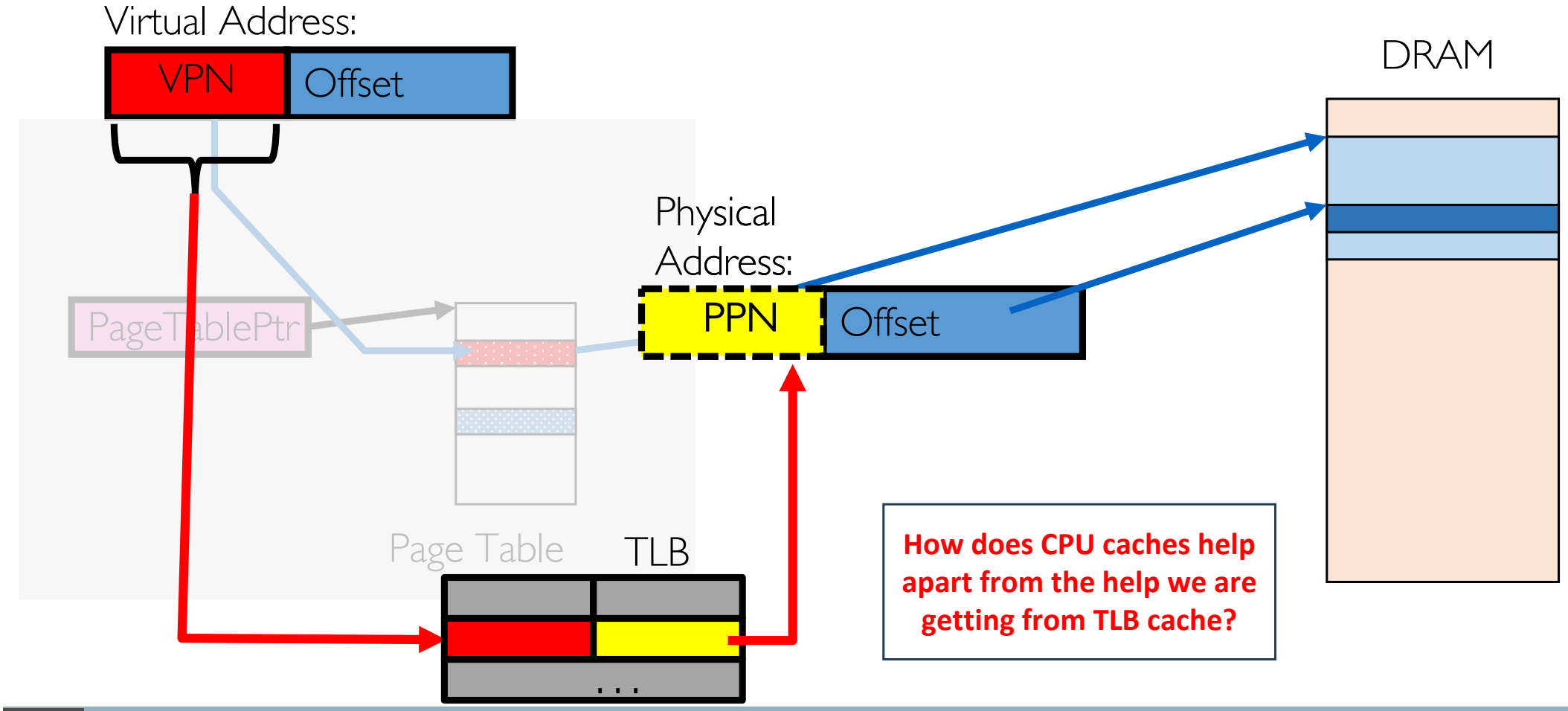
What to do at Context Switch?

- Need to do something, since TLBs map virtual addresses to physical addresses
  - Address Space just changed, so TLB entries no longer valid!
- Options?
  - Invalidate ("Flush") TLB: simple but might be expensive
    - What if switching frequently between processes?
  - Include ProcessID in TLB
    - This is an architectural solution: needs hardware

Putting Everything Together: Address Translation



Putting Everything Together: TLB



How does CPU caches help apart from the help we are getting from TLB cache?

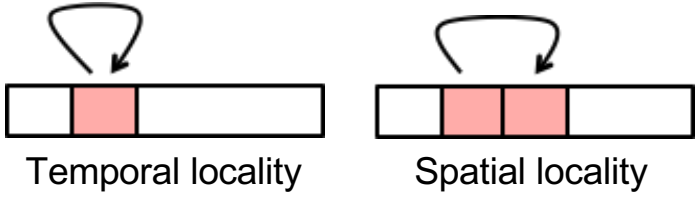
Next Lecture

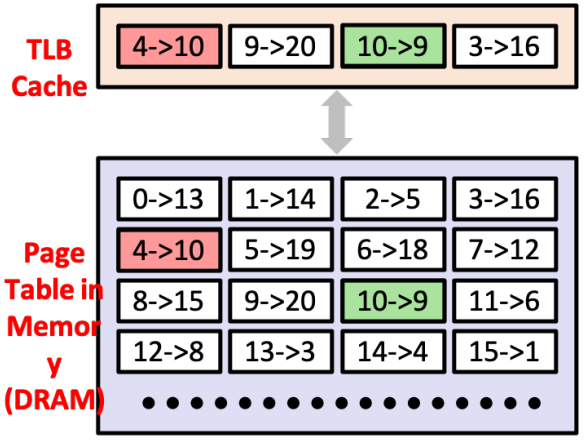
- Space overheads with the page table

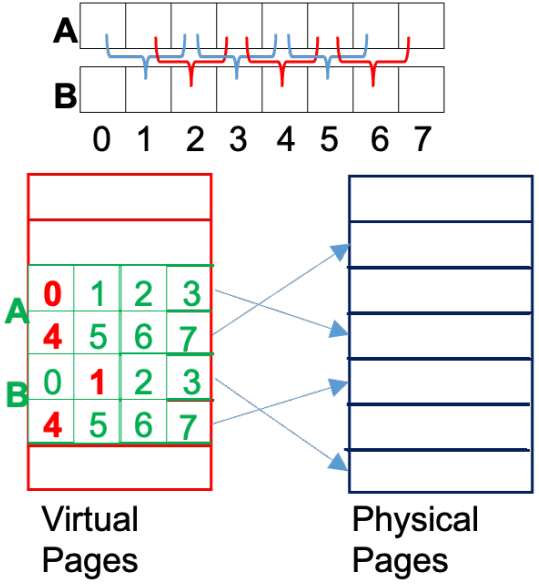
Lecture 19: Multi-Level Page Table

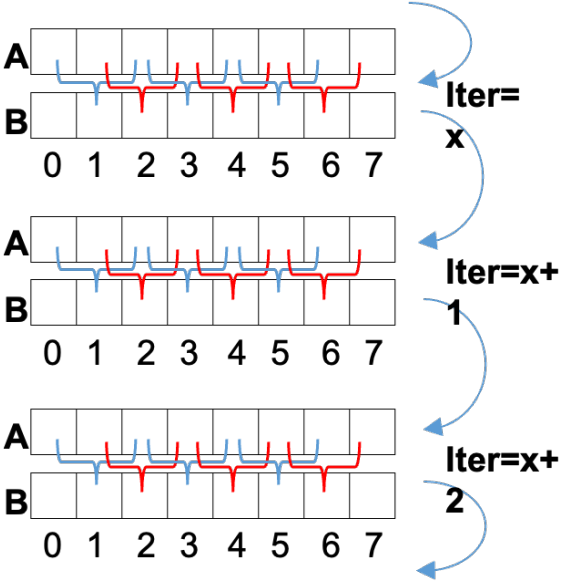
Vivek Kumar
Computer Science and Engineering
IIIT Delhi
vivekk@iiitd.ac.in

Last Lecture







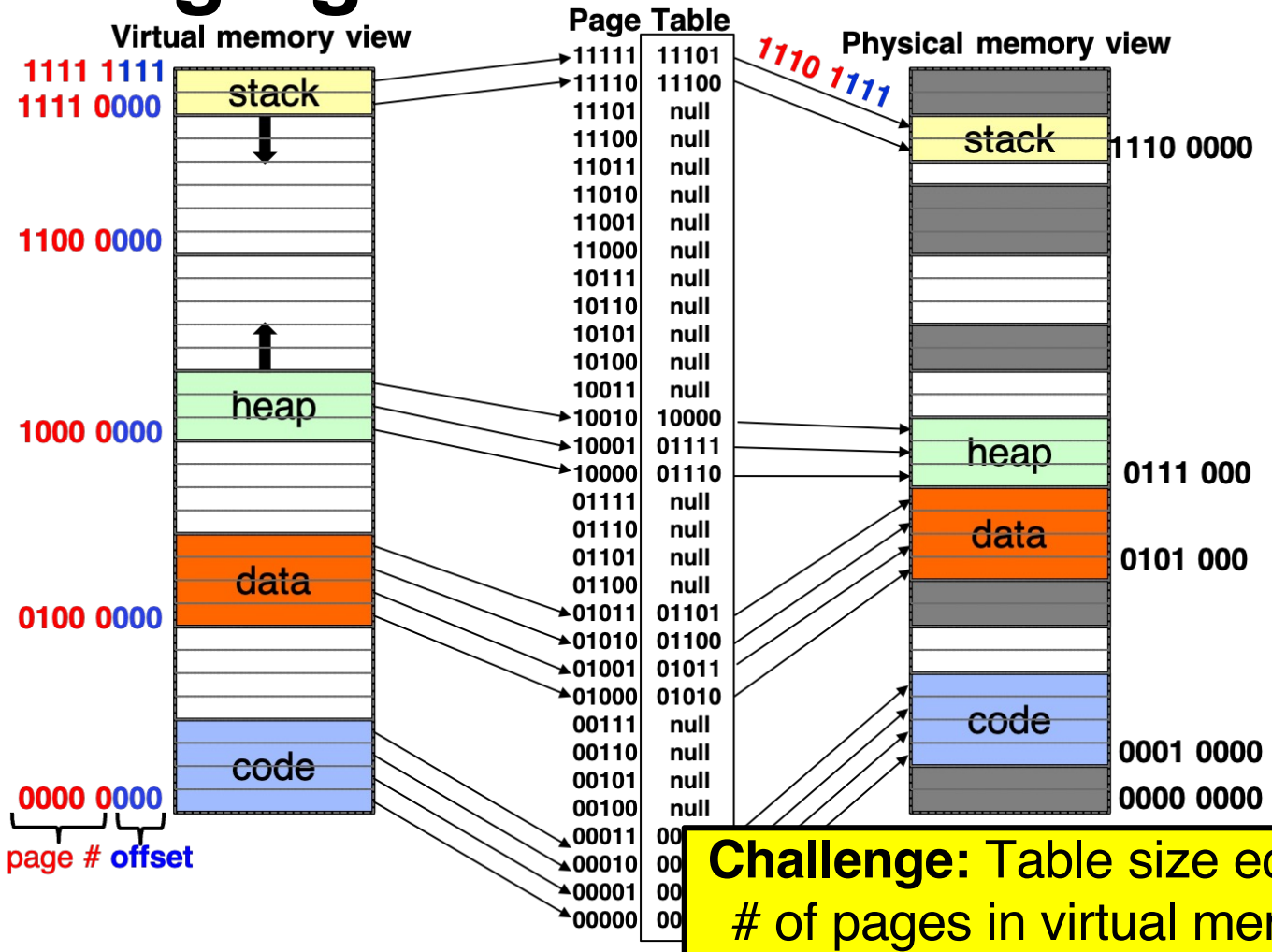


- Time overheads with page table access
  - PT base address in PTBR (1-cycle)
  - Accessing PT index in DRAM (several cycles)
- Translation look aside buffer (TLB)
  - hardware cache per CPU for saving frequently accessed PTE entries (part of MMU)
- Locality helps in caching
  - Temporal locality
  - Spatial locality

Today's Class

- Challenges with simple page table
- Multi-Level Page Table (MLPT)
- MLPT with TLB

Paging



Challenge: Table size equal to # of pages in virtual memory!

- Recall, process address space is not contiguous
  - Inter-segment memory is not contiguous
  - Only intra-segment virtual memory is contiguous (the physical memory may/may-not be contiguous)
- Page table until now is an **array** where the length of the array will be total number of VPN being used in the process address space
  - Not all the VPNs will be used as the segments are not contiguous (there could be gaps between them)

Overheads with the Linear Page Table

Physical Page Number (PPN)	Free (OS)	0	PS	D	A	PCD	PWT	U	W	P
31-12	11-9	8	7	6	5	4	3	2	1	0

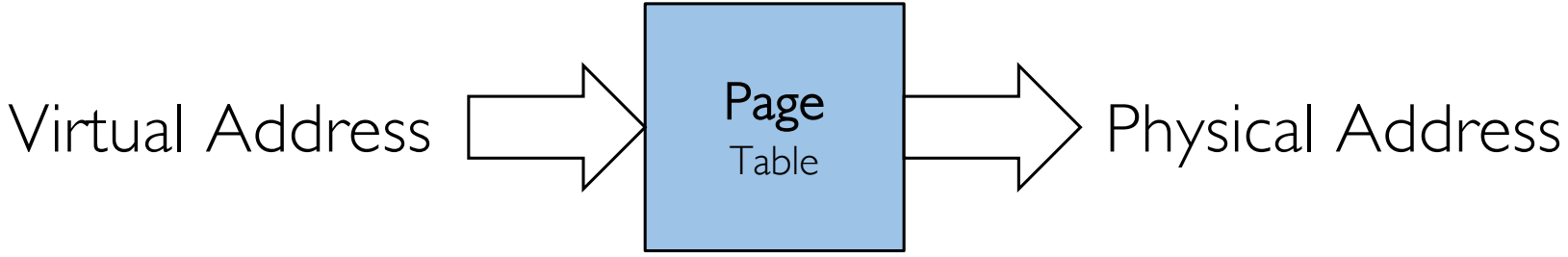
- **Time overhead**
  - Covered in last lecture
- Space overhead

Discussion for today

  - Single entry in PTE on x86 (32-bit) = 4 bytes
  - Total number of unique VPNs = 2^{20}
  - Total memory for one page table = 2^{20} x 4 = **4MB**
  - If there are 100 processes, total memory requirement for page table = **400MB**!
  - On 64-bit: 2^{52} unique VPNs x 8 bytes = **36 Exa-bytes!**

Challenge: How to Structure a Page Table?

- Page Table is a *map* (function) from VPN to PPN



- Simple (linear) page table corresponds to a *very large* lookup table
  - VPN is index into table, each entry contains PPN
- What other map structures can you think of?
  - Trees?
  - Hash Tables?

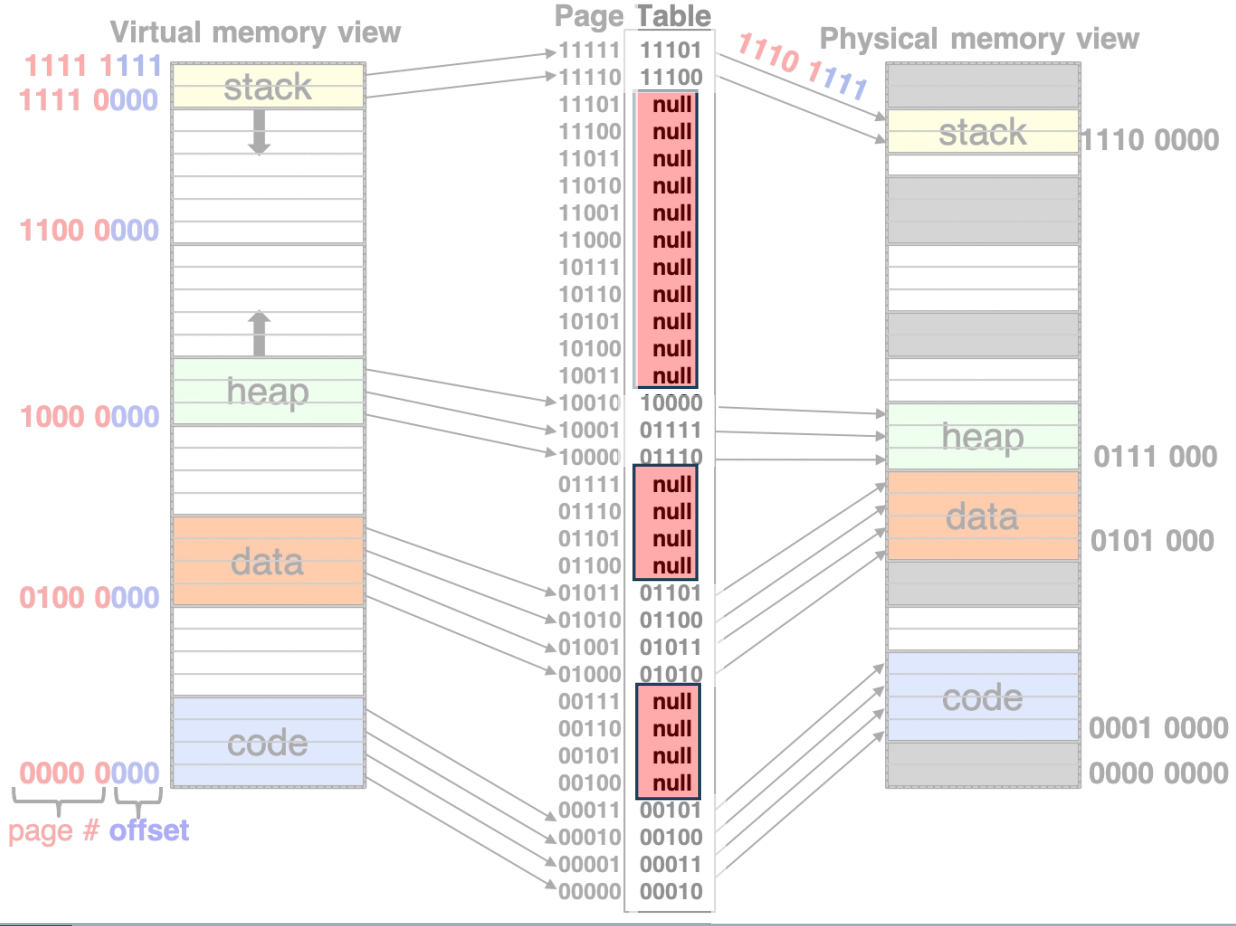
A Simple Fix

- How about using a page size bigger than 4KB?
- Let's try using a 16KB page size in 32-bit addressing
  - Total unique VPNs are 2^{32}/16KB = 2^{18}
  - Space overhead
    - Single entry in PTE on x86 (32-bit) = 4 bytes
    - Total number of unique VPNs = 2^{18}
    - Total memory for one page table = 2^{18} x 4 = **1MB**
    - We are able to reduce single PT size by a factor of 4!
- Although we are able to alleviate the space overheads by increasing the page size, but do you see any issues in this approach?
  - Internal fragmentation!

Mapping Virtual to Physical Address

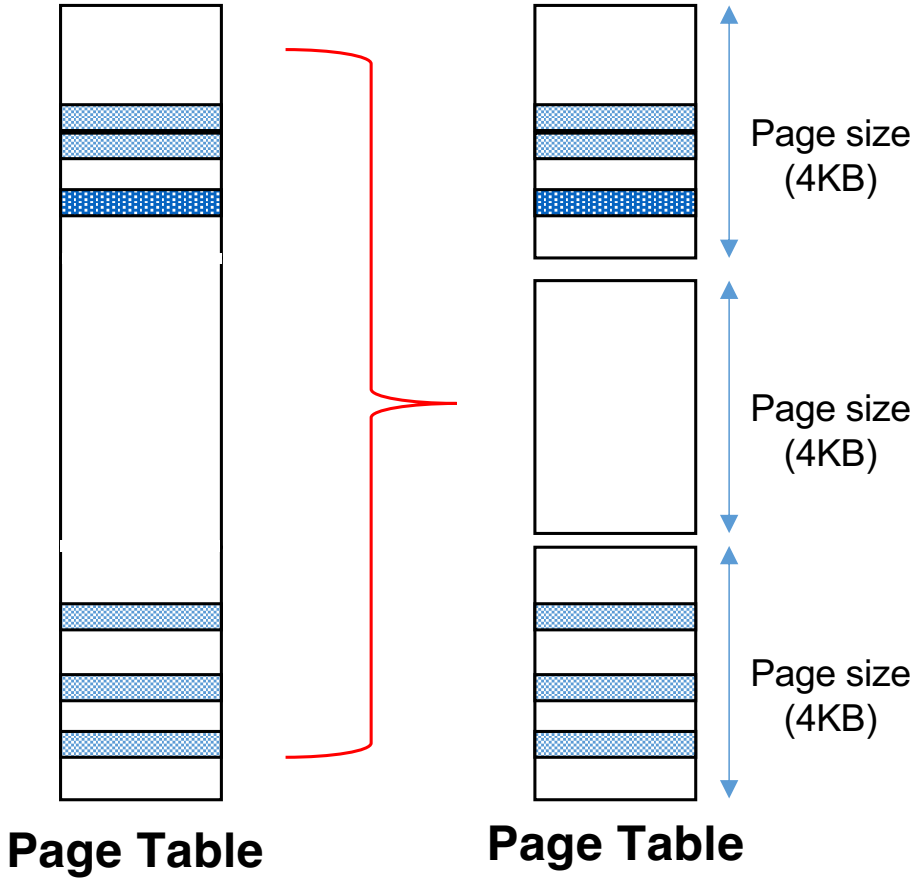
- Approaches
  1. Virtual address is same as the physical address
  2. Virtual address is not the same as physical address
     a) Using base & bound for VA to PA mapping
     b) Using segmentation for VA to PA mapping
     c) Paging
        a) Using Page Table (PT) in RAM and having a base register for PT
        b) Using Page Table (PT) in RAM and having a base register for PT as well as a TLB cache
        c) Using Multi-level Page Table

Reason for Large Page Table



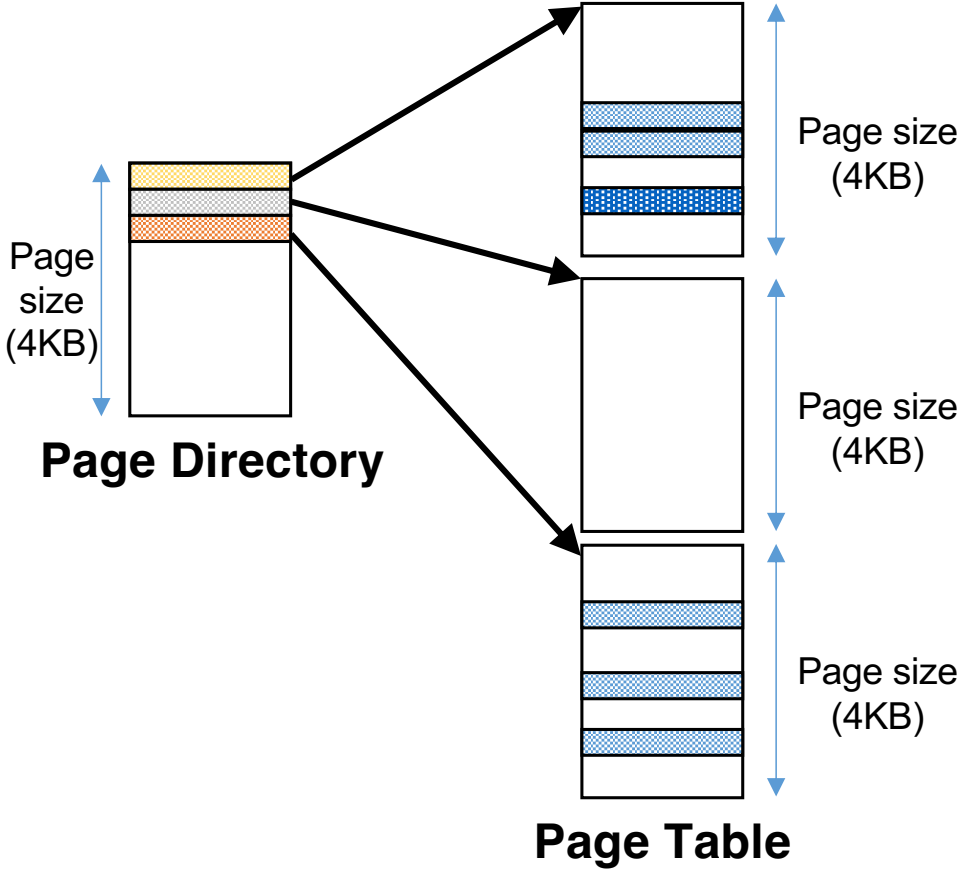
- There are several invalid ("null") entries in an array-based implementation of the page table
  - Not all indices are going to be valid in the look up table
- We can reduce the size of the page table if we can use a data structure that can help in removing the invalid "null" entries inside the page table
  - **Multi-Level Page Table (MLPT)**
  - We can also combine paging along with segmentation
    - We are skipping it in our discussion as anyway segmentation leads to complexity as we have already seen

Multi-Level Page Table (MLPT)



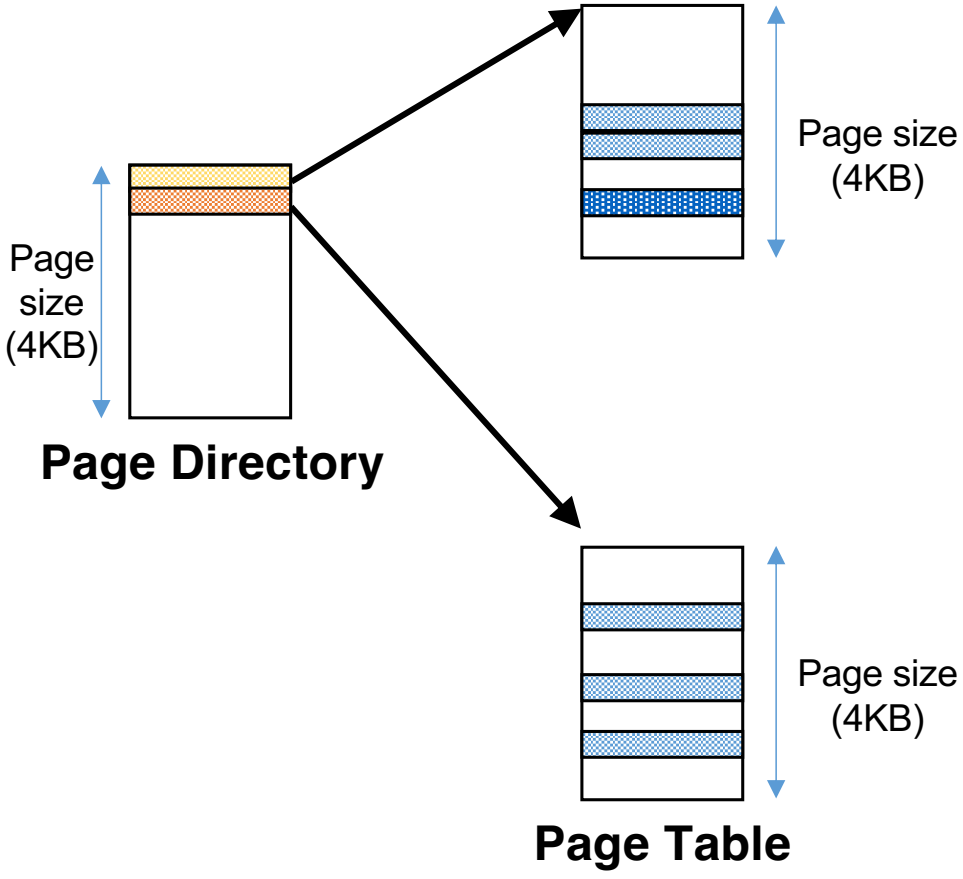
- Idea-1
  - Split the entire page table into page size entries
  - On x86 with 4KB page size, there would be 1024 VPN entries inside each page size blocks (4KB/4bytes)
  - Types of “Pages”
    - All invalid (“null”) VPNs
    - At least one valid VPN
  - Guess what we can do next?

Multi-Level Page Table (MLPT)



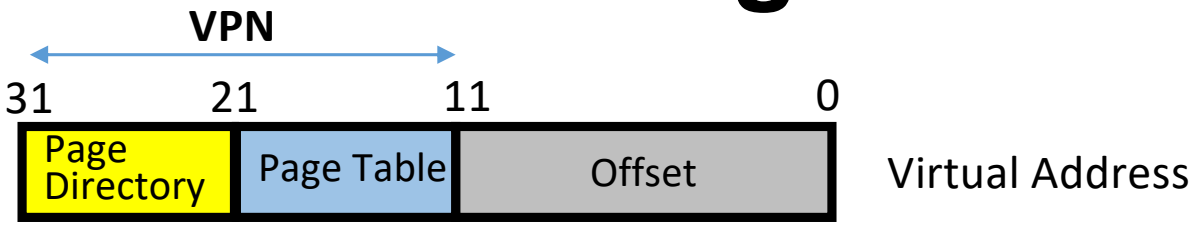
- Idea-2
  - Create a separate per-process data structure (Page Directory) that holds references to the first entry inside each of the pages of page table (see previous slide)

Multi-Level Page Table (MLPT)



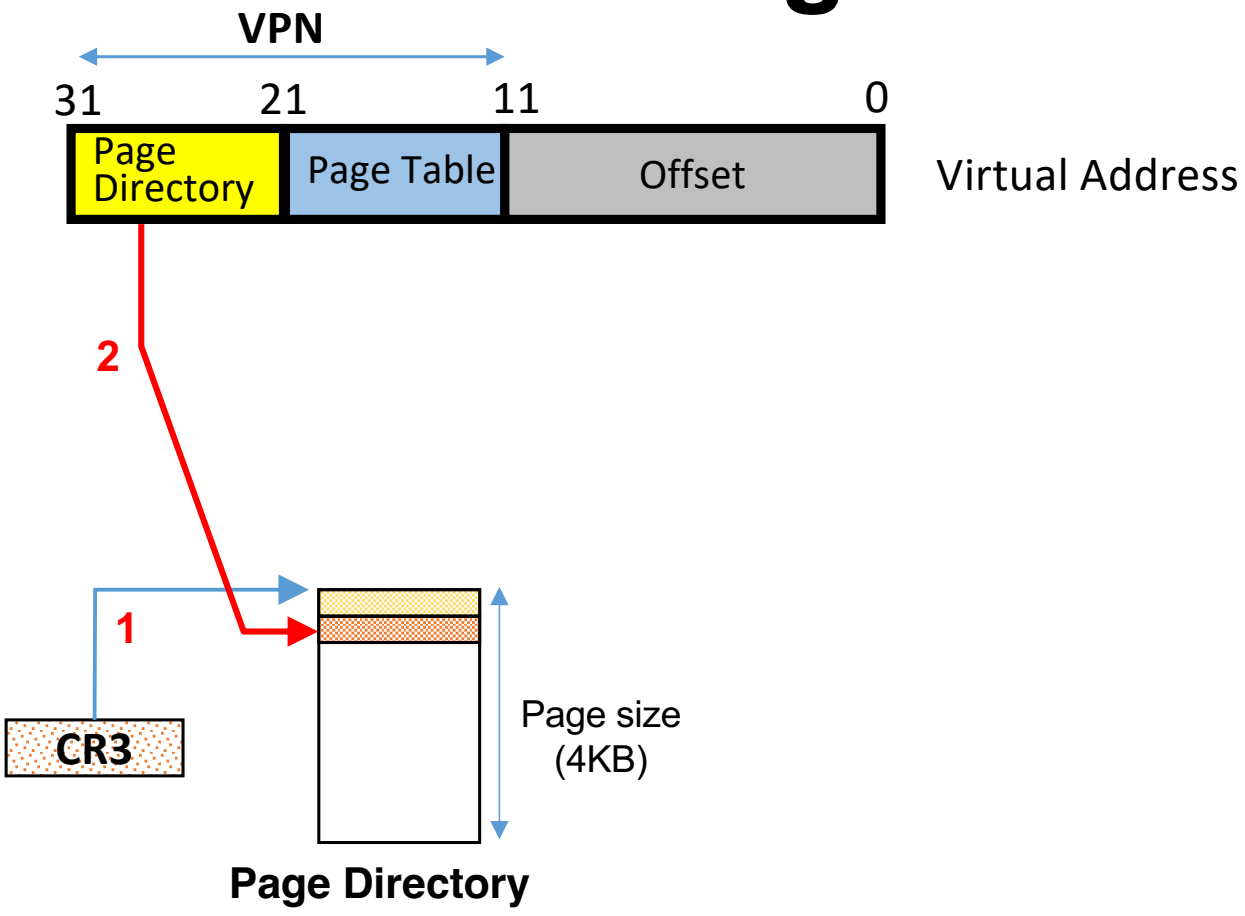
- Idea-3
  - Don't allocate a page of page table if there is not a single valid entry inside it
- This example of MLPT is a two-level page table
- There could be more than two-levels of page table!
  - Out of scope of this course

Two-Level Page Table in x86



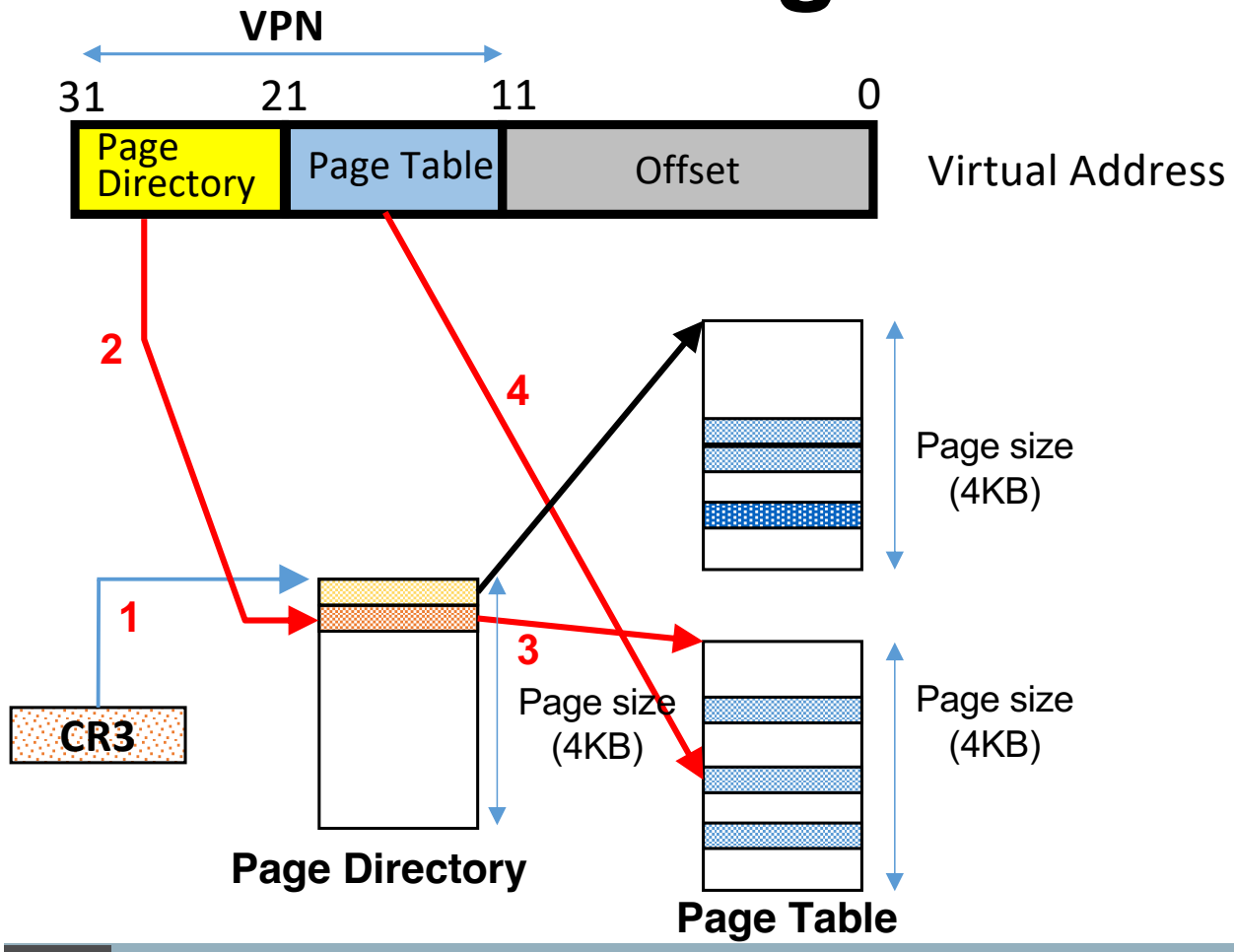
- Virtual address (32-bit) contains three different entries (instead of two that we saw earlier in case of linear page table)
  - Page Directory index (PDIndex)
  - Index into page table corresponding to the above entry inside page directory
  - Offset to address in the physical page

Two-Level Page Table in x86



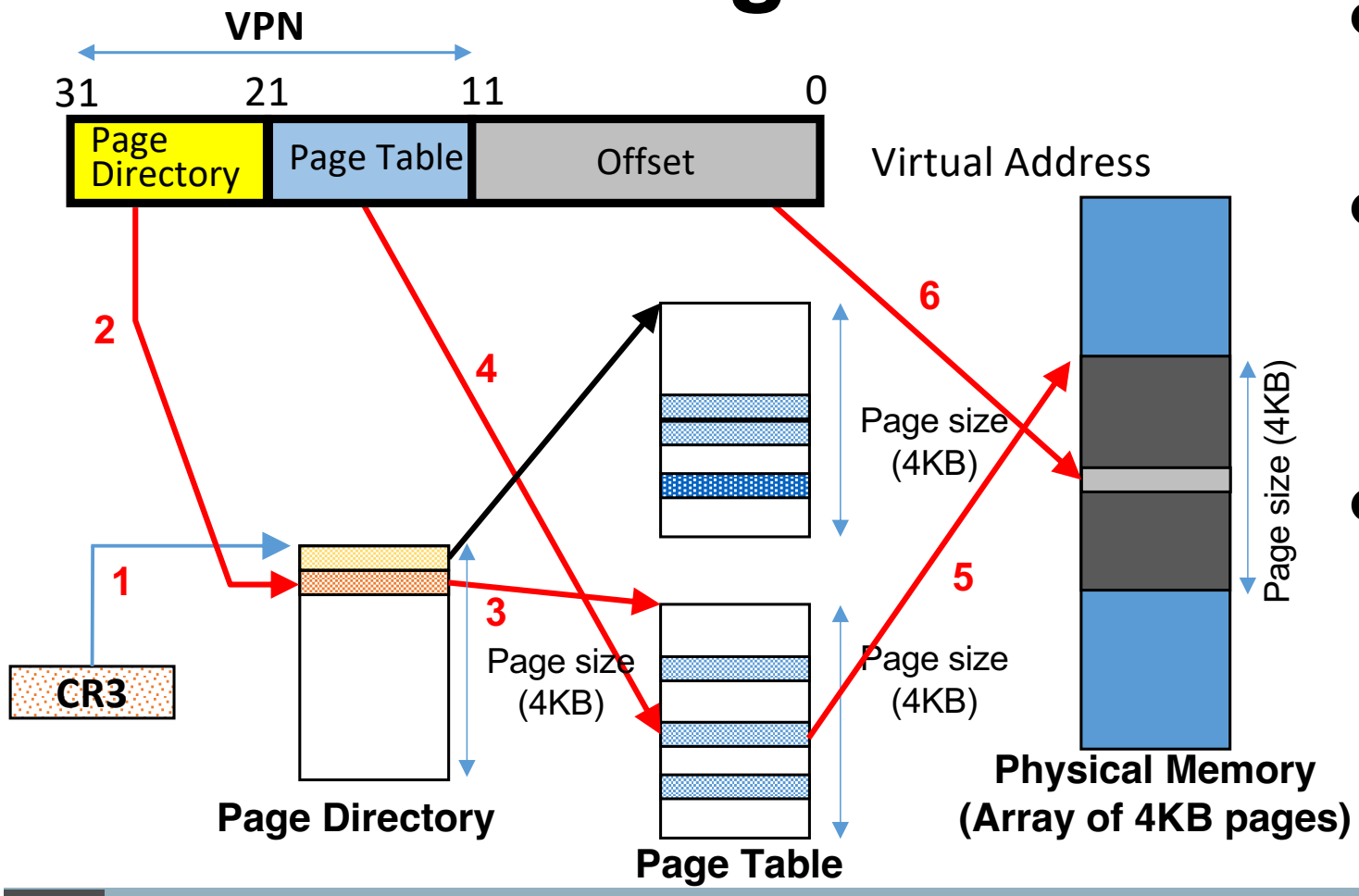
- Control Register (CR3) register (part of MMU) stores the starting address of the Page Directory (PD)
- PD contains 1024 Page Directory Entry (PDEs)
  - Each PDE contains the Page Frame Number and a status bit
  - Status=0 if there is not a single valid entry in the PFN, otherwise status =1 (valid PFN only if status=1)
- Address of the PDE associated with this VPN
  - CR3 + PDIndex x 4 bytes

Two-Level Page Table in x86



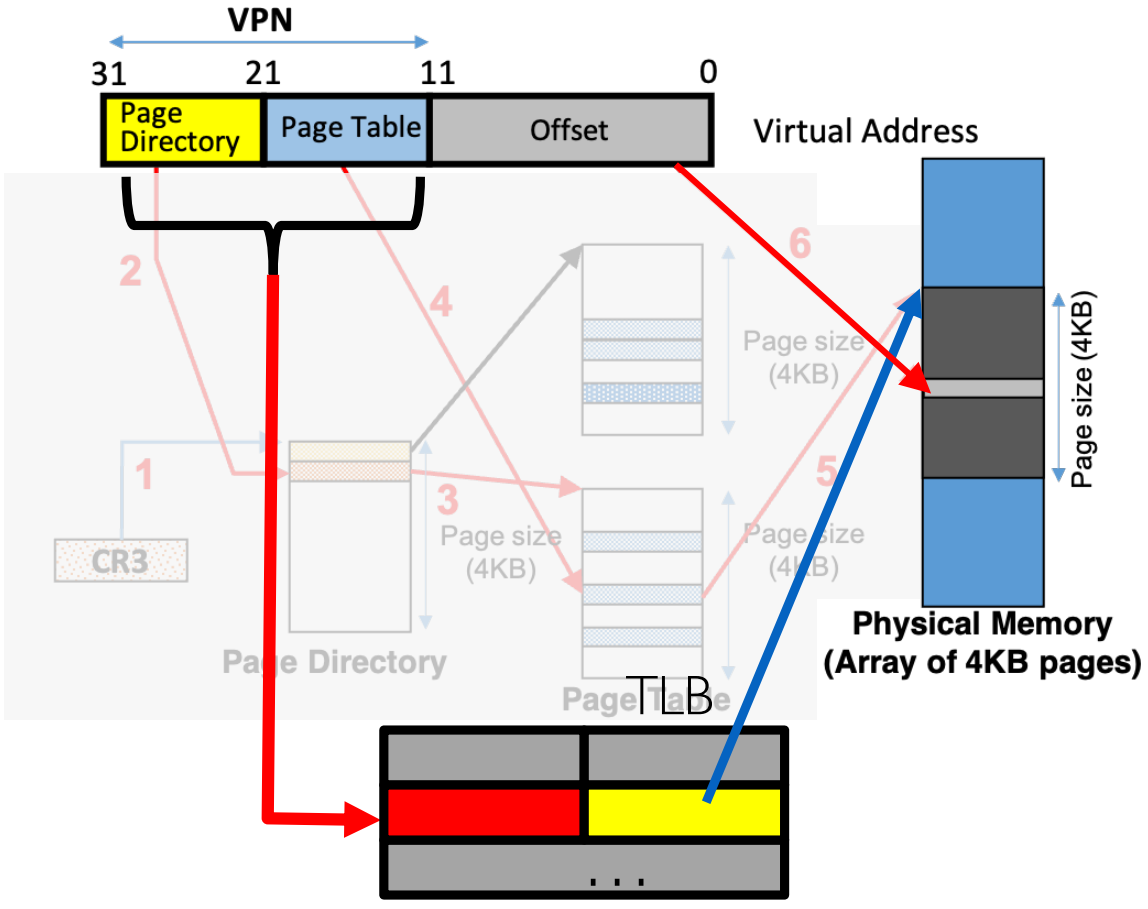
- Page Table Entry (PTE) can be now found from the PDE
  - PDE + (Page Table index * 4 bytes)

