EE309
Lecture 3: EE209/EE485
Review 2

INSU YUN (윤인수)

School of Electrical Engineering, KAIST

[Lecture slides based on EE209]

Lecture 17:
Memory Management

KAIST EE

Motivation for Memory Hierarchy

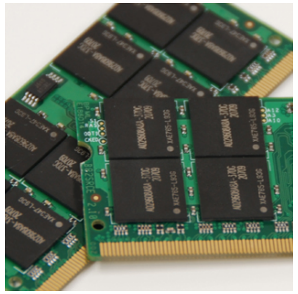
- ***Faster*** storage technologies are ***more expensive***
  - Cost more money per byte
  - Have lower storage capacity
  - Require more power and generate more heat
- The gap between processing and memory is widening
  - Processors have been getting faster and faster
  - Memory speed is not improving as dramatically
- Well-written programs tend to exhibit ***good locality***
  - Across time: repeatedly referencing the same variables
  - Across space: often accessing other variables located nearby

Want the ***speed*** of fast storage with the ***cost*** and ***capacity*** of slow storage

Key idea: ***memory hierarchy***!

Simple Three-Level Hierarchy

- **Registers**
  - Usually reside directly on the processor chip
  - Essentially no latency, referenced directly in instructions
  - Low capacity (e.g., 32-512 bytes)
- **Main memory**
  - Around 100 times slower than a clock cycle
  - Constant access time for any memory location
  - Modest capacity (e.g., 1 GB-512GB)

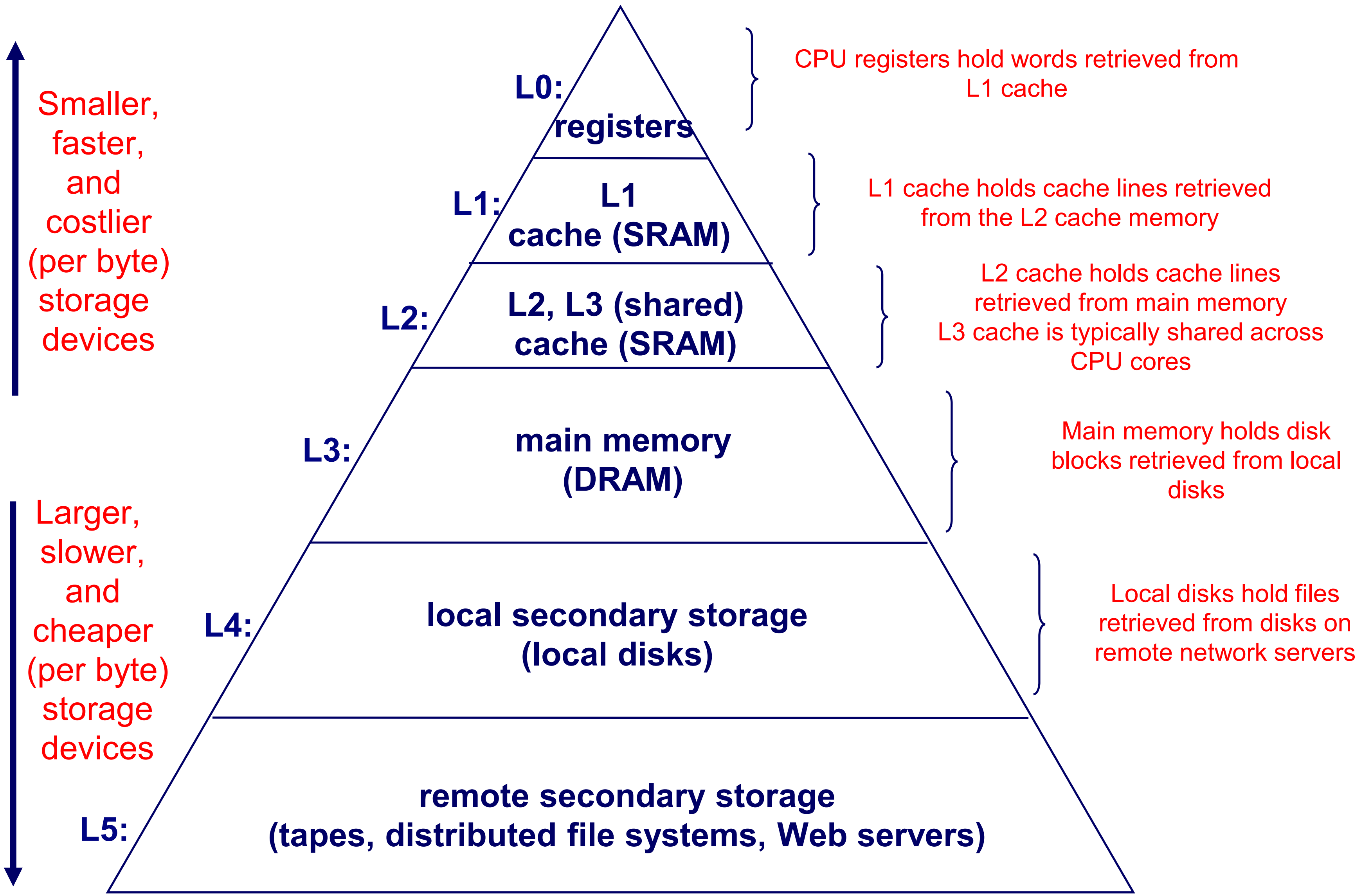


- **Disk**
  - Around 100,000 times slower than main memory
  - Faster when accessing many bytes in a row
  - High capacity (e.g., 1-10s of TB)

Widening Processor/Memory Gap

- Gap in speed increasing from 1986 to 2000
  - CPU speed improved ~55% per year
  - Main memory speed improved only ~10% per year
- Main memory as major performance bottleneck
  - Many programs stall waiting for reads and writes to finish
- Changes in the memory hierarchy
  - Increasing the number of registers
    - 8 integer registers in the x86 vs 16 in x86_64
  - Adding caches between registers and main memory
    - Level-1, -2, -3 cache on chip

An Example Memory Hierarchy



Locality of Reference

- Two kinds of locality
  - **Temporal locality**: recently referenced items are likely to be referenced in near future
  - **Spatial locality**: items with nearby addresses tend to be referenced close together in time

Locality of Reference

- Two kinds of locality
  - **Temporal locality**: recently referenced items are likely to be referenced in near future
  - **Spatial locality**: items with nearby addresses tend to be referenced close together in time
- Locality example

```
sum = 0;
for (i = 0; i < n; i++)
   sum += a[i];
return sum;
```

