

# **Operating Systems (234123)**

## ***Introduction***

Dan Tsafrir (2025-04-06, 2025-04-07)

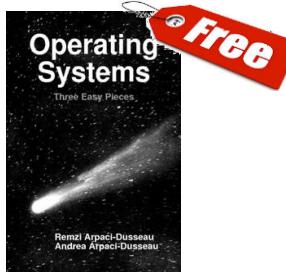
# **Admin – staff**

- **Lecturers**
  - 1) Prof. Dan Tsafrir (in charge)
  - 2) Dr. Leonid Raskin
- **TAs**
  - 1) Daniel Bransky (in charge administratively)
  - 2) Malik Khalaf
  - 3) Ahmad Agbaria
  - 4) Alex Zhybirov
  - 5) Shachar Asher Cohen
  - 6) Omer Daube
  - 7) Tsvika Lazar

# Admin – course material & grading

- **Course website**
  - <http://webcourse.cs.technion.ac.il/234123>
  - Contains all course materials, e.g., office hours times & locations
- **Lecture material**
  - Sometimes updated (published before the lecture)
- **Grades**
  - Homework: 30% – 35%
    - 4 wet, 4 dry, uneven weights, נקודות
    - Weighted average must be  $\geq 55$  to pass & take the exam
  - Exam: 65% – 70%
    - Must be  $\geq 55$  to pass
- **Is it possible to use homework grades from the last time I took OS because <reasons>?**
  - No. Really. We're not doing that.

# Admin – books



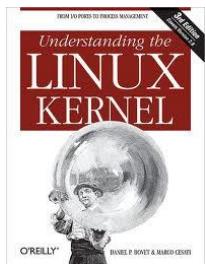
## Operating Systems: Three Easy Pieces

- By Remzi & Andea Arpaci-Dusseau
- <http://pages.cs.wisc.edu/~remzi/OSTEP> (free online)



- **Notes on Operating Systems**

- By Dror G. Feitelson
- <https://moodle.cs.huji.ac.il/cs14/file.php/67808/top.pdf>
- *This lecture: Chapters 1, Appendix A, and some of Chapter 2*

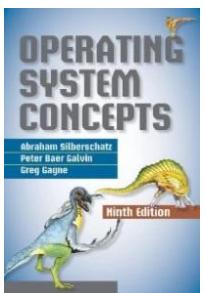


- **“Understanding the Linux Kernel”**

- By Daniel P. Bovet & Marco Cesati
- 3<sup>rd</sup> edition

- **“Operating System Concepts”**

- By A. Silberschatz, P.B. Galvin, and G. Gagne
- 10<sup>th</sup> edition (>= 6 is fine)



# Admin – programming courses revision

- Since 2019, ATAM and OS are taught in parallel, and, for various reasons
  - The ‘system calls’ and ‘interrupts’ topics have migrated to ATAM
  - But are still being treated as an integral part of the course!
- ~~If you studied the previous ATAM version~~
  - You didn’t learn these topics,  
~~and won’t learn them here (except in high level)~~
  - You should attend the relevant lectures in ATAM

A note about

# **QUESTIONS**

Having gone over the admin stuff

**LET US BEGIN**

# In an OS-less world...

- **Compiler**

- Program that transforms C code to assembly code  
(textual representation of machine code)

```
1 int exchange(int *xp, int y)
2 {
3     int x = *xp;
4
5     *xp = y;
6     return x;
7 }
```

compiler

1	movl 8(%ebp), %eax	Get xp
2	movl 12(%ebp), %edx	Get y
3	movl (%eax), %ecx	Get x at *xp
4	movl %edx, (%eax)	Store y at *xp
5	movl %ecx, %eax	Set x as return value

- **Assembler**

- Transforms assembly code to executable machine code

- **When we say “application”**

- Or “program” or “process”
- We mean a running instance of this

```
// I'15;
MOV R3, #15
STR R3, [R11, #-8]

// J'25;
MOV R3, #25
STR R3, [R11, #-12]

// I'1'J;
LDR R2, [R11, #-8]
LDR R3, [R11, #-12]
ADD R3, R2, R3
STR R3, [R11, #-8]
```

assembly

```
1100 1010 1011 0011
1100 1010 1011 0011
1100 1010 1011 0011
1100 1010 1011 0011
1100 1010 1011 0011
1100 1010 1011 0011
```

assembler

```
1100 1010 1011 0011
1100 1010 1011 0011
1100 1010 1011 0011
1100 1010 1011 0011
1100 1010 1011 0011
1100 1010 1011 0011
```

machine  
code

# Become aware of the OS



# Become aware of the OS

- **From the point of view of a process  
(or its newbie programmer)**
  - The OS is the Matrix
  - It creates the world view of processes

# Become aware of the OS

*The Matrix is everywhere.*

*It is all around us.*

*Even now in this very room.*

*You can see it when you look out the window or when you turn on your television.*

*You can feel it when you go to work [...]*

*It is the world that has been pulled over your eyes to blind you from the truth*



# Become aware of the OS

---

*What truth?*



# Become aware of the OS

*That you are a slave, Neo.*

*Like everyone else, you were born into bondage.*

*Born into a prison that you cannot small or taste or touch.*

*A prison for your mind.*



# Become aware of the OS

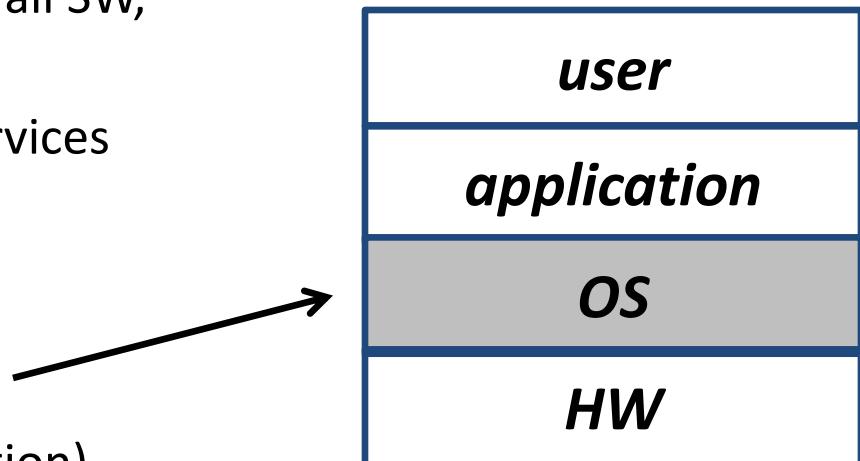
---

- Understanding the OS means taking the red pill
- Answering the question:  
  
How do computer systems *really* work?
- (Despite the course's name, it includes lots of hardware material)

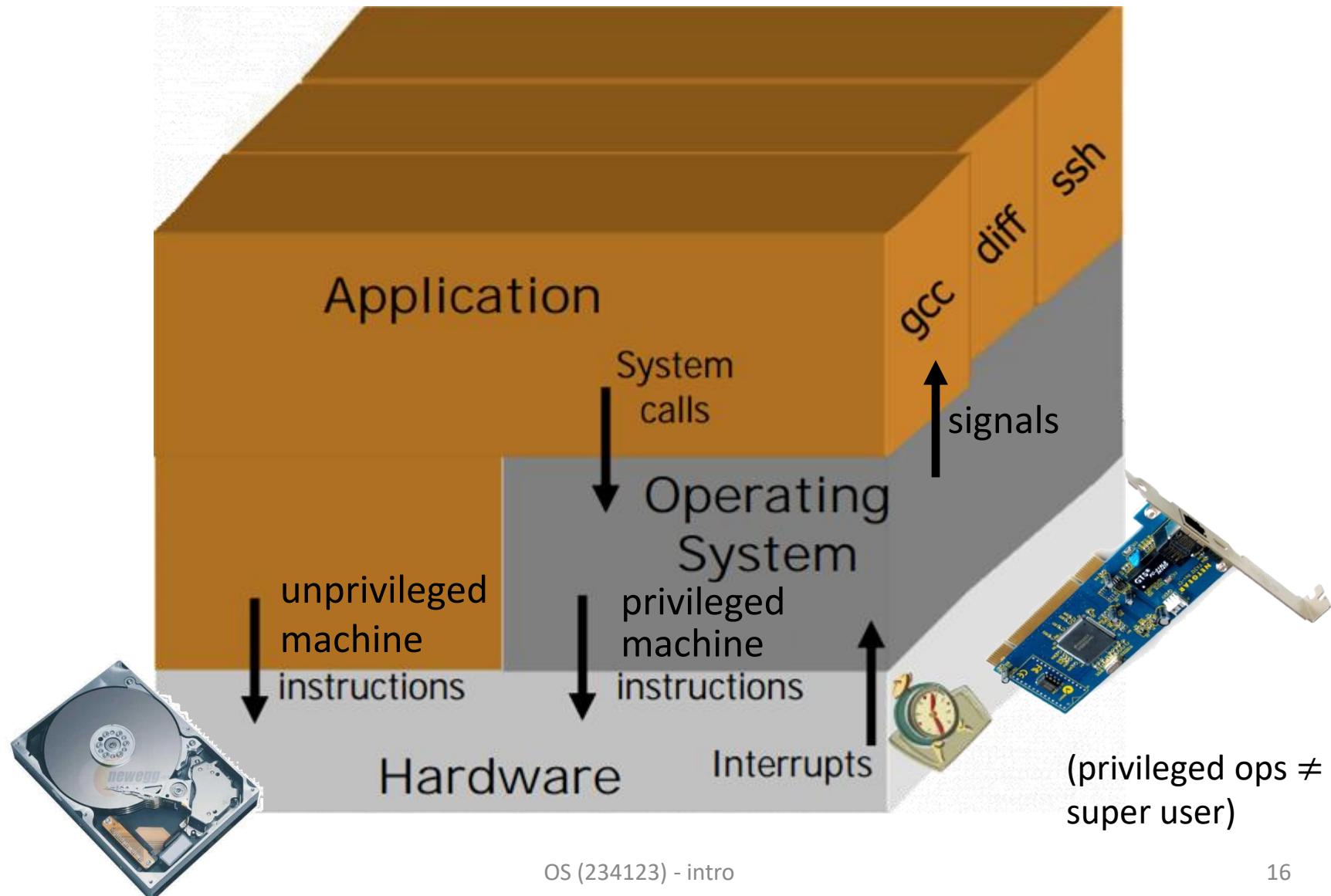


# So, what \*is\* the OS?

- A piece of software (**SW**) acting as an intermediary between user applications (**apps**) and the computer hardware (**HW**)
- First piece of SW to run on a computer when it's booted
- Its job
  - Coordinate the execution of all SW, mainly user apps
  - Provide various common services needed by users & apps
- Often drawn like so
  - That's a lie (= oversimplification)



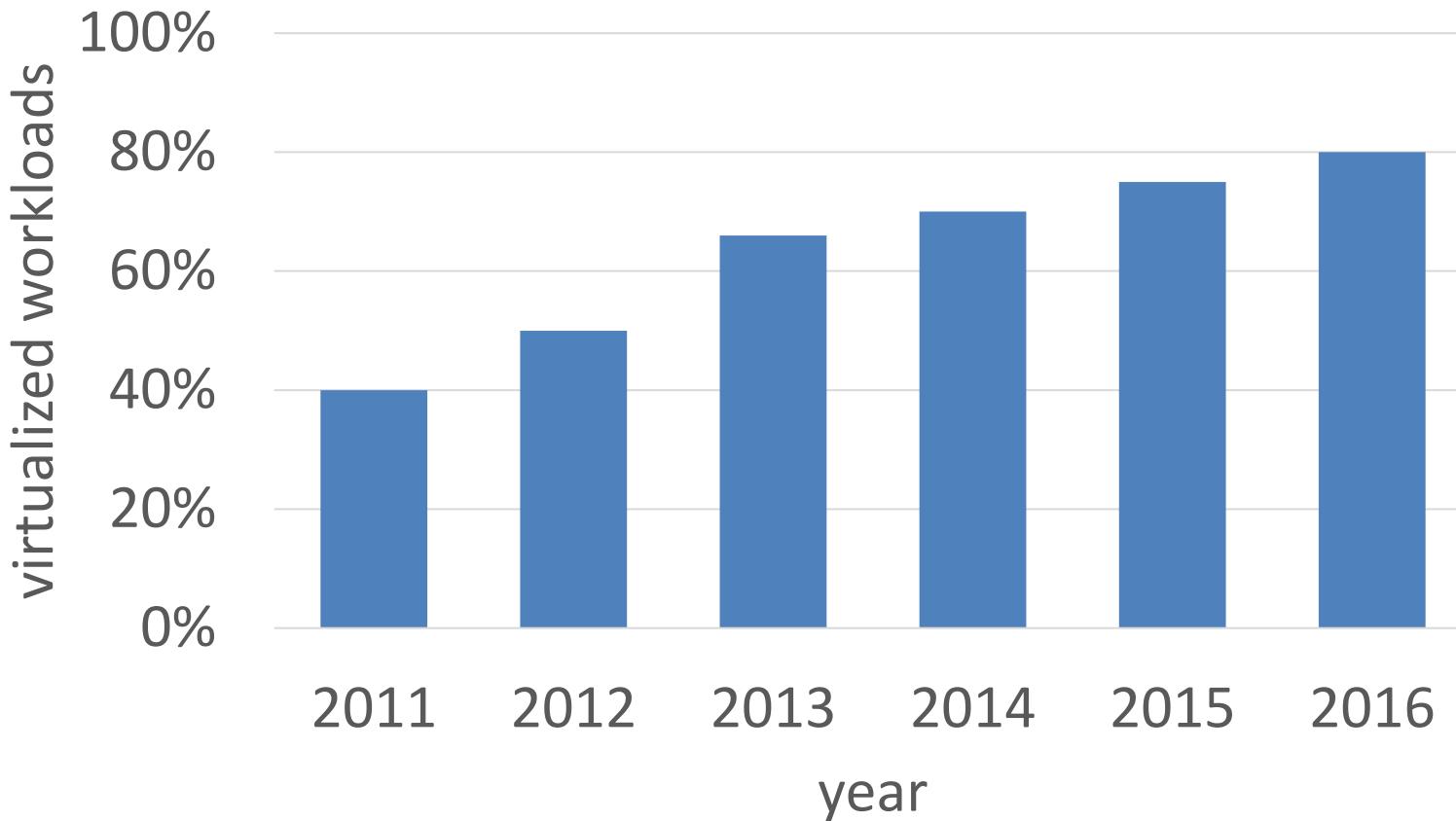
# It's a bit more complex



# A bit more complex due to “virtualization”

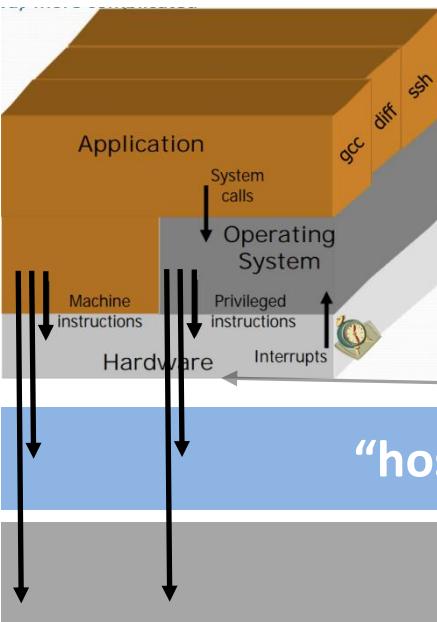
- **An OS of a physical server controls its physical devices**
  - CPU, memory, disks, etc.
- **An OS of a virtual server only *believes* it does**
  - There's another OS underneath, called “hypervisor,” which fakes it
- **Benefits of virtualization**
  - “Server consolidation” (multiple virtual servers on one physical server)
    - Better utilization of physical machines
    - Smaller space
    - Less energy consumption for powering and cooling servers
  - Disentangling SW from HW allows for
    - Backup/restore, live migration, HW upgrade
    - Easier provisioning of new (virtual) server = “virtual machines”
    - Easier OS-level development and testing

**“About 80% of x86 server workloads are virtualized”** [Gartner report; from Aug 2016]

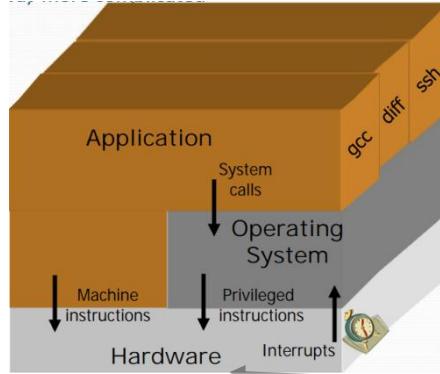


# So it's actually more complex

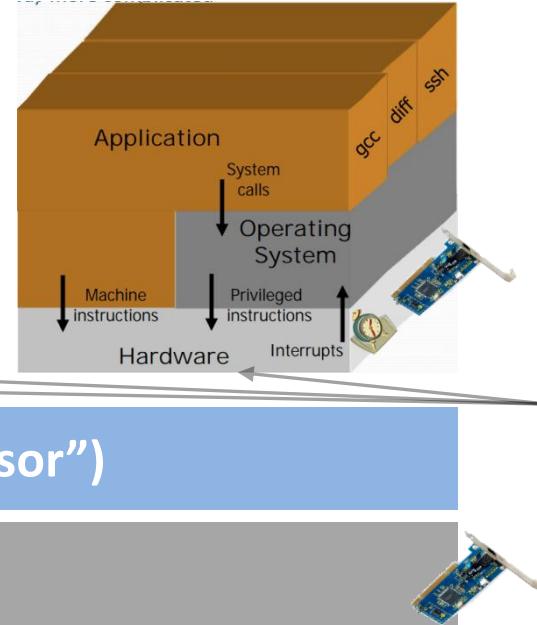
Windows 10 “guest”  
virtual machine (VM)



Linux  
VM



Another  
VM



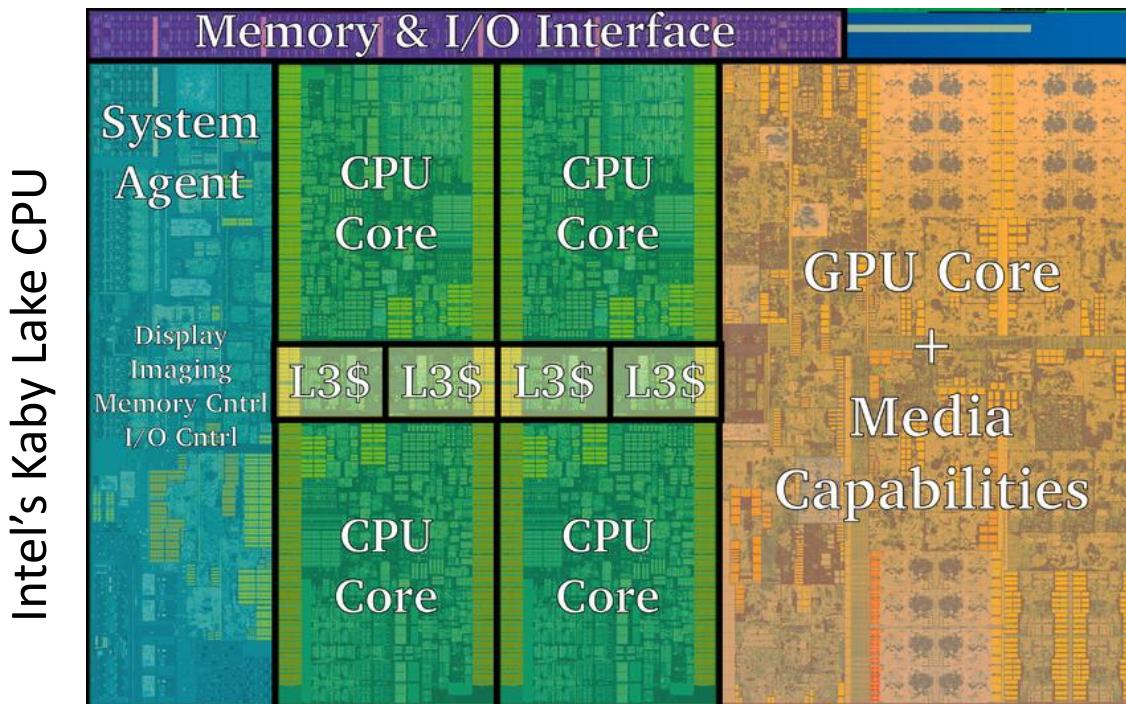
“host” OS (called “hypervisor”)

physical HW

- The hypervisor (which is also an “OS”) can be:  
Linux/KVM, Xen, Microsoft’s HyperV, VMware ESXi or Workstation, ...
- VMs access physical HW similarly to apps, when doing non-privileged ops
- Topic is called: “virtualization”, a concept that is directly supported by HW
- See also WSL ([https://en.wikipedia.org/wiki/Windows\\_Subsystem\\_for\\_Linux](https://en.wikipedia.org/wiki/Windows_Subsystem_for_Linux))

# For simplicity

- In *this* lecture, unless stated otherwise, we'll focus on a simple OS setup
  1. Non-virtualized OS – also called “bare metal” OS
  2. One ‘processor core’ = ‘CPU core’ (the term “CPU” alone is ambiguous: could mean one core in the CPU or the entire multicore)



# For simplicity

- In *this* lecture, unless stated otherwise, we'll focus on a simple OS setup
  1. Non-virtualized OS – also called “bare metal” OS
  2. One ‘processor core’ = ‘CPU core’ (the term “CPU” alone is ambiguous: could mean one core in the CPU or the entire multicore)
- Both of the above
  - Becoming less common in today's world
  - But simplifying is helpful
- So, under the above assumptions...