Two-Level Page Table in x86



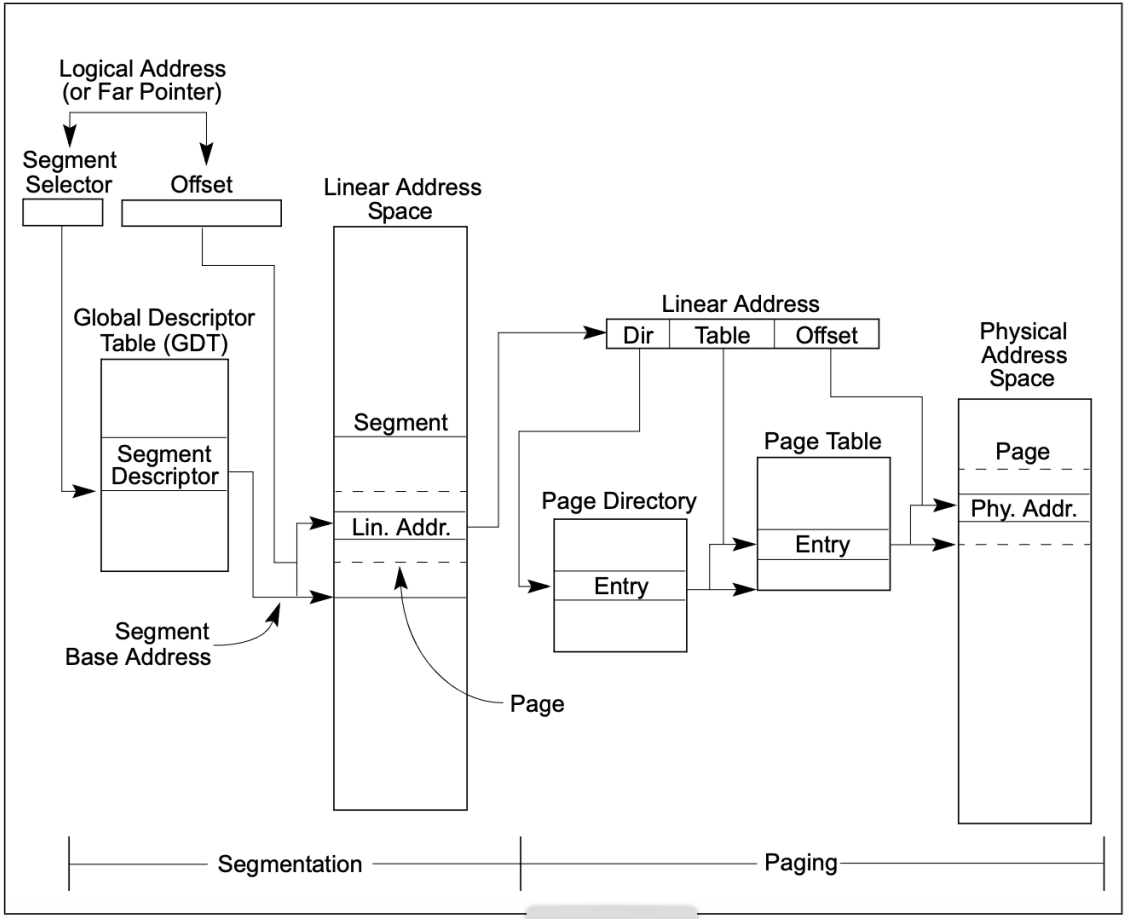
- Physical Page Frame Number (PFN) can be found from the PTE
- The physical address is obtained by adding the "offset" into the starting address of PFN
- What would happen when considering TLB?
  - None of these complex lookups are required if TLB contains the VPN->PPN

MLPT Along with TLB



- Look ups are required only in case of TLB misses
- Access latency significantly low when TLB hits
  - See Lecture-18

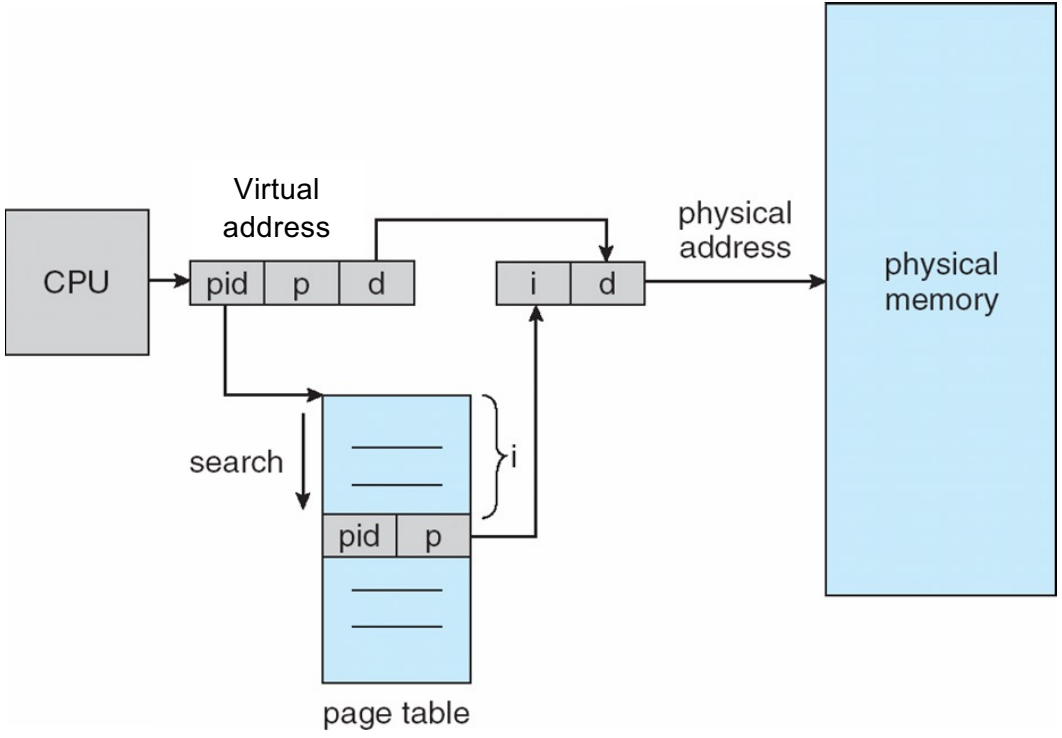
Memory Management in Intel x86



Mapping Virtual to Physical Address

- Approaches
  1. Virtual address is same as the physical address
  2. Virtual address is not the same as physical address
     a) Using base & bound for VA to PA mapping
     b) Using segmentation for VA to PA mapping
     c) Paging
        a) Using Page Table (PT) in RAM and having a base register for PT
        b) Using Page Table (PT) in RAM and having a base register for PT as well as a TLB cache
        c) Using Multi-level Page Table
        d) Using Inverted Page Table

Inverted Page Table



Picture source: https://www.os-book.com/OS10/

- Issues with MLPT
  - Easy implementation but many page tables
- Inverted Page Table
  - Size is independent of the virtual address space, and is directly related to the amount of physical memory
    - Each entry points to a page in the physical memory
    - Each entry corresponds to the VPN (“p”) and process (“pid”) that owns that page
    - The tuple (pid, p) inside a virtual address is used as a hash key to find the entry in the page table
  - Single page table contains entries for all the processes
    - Complex hash function!
  - It is called as inverted page table, as total entries is equal to the total number of PPN
    - In contrast, the page tables we have seen so far (a.k.a. forward page tables) the entries are according to the VPN

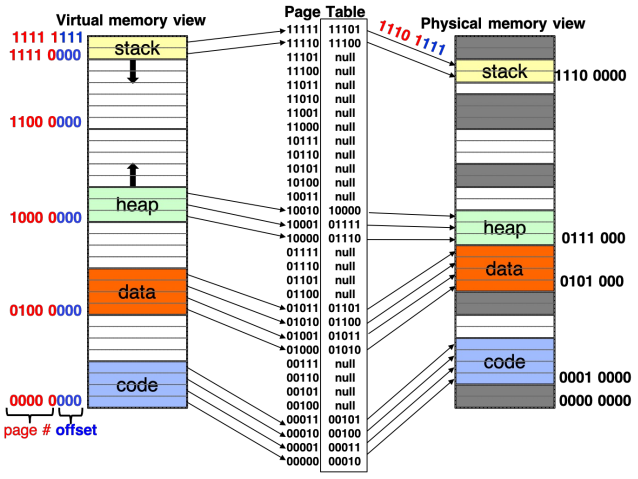
Next Lecture

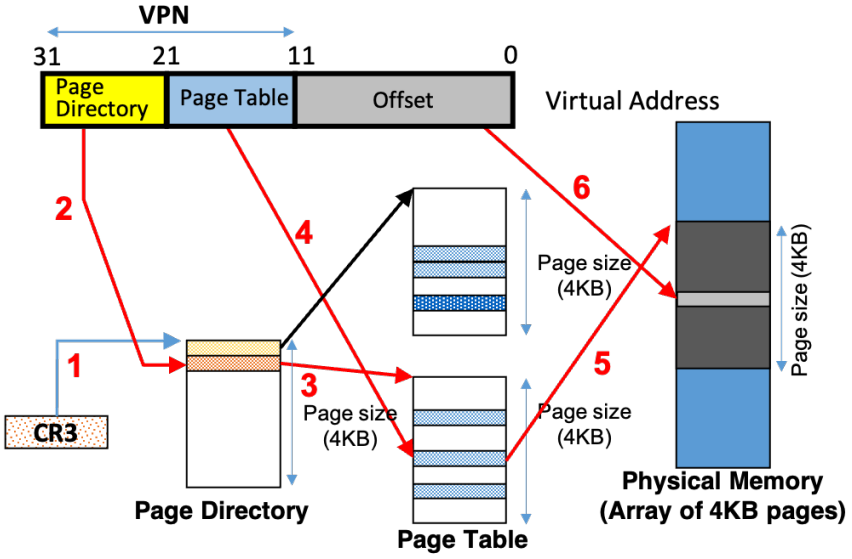
- Demand paging
  - Last remaining topic in virtualization of memory

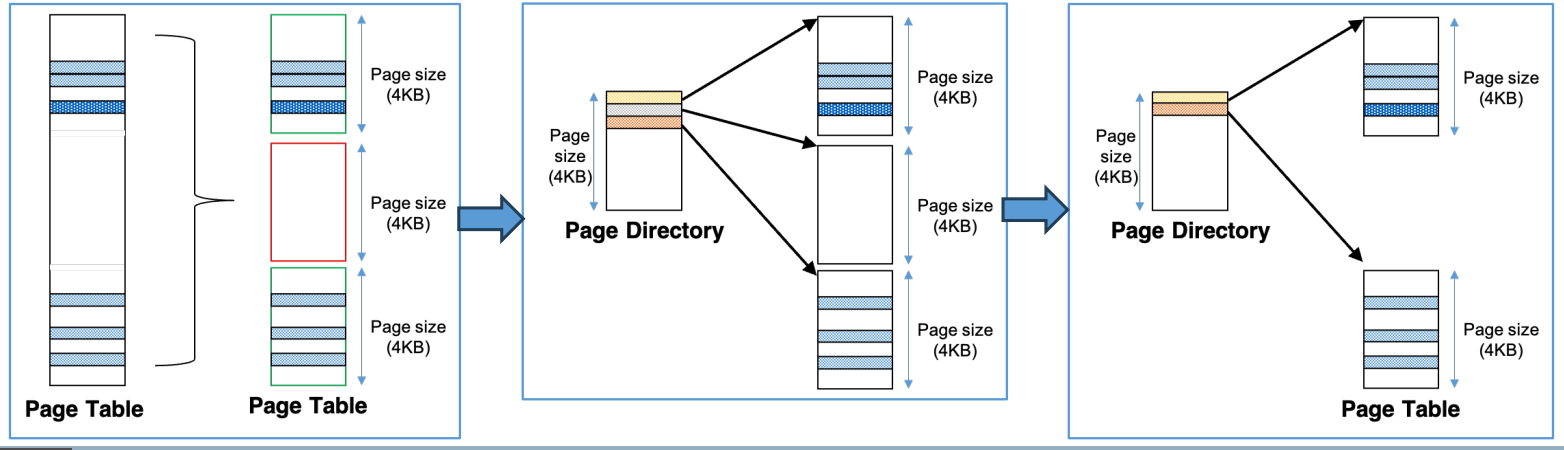
Lecture 20: Demand Paging

Vivek Kumar
Computer Science and Engineering
IIIT Delhi
vivekk@iiitd.ac.in

Last Lecture







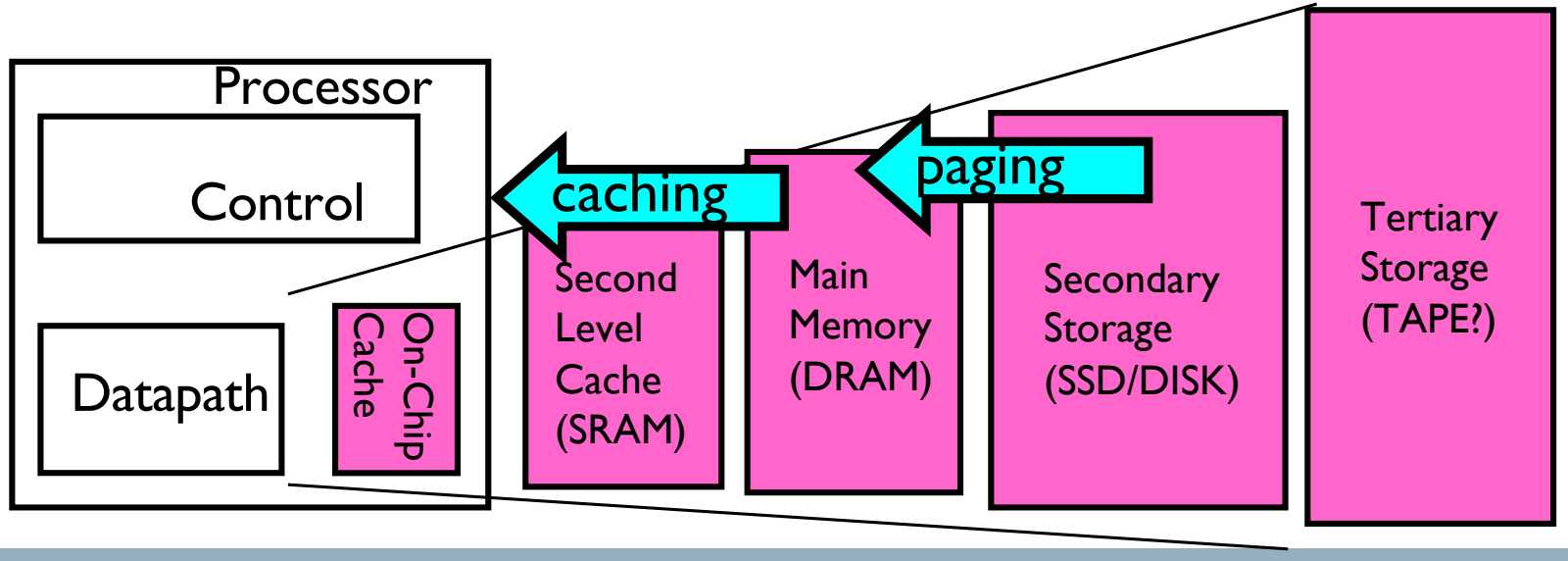
- Process address space is not contiguous. Hence, having a linear page table would lead to huge space overheads
- Increasing the page size can alleviate the overhead, but it would lead to fragmentation
- Multi-Level Page Table (MLPT)

Today's Class

- Demand paging
- Page replacement policies
  - FIFO
  - Random
  - LRU
    - Implementing LRU

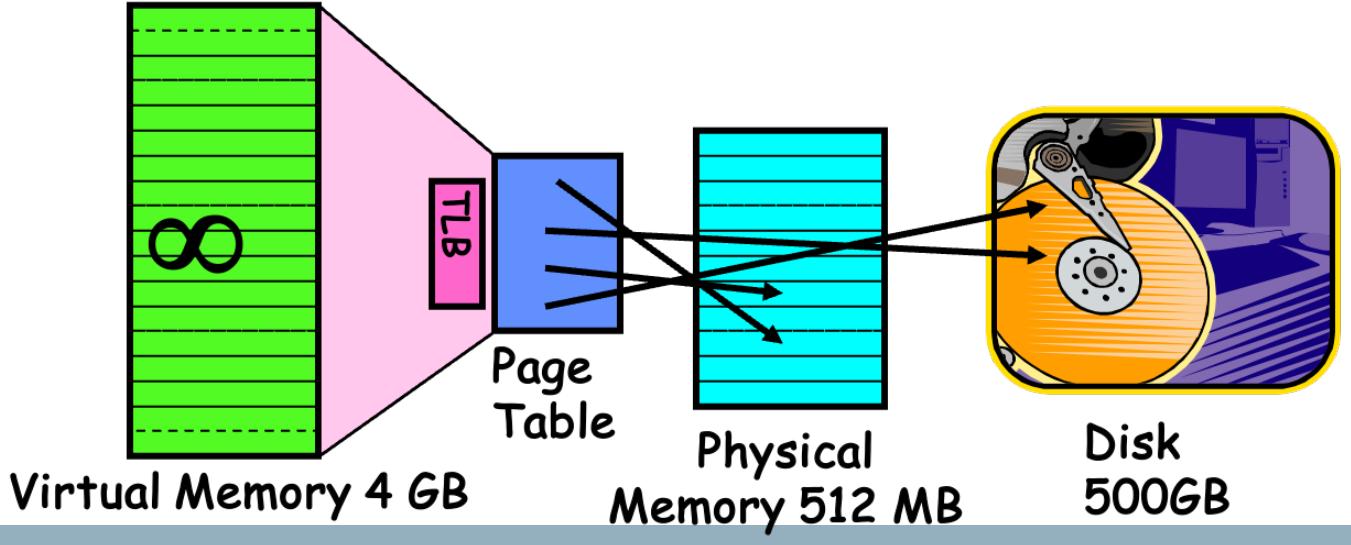
The Illusion of Infinite Memory

- Modern programs require a lot of physical memory
- But they don't use all their memory all of the time
  - 90-10 rule: programs spend 90% of their time in 10% of their code
  - Wasteful to require all of user's code to be in memory
- Solution: use main memory as "cache" for disk

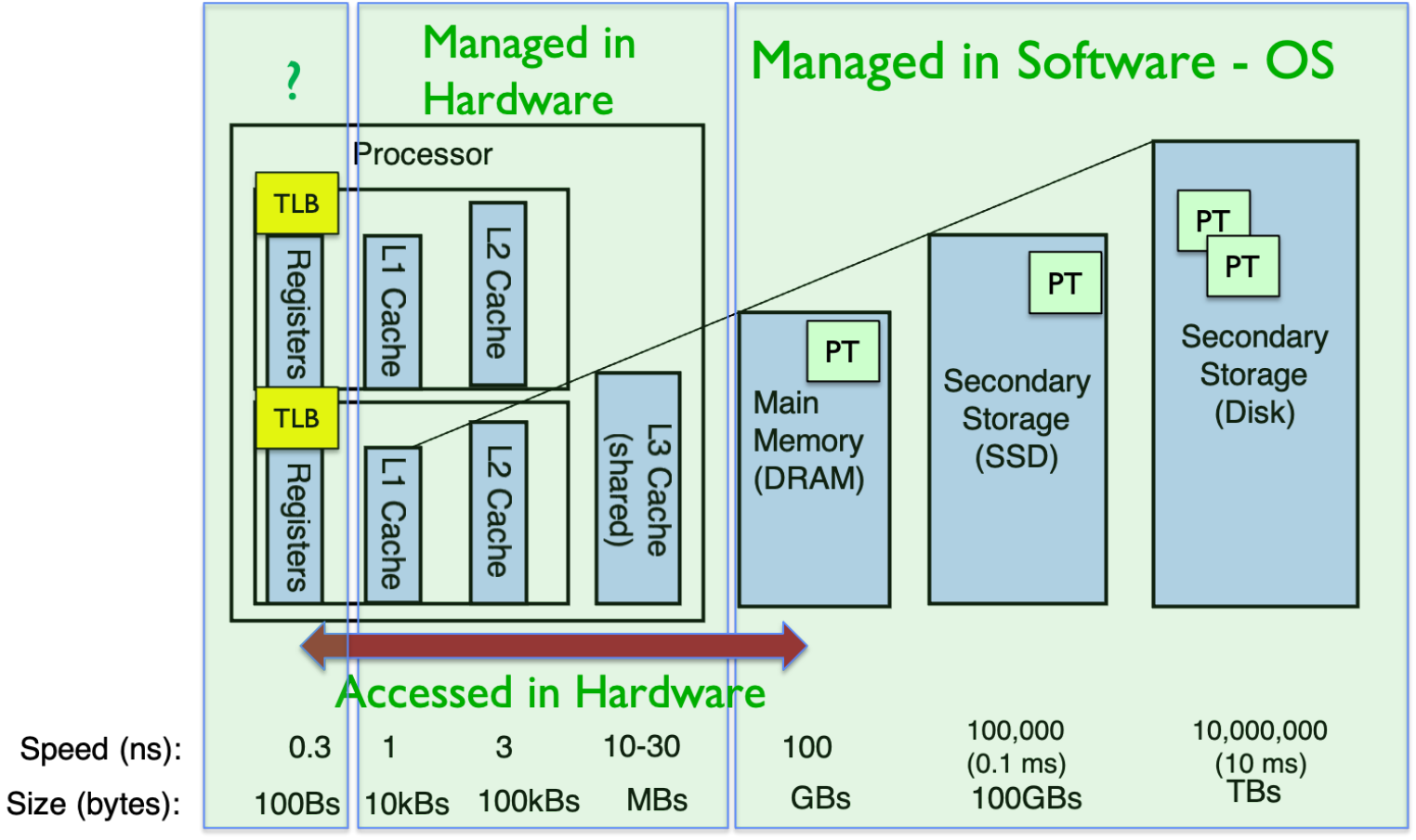


The Illusion of Infinite Memory

- Disk is larger than physical memory ⇒
  - In-use virtual memory can be bigger than physical memory
  - Combined memory of running processes much larger than physical memory
    - More programs fit into memory, allowing more concurrency
- Principle: Transparent Level of Indirection (page table)
  - Supports flexible placement of physical data
    - Data could be on disk or somewhere across network
  - Variable location of data transparent to user program
    - Performance issue, not correctness issue

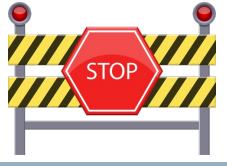


Memory Hierarchy



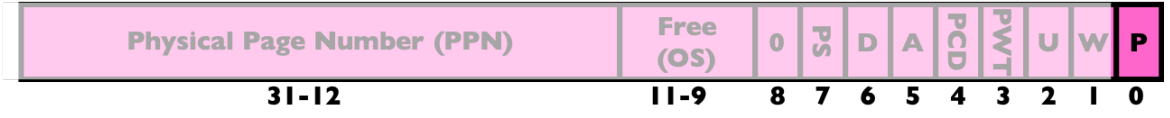
Mapping Virtual to Physical Address

- Approaches
  1. Virtual address is same as the physical address
  2. Virtual address is not the same as physical address
     a) Using base & bound for VA to PA mapping
     b) Using segmentation for VA to PA mapping
     c) Paging
        a) Using Page Table (PT) in RAM and having a base register for PT
        b) Using Page Table (PT) in RAM and having a base register for PT as well as a TLB cache
        c) Using Multi-level Page Table
        d) Demand paging

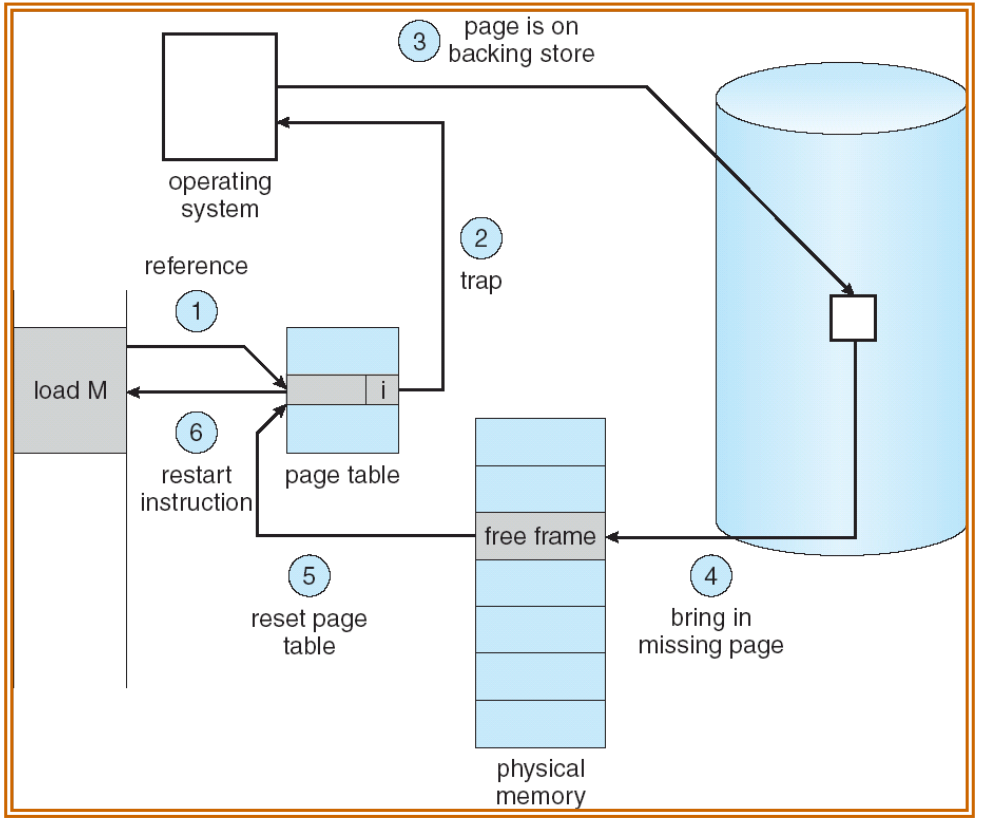


Demand Paging

- It the mechanism to provide the illusion of infinite memory
- OS brings the pages into memory when it is accessed (i.e., on demand)
- PTE makes demand paging implementable using the Present bit
  - P=Valid ⇒ Page in memory, PTE points at physical page
  - P=Not Valid ⇒ Page not in memory; use info in PTE to find it on disk



Handling Page Fault in Demand Paging



- When translating VA to PA, MMU reads present bit
  - Present bit in page table entry indicates if a page of a process resides in memory or not
- If page present in memory, directly accessed, otherwise generate page fault exception and move into kernel mode
- OS fetches disk address of page and issues read to disk
  - OS keeps track of disk address
- OS context switches to another process
  - Current process is blocked and cannot run
- When disk read completes, OS updates page table of process, and marks it as ready
- When process is scheduled again the OS restarts the instruction that caused page fault

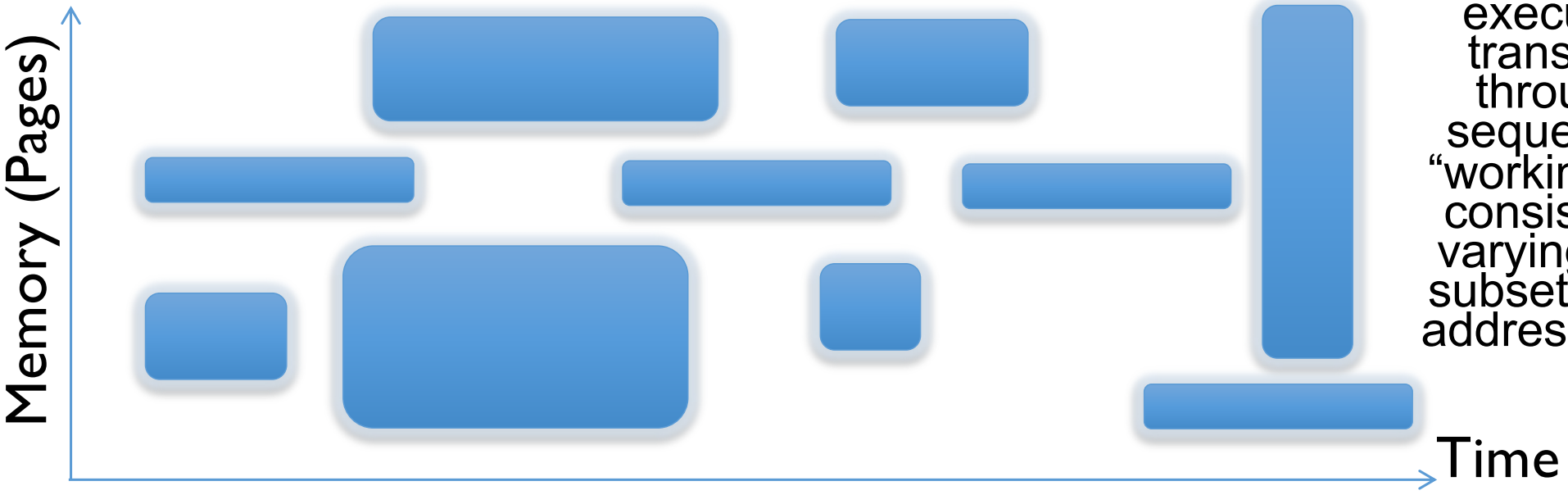
processes or users. Daemons are typically started during the system boot process and continue to run in the background, waiting for specific events or performing routine tasks. Here are some key characteristics and examples of daemon processes:

1. **Background Processes:**
   - Daemons run in the background, detached from any terminal session. They are not associated with any user interface.
2. **Long-Running:**
   - Daemons are designed to be long-running processes, often initiated at system startup and terminated only when the system is shut down.
3. **Independent of User Sessions:**
   - Daemons are not tied to any specific user's session. They operate independently and may provide services to multiple users or the entire system.
4. **No User Interaction:**
   - Daemons typically do not interact directly with users. They work silently in the background, handling tasks such as system maintenance, network services, or automated processes.
5. **Examples of Daemon Processes:**
   - **HTTP Server Daemons:** Web servers like Apache or Nginx run as daemons, handling incoming web requests.
   - **Print Spooler Daemon:** Manages print jobs in the background.
   - **Database Daemons:** Database management systems often have daemon processes handling tasks like background updates or scheduled maintenance.

Moving Unused Pages to Disk

- OS keeps some amount of DRAM always free by deciding some upper cap
- Whenever the upper cap memory level is breached, unused pages are moved out to disk
  - OS uses a daemon process for carrying out this activity
- To help decide which pages are really “old”/”unused”, there are several page replacement policies

Working Set Model



As a program executes it transitions through a sequence of "working sets" consisting of varying sized subsets of the address space

- OS keeps some amount of DRAM always free by deciding some upper cap
- Whenever the upper cap memory level is breached, unused pages are moved out to disk
  - OS uses a daemon process for carrying out this activity
- **Page replacement policies** are used to help decide which pages are really "old"

Cost of Demand Paging

- Average Memory Access Time (AMAT)
  - $MAT + P_{Miss} \times PF_{Time}$
  - Memory Access Time + Probability of Page Fault (a miss) x Page fault service time
- Example
  - Memory access time = 200 nanoseconds
  - Suppose p = Probability of page fault (miss)
  - Average page-fault service time = 8 milliseconds
  - Then, we can compute AMAT as follows:

$$AMAT = 200ns + p \times 8\ ms = 200ns + p \times 8{,}000{,}000ns$$

  - If one access out of 1,000 causes a page fault, then EAT = 8.2 μs

Page Replacement Policy (1/2)

- Why do we care about Replacement Policy?
  - Replacement is an issue with any cache
    - Recall, with demand paging we are using DRAM as a cache for keeping pages
  - Particularly important with pages
    - The cost of being wrong is high: must go to disk
    - Must keep important pages in memory, not toss them out

Page Replacement Policy (2/2)

- FIFO
  - Throw out oldest page
  - Bad – throws out heavily used pages instead of infrequently used
- Random
  - Pick random page for every replacement
  - Better than FIFO, but it goes by luck – no guarantees!
- MIN (Minimum) – a.k.a. the optimal policy
  - Replace page that will be used furthest in the future
  - But we can't really know future
- LRU (Least Recently Used)
  - Replace page that hasn't been used for the longest time
  - Programs have locality, so if something not used for a while, unlikely to be used in the near future

FIFO Policy

- Suppose we have the following page reference stream:
  - A B C A B D A D B C B
- Consider FIFO Page replacement:

Ref: / Page:	A	B	C	A	B	D	A	D	B	C	B
1	A					D				C	
2		B					A				
3			C						B		

- FIFO: 7 faults
- When referencing D, replacing A is bad choice, since we need A again right away

MIN Policy (= LRU in this Stream)

- Suppose we have the same reference **stream**:
  - A B C A B D A D B C B
- Consider MIN Page replacement:

Ref: Page:	A	B	C	A	B	D	A	D	B	C	B
1	A									C	
2		B									
3			C			D					

- MIN: 5 faults
  - Where will D be brought in? Look for page not referenced farthest in future
- LRU will also function exactly the same for this reference stream

LRU May Perform Bad Also..

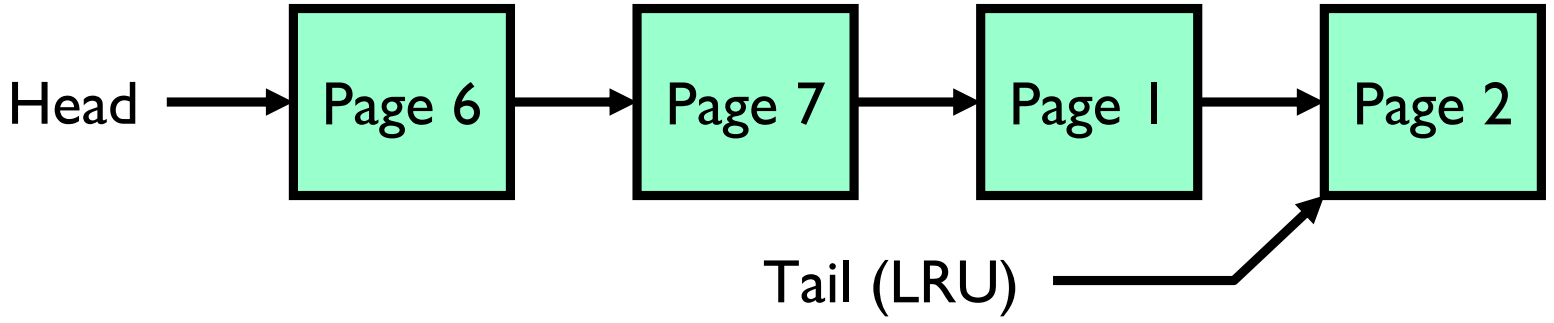
- Consider another reference stream: A B C D A B C D A B C D
- LRU Performs as follows (same as FIFO here):

Ref: Page:	A	B	C	D	A	B	C	D	A	B	C	D
1	A			D			C			B		
2		B			A			D			C	
3			C			B			A			D

  - Every reference is a page fault!

Implementing LRU

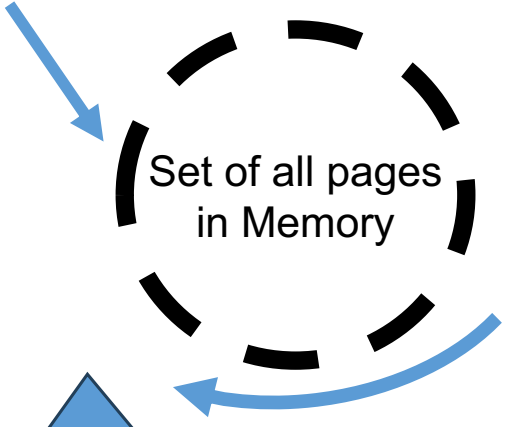
- LRU can be implemented using a list



- On each use, remove page from list and place at head
- LRU page is at tail
- Problems with this scheme for paging?
  - Need to know immediately when page used so that can change position in list
  - Lot of work required at each memory access for this implementation

Approximating LRU: Clock Algorithm

Single Clock Hand:
Advances only on page fault!
Check for pages not used recently
Mark pages as not used recently



There are other variants of clock algorithm, but we are not going to discuss them due to lack of time

- There is one "**use**" bit (hardware supported) per physical page (initially 0) and is set to 1 by the hardware on each memory reference on a page
- All the pages are arranged in a circular list
- A clock hand (single hand) points to any one of the page in this list
- At each page fault, advance the clock hand (not real time), and check the **use** bit
  - 1→ used recently; set to 0 and move to next page
  - 0→ selected candidate for replacement

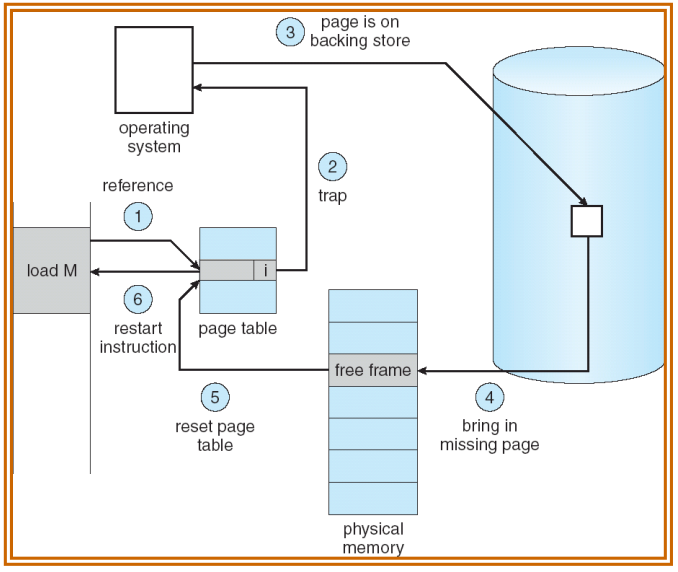
Next Lecture

- Introduction to multithreading
  - Start of a new module – concurrency!
    - Total 3 lectures

Lecture 21: Introduction to Multithreading

Vivek Kumar
Computer Science and Engineering
IIIT Delhi
vivekk@iiitd.ac.in

Last Lecture



Ref: / Page:	A	B	C	A	B	D	A	D	B	C	B
1	A					D				C	
2		B					A				
3			C						B		

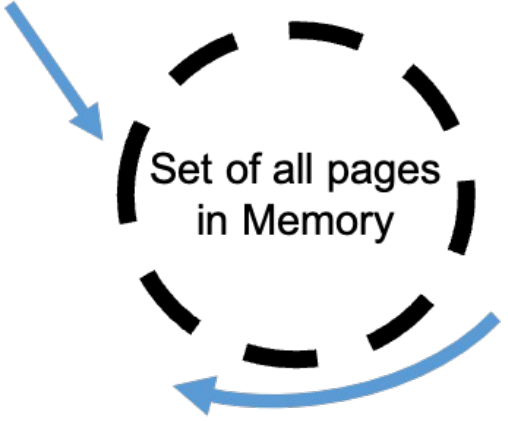
Ref: / Page:	A	B	C	A	B	D	A	D	B	C	B
1	A									C	
2		B									
3			C			D					

Single Clock Hand in Clock Algorithm:

Advances only on page fault!