Locality of Reference

- Two kinds of locality
  - **Temporal locality**: recently referenced items are likely to be referenced in near future
  - **Spatial locality**: items with nearby addresses tend to be referenced close together in time

```
sum = 0;
for (i = 0; i < n; i++)
   sum += a[i];
return sum;
```

- Locality example
  - Program data
    - Temporal: the variable **sum**
    - Spatial: variable `a[i+1]` accessed soon after `a[i]`

Locality of Reference

- Two kinds of locality
  - **Temporal locality**: recently referenced items are likely to be referenced in near future
  - **Spatial locality**: items with nearby addresses tend to be referenced close together in time

```
sum = 0;
for (i = 0; i < n; i++)
    sum += a[i];
return sum;
```

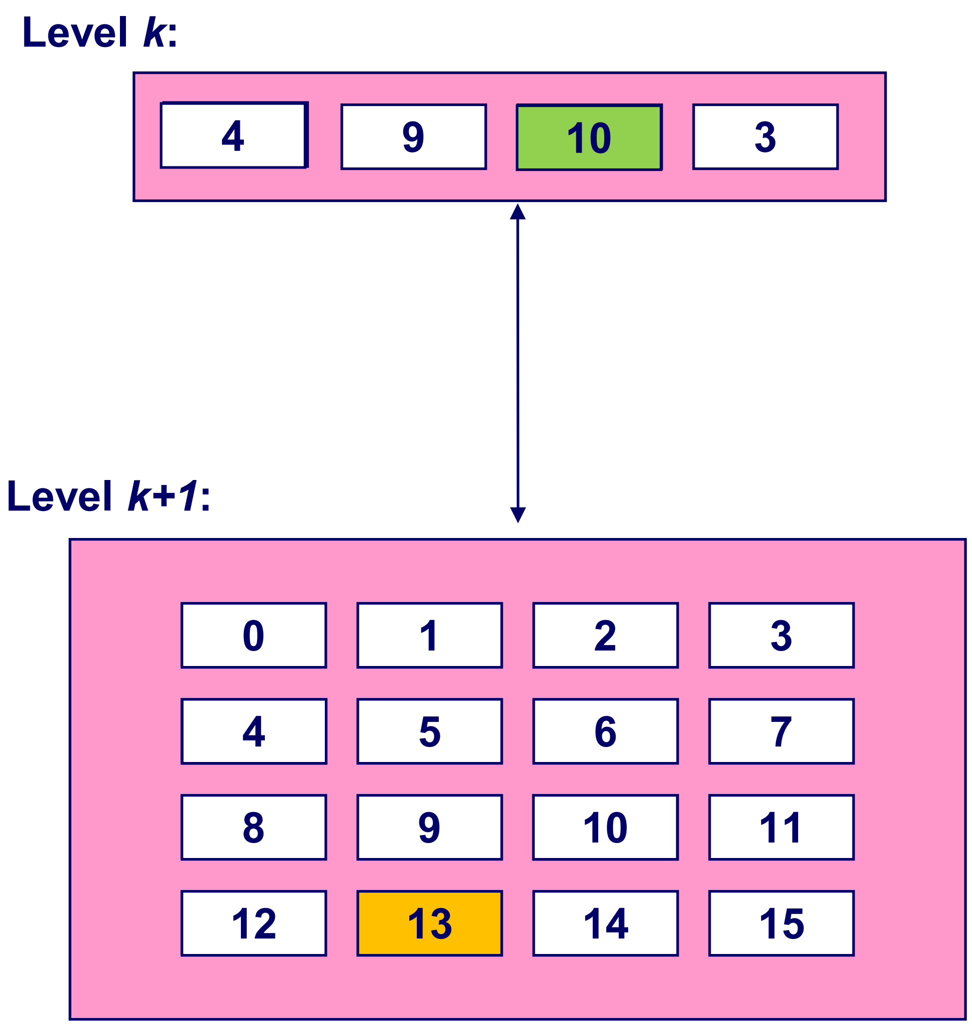
- Locality example
  - Program data
    - Temporal: the variable **sum**
    - Spatial: variable `a[i+1]` accessed soon after `a[i]`
  - Instructions
    - Temporal: cycle through the for-loop repeatedly
    - Spatial: reference instructions in sequence

Locality Makes Caching Effective

- Cache
  - Smaller and faster storage device that acts as a staging area
  - … for a *subset* of the data in a larger, slower device
- Caching and the memory hierarchy
  - Storage device at level *k* is a cache for level *k+1*
  - Registers as cache of L1/L2 cache and main memory
  - Main memory as a cache for the disk
  - Disk as a cache of files from remote storage
- ***Locality*** of access is the key
  - Most accesses satisfied by first few (faster) levels
  - Very few accesses go to the last few (slower) levels

Cache Hit and Miss

- Cache hit
  - Program accesses a block available in the cache
  - Satisfy directly from cache
  - e.g., request for “10”
- Cache miss
  - Program accesses a block not available in the cache
  - Bring item into the cache
  - e.g., request for “13”
- Where to place the item?
- Which item to evict?