# The OS is reactive

	typical programs (= app = process) we've seen thus far	OS
what does it typically do?	get some input, do some processing, produce output, terminate	waits & reacts to “events”
structure	has a <code>main</code> function, which is (more or less) the entry point	no <code>main</code> ; multiple entry points, one per event
termination	end of <code>main</code>	power shutdown
typical goal	~ finish as soon as possible	handle events as quickly as possible => more time for apps to run

OS events can be classified into two

- Asynchronous interrupts – keyboard, clock, disk drive, network card, ...
- Synchronous – systems calls (apps request service), divide by 0, ...

# Example: context switch & system call

CPU-bound app#1 is replaced by app#2, which invokes a “blocking” system call:

1. OS executes – scheduling app#1 to run
2. App#1 runs – core executes its non-privileged ops (OS remains uninvolved)
3. HW clock interrupt fires – invokes OS’s “interrupt handler” routine
4. Handler updates OS time & calls scheduler if  
app#1 has been running for too long (we say: exhausted its “quantum”)
5. Scheduler chooses app#2 – to run instead of app#1 = “context switch”
6. App#2 runs & performs a “system call” – e.g., to read from a file
7. OS regains control – initiates I/O op (privileged) & puts app#2 to “sleep”  
(wait in non-runnable state until I/O op completes)
8. OS scheduler selects app#1 to run...

**Note:** either OS or app run on core, not both; OS regains control upon interrupts

# The OS performs resource management

- Previous slide describes the “**multiprogramming**” OS feature
  - Multiple apps run concurrently (seemingly simultaneously) on one core
- **Multiprogramming is achieved through resource management**
  - CPU cores (or cycles) constitute resources that are managed by the OS
- **Resources**
  - Obvious ones are physical
    - CPU, memory, disk, network, keyboard, mouse, screen display, ...
  - But many are virtual (internal OS structures)
    - System tables
      - Processes, open files, ...
    - “Address” spaces
      - Virtual memory, network ports, process IDs, file IDs, ...

# The OS performs resource management

- **A primary goal – resource multiplexing ("ריבוב")**
  - Method used by networks to consolidate multiple signals – digital or analog – into a single composite signal that is transported over a common medium
- **In OS context, multiplexing means**
  - Make it look as if each app has its own dedicated resources
  - OS “juggle” resources between apps
  - Decides which app gets which resource and when
  - In so doing, the OS “virtualizes” (an overloaded term) the resources
- **Multiprogramming vs. multiplexing**
  - Multiprogramming means multiplexing the CPU resource

# The OS provides services – #1

End of 1<sup>st</sup>  
lecture

- The OS can be viewed as the collection of services it provides, for example
  - The ability to execute on the CPU
  - The ability to communicate with local & remote apps via shared memory & networking, respectively
  - The ability to draw on screen
  - The ability to store data in files persistently
  - The ability to control running apps (suspend, resume, terminate)
  - ...

# The OS provides services – #2

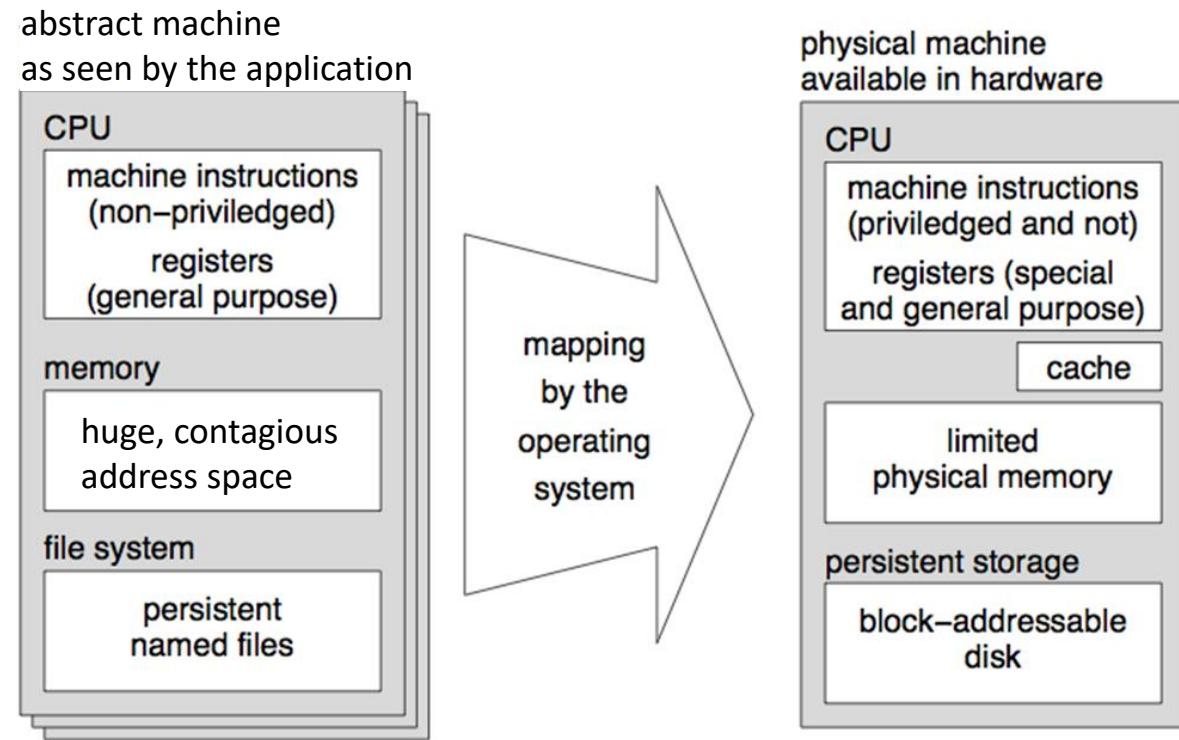
- **Each service involves two aspects**
  1. *Isolation* – allow multiple apps to coexist using the same resources without stepping on each other's toes
    - E.g., several apps send/receive data via one NIC (network controller)
      - The OS maintains separate data streams for each app
    - Multiplexing is typically how the OS isolates apps from each other
    - But some physical resources know how to multiplex themselves
      - E.g., one NIC can provide multiple NIC instances
      - OS can hand one instance for each app and get out of the way
      - Sometimes called “**self-virtualizing devices**”
  2. *Abstraction* – provides convenience & portability by:
    - (i) offering more meaningful, higher-level interfaces, and by
    - (ii) hiding HW details, making interaction with HW easier, e.g.,
    - Apps use “**files**” instead of handing numerous raw block devices

# The OS provides services – #3

- **Multiplexing is complex**
  - Hard for users/apps to track what's being done at any given time
- **So OS presents an abstraction that hides all this complexity**
  - Each app sees/gets its own “**abstract machine**”
  - There are **multiple abstract machines**, isolated from each other
  - Apps are unaware of multiplexing activity
    - The abstraction hides it
  - So each app sees the system as if it's **dedicated to the app**, and it's the **job of the OS to juggle resources** to support this illusion
- **Notably, the per-app computation is still *correct***
  - Despite the multiplexing that occurs behind the scenes

# The abstract machine is “better”

- ...Better than the physical HW, as it allows the OS to
  - Hide physical restrictions, and
  - Support more convenient interfaces
- Apps often don't access physical resources directly
  - There's a **level of indirection**
- Making it possible
  - To, e.g., support, say, a 32GB of (virtual) memory with only, say, 8GB of (physical) memory



# The abstract machine is “better”

- ...**Better than the physical HW, as it provides “portability”**
  - Ability to correctly run the same exact program in different environments
    - In terms of source code, not necessarily binary code
  - No need to rewrite user apps when changing HW or OS, namely...
- **Portability across different HW devices**
  - CPUs: by Intel (x86-32bit/64bit, Itanium), AMD (x86-32/64), IBM (PowerPC, z/Architecture(=mainframe)), Oracle (SPARC), Apple (ARM), ...
    - Linux, for example, supports (runs on) dozens of CPU architectures:  
[https://en.wikipedia.org/wiki/List\\_of\\_Linux-supported\\_computer\\_architectures](https://en.wikipedia.org/wiki/List_of_Linux-supported_computer_architectures)
  - NICs: by Intel, Mellanox, Chelsio, Emulex, HotLava, SolarFlare, ...
  - Many other devices/vendors: disks, mice, keyboards, displays, GPUs, ...
- **Portability across different OSes**
  - Lots of UNIX variants: Linux, HPUX (HP), Solaris (Oracle), AIX (IBM), Darwin (Apple), FreeBSD (used by Netflix), NetBSD, OpenBSD, ...
  - All implement **POSIX** = “Portable Operating System Interface”
    - A standard that defines, e.g., system calls
      - Name, arguments, return value, and behavior

# At least 3 different ways to view an OS

## 1. By its programming interface

- Its system calls

## 2. According to the services it provides

- Time slicing (how multiprogramming is implemented as explained earlier), filesystems and their operations, etc.

## 3. According to its internals, algorithms, & data structures

- Arguably, an OS is *defined* by its interface

- Different implementation of the same interface are equivalent as far as users and apps are concerned

- But the lectures are organized according to services

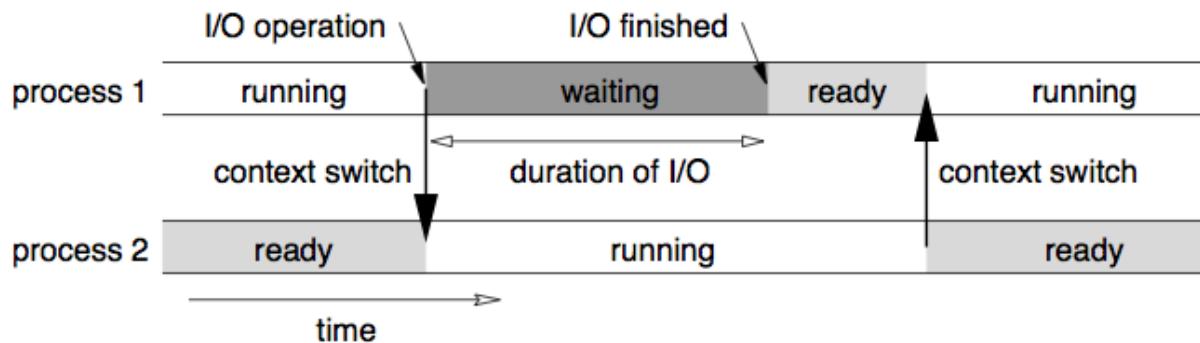
- For each, we'll usually detail the internal structures & algorithms used
  - Occasionally, we'll provide examples of interfaces, mainly for Unix (POSIX) OSes, focusing on Linux

# Subsystems we'll study

- **OS components can be studied in isolation; we'll focus on**
  - Process handling
    - Processes are the agents of processing; the OS creates them, schedules them, and coordinates their interactions
  - Inter-process communication (“IPC”)
    - Ways for different processes to communicate with each other
  - Memory management
    - Memory is allocated to processes as needed, but there might not be enough for all, so paging is used
  - File system
    - Files are an abstraction providing named data persistent repositories
    - On top of disks that store individual blocks
    - The OS does the bookkeeping

# Interaction between subsystems

- But OS components often interact, e.g.,
  - OS improves utilization by, for example, overlapping computation of one process with the read ops of another (see context switch example above)



- Likewise, if a process requires a memory page that currently resides on disk (due to paging), it suffers a page fault, which results in an I/O op; so another process is executed in the meantime

# Scope – focusing on the kernel

- All things we've mentioned thus far relate to the OS “kernel”
  - *“The central part of an OS; it manages the tasks of the computer and the hardware - most notably memory and CPU time.”* [[Wikipedia](#)]
  - The kernel will indeed be our focus
- But when one talks about an OS, one typically refers to a “distribution”, which contains the following elements
  - The Unix kernel itself
  - The libc library; provides runtime environment for programs written in C (e.g., contains the implementation of printf, strcpy, malloc, etc.)
  - Various tools, such as gcc, the GNU C compiler, package manager
  - Many useful utilities/programs you may need, e.g., windowing system, desktop, and shell
- With some exceptions, we'll typically focus on the kernel
  - What it is supposed to do, and how it does it

# **HW SUPPORT FOR THE OS**

# Privileged operations & privileged mode

- **Sensitive/privileged ops can only be used by the OS**
  - Blocking/unblocking interrupts (Why would we want to do that? Why is this privileged? These questions also apply to...)
  - All operations related to I/O devices
  - All operations related to memory indirection layers, mem usage bits
- **HW supports at least 2 privilege modes**
  - Kernel mode (ring 0 on x86)
    - Kernel runs in this mode
  - User mode (ring 3 on x86)
    - User apps run in this mode
  - Privileged processor operations work only in kernel mode
- **Host vs. guest modes**
  - For virtualization

# Privileged (admin / root) processes

- **Not allowed to use privileged instructions**
  - Only the operating system does
  - A root/admin process is still a user process
- **It's the OS that agrees to do more for these processes**
  - Providing them with more permissions than regular processes

# Some HW mechanisms to support the OS

- **Protected/unprotected modes & privileged instructions**
  - As we just discussed
- **Interrupts**
  - Such as HW clock, network, disk, ...
- **System calls**
- **Atomic synchronization operations**
- **Memory virtualization & protection**
- **I/O**
  - **DMA** (= direct memory access), MMIO (= memory-mapped I/O) + PIO (= programmed I/O), IO MMU (= I/O memory management unit)
- ...

# Reading from disk on Linux/x86

- **User app invokes ‘read’ system call**
  - int `read` ( int fileDescriptor , char \*buffer , size\_t size )
- **libc does the following (in user mode)**
  - Assigns the number ID associated with ‘`read`’ into register ‘`eax`’
  - Prepares the other `parameters` (according to some convention)
  - **Legacy**: executes: `int 0x80` (called “trap”) ( $0x80 = 128$  in decimal), which generates a synchronous interrupt, initiated by SW  
**Nowadays**: executes the “`syscall`” assembly instructions
- **Control transfers to kernel**
  - HW changes mode from unprivileged (ring 3) to privileged (ring 0)
  - HW accesses IDT (interrupt descriptor table) array in entry 128
  - IDT contains pointer-to-functions (addresses to kernel code;  
Content of IDT is set by OS at boot time; user code can’t access it)
  - Entry 128 of IDT points to the kernel’s “system call handler” routine
  - HW invokes this handler

# Reading from disk on Linux/x86

- **Within the kernel**
  - The system call identifier ('read') is used to find & call the appropriate OS function that actually perform the desired service ('`sys_read`').
  - The latter function then starts by retrieving/validating its arguments from where they were stored by the `wrapper libc function`
  - It then `generates an I/O` read request that will be processed by the disk using `DMA` (no need to further involve the CPU)
  - If needed, the OS `suspends` the app until the I/O read is completed
  - OS `resumes` another ready-to-run app
- **I/O read is completed (a few micro- to milliseconds later)**
  - An `interrupt` is generated, invoking the matching handler from IDT
  - OS figures out that the app's `I/O is completed`
  - OS changes `app's status` from 'waiting' to 'ready to run'
  - When app is scheduled next, it will resume from after "int 0x80"

# **Operating Systems (234123)**

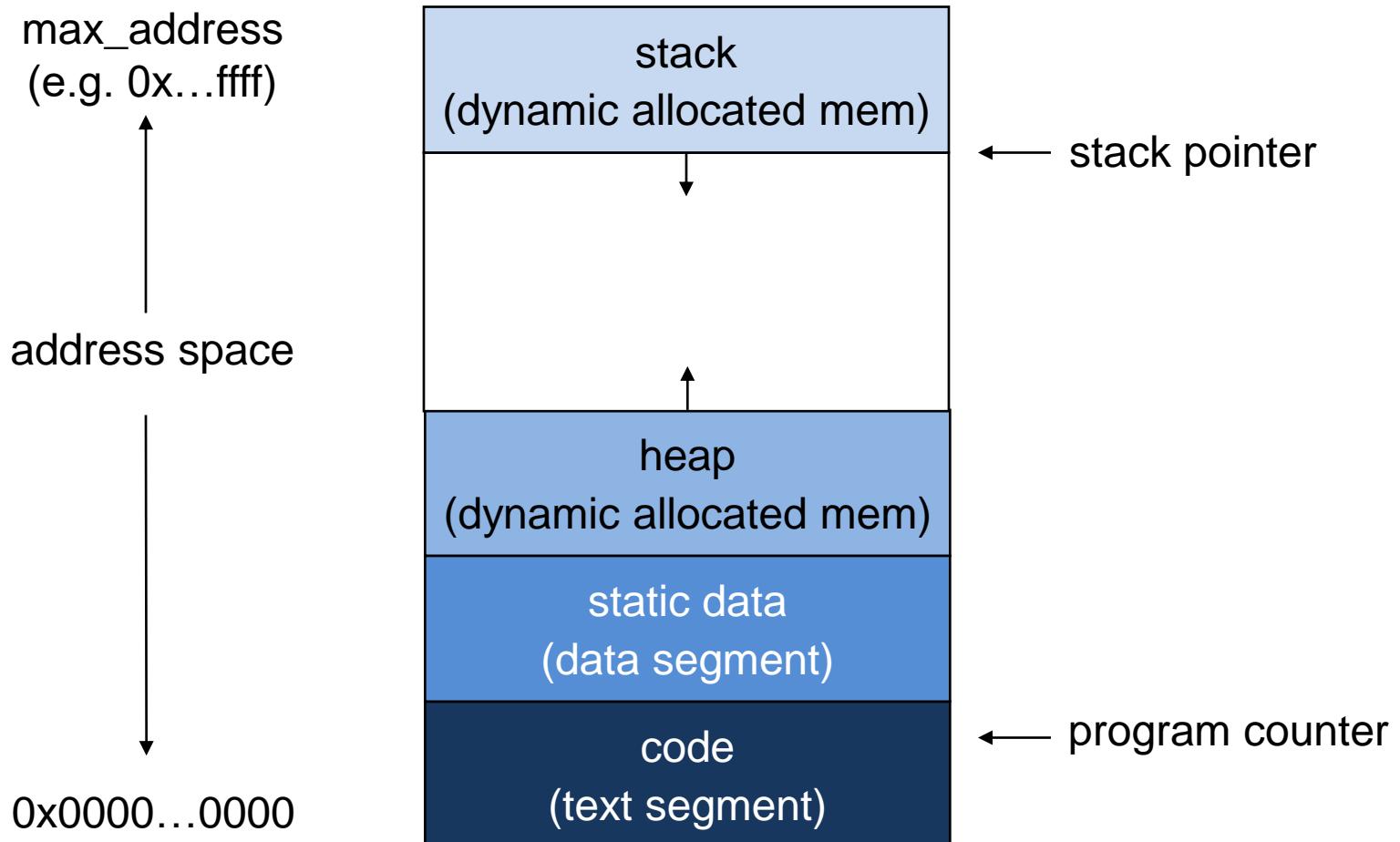
## ***Processes & Signals***

Dan Tsafrir  
2025-04-07, 2025-04-14

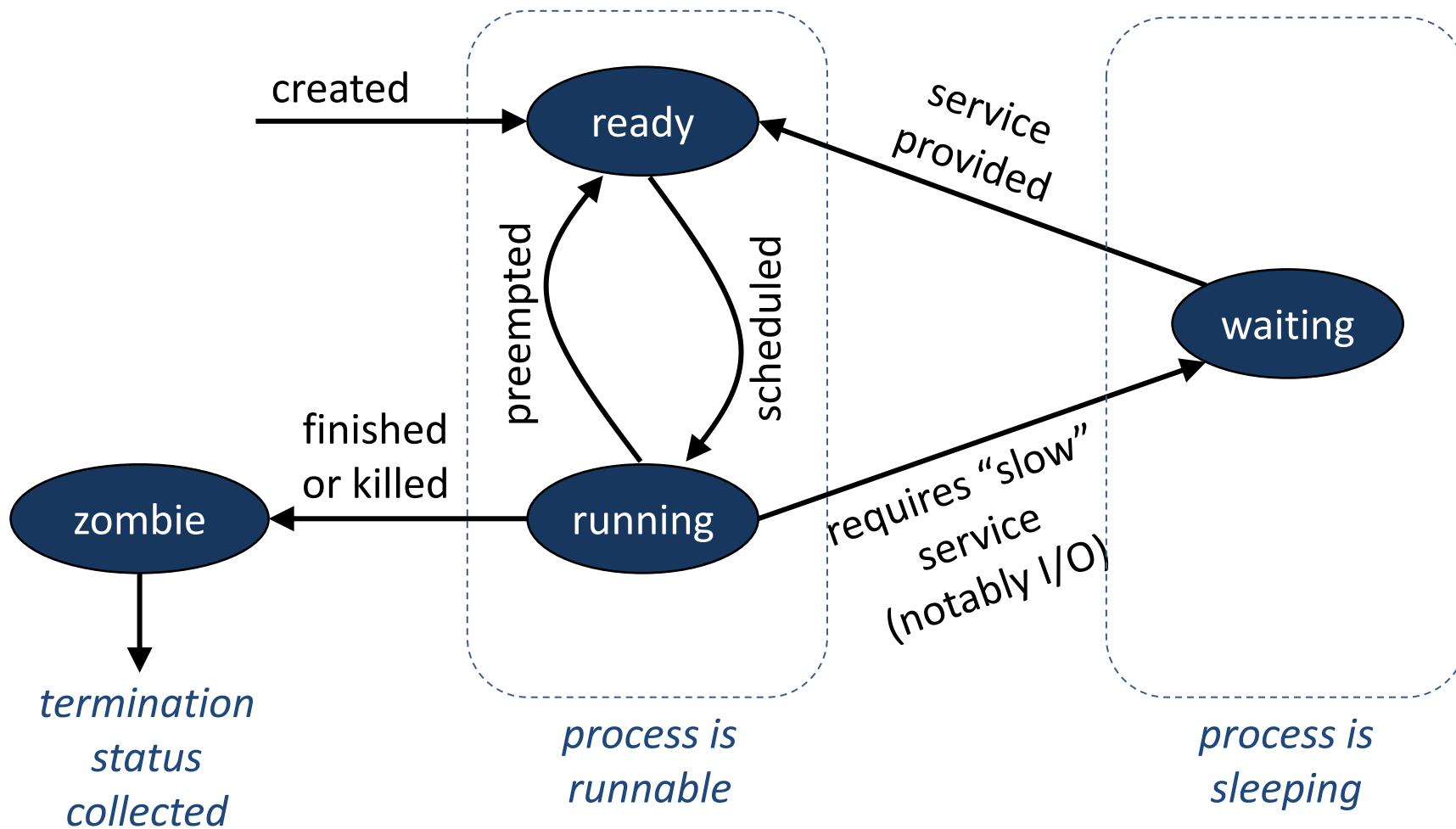
# What's a process

- **An implementation of the abstract machine concept**
  - Which we discussed in the previous lecture
- **A running instance of an executable, invoked by a user**
  - Can have multiple independent processes of the same executable
- **A schedulable entity, on the CPU**
  - OS decides which of these entities gets to run on a CPU core, and when
- **Sometimes called**
  - Task or job
- **The OS kernel is neither a process nor a schedulable entity**
  - Rather, it's a set of procedures executing in response to events (≈ interrupts)
  - Albeit sometimes the OS runs some code within schedulable entities
    - But then we prefer not to refer to these entities as “processes”, which correspond to *user* programs; we may refer to them as “**kernel threads**” instead