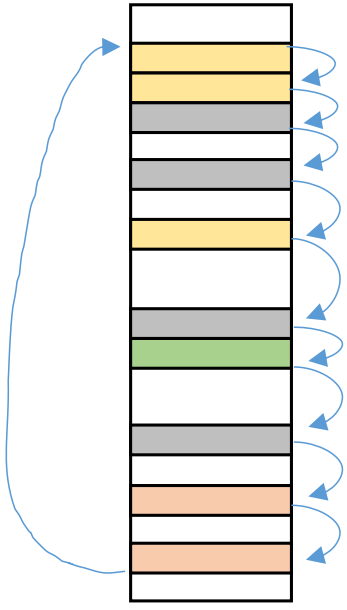
Check for pages not used recently

Mark pages as not used recently



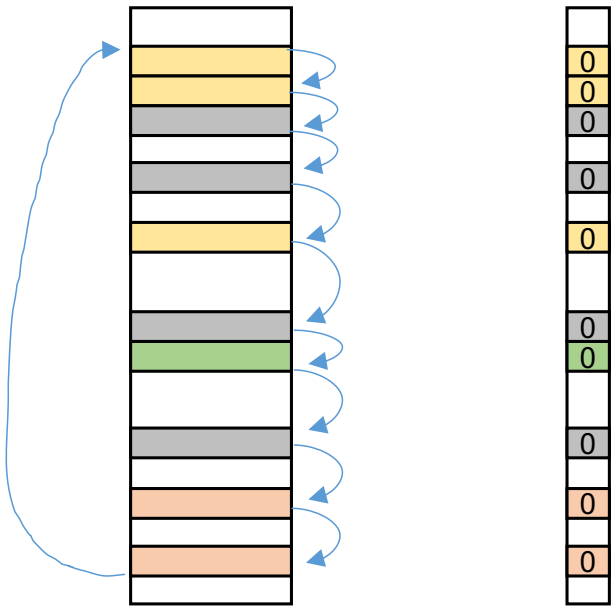
- Demand paging
  - Use DRAM as a cache for recently used pages and move the unused pages to disk
  - Bring back the page from disk to ram on demand (upon page fault)
- Page replacement policies (Suppose we have the following page reference stream: A B C A B D A D B C B)
  - FIFO
  - MIN / LRU
- Approximating LRU using clock algorithm (each page has a “use” bit)

Approximating LRU: Clock Algorithm (1/7)



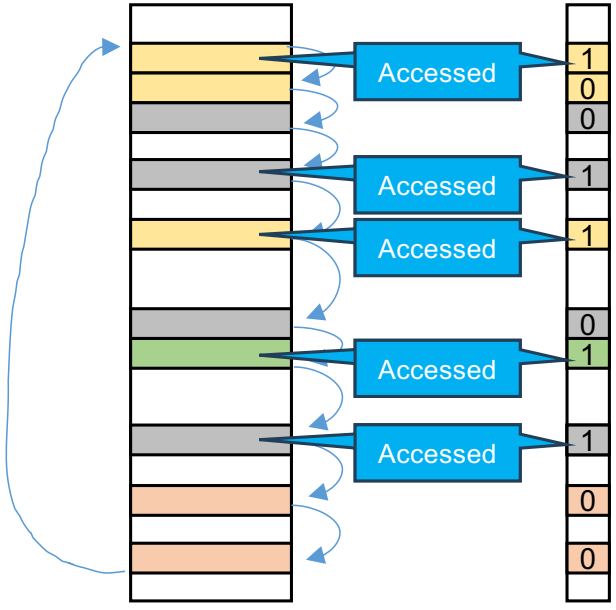
- Each physical pages (**used ones**) are arranged as circular linked list

Approximating LRU: Clock Algorithm (2/7)



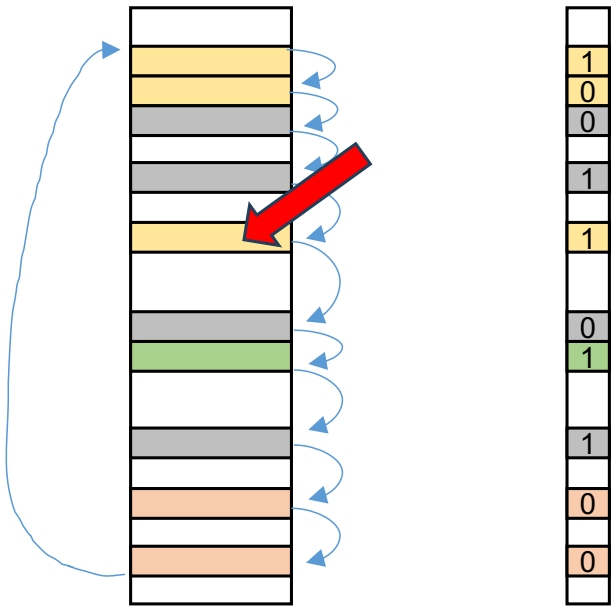
- Each physical pages (**used ones**) have a dedicated bit stored in RAM known as “use” bit
  - Imagine OS maintaining another array of “use” bits inside DRAM, where each index is a “use” bit for one dedicated physical page
- Values could be 0 or 1
  - Default value set to 0

Approximating LRU: Clock Algorithm (3/7)



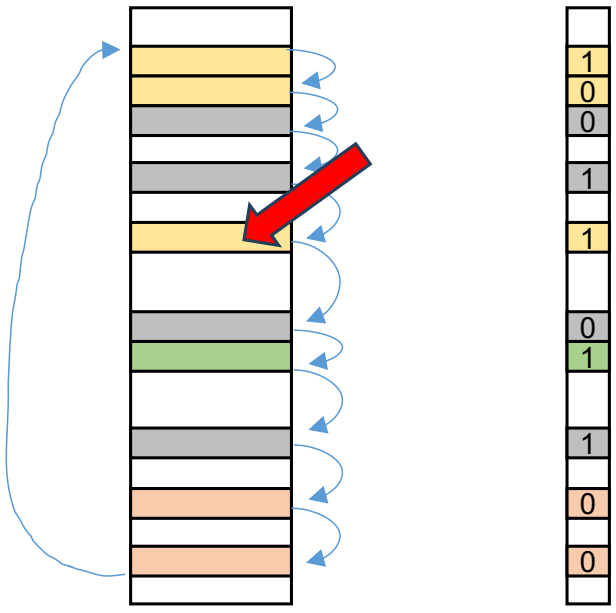
- Whenever a page is accessed (load/store of some address inside this page) by the process, the **hardware** changes its "use" bit to 1

Approximating LRU: Clock Algorithm (4/7)



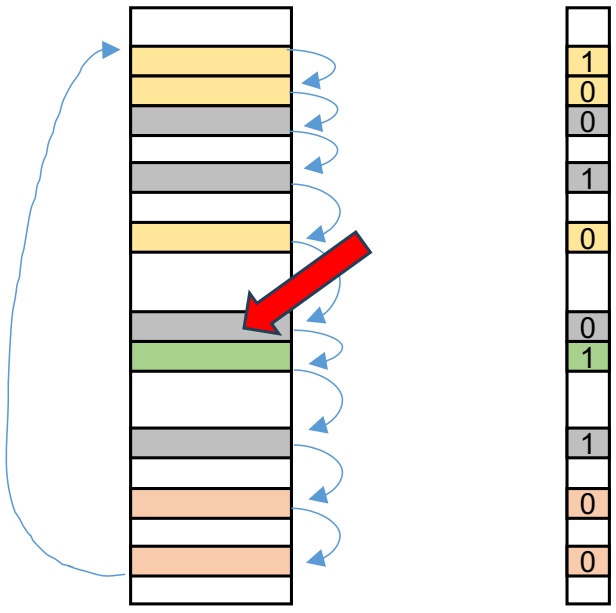
- A clock hand (single hand) points to any one of the page in this list
  - It doesn't matter which one
- This single clock hand advances to a next page in the list **only** upon a page fault for demand paging

Approximating LRU: Clock Algorithm (5/7)



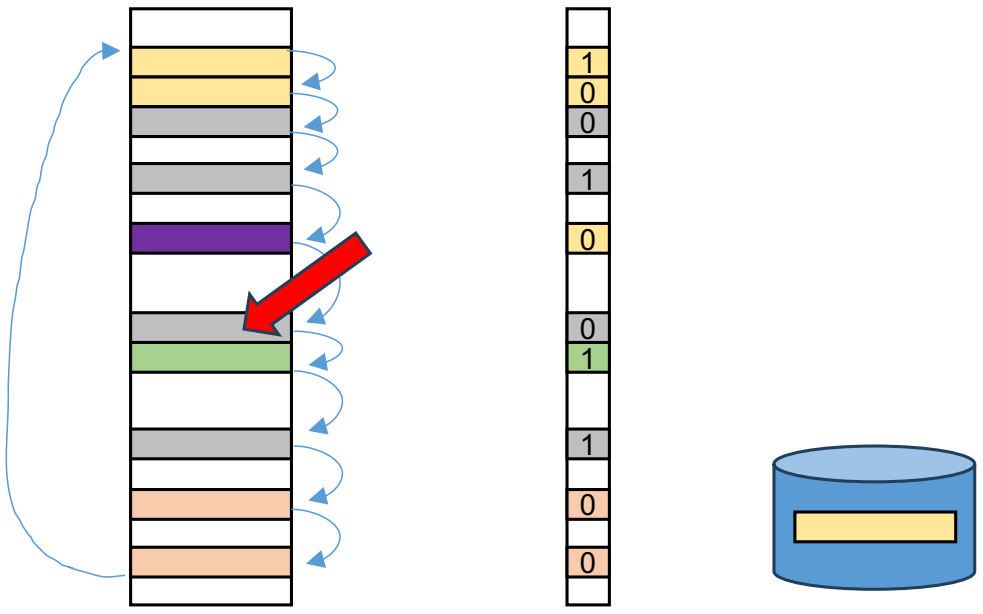
- At each page fault (request for demand paging), advance the clock hand (not real time), and check the **use** bit
  - **1→ used recently; set to 0 and move to next page**

Approximating LRU: Clock Algorithm (6/7)



- At each page fault (request for demand paging), advance the clock hand (not real time), and check the **use** bit
  - **0→ selected candidate for replacement**

Approximating LRU: Clock Algorithm (7/7)

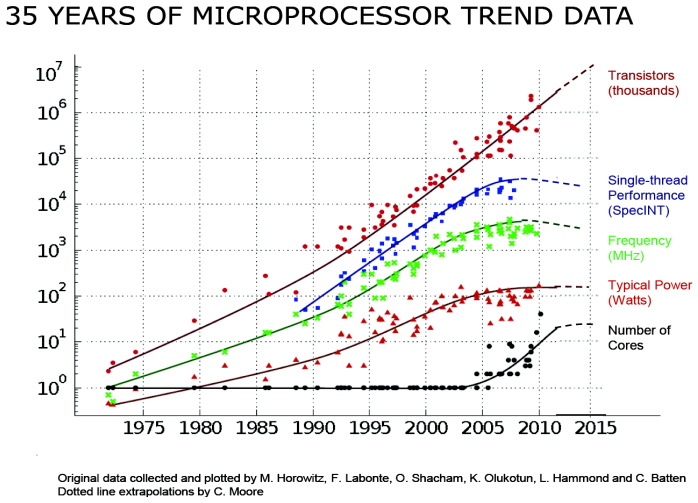


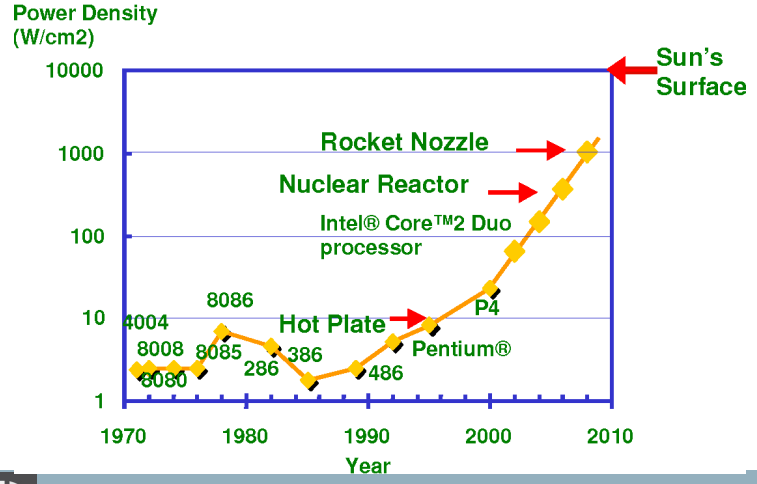
- The candidate page is moved to the disk and required page from the disk is brought into the newly created page slot on the DRAM
- There are several optimizations possible in this simple clock algorithm

Today's Class

- Why multicore processors
- Motivation for multithreading
- Thread creation and termination

Processor Technology Trend

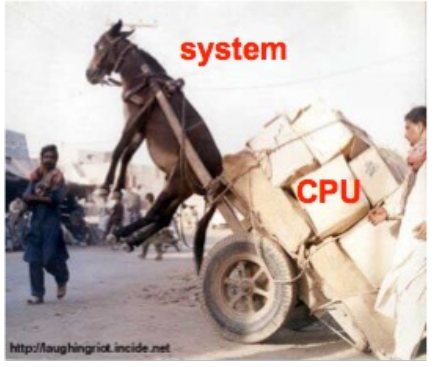


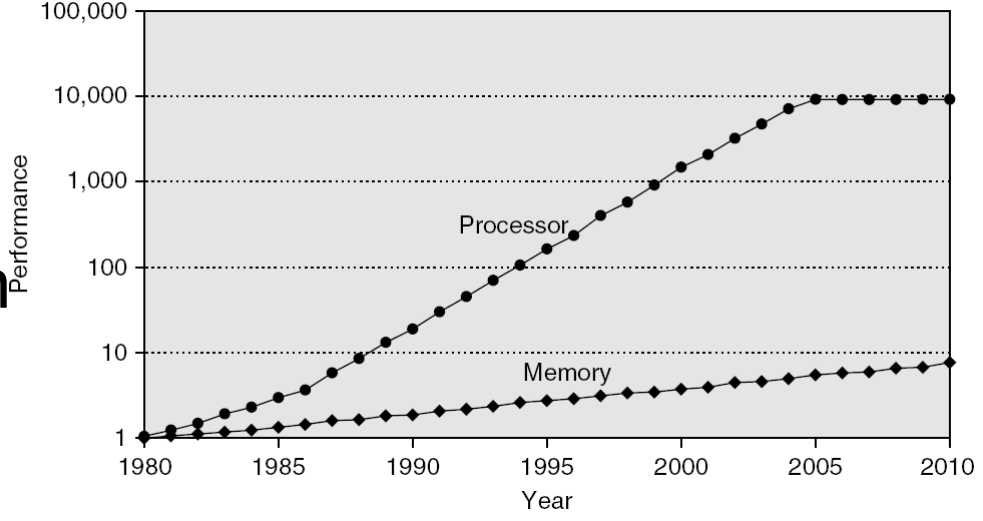


- Moore's law (1964)
  - Area of transistors halves roughly every two years
    - I.e., Total transistors on processor chip gets doubled roughly every two years
- Dennard scaling (1974)
  - Power for fixed chip are remains almost constant as transistors become smaller
- No more free lunch!
  - Thermal wall hit around 2004
  - Power is proportional to cube of frequency
    - It restricts frequency growth, but opens up the multicore era

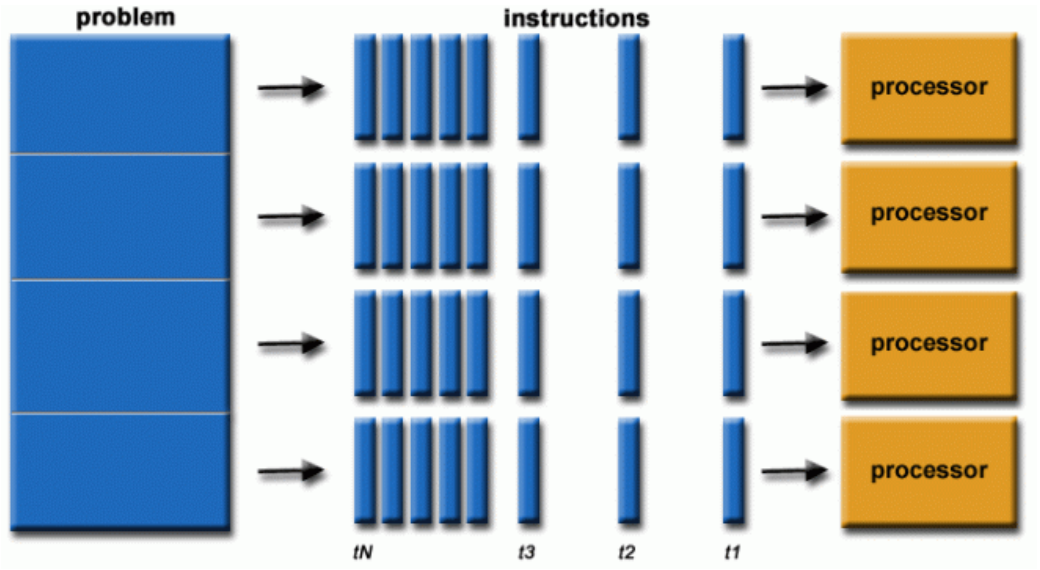
Adding More Cores Improves performance?

- Computation is just part of the picture
- Memory latency and bandwidth
  - CPU rates have increased 4x as fast as memory over last decade
  - Bridge speed gap using memory hierarchy
  - Multicore exacerbates demand
- Inter-processor communication
- Input/Output





Free Lunch is Over!



- Industrial and commercial users of parallel computing
  - BigData, Databases, Data mining
  - Artificial intelligence
  - Oil exploration
  - Web search engines, web based business services (Facebook, Twitter, etc.)
  - Medical imaging and diagnosis
  - Financial and economic modelling
  - Advanced graphics and virtual reality
  - Collaborative work environment

Let's Revisit Array Sum Program (Lecture #9)

```
int main(int argc, char **argv) {
    int rank=0, nproc=4;
    MPI_Init(&argc, &argv);
    // 1. Get to know your world
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &nproc);
    int array[SIZE]; // initialized and assume (SIZE % nproc = 0)
    // 2. calculate local sum
    int my_sum = 0, chunk = SIZE/nproc;
    for (int i=rank*chunk; i<(chunk+1)*rank; i++) my_sum += array[i];
    // 3. All non-root processes send result to root processes (rank=0)
    if(rank > 0) {
        MPI_Send(&my_sum, 1, MPI_INT, 0, 0, MPI_COMM_WORLD);
    }
    else { // executed only at rank=0
        int total_sum = my_sum, tmp;
        for(int src=1; src<nproc; src++) {
          MPI_Recv(&tmp, 1, MPI_INT, src, 0, MPI_COMM_WORLD, NULL);
          total_sum += tmp;
        }
    }
    MPI_Finalize();
}
```

- Note that it is an MPI program that is aimed to be run on a distributed memory machine (cluster / supercomputer)
- However, we can still run it on a single machine (laptop/desktop) with multicore processors
  - Total process equals to total cores

IPC Using MPI Within a Multicore Processor

- Inter-process communication in shared memory (only)
  - Transfer of control from user space to kernel space and vice-versa
- Complicated IPC mechanism for communication
- OS has to reserve extra memory / resources
  - Separate heap, stack, .text segment, etc. for each process
  - Same copy of .text segment in each process
- Separate page table for each process
- Cost of IPC may exceed the cost of actual computation!

Let's Revisit Array Sum Program (Lecture #8)

```
int main() {
    shm_t* shm = setup();
    sem_init(&shm->mutex, 1, 1);
    int chunks = SIZE/NPROCS;
    for(int i=0; i<NPROCS; i++) {
        if(fork()==0) {
            int local=0;
            int start = i*chunks;
            int end = start+chunk;
            for(int j=start; j<end; j++) local += shm->array[j];
            sem_wait(&shm->mutex);
            shm->sum += local;
            sem_post(&shm->mutex);
            cleanup_and_exit();
        }
    }
    for(int i=0; i<NPROCS; i++) wait(NULL);
    cleanup();
    return 0;
}
```

```
typedef struct shm_t {
    int A[SIZE];
    int sum[NPROCS];
    sem_t mutex;
} shm_t;
```

- Recall, we also parallelized the array sum program using normal IPC mechanisms
  - Inter-process communication using shared memory and semaphores

Same Program but without Semaphore

```
int main() {
    shm_t* shm = setup();

    int chunks = SIZE/NPROCS;
    for(int i=0; i<NPROCS; i++) {
        if(fork()==0) {
            int local=0;
            int start = i*chunks;
            int end = start+chunk;
            for(int j=start; j<end; j++) local += shm->array[j];

            shm->sum[i] = local;

            cleanup_and_exit();
        }
    }
    for(int i=0; i<NPROCS; i++) wait(NULL);
    int total=0; for(int j=0; j<NPROCS; j++) total += sum[j];
    cleanup();
    return 0;
}
```

```
typedef struct shm_t {
    int A[SIZE];
    int sum[NPROCS];

} shm_t;
```

- It is another version of the same program that doesn't use semaphores
  - Removing it just to simplify the motivation for today's topic

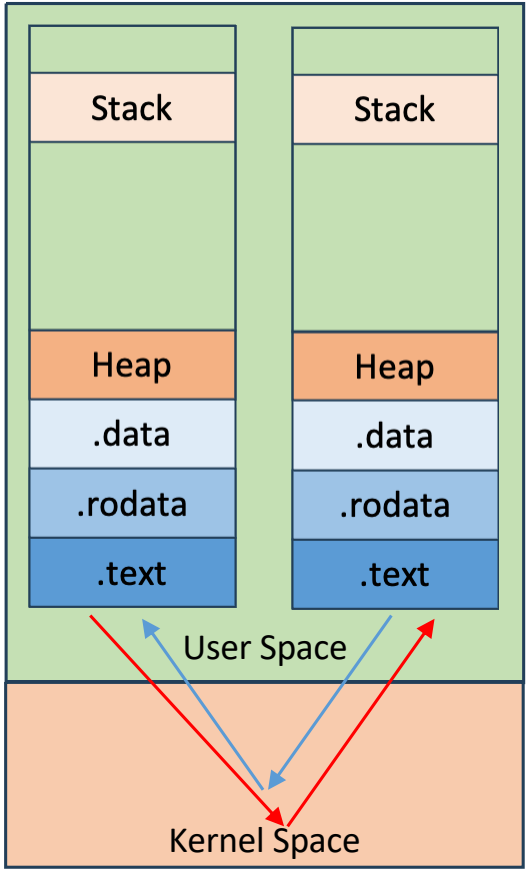
IPC Using SHM Within a Multicore Processor

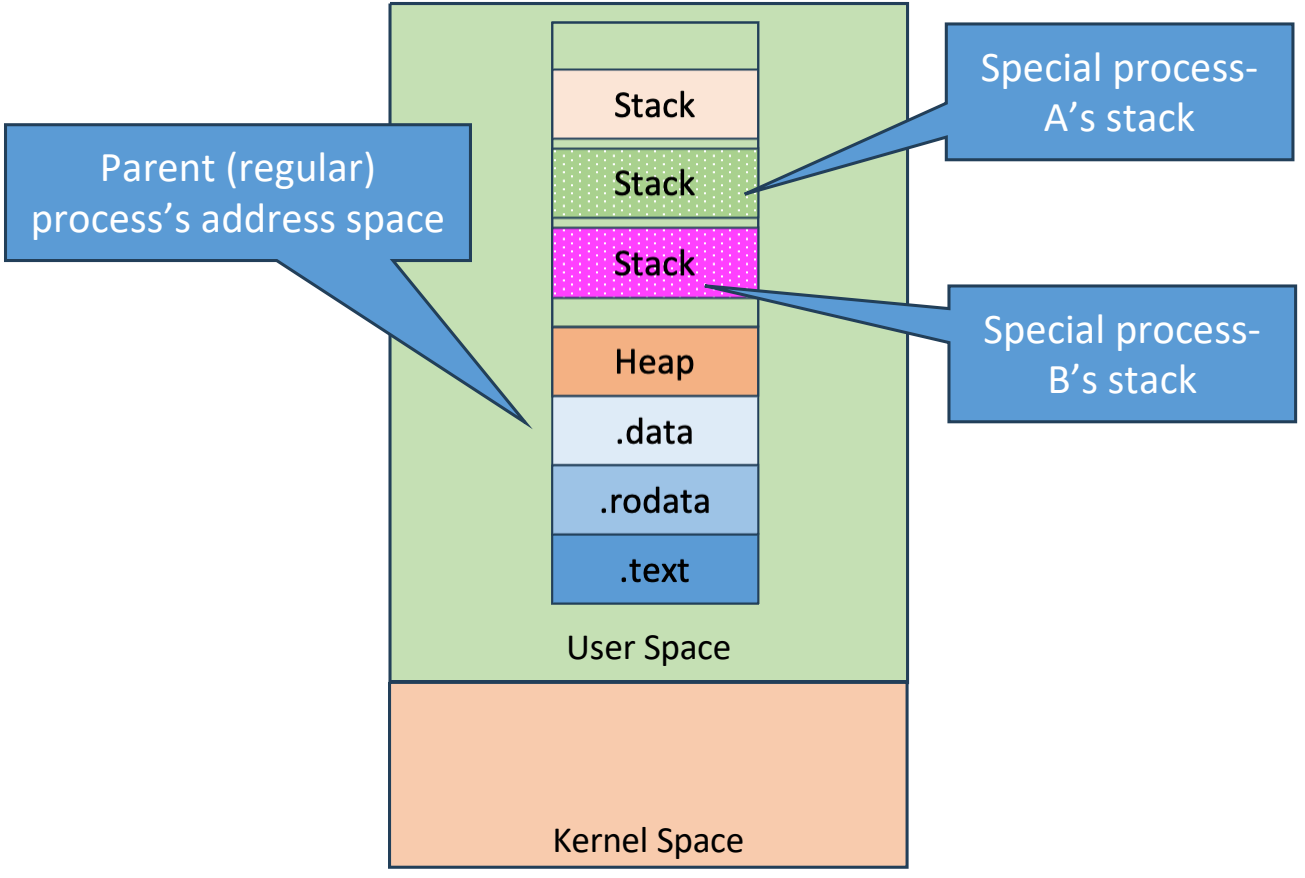
- Inter-process communication in shared memory (only)
  - Transfer of control from user space to kernel space and vice-versa
- Complicated IPC mechanism for communication
- OS has to reserve extra memory / resources
  - Same copy of .text segment in each process
- Separate page table for each process
- ~~Cost of IPC may exceed the cost of actual computation!~~
  - We can neglect it as the processes communicate in shared memory without using any explicit IPC calls after the setup of SHM

How to Fix the Issues we Highlighted?

- Asking the OS to allow creation of "**special**" processes that has **special** powers
- These **special** processes can be created by the parent process, and they all live in harmony like an **ideal** Indian joint family, where entire resources in the house are shared within the family members
  - Sharing of page table of the parent process
  - Sharing of parent's process address space
    - Shared heap, .text, .data segment, etc.
      - Stack cannot be shared as we need each of these "**special**" processes to execute a different method call chain inside the same program
      - Likewise, PC, registers also cannot be shared. Hence, they go through the same set of steps during context switch similar to **regular** processes
  - They can communicate with each other without using any special APIs or without going into the kernel space (zero overheads in communication!)

2 Regular Process v/s 2 Special Process





These "**special**" processes are called as **Threads**!

Thread Creation in Linux

```
//Asynchronously invoke func in a new thread
int pthread_create(
            //returned identifier for the new thread
            pthread_t *thread,

            //specifies the size of thread's stack and
            //how the thread should be scheduled by OS
            const pthread_attr_t *attr,

            //routine executed after creation
            void *(*func)(void *),

            //a single argument passed to func
            void *arg
) //returns error status
```

Waiting for Thread Termination in Linux

```
//Suspend execution of calling thread until thread
//terminates
int pthread_join(
  //identifier of thread to wait for
              pthread_t thread,

  //terminating thread's status (NULL to ignore)
              void **status
) //returns error status
```

Array Sum using Pthread

```c
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
int A[SIZE]; // Initialization code elided
int array_sum(int low, int high) {
  int sum = 0;
  for (int i=low; i<high; i++) {
    sum += A[i];
  }
  return sum;
}

typedef struct {
  int low;
  int high;
  int sum;
} thread_args;

void *thread_func(void *ptr) {
  thread_args * t = ((thread_args *) ptr);
  t->sum = array_sum(t->low, t->high);
  return NULL;
}
```

```c
int main(int argc, char *argv[]) {
  int result;
  if (SIZE < 1024) {
    result = array_sum(0, SIZE);
  } else {
    pthread_t tid[NTHREADS];
    thread_args args[NTHREADS];
    int chunk = SIZE/NTHREADS;
    for (int i=0; i<NTHREADS; i++) {
        args[i].low=i*chunk; args[i].high=(i+1)*chunk;
        pthread_create(&tid[i],
                       NULL,
                       thread_func,
                       (void*) &args[i]);
    }
    for (int i=0; i<NTHREADS; i++) {
        pthread_join(tid[i] , NULL);
        result += args[i].sum;
    }
  }
  printf("Total Sum is %d\n", result);
  return 0;
}
```

Array Sum using Pthread

```c
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
int A[SIZE]; // Initialization code elided
int array_sum(int low, int high) {
  int sum = 0;
  for (int i=low; i<high; i++) {
    sum += A[i];
  }
  return sum;
}

typedef struct {
  int low;
  int high;
  int sum;
} thread_args;

void *thread_func(void *ptr) {
  thread_args * t = ((thread_args *) ptr);
  t->sum = array_sum(t->low, t->high);
  return NULL;
}
```

Original code

```c
int main(int argc, char *argv[]) {
  int result;
  if (SIZE < 1024) {
    result = array_sum(0, SIZE);
  } else {
    pthread_t tid[NTHREADS];
    thread_args args[NTHREADS];
    int chunk = SIZE/NTHREADS;
    for (int i=0; i<NTHREADS; i++) {
        args[i].low=i*chunk; args[i].high=(i+1)*chunk;
        pthread_create(&tid[i],
                       NULL,
                       thread_func,
                       (void*) &args[i]);
    }
    for (int i=0; i<NTHREADS; i++) {
        pthread_join(tid[i] , NULL);
        result += args[i].sum;
    }
  }
  printf("Total Sum is %d\n", result);
  return 0;
}
```

Array Sum using Pthread

```
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
int A[SIZE]; // Initialization code elided
int array_sum(int low, int high) {
  int sum = 0;
  for (int i=low; i<high; i++) {
    sum += A[i];
  }
  return sum;
}

typedef struct {
  int low;
  int high;
  int sum;
} thread_args;

void *thread_func(void *ptr) {
  thread_args * t = ((thread_args *) ptr);
  t->sum = array_sum(t->low, t->high);
  return NULL;
}
```

Structure for thread arguments

```
int main(int argc, char *argv[]) {
  int result;
  if (SIZE < 1024) {
    result = array_sum(0, SIZE);
  } else {
    pthread_t tid[NTHREADS];
    thread_args args[NTHREADS];
    int chunk = SIZE/NTHREADS;
    for (int i=0; i<NTHREADS; i++) {
        args[i].low=i*chunk; args[i].high=(i+1)*chunk;
        pthread_create(&tid[i],
                       NULL,
                       thread_func,
                       (void*) &args[i]);
    }
    for (int i=0; i<NTHREADS; i++) {
        pthread_join(tid[i] , NULL);
        result += args[i].sum;
    }
  }
  printf("Total Sum is %d\n", result);
  return 0;
}
```

Array Sum using Pthread

```
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
int A[SIZE]; // Initialization code elided
int array_sum(int low, int high) {
  int sum = 0;
  for (int i=low; i<high; i++) {
    sum += A[i];
  }
  return sum;
}

typedef struct {
  int low;
  int high;
  int sum;
} thread_args;

void *thread_func(void *ptr) {
  thread_args * t = ((thread_args *) ptr);
  t->sum = array_sum(t->low, t->high);
  return NULL;
}
```

Function called when thread is created

```
int main(int argc, char *argv[]) {
  int result;
  if (SIZE < 1024) {
    result = array_sum(0, SIZE);
  } else {
    pthread_t tid[NTHREADS];
    thread_args args[NTHREADS];
    int chunk = SIZE/NTHREADS;
    for (int i=0; i<NTHREADS; i++) {
        args[i].low=i*chunk; args[i].high=(i+1)*chunk;
        pthread_create(&tid[i],
                       NULL,
                       thread_func,
                       (void*) &args[i]);
    }
    for (int i=0; i<NTHREADS; i++) {
        pthread_join(tid[i] , NULL);
        result += args[i].sum;
    }
  }
  printf("Total Sum is %d\n", result);
  return 0;
}
```

Array Sum using Pthread

```c
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
int A[SIZE]; // Initialization code elided
int array_sum(int low, int high) {
  int sum = 0;
  for (int i=low; i<high; i++) {
    sum += A[i];
  }
  return sum;
}

typedef struct {
  int low;
  int high;
  int sum;
} thread_args;

void *thread_func(void *ptr) {
  thread_args * t = ((thread_args *) ptr);
  t->sum = array_sum(t->low, t->high);
  return NULL;
}
```

```c
int main(int argc, char *argv[]) {
  int result;
  if (SIZE < 1024) {
    result = array_sum(0, SIZE);
  } else {
    pthread_t tid[NTHREADS];
    thread_args args[NTHREADS];
    int chunk = SIZE/NTHREADS;
    for (int i=0; i<NTHREADS; i++) {
        args[i].low=i*chunk; args[i].high=(i+1)*chunk;
        pthread_create(&tid[i],
                       NULL,
                       thread_func,
                       (void*) &args[i]);
    }
    for (int i=0; i<NTHREADS; i++) {
        pthread_join(tid[i] , NULL);
        result += args[i].sum;
    }
  }
  printf("Total Sum is %d\n", result);
  return 0;
}
```

No point in creating thread if there isn't enough to do

Array Sum using Pthread

```c
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
int A[SIZE]; // Initialization code elided
int array_sum(int low, int high) {
  int sum = 0;
  for (int i=low; i<high; i++) {
    sum += A[i];
  }
  return sum;
}

typedef struct {
  int low;
  int high;
  int sum;
} thread_args;

void *thread_func(void *ptr) {
  thread_args * t = ((thread_args *) ptr);
  t->sum = array_sum(t->low, t->high);
  return NULL;
}
```

```c
int main(int argc, char *argv[]) {
  int result;
  if (SIZE < 1024) {
    result = array_sum(0, SIZE);
  } else {
    pthread_t tid[NTHREADS];
    thread_args args[NTHREADS];
    int chunk = SIZE/NTHREADS;
    for (int i=0; i<NTHREADS; i++) {
        args[i].low=i*chunk; args[i].high=(i+1)*chunk;
        pthread_create(&tid[i],
                       NULL,
                       thread_func,
                       (void*) &args[i]);
    }
    for (int i=0; i<NTHREADS; i++) {
        pthread_join(tid[i] , NULL);
        result += args[i].sum;
    }
  }
  printf("Total Sum is %d\n", result);
  return 0;
}
```

Marshal input argument to thread

Array Sum using Pthread

```c
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
int A[SIZE]; // Initialization code elided
int array_sum(int low, int high) {
  int sum = 0;
  for (int i=low; i<high; i++) {
    sum += A[i];
  }
  return sum;
}

typedef struct {
  int low;
  int high;
  int sum;
} thread_args;

void *thread_func(void *ptr) {
  thread_args * t = ((thread_args *) ptr);
  t->sum = array_sum(t->low, t->high);
  return NULL;
}
```

Create threads to execute `array_sum`

```c
int main(int argc, char *argv[]) {
  int result;
  if (SIZE < 1024) {
    result = array_sum(0, SIZE);
  } else {
    pthread_t tid[NTHREADS];
    thread_args args[NTHREADS];
    int chunk = SIZE/NTHREADS;
    for (int i=0; i<NTHREADS; i++) {
        args[i].low=i*chunk; args[i].high=(i+1)*chunk;
        pthread_create(&tid[i],
                       NULL,
                       thread_func,
                       (void*) &args[i]);
    }
    for (int i=0; i<NTHREADS; i++) {
        pthread_join(tid[i] , NULL);
        result += args[i].sum;
    }
  }
  printf("Total Sum is %d\n", result);
  return 0;
}
```

Array Sum using Pthread

```c
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
int A[SIZE]; // Initialization code elided
int array_sum(int low, int high) {
  int sum = 0;
  for (int i=low; i<high; i++) {
    sum += A[i];
  }
  return sum;
}

typedef struct {
  int low;
  int high;
  int sum;
} thread_args;

void *thread_func(void *ptr) {
  thread_args * t = ((thread_args *) ptr);
  t->sum = array_sum(t->low, t->high);
  return NULL;
}
```

Main program blocks until threads terminate

```c
int main(int argc, char *argv[]) {
  int result;
  if (SIZE < 1024) {
    result = array_sum(0, SIZE);
  } else {
    pthread_t tid[NTHREADS];
    thread_args args[NTHREADS];
    int chunk = SIZE/NTHREADS;
    for (int i=0; i<NTHREADS; i++) {
        args[i].low=i*chunk; args[i].high=(i+1)*chunk;
        pthread_create(&tid[i],
                       NULL,
                       thread_func,
                       (void*) &args[i]);
    }
    for (int i=0; i<NTHREADS; i++) {
        pthread_join(tid[i] , NULL);
        result += args[i].sum;
    }
  }
  printf("Total Sum is %d\n", result);
  return 0;
}
```

Array Sum using Pthread

```c
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
int A[SIZE]; // Initialization code elided
int array_sum(int low, int high) {
  int sum = 0;
  for (int i=low; i<high; i++) {
    sum += A[i];
  }
  return sum;
}

typedef struct {
  int low;
  int high;
  int sum;
} thread_args;

void *thread_func(void *ptr) {
  thread_args * t = ((thread_args *) ptr);
  t->sum = array_sum(t->low, t->high);
  return NULL;
}
```

```c
int main(int argc, char *argv[]) {
  int result;
  if (SIZE < 1024) {
    result = array_sum(0, SIZE);
  } else {
    pthread_t tid[NTHREADS];
    thread_args args[NTHREADS];
    int chunk = SIZE/NTHREADS;
    for (int i=0; i<NTHREADS; i++) {
        args[i].low=i*chunk; args[i].high=(i+1)*chunk;
        pthread_create(&tid[i],
                       NULL,
                       thread_func,
                       (void*) &args[i]);
    }
    for (int i=0; i<NTHREADS; i++) {
        pthread_join(tid[i] , NULL);
        result += args[i].sum;
    }
  }
  printf("Total Sum is %d\n", result);
  return 0;
}
```

Add the results together to produce the final output

Advantages of Multithreading

- Responsiveness
  - Even if part of program is blocked or performing lengthy operation, multithreading allows the program to continue
- Economical resource sharing
  - Threads share memory and resources of their parent process which allows multiple tasks to be performed simultaneously inside the process
- Utilization of multicores
  - Easily scale on modern multicore processors

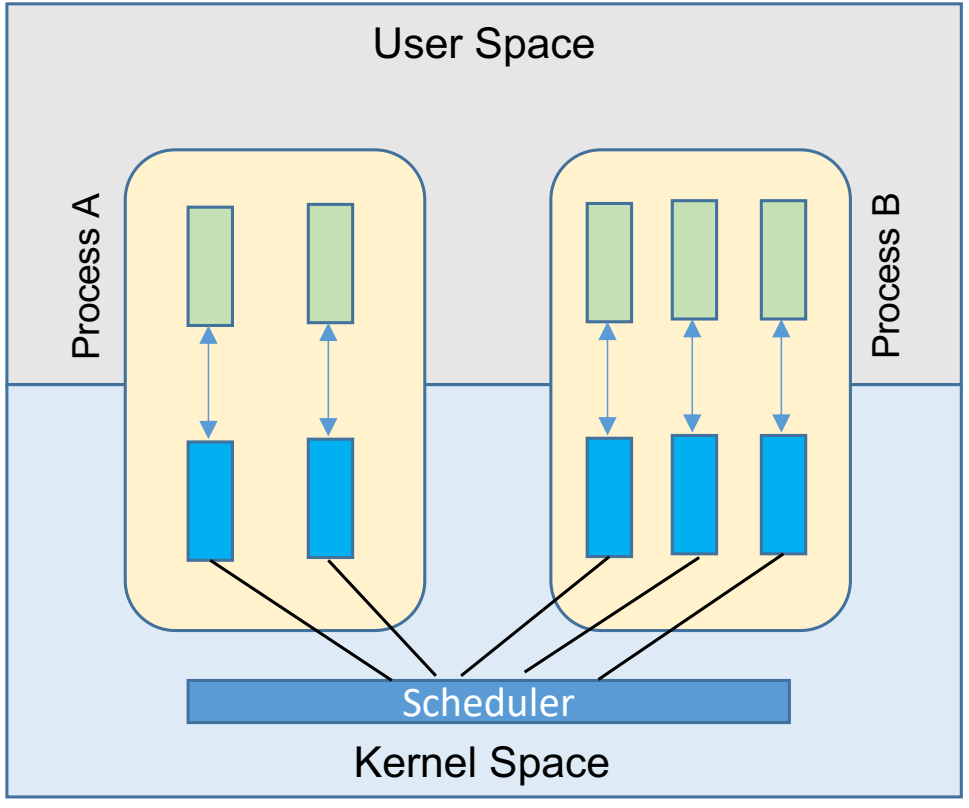
Thread Scheduling

- OS schedules threads that are ready to run independently, much like processes
- The context of a thread (PC, registers) is saved into/restored from thread control block (TCB)
  - Every PCB has one or more linked TCBs

Types of Threads

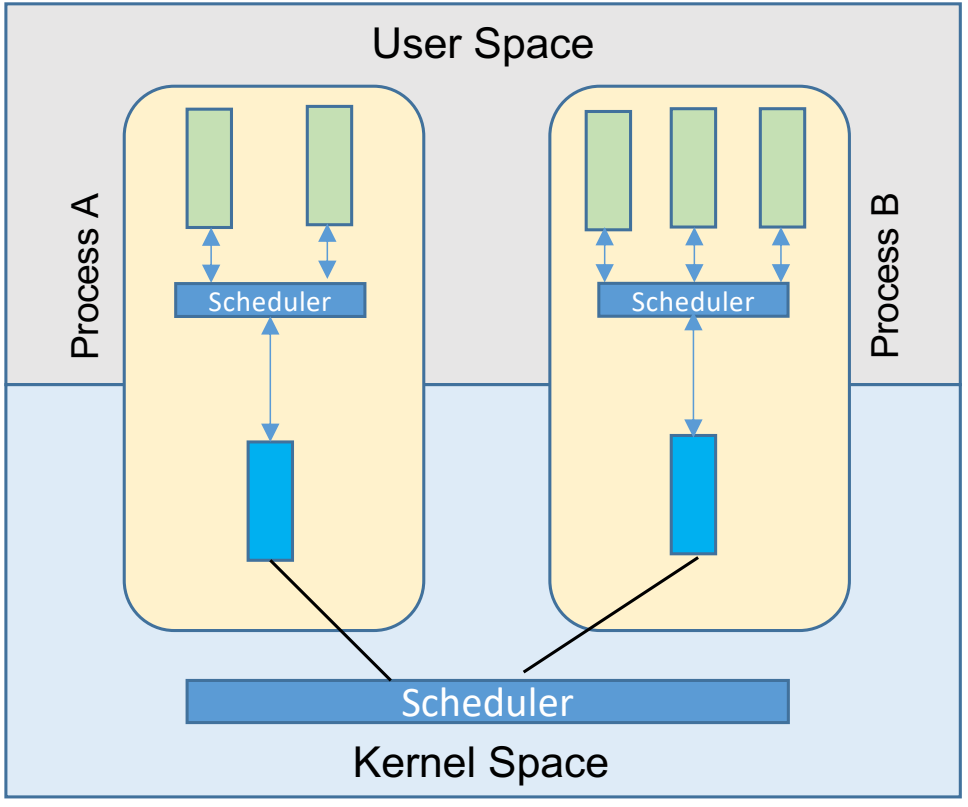
- **1x1** threading Model (Kernel Level Threads)
- **MxN** threading model (User Level Threads)

1x1 Threading Model



- Every thread created by the user has 1x1 mapping with the kernel thread
  - E.g., pthread library on Linux
- OS manages scheduling policy
- Switching between threads of same process much cheaper than switching between two processes
  - No need to switch address space (page table)
  - However, creating large number of threads would add to overheads
    - Cycles lost in thread creation
    - Frequent context switches

MxN Threading Model



- User gets to create several threads, but each of these threads can be mapped to a single kernel level thread
  - E.g., fibers in boost C++ library
- Runtime library (in user space) manages all thread operations (including scheduling policy)
  - Lightweight operations (OS is totally unaware of user level thread operations)
    - Cost of thread creation is low
    - Infrequent context switches due to lesser number of kernel threads than user threads
  - Covered in depth in CSE513 (Parallel Runtimes for Modern Processors)

Measures of parallel performance

- Speedup = $T_{serial}/T_{parallel}$
- Parallel efficiency = $T_{serial}/(pT_{parallel})$

Fig. source: http://www.drdobbs.com/cpp/going-superlinear/206100542

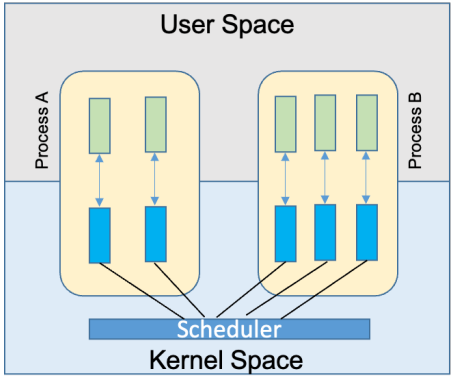
Next Lecture

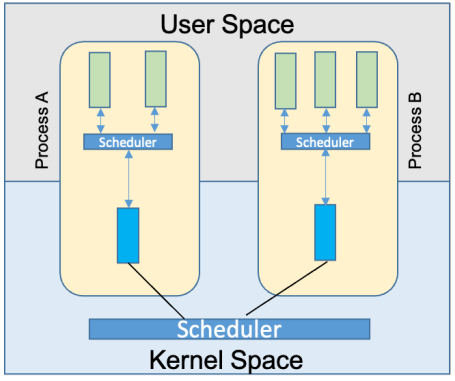
- Race condition in multithreading
- Assignment-5 will be released today with a deadline of one week (**No extensions**!)