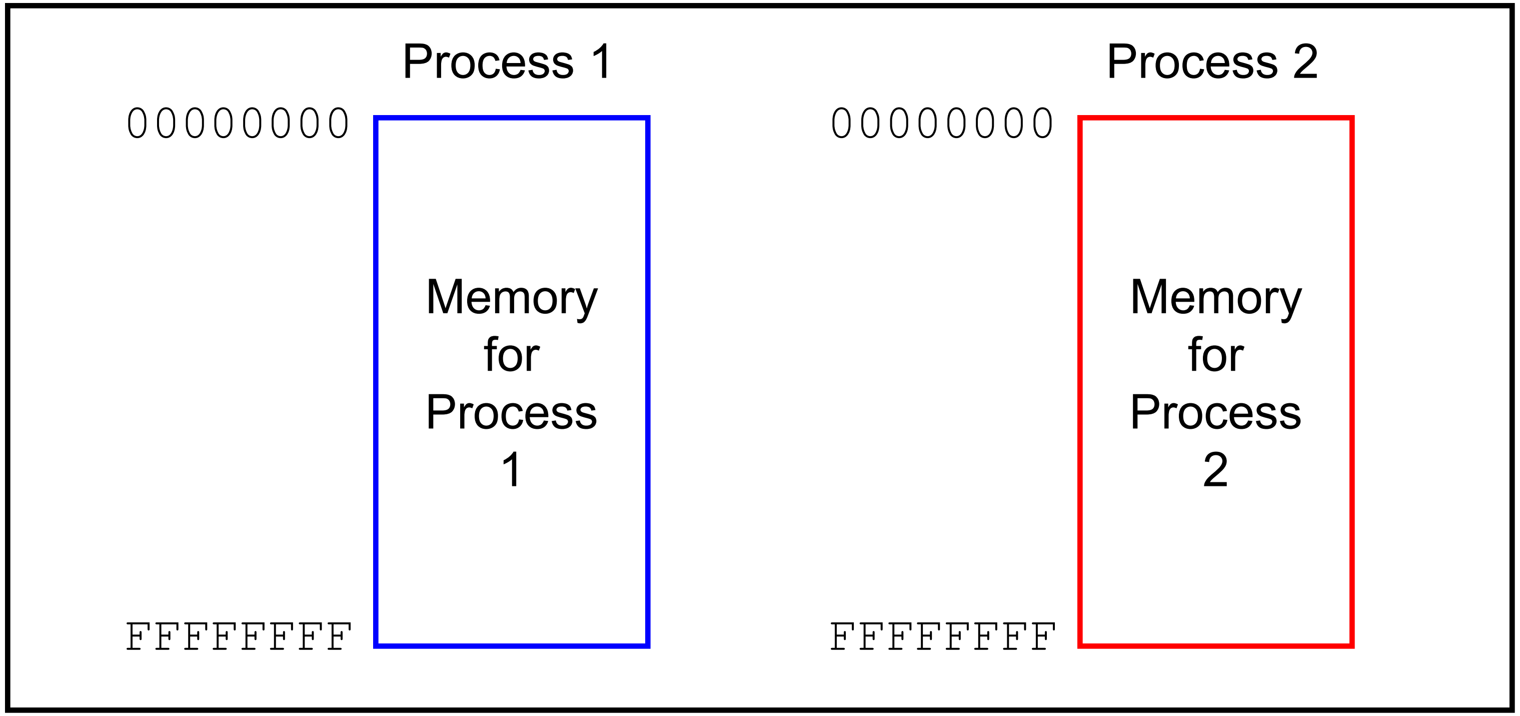


Automatic Allocation: Virtual Memory

- Give programmer the illusion of a very large memory
  - Large: 4 GB of memory with 32-bit addresses
  - Uniform: contiguous memory locations, from 0 to 2^{32}-1
- Independent of
  - the actual size of the main memory
  - the presence of any other processes sharing the computer
- **Key idea #1**: separate "address" from "physical location"
  - Virtual addresses: generated by the program
  - Memory locations: determined by the hardware and OS
- **Key idea #2**: caching
  - Swap virtual pages between main memory and the disk

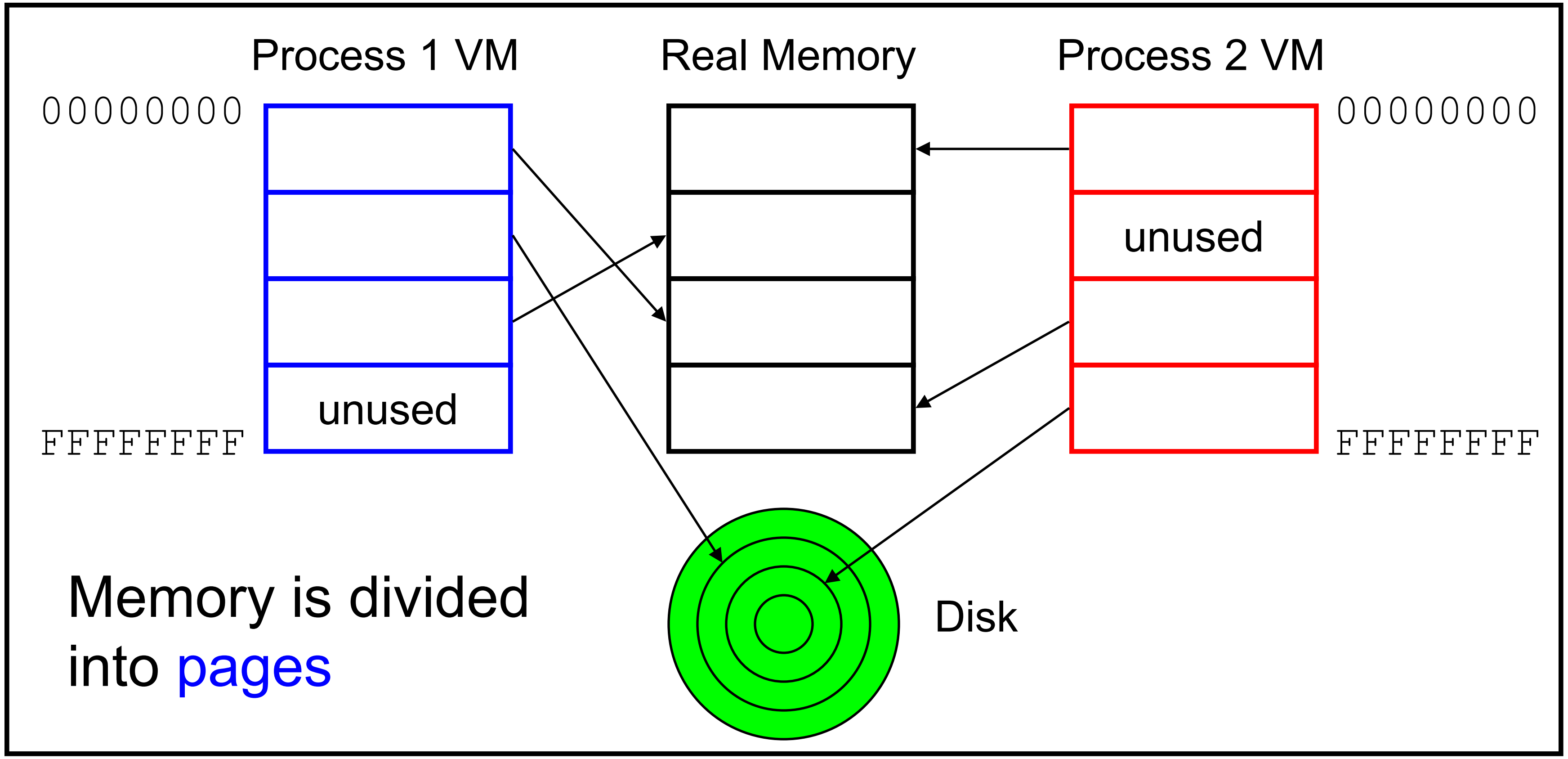
One of the best ideas in computer systems!