# Process address space is contiguous



# Process states



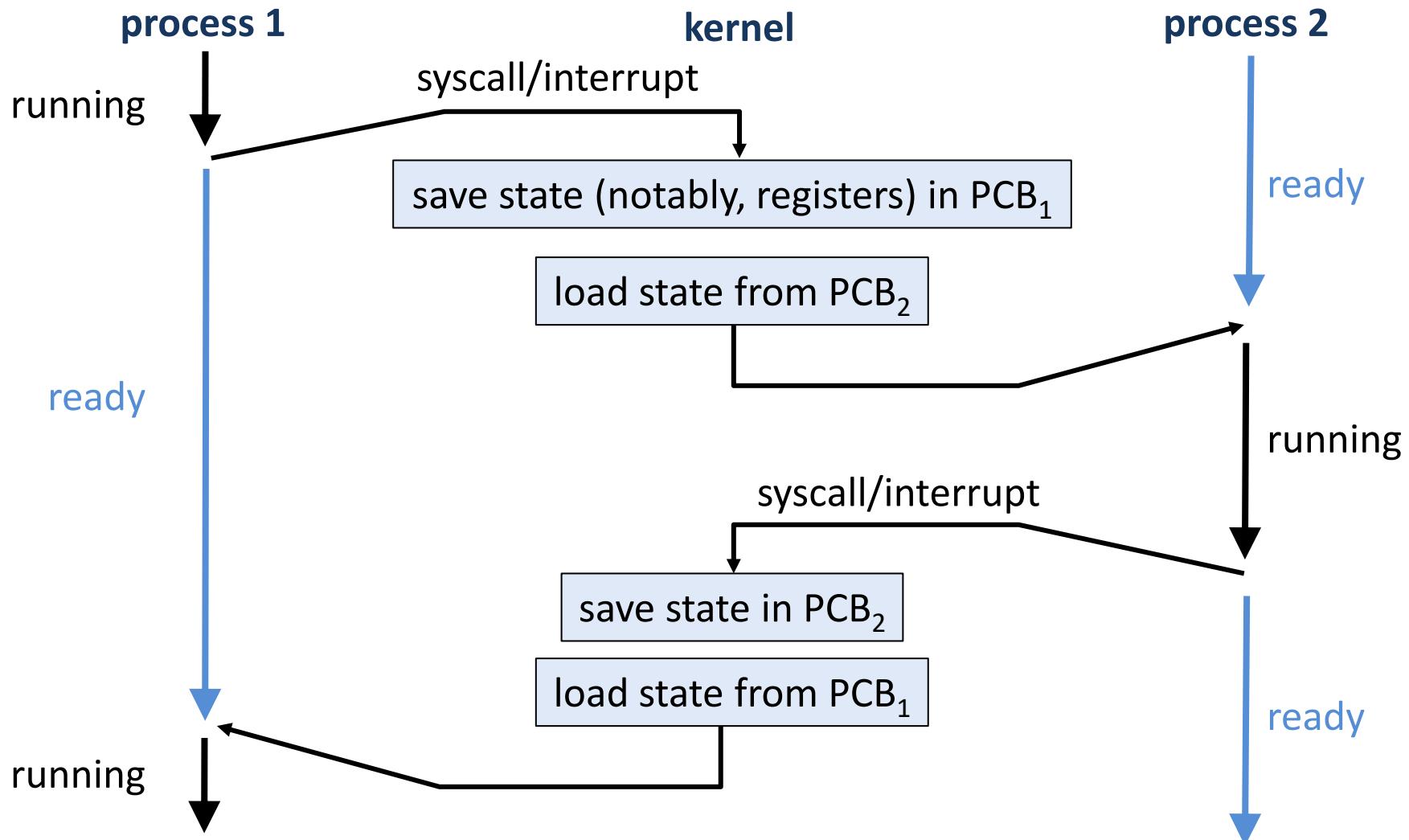
# Process control block (“PCB”)

- The OS maintains a “state” for every process
  - Encapsulated in a PCB
- In Linux
  - Called a “process descriptor”
  - Of type task\_struct (C struct)
  - Has O(100) fields
- Used in context switches
  - Updated upon preemption
  - Loaded upon resumption
- Question
  - Can a process access its own PCB?

- PID (process ID)
- UID (user ID)
- Pointer to address space
- Registers
- Scheduling priority
- Resources usage limits (e.g., memory, CPU, num of open files)
- Resources consumed
- State (previous slide)
- Current/present working directory (pwd=cwd)
- Open files table
- ...

process attributes  
in PCB

# Context switching (in runnable state)



# Process creation & termination

- **One process (the “parent”) can create another (the “child”)**
  - A new PCB is allocated and initialized
  - Homework: run ‘ps auxwww’ in the shell; PPID is the parent’s PID
- **In POSIX, child process inherits most of parent’s attributes**
  - UID, open files (should be closed if unneeded; why?), cwd, etc.
- **While executing, PCB moves between different queues**
  - According to state change graph
  - Queues: runnable, sleep/wait for event i (i=1,2,3...)
- **After a process dies (exit()s / interrupted), it becomes a zombie**
  - Parent uses `wait*` syscall to clear zombie from the system (why?)
  - Wait syscall family: wait, waitpid, waitid, wait3, wait4; example:
  - `pid_t wait4(pid_t, int *wstatus, int options, struct rusage *rusage);`
- **Parent can sleep/wait for its child to finish or run in parallel**
  - `wait*()` will block unless WNOHANG given in ‘options’
  - Homework: read ‘man 2 wait’

# fork() – spawn a child process

- **fork() initializes a new PCB**
  - Based on parent's value
  - PCB added to runnable queue
- **Now there are 2 processes**
  - At same execution point
- **Child's new address space**
  - Complete copy of parent's space, with one difference...
- **fork() returns twice**
  - At the parent, with pid>0
  - At the child, with pid=0
- **What's the printing order?**
- **'errno' – a global variable**
  - Holds error num of last syscall

```
int main(int argc, char *argv[])
{
    int pid = fork();
    if( pid==0 ) {
        //
        // child
        //
        printf("parent=%d son=%d\n",
               getppid(), getpid());
    }
    else if( pid > 0 ) {
        //
        // parent
        //
        printf("parent=%d son=%d\n",
               getpid(), pid);
    }
    else { // print string associated
            // with errno
            perror("fork() failed");
    }
    return 0;
}
```

# System call errors

```
// int errno = number of last system call error.  
// Errors aren't zero. (If you want to test value of  
// errno after a system call, need to zero it before.)  
#include <errno.h> // see man 3 errno  
  
// const char * const sys_errlist[];  
// char* strerror(int errnum) {  
//     // check errnum is in range  
//     return sys_errlist[errnum];  
// }  
#include <string.h>  
  
// void perror(const char *prefix);  
// prints: "%s: %s\n" , prefix, sys_errlist[errno]  
#include <stdio.h>
```

# **exec\*() – replace current process image**

- **To start an entirely new program**
  - Use the exec\*() syscall family; for example:
    - int `execv(const char *progamPath, char *const argv[]);`
  - Homework: read ‘man execv’
- **Semantics**
  - Stops the execution of the invoking process
  - Loads the executable ‘programPath’
  - Starts ‘programPath’, with ‘argv’ as its argv
  - Never returns (unless fails)
  - *Replaces* the new process; doesn’t create a new process
    - In particular, PID and PPID are the same before/after exec\*()

# Simplistic UNIX shell loop example

```
int main(int argc, char *argv[])
{
    for(;;) {
        int stat;
        char **argv;
        char *c = readNextCom(&argv);
        int pid = fork();

        if( pid < 0 ) {
            perror("fork failed");
        }
        else if( pid==0 ) { // child
            execv(c, argv);
            perror("execv failed");
        }
        else { // parent
            if( wait(&stat) < 0 )
                perror("wait failed");
            else
                chkStatus(pid,stat);
                release(argv);
        }
    }
    return 0;
}
```

```
void chkStatus(int pid, int stat)
{
    if( WIFEXITED(stat) ) {
        printf("%d exit code=%d\n",
               pid, WEXITSTATUS(stat));
    }
    else if( WIFSIGNALED(stat) ) {
        // the topic we're going
        // to learn next
        printf("%d died on signal=%d\n",
               pid, WTERMSIG(stat));
    }
    else if
        // a few more options...
    }
}
```

# Who wait()-s for an “orphan” process?

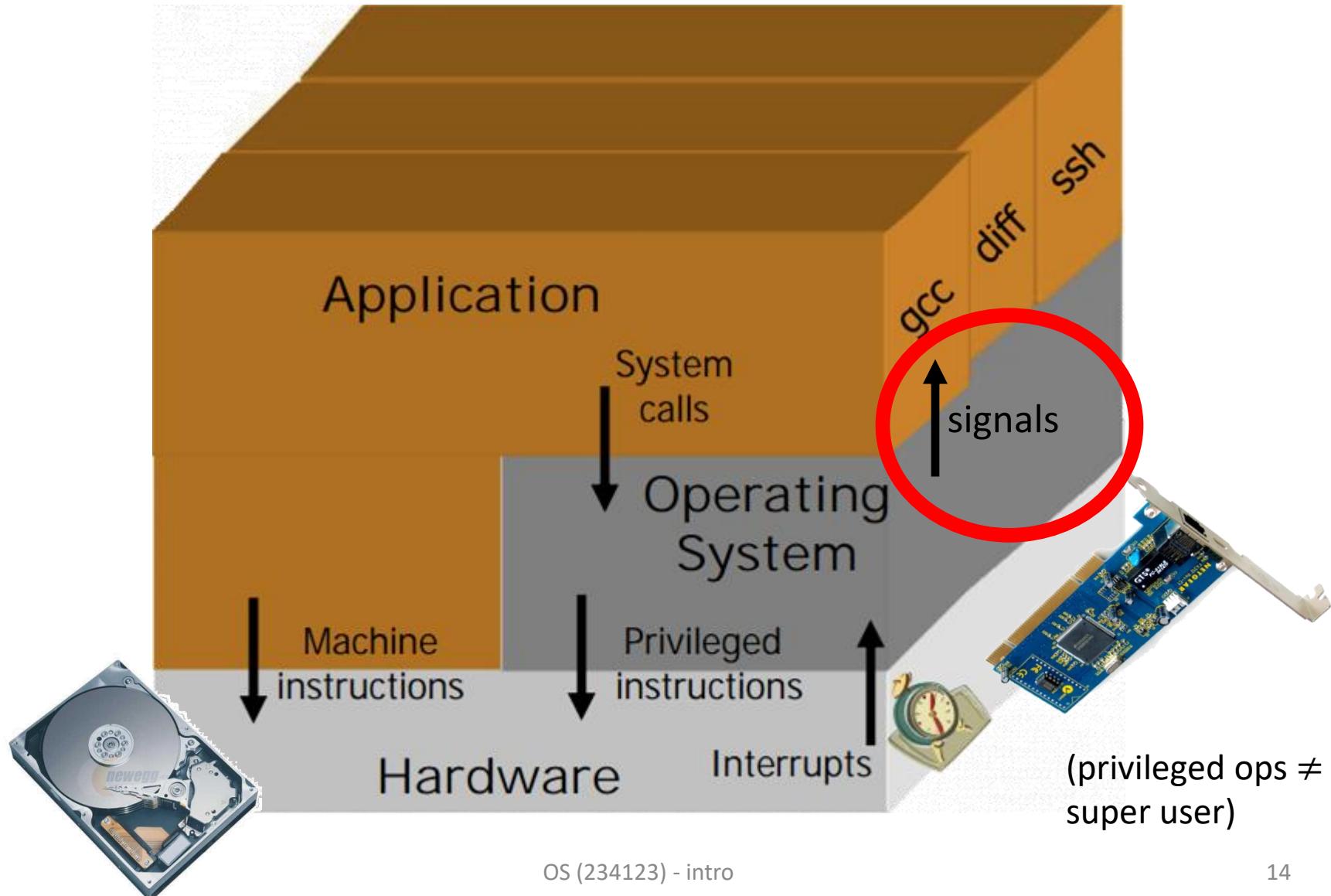
- **POSIX specification says:**
  - *“If a parent process terminates without waiting for all of its child processes to terminate, the remaining child processes shall be assigned a new parent process ID corresponding to an implementation-defined system process”*
    - <https://pubs.opengroup.org/onlinepubs/9699919799/functions/wait.html>
- **Linux manual says:**
  - *“init [...becomes] the parent of all processes whose natural parents have died, and it is responsible for reaping those when they die. [...] init expects to have a process id of 1”*
    - <https://linux.die.net/man/8/init>
  - *“If a parent process terminates, then its zombie children (if any) are adopted by init”*
    - <https://linux.die.net/man/2/wait>

OS-supported asynchronous notifications

# **POSIX SIGNALS**

# Reminder from the 1<sup>st</sup> lecture

Finished  
here



# What are signals & signal handlers

- **Signal = notification “sent” to a process**
  - To asynchronously notify it that some event occurred
- **Receiving a signal occurs when returning from kernel mode**
  - Instead of resuming the process ‘main’ thread
  - The kernel invokes the corresponding “signal handler”
- **Default signal handling action**
  - Either die or ignore (depends on the signal type)
- **The process can configure how it handles most signals**
  - Different signals can have a different handlers, and they can be temporarily blocked/unblocked = “**masked**”/“**unmasked**”
    - Except for 2 signals (well, actually 3 – discussed shortly)
- **Signals have names and numbers standardized by POSIX**
  - Do ‘man 7 signal’ in shell/Google for a listing/explanation of all signals
  - For example: <http://man7.org/linux/man-pages/man7/signal.7.html> , <https://pubs.opengroup.org/onlinepubs/9699919799/basedefs/signal.h.html>
  - **HOMEWORK:** take a few minutes to quickly survey all signals

# Silly example

```
#include <signal.h>
#include <stdio.h>
#include <stdlib.h>

void sigfpe_handler(int signum) {
    fprintf(stderr, "I divided by zero (sig=%d) !\n",
            signum); // prints SIGFPE's const value
exit(EXIT_FAILURE); // what happens if not exiting?
}

int main() {
    signal(SIGFPE, sigfpe_handler);
    int x = 1/0; // processor interrupt, then OS signal
    return 0;
}
```

# Another silly example

```
#include <signal.h>
#include <stdio.h>
#include <stdlib.h>

void sigint_handler(int signum) {
    printf("I'm disregarding your ctrl-c!\n");
}

int main() {
    // when pressing ctrl-c in the shell
    // => SIGINT is delivered to foreground process
    // (who makes this happen?)
    signal(SIGINT,sigint_handler);
    for(;;) { /*endless loop*/ }
    return 0;
}
```

# Another silly example

```
<0>dan@csa:~$ ./a.out
# here I clicked ctrl-c => delivered SIGINT
I'm disregarding your ctrl-c!
# here I clicked ctrl-c again => delivered SIGINT
I'm disregarding your ctrl-c!
# here I clicked ctrl-z => delivered SIGSTOP, must obey
[1]+  Stopped                  ./a.out
<148>dan@csa:~$ ps
 PID TTY          TIME CMD
 10148 pts/19    00:00:00 bash
 21709 pts/19    00:00:12 a.out
 21710 pts/19    00:00:00 ps
<0>dan@csa:~$ kill -9 21709    # 9=SIGKILL, must obey
[1]+  Killed                  ./a.out
<0>dan@csa:~$
```

# Another silly example

```
<0>dan@csa:~$ ./a.out  
# here I clicked ctrl-c (=> deliver SIGINT)  
I'm ignoring your crt1-c!  
# here I  
I'm ignor  
# here I  
[1]+  Sto  
<148>dan@
```

When I click ctrl-c, the OS gets an interrupt from the keyboard, which the OS then translates into a SIGINT signal delivered to the relevant process.