Lecture 22: Race Condition in Multithreading

Vivek Kumar
Computer Science and Engineering
IIIT Delhi
vivekk@iiitd.ac.in

Last Lecture





- Inter-process communication in shared memory (only)
  - Transfer of control from user space to kernel space and vice-versa
- Complicated IPC mechanism for communication
- OS has to reserve extra memory / resources
  - Separate heap, stack, .text segment, etc. for each process
  - Same copy of .text segment in each process
- Separate page table for each process
- Cost of IPC may exceed the cost of actual computation!

```c
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
int A[SIZE]; // Initialization code elided
int array_sum(int low, int high) {
  int sum = 0;
  for (int i=low; i<high; i++) {
    sum += A[i];
  }
  return sum;
}

typedef struct {
  int low;
  int high;
  int sum;
} thread_args;

void *thread_func(void *ptr) {
  thread_args * t = ((thread_args *) ptr);
  t->sum = array_sum(t->low, t->high);
  return NULL;
}
```

```c
int main(int argc, char *argv[]) {
  int result;
  if (SIZE < 1024) {
    result = array_sum(0, SIZE);
  } else {
    pthread_t tid[NTHREADS];
    thread_args args[NTHREADS];
    int chunk = SIZE/NTHREADS;
    for (int i=0; i<NTHREADS; i++) {
        args[i].low=i*chunk; args[i].high=(i+1)*chunk;
        pthread_create(&tid[i],
                       NULL,
                       thread_func,
                       (void*) &args[i]);
    }
    for (int i=0; i<NTHREADS; i++) {
        pthread_join(tid[i] , NULL);
        result += args[i].sum;
    }
  }
  printf("Total Sum is %d\n", result);
  return 0;
}
```

Today's Class

- Mutual exclusion
- Condition variables
- Producer-consumer problem

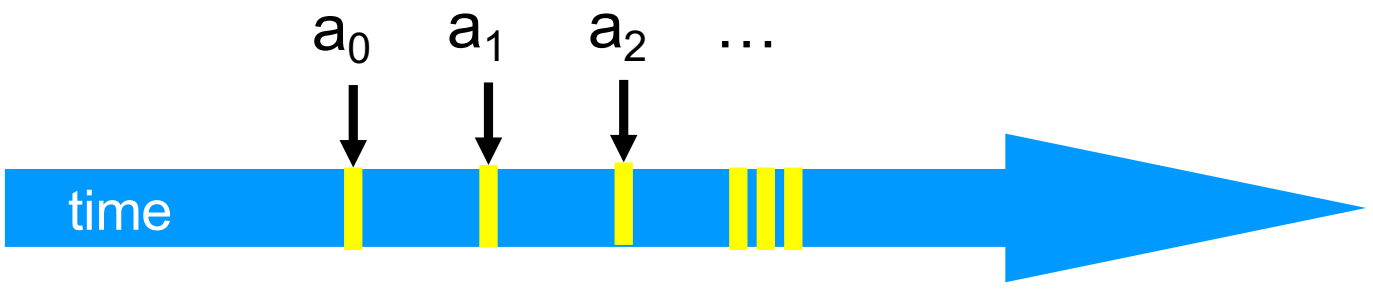
Mutual Exclusion

- ***Critical section:*** a block of code that access shared modifiable data or resource that should be operated on by only one thread at a time
- ***Mutual exclusion:*** a property that ensures that a critical section is only executed by a thread at a time.
  - *Otherwise it results in a race condition!*

Acknowledgement: Slides adopted from the companion slides for the book "The Art of Multiprocessor Programming" by Maurice Herlihy and Nir Shavit

Threads

- A ***thread*** A is (formally) a sequence a_0, a_1, ... of events
  - Notation: $a_0 \rightarrow a_1$ indicates order



Acknowledgement: Slides adopted from the companion slides for the book "The Art of Multiprocessor Programming" by Maurice Herlihy and Nir Shavit

Example Thread Events

- Assign to shared variable
- Assign to local variable
- Invoke method
- Return from method
- Lots of other things …

Acknowledgement: Slides adopted from the companion slides for the book "The Art of Multiprocessor Programming" by Maurice Herlihy and Nir Shavit

Concurrency

- Thread A

- Thread B

Acknowledgement: Slides adopted from the companion slides for the book "The Art of Multiprocessor Programming" by Maurice Herlihy and Nir Shavit

Interleavings

- Events of two or more threads
  - Interleaved
  - Not necessarily independent (why?)

Acknowledgement: Slides adopted from the companion slides for the book "The Art of Multiprocessor Programming" by Maurice Herlihy and Nir Shavit

Array Sum using Pthread (Version-1)

```c
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
int A[SIZE]; // Initialization code elided
int array_sum(int low, int high) {
  int sum = 0;
  for (int i=low; i<high; i++) {
    sum += A[i];
  }
  return sum;
}

typedef struct {
  int low;
  int high;
  int sum;
} thread_args;

void *thread_func(void *ptr) {
  thread_args * t = ((thread_args *) ptr);
  t->sum = array_sum(t->low, t->high);
  return NULL;
}
```

We went through this version of the parallel array sum using pthreads in the previous lecture

```c
int main(int argc, char *argv[]) {
  int result;
  if (SIZE < 1024) {
    result = array_sum(0, SIZE);
  } else {
    pthread_t tid[NTHREADS];
    thread_args args[NTHREADS];
    int chunk = SIZE/NTHREADS;
    for (int i=0; i<NTHREADS; i++) {
        args[i].low=i*chunk; args[i].high=(i+1)*chunk;
        pthread_create(&tid[i],
                       NULL,
                       thread_func,
                       (void*) &args[i]);
    }
    for (int i=0; i<NTHREADS; i++) {
        pthread_join(tid[i]);
        result += args[i].sum;
    }
  }
  printf("Total Sum is %d\n", result);
  return 0;
}
```

Array Sum using Pthread (Version-2)

```
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
int A[SIZE], result=0;
void array_sum(int low, int high) {
  int sum = 0;
  for (int i=low; i<high; i++) {
    sum += A[i];
  }
  return sum; result += sum;
}

typedef struct {
  int low;
  int high;
  int sum;
} thread_args;

void *thread_func(void *ptr) {
  thread_args * t = ((thread_args *) ptr);
  t->sum = array_sum(t->low, t->high);
  return NULL;
}
```

Any thread interleavings?

```
int main(int argc, char *argv[]) {
  int result;
  if (SIZE < 1024) {
    result = array_sum(0, SIZE);
  } else {
    pthread_t tid[NTHREADS];
    thread_args args[NTHREADS];
    int chunk = SIZE/NTHREADS;
    for (int i=0; i<NTHREADS; i++) {
        args[i].low=i*chunk; args[i].high=(i+1)*chunk;
        pthread_create(&tid[i],
                       NULL,
                       thread_func,
                       (void*) &args[i]);
    }
    for (int i=0; i<NTHREADS; i++)
        pthread_join(tid[i]);
        result += args[i].sum;
    }
  }
  printf("Total Sum is %d\n", result);
  return 0;
}
```

Do you see any issues in this version of the parallel array sum?

Array Sum using Pthread (Version-2)

```
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
int A[SIZE], result=0;
void array_sum(int low, int high) {
  int sum = 0;
  for (int i=low; i<high; i++) {
    sum += A[i];
  }
  return sum; result += sum;
}

typedef struct {
  int low;
  int high;
  int sum;
} thread_args;

void *thread_func(void *ptr) {
  thread_args * t = ((thread_args *) ptr);
  t->sum = array_sum(t->low, t->high);
  return NULL;
}
```

Race condition !!!

```
int main(int argc, char *argv[]) {
  int result;
  if (SIZE < 1024) {
    result = array_sum(0, SIZE);
  } else {
    pthread_t tid[NTHREADS];
    thread_args args[NTHREADS];
    int chunk = SIZE/NTHREADS;
    for (int i=0; i<NTHREADS; i++) {
        args[i].low=i*chunk; args[i].high=(i+1)*chunk;
        pthread_create(&tid[i],
                       NULL,
                       thread_func,
                       (void*) &args[i]);
    }
    for (int i=0; i<NTHREADS; i++) {
        pthread_join(tid[i]);
        result += args[i].sum;
    }
  }
  printf("Total Sum is %d\n", result);
  return 0;
}
```

Critical Sections and Mutual Exclusion

- Critical section is the code (block of code) that should be executed by only one thread at a time
- **Mutex locks** enforce mutual exclusion in Pthreads
  - Mutex lock states: locked and unlocked
  - Only one thread can lock a mutex lock at any particular time
- Using mutex locks
  - Request lock before executing critical section
  - Enter critical section when lock granted
  - Release lock when leaving critical section

Pthread Mutex Locks

- Initialize the mutex variable (statically)
  - `pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;`
- Lock the mutex
  - `pthread_mutex_lock(&mutex);`
- Unlock the mutex
  - `pthread_mutex_unlock(&mutex);`

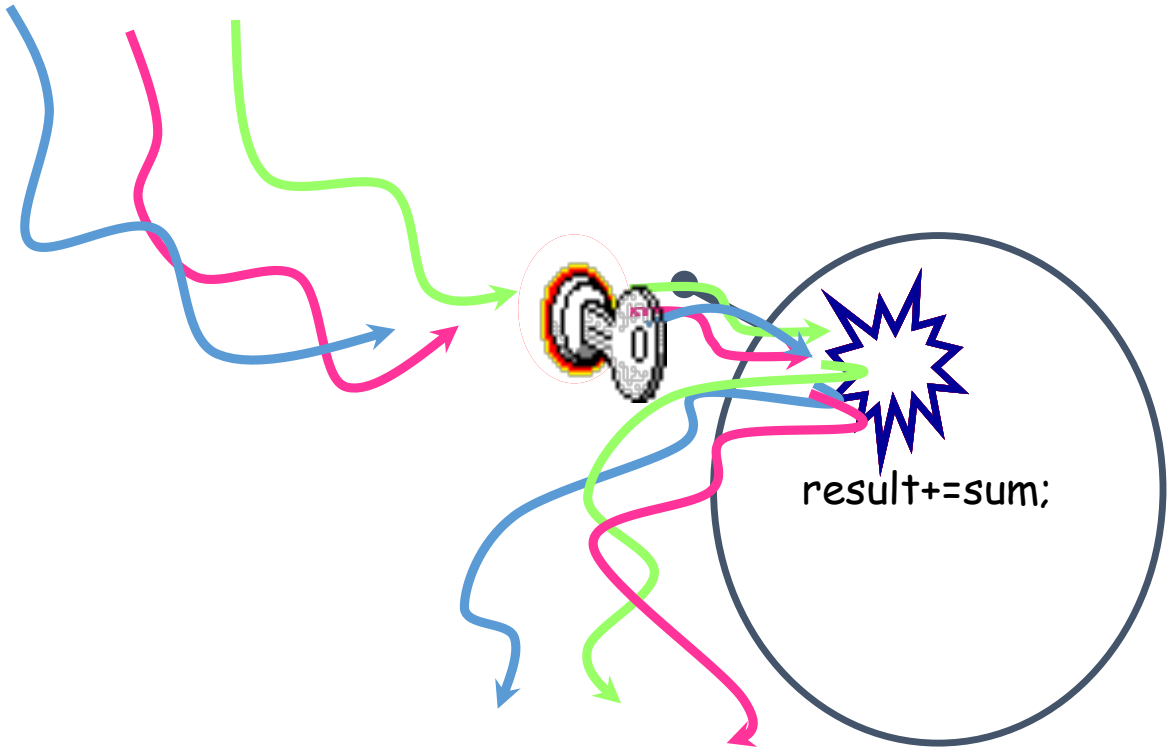
Array Sum using Pthread (Version-2)

```c
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
int A[SIZE], result=0;
pthread_mutex_t m=PTHREAD_MUTEX_INITIALIZER;
void array_sum(int low, int high) {
  int sum = 0;
  for (int i=low; i<high; i++) {
    sum += A[i];
  }
  pthread_mutex_lock(&m);
  result += sum;
  pthread_mutex_unlock(&m);
}

typedef struct {
  int low;
  int high;
} thread_args;

void *thread_func(void *ptr) {
  thread_args * t = ((thread_args *) ptr);
  array_sum(t->low, t->high);
  return NULL;
}
```

Race condition is fixed using mutual exclusion

```c
int main(int argc, char *argv[]) {
  if (SIZE < 1024) {
    array_sum(0, SIZE);
  } else {
    pthread_t tid[NTHREADS];
    thread_args args[NTHREADS];
    int chunk = SIZE/NTHREADS;
    for (int i=0; i<NTHREADS; i++) {
        args[i].low=i*chunk; args[i].high=(i+1)*chunk;
        pthread_create(&tid[i],
                       NULL,
                       thread_func,
                       (void*) &args[i]);
    }
    for (int i=0; i<NTHREADS; i++) {
        pthread_join(tid[i]);
    }
  }
  printf("Total Sum is %d\n", result);
  return 0;
}
```

Visualizing Mutual Exclusion



- Only one **thread** can get the “key” to enter the critical section
- Rest all **threads** will be queued to get the key

The Issue with the Locks

- Threads waiting for acquiring the lock continue wasting CPU cycles while spinning to acquire lock currently owned by another thread
  - Bigger problem if there are several threads waiting to acquire the same lock
- Solution
  - Instead of blocking, try (attempt) to acquire the lock
  - If failed to acquire then do something else
  -
```
if(pthread_mutex_trylock(&mutex) == 0) {
    /* Critical Section */
} else {
    /* Do something else */
}
```

Condition Variables for Synchronization

- **Condition variable: associated with a predicate and a mutex**
- Using a condition variable
  - thread can block itself until a condition becomes true
    - thread locks a mutex
    - tests a predicate defined on a shared variable

      if predicate is false, then wait on the condition variable

      waiting on condition variable unlocks associated mutex
  - when some thread makes a predicate true
    - that thread can signal the condition variable to either

      wake one waiting thread

      wake all waiting threads
    - when thread releases the mutex, it is passed to first waiter

Pthread Condition Variable APIs

```
/* initialize a condition variable (statically) */
pthread_cond_t cond = PTHREAD_COND_INITIALIZER;

/* block until a condition is true */
int pthread_cond_wait(pthread_cond_t *cond, pthread_mutex_t *mutex);

/* signal one or all waiting threads that condition is true */
int pthread_cond_signal(pthread_cond_t *cond);        // Wake one thread
int pthread_cond_broadcast(pthread_cond_t *cond);     // Wake all threads
```

Wait/Notify Sequence in Pthread

```
pthread_mutex_lock(&mutex);
while(task_queue_size() == 0)
  pthread_cond_wait(&cond, &mutex);
}
task = pop_task_queue();
pthread_mutex_unlock(&mutex);
execute_task (task);
```

Consumer(s)

```
pthread_mutex_lock(&mutex);
int queue_size = task_queue_size();
push_task_queue(&task);
if(queue_size == 0) {
  pthread_cond_broadcast(&cond);
}
pthread_mutex_unlock(&mutex);
```

Producer

© Vivek Kumar

Wait/Notify Sequence in Pthread

```
pthread_mutex_lock(&mutex);
while(task_queue_size() == 0)
  pthread_cond_wait(&cond, &mutex);
}
task = pop_task_queue();
pthread_mutex_unlock(&mutex);
execute_task (task);
```

Consumer Thread

```
pthread_mutex_lock(&mutex);
int queue_size = task_queue_size();
push_task_queue(&task);
if(queue_size == 0) {
  pthread_cond_broadcast(&cond);
}
pthread_mutex_unlock(&mutex);
```


© Vivek Kumar

Wait/Notify Sequence in Pthread

Mutex lock

```
pthread_mutex_lock(&mutex);
while(task_queue_size() == 0)
  pthread_cond_wait(&cond, &mutex);
}
task = pop_task_queue();
pthread_mutex_unlock(&mutex);
execute_task (task);
```

```
pthread_mutex_lock(&mutex);
int queue_size = task_queue_size();
push_task_queue(&task);
if(queue_size == 0) {
  pthread_cond_broadcast(&cond);
}
pthread_mutex_unlock(&mutex);
```

Consumer Thread

© Vivek Kumar

Wait/Notify Sequence in Pthread

Mutex lock

```
pthread_mutex_lock(&mutex);
while(task_queue_size() == 0)
  pthread_cond_wait(&cond, &mutex);
}
task = pop_task_queue();
pthread_mutex_unlock(&mutex);
execute_task (task);
```

```
pthread_mutex_lock(&mutex);
int queue_size = task_queue_size();
push_task_queue(&task);
if(queue_size == 0) {
  pthread_cond_broadcast(&cond);
}
pthread_mutex_unlock(&mutex);
```

Consumer Thread

© Vivek Kumar

Wait/Notify Sequence in Pthread

```
pthread_mutex_lock(&mutex);
while(task_queue_size() == 0)
  pthread_cond_wait(&cond, &mutex);
}
task = pop_task_queue();
pthread_mutex_unlock(&mutex);
execute_task (task);
```

Consumer Thread

```
pthread_mutex_lock(&mutex);
int queue_size = task_queue_size();
push_task_queue(&task);
if(queue_size == 0) {
  pthread_cond_broadcast(&cond);
}
pthread_mutex_unlock(&mutex);
```

Producer Thread

© Vivek Kumar

Wait/Notify Sequence in Pthread

Mutex lock

```
pthread_mutex_lock(&mutex);
while(task_queue_size() == 0)
  pthread_cond_wait(&cond, &mutex);
}
task = pop_task_queue();
pthread_mutex_unlock(&mutex);
execute_task (task);
```

Consumer Thread

```
pthread_mutex_lock(&mutex);
int queue_size = task_queue_size();
push_task_queue(&task);
if(queue_size == 0) {
  pthread_cond_broadcast(&cond);
}
pthread_mutex_unlock(&mutex);
```

Producer Thread

© Vivek Kumar

Wait/Notify Sequence in Pthread

```
pthread_mutex_lock(&mutex);
while(task_queue_size() == 0)
  pthread_cond_wait(&cond, &mutex);
}
task = pop_task_queue();
pthread_mutex_unlock(&mutex);
execute_task (task);
```

```
pthread_mutex_lock(&mutex);
int queue_size = task_queue_size();
push_task_queue(&task);
if(queue_size == 0) {
  pthread_cond_broadcast(&cond);
}
pthread_mutex_unlock(&mutex);
```

Consumer Thread

Producer Thread

© Vivek Kumar

Wait/Notify Sequence in Pthread

Mutex lock

```
pthread_mutex_lock(&mutex);
while(task_queue_size() == 0)
  pthread_cond_wait(&cond, &mutex);
}
task = pop_task_queue();
pthread_mutex_unlock(&mutex);
execute_task (task);
```

Consumer Thread

```
pthread_mutex_lock(&mutex);
int queue_size = task_queue_size();
push_task_queue(&task);
if(queue_size == 0) {
  pthread_cond_broadcast(&cond);
}
pthread_mutex_unlock(&mutex);
```

Producer Thread

© Vivek Kumar

Wait/Notify Sequence in Pthread

```
pthread_mutex_lock(&mutex);
while(task_queue_size() == 0)
  pthread_cond_wait(&cond, &mutex);
}
task = pop_task_queue();
pthread_mutex_unlock(&mutex);
execute_task (task);
```

Consumer Thread

```
pthread_mutex_lock(&mutex);
int queue_size = task_queue_size();
push_task_queue(&task);
if(queue_size == 0) {
  pthread_cond_broadcast(&cond);
}
pthread_mutex_unlock(&mutex);
```

Producer Thread

© Vivek Kumar

Wait/Notify Sequence in Pthread

Mutex lock

```
pthread_mutex_lock(&mutex);
while(task_queue_size() == 0)
  pthread_cond_wait(&cond, &mutex);
}
task = pop_task_queue();
pthread_mutex_unlock(&mutex);
execute_task (task);
```

```
pthread_mutex_lock(&mutex);
int queue_size = task_queue_size();
push_task_queue(&task);
if(queue_size == 0) {
  pthread_cond_broadcast(&cond);
}
pthread_mutex_unlock(&mutex);
```

Consumer Thread

Producer Thread

© Vivek Kumar

Wait/Notify Sequence in Pthread

Mutex lock

```
pthread_mutex_lock(&mutex);
while(task_queue_size() == 0)
  pthread_cond_wait(&cond, &mutex);
}
task = pop_task_queue();
pthread_mutex_unlock(&mutex);
execute_task (task);
```

Consumer Thread

```
pthread_mutex_lock(&mutex);
int queue_size = task_queue_size();
push_task_queue(&task);
if(queue_size == 0) {
  pthread_cond_broadcast(&cond);
}
pthread_mutex_unlock(&mutex);
```

Producer Thread

© Vivek Kumar

Next Lecture

- Producer consumer (contd.) and deadlock avoidance
  - Last remaining topic in multithreading

Lecture 23: Deadlock Avoidance

Vivek Kumar
Computer Science and Engineering
IIIT Delhi
vivekk@iiitd.ac.in

Last Lecture

```
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
int A[SIZE], result=0;
void array_sum(int low, int high) {
  int sum = 0;
  for (int i=low; i<high; i++) {
    sum += A[i];
  }
  return sum;  result += sum;
}

typedef struct {
  int low;
  int high;
  int sum;
} thread_args;

void *thread_func(void *ptr) {
  thread_args * t = ((thread_args *) ptr);
  t->sum = array_sum(t->low, t->high);
  return NULL;
}
```

Race condition !!!

```
int main(int argc, char *argv[]) {
  int result;
  if (SIZE < 1024) {
    result = array_sum(0, SIZE);
  } else {
    pthread_t tid[NTHREADS];
    thread_args args[NTHREADS];
    int chunk = SIZE/NTHREADS;
    for (int i=0; i<NTHREADS; i++) {
        args[i].low=i*chunk; args[i].high=(i+1)*chunk;
        pthread_create(&tid[i],
                       NULL,
                       thread_func,
                       (void*) &args[i]);
    }
    for (int i=0; i<NTHREADS; i++) {
        pthread_join(tid[i]);
        result += args[i].sum;
    }
  }
  printf("Total Sum is %d\n", result);
  return 0;
}
```

- Race condition
- Producer consumer problem

```
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
int A[SIZE], result=0;
pthread_mutex_t m=PTHREAD_MUTEX_INITIALIZER;
void array_sum(int low, int high) {
  int sum = 0;
  for (int i=low; i<high; i++) {
    sum += A[i];
  }
  pthread_mutex_lock(&m);
  result += sum;
  pthread_mutex_unlock(&m);
}
```

Race condition is fixed using mutual exclusion

```
1. pthread_mutex_lock(&mutex);
2. while(task_queue_size() == 0)
3.   pthread_cond_wait(&cond, &mutex);
4. }
5. task = pop_task_queue();
6. pthread_mutex_unlock(&mutex);
7. execute_task (task);
```

Consumer(s)

```
1. pthread_mutex_lock(&mutex);
2. int queue_size = task_queue_size();
3. push_task_queue(&task);
4. if(queue_size == 0) {
5.   pthread_cond_broadcast(&cond);
6. }
7. pthread_mutex_unlock(&mutex);
```

Producer

Today's Class

- Properties of good locking algorithm
- Deadlock creation
- Deadlock avoidance
- Out of syllabus discussion

Properties of a Good Locking Algorithm

- Safety guarantee
  - Mutual exclusion
- Progress guarantee
  - Deadlock freedom
  - Starvation freedom

Acknowledgement: Slides adopted from the companion slides for the book "The Art of Multiprocessor Programming" by Maurice Herlihy and Nir Shavit

Properties of a Good Locking Algorithm

- Mutual exclusion
- ***Deadlock freedom:*** *system as a whole makes progress.* If some thread calls **lock()** and never returns, then other threads must complete **lock()** and **unlock()** calls infinitely often.
- Starvation freedom

Acknowledgement: Slides adopted from the companion slides for the book "The Art of Multiprocessor Programming" by Maurice Herlihy and Nir Shavit

Properties of a Good Locking Algorithm

- Mutual exclusion
- Deadlock freedom: *system as a whole makes progress.* If some thread calls **lock()** and never returns, then other threads must complete **lock()** and **unlock()** calls infinitely often.
- ***Starvation freedom*** : A thread should not indefinitely hold the lock for doing some big computation while other threads keep waiting to get this lock

Acknowledgement: Slides adopted from the companion slides for the book "The Art of Multiprocessor Programming" by Maurice Herlihy and Nir Shavit

Money Transaction Between Accounts

```
class Account {
  int id;
  double balance;
  void debit(double amount);
  void credit(double amount);
};
```

```
class Transfer {
  Account source, destination;
  double amount;
  void run() {
    source.debit(amount);
    destination.credit(amount);
  }
};
```

```
class Bank {
  void fund_transfer() {
    Accounts numAccounts[N];
    Transfer pending[TOTAL];
    for(int i=0; i<TOTAL; i++) {
      pending[i].run();
    }
  }
};
```

- How to parallelize?
  - The for-loop is similar to the for-loop in parallel array sum we discussed in last lecture
  - Parallelize using multithreading

Money Transaction Between Accounts

```
class Account {
  int id;
  double balance;
  void debit(double amount);
  void credit(double amount);
};
```

```
class Transfer {
  Account source, destination;
  double amount;
  void run() {
    source.debit(amount);
    destination.credit(amount);
  }
};
```

```
class Bank {
  void fund_transfer() {
    Accounts numAccounts[N];
    Transfer pending[TOTAL];
    parallel_for(int i=0; i<TOTAL; i++) {
      pending[i].run();
    }
  }
};
```

- **parallel_for**
  - Shorthand to denote parallelization approach similar to parallel array sum (as in your assignment-5)
- Do you see any issues?
  - Race condition !!

Money Transaction Between Accounts

```
class Account {
  int id;
  double balance;
  void debit(double amount);
  void credit(double amount);
};
```

```
class Bank {
  void fund_transfer() {
    Accounts numAccounts[N];
    Transfer pending[TOTAL];
    parallel_for(int i=0; i<TOTAL; i++) {
      pending[i].run();
    }
  }
};
```

```
pthread_mutex_t m = PTHREAD_MUTEX_INITIALIZER;
class Transfer {
  Account source, destination;
  double amount;
  void run() {
    pthread_mutex_lock(&mutex);
    source.debit(amount);
    destination.credit(amount);
    pthread_mutex_unlock(&mutex);
  }
};
```

- We can use mutex lock to fix race condition
- Do we still have parallelism?

Money Transaction Between Accounts

```
class Account {
  int id;
  double balance;
  pthread_mutex_t m =
         PTHREAD_MUTEX_INITIALIZER;
  void debit(double amount);
  void credit(double amount);
};
```

```
class Transfer {
  Account source, destination;
  double amount;
  void run() {
    source.lock(); destination.lock();
    source.debit(amount);
    destination.credit(amount);
    destination.unlock(); source.unlock();
  }
};
```

```
class Bank {
  void fund_transfer() {
    Accounts numAccounts[N];
    Transfer pending[TOTAL];
    parallel_for(int i=0; i<TOTAL; i++) {
      pending[i].run();
    }
  }
};
```

- Is this correct?

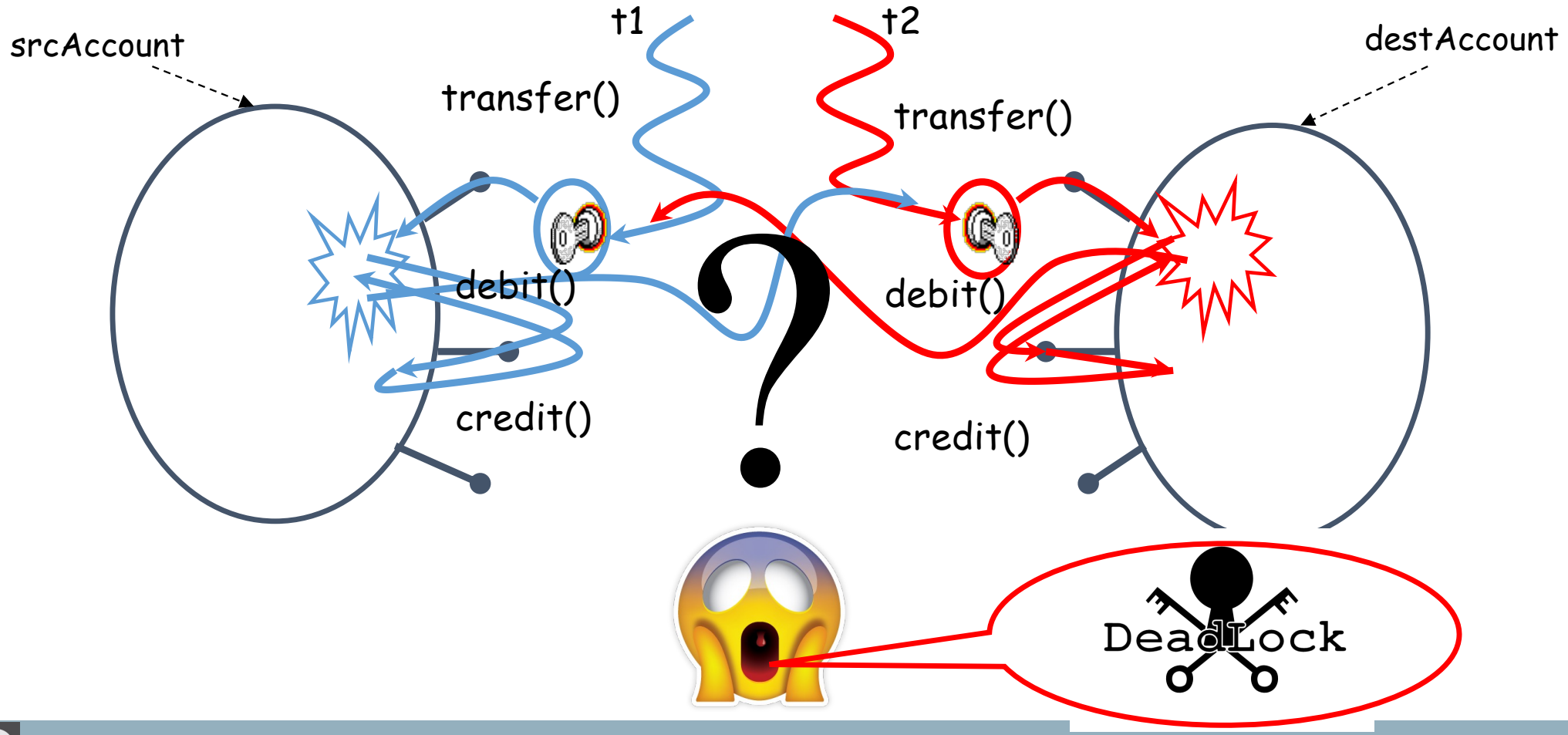
Money Transaction Between Accounts

```
class Account {
  int id;
  double balance;
  pthread_mutex_t
          PTHREAD_
  void debit(doub
  void credit(dou
};
```

```
class Bank {
  void fund_trans
    Accounts numA
    Transfer pend
    parallel_for(
      pending[i].
    }
  }
};
```

```
class Transfer {
                  .lock();
                  ;
                  rce.unlock();
```

Let's Analyze Our Money Transaction



Deadlock Avoidance

- **Deadlock occurs when multiple threads need the same locks but obtain them in different order**
- Not so easy to avoid deadlocks
- It's an active research area

Deadlock Avoidance

- Lock timeout
  - Put a timeout on lock attempts
    - pthread_mutex_timedlock
- Lock ordering
  - Ensure that all locks are taken in same order by any thread
    - **Let's try using it to fix our Bank Transaction program**

Money Transaction Between Accounts

```
class Account {
  int id;
  double balance;
  pthread_mutex_t m =
         PTHREAD_MUTEX_INITIALIZER;
  void debit(double amount);
  void credit(double amount);
};
```

```
class Bank {
  void fund_transfer() {
    Accounts numAccounts[N];
    Transfer pending[TOTAL];
    parallel_for(int i=0; i<TOTAL; i++) {
      pending[i].run();
    }
  }
};
```

```
class Transfer {
  Account source, destination;
  double amount;
  void run() {
    Account a1, a2;
    if(source.id < destination.id) {
      a1 = source; a2 = destination;
    } else {
      a1 = destination; a2 = source;
    }
    a1.lock(); a2.lock();
    source.debit(amount);
    destination.credit(amount);
    a2.unlock(); a1.unlock();
  }
};
```

Deadlock resolved using lock ordering

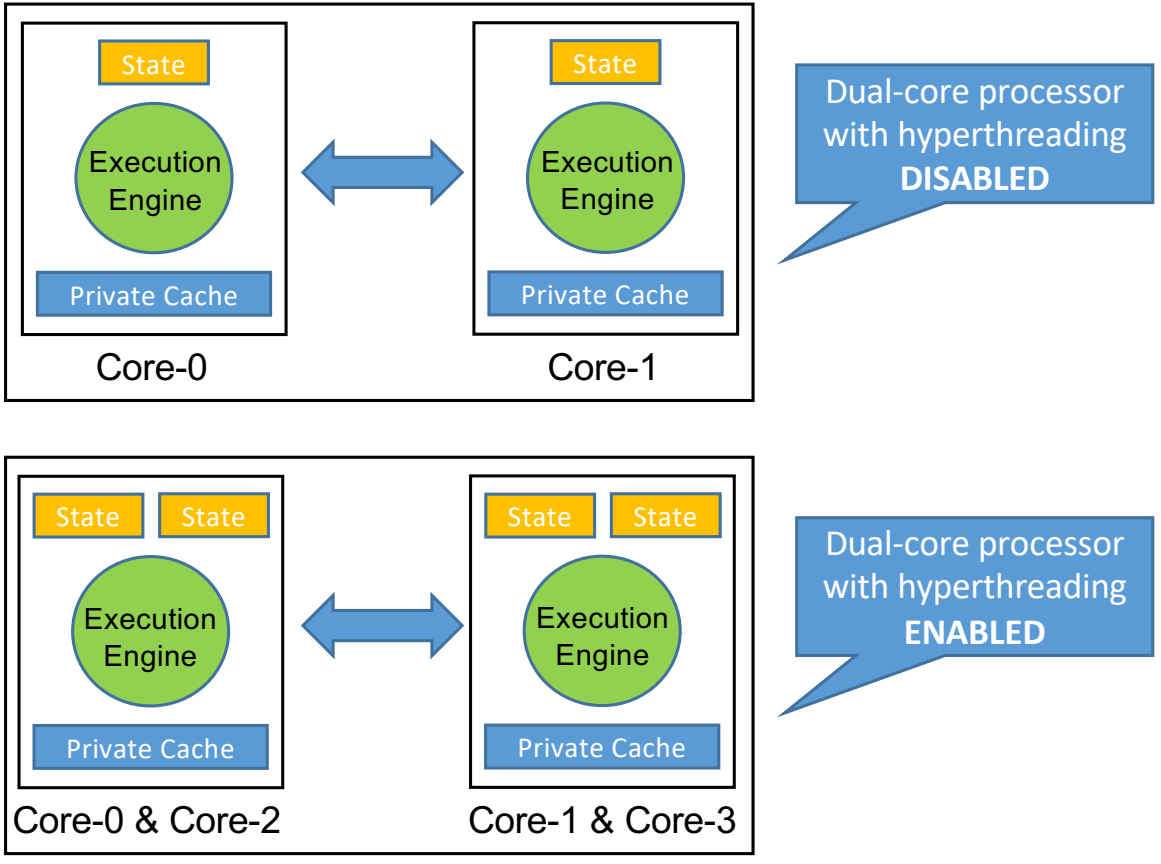
Miscellaneous Topics (Out of Syllabus)

- Physical core v/s logical cores
- NUMA architecture
- Power consumption
- Challenges with multithreading on modern processors

None of it will come in your exams (Enjoy!)

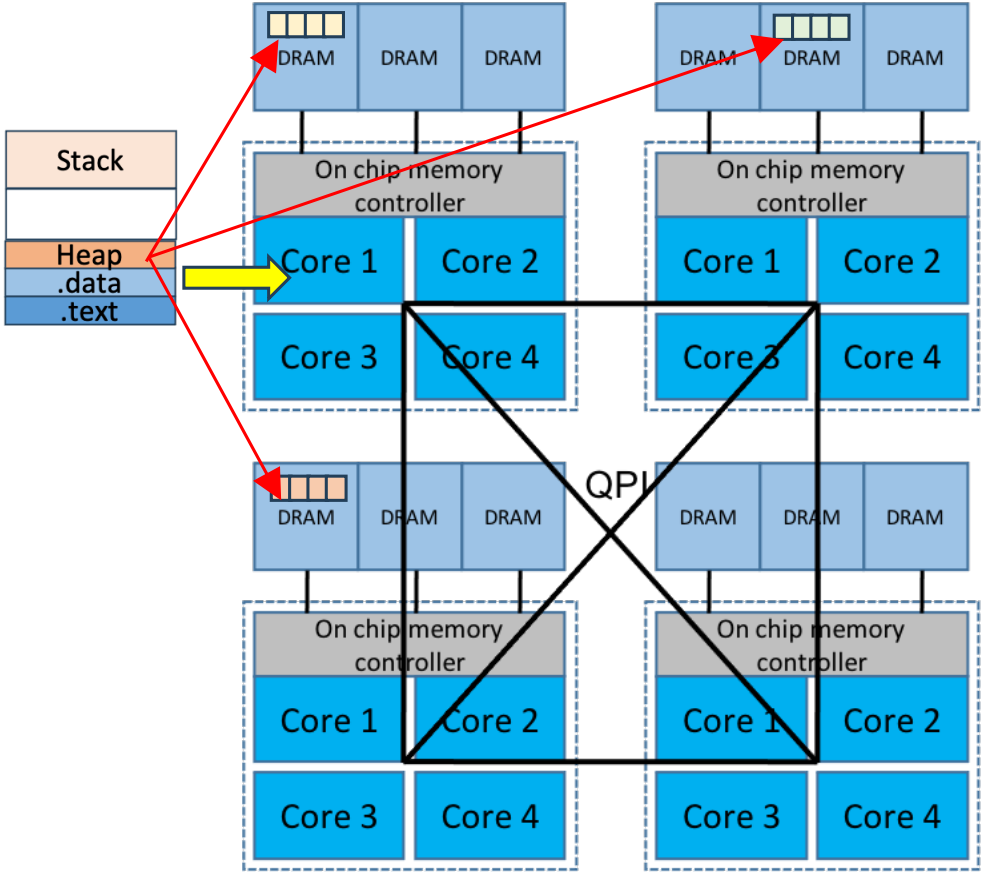
Food For Thought

Physical VS. Logical Cores



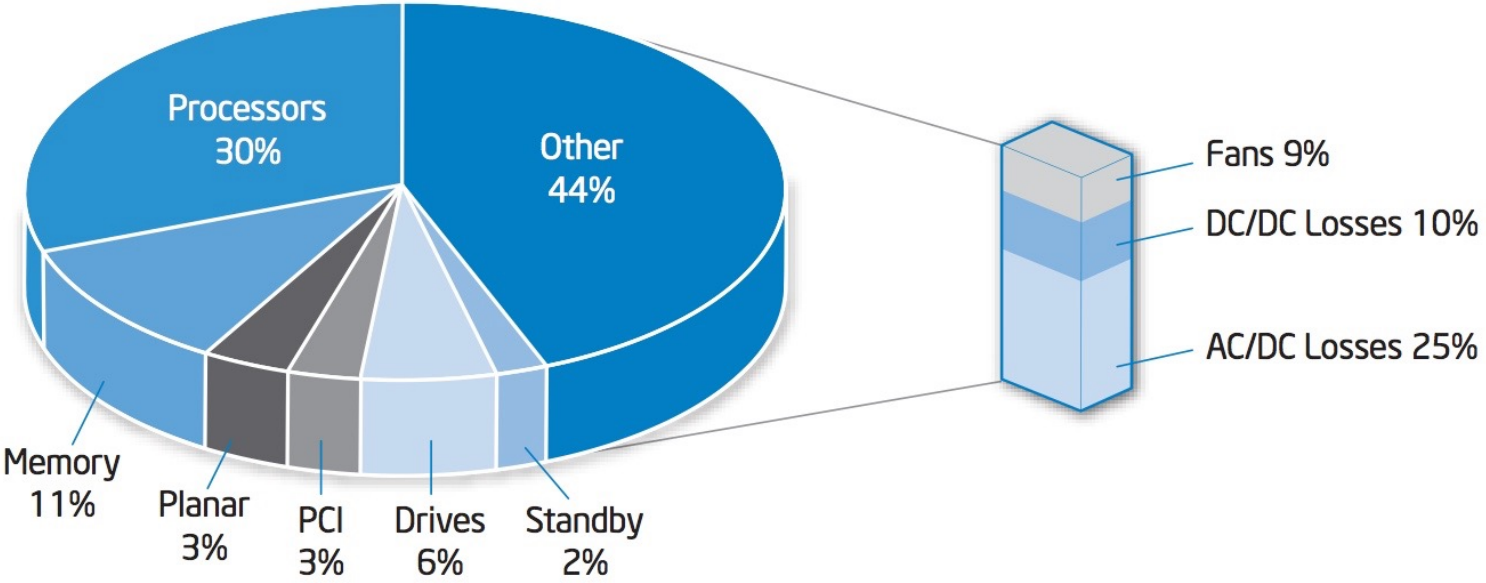
- Architectural state of a core are the registers (EBP, ESP, EIP, etc.)
- Logical cores of a processor share
  - Private cache
  - Execution engine
  - System bus interface
- If the execution of one of the logical core blocks (e.g., Core-0 waiting for a memory fetch from the DRAM) then the other logical core (Core-2) can resume its execution with its own state

Non Uniform Memory Access (NUMA)



- Multiple processors (sockets) on a single motherboard, each with local DRAM(s)
  - Connected together using fast interconnect that also offers cache coherency (e.g., Quick Path Interconnect in Intel)
- One socket can directly access memory of another socket
  - Non-uniform memory access time to local v/s remote memory
- Virtual page (VP) to physical page (PP) mapping matters
  - PP on local DRAM has faster access v/s PP on remote DRAM

Component wise Power Consumption



Modern processors provide several features in userspace for achieving energy efficiency

- As per studies of power consumption in a data center
  - 50% of incoming power is consumed by air-conditioning and power-delivery subsystems, even before reaching the servers in a rack
  - Rest 50% consumed by the servers, which can be further broken down into the various elements as shown above

Source: https://www.intel.com/content/dam/support/us/en/documents/motherboards/server/sb/power_management_of_intel_architecture_servers.pdf

Challenges With Multithreading

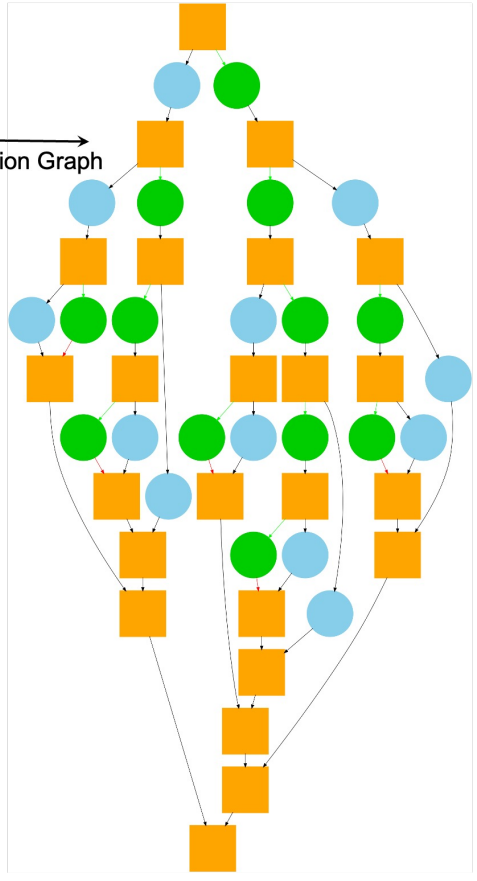
- Challenging in achieving high productivity and high performance in parallel programming over large number of cores
  - Same program should run over a variety of multicore processors without requiring any modifications (high productivity!)
    - One-to-one mapping between thread and core
      - Using logical cores might not benefit in each program
  - Threads should attempt to equally divide the total work (high performance!)
    - If there is not enough parallelism then having one-to-one mapping between thread and core may not benefit (avoiding oversubscribing)
    - Thread running on a core should have most of its data (physical pages) allocated on the local DRAM (avoiding NUMA overheads)
    - Parallel programs should achieve optimal of performance and energy utilization (achieving energy efficiency)

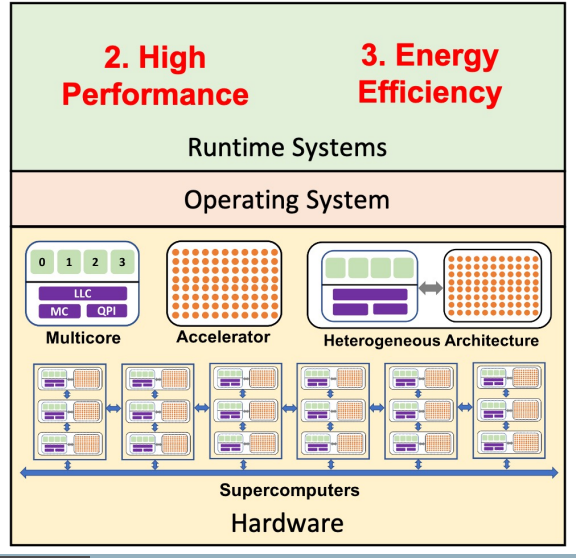
How to Deal With Those Challenges?

```
void recursive(int low, int high) {
  if ((high-low) > threshold) {
    int mid = (high-low) / 2;
    finish {
      async recursive(low, mid);
      recursive(mid, high);
    }
  } else {
    serial_computation(low, high);
  }
}
```