Private Address Space: Illusion



Hardware and OS give each application process the illusion that it is the only process using memory

Private Address Space: Reality



All processes use the same real memory
Hardware and OS provide application programs with a virtual view of memory, i.e. virtual memory (VM)

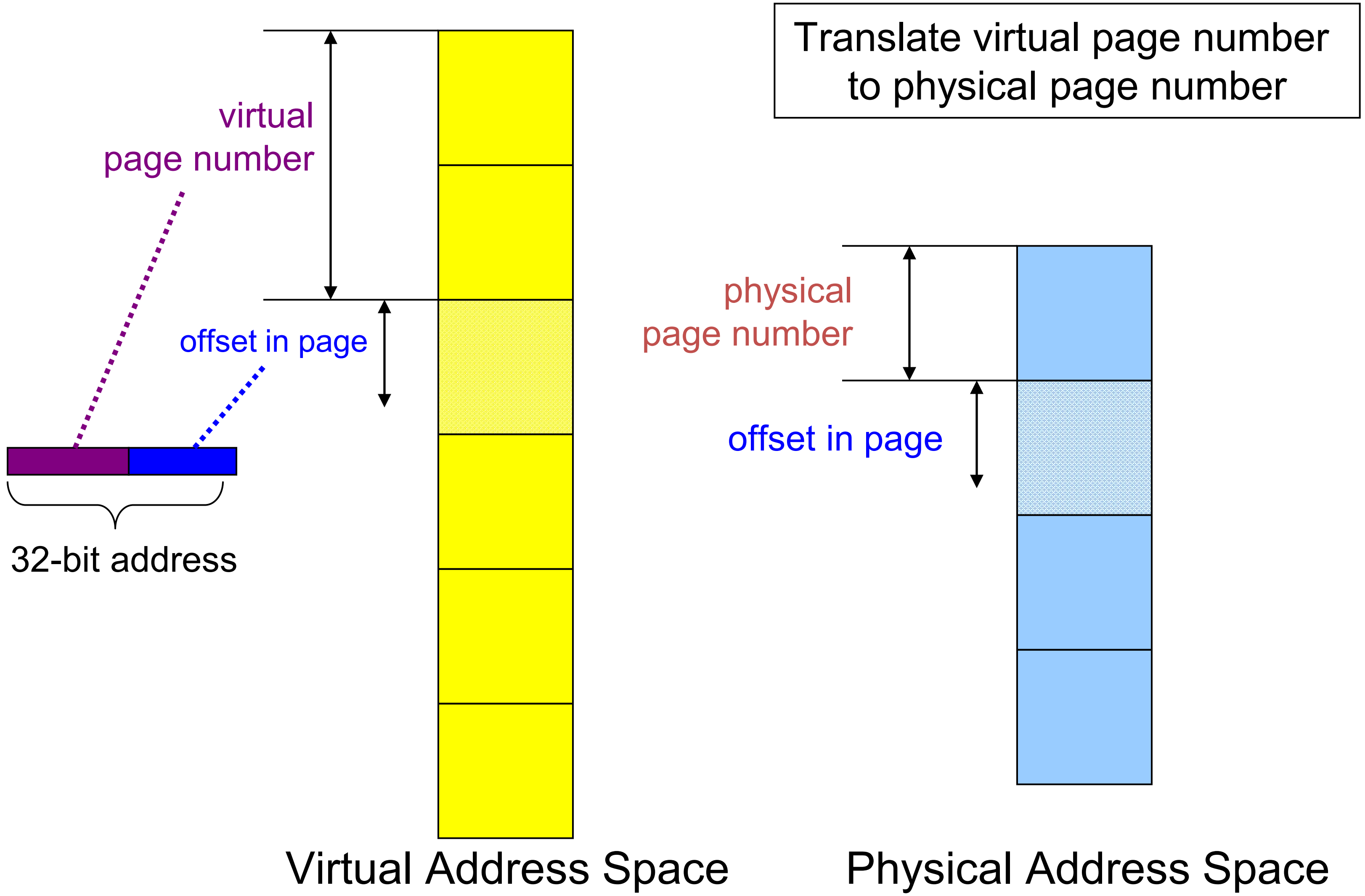
Making Good Use of Memory and Disk

- Good use of the disk
  - Read and write data in large “pages”
  - … to amortize the cost of “seeking” on the disk
  - e.g., page size of 4 KB
- Good use of main memory
  - Although the address space is large
  - … programs usually access only small portions at a time
  - Keep the “working set” in main memory
    - Demand paging: only bring in a page when needed
    - Page replacement: selecting good page to swap out
- Goal: avoid thrashing
  - Continually swapping between memory and disk

Virtual Address for a Process

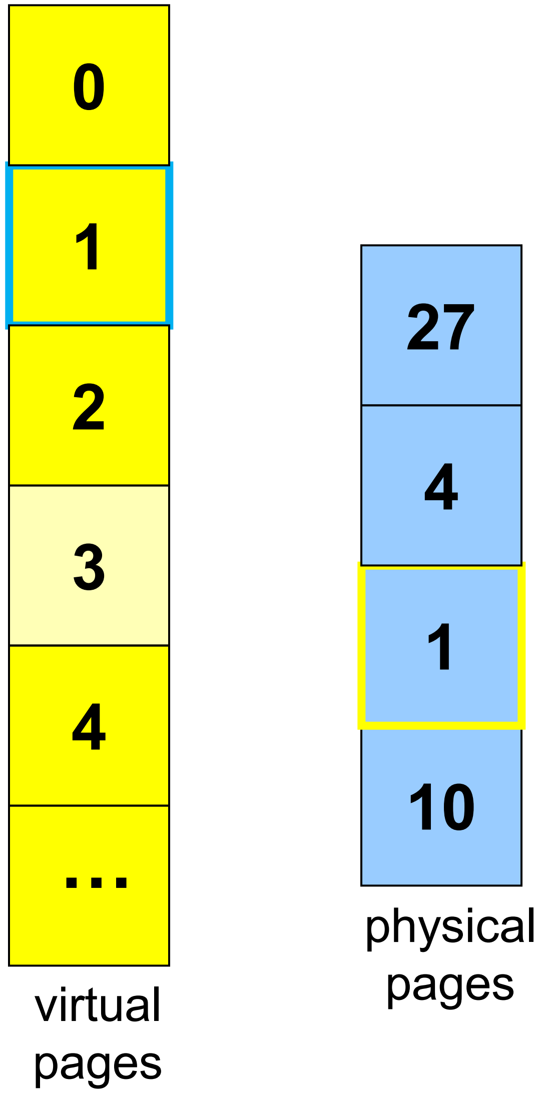
- Virtual page number
  - Number of the page in the virtual address space
  - Extracted from the upper bits of the (virtual) address
  - … and then mapped to a physical page number
- Offset in a page
  - Number of the byte within the page
  - Extracted from the lower bits of the (virtual) address
  - … and then used as offset from start of physical page
- Example: 4 KB pages
  - 20-bit page number: 2^{20} virtual pages
  - 12-bit offset: bytes 0 to 2^{12}-1