PID	TTY	TIME	CMD
10148	pts/19	00:00:00	bash
21709	pts/19	00:00:12	a.out
21710	pts/19	00:00:00	ps

```
<0>dan@csa:~$ kill -9 21709      # 9=SIGKILL, can't ignore  
[1]+  Killed                  ./a.out  
<0>dan@csa:~$
```

# Notice

- **Argument for the ‘kill’ shell utility**
  - Any signal (not just 9=kill)
    - kill -9 <pid>
    - kill -s KILL <pid>
    - kill -s SIGKILL <pid>
- **There are 2+1=3 signals that a process can’t ignore**
  - SIGKILL = terminate the receiving process
  - SIGSTOP = suspend the receiving process (make it sleep)
- **Is the effect of SIGSTOP reversible?**
  - Yes, when you send to the process SIGCONT
- **SIGCONT can’t be ignored either...**
  - A SIGSTOP-ed process *\*will\** continue
  - But process can set a handler for it, which will be invoked immediately when the process gets hit by the SIGCONT and is resumed as a result
- **What can you do with SIGSTOP/SIGCONT?**

# Job control

- **In the shell, assume we run a program ‘loop’ that does this**
  - `int main() { while(1) {}; return 0; }`
- **As noted, clicking ctrl-z in the shell**
  - Will make ‘loop’ sleep
- **Subsequently, invoking ‘fg’ in the shell**
  - Will wake ‘loop’ up in the **foreground** (meaning, the shell sleeps, and any typed input is directed to ‘loop’)
- **Alternatively, invoking ‘bg’ in the shell**
  - Will wake ‘loop’ up in the **background** (as if we executed it with “&”, so shell becomes operational, and typed input goes to the shell)
- **Simplifying lie I told**
  - For reasons related to job control, actually, ctrl-z
    - Generates STGTSTP, not SIGSTOP
  - There are subtle differences between the two, which we won’t learn

# This is how a signal is truly ignored

```
#include <signal.h>
#include <stdio.h>
#include <stdlib.h>

int main() {
    // Before, we used sigint_handler as the arg.
    // Now, we use the SIG_IGN macro, which means
    // no handler will be called.
    // There's also SIG_DFL, to restore the default
    // behavior

    signal(SIGINT, SIG_IGN);
    for(;;) { /*endless loop*/ }
    return 0;
}
```

# Example – ask a running daemon how much “work” it did thus far



- A “daemon” is
  - Background process, not controlled interactively by user
  - In shell: **nohup** <command> & (see: <https://linux.die.net/man/1/nohup>)
    - stdin = /dev/null; stdout = terminal if exists, nohup.out otherwise
  - Daemon name typically ends with “d” (e.g., sshd, syslogd, swapd)

```
void do_work() { for(int i=0; i<10000000; i++); }
int g_count=0; // counts num. of times do_work was invoked
void sigusr_handler(int signum) {
    printf("Work done so far: %d\n", g_count);
}
int main() {
    signal(SIGUSR1,sigusr_handler);
    for(;;) { do_work(); g_count++; }
    return 0;
}
```

# Example – ask a running daemon how much “work” it did thus far



```
<0>dan@csa:~$ ./a.out &  
[1] 23998  
# recall: kill utility also accepts strings as signals  
<0>dan@csa:~$ kill -s USR1 23998  
Work done so far: 626  
<0>dan@csa:~$ kill -s USR1 23998  
Work done so far: 862  
<0>dan@csa:~$ kill -s USR1 23998  
Work done so far: 1050  
<0>dan@csa:~$ kill -s KILL 23998  
[1]+  Killed ./a.out
```

# Enumerate signals

- **SIGSEGV, SIGBUSS, SIGILL, SIGFPE**
  - **ILL** = illegal instruction (trying to invoke privileged instruction)
  - **SEGV** = segmentation violation (illegal memory ref, e.g., outside an array)
  - **BUS** = dereference invalid address (null/misaligned, assume it's like SEGV)
  - **FPE** = floating point exception (despite name, actually *all* arithmetic errors, not just floating point; example: divide by zero)
  - These are driven by the associated **(HW) interrupts**
    - The OS gets the associated interrupt
    - The OS interrupt handler sees to it that the misbehaving process gets the associated signal
    - The default signal handler for these signals: core dump + die
- **SIGCHLD**
  - Parent gets it whenever fork()ed child terminates or is SIGSTOP-ed
- **SIGALRM**
  - Get a signal after some specified time
  - Set by system calls: **alarm(2)** & **setitimer(2)** (homework: read man)

# Enumerate signals

- **SIGTRAP**
  - When debugging / single-stepping a process
  - E.g., can be delivered upon each instruction
- **SIGUSR1, SIGUSR2**
  - User decides the meaning (e.g., see our daemon example)
- **SIGXCPU**
  - Delivered when a process used up more CPU than its soft-limit allows
  - Soft/hard limits are set by the system call: `setrlimit()`
  - Soft-limits warn the process it's about to exceed the hard-limit
  - Exceeding the hard-limit => SIGKILL will be delivered
- **SIGPIPE**
  - Write to pipe with no readers (we'll learn about pipes later, for the time being think about the shell's pipe: "|")

# Enumerate signals

- **SIGIO**
  - Can configure file descriptors such that a signal will be delivered whenever some I/O is ready
  - Typically makes sense when also configuring the file descriptors to be “non blocking”
    - E.g., when `read()`ing from a non-blocking file descriptor, the system call immediately returns to user if there’s currently nothing to read
    - In this case, `errno` will be set to `EAGAIN` = `EWOULDBLOCK`
- **And a few more**
  - man 7 signal
  - <http://man7.org/linux/man-pages/man7/signal.7.html>

# Signal vs. interrupts

	interrupts	signals
Who triggers them? Who defines their meaning?	Hardware: CPU cores (sync) & other devices (async)	Software (OS), HW is unaware
Who handles them? Who (un)blocks them?	OS	processes
When do they occur?	Both synchronously & asynchronously	Likewise, but, technically, invoked when returning from kernel to user

# Signal system calls – sending

- **int kill(pid\_t pid, int sig)**
  - (Not the shell utility, the actual system call)
  - Allows a process to send a signal to another process (or to itself)
    - **Homework:** How?
  - man 2 kill – <http://linux.die.net/man/2/kill>
    - “2” is for system calls
    - “1” is for shell utilities

# Signal system calls – (un)blocking

- **Signals can be asynchronous => might lead to "race conditions"**
  - Therefore, as noted, all signals (except kill/stop) can be blocked
  - Like how OS disables/enables interrupt
- **How**
  - The PCB maintains a set of currently blocked signals
  - Which can be manipulated by users via the following syscall
- **int sigprocmask(int *how*, const sigset\_t \**set*, sigset\_t \**oldset*)**
  - ‘how’ = SIG\_BLOCK (+=), SIG\_UNBLOCK (-=), SIG\_SETMASK (=)
  - ‘set’ can be manipulated with sigset ops **sigemptyset**(sigset\_t \*set),  
**sigfillset**(sigset\_t \*), **sigaddset**(sigset\_t \*set, int signum),  
**sigismember**(sigset\_t \*set, int signum)
- **Manual**
  - man 2 sigprocmask – <http://linux.die.net/man/2/sigprocmask>
  - man 3 sigsetops – <https://linux.die.net/man/3/sigsetops>

# (Un)blocking example

```
Record_t db[N];  
  
void my_handler(int signum) {  
    /* may read/update db */  
}  
  
int main(int argc, char *argv[])  
{  
    sigset_t mask, orig_mask;  
    sigemptyset(&mask);  
    sigaddset(&mask, SIGTERM);  
  
    signal(SIGTERM, my_handler);  
  
    for(;;) {  
        char *cmd = read_command();  
        sigprocmask(SIG_BLOCK, &mask, &orig_mask);  
        // do stuff that may read/update db...  
        sigprocmask(SIG_SETMASK, &orig_mask, NULL);  
    }  
    return 0;  
}
```

Recall that every syscall might fail;  
the example ignores this for brevity

# Implementation

- **The kernel maintains two bitmaps**
  - Blocked signals
  - Pending signals
- **A matching array of pointer to functions**
  - Initialized to SIG\_DFL
  - May hold SIG\_IGN
  - Assigned values via syscall
- **When in kernel, before returning to user**
  - Check if there's a pending, non-blocked signal
  - In which case invoke the corresponding handler

# Signal system calls – control & more info

- Additional info about signal & fine-grain control
  - over how signals operate is provided via the following syscall
- **int sigaction(int signum,  
const struct sigaction \*act,  
struct sigaction \*oldact)**
  - man 2 sigaction – <http://linux.die.net/man/2/sigaction>  
(homework: read it)

# Signals interact with other system calls

- **ssize\_t read(int fd, void \*buf, size\_t count);**
  - What happens if getting signal while read()ing?
  - The read system call returns -1, and it sets the global variable ‘errno’ to hold EINTR
  - An example whereby read() might fail and user should simply retry

```
int readn( int sockfd /*learn later*/, char *ptr, int nbytes )
{
    int nleft = nbytes;

    while( nleft > 0 ) { // 'read' is typically done in a loop (why)?
        int nread = read(sockfd, ptr, nleft);
        if( (nread == -1) && (errno != EINTER) ) {
            fprintf(stderr, "read failed, errno=%m", errno);
            return -1;
        }
        else if( nread == 0 )
            break; /*EOF*/

        nleft -= nread;
        ptr    += nread;
    }
    return nbytes - nleft;
}
```

# Operating Systems (234123)

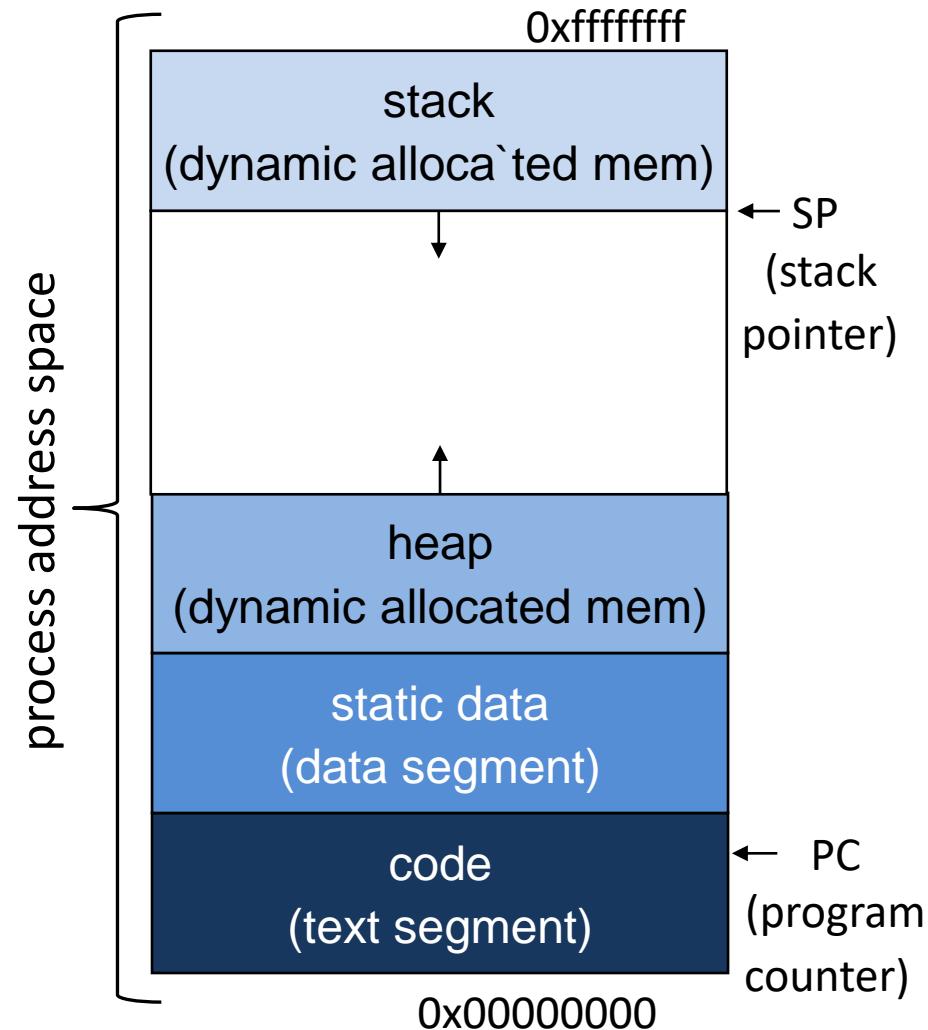
*Parallelizing  
with processes & threads  
using different IPC mechanisms  
(IPC = inter-process communication)*

*[partially: chapter #3 in Feitelson's OS notes]*

Dan Tsafrir (2021-04-12)

# Reminder: process = running program instance

- **Has**
  - Data, stack, code, PC pointing to code, registers
- **Associated with a state**
  - Running
  - Ready (to run)
  - Waiting
- **Associated with a PCB (process control block)**
  - Saves all kernel objects & info associated with the process: PID, open files, priority...
  - Save state of process when not running (registers)



# Multiprocessing

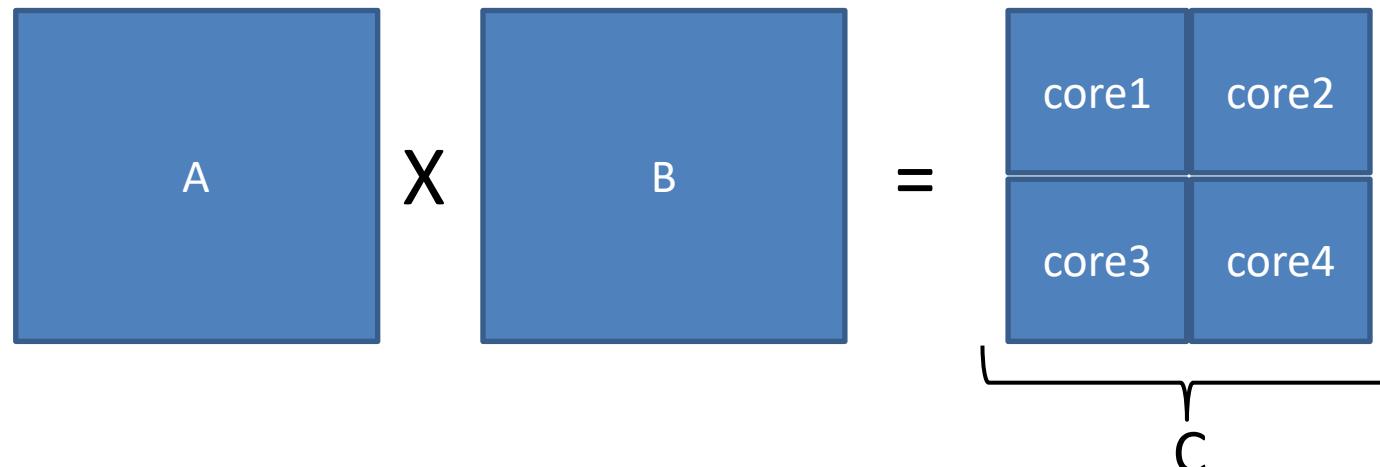
- **Definition**
  - Using several CPU cores (= processors) for running a single job to solves single “problem”
- **Motivation**
  - Finish the job faster
- **Example**
  - Web server, which gets requests and returns replies
  - Replies can be
    - Static (existing files) or
    - Dynamic (computed on the fly)

# Web server implementation

- **Without multiprocessing (problem)**
  - `while ( true )`  
`< client , request > = get_next_request();`  
`reply = handle( request ); /* wait for it to finish... */`  
`send( client , reply );`
  - Simple, but handling might take a long time (potentially doing slow I/O), delaying other pending requests
- **With multiprocessing (solution)**
  - `while ( true ):`  
`< client , request > = get_next_request();`  
`if( fork() == 0 ) /* multi-processing! child is doing the work*/`  
`reply = handle( request );`  
`send( client , reply );`
  - Not as simple, but processing is done in parallel on different cores
- **Comment**
  - Often done using bounded number of pre-fork()ed children (why?)

# How about matrix multiplication?

- Let A and B be matrices and assume we want to compute  $C = A \times B$ 
  - The matrices are BIG
  - Yet, the physical memory is big enough to hold them
- Question
  - Assuming we have a 4-way multicore, can we use it to speed up the computation in a similar manner to the web server?
- Possible answer
  - Let's divide C into quarters and have each core run its own process and compute its own quarter



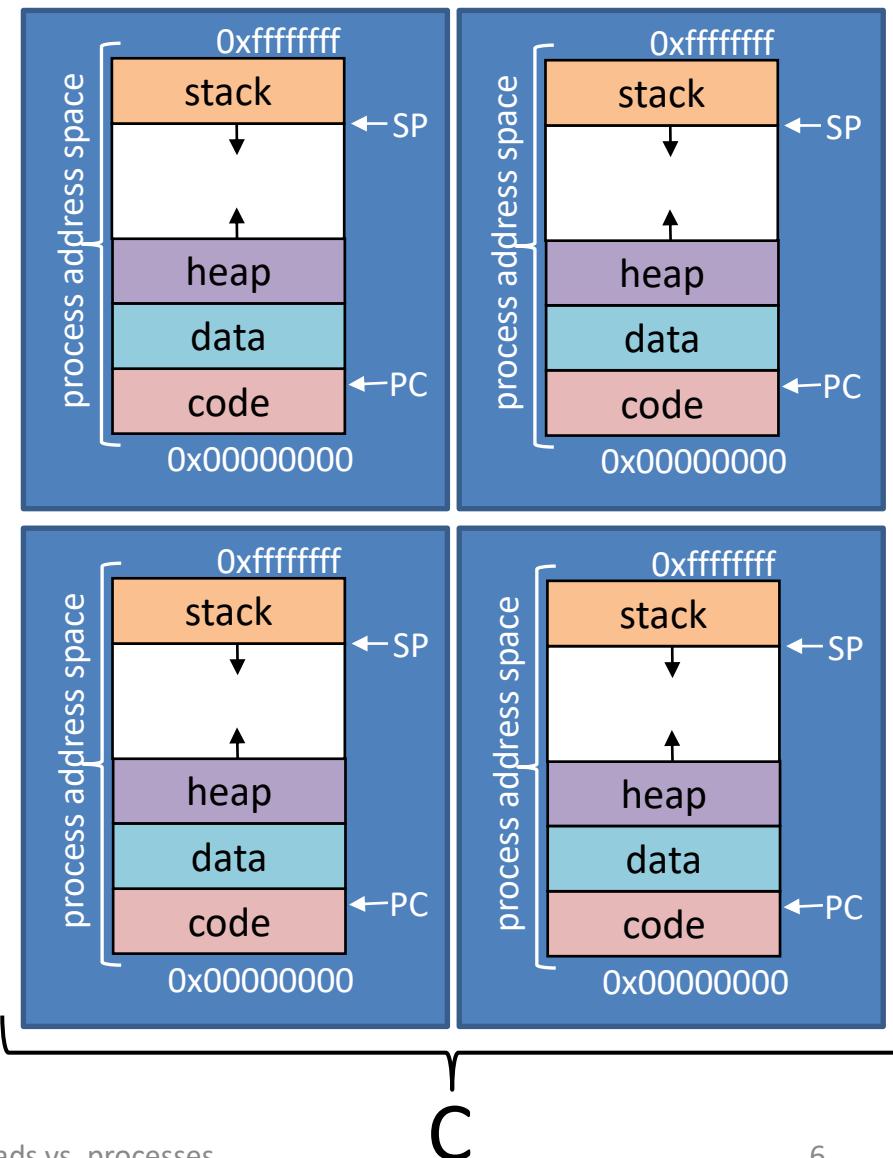
# How about matrix multiplication?

- **Problem**

- The OS protects the 4 processes from one another
- They are isolated & don't share
- Each has its own memory space and, specifically, its own data & heap

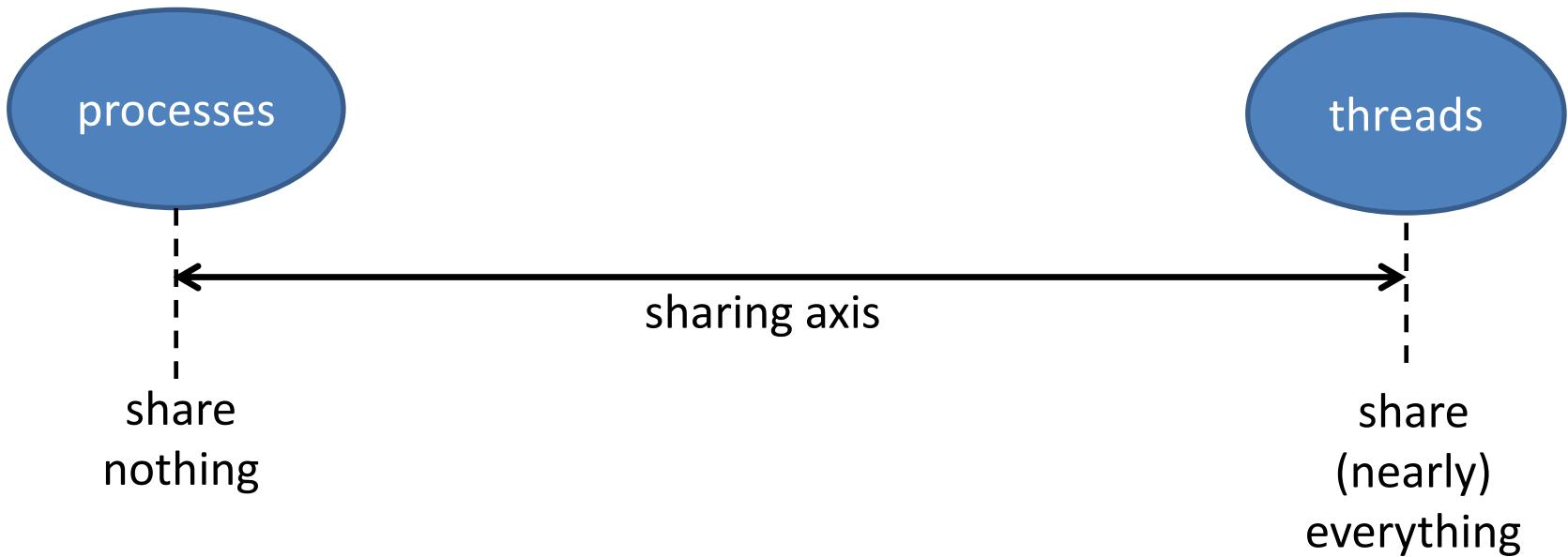
- **So 4 copies of matrices are created**

- If we're not careful, the copies might be bigger than the memory
- Also, need to perform lots of communication (=copying) such that one core will be able to combine the four quarters into one result (details: soon)