1. High Productivity

Execution Graph





- Parallel programming model
  - Use tasks instead of threads
  - Tasks (async) are composed of a function pointer and the argument to the function
    - Much lightweight than threads
- Parallel runtime systems
  - Performs dynamic load balancing of tasks by mapping tasks to threads based on their current workload
  - Provides several opportunities for achieving high performance and energy efficiency

Next Lecture

- Introduction to Filesystem
- Quiz-4
  - Syllabus: Lectures 18-23

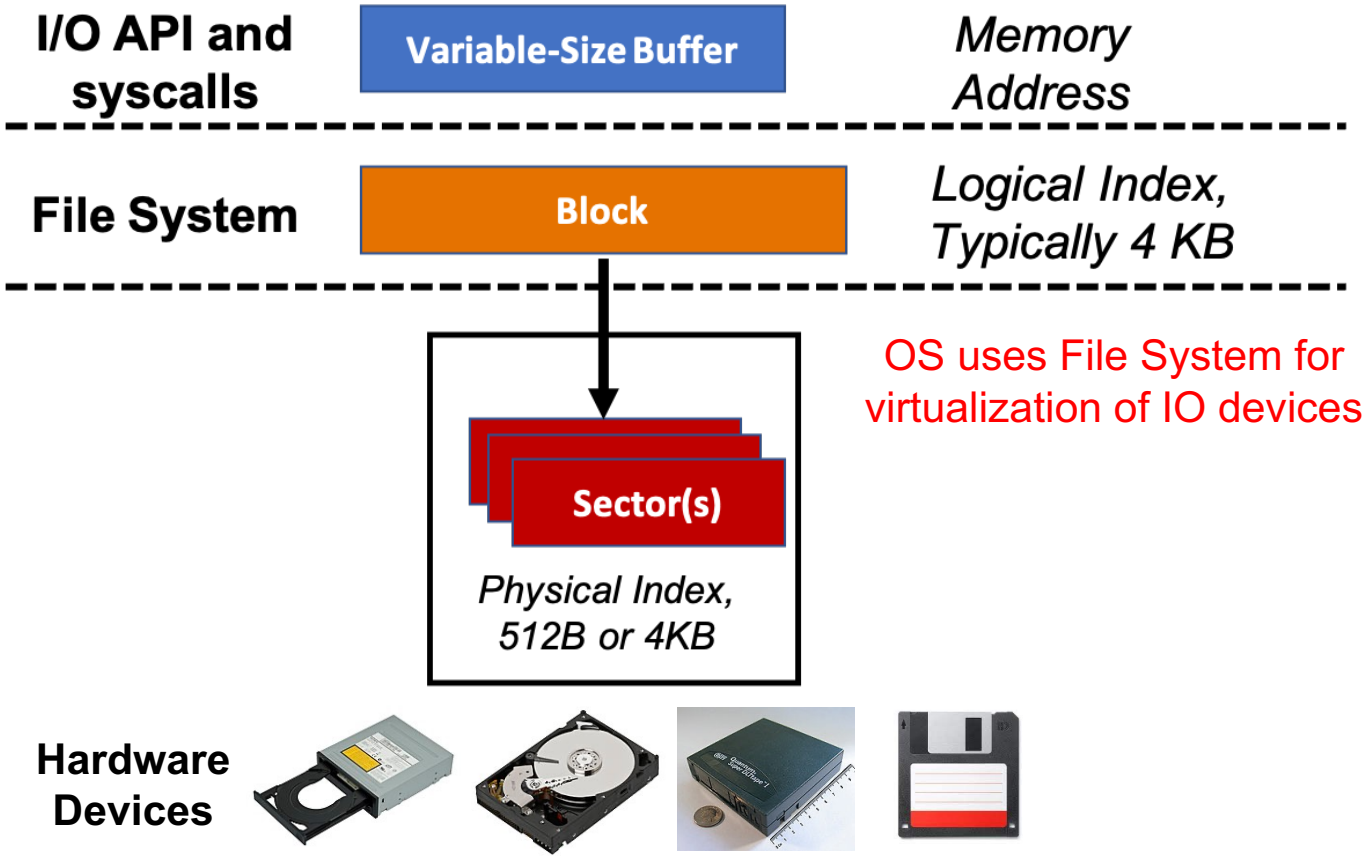
Lecture 24: Introduction to File Systems

Vivek Kumar
Computer Science and Engineering
IIIT Delhi
vivekk@iiitd.ac.in

Today's Class

- Quiz-4 (last remaining quiz)
- Persistent storage (IO) devices
- Virtualization of IO devices

File System

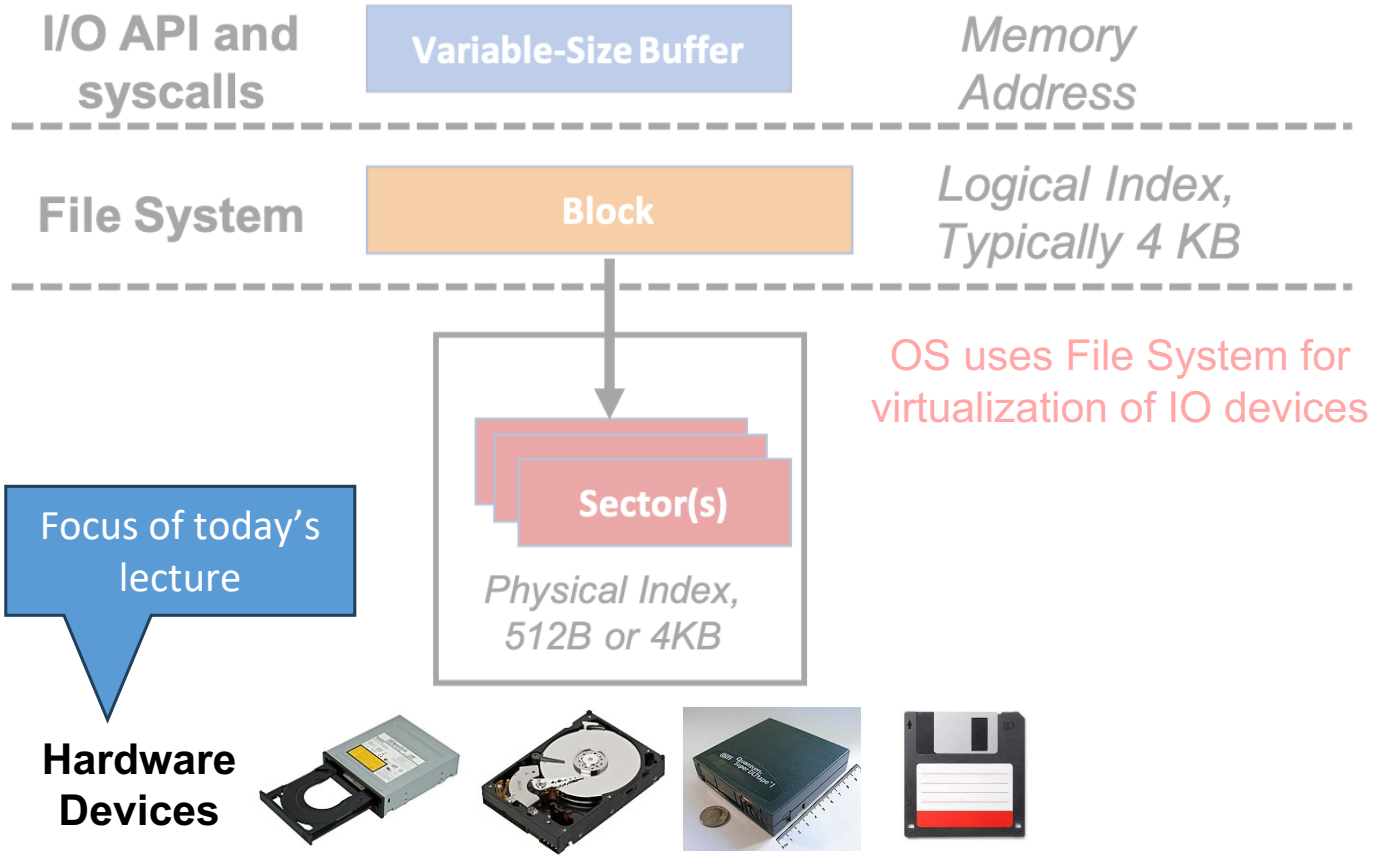


- User's view of File:
  - Durable Data Structures
- System's view of File (system call interface):
  - Collection of Bytes (UNIX)
  - Oblivious to specific data structures user wants to store
- System's view of file (inside OS):
  - File is a collection of blocks

Role of File System

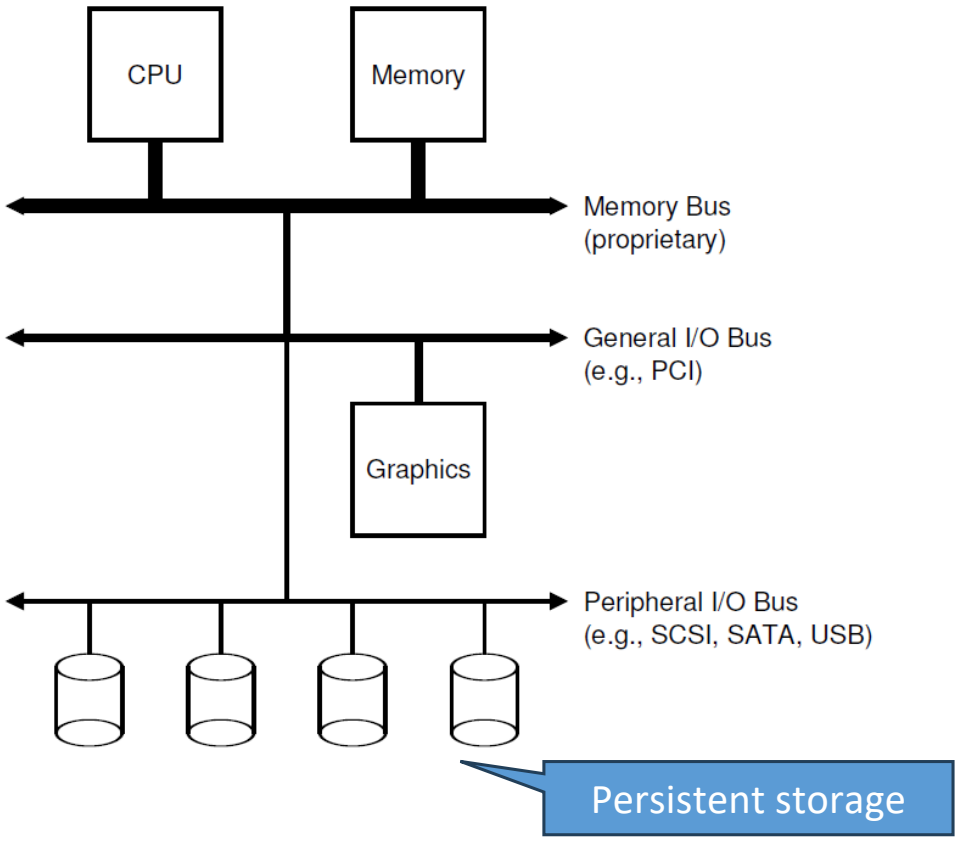
- Virtualization of IO devices
  - Provide abstraction over a variety of IO devices by providing a uniform interface for accessing data
- Provide support for crash recovery
  - The data must be available even after a reboot or accidental power cycle
- Hide the access latency
  - Disk access is several orders of magnitude slower than DRAM access

File System



- User's view of File:
  - Durable Data Structures
- System's view of File (system call interface):
  - Collection of Bytes (UNIX)
  - Oblivious to specific data structures user wants to store
- System's view of file (inside OS):
  - File is a collection of blocks

Computer System Architecture

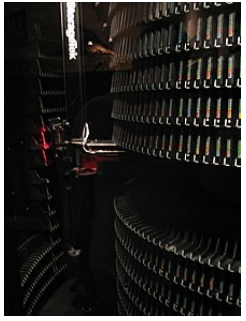


- I/O devices connect to the CPU and memory via a bus
  - High speed bus, e.g., PCI
  - Other: SCSI (Small Computer System Interface), USB, SATA
- "Port" is the point of connection to the system

Persistent Storage (IO Devices)

Hard disk

Tape cartridge



Tape library

DVD drive

Floppy drive

USB drive

Image source: Google images

- Persistence refers to the characteristic of state that outlives the process that created it
  - Otherwise, data can only reside in DRAM which will be lost at power cycle
- Persistent storage device is any data storage media that retains the data even after a power cycle

Prototype of a Simple IO Device

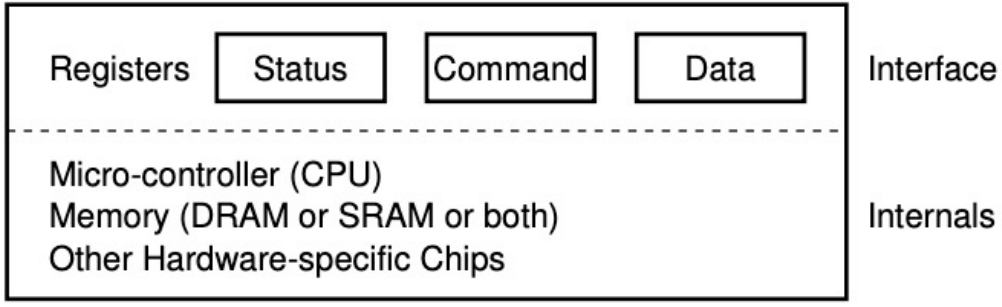
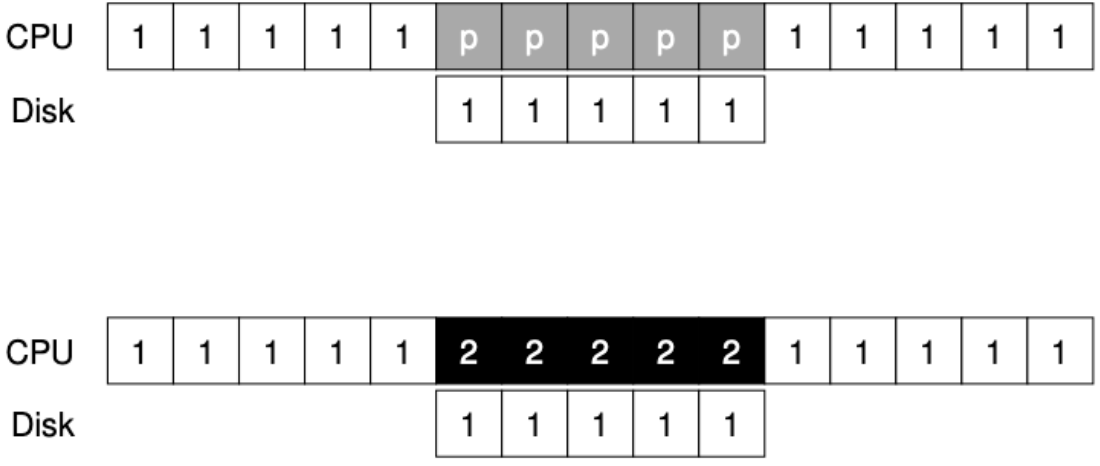


Figure 36.3: **A Canonical Device**

```
While (STATUS == BUSY)
    ; // wait until device is not busy
Write data to DATA register
Write command to COMMAND register
    (starts the device and executes the command)
While (STATUS == BUSY)
    ; // wait until device is done with your request
```

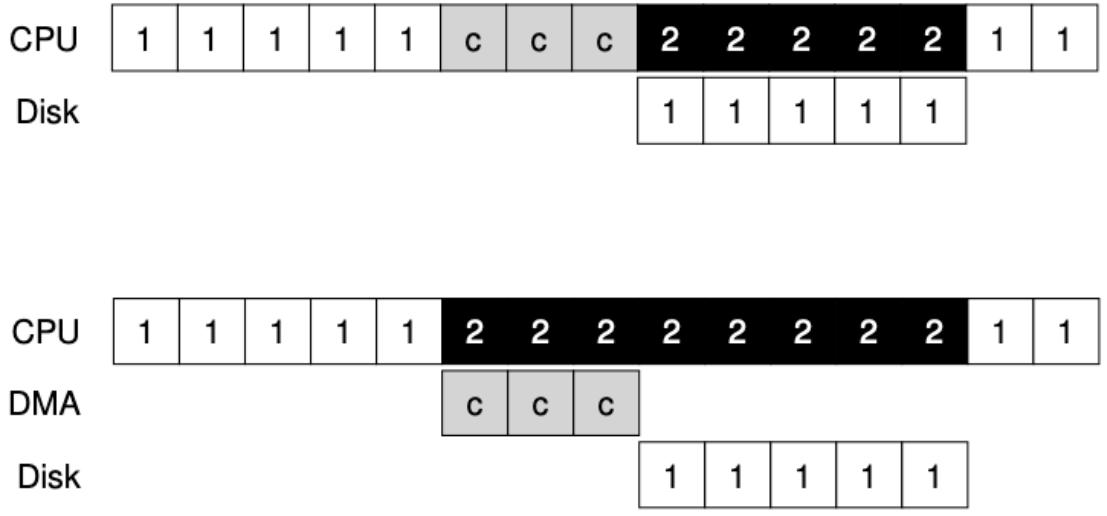
- OS interact with the IO device via a software layer called as device driver
- Devices expose an interface of memory registers
  - Current status of device
  - Command to execute
  - Data to transfer
- The internals of device are usually hidden
- Device interaction in four steps
  - Polling for the device to be ready
  - Sending data to data register
  - Inform the completion of data transfer by writing to command register
  - Polling for device for completion
- Do you see any issues here?
  - Wastage of CPU in polling

Avoiding Polling Overheads



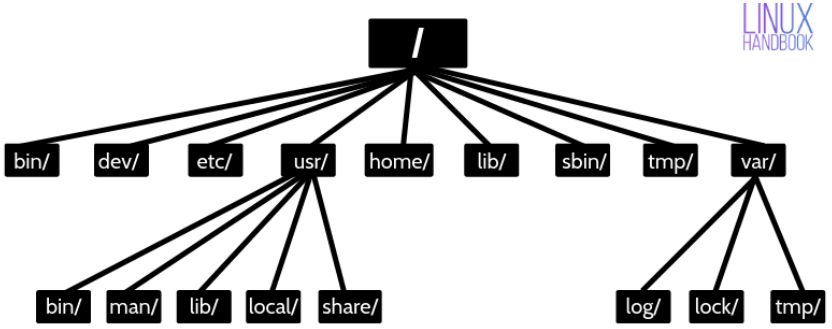
- Polling
  - CPU cycles wasted as CPU simply spins without doing any other job
- Interrupt
  - OS can put the polling process in wait queue and bring another process from ready queue
  - Interrupt sent to blocked process by the IO device upon completion of IO
- Hybrid
  - Start with a poll and if IO did not complete quickly then wait for interrupt

Direct Memory Access (DMA)



- CPU cycles wasted in copying data to/from device
- Instead, a special piece of hardware (DMA engine) copies from main memory to device
  - CPU gives DMA engine the memory location of data
  - In case of read, interrupt raised after DMA completes
  - In case of write, disk starts writing after DMA completes

Virtualization of IO Devices



Linux directory structure

- OS provides abstraction over different types of IO devices by having a uniform interface of files and directories
- **File** is the smallest unit of storage device as seen by the user
  - User cannot write data to storage device unless data is within a file
  - File is an abstract data type with basic set of operations
    - Create, Write, Read, Delete, Reposition (seek), Truncate, etc.
  - OS refers to file name as its "**inode**" number (Unix based OS)
- **Directory** is also a file but with the information on how to find other files
  - It also has its **inode** number

File Descriptor and File Table (Unix)

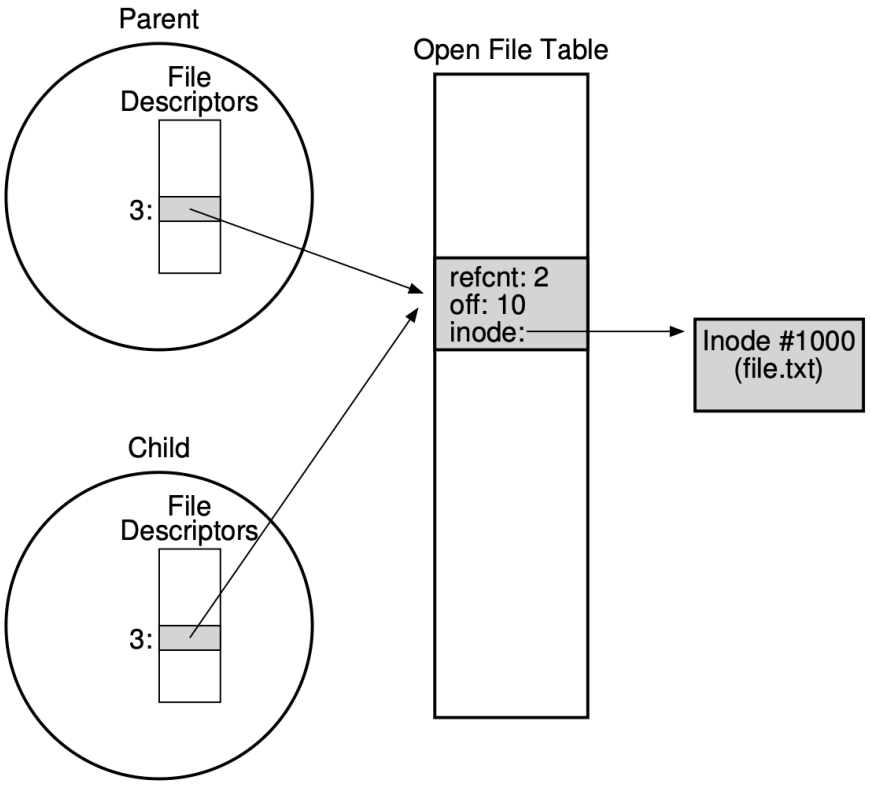


Figure 39.3: **Processes Sharing An Open File Table Entry**

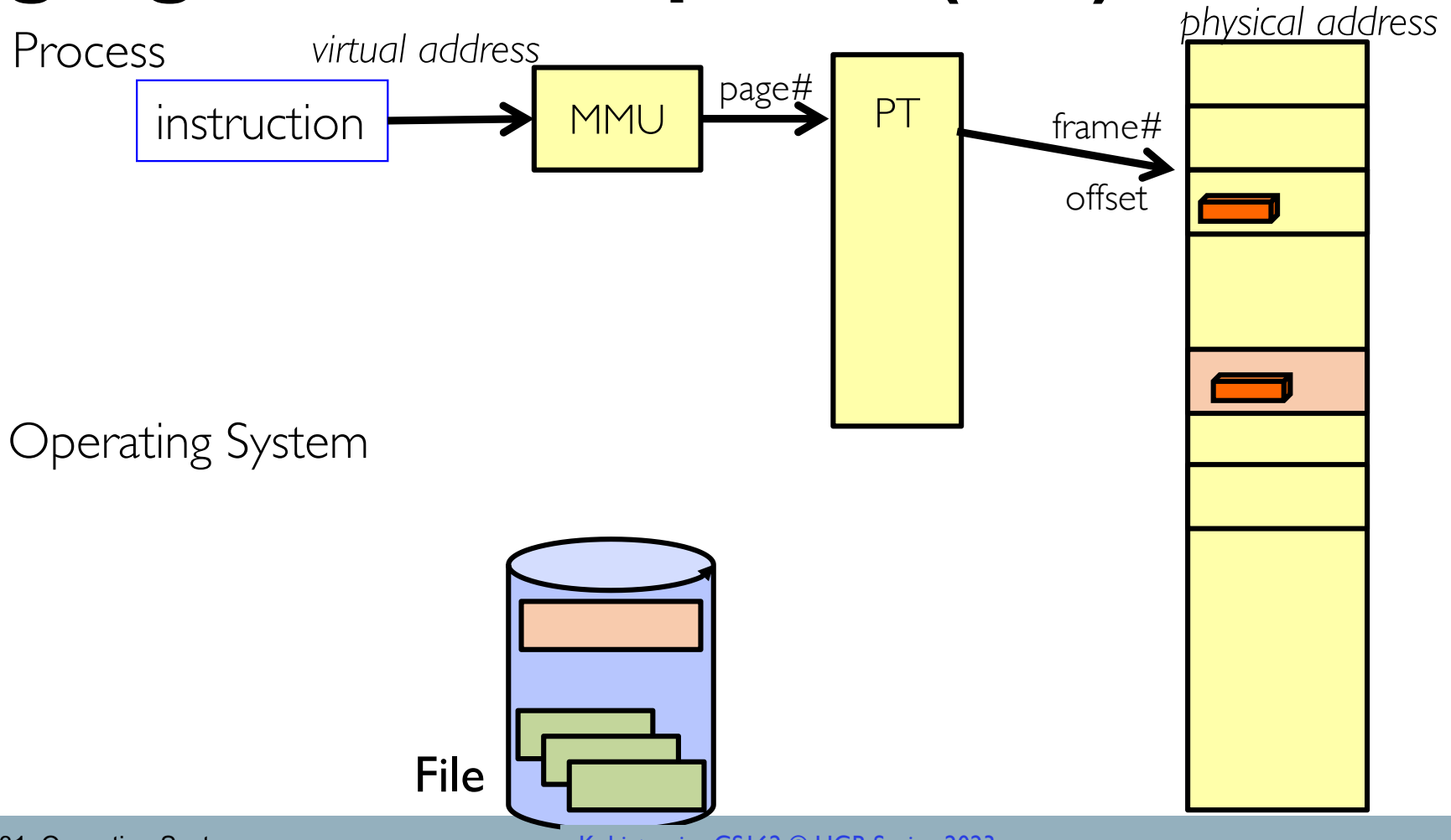
- Each process has an array of open **file descriptors**
- Each file descriptor points to a entry inside system wide open **file table**
- "**open**" call will create a new entry inside file table if no entry found, otherwise a copy of the previous entry is added, but with its file offset reset to zero
- Each entry inside file table keeps track of total processes **sharing** this file descriptor (e.g., sharing due to parent-child relation), current offset inside file, inode pointer to the file, file permissions etc.)
- File table entry is removed once the reference count reaches zero

Types of IO

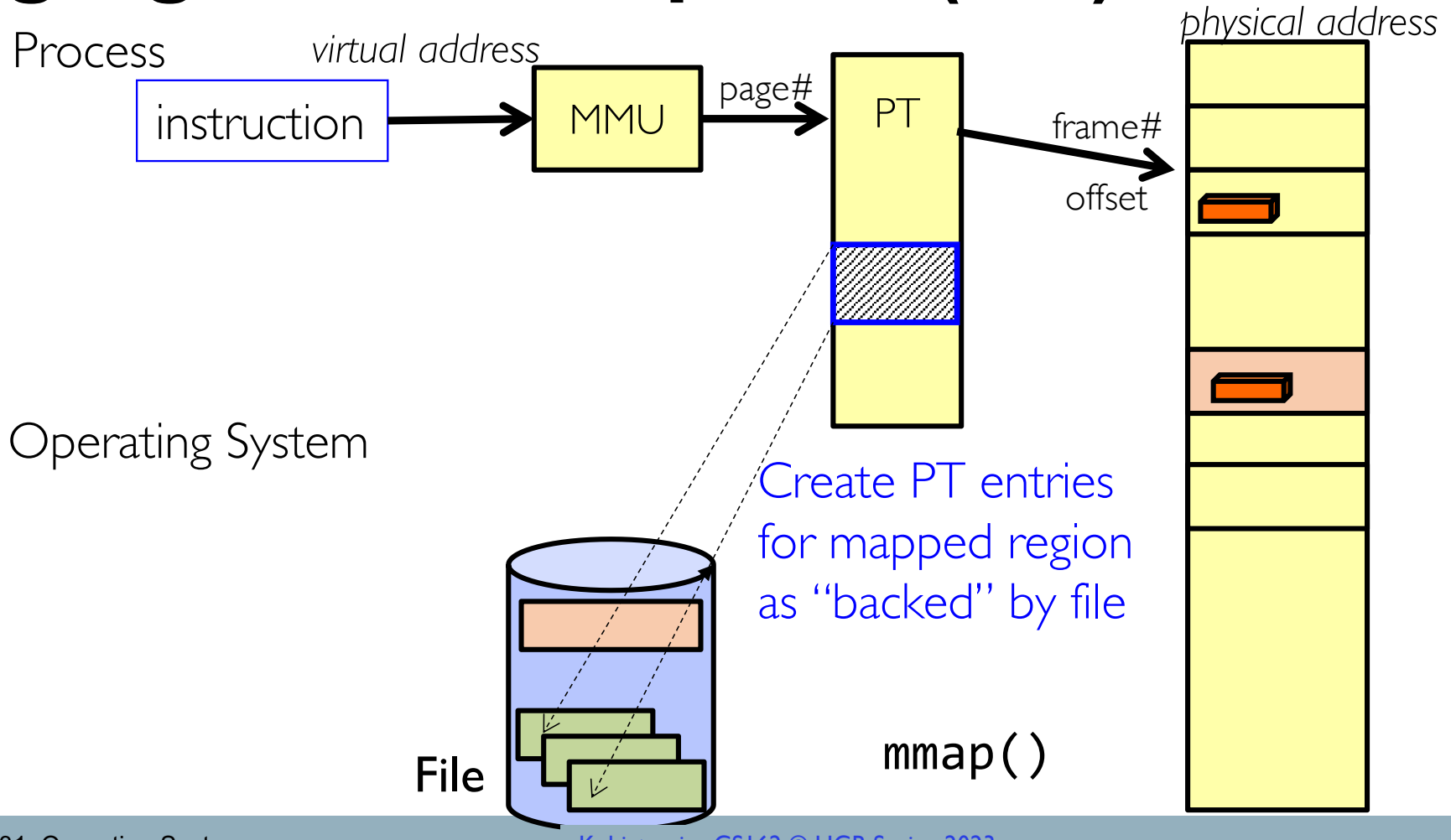
3. Size of the file.
4. File type: regular,directory,device etc...
5. Date and Time of Last modification of the file data.
6. Date and Time of Last access of file data.
7. Date and Time of Last change of the I-node.

- File IO
  - Explicit transfer between buffers in process address space to regions of a file
  - Overhead due to multiple copies in memory and system calls
- Memory mapped IO
  - **mmap** file directly into an empty region of a process address space
  - ELF file is treated this way when we execute a process (recall your assignment-4)

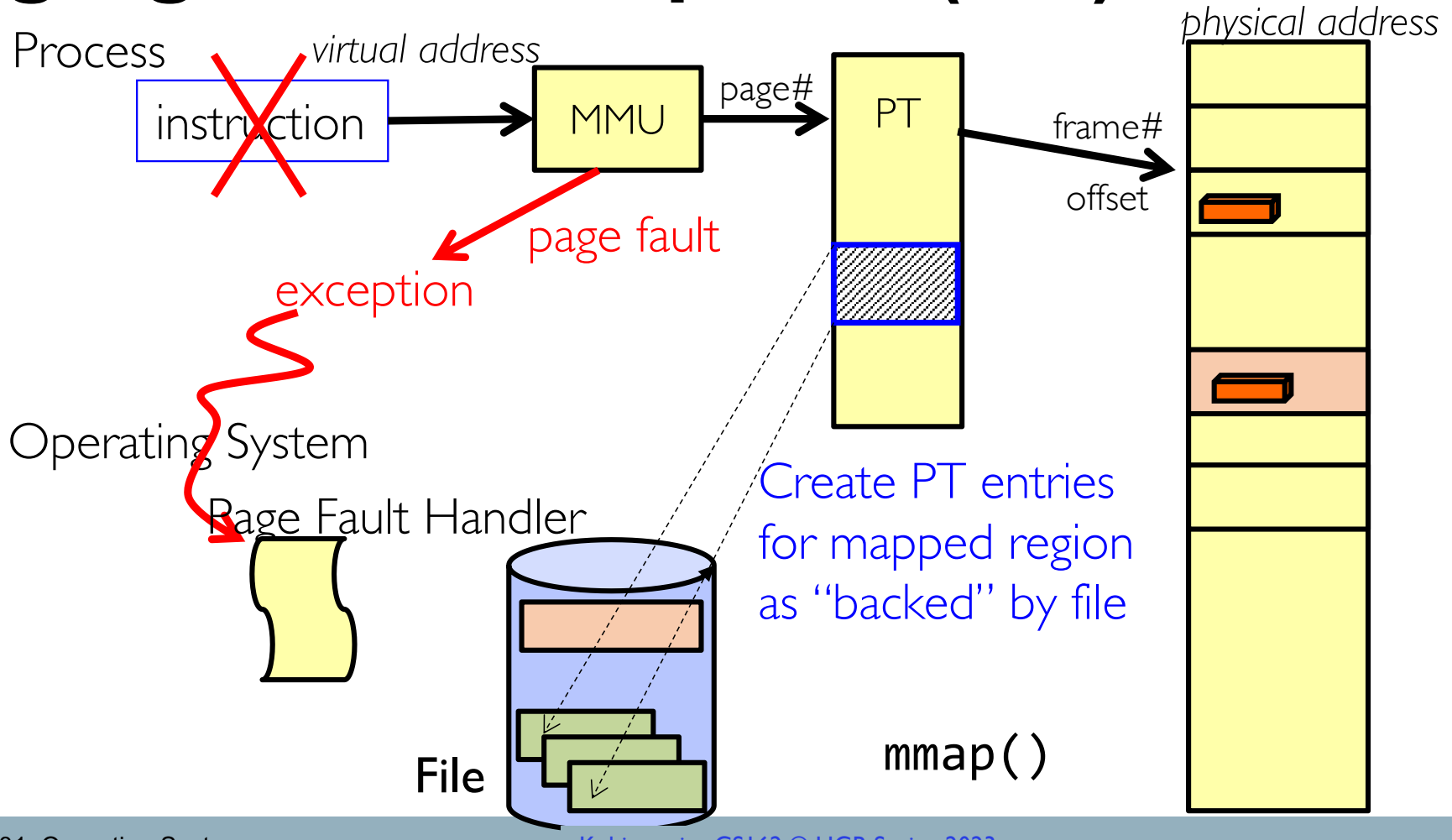
Paging With mmap File (1/6)



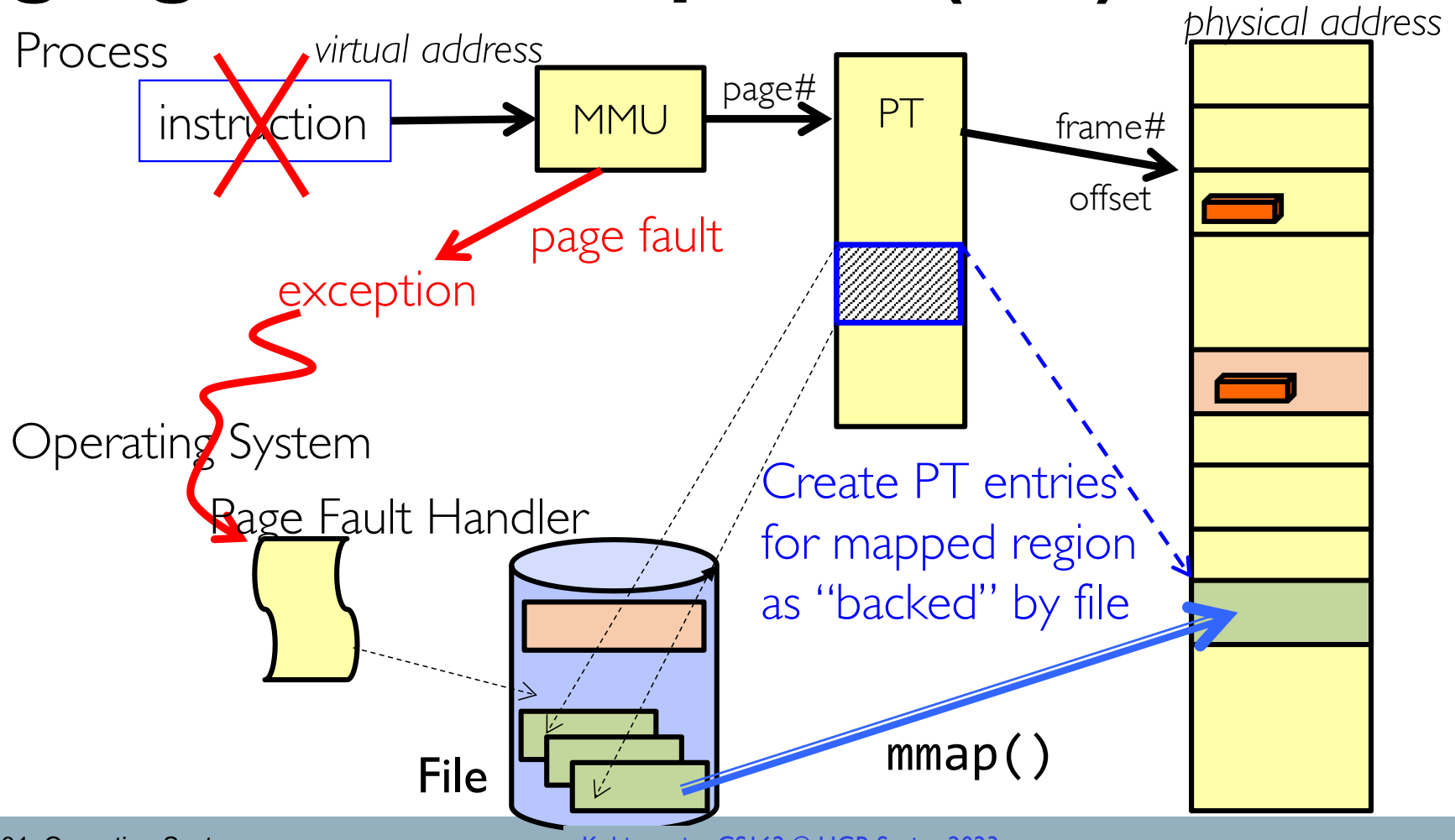
Paging With mmap File (2/6)



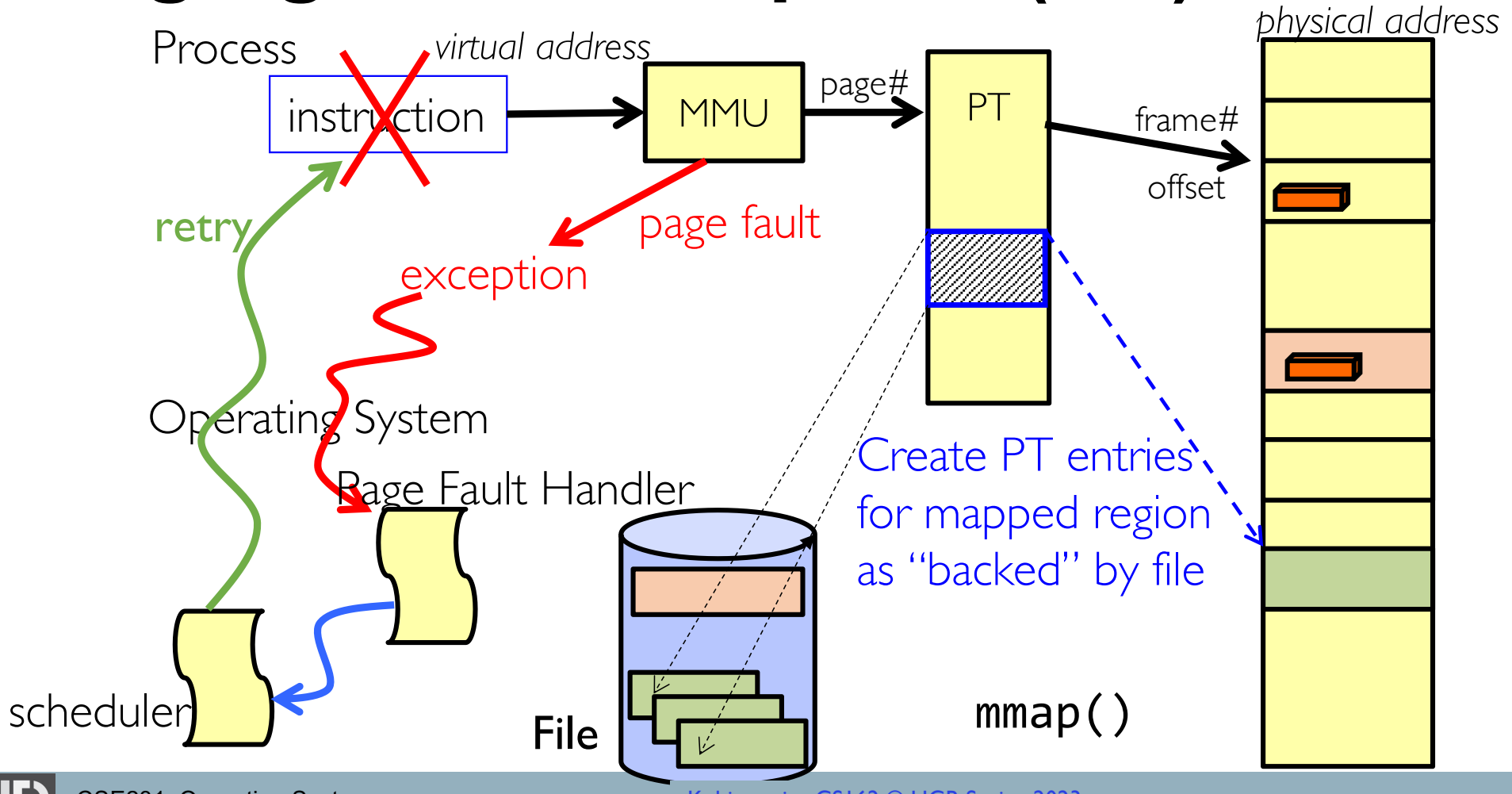
Paging With mmap File (3/6)