Virtual Address for a Process



Page Table to Manage the Cache

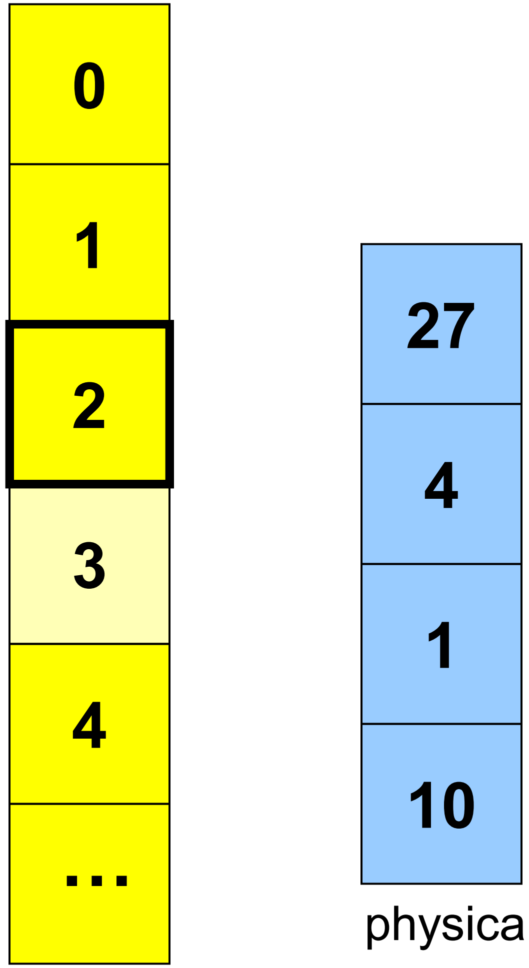
- Current location of each virtual page
  - Physical page number, or
  - Disk address (or null if unallocated)
- Example
  - Page 0: at location xx on disk
  - Page 1: at physical page 2
  - Page 3: not yet allocated
- Page "hit" handled by hardware
  - Compute the physical address
    - Map virtual page # to physical page #
    - Concatenate with offset in page
  - Read or write from main memory
    - Using the physical address
- Page "miss" triggers an exception…



“Miss” Triggers Page Fault

- Accessing the page not in main memory

	V	Physical or disk address
0	0	xx
1	1	2
2	**0**	**yy**
3	0	null
4	1	1
		…



virtual pages

physical pages

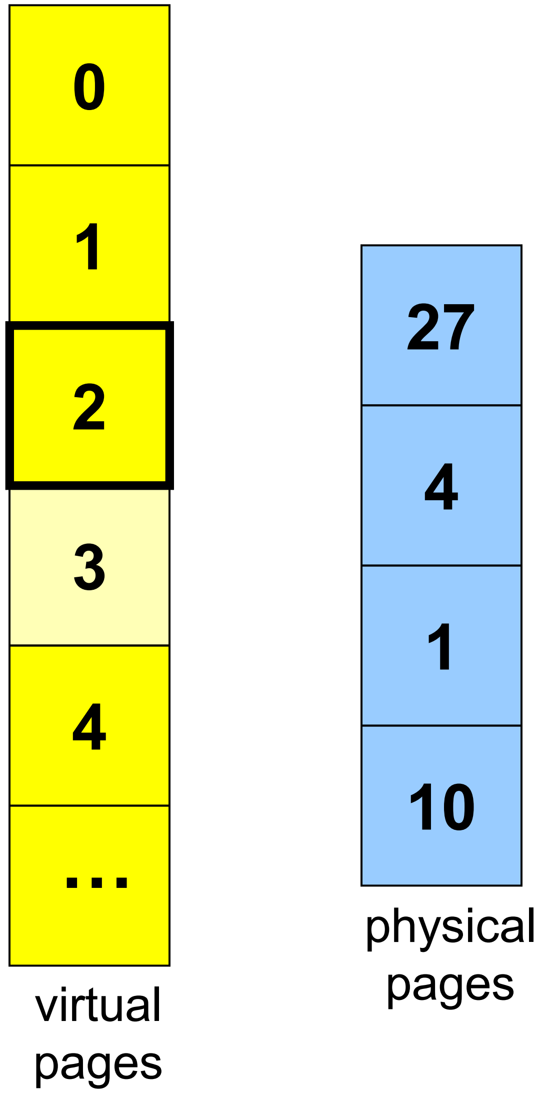
"Miss" Triggers Page Fault

- Accessing the page not in main memory

	V	Physical or disk address
0	0	xx
1	1	2
2	**0**	**yy**
3	0	null
4	1	1
		…

```
movl 00002104, %eax
```

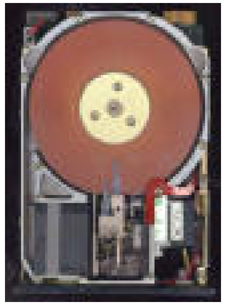
Virtual page #2 at location yy on disk!

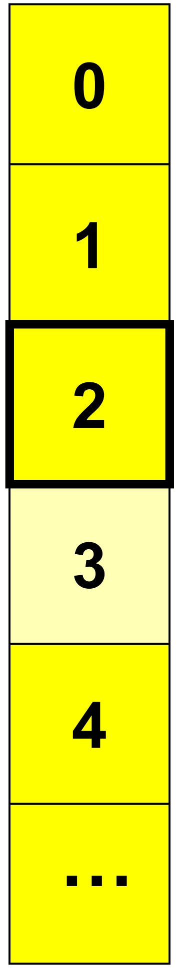


OS Handles the Page Fault

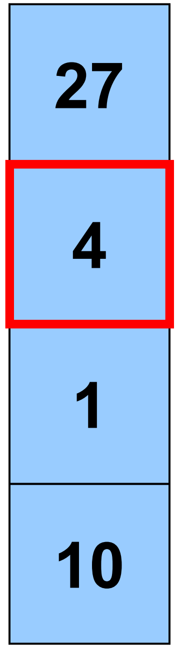
- Bringing page in from the disk
  - If needed, swap out old page (e.g., #4)
  - Bring in the new page (page #2)
  - Update the page table entries