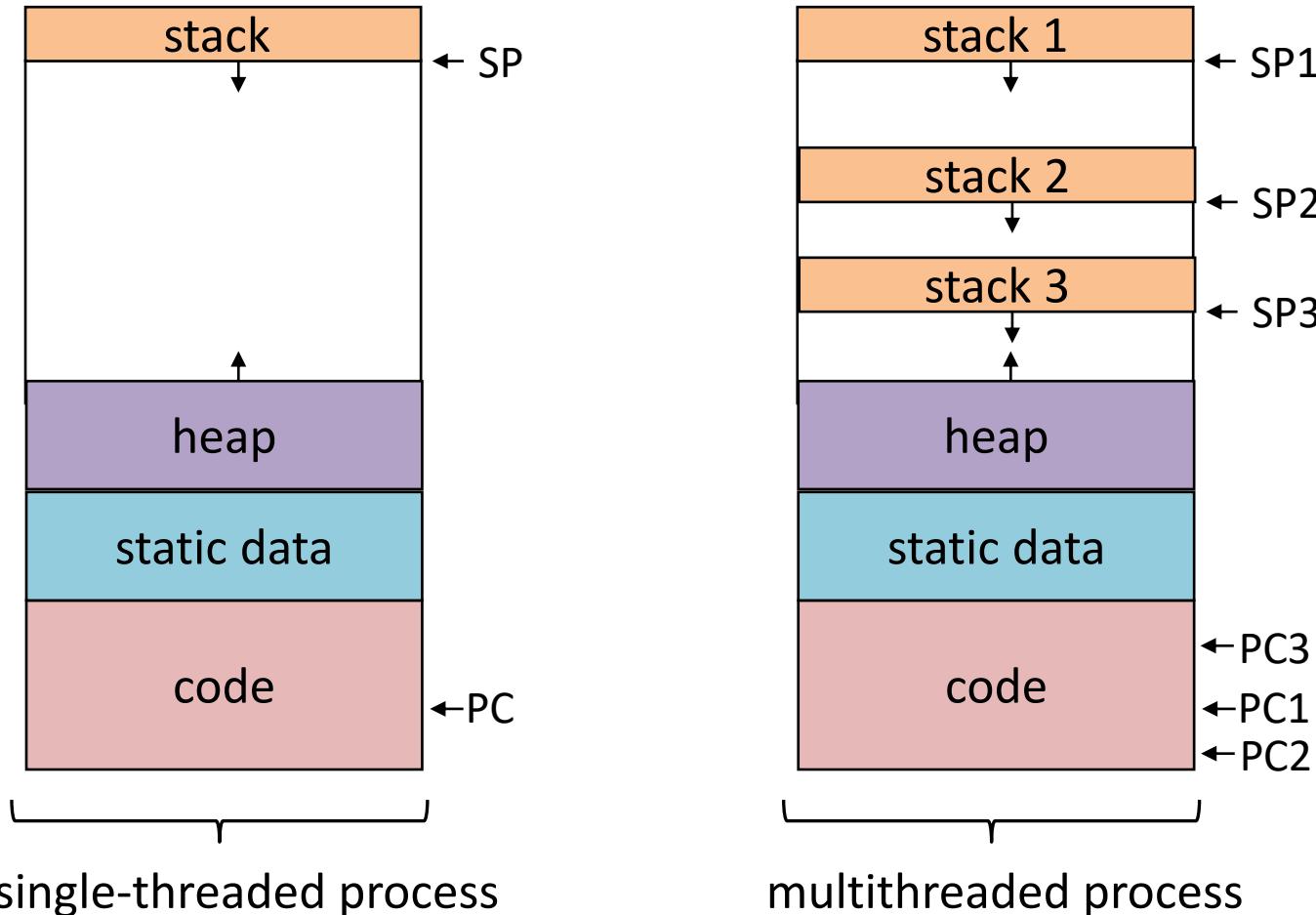
# Solution: multithreading

- At the very least, computing entities must have their own
  - Registers & stack
- But all of their other state could be shared
  - Notably heap & data



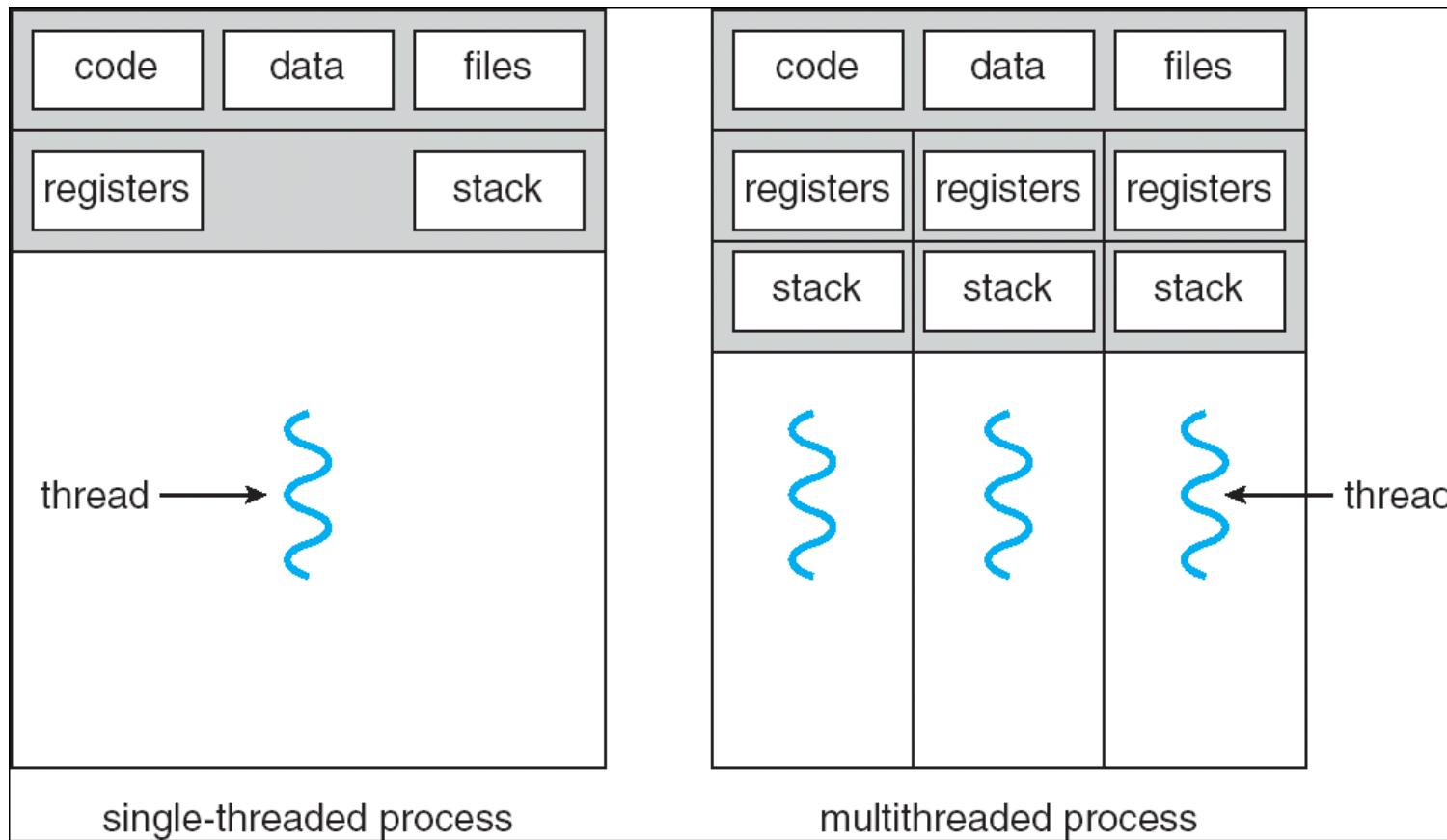
# Solution: multithreading

A process has (or “contains”) several threads of execution, sharing everything but the stack & registers



# Solution: multithreading

(Sometimes people use the following alternative illustration, which depicts exactly the same thing)



# Standard APIs for threads – OpenMP

- Stands for “Open Multi-Processing”
- Consists of a set of compiler ‘pragma’ directives, such that the code looks like it’s serial, but the compiler directives make it parallel

```
#pragma omp parallel for
for (i = 0; i < N; i++)
    arr[i] = 2 * i;
```
- Namely, the serial and parallel versions of the code are unified
  - It’s the same code
- Implemented in
  - C, C++, and Fortran
  - In gcc, use the compilation flag -fopenmp
- Supported by all major operating systems
  - Including Windows
- More details:
  - <http://en.wikipedia.org/wiki/OpenMP>

# Standard APIs for threads – language

- Thread support may be provided by the runtime of the programming language runtimes
  - Java (has no fork)
    - import java.lang.Thread
  - C++ (as of 2011)
    - #include <thread> // for std::thread
  - ...
- May enjoy explicit language support (such as keywords)
  - Java
    - “synchronized” (for methods or statements)
  - C++
    - thread\_local (storage class specifier)

# Standard APIs for threads – pthreads

- **Stands for “POSIX threads”**
  - Recall: POSIX is the standard for UNIX operating systems (POSIX = “portable operating system interface”)
- **A standard C library**
  - Include <pthread.h> and when compiling use gcc -lpthread ...
- **Also implemented on Windows**
  - pthreads-w32, which also support 64bit... ☺
- **The library we use in this course**
  - We'll soon see an example

# Sharing

	unique to process	unique to pthread
registers (notably PC)	Y	Y
execution stack	Y	Y
memory address space	Y	N
open files	Y	N
per-open-file position (=offset)	Y	N
working directory	Y	N
user/group credentials	Y	N
signal handling	Y	N
...		

- In some OSes, the kernel implements processes and threads differently
  - Such that, internally, in the kernel, each process really is a container of threads
  - Examples: Windows and Solaris

# Sharing

	unique to process	unique to pthread
registers (notably PC)	Y	Y
execution stack	Y	Y
memory address space	Y	N
open files	Y	N
per-open-file position (=offset)	Y	N
working directory	Y	N
user/group credentials	Y	N
signal handling	Y	N
...		

- **In Linux, internally, ‘process’ and ‘thread’ are exactly the same**
  - Process = thread = “task”; each has its own PCB (= “task\_struct”)
  - Logically, every item in above table is a pointer within task\_struct, such that when 2 tasks share an item, it means they both point to it
- **Users have fine-grain control on what’s being shared via the ‘clone’ syscall**
  - fork() and pthread\_create() are simply wrapper functions of clone()

# Linux's clone system call

- **Creating a (share-everything) “thread”**
  - `clone( child_stack=0x420cc260, flags=CLONE_VM|CLONE_FS|CLONE_FILES|CLONE_SIGHAND|CLONE_THREAD|CLONE_SYSVSEM|CLONE_SETTLS|CLONE_PARENT_SETTID|CLONE_CHILD_CLEARTID, parent_tidptr=0x420cc9e0, tls=0x420cc950, child_tidptr=0x420cc9e0 )`
- **Creating a (share-nothing) process**
  - `clone( child_stack=0, flags=CLONE_CHILD_CLEARTID|CLONE_CHILD_SETTID|SIGCHLD, child_tidptr=0x7f4936ecc770 )`
- **See ‘man clone’**
  - No need to understand every flag, just the principle
  - Example: if CLONE\_VM is set, then parent+child share memory space

# Intermediate summary

- **So far**
  - Abstract discussion
- **Next**
  - Examples:
    1. For communication via explicit “message passing”
      - Which is required when using processes, because they don’t share
    2. For communication via “shared memory”
      - Which is possible for threads, because they share their address space

# In POSIX, “everything is a file”

- **File descriptor**
  - A nonnegative integer representing an I/O “channel” on some device
- **It can be obtained, for example, like so**
  - `int fd = open("/some/file", flags);`
- **Given a file descriptor, one can interact with the underlying device to perform I/O via the ‘read’ and ‘write’ system calls**
  - `int read(int fd, char *buf, size_t n);`
  - `int write(int fd, char *buf, size_t n);`
- **Another way to obtain a file descriptor is....**

# Message passing communication – example

- **We will discuss a (POSIX) communication channel called a “pipe”**
  - There are other types of communication channels that allow us to send messages, but this is perhaps the simplest example
  - We’ll use the pipe to deliver a single message from one process (parent) to another (child)
- **A pipe is a pair of two file descriptors (fd for short)**
  - `int pipe_fd[2]`
- **Such that each integer is a handle to a kernel communication object**
  - `pipe_fd[0]` = read side of the communication channel
  - `pipe_fd[1]` = write side of the communication channel
  - Everything written via `pipe_fd[1]` can be read via `pipe_fd[0]`
  - Sending and receiving messages
    - Done through the kernel using the `write()` & `read()` system calls
  - The kernel saves everything that is being written via `pipe_fd[1]`, such that later it’ll be able to serve subsequent reads from `pipe_fd[0]`

# Message passing communication – example

- **Code:**
  - int pipe\_fd[2];
  - char src\_buf[N] = “...”;
  - char dst\_buf[N];
  - **write( pipe\_fd[1], src\_buf, N) ; // returns num of bytes written**
    - Go to the kernel and copy (at most) N bytes from src\_buf to the communication channel identified by pipe\_fd[1]
  - **read( pipe\_fd[0], dst\_buf, N); // returns num of bytes read**
    - Go to the kernel and copy to dst\_buf (at most) N bytes; these were previously written to the communication channel identified by pipe\_fd[1] (which, as noted, is associated with pipe\_fd[0])
- **Notes**
  - When a running process attempts to read through pipe\_fd[0] but nothing has been written yet via pipe\_fd[1], the process will block
  - Writing to a pipe whose read end is close()d => SIGPIPE (“broken pipe”)

# Message passing communication – example

```
/*
 * Macro providing a "safe" way to invoke system calls
 */
#define DO_SYS( syscall ) do { \
    /* safely invoke a system call */ \
    if( (syscall) == -1 ) { \
        perror( #syscall ); \
        exit(1); \
    } \
} while( 0 ) \

```

# Message passing communication – example

```
/*
 * Why do we need the do-while trick?
 * Why not just the if?
 */

#define DO_SYS( syscall ) \
    if( (syscall) == -1 ) { \
        perror( #syscall ); \
        exit(1); \
    }

/*
 * Because then, code like this won't compile
 */

if( condition ) \
    DO_SYS( read(fd,buf,bufsiz) );
else
    printf("can't read()");
```

# Message passing communication – example

```
char g_msg[N];           /* "g" stands for "global" */
enum {RD=0, WT=1};
int main() { // send g_msg from parent to child
    int fd[2];
    DO_SYS( pipe(fd) ); // establish communication channel
    if( fork() != 0 ) { // parent writes, so
        DO_SYS( close(fd[RD]) ); // close read side
        fill_g_msg(); // don't care how
        DO_SYS( write( fd[WT], g_msg, N ) );
        DO_SYS( wait( NULL ) ); // for child to end
    }
    else { // child reads, so
        DO_SYS( close(fd[WT]) ); // close write side
        DO_SYS( read( fd[RD], g_msg, N ) );
    }
    return 0;
}
```

# Message passing communication – example

- **Actually, there are two g\_msg arrays**
  - One for parent and another for child
  - They appear to be the same array, but they are not
  - Reason: with fork, everything is a copy
- **Actually, there are 4 file descriptors**
  - 2 of parent + 2 of child
  - They appear to be the same fd-pair, but they are not (though both read-s are connected to both write-s)
  - Reason: with fork, everything is a copy
- **The write / read system calls**
  - Copy g\_msg to and from the kernel
- **A file descriptor is in fact**
  - An index to a kernel array of channels
- **Question**
  - Can the child send msg back to parent?

```
r "global" */  
  
nt to child  
  
sh communication channel  
// parent writes, so  
; // close read side  
// don't care how  
g_msg, N ) );  
// for child to end  
  
// child reads, so  
; // close write side  
_msg, N ) );
```

# Message passing communication – example

- **Comment: read() and write() should often be wrapped in a loop**
    - Depending on what the program needs
    - Recall that both syscalls return the number of bytes they read/written

- Comment: `read()` and `write()` should often be wrapped in a loop
  - Depending on what the program needs
  - Recall that both syscalls return the number of bytes they read/written

```
r "global" */  
nt to child  
lsh communication channel  
    // parent writes, so  
    DO_SYS( close(fd[RD]) );      // close read side  
    fill_g_msg();                // don't care how  
    DO_SYS( write( fd[WT], g_msg, N ) );  
    DO_SYS( wait( NULL ) );      // for child to end  
}  
  
else {  
    // child reads, so  
    DO_SYS( close(fd[WT]) );      // close write side  
    DO_SYS( read( fd[RD], g_msg, N ) );  
}  
return 0;  
}
```

# Shared memory communication – example

```
// Assume that filling g_msg requires a lot of
// computational work.
// So we want to use 2 threads, one thread to fill the
// first half of g_msg, and another to fill the second half
void fill_g_msg( void )
{
    pthread_t t1, t2;

    // launch the two threads
    pthread_create(&t1, NULL, thread\_fill, "first");
    pthread_create(&t2, NULL, thread\_fill, "second");

    // wait for both threads to finish
    pthread_join(t1, NULL);
    pthread_join(t2, NULL);
}
```

# Shared memory communication – example

```
void* thread_fill(void *arg)
{
    // initially, assume I'm the "first" thread
    int i;
    int lo = 0;
    int hi = N/2;

    if( strcmp((char*)arg, "second") == 0 ) {
        // I'm the "second" thread, filling 2nd half
        lo = N/2 + 1;
        hi = N - 1;
    }

    for(i = lo; i <= hi; i++)
        g_msg[i] = /*do some really hard work here*/ ;

    return null;
}
```

# Shared memory communication – example

```
void* thread_fill(void *  
{  
    // initially, ass  
    int i;  
    int lo = 0;  
    int hi = N/2;  
  
    if( strcmp((char*)  
                // I'm the  
                lo = N/2 +  
                hi = N - 1  
    }  
  
    for(i = lo; i <=  
        g_msg[i] =  
  
    return null;  
}
```

- **It's only the parent who invokes fill\_g\_msg()**
  - Which means that it's only the parent (not child) who spawns threads
- **Since the parent uses threads**
  - They share the same g\_msg
  - (it's a global variable)
- **Consequently**
  - It's fine that the first thread fills the first half of g\_msg, and that the second thread fills the remaining half of g\_msg
  - Together, the two threads fill g\_msg in its entirety

# Bottom line

- **Thread or process?**
  - Depends on how you want your tasks to communicate
- **Threads = communication is done via shared memory**
  - In principle, more efficient, as doesn't create a copy ("zero copy")
  - But often means we have to access the memory in a synchronized manner, which, as we will see, could be \*challenging\* to get right
    - Right = correct & efficient; more on that in future lectures
  - When synchronization is not an issue (because it's done very coarsely, or because we have a programming model / data structure that hides synchronization issues from ordinary programmers), then it is easy and convenient
- **Processes = communication is done via explicit message passing**
  - Tasks explicitly exchange messages via read/write or send/receive
  - Sometimes requires making a copy of data => less efficient
  - Typically requires the use of system calls => less efficient
  - Much easier to understand, get right, and reason about (prove stuff)

Let's discuss

# **THE COST OF CONTEXT SWITCHING**

# Cost of context switch is relative

- The overall overhead of context switching is largely determined by how long tasks run before they are preempted
  - Assume that a context switch takes  $C$  time (say, microseconds)
  - Assume tasks run at least  $K \cdot C$  microseconds before they are preempted
  - Then the rate of cycles we spend on context switching is:
    - $C / (K \cdot C + C) = 1/(K+1)$  //  $K > 0$ , real number (not necessarily integral)
    - Namely, we spend  $1/(K+1)$  fraction of the time on context switching
    - E.g., with  $K=1$ , half the time is spent on context switching
  - Consequently, bigger  $K$  implies a smaller cost
  - Or, in other words, the longer tasks run before they are preempted, the smaller the cost of context switching becomes
- How is  $K$  determined?
  - By the kernel (quantum duration)
  - By the application (how long can it run before having to wait)

# Context switch overhead consists of 2 components

- **Direct overhead**
  - How long it takes for the kernel to save the state (= registers) of the previous task and resume the state of the next task
  - Can be roughly measured within the kernel as follows
    - `cycles_t before = get_cycles(); /* CPU cycles; nanosecs if 1GHz */  
do_context_switch(prev_task, next_task);  
cycles_t after = get_cycles();  
cycles_t direct_overhead = after - before;`
- **Indirect overhead**
  - The time it takes for the hardware to reconstruct the state it (= the hardware) created in order to accelerate the execution of the task
  - The hardware does this reconstruction while the task is running
  - To understand this component, we need to have some idea about how processors work