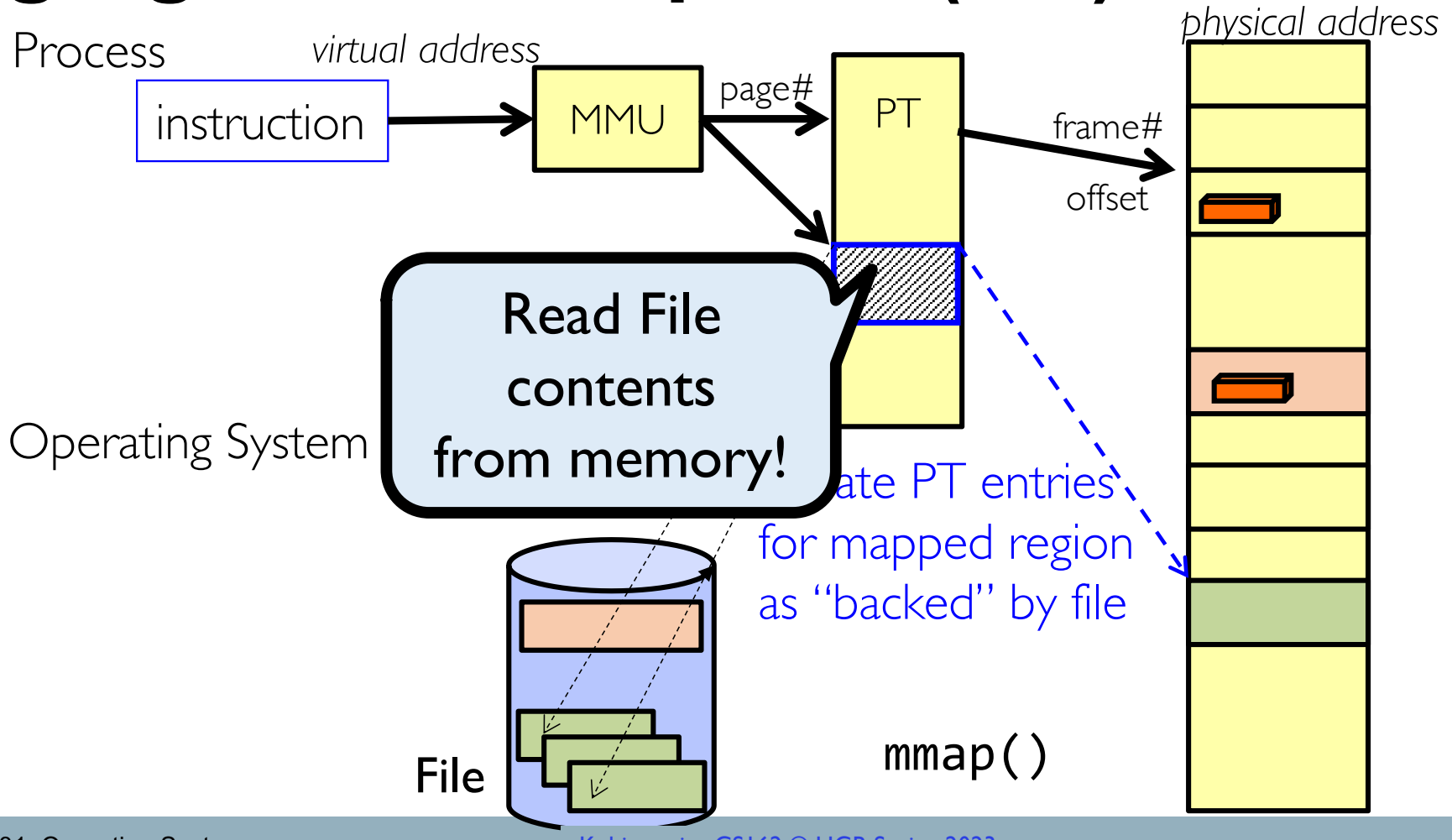
Paging With mmap File (4/6)



Paging With mmap File (5/6)



Paging With mmap File (6/6)



Next Class

- File system implementation

Lecture 25: File System Implementation

Vivek Kumar
Computer Science and Engineering
IIIT Delhi
vivekk@iiitd.ac.in

Last Lecture

```
While (STATUS == BUSY)
    ; // wait until device is not busy
Write data to DATA register
Write command to COMMAND register
    (starts the device and executes the command)
While (STATUS == BUSY)
    ; // wait until device is done with your request
```

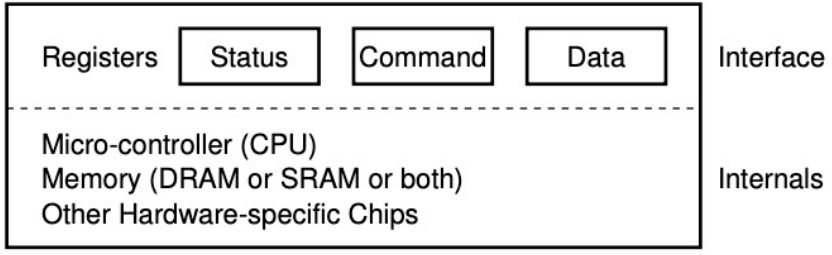
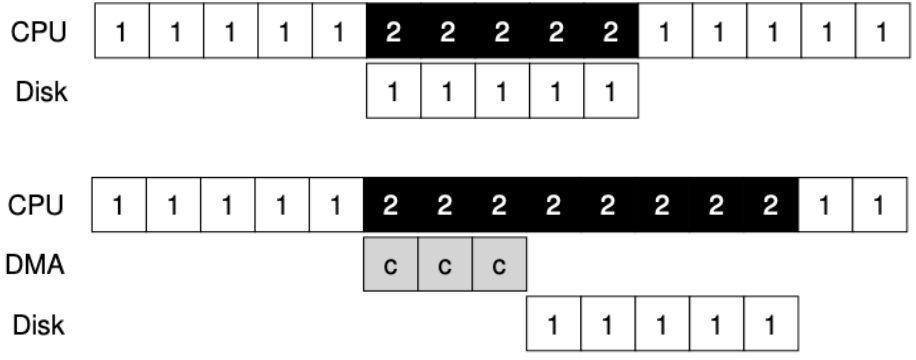
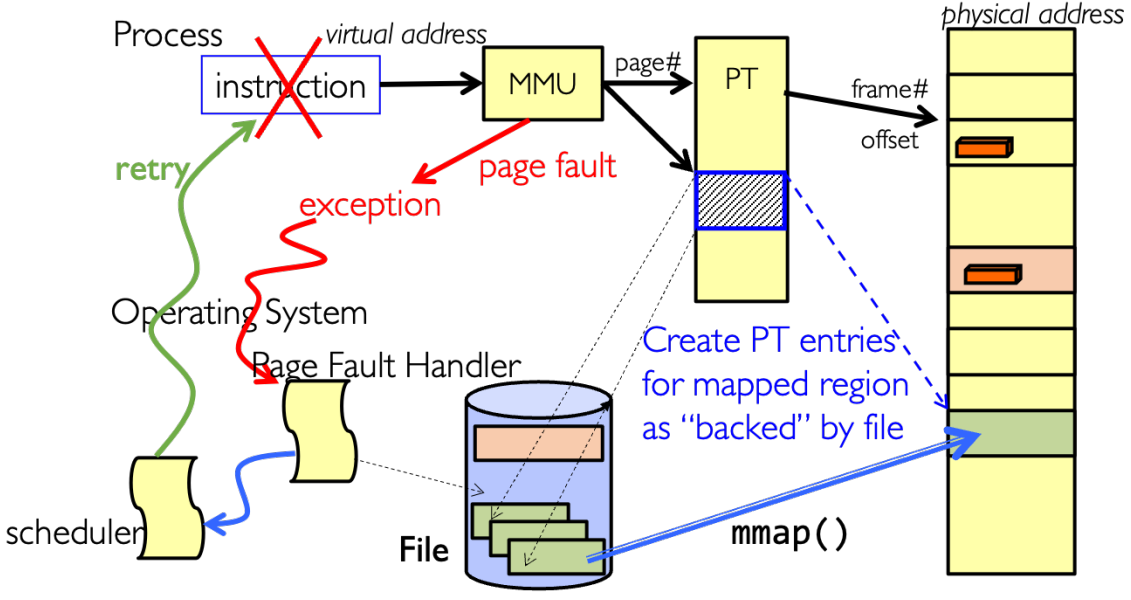


Figure 36.3: **A Canonical Device**



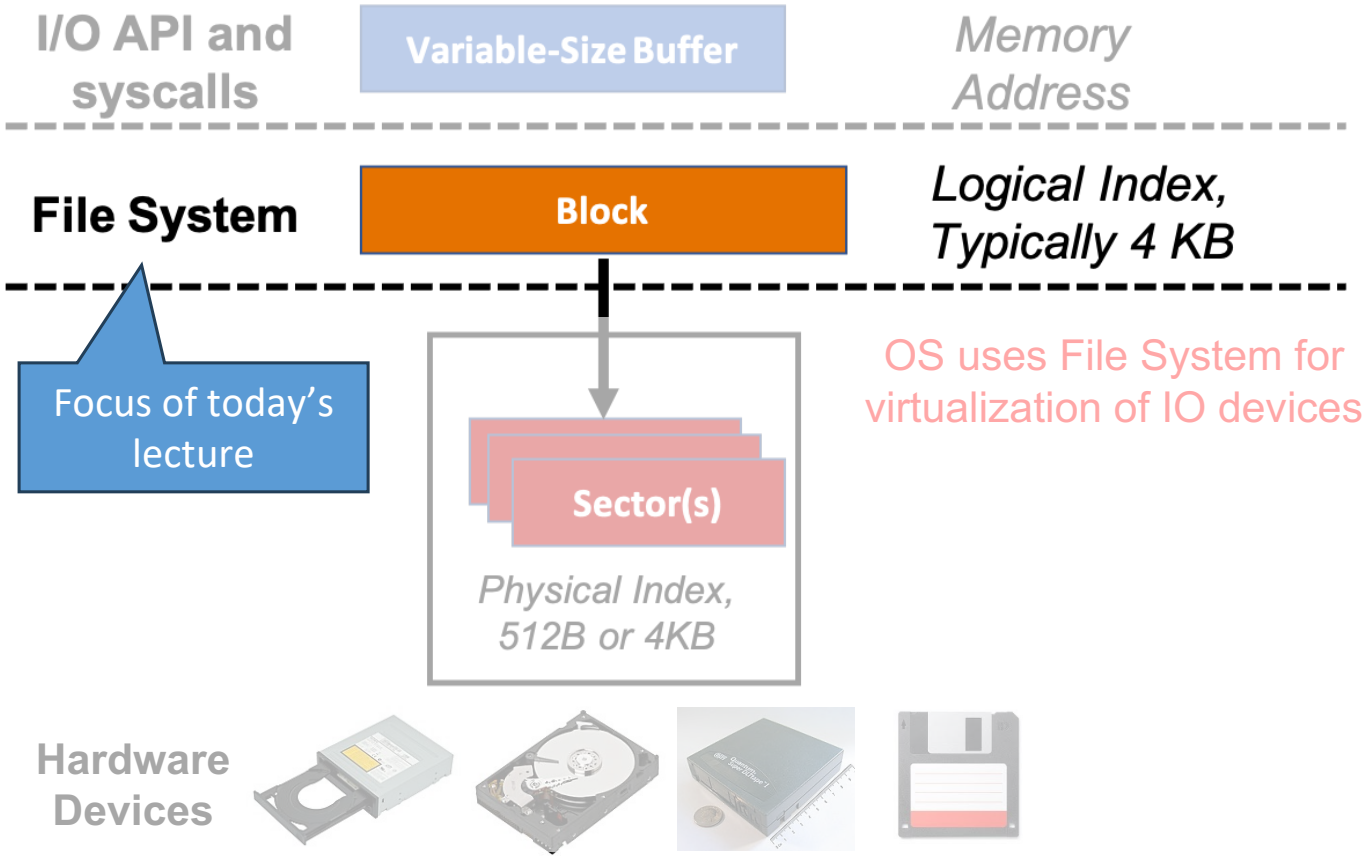


- Prototype of a simple IO device
- Using interrupts to avoid polling overheads
- Using DMA to avoid wastage of CPU cycles during copying of data
- Paging with memory mapped files

Today's Class

- Design of FAT and FFS file systems
- Buffer cache
- Dealing with system crashes

File System



- User's view of File:
  - Durable Data Structures
- System's view of File (system call interface):
  - Collection of Bytes (UNIX)
  - Oblivious to specific data structures user wants to store
- System's view of file (inside OS):
  - File is a collection of blocks

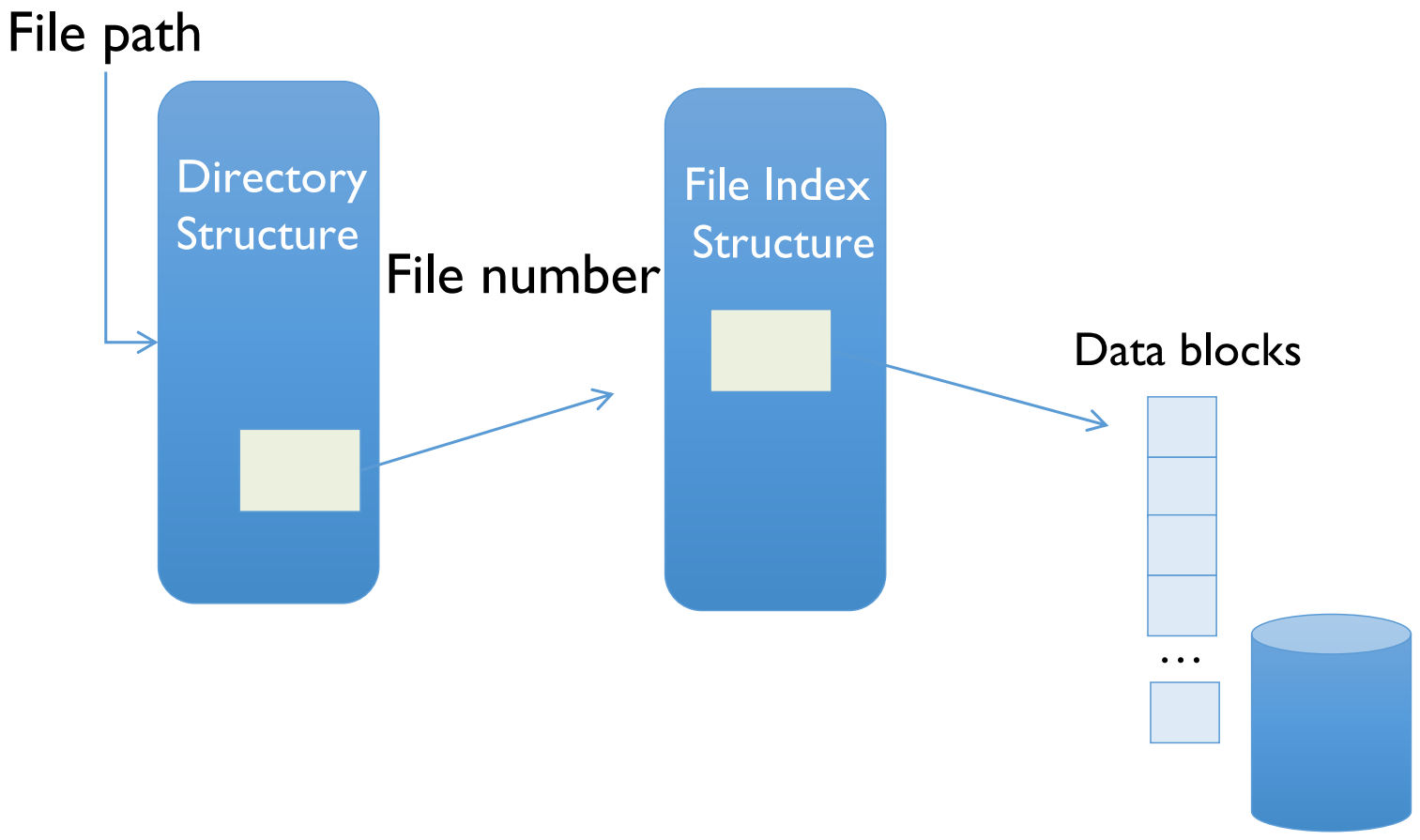
Building a File System

- File organization
  - Organize files by names with directories
- Protection
  - Provide access restriction
- Fast access
  - Access to disk is several orders of magnitude slower than DRAM access
- Reliability
  - Keep files intact despite crashes, hardware failures, etc.

What Does the File System Needs?

- Track free disk blocks
  - Need to know where to put newly written data
- Track which blocks contain data for which files
  - Need to know where to read a file from
- Track files in a directory
  - Find list of file's blocks given its name
- Where do we maintain all of this?
  - Somewhere on disk

Basic File System Components



Components of a File System

file name $\xrightarrow{\text{directory}}$ *file number* *offset* $\xrightarrow{\text{index structure}}$ *Storage block*

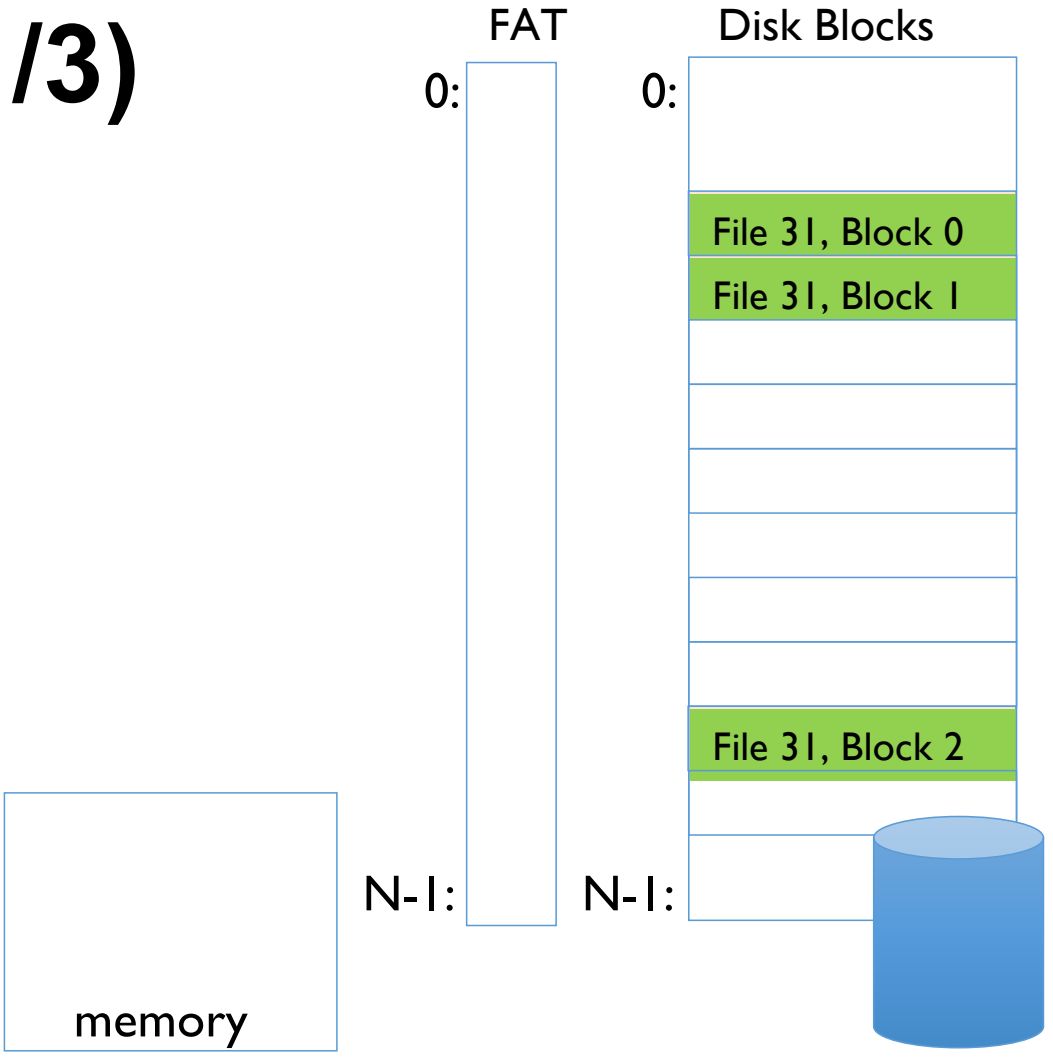
- Open performs **Name Resolution**
  - Translates pathname into a "file number"
    - Used as an "index" to locate the blocks
  - Creates a file descriptor in PCB within kernel
  - Returns a file descriptor (int) to user process
- Read and Write operation on the file number
  - Use file number as an "index" to locate the blocks on disk

FAT (File Allocation Table)

- FAT is a file system architecture which is really simple and robust
  - Used in several places (MS-DOS, Windows (sometimes) and External drives)
- It uses linked allocation, where each file is a linked list of blocks on disk
  - These blocks could be scattered any where on the disk
- What is benefit of linked allocation v/s contiguous allocation of blocks on disk?
  - Each file occupying a set of contiguous blocks on the disk is really simple to implement but doesn't allow file growth
  - To avoid file growth, extra space must be allocated upfront leading to space wastage (disk fragmentation)

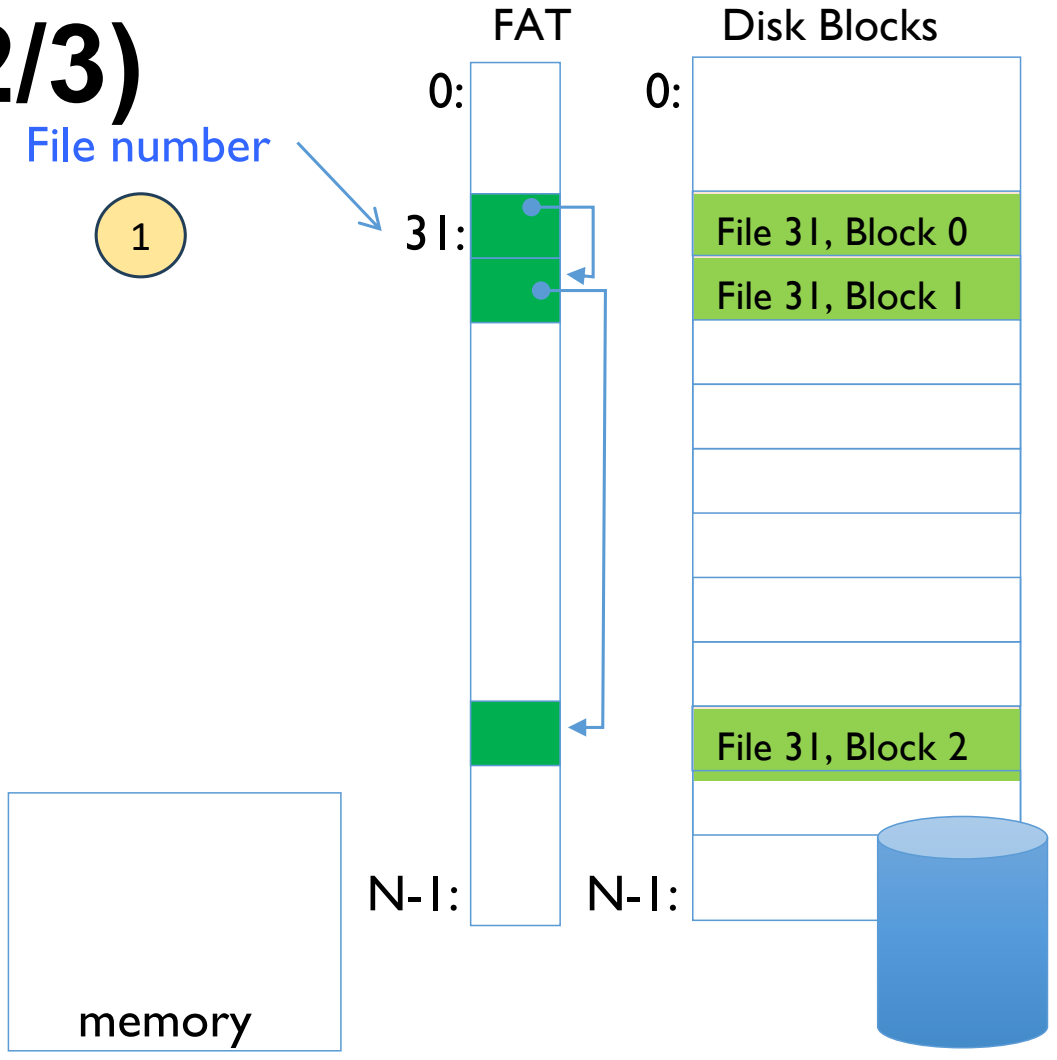
FAT: Reading File (1/3)

- Example: read file 31, block 2



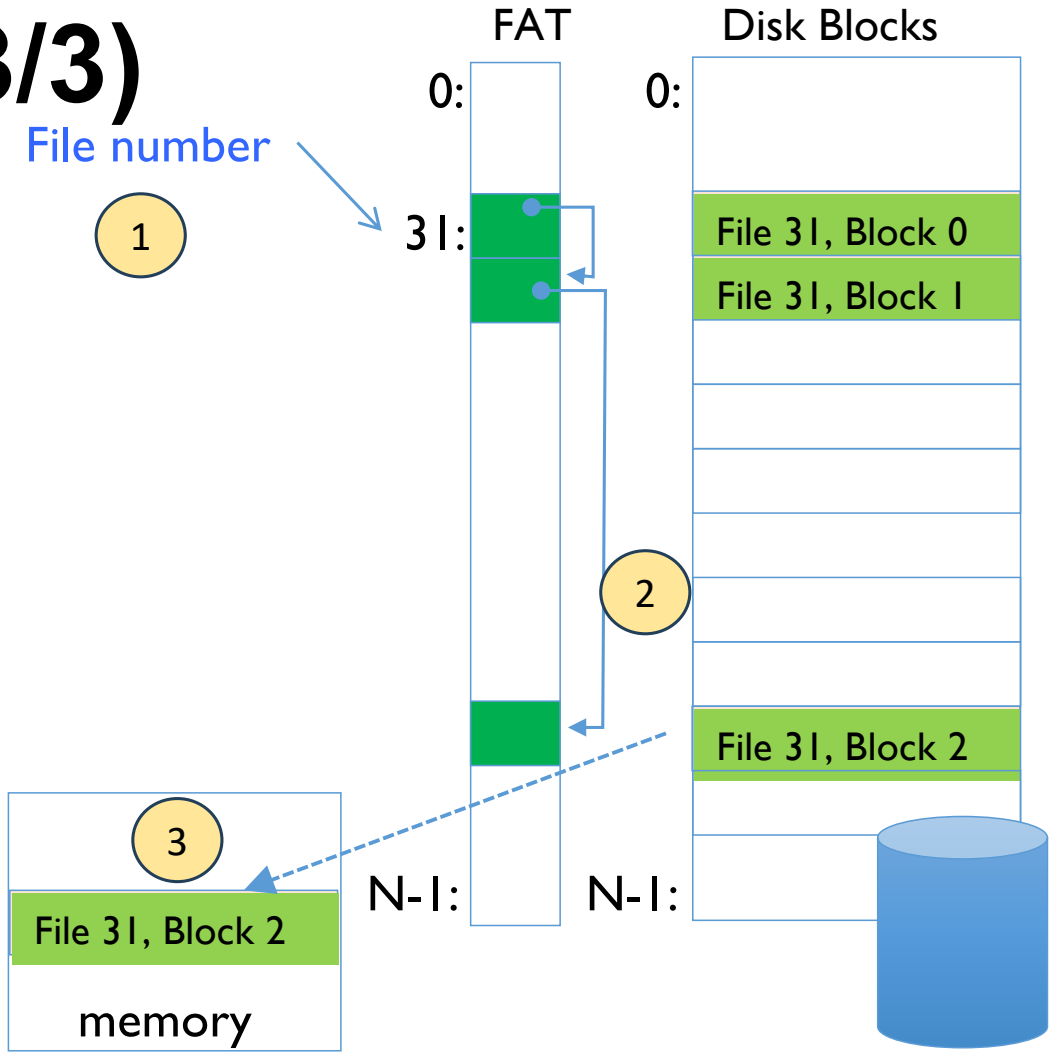
FAT: Reading File (2/3)

- Example: read file 31, block 2
  1. Index into FAT with file num
     - Simple way to store blocks of a file: **Linked List** structure
     - File number is just the **first block**
     - One entry in table per data block
     - FAT contains pointer to **the next block** for each entry (or special END value)



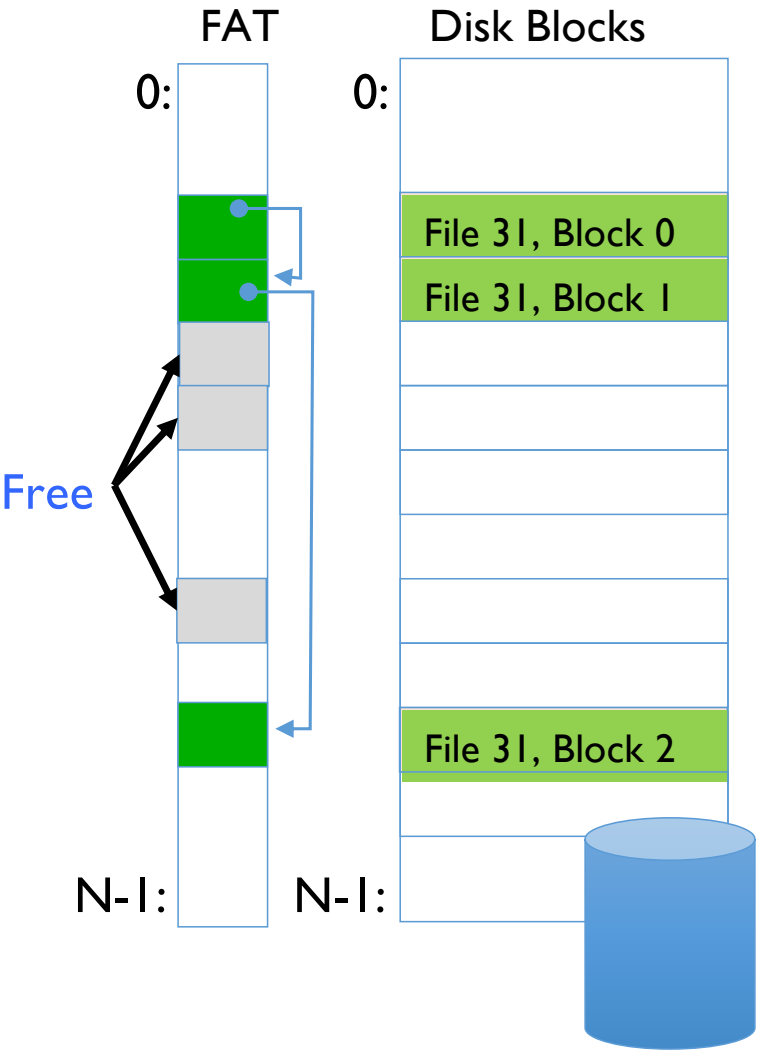
FAT: Reading File (3/3)

- Example: read file 31, block 2
  1. Index into FAT with file num
  2. Follow linked list to block 2
  3. Read block from disk into mem



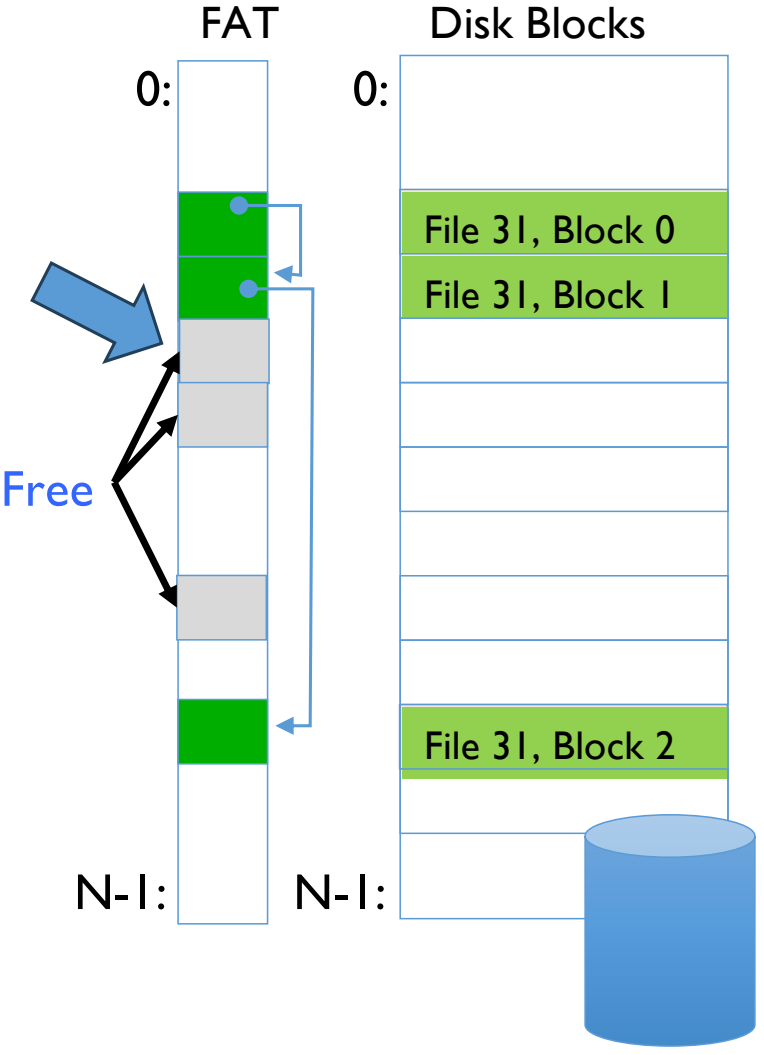
FAT: Expanding File (1/4)

- Entries in table corresponding to free block have value **0**
- Must scan through FAT to find free space



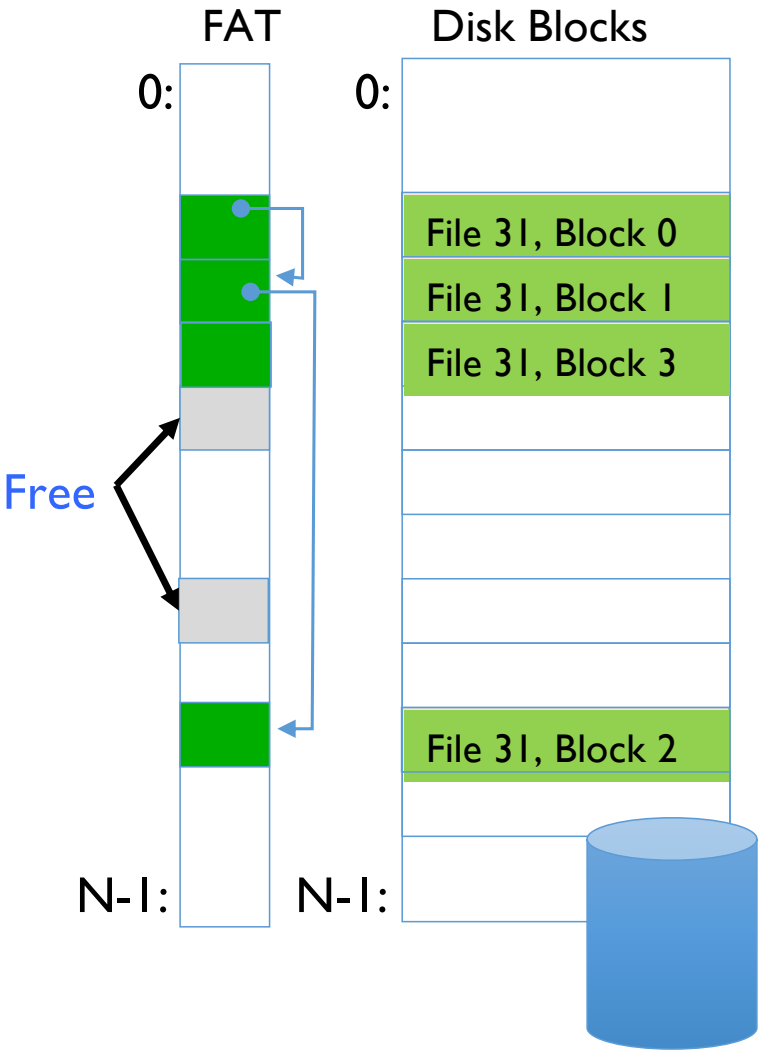
FAT: Expanding File (2/4)

- Find a new free block



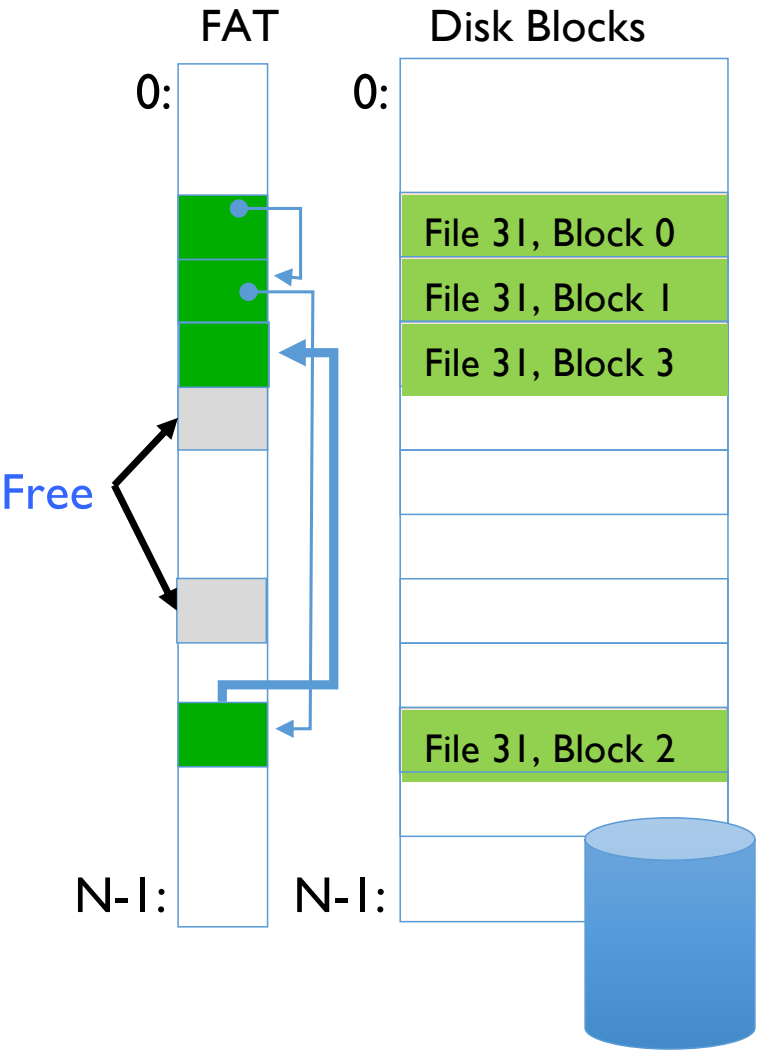
FAT: Expanding File (3/4)

- Add new content of the file at Block 3 corresponding to this FAT entry



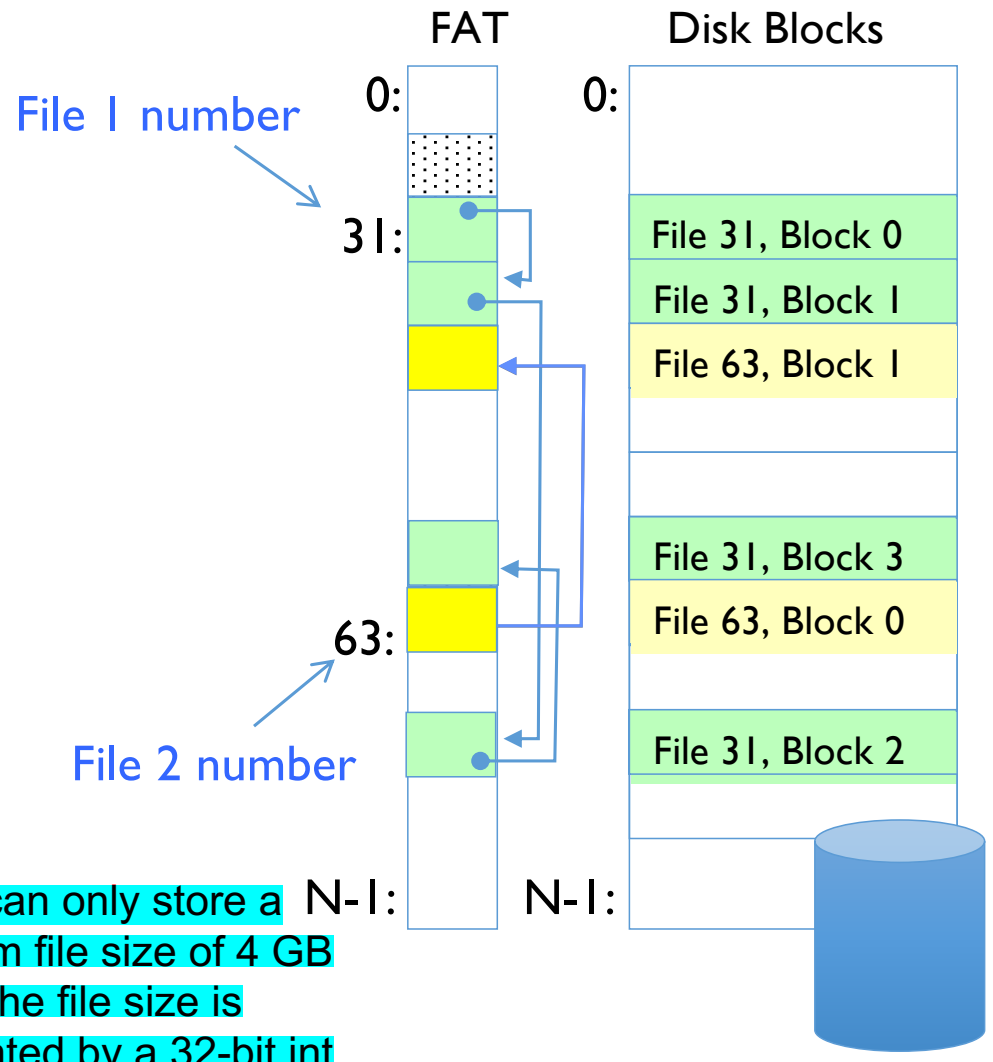
FAT: Expanding File (4/4)

- Link the new FAT entry with its predecessor



Storing the FAT

- Saved to disk when system is shut down
- **Copied into memory when OS is running**
  - Makes accesses, updates fast
  - Otherwise lots of random reads to locate the blocks of a file
- When drive is formatted, make all FAT entries 0



FAT can only store a maximum file size of 4 GB as the file size is represented by a 32-bit int

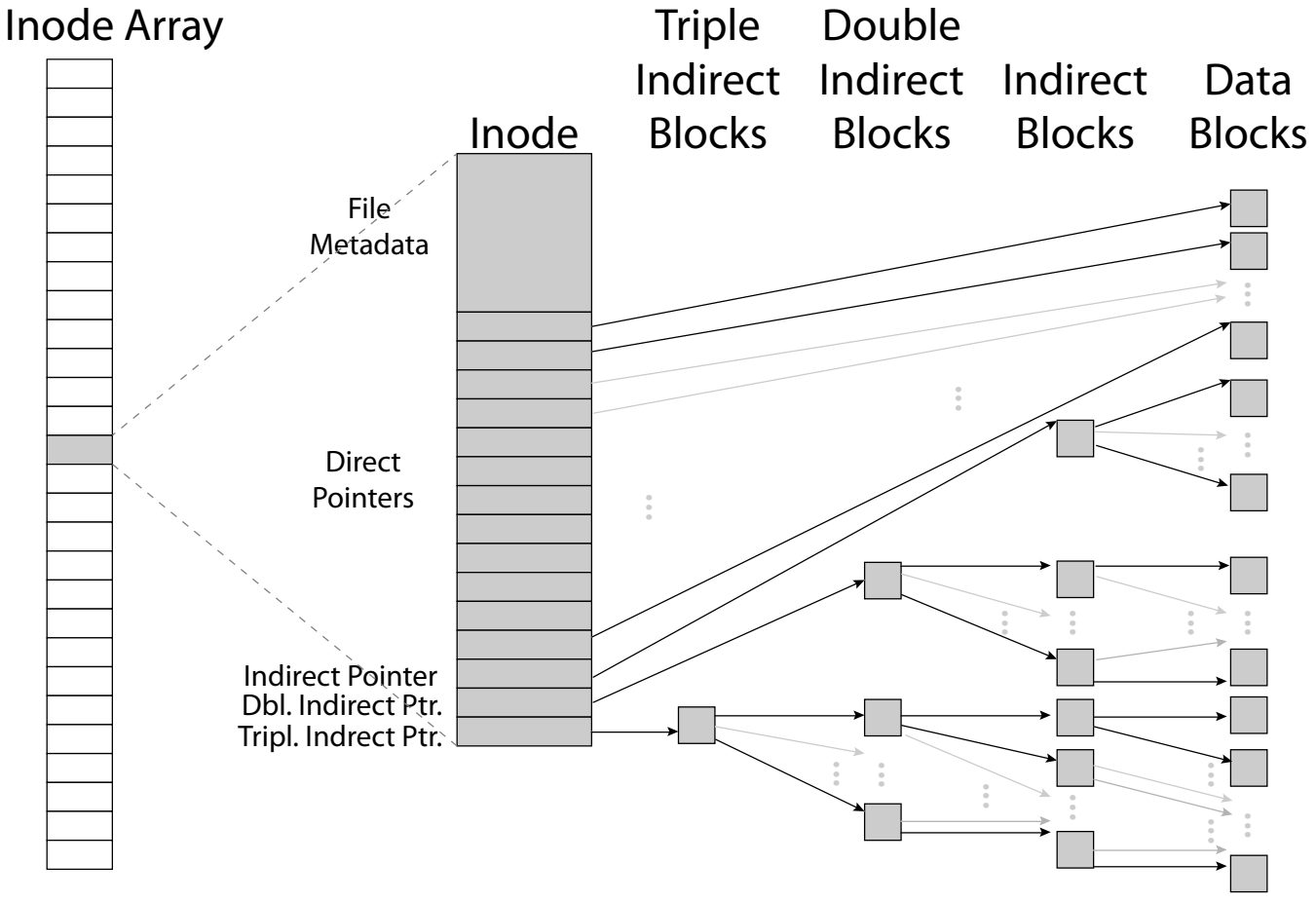
FAT Assessment

- FAT has no access rights
- FAT has no header in the file blocks
- Just gives an index into the FAT
  - File number is the block number