	V	Physical or disk address
0	0	xx
1	1	2
2	**0**	**yy**
3	0	null
4	1	1
		…





virtual pages

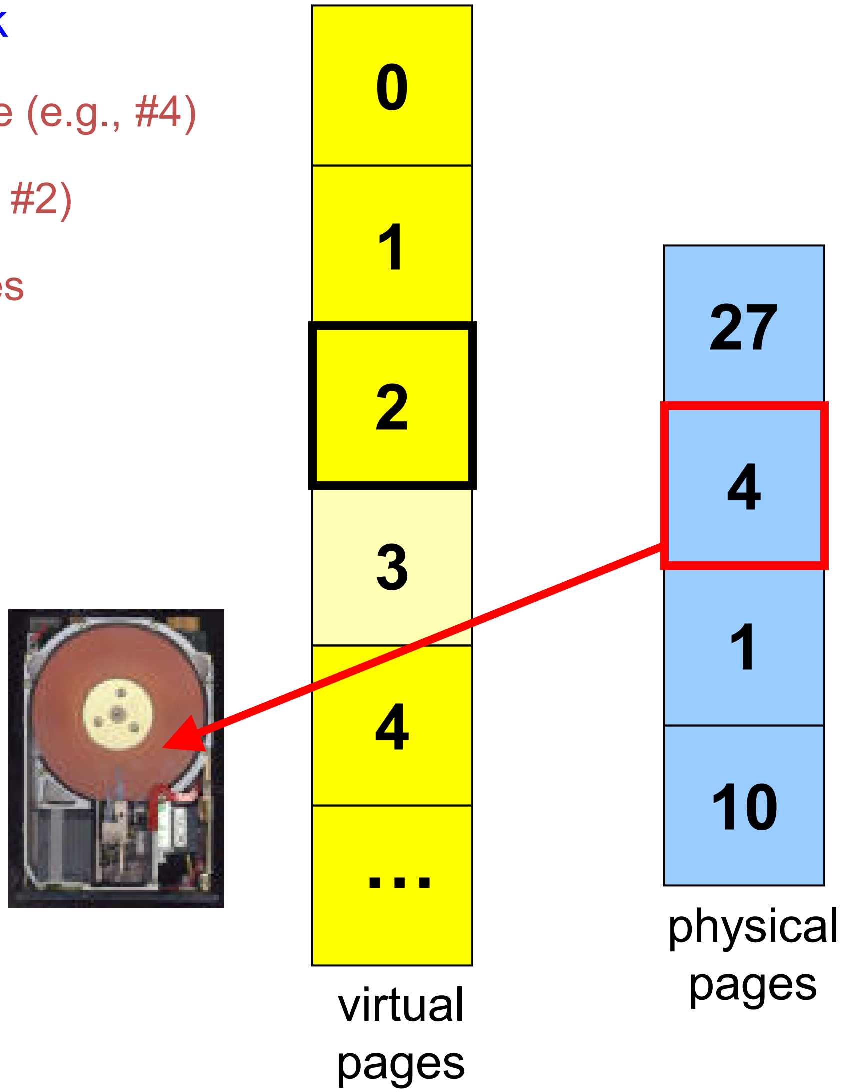


physical pages

OS Handles the Page Fault

- Bringing page in from the disk
  - If needed, swap out old page (e.g., #4)
  - Bring in the new page (page #2)
  - Update the page table entries

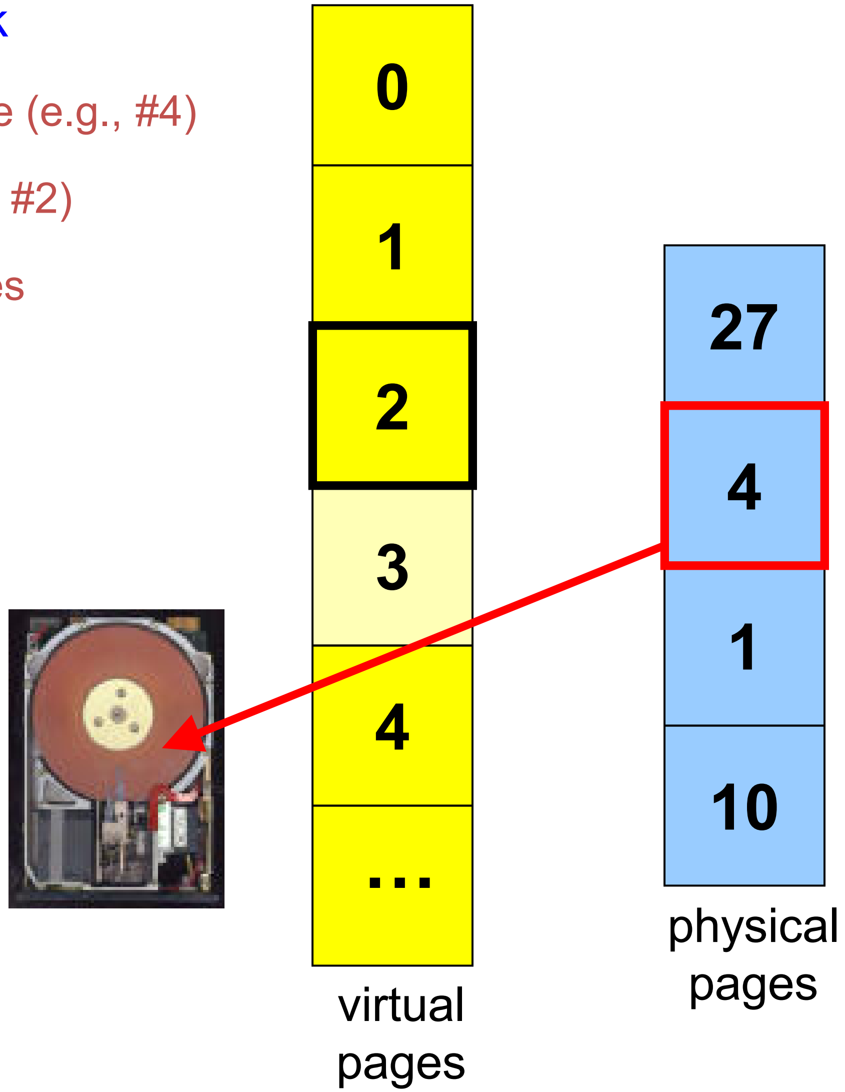
	V	Physical or disk address
0	0	xx
1	1	2
2	**0**	**yy**
3	0	null
4	1	1
		…



OS Handles the Page Fault

- Bringing page in from the disk
  - If needed, swap out old page (e.g., #4)
  - Bring in the new page (page #2)
  - Update the page table entries