Computer architecture course (“MAMAS”) in a nutshell

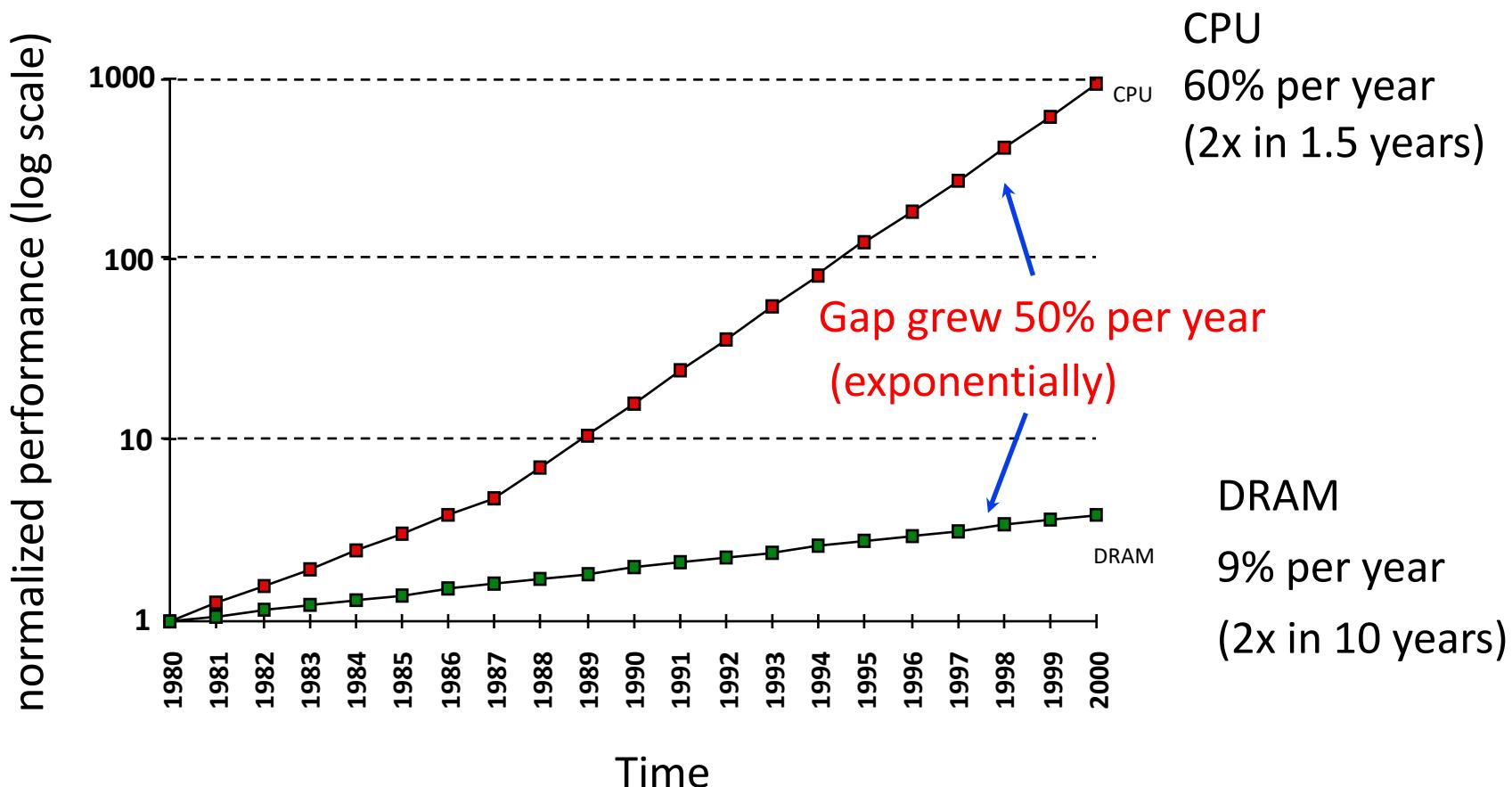
# **CACHING & THE MEMORY HIERARCHY**

# Problem: CPU faster than main memory

- **Reminder**
  - milli=1:1000, micro=1:1,000,000, nano=1:1,000,000,000
- **CPU**
  - Each core can do up to billions of operations in one second
    - If the speed of the processors is “1GHz”, it (roughly) means that the processor can do one billion ( $10^9$ ) instructions per second
    - Or one instruction per nanosecond
  - The time-per-instruction is called the “cycle” of the CPU
- **Main memory (DRAM)**
  - Has a latency (= time it takes to read/write data from/to memory) which is typically longer than 100 CPU cycles
  - So memory is nowadays oftentimes  $>100x$  slower than CPU

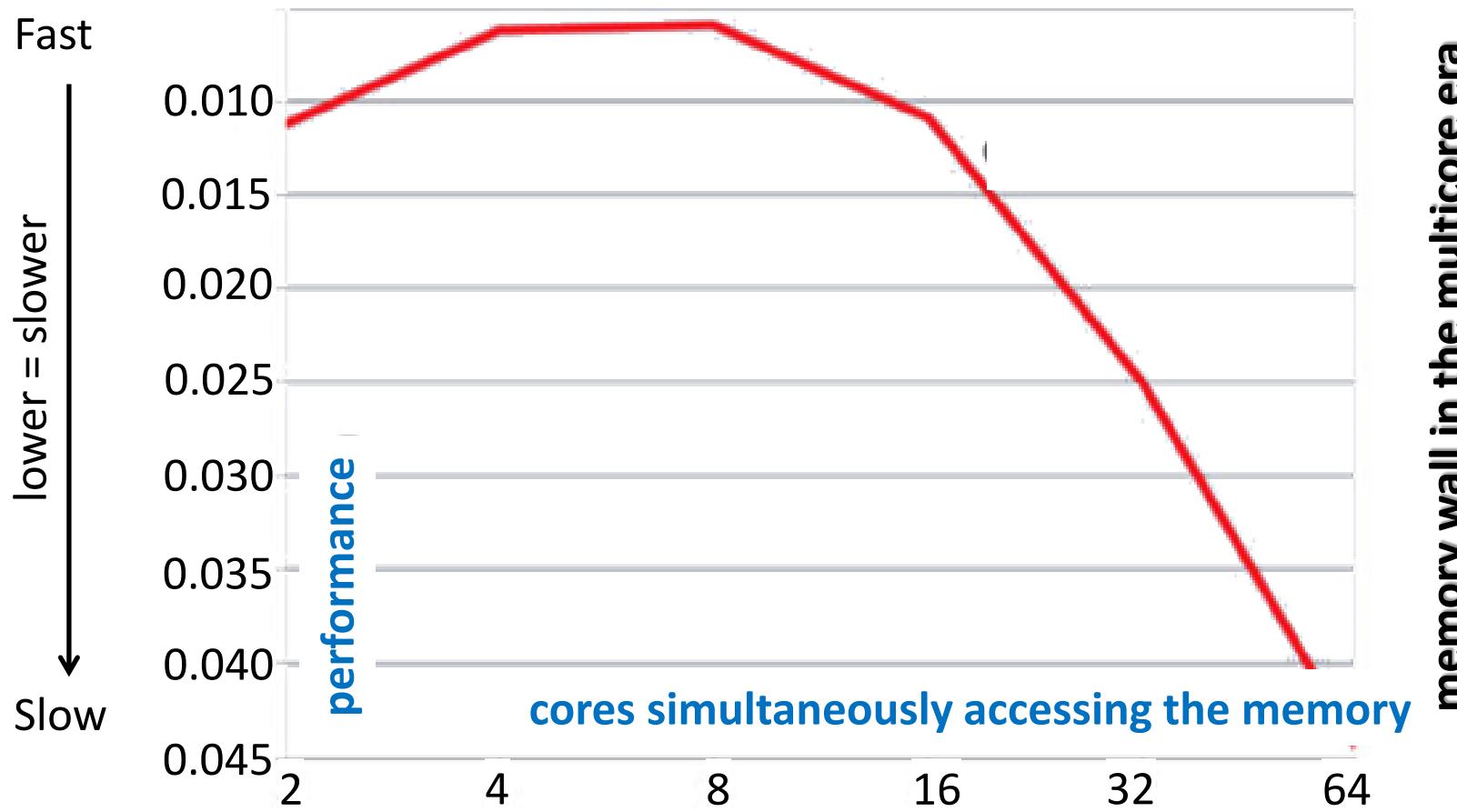
# Problem: CPU faster than main memory

- In the past this problem was called “the memory wall”
- Nowadays, speed of core & memory hardly grows



# Problem: CPU faster than main memory

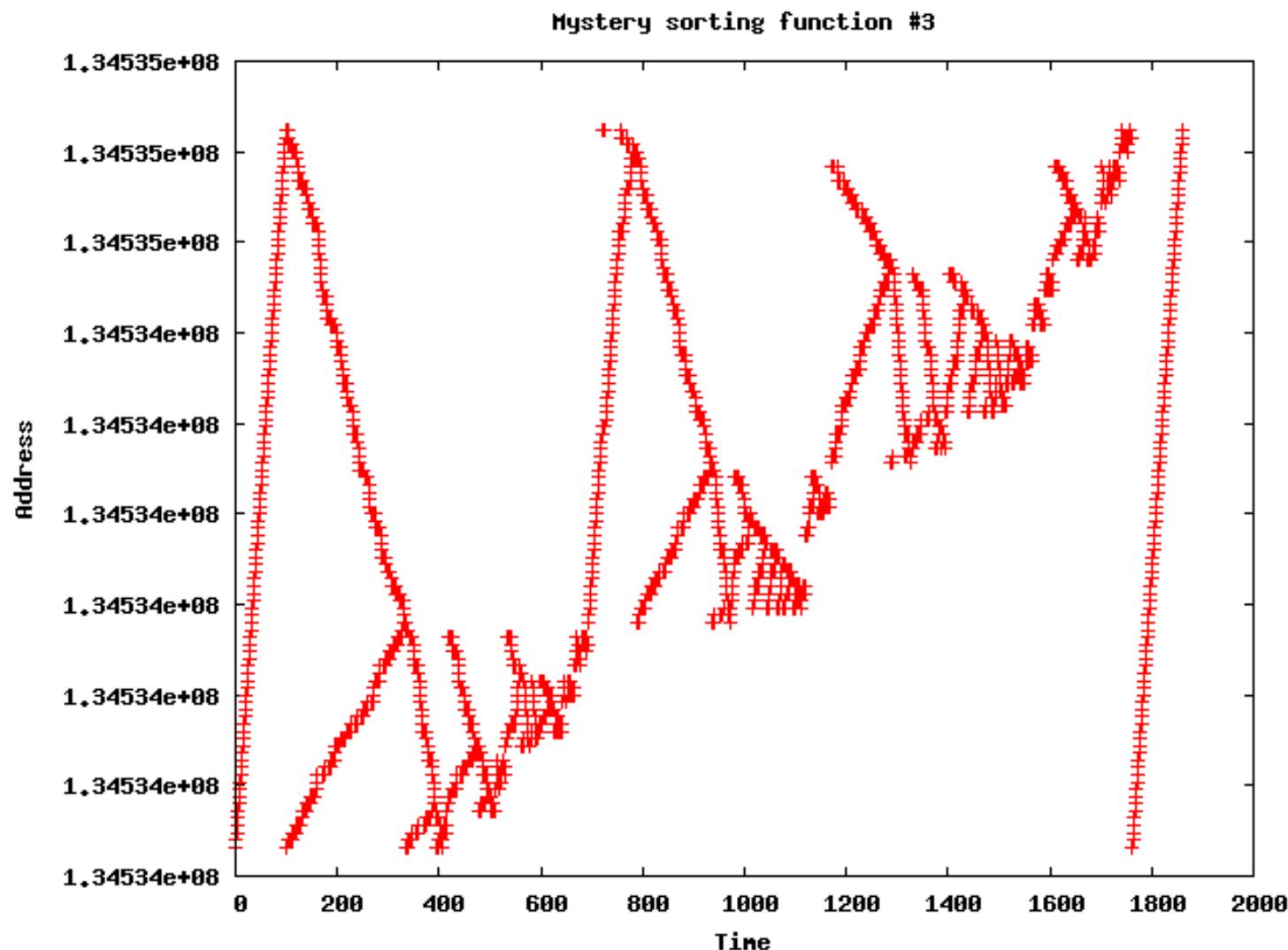
More recently: CPUs aren't getting that much faster, but memory bandwidth might become an issue if more cores simultaneously access the memory...



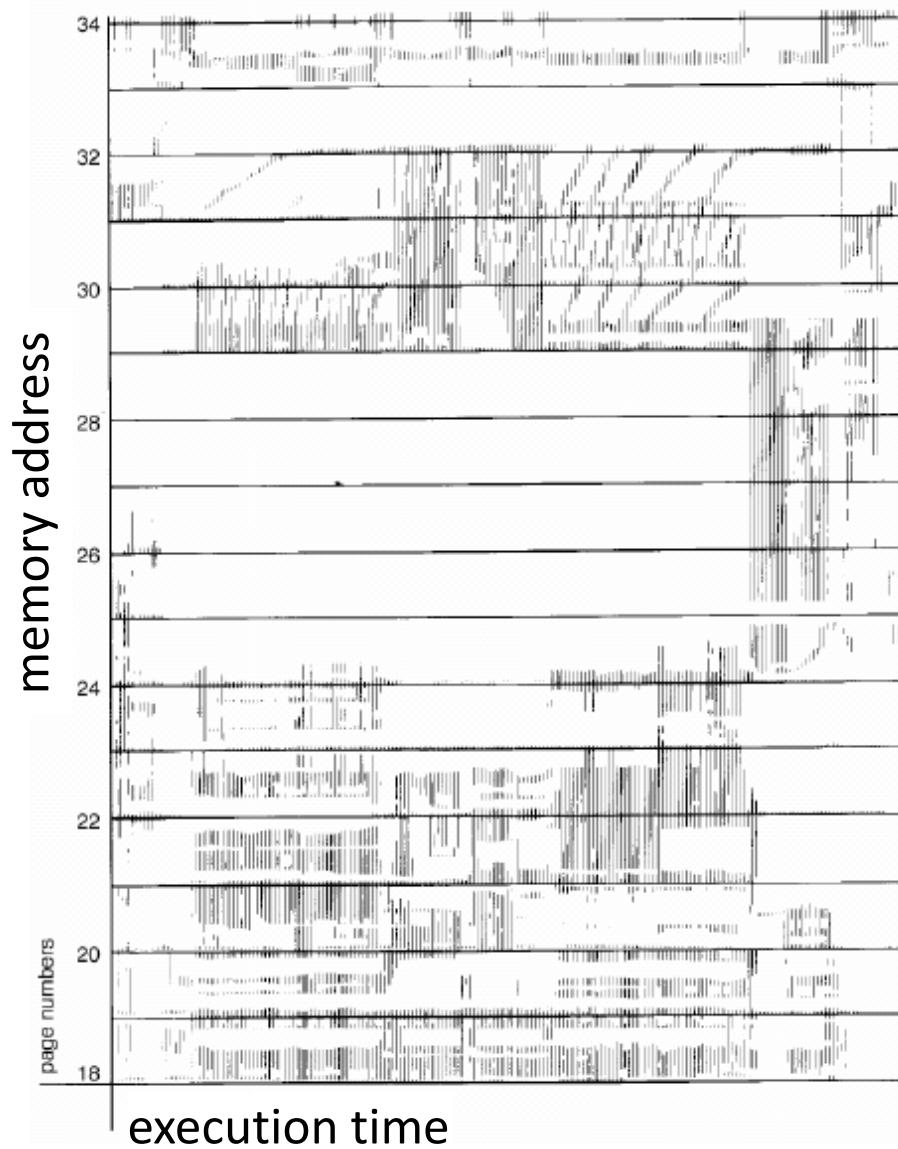
# Empirical observation

- **Principle of Locality (= Locality of Reference)**
  - A phenomenon commonly observed while computer programs run:
  - The collection of the memory locations that are referenced in a short period of time by a running program often consists of relatively well predictable, small clusters of locations
- **Two important special cases:**
  - Temporal locality
    - If at one point in time a particular memory location is referenced, then it is likely that the same location will be referenced again soon
  - Spatial locality
    - If a particular memory location is referenced at a particular time, then it is likely that nearby memory locations will be referenced soon
    - (With these definitions, temporal locality is said to be a special case of spatial locality)

# Locality of reference: example



# Locality of reference: example

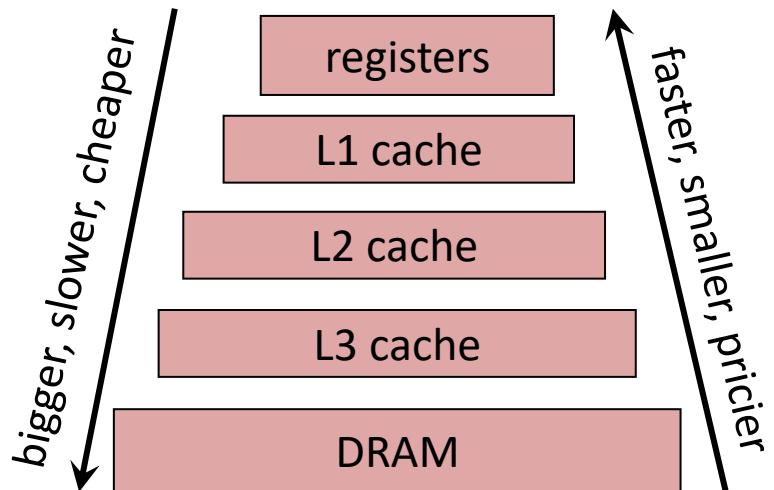


# Solution: caching

- **Cache is**
  - Smaller-but-faster hardware memory structure (faster than main memory)
  - Used to hold a few recently used DRAM locations
  - The act of holding said recently used locations (in the cache structure) is called “caching”
- **How it works**
  - Handled by hardware, automatically
  - That is, the cache is being filled by the hardware, on the fly
  - While the CPU generates memory accesses (read and write ops), the hardware **MMU** (memory management unit) arranges things such that the accessed locations are being cached
  - When cache space runs out, the hardware must also choose which locations to evict from the cache
- **Why it works**
  - Because of the principle of locality

# Cache hierarchy: Intel i7-4770 (Haswell, Q2'13)

- **4 cores; core speed**
  - 3.4 GHz (Turbo Boost off)
  - 3.9 GHz (max on)
- **Memory hierarchy**
  - 32 KB L1d + 32 KB L1i =  
64 KB L1 cache (per core)
    - Latency = 4–5 cycles  $\approx$  1.2–1.5 nanoseconds
  - 256 KB L2 (per core)
    - Latency = 12 cycles  $\approx$  3.5 nanoseconds
  - 8 MB L3 (shared)
    - Latency = 36 cycles  $\approx$  11 nanoseconds
  - Up to 32 GB DRAM (shared)
    - Latency = 230 cycles  $\approx$  68 nanoseconds



# **LET'S RETURN TO THE COST OF CONTEXT SWITCH**

# The indirect context switch component

- **Includes**
  - The time it takes for to (re)populate the caches with useful content after the context switch occurs – aka “warm up the caches”
  - There are other such HW components
- **Threads vs. processes**
  - If threads actually do work on same data / utilize same instructions (not always the case), then the indirect overhead of context switching could be smaller
  - So switching between threads could be cheaper
  - (As we will see in a future lecture, this is why the 2.4 Linux scheduler gives a “bonus” to tasks sharing the same address space)
- **Direct vs. indirect**
  - Experimental data shows that indirect component of context switching *might* reach up to as high as two orders of magnitude more than the direct component (obviously, depending on what's running)

Lecture ended here

**FYI-S**

# Processes can share (some) memory too

- **Via system calls**
  1. `shmget(key, sizem, attributes)` // get
  2. `shmat(key, address, attributes)` // attach
  3. `shmdt(address)` // detach
  4. `shmctl(key, command, struct shmid_ds *buf)` // control
  5. `shm_open` // as file
  6. `shm_unlink` // as file
- **Different (virtual) memory addresses that refer to the same (physical) memory location**
- **Homework**
  - Try it: share memory between two processes

# The copy-on-write optimization

- **The fork() system call creates a copy of the address space of the parent**
  - But it only creates a *logical* copy
  - There is no physical copy until we really need to have one
  - Which is when either child or parent write
- **Even then, copying is not of the entire address space**
  - OS copies only the “page” of the target memory location (typically 4KB)
  - One page at a time
  - More on that in lectures to follow

# Terminology

- **Multitasking**
  - Having multiple processes **time** slice on the same core
- **Multiprogramming**
  - Having **multiple jobs** in the system (either on the same core or on different cores)
- **Multiprocessing**
  - Using **multiple processors for the same job** in parallel
- **IPC (= inter-process communication) mechanisms**
  - Message passing (via pipes for example)
  - Shared memory
  - Signals

# **USER-LEVEL THREADS**

# Threads can be implemented in user-level

- **Motivation**
  - Sometimes programmers have domain-specific knowledge that allows them to implement multithreading in a more efficient manner
    - Creating a stack for a function in user-level can be done in only a few **10s of cycles** (as opposed to kernel-level, which usually take at least **1000s of cycles**)
  - Typically (not always), this is done for “runtimes” that present to users some programming model that makes it easier for them to exploit parallelism
  - Typically (not always), such runtimes allow tasks to run to completion once they start to run

# Example – OmpSs [https://pm.bsc.es/ompss]

```
#pragma omp task inout( arr[lo:hi] )
void sort ( int lo, int hi, int *arr ) {
    if ( hi-lo < THREASHOLD )
        sequential_sort ( lo, hi, arr) ;
    else { // in parallel, while tracking dependencies...
        int mid = (lo+hi) / 2;
        sort( lo, mid, arr );           // this task is done in parallel
        sort ( mid+1, hi, arr);       // with this task
        merge (lo,mid,hi, arr);      // while this task waits
    }
}
```

- **A function can be declared a task**
  - Which means it is run in parallel with other instances of that function so long as there are no input/output dependencies
- **The OmpSs runtime automatically tracks tasks' input/output dependencies**
  - It runs a task only after its inputs are ready
  - E.g., ‘merge’ (also a task) must wait for the two ‘sort’s that come before it