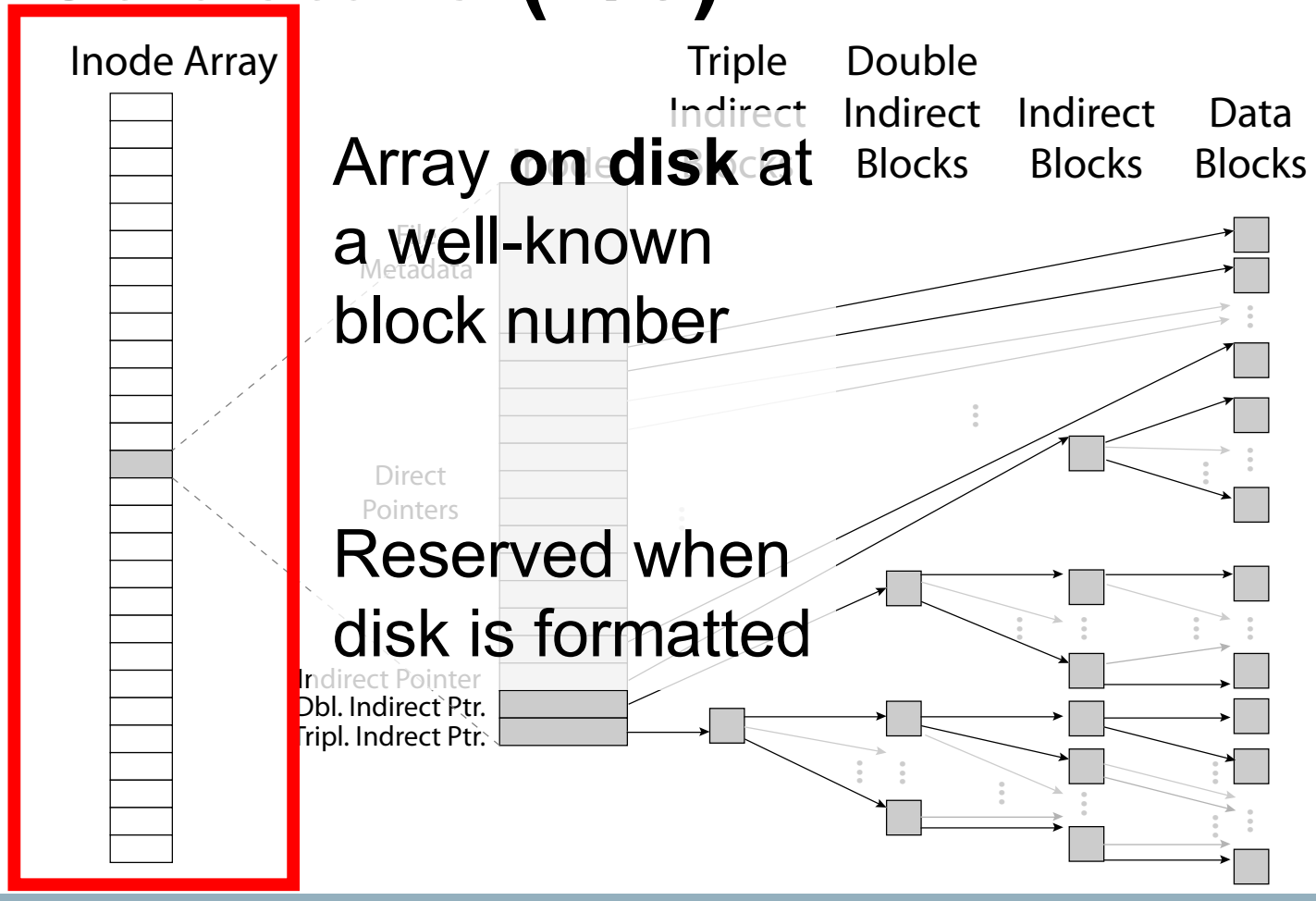
Inodes in Unix

- File number is index into a set of inode arrays
  - Suppose an Inode occupies 128 bytes
    - Byte offset of Inode-1000 = 128x1000 bytes
- Inode maintains a multi-level tree structure to find storage blocks for files
  - Great for little and large files
  - Asymmetric tree with fixed sized blocks
- Original **inode** format appeared in Berkeley Standard Distribution (BSD) Unix
  - Still used in modern Linux filesystems (e.g., ext4)

Inode Structure (1/5)



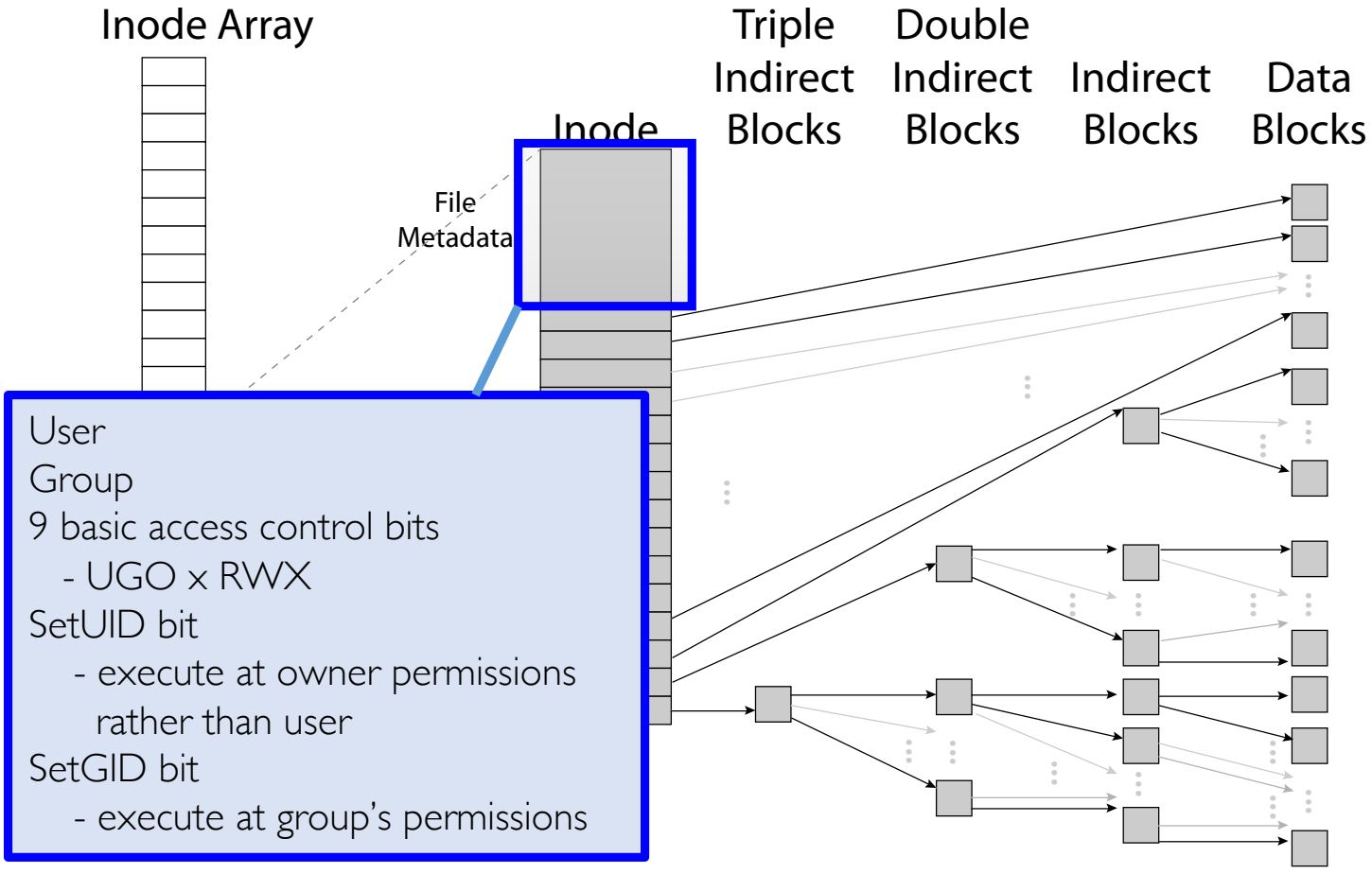
Inode Structure (2/5)



Array **on disk** at a well-known block number

Reserved when disk is formatted

Inode Structure (3/5)

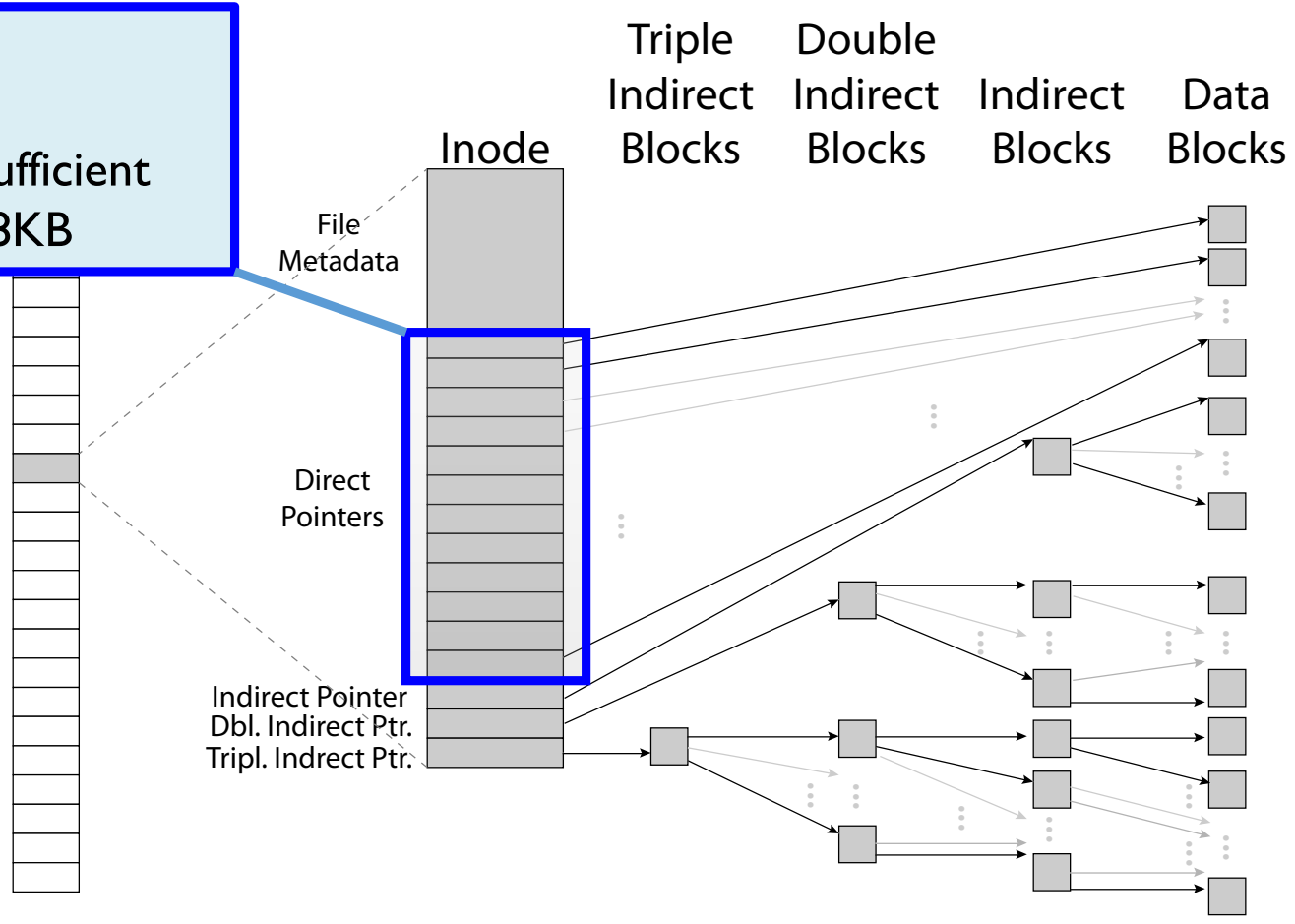


Inode Structure (4/5)

Direct pointers

4kB blocks ⇒ sufficient for files up to 48KB

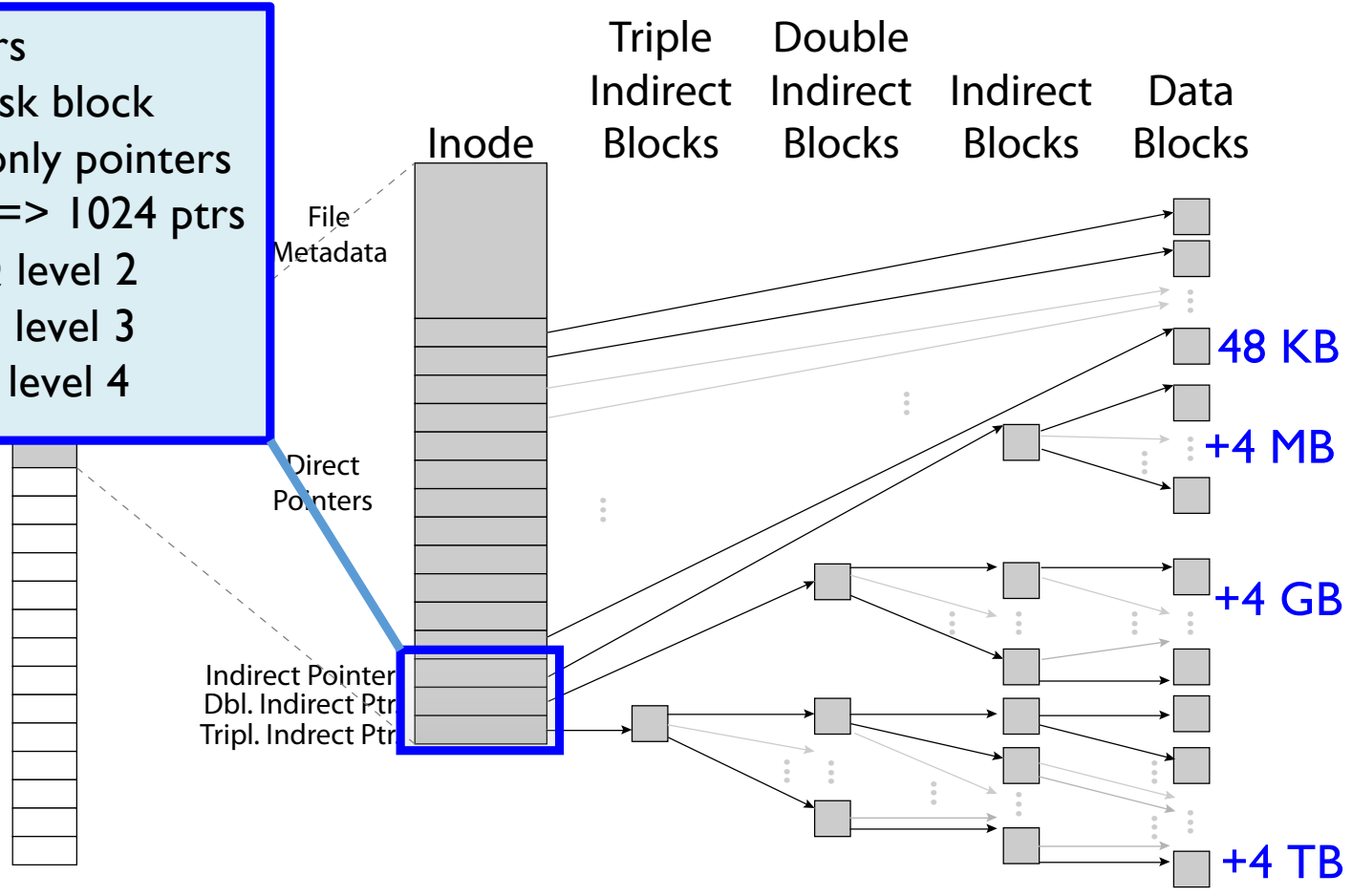
Small files: Level-1 has 12 pointers direct to data blocks (files up to 48KB)



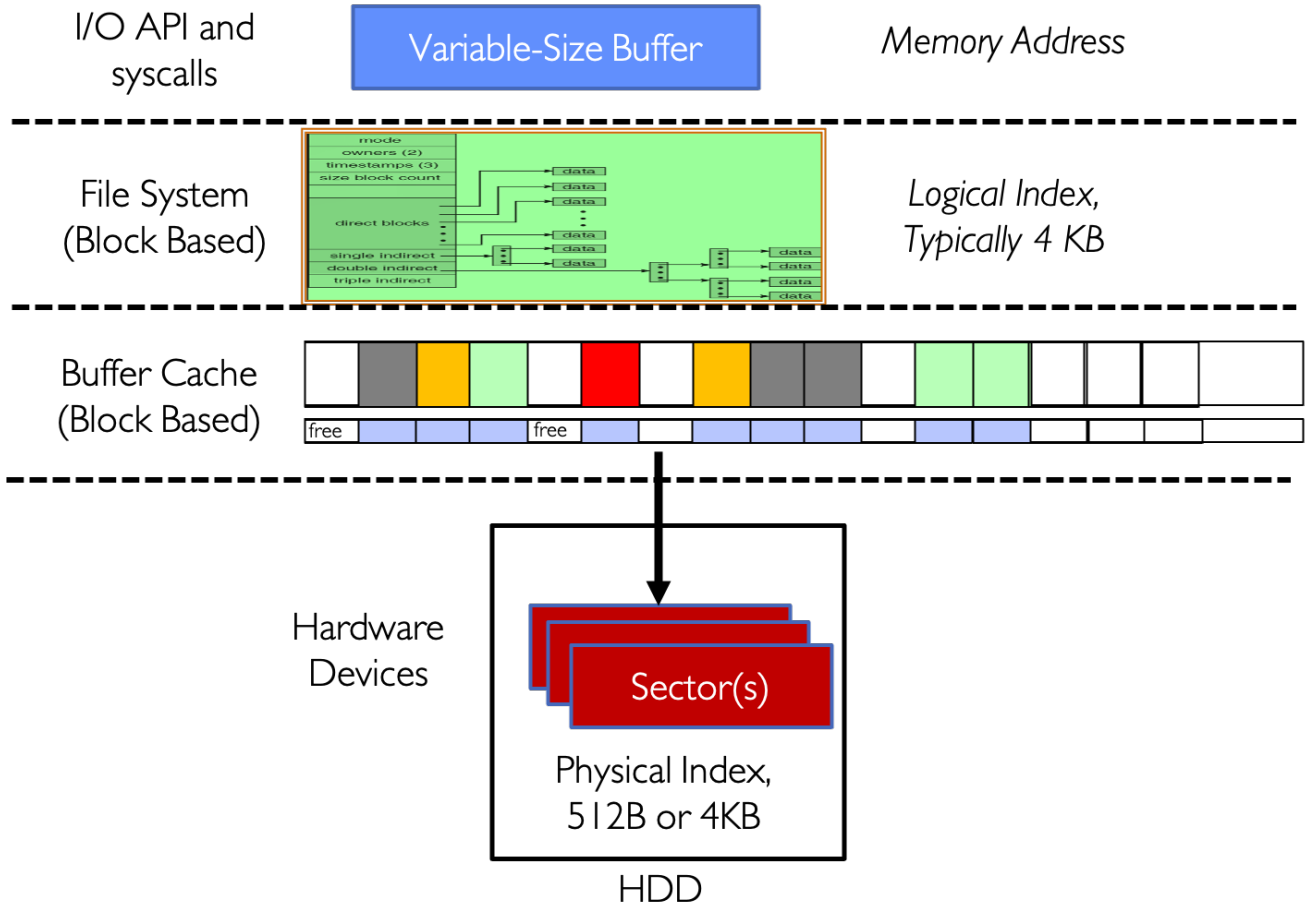
Inode Structure (5/5)

Indirect pointers
- point to a disk block containing only pointers
- 4 kB blocks => 1024 ptrs
  => 4 MB @ level 2
  => 4 GB @ level 3
  => 4 TB @ level 4

Large files: 1,2,3 level indirect pointers



Buffer Cache



Not block-sized or block-aligned access

Reuse of inodes, indirect blocks, data blocks

Speed up access to file system path and data

Apparent speed and flexibility is greater because of Buffer Cache

- Kernel *must* copy disk blocks to main memory (Buffer Cache) to access their contents and write them back if modified

Dealing With Persistent State

- Buffer Cache: write back dirty blocks periodically, even if used recently
  - Why? To minimize data loss in case of a crash
  - Linux does periodic flush every 30 seconds
- Not foolproof! Can still crash with dirty blocks in the cache
  - What if the dirty block was for a directory?
    - Lose pointer to file's inode (leak space)
    - **File system now in inconsistent state ☹**

Takeaway: File systems need recovery mechanisms

Resilience Using Transactions

- Closely related to critical sections for manipulating shared data structures
- Extend concept of an atomic update from *memory* to *persistent storage*
  - Atomically update multiple persistent data structures

Transaction Structure

- Begin a transaction – get transaction id
- Do a bunch of updates
  - If any fail along the way, roll-back
  - Or, if any conflicts with other transactions, roll-back
- Commit the transaction

Journaling File Systems

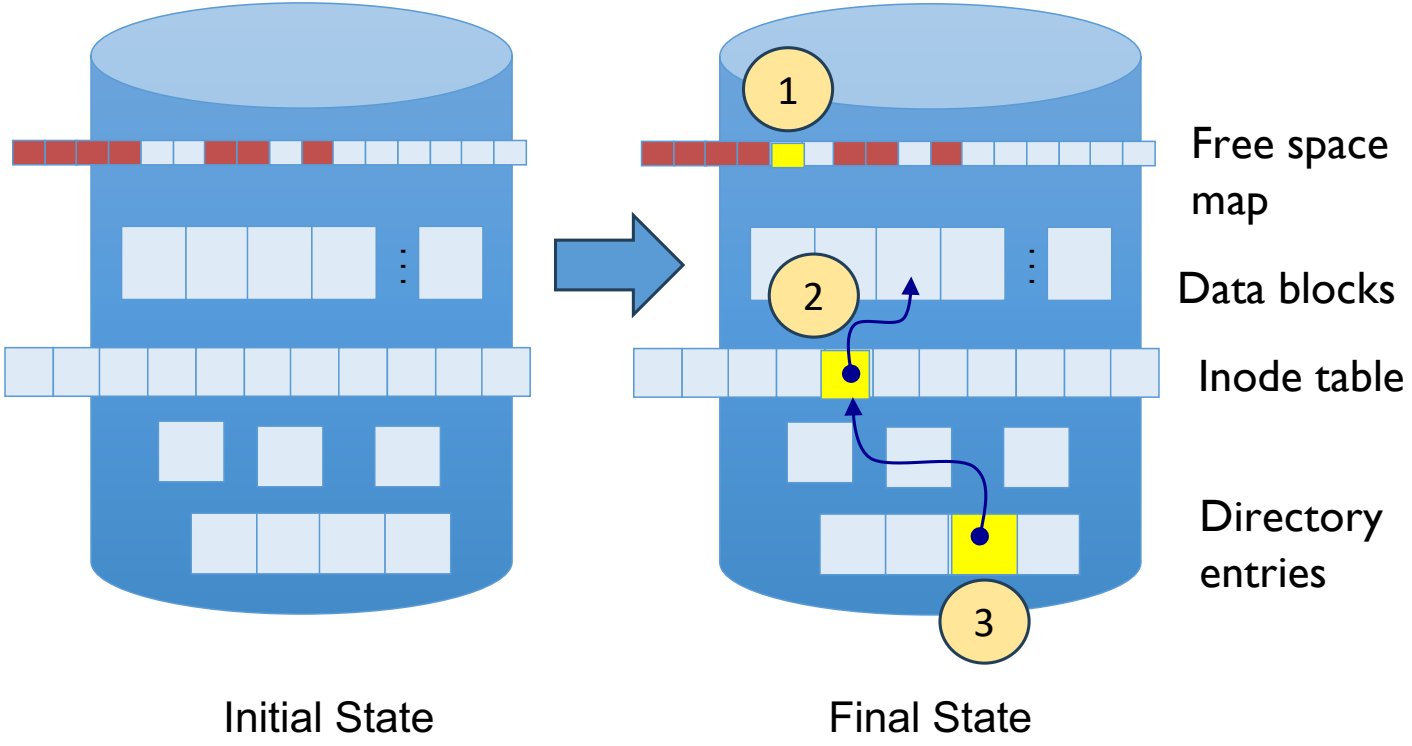
- Don't modify data structures on disk directly
- Write each update as transaction recorded in a log
  - Commonly called a *journal* or *intention list*
  - Also maintained on disk (allocate blocks for it when formatting)
- Once changes are in the log, they can be safely applied
  - e.g. modify inode pointers and directory mapping
- Garbage collection: once a change is applied, remove its entry from the log

Creating File Without Journaling

- Find free data block(s)
- Find free inode entry
- Find directory entry insertion point

---

1. Write map (i.e., mark used)
2. Write inode entry to point to block(s)
3. Write directory entry to point to inode

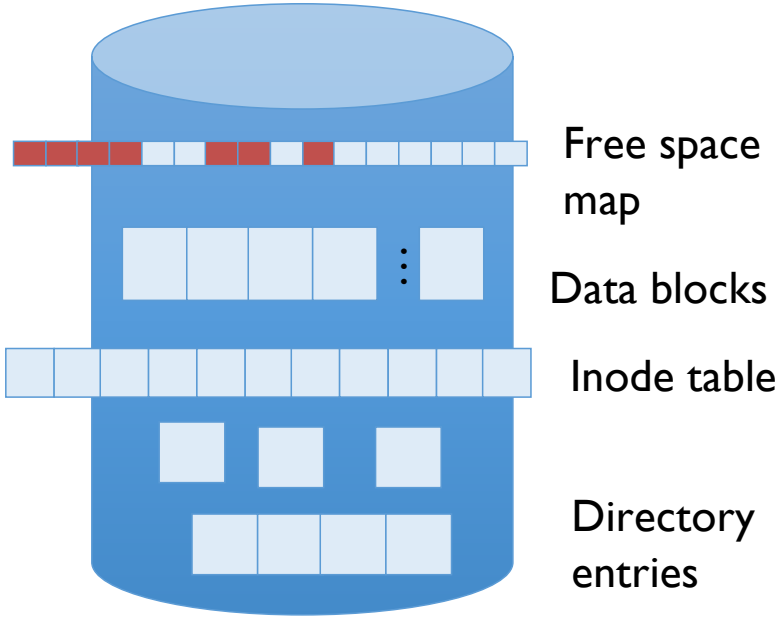


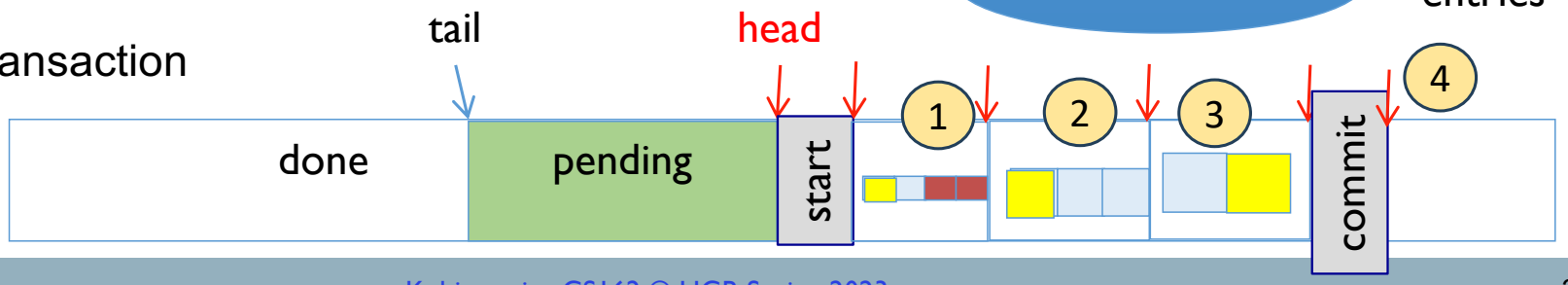
Creating File With Journaling (1/2)

- Find free data block(s)
- Find free inode entry
- Find directory entry insertion point

---

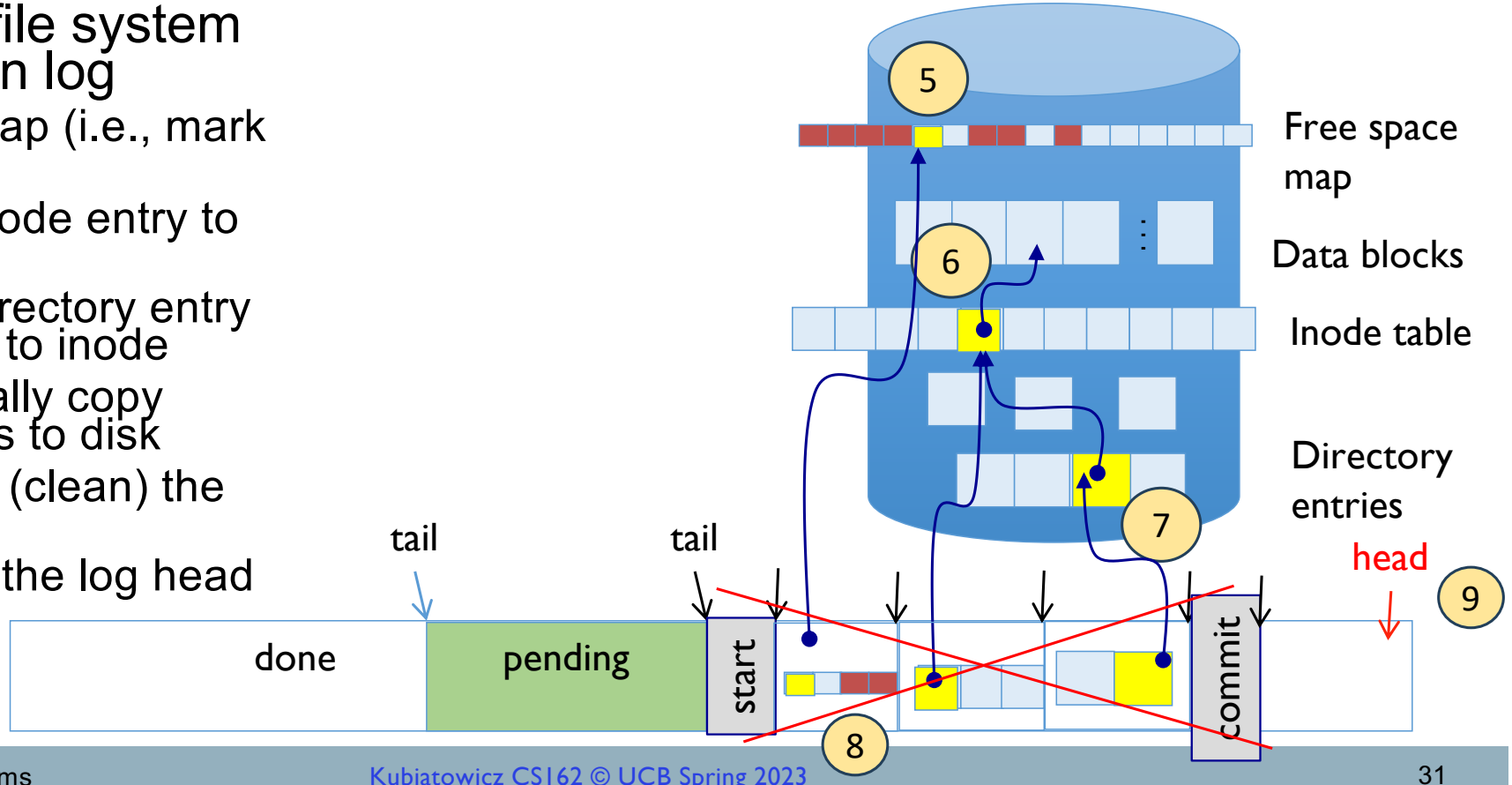
1. [log] Write map (used)
2. [log] Write inode entry to point to block(s)
3. [log] Write directory entry to point to inode
4. Commit the transaction





Creating File With Journaling (2/2)

- During commit all access to file system first looks in log
  4. Write map (i.e., mark used)
  5. Write inode entry to point
  6. Write directory entry to point to inode
  7. Eventually copy changes to disk
  8. Discard (clean) the old log
  9. Update the log head



Dealing with System Crash in Journaling

- Uncommitted transaction
  - Upon recovery scan the log
  - Detect transaction start with no commit
  - Discard log entries
  - Disk remains unchanged
- Committed transaction
  - Scan the log, find start
  - Find matching commit
  - Copy changes to the disk as usual

Where are we as of now

- **CSE231 Post Conditions**
  1. Students are able to create a Unix shell with complete clarity about process creation and process execution
  2. Students are able to write multi- threaded applications with synchronization primitives and ability to analyze effects of concurrency on process execution and correctness
  3. Students are able to analyze the impact of OS concepts, e.g. virtual memory, concurrency, on program execution and ability to fine-tune the program to run efficiently on a given OS
  4. Students are able to demonstrate deeper understanding of the Unix-like OSes and kernel programming

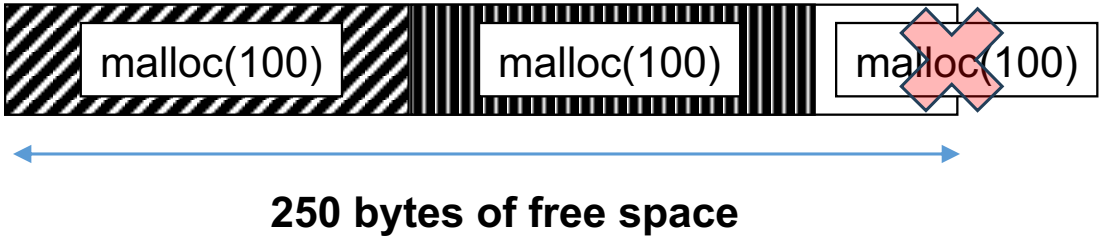
Next Lecture

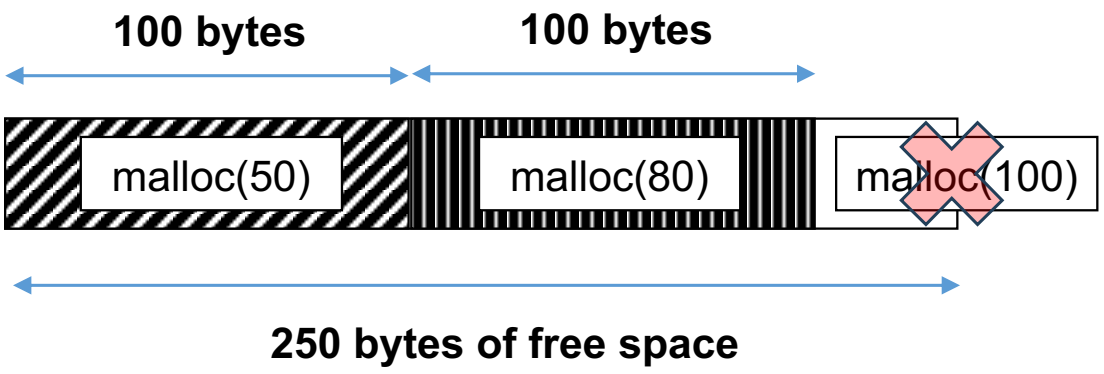
- End semester review

Lecture 26: End Semester Review

Vivek Kumar
Computer Science and Engineering
IIIT Delhi
vivekk@iiitd.ac.in

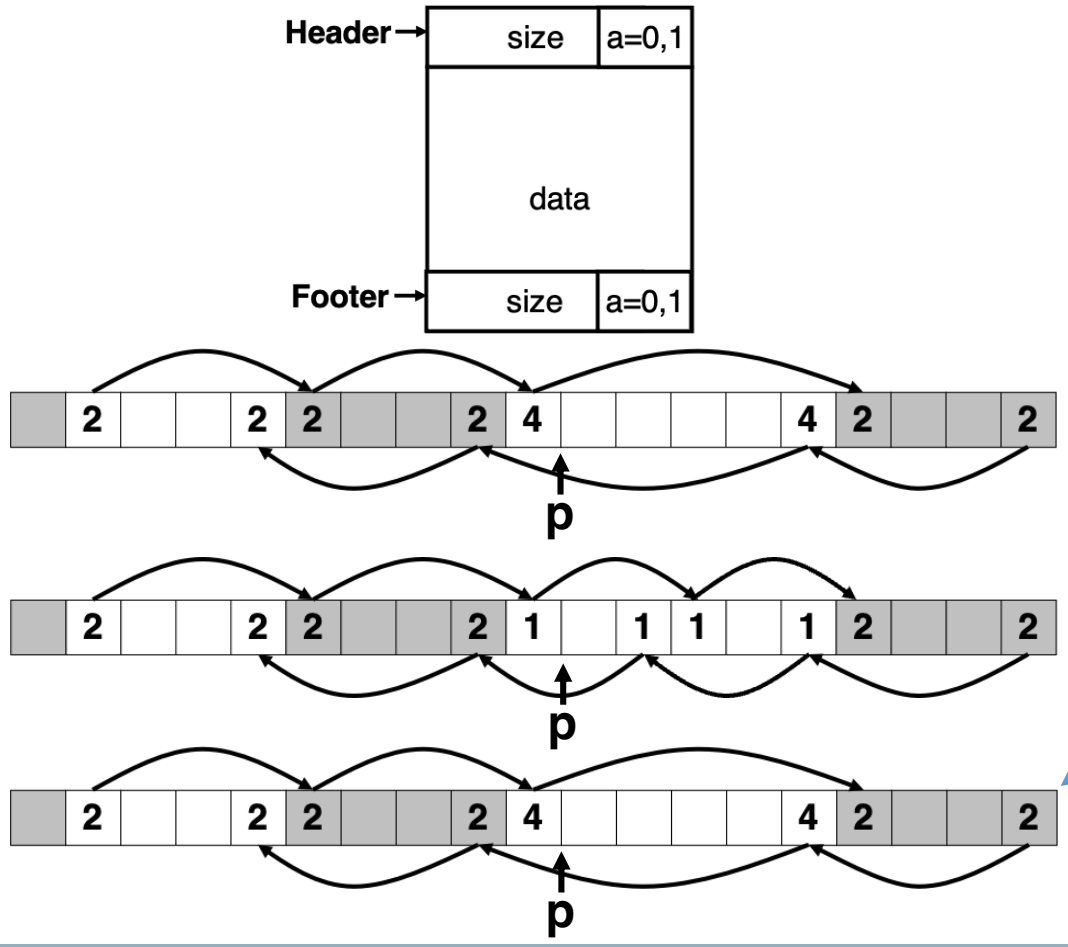
Fragmentation





- Segments of memory can become unusable due to the result of allocation scheme
- Two types of fragmentation
  - External fragmentation
    - Memory remains unallocated
    - Variable allocation sizes
  - Internal fragmentation
    - Memory is allocated but unused
    - Fixed allocation sizes

Malloc using Implicit List



- Each memory block has a header/footer
- Blocks connected as linked list
- Block splitting during malloc
  - E.g., p=malloc(1)
- Block coalescing during free
  - E.g., free(p)

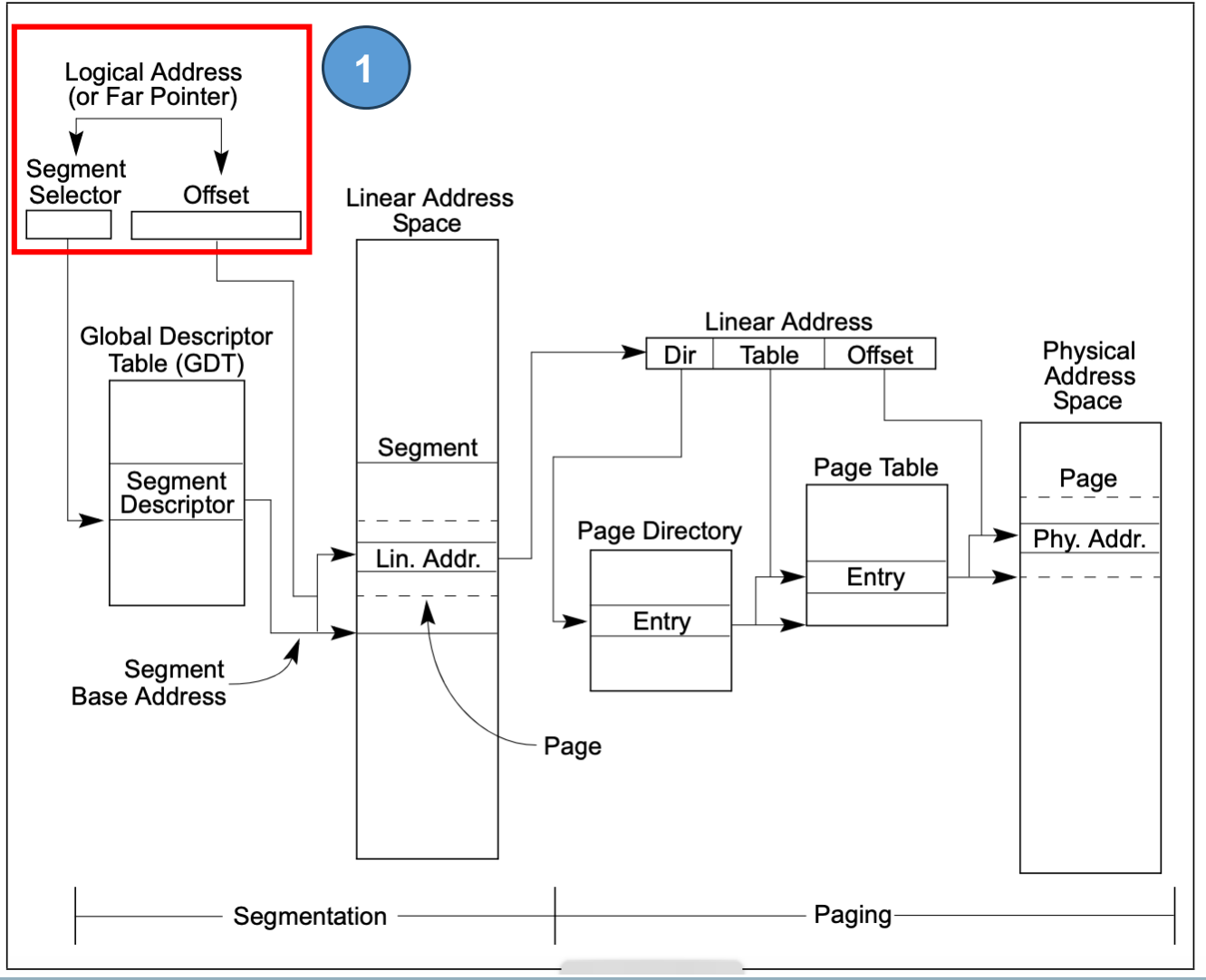
Address Spaces

- Using some mapping technique to translate the address seen by the user to the actual address on the RAM gives two different views of addresses
  - Virtual address space (as seen by the process)
  - Physical address space (as seen by the OS depending on DRAM size)