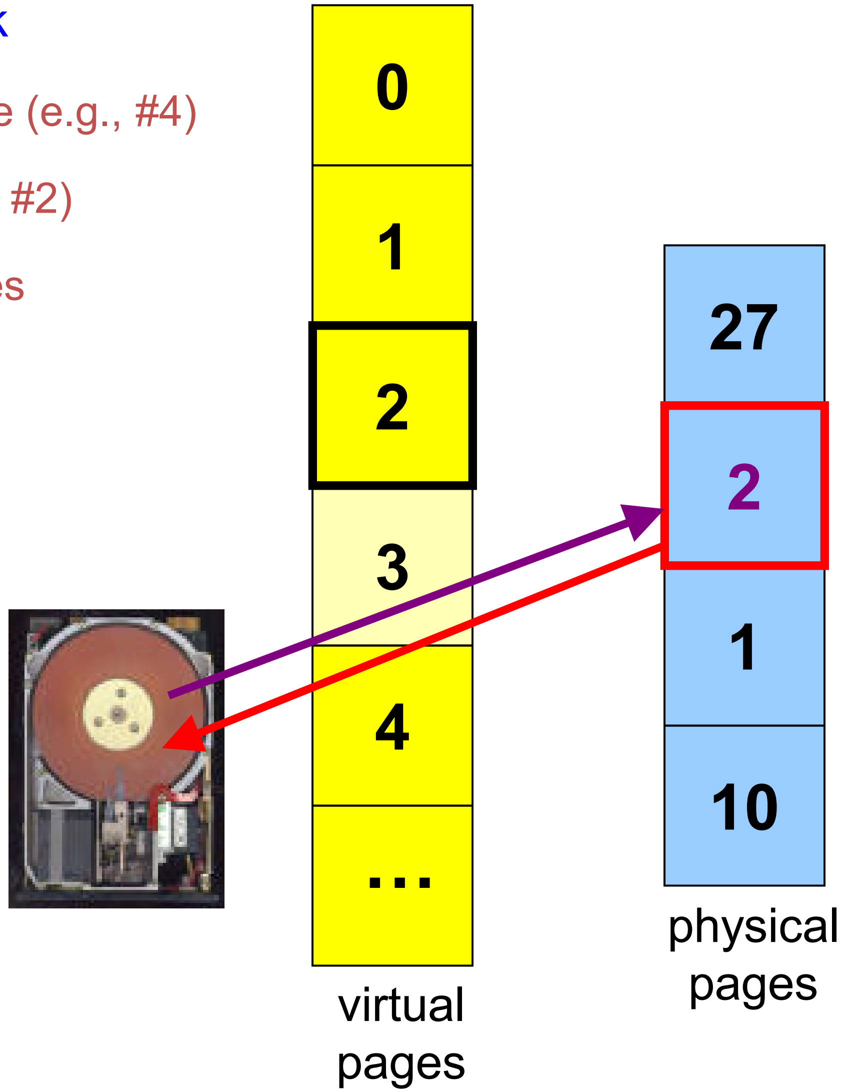
	V	Physical or disk address
0	0	xx
1	1	2
2	**0**	**yy**
3	0	null
4	~~1~~ 0	~~1~~ zz
		…



OS Handles the Page Fault

- Bringing page in from the disk
  - If needed, swap out old page (e.g., #4)
  - Bring in the new page (page #2)
  - Update the page table entries

	V	Physical or disk address
0	0	xx
1	1	2
2	~~0~~ 1	~~yy~~ 1
3	0	null
4	~~1~~ 0	~~1~~ zz
		…



VM as a Tool for Memory Protection

- Memory protection
  - Prevent processes from unauthorized reading or writing of memory

- User process should not be able to
  - Modify the read-only text section in its own address space
  - Read or write operating-system code and data structures
  - Read or write the private memory of other processes

- Hardware support
  - Permission bits in page-table entries (e.g., read-only)
  - Separate identifier for each process (i.e., process-ID)
  - Switching between *unprivileged* mode (for user processes) and *privileged* mode (for the operating system)

EE209 #19:
I/O Management

KAIST EE

Example: Opening a File

- `FILE *fopen("myfile.txt", "r")`
  - Opens the named file and return a stream
  - Includes a mode, such as “r” for read or “w” for write
- Creates a FILE data structure for the file
  - Mode, status, buffer, …
  - Assigns fields and returns a pointer
- Opens or creates the file, based on the mode
  - Write (‘w’): create the file with default permissions
  - Read (‘r’): open the file as read-only
  - Append (‘a’): open or create file, and seek to the end