# Example – OmpSs [<https://pm.bsc.es/ompss>]

- **The OmpSs runtime maintains one OS-thread per core**
  - On each core it does its own scheduling of OmpSs “tasks”
  - The kernel is not aware of this at all
- **Which means tasks are not allowed to make system calls**
  - That is, they are allowed, but then the core will stand idle (as far as the OmpSs runtime is concerned), because the kernel is not aware of the fact that there are other waiting tasks
  - So users are instructed not to invoke system calls in OmpSs tasks

# setjmp & longjmp

- **Standard C library functions**
  - Which allow programmers to do their own user-level scheduling
  - Again, OS remains unaware (any blocking syscall will block all of them)
- **The gist of it**
  - ```
switch() {
    if( setjmp( g_buf[g_current] ) == 0 )
        schedule();      // run another thread...
    else
        // and we're back...
}
– schedule() {
    new_t = select_next_thread_to_run();
    g_current = new_t;
    longjmp( g_buf[g_current] ); // back to corresponding else
}
```
- **Homework: read page 32 in Feitelson's OS notes**
- **Also <https://www.cs.purdue.edu/homes/cs240/lectures/Lecture-19.pdf>**

# **Operating Systems (234123)**

## *Scheduling*

Dan Tsafrir (2024-06-23)

# **BATCH (NON-PREEMPTIVE) SCHEDULERS**

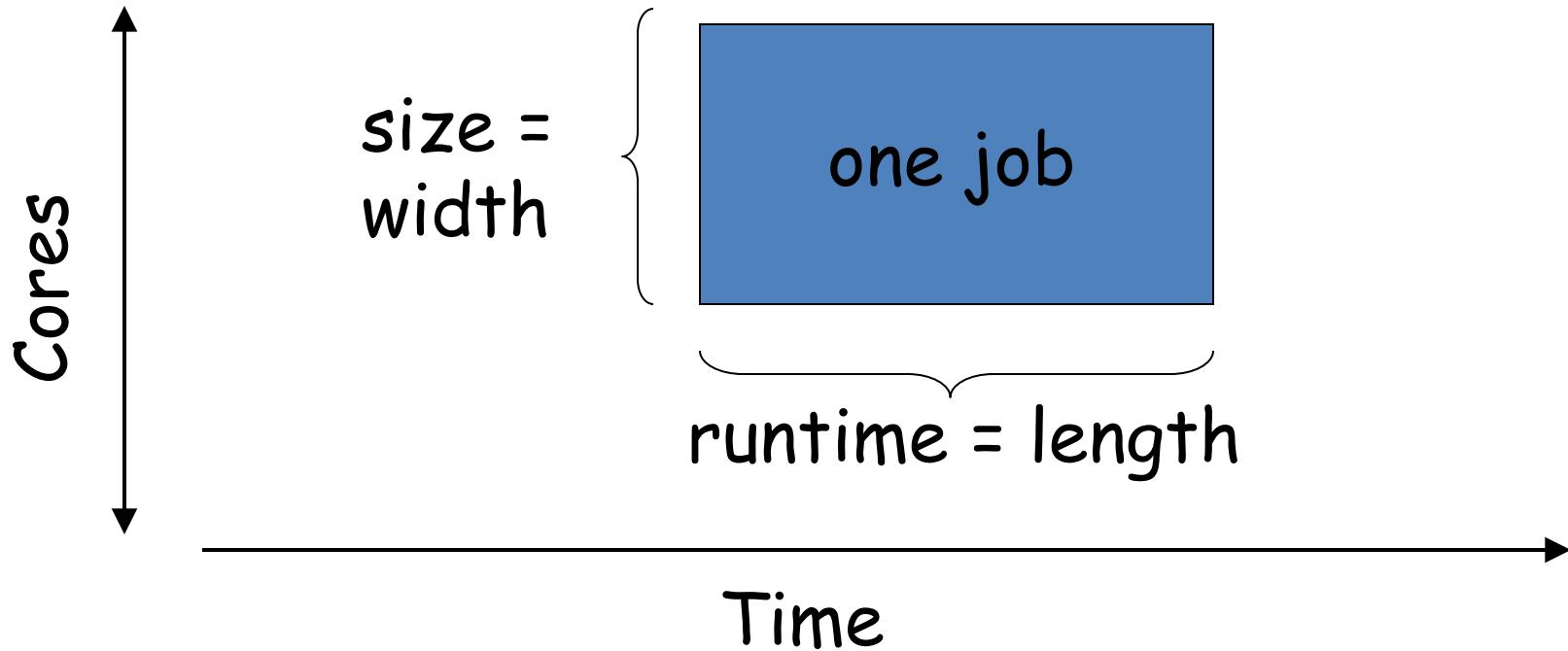
# **Consider a different context...**

- **In this course, we typically consider general purpose OS**
  - Such as the one running in our own computers
- **For some of the topics in this lecture to make sense, need to change the mind set**
  - (Although some of them nevertheless apply to general purpose OSes)
- **The new context**
  - “Batch scheduling” on “supercomputers”
  - We will explain both of these terms next

# Supercomputers

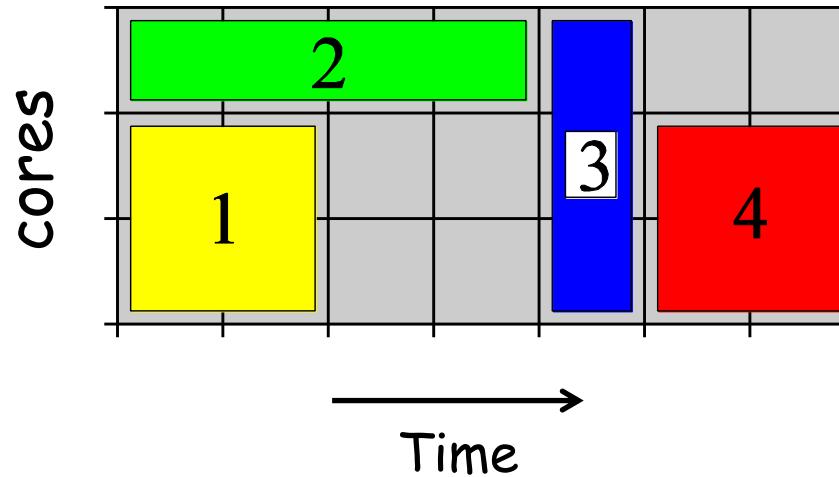
- **Comprised of hundreds (1990s) to millions (2020s) of CPU cores**
  - “Nodes” (=computers) are connected via a high-speed network
- **Used by 100s to 1000s of scientific users**
  - Physicists, chemists...
- **Users submit “jobs” (=programs)**
  - Jobs can be **serial** (one core) or **parallel** (many cores)
  - A parallel job simultaneously uses N cores to solve a **single scientific problem quickly** (ideally N times faster)
  - N is the “size” of the job (determined by the submitting user)
    - Different jobs have different sizes
    - For a given running job, size is fixed throughout its lifetime
  - While the program is running, **threads/processes communicate** with each other and exchange information
  - Typical workload: jobs running from a few seconds to 10s–100s of hours

# Think of jobs as rectangles in core X time plane



- **Terminology:**
  - Size: jobs are said to be “wide” = “big” or “narrow” = “small”
  - Runtime: jobs are said to be “short” or “long”

# A FCFS schedule example (FCFS = First Come First Serve)



(Numbers indicate arrival order)

# Batch scheduling

- It's important that all the processes of a job run **together** (as they repeatedly & frequently communicate)
  - Each process runs on a different core
  - What will happen if they don't run together?
- Jobs are sometimes tailored to use much of (if not the entire) physical **memory** of each individual multicore CPU
  - So can't easily share cores with other jobs via multiprogramming
- Thus, supercomputers typically use “**batch scheduling**”
  - When a job is scheduled to run, it gets its own cores
  - Cores are dedicated (not shared)
  - Each job runs to completion (until it terminates)
  - Only after, the cores are allocated to other waiting jobs
  - Such scheduling is said to be “non-preemptive”

Wait time, response time, slowdown, utilization, throughput

## **METRICS TO EVALUATE PERFORMANCE OF BATCH SCHEDULERS**

# Average wait time & response time

- **Average wait time**
  - The “wait time” of a job is the interval between the time the job is submitted to the time the job starts to run
    - $\text{waitTime} = \text{startTime} - \text{submitTime}$
  - The shorter the average wait time, the better the performance
- **Average response time**
  - The “response time” of a job is the interval between the time the job is submitted to the time the job terminated
    - $\text{responseTime} = \text{terminationTime} - \text{submitTime}$
    - $\text{responseTime} = \text{waitTime} + \text{runTime}$
  - The shorter the average response time, the better the performance
- **Wait vs. response**
  - Users typically care most about response time (wait for their job to end)
  - But batch schedulers primarily only affect wait time

# Avg. wait vs. response time – the connection

- **Claim**

- In our context, we assume that job runtimes (and thus their average) are a given; they stay the same regardless of the scheduler
- Therefore, for batch schedulers, the difference between average wait time and average response time of a given schedule is a constant
- The constant is the average runtime of all jobs

- **Proof**

- For each job  $i$  ( $i = 1, 2, 3, \dots, N$ )
  - Let  $W_i$  be the wait time of job  $i$
  - Let  $R_i$  be the runtime of job  $i$
  - Let  $T_i$  be the response time of job  $i$  ( $T_i = W_i + R_i$ )
- With this notation we have

$$\frac{1}{N} \sum_{i=1}^N T_i = \frac{1}{N} \sum_{i=1}^N (W_i + R_i) = \frac{1}{N} \sum_{i=1}^N W_i + \frac{1}{N} \sum_{i=1}^N R_i$$

*avg response*                                   *avg wait*                           *avg runtime*

# Average slowdown (aka “expansion factor”)

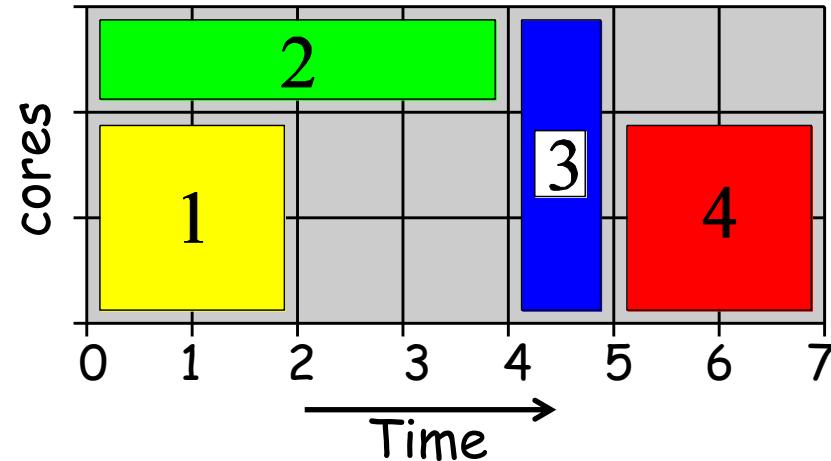
- **The “slowdown” (or “expansion factor”) of a job is**
  - The ratio between its response time & its runtime
  - $\text{slowdown} = \text{responseTime} / \text{runTime} = T_i/R_i$
  - $\text{slowdown} = (\text{waitTime} + \text{runTime}) / \text{runTime} = (W_i+R_i) / R_i = W_i/R_i + 1$
- **Examples**
  - If  $R_i = 1$  minute, and  $W_i = 1$  minute
    - Job has a slowdown of  $(W_i+R_i)/R_i = (1+1)/1 = 2$
    - Namely, job was slowed by a factor of 2 relative to its runtime
  - If  $R_i = 1$  hour, and  $W_i = 1$  minute, then the slowdown is much smaller
    - $(W_i+R_i)/R_i = (60+1)/60 \approx 1.02$
    - The delay was insignificant relative to the job’s runtime
- **Like wait & response, we aspire to minimize avg. slowdown**
  - $\text{slowdown} = 1 \Leftrightarrow$  job was immediately scheduled
  - The greater the slowdown, the longer the job is delayed

# Utilization & throughput

- **Utilization (aspire to maximize)**

- Percentage of time the resource (CPU in our case) is busy
  - In this example, the utilization is:

$$100 \times \frac{(3 \times 7 - 6)}{3 \times 7} = 71\%$$



- **Throughput (aspire to maximize)**

- How much work is done in one time unit
  - Examples
    - Typical **hard disk** throughput: 100 MB/second (sequential access)
    - **Database** server throughput: transactions per second
    - **Web server** throughput: requests per second
    - **Supercomputer** throughput: job completions per second
      - In above example:  $4(\text{jobs})/7(\text{time units}) \approx 0.57 \text{ jobs per time unit}$

# Which metric is more important?

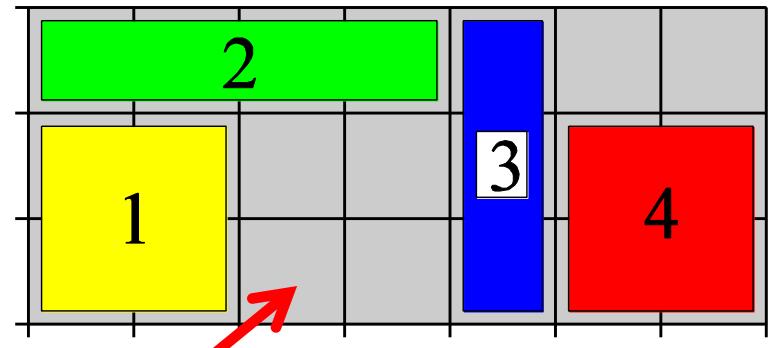
- **Depends on your point of view**
  - Users typically care about wait/response time and slowdown
  - System owners typically care more about utilization & throughput
- **For users: wait time or slowdown?**
  - Different notion of “fairness”
    - Wait time => FCFS, which humans typically consider as fairest
    - Slowdown depends on your runtime (a.k.a. service time) – “ רק שאלה ”
    - What’s fairer? No definite answer – system owners decide
    - Performance analysts typically keep an eye on (=evaluate) both
  - Notice that
    - Average wait time tends to be dominated by long jobs
    - Average slowdown tends to be dominated by short jobs
    - Why? Slowdown: easy. Wait time: try to answer after next few slides

FCFS, EASY, backfilling, RR, SJF

## BATCH SCHEDULING EXAMPLES

# FCFS (First-Come First-Served) scheduling

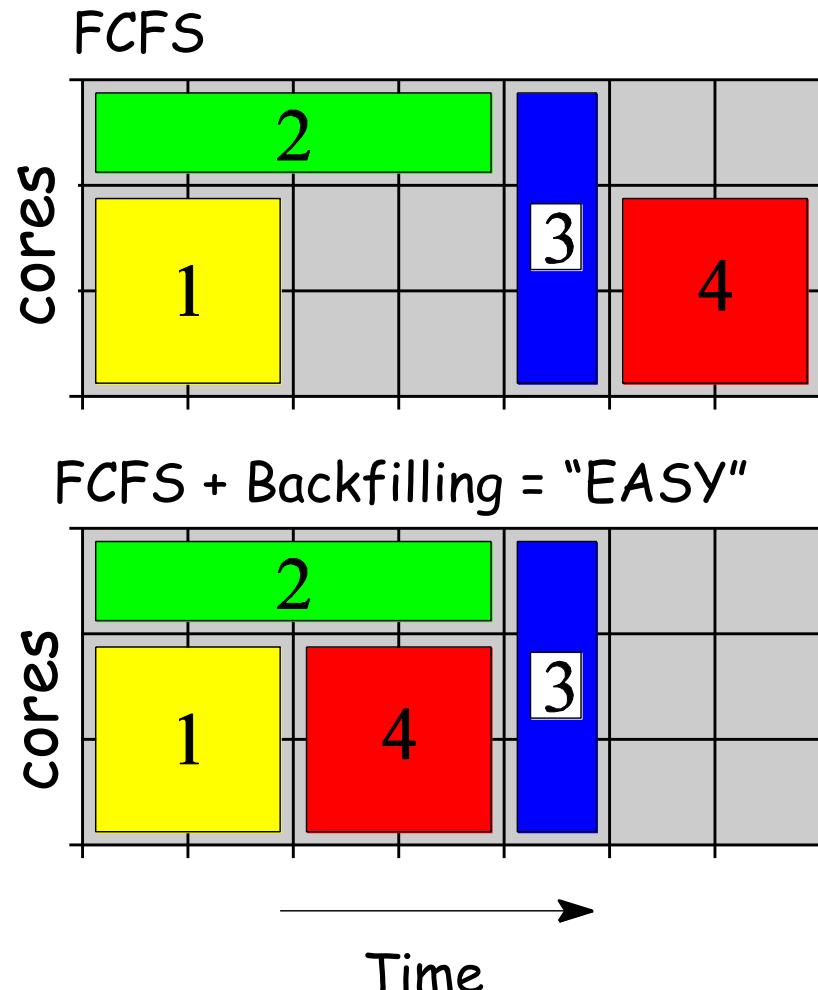
- **Jobs are scheduled by their arrival time**
  - If there are enough free cores, a newly arriving job starts to run immediately
  - Otherwise, it waits, sorted by arrival time, until enough cores are freed
- **Pros:**
  - Easy to implement (FIFO wait queue)
  - Typically perceived as most fair (we tend to dislike “אני רק שאלת”)
- **Cons:**
  - Creates fragmentation (=unutilized cores)
  - Small/short jobs might wait for a long, long while.. (”אני רק שאלת”)



(Numbers indicate arrival order)

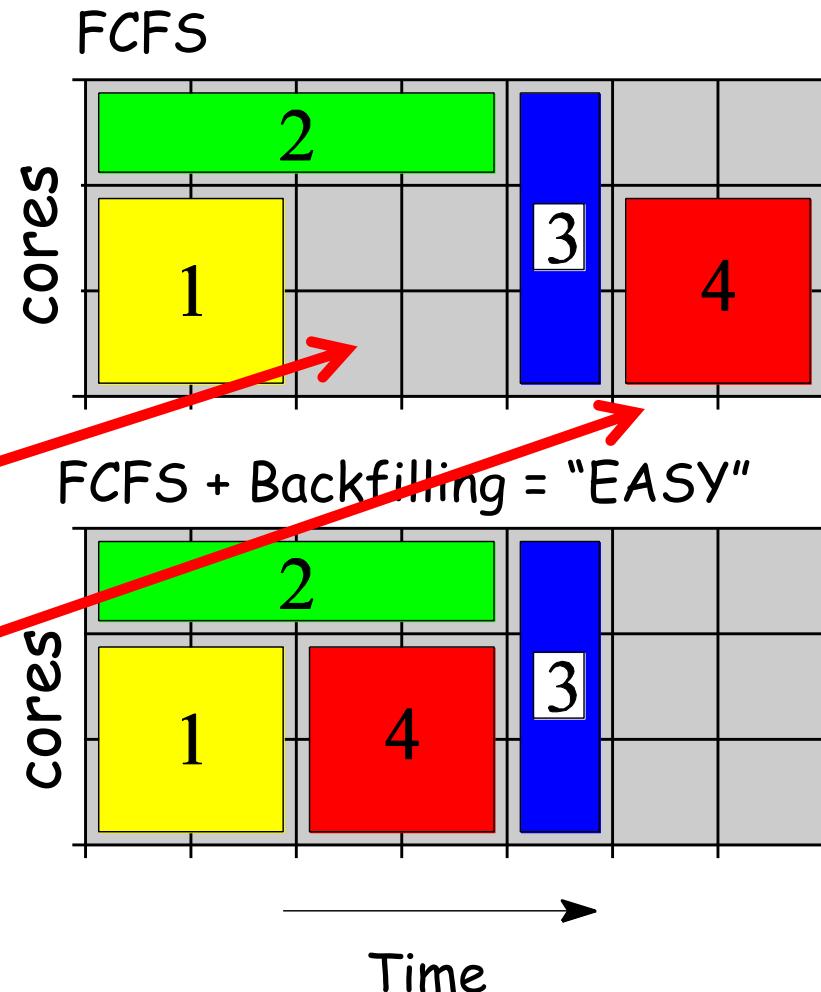
# EASY (= FCFS + backfilling) scheduling

- The “backfilling” optimization
  - A short waiting job can jump over the head of the wait queue
  - Provided it doesn’t delay the job @ head of the FCFS wait queue
- **EASY algorithm:** whenever a job arrives (=submitted) or terminates:
  1. Try to start the job @ head of the FCFS wait queue
  2. Then, iterate over the rest of the waiting jobs (in FCFS order) and try to backfill them



# EASY (= FCFS + backfilling) scheduling

- **Pros**
  - Better utilization (less fragmentation)
  - Narrow/short jobs have better chance to run sooner
- **Con:**
  - Must know runtimes in advance
    - To know width of holes
    - To know if backfill candidates are short enough to fit holes