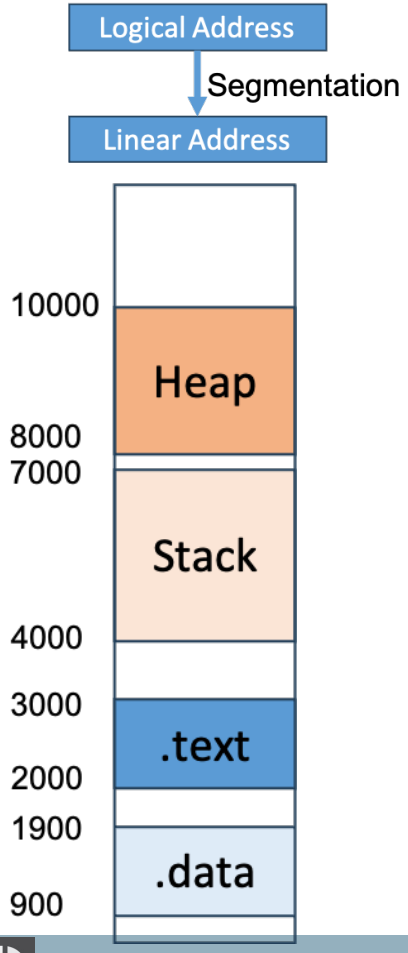
- Virtual address space: Set of $N = 2^n$ virtual addresses
  $\{0, 1, 2, 3, \ldots, N-1\}$

- Physical address space: Set of $M = 2^m$ physical addresses
  $\{0, 1, 2, 3, \ldots, M-1\}$

Memory Management in Intel x86



Logical Address to Linear Address

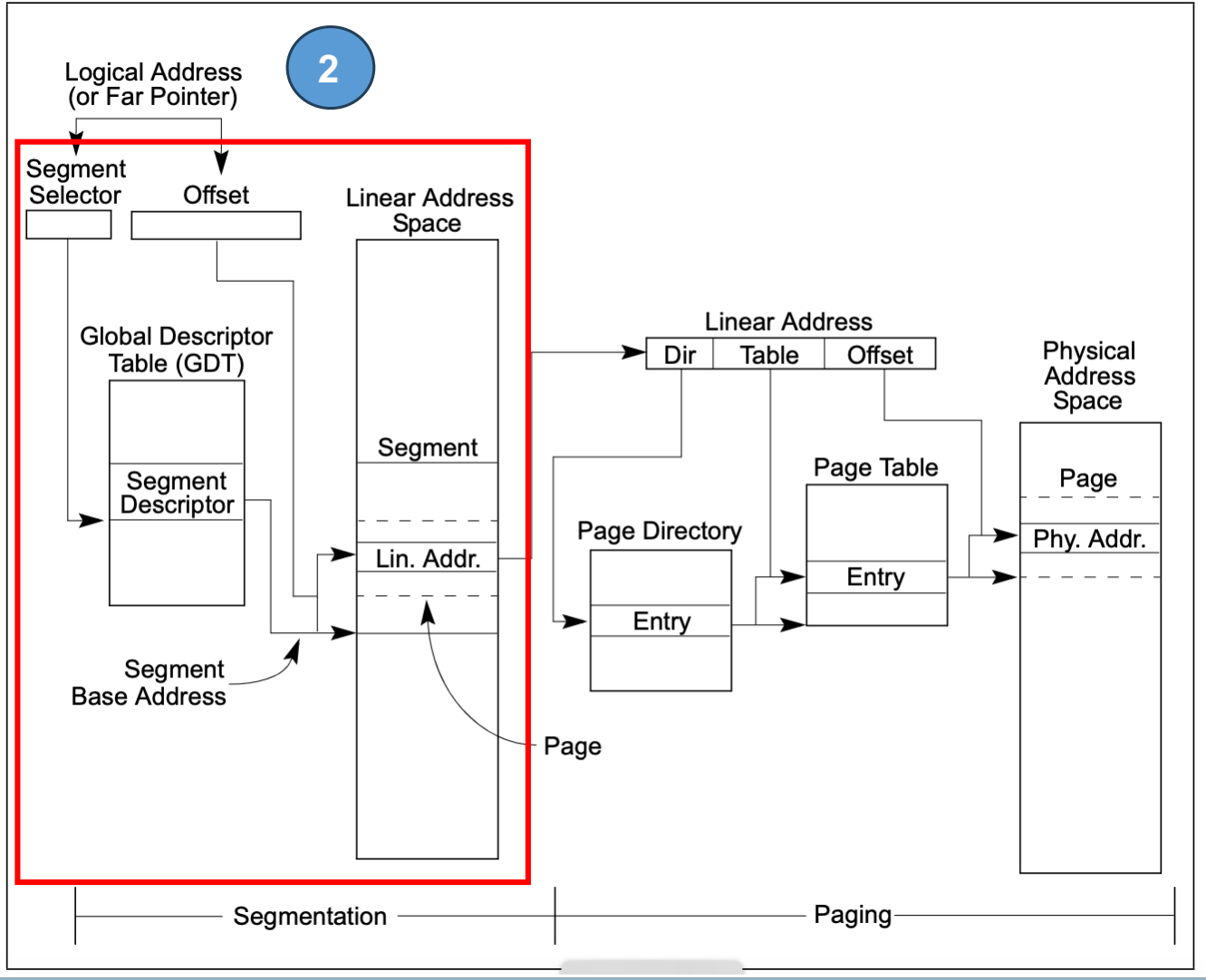


- The address visible to a programmer is actually called as Logical address, as it has an associated segment
  - **Segmentation** converts this logical address to linear address
- Each process **segment** has its own ("personal") base and bound values for its contiguous physical memory
- Each logical address can have few bits to specify the segment they belong (e.g., 2 bits to specify 4 different segments), and remaining bits to specify the offset into that segment

Segment Selector	Offset

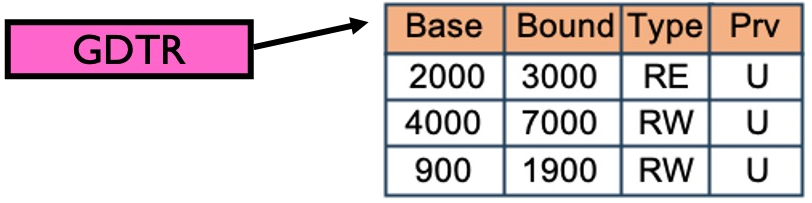
- Example: How to access virtual address 250 (binary: **11111010**)?
  1. Find the segment from the segment selector (e.g., data for first two bits)
  2. Find the offset (111010) = 58
  3. Find the **base** for this segment (Base=900)
  4. Check if offset within **bounds** (Bound=1000)
  5. Physical address = **958**

Memory Management in Intel x86



Hardware Support for Segmentation (1/2)

- x86 platform supports six segment registers per CPU
  - **CS (Code Segment)**
  - **SS (Stack Segment)**
  - **DS (Data Segment)**
  - ES, FG, and GS
    - General purpose



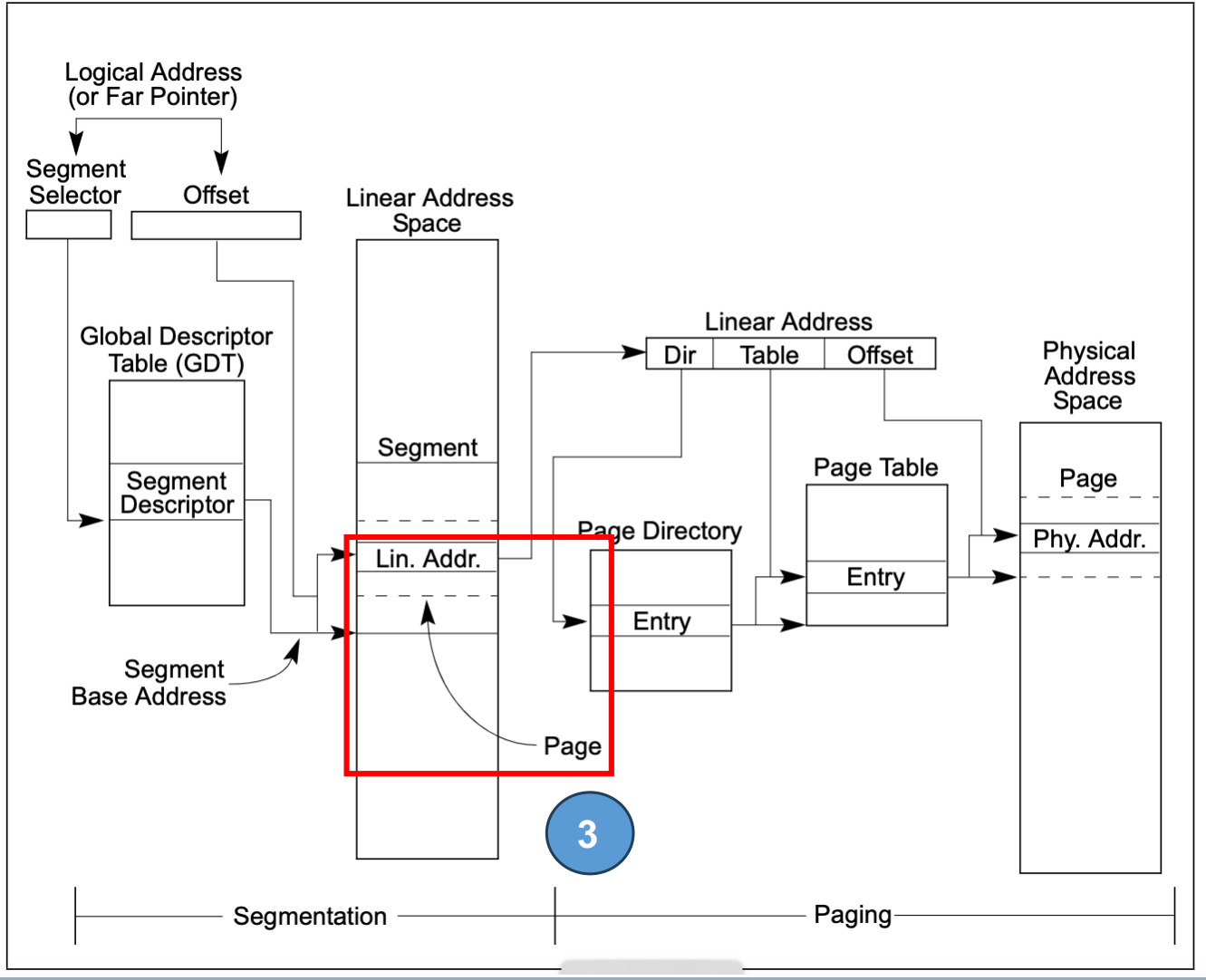
- Global Descriptor Table (GDT) that stores entries for segments from different processes
  - Base-bound pair, privilege mode for this physical memory segment (kernel vs user space), memory access type (read/write/executable), etc.
- GDT is stored in RAM
- Each CPU has a dedicated register (GDTR) that holds the starting address of this GDT

Hardware Support for Segmentation (2/2)

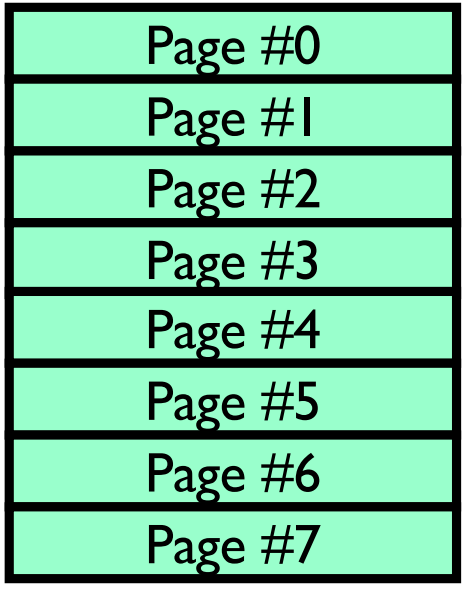
- GDT requires extra operations for each virtual memory accesses
  1. Load segment selector from the virtual address
  2. Load GDT **offset** by looking into segment register
  3. Load GDTR and add the above **offset** to get index
  4. Load base and bound values stored at that index
- Each segment register has a visible and a hidden part
  - Visible part is programable and it is an index into the GDT for that particular segment register
  - The hidden part is non-programable and it stores the base, bound (a.k.a limit) and access information for each segment register (cache)
    - It is cached automatically the first time a GDT index is accessed
    - No need to access GDT until a context switch is done
    - No extra CPU cycles wasted for accessing GDT in VA to PA translation

Visible Part	Hidden Part	
Segment Selector	Base Address, Limit, Access Information	CS
		SS
		DS
		ES
		FS
		GS

Memory Management in Intel x86



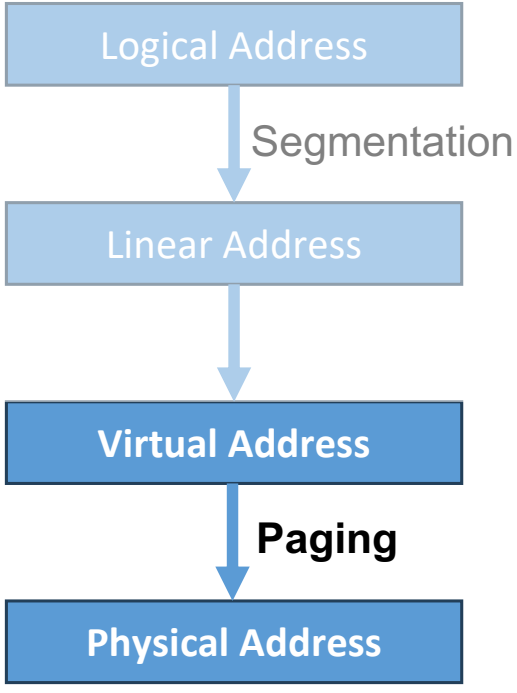
Paging



Physical Memory

- Segmentation is easy to implement but it has major drawbacks because segments could have variable sized memory allocation
  - Can lead to **external** fragmentation
  - To allocate space for a new process, segments of some other process has to be swapped to the disk (huge overhead)
- Paging solves both the above issues by dividing the available memory into small and fixed size blocks (pages)
  - The memory can then be viewed as an array of fixed sized slots
- Default page size on Linux is 4096 bytes (4KB)
  - Total number of pages in **32-bit** addressing mode is 2^{32}/4KB, i.e., 2^{20}
  - Total number of pages in **64-bit** addressing mode is 2^{64}/4KB, i.e., 2^{52}

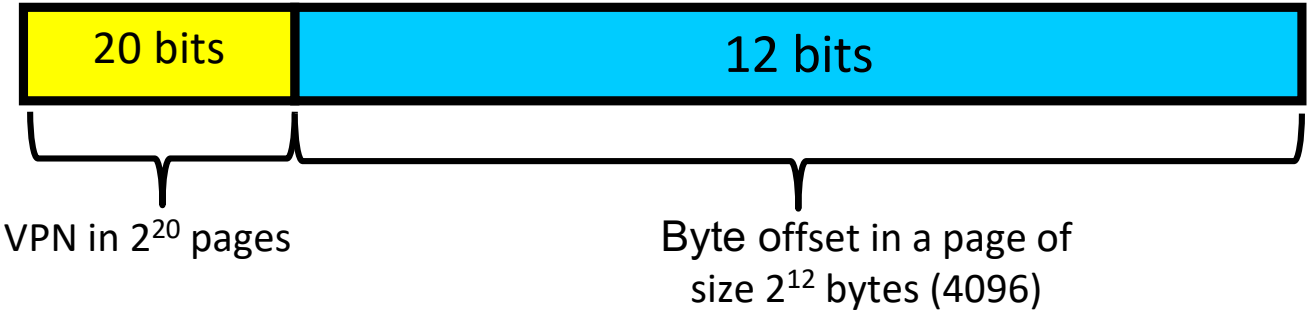
Segmentation Along with Paging



- Segmentation cannot be disabled in x86 processors, but address translation is complex using the segmentation-based approach
  - Hence, to simplify the address translation, segment registers are set to have the same base (0) and bound (2^{32}-1) values (Lecture #16)
- Segmentation converts the logical address to linear address, which is then converted into virtual address to support paging
- However, as the base and bound of each segment registers maps to entire 2^{32} memory size (4GB), logical address will be the same as virtual address

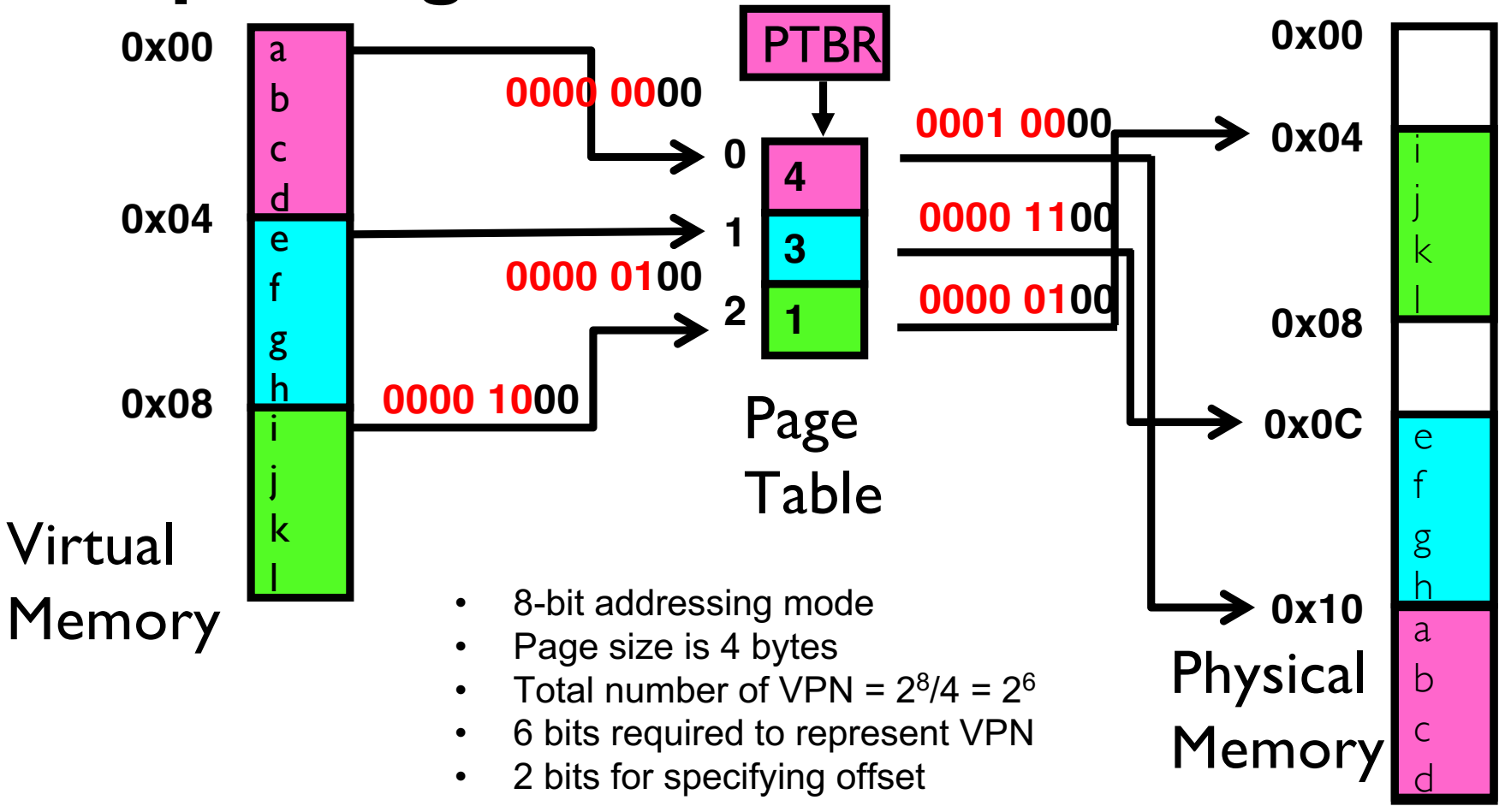
Two Components of a Virtual Address

- Dedicate few bits to identify VPN and the remaining bits to identify the offset into that VPN
- How many bits required to represent a VPN for pages of N bytes in a virtual memory of M bytes (total M/N pages)?
  - M=4GB (2^{32}) and N=4KB
    - Pages = $2^{32}/4096 = 2^{20}$ and bits required to identify 2^{20} pages are 20



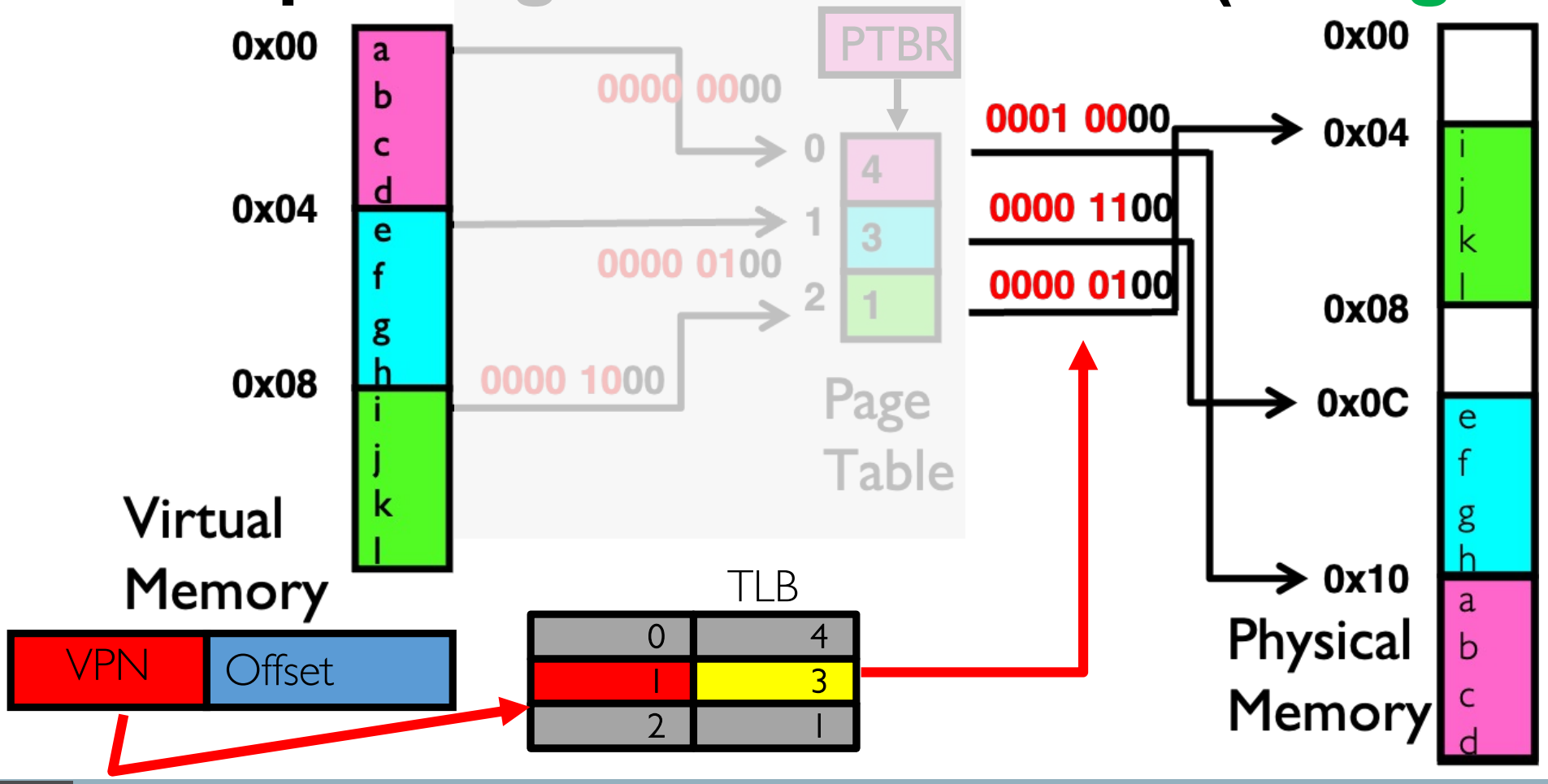
32-bit addressing with page size of 4KB

A Simple Page Table in Action

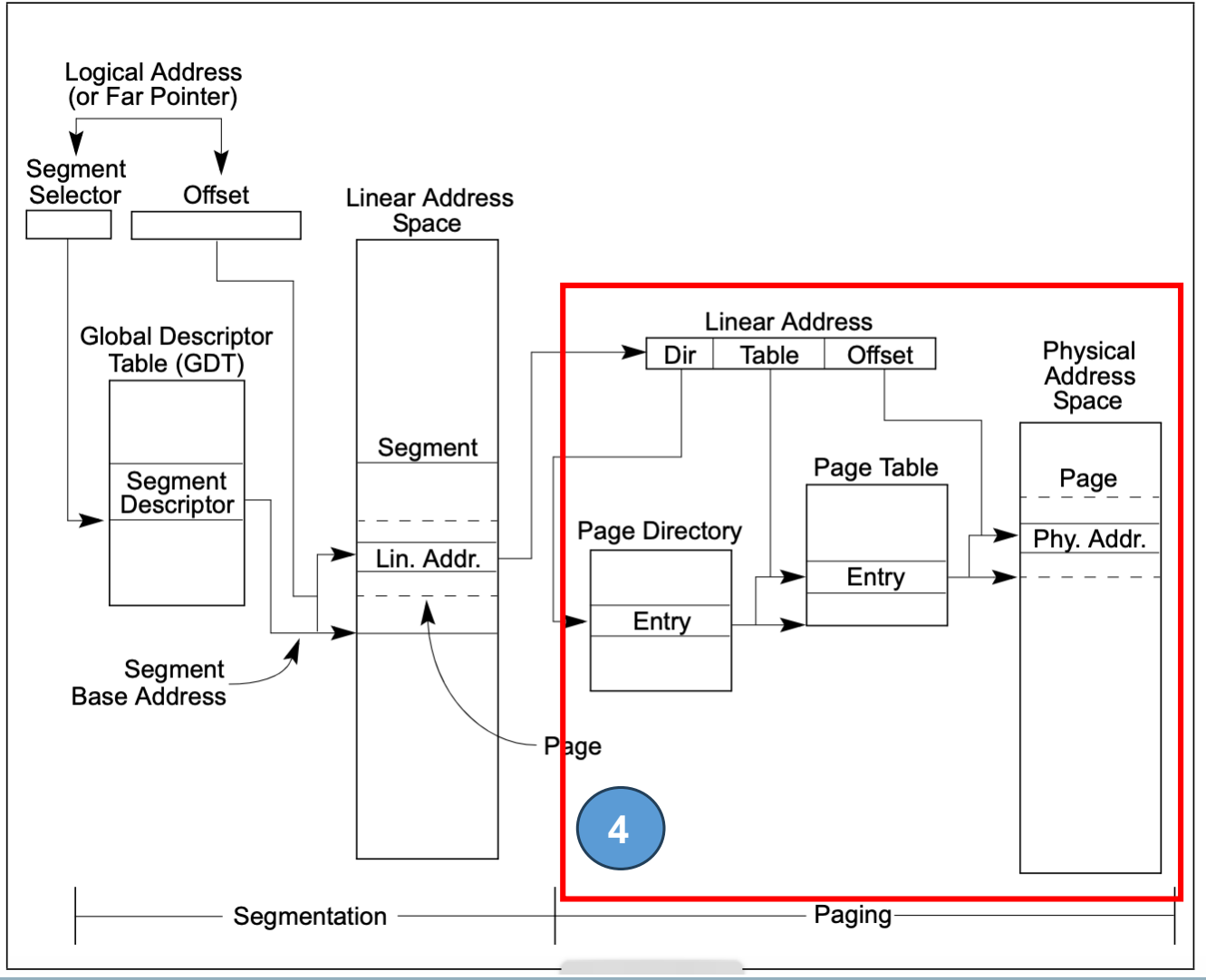


- 8-bit addressing mode
- Page size is 4 bytes
- Total number of VPN = $2^8/4 = 2^6$
- 6 bits required to represent VPN
- 2 bits for specifying offset

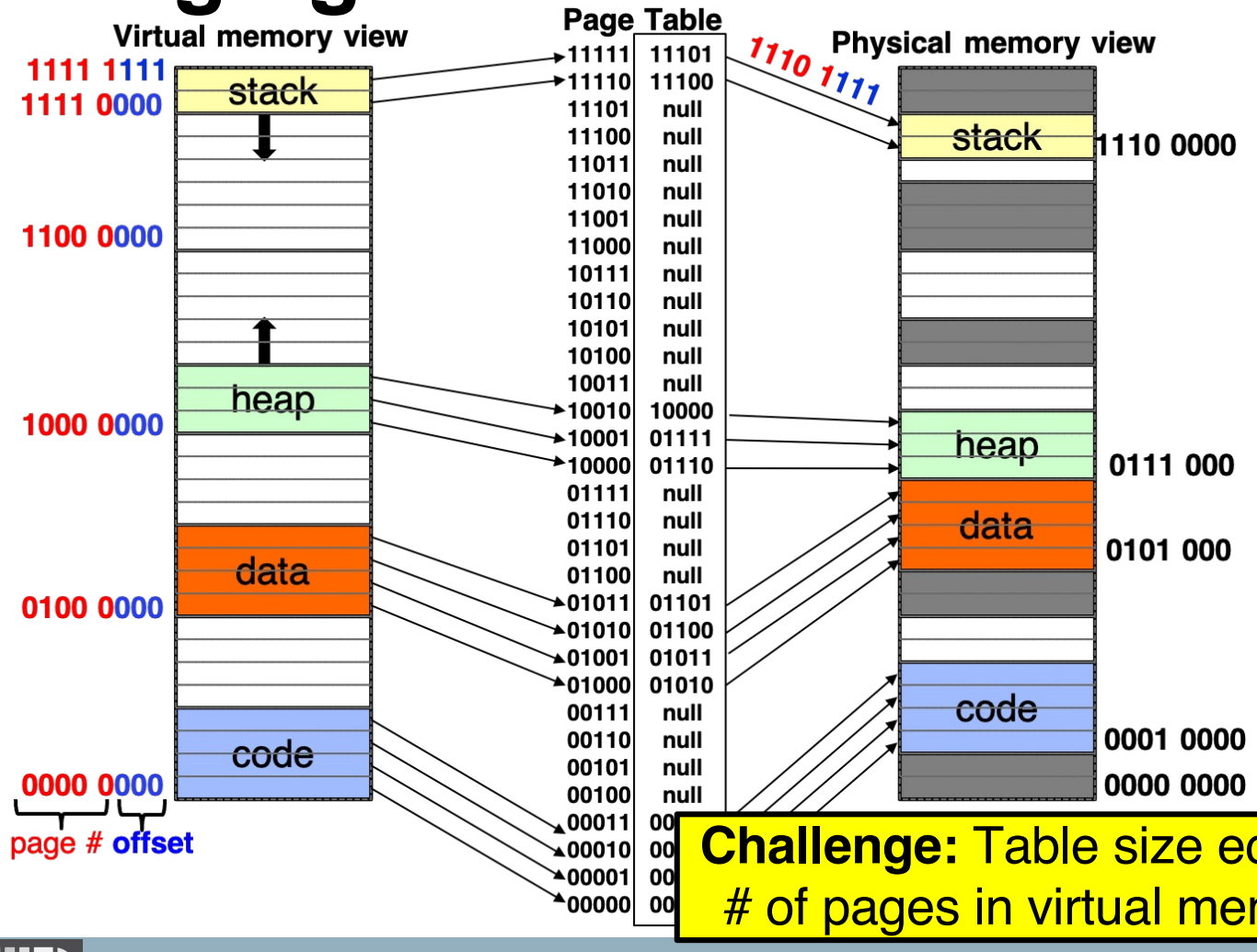
A Simple Page Table in Action (Using TLB)



Memory Management in Intel x86



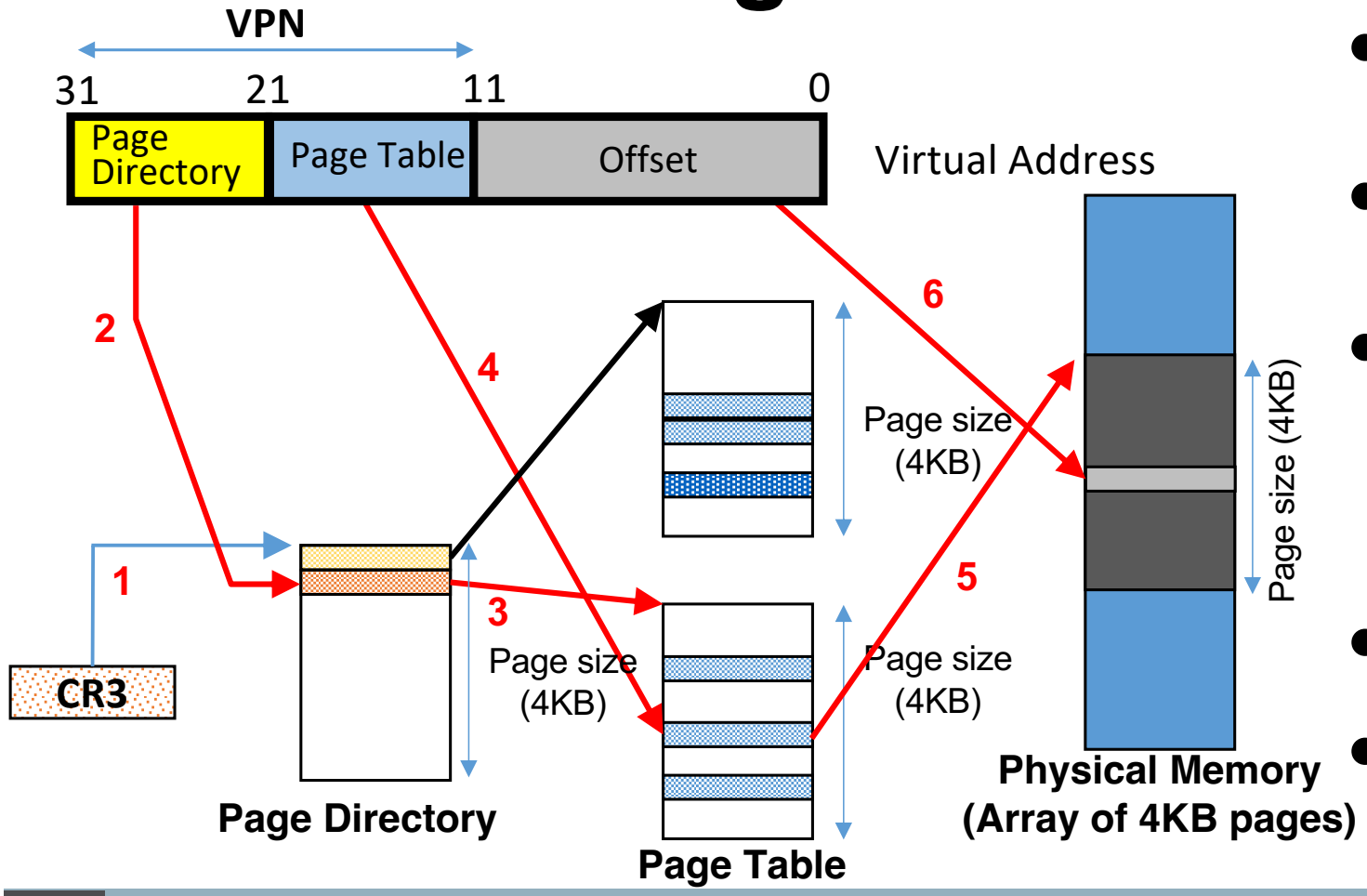
Paging



Challenge: Table size equal to # of pages in virtual memory!

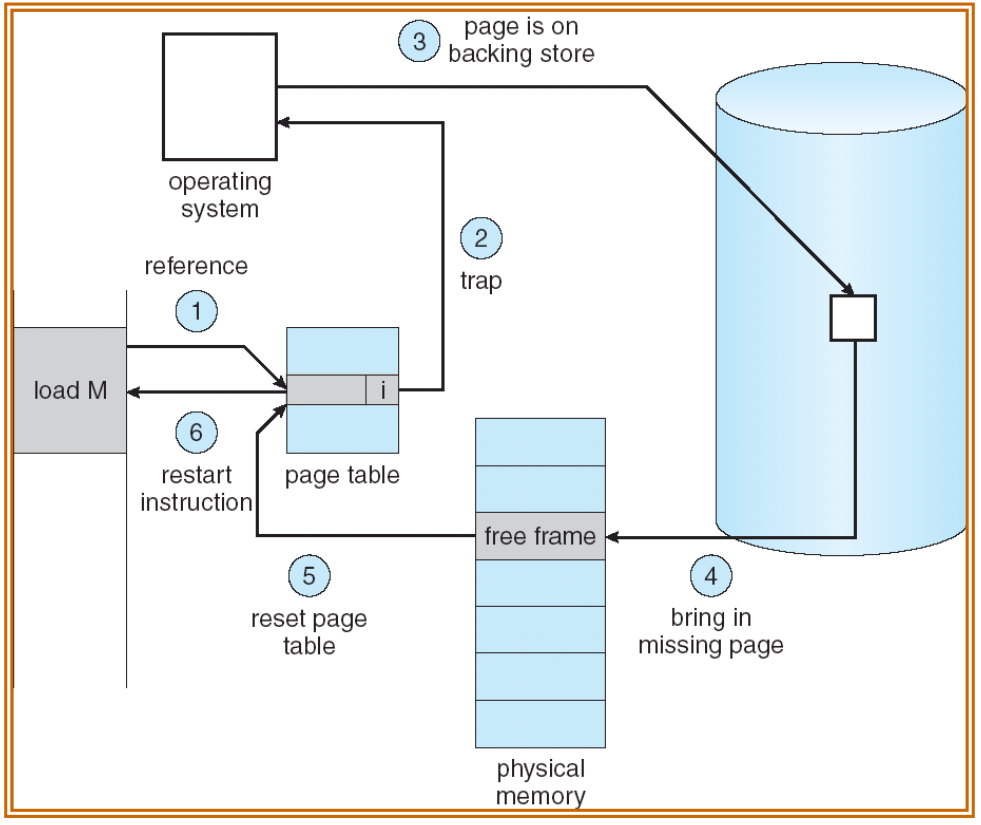
- Recall, process address space is not contiguous
  - Inter-segment memory is not contiguous
  - Only intra-segment virtual memory is contiguous (the physical memory may/may-not be contiguous)
- Page table until now is an **array** where the length of the array will be the total number of VPN being used in the process address space
  - Not all the VPNs will be used as the segments are not contiguous (there could be gaps between them)

Two-Level Page Table in x86



- VA stores page directory index, page table index, and address offset in PP
- CR3 register stores the starting address of the Page Directory
- Each PDE contains the Page Frame Number and a status bit
  - Status=0 if there is not a single valid entry in the PFN, otherwise status =1 (valid PFN only if status=1)
- Page Table Entry (PTE) is found from the PDE
- Physical Page Frame Number (PFN) is found from the PTE

Demand Paging



- It the mechanism to provide the illusion of infinite memory
- OS keeps some amount of DRAM always free by deciding some upper cap
  - Whenever the upper cap memory level is breached, unused pages are moved out to disk
- OS brings the pages into memory back into DRAM from disk when it is accessed
  - Happens during page fault
- PTE makes demand paging implementable using the Present bit
  - P=Valid ⇒ Page in memory, PTE points at physical page
  - P=Not Valid ⇒ Page not in memory

Page Replacement Policy

- FIFO
  - Throw out oldest page
  - Bad – throws out heavily used pages instead of infrequently used
- Random
  - Pick random page for every replacement
  - Better than FIFO, but it goes by luck – no guarantees!
- MIN (Minimum) – a.k.a. the optimal policy
  - Replace page that will be used furthest in the future
    - But we can't really know future!
- LRU (Least Recently Used)
  - Replace page that hasn't been used for the longest time
    - We cannot know the future, but we can use the history to predict the future!
  - Programs have locality, so if something not used for a while, unlikely to be used in the near future

IPC Within a Multicore Processor

- Inter-process communication in **shared memory**
  - Transfer of control from user space to kernel space and vice-versa
- Complicated IPC mechanism for communication
- OS has to reserve extra memory / resources
  - Separate heap, stack, .text segment, etc. for each process
  - Same copy of .text segment in each process
- Separate page table for each process
- Cost of IPC may exceed the cost of actual computation!

Array Sum using Pthread

```c
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
int A[SIZE]; // Initialization code elided
int array_sum(int low, int high) {
  int sum = 0;
  for (int i=low; i<high; i++) {
    sum += A[i];
  }
  return sum;
}

typedef struct {
  int low;
  int high;
  int sum;
} thread_args;

void *thread_func(void *ptr) {
  thread_args * t = ((thread_args *) ptr);
  t->sum = array_sum(t->low, t->high);
  return NULL;
}
```

```c
int main(int argc, char *argv[]) {
  int result;
  if (SIZE < 1024) {
    result = array_sum(0, SIZE);
  } else {
    pthread_t tid[NTHREADS];
    thread_args args[NTHREADS];
    int chunk = SIZE/NTHREADS;
    for (int i=0; i<NTHREADS; i++) {
        args[i].low=i*chunk; args[i].high=(i+1)*chunk;
        pthread_create(&tid[i],
                       NULL,
                       thread_func,
                       (void*) &args[i]);
    }
    for (int i=0; i<NTHREADS; i++) {
        pthread_join(tid[i] , NULL);
        result += args[i].sum;
    }
  }
  printf("Total Sum is %d\n", result);
  return 0;
}
```

Producer Consumer using Pthreads

```
pthread_mutex_lock(&mutex);
while(task_queue_size() == 0)
  pthread_cond_wait(&cond, &mutex);
}
task = pop_task_queue();
pthread_mutex_unlock(&mutex);
execute_task (task);
```

Consumer(s)

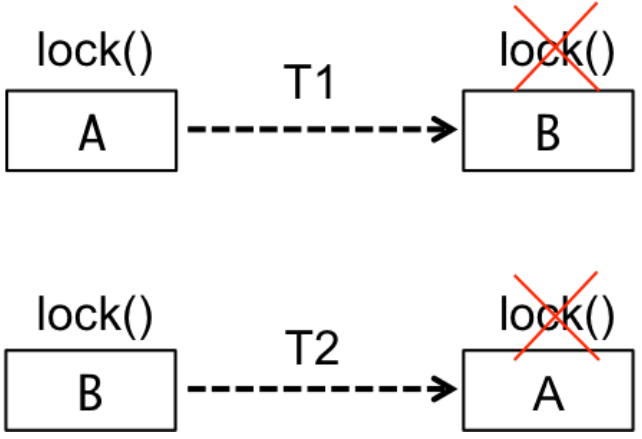
```
pthread_mutex_lock(&mutex);
int queue_size = task_queue_size();
push_task_queue(&task);
if(queue_size == 0) {
  pthread_cond_broadcast(&cond);
}
pthread_mutex_unlock(&mutex);
```

Producer

- pthread_cond_**wait** causes the current thread to relinquish the CPU and wait until another thread invokes the **signal** or the **broadcast**
- Upon call for **wait**, the thread releases ownership of the mutex and waits until another thread signals the waiting threads

Deadlock

```
class Transfer {
    Account A, B;
    int amount;

    void run() {
        A.lock();
        B.lock();
        A.debit(amount);
        B.credit(amount);
        A.unlock();
        B.unlock();
    }
}
```



- Deadlock occurs when multiple threads need the same locks but obtain them in different order
  - It could be avoided by using lock ordering
    - Ensure that all locks are taken in same order by any thread

I hope you enjoyed the course..

All the best for your exams !