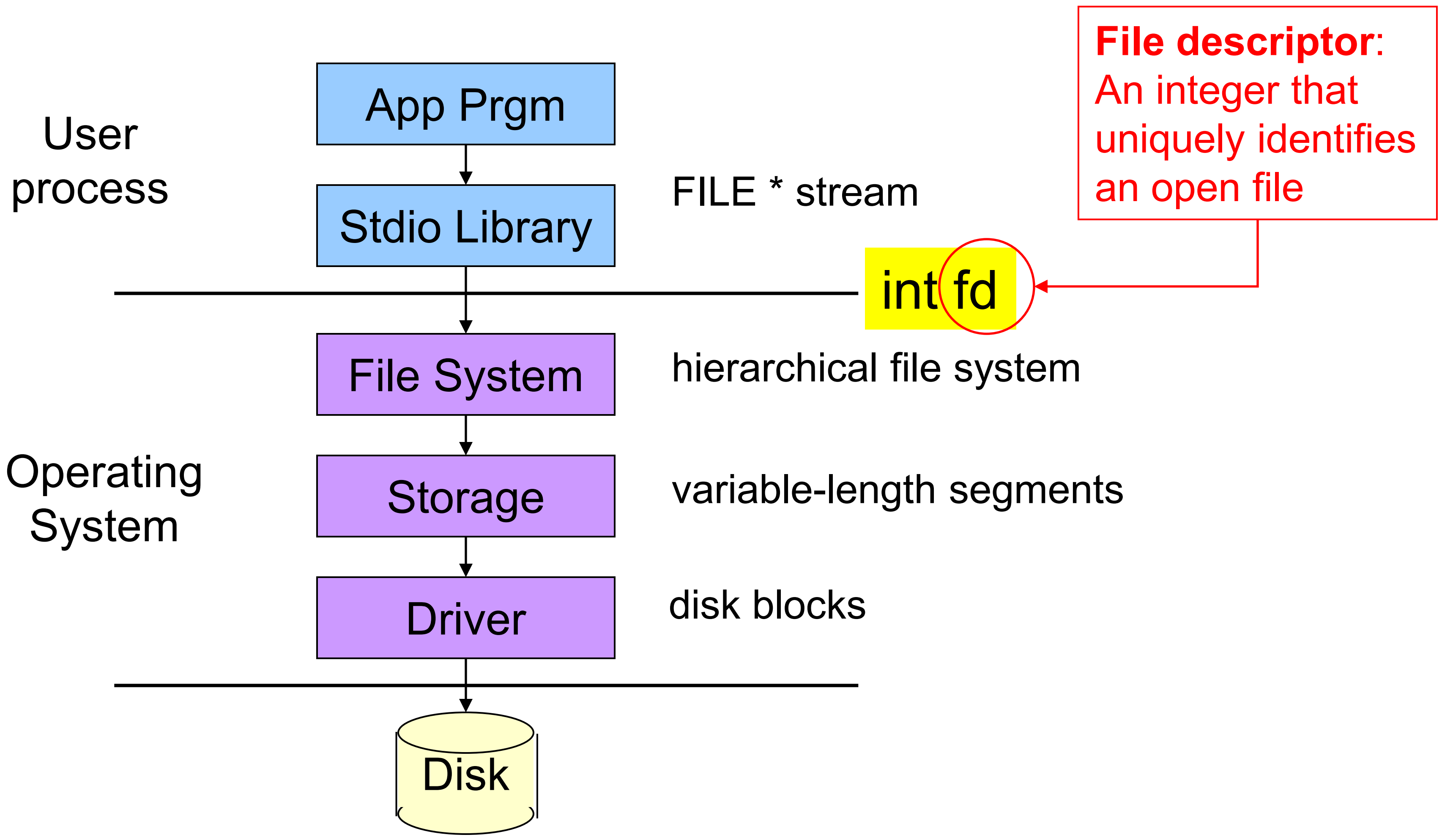
Example: Formatted I/O

- `int fprintf(fp1, "Number: %d\n", i)`
  - Convert and write output to stream in specified format
- `int fscanf(fp1, "FooBar: %d", &i)`
  - Read from stream in format and assign converted values

- Specialized versions
  - `printf(…)` is just `fprintf(stdout, …)`
  - `scanf(…)` is just `fscanf(stdin, …)`

  - `<stdio.h> has a variable FILE* stdin;`

Layers of Abstraction



System-Level Functions for I/O

```
int creat(char *pathname, mode_t mode);
```

- Creates a new file named `pathname`, and returns a file descriptor

```
int open(char *pathname, int flags, mode_t mode);
```

- Opens the file `pathname` and returns a file descriptor

```
int close(int fd);
```

- Closes `fd`

```
int read(int fd, void *buf, int count);
```

- Reads up to `count` bytes from `fd` into the buffer at `buf`

```
int write(int fd, void *buf, int count);
```

- Writes up to `count` bytes into `fd` from the buffer at `buf`

```
int lseek(int fd, int offset, int whence);
```

- Assigns the file pointer of `fd` to a new value by applying an `offset`

Example: `open()`

- Converts a path name into a file descriptor
  - `int open(const char *pathname, int flags, mode_t mode);`
- Arguments
  - `pathname`: name of the file
  - `flags`: bit flags for `O_RDONLY, O_WRONLY, O_RDWR`
  - `mode`: permissions to set if file must be created
- Returns
  - File descriptor (or -1 if error)
- Performs a variety of checks
  - e.g., whether the process is entitled to access the file
- Underlies `fopen()`

Example: `read()`

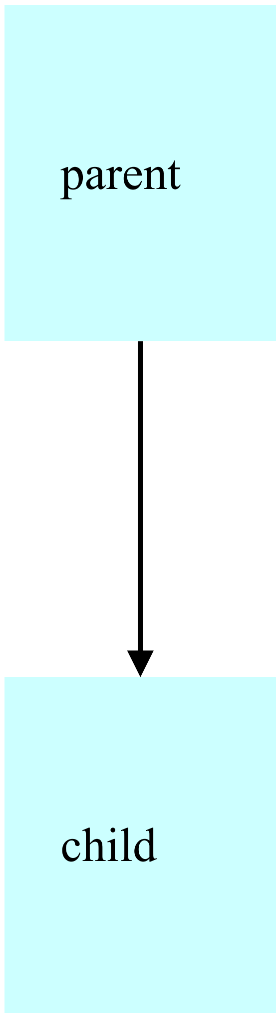
- Reads bytes from a file descriptor
  - `int read(int fd, void *buf, int count);`
- Arguments
  - File descriptor: integer descriptor returned by `open()`
  - Buffer: pointer to memory to store the bytes it reads
  - Count: maximum number of bytes to read
- Returns
  - Number of bytes read
    - Value of 0 if nothing more to read
    - Value of -1 if an error
- Performs a variety of checks
  - Whether file has been opened, whether reading is okay
- Underlies `getchar()`, `fgets()`, `scanf()`, etc.

EE209 #20: Process Management

KAIST EE

Creating a New Process

- Cloning an existing process
  - Parent process creates a new child process
  - The two processes then run concurrently
- Child process inherits state from parent
  - Identical (but separate) copy of virtual address space
  - Copy of the parent's open file descriptors
  - Parent and child share access to open files
- Child then runs independently
  - Executing independently, including invoking a new program
  - Reading and writing its own address space



Fork System-Level Function

- `fork()` is called once
  - but returns twice, once in each process
  - because a new process is created, as a result of fork()
  - 1+1 = 2

- Telling which process is which
  - Parent: `fork()` returns the child's process ID
  - Child: `fork()` returns 0

```
pid = fork();
if (pid != 0) {
   /* in parent */
  …
} else {
   /* in child */
    …
}
```

Executing a New Program

- `fork()` copies the state of the parent process
  - Child continues running the parent program
  - … with a copy of the process memory and registers
- Need a way to invoke a new program
  - In the context of the newly-created child process
- Example

program

NULL-terminated array
Contains command-line arguments
(to become "argv[]" of ls)

```
execvp("ls", argv);
fprintf(stderr, "exec failed\n");
exit(EXIT_FAILURE);
```

Waiting for the Child to Finish

- Parent should wait for children to finish
  - Example: a shell waiting for operations to complete
- Waiting for a child to terminate: `wait()`
  - Blocks until some child terminates
  - Returns the process ID of the child process
  - Or returns -1 if no children exist (i.e., already exited)
- Waiting for specific child to terminate: `waitpid()`
  - Blocks till a child with particular process ID terminates

```
#include <sys/types.h>
#include <sys/wait.h>
pid_t wait(int *status);
pid_t waitpid(pid_t pid, int *status, int options);
```