# **EASY (= FCFS + backfilling) scheduling**

- Backfilling mandates users to **estimate how long their job will run**
  - Upon job submission
- If a job tries to **overrun its estimate?**
  - It is killed by the system
  - Provides **incentive** to supply accurate estimates
    - Short estimate => better chance to backfill
    - Too short => jobs will be killed
- **EASY (and FCFS) are popular**
  - Many supercomputer schedulers use them by default
- **BTW, EASY stands for**
  - Extensible Argonne Scheduling sYstem
  - Developed @ Argonne National Laboratory (USA) circa 1995

# SJF (Shortest-Job First) scheduling

- **Instead of**
  - Ordering jobs (or processes) by their arrival time (FCFS)
- **Order them by**
  - Their (typically estimated) runtime
- **Perceived as unfair**
  - At the extreme: causes starvation (whereby a job waits forever)
- **But optimal (in a sense) for performance**
  - As we will see later, using some math
- **NOTE: job-scheduling theoretical reasoning is limited (called: queuing theory)**
  - Hard (impossible?) to do it for arbitrary parallel workloads
  - Therefore, for theoretical reasoning
    - We'll assume jobs are serial (job=process)
  - Empirically, intuition for serial jobs often also applies to parallel jobs

# Average wait time example: FCFS vs. SJF

- **Assume**

- Processes P1, P2, P3 arrive together in the very same second
- Assume their runtimes are 24, 3, and 3, respectively
- Assume FCFS orders them by their index: P1, P2, P3
- Whereas SJF orders them by their runtime: P2, P3, P1

- **Then**

- The average wait time under FCFS is

- $(0+24+27)/3 = 17$



- The average wait time under SJF is

- $(0+3+6)/3 = 3$



- **SJF seems “better” than FCFS**

- In terms of optimizing the average wait time metric

# “Convoy effect”

- **Slowing down all (possibly short) processes due to currently servicing a very long process**
  - As we've seen in the previous slide
- **Does EASY suffer from convoy effect?**
  - Yes (why?), but less than FCFS:
  - Empirically, when examining real workloads (recordings of the activity of real supercomputers), we find that there are often holes in which short/narrow jobs can fit, such that many of them manage to start immediately shortly after they arrive
- **Does SJF suffer from convoy effect?**
  - Yes (why?), but less than FCFS (why?)
- **Q: Can we eliminate convoy effect altogether?**

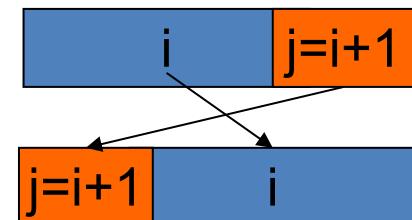
# Optimality of SJF for average wait time

- **Claim**

- Given: a 1-core system where all jobs are serial
- If: (a) all processes arrive together and (b) their runtimes are known
- Then: the average wait time of SJF is equal to or smaller than the average wait time of any other batch scheduling order S

- **Proof outline**

- Assume the scheduling order S is:  $P(1), P(2), \dots, P(n)$
- If S isn't SJF, then there exist two processes  $P(i), P(j=i+1)$  such that  $R(i) = P(i).\text{runtime} > P(i+1).\text{runtime} = R(j=i+1)$
- If we swap the scheduling order of  $P(i)$  and  $P(i+1)$  under S, then we've increased the wait time of  $P(i)$  by  $R(i+1)$ , we've decreased the wait time of  $P(i+1)$  by  $R(i)$  and this sums up *all* the changes that we've introduced
- And since  $R(i) > R(i+1)$ , the overall average is reduced
- We do the above repeatedly until we reach SJF



# Fairer variants of SJF

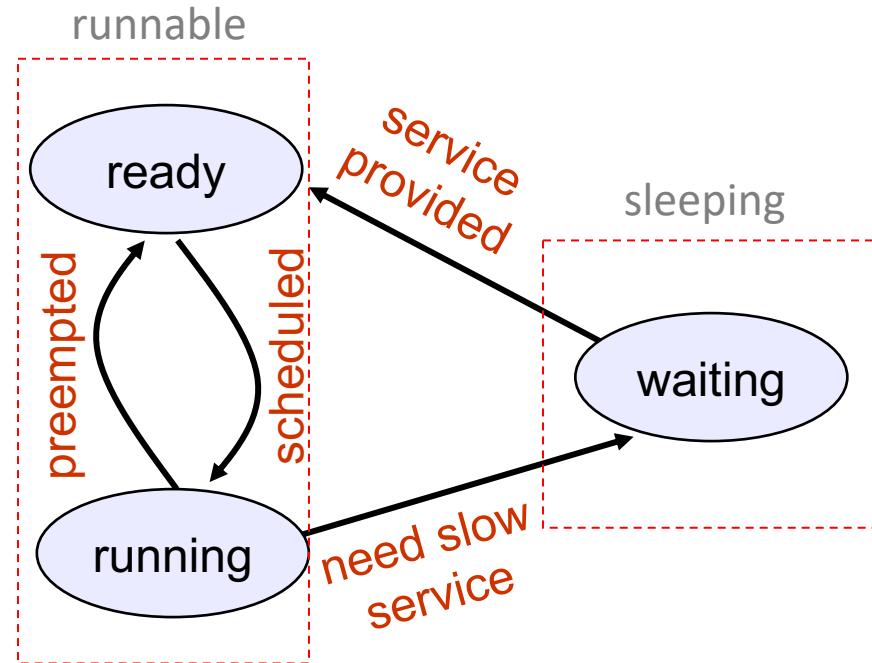
- **Motivation: disallow job starvation**
- **SJBF (Shortest-job *Backfilled* First)**
  - Exactly like EASY in terms of servicing the head of the wait queue in FCFS order (and not allowing anyone to delay it)
  - But the *backfilling* traversal is done in SJF order
- **LXF (Largest eXpansion Factor)**
  - Recall that the “slowdown” or “expansion factor” metric for a job is defined to be:
    - $\text{slowdown} = (\text{waitTime} + \text{runtime}) / \text{runtime}$
  - LXF is similar to EASY, but instead of ordering the wait queue in FCFS, it orders jobs based on their current slowdown (greater slowdown means higher priority)
  - The backfilling activity is done relative the job with the largest current slowdown (= the head of the LXF wait queue)
  - Note that every scheduling decision (when jobs arrive/finish) requires a re-computation of slowdowns (because wait time has changed)

RR, selfish RR, negative feedback, multi-level priority queue

# **PREEMPTIVE SCHEDULERS**

# Reminder

- In a previous lecture, we
  - Talked about three processes states
- In this lecture, we often (e.g., when we want to prove things)
  - Focus on only two
    - Ready & running
- Namely, we'll assume that
  - Processes only consume CPU
  - And don't do I/O (of course, actually, they do, but we ignore that now)
  - They're either running or ready-to-run



# Preemption

- **The act of suspending one job (process) in favor of another**
  - Even though it is not finished yet
- **Exercise**
  - Assume a one-core system
  - Assume two processes, each requiring 10 hours of CPU time
  - Does it make sense to do preemption (say, every few milliseconds)?
- **When would we want a scheduler to be preemptive?**
  1. When responsiveness to user matters (they actively wait for the output of the program), and
  2. When runtimes vary (some jobs are shorter than others)
- **Examples**
  - Two processes, one needs 10 hours of CPU time, the other needs 10 seconds (and a user is actively waiting for it to complete)
  - Two processes, one needs 10 hours of CPU time, the other is a word processor (like MS Word or Emacs) that has just been awakened because the user clicked on a keyboard key

# Quantum

- **The maximal amount of time a process is allowed to run before it is preempted**
  - Typically, milliseconds to 10s to milliseconds in general-purpose OSes (like Linux or Windows)
- **Quantum is oftentimes set per-process**
  - Processes that behave differently get different quanta, e.g., in [Solaris](#):
    - A CPU-bound process gets long quanta but with low priority
    - Whereas an I/O-bound process gets short quanta with high priority
  - In Linux, the process “[nice](#)” value affects the quantum duration
    - “Nice” is a system call and a shell utility which allow one process to be “nicer” to the other processes in terms of scheduling (see man)
    - We’ll see this later

# Performance metrics for preemptive schedulers

- **Wait time (try to minimize)**
  - Same as in batch (non-preemptive) systems
- **Response time (or “turnaround time”; try to minimize)**
  - Like batch systems, stands for
    - Time from process submission to process completion
  - But unlike batch systems, instead of
    - $\text{responseTime} = \text{waitTime} + \text{runTime}$
  - We have
    - $\text{responseTime} \geq \text{waitTime} + \text{runTime}$
  - Because
    - Processes can be preempted, plus **context switches** have a price
- **Overhead (try to minimize); consists of**
  - How long a context switch takes, and how often context switches occur
- **Utilization & throughput (try to maximize)**
  - Same as in batch (non-preemptive) systems
  - Although may want to account for context switching overhead

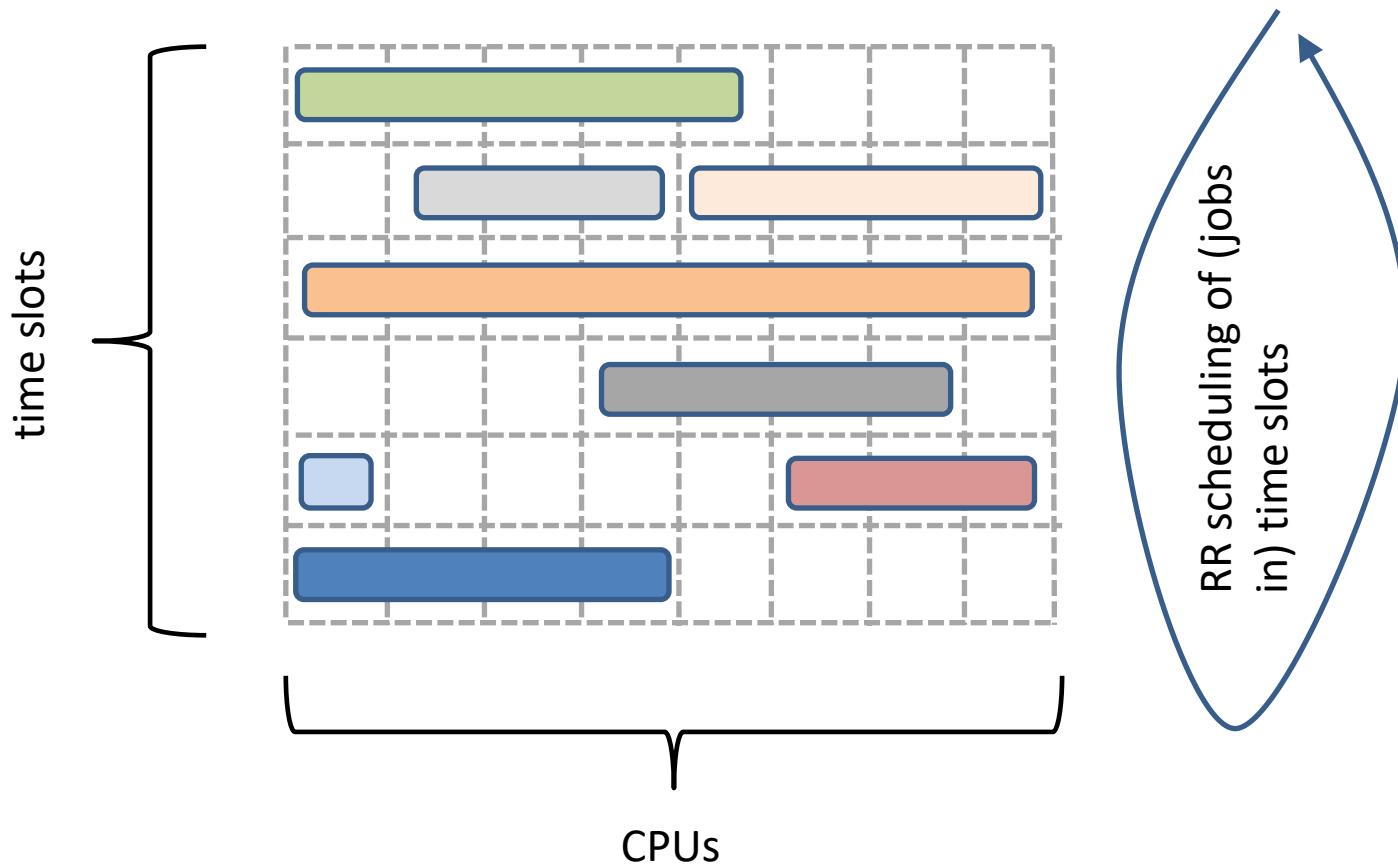
# RR (Round-Robin) scheduling

- **Processes are arranged in a cyclic ready-queue**
  - The head process runs, until its quantum is exhausted
  - The head process is then preempted (suspended)
  - The scheduler resumes the next process in the circular list
  - When we've cycled through all processes in the run-list (and we reach the head process again), we say that the current “epoch” is over, and the next epoch begins
- **Requires a timer interrupt**
  - Typically, it's a periodic interrupt (fires every few millisecond)
  - Upon receiving the interrupt, the OS checks if its time to preempt
- **Features**
  - For small enough quantum, it's like everyone of the N processes advances in  $1/N$  of the speed of the core (sometime called “virtual time”)
  - With a huge quantum (infinity), RR becomes FCFS

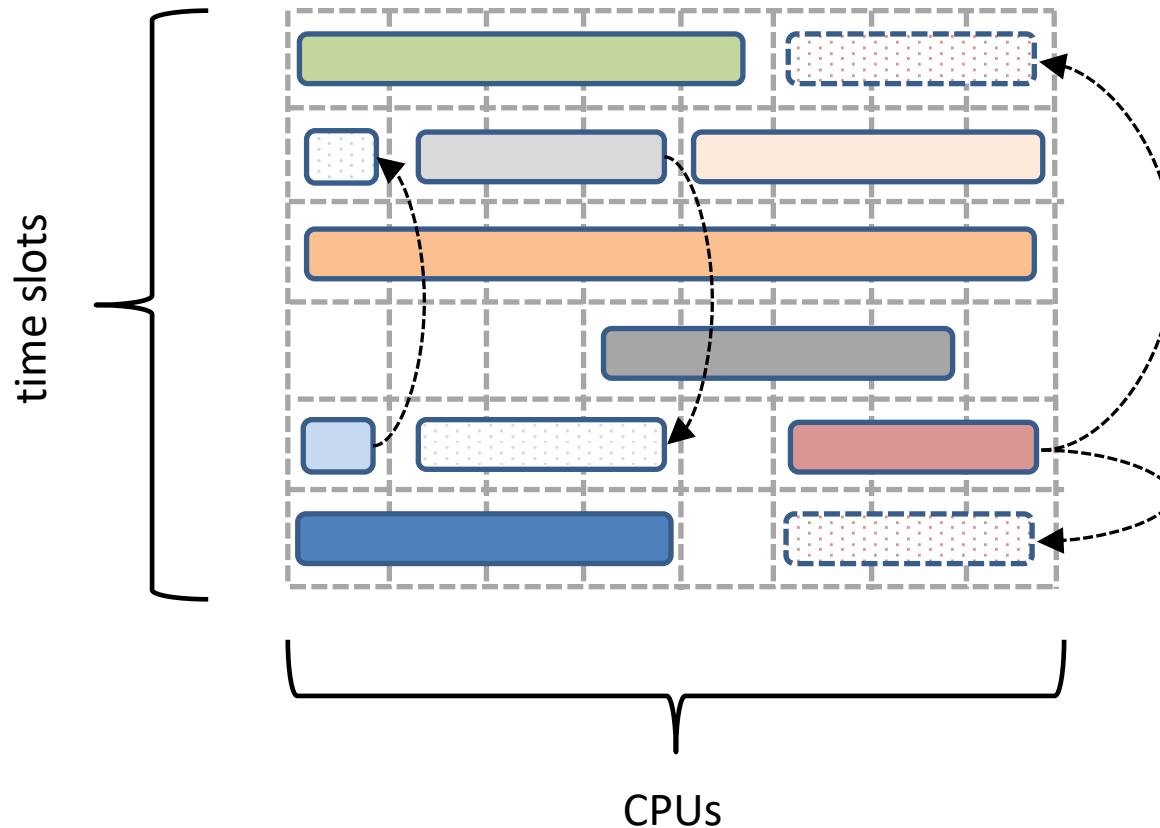
# Is there RR in parallel systems?

- Yes! It's called “gang scheduling”
  - Time is divided to slots (seconds, or minutes)
  - Every job has a “native” time slot
  - Algorithm attempts to fill holes in time slots by assigning to them jobs from other native slots (called “alternative” slots)
    - Challenge: to keep contiguous chunks of free cores
    - Uses a “buddy system” algorithm
  - <http://www.cs.huji.ac.il/~feit/parsched/jsspp96/p-96-6.pdf>
  - Algorithm attempts to minimize slot number using slot unification when possible
  - Why might gang scheduling be useful?
    - Don't need to know runtime in advance (but what about memory?)
  - Supported by most commercial supercomputer schedulers
    - However, seems to be rarely used
    - If lots of memory is “swapped out” upon context switch, context switching will take a very long time... (so it won't be worth it)

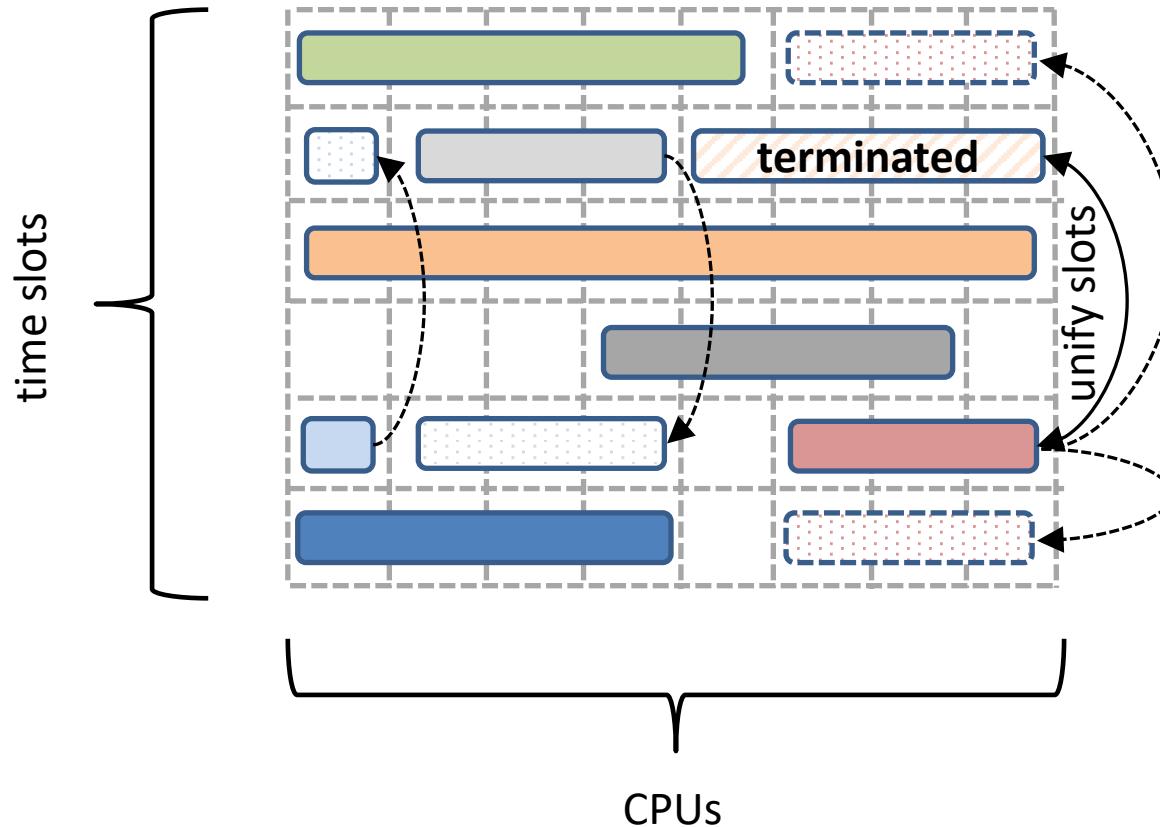
# Gang scheduling - example



# Gang scheduling – alternative slots



# Gang scheduling – native slot unification



# Price of preemption: example

- **Assume**
  - One core, 1 second quantum
  - 10 processes, each requires 100 seconds of CPU time
- **Assuming no context switch overhead (takes zero time)**
  - Then the “makespan” metric (time to completion of all processes) is
    - $10 \times 100 = 1000$  seconds
  - Both for FCFS and for RR
- **Assume context switch takes 1 second**
  - The makespan of FCFS is
    - $10 \text{ (proc)} \times 100 \text{ (sec)} + 9 \text{ (ctx-sw)} \times 1 \text{ (sec)} = 1009$
  - Whereas the makespan of RR is
    - $10 \text{ (proc)} \times 100 \text{ (sec)} + 100 \text{ (proc)} \times 100 \text{ (ctx-sw)} - 1 = 10,999$
- **Shorter quantum**
  - Potentially quicker response/wait time (why?), but higher overhead price

# Batching vs. preemption

- **Claim**

- Let  $\text{avgResp}(X)$  be the average response time under algorithm X
- Assume a single core system, and that all processes arrive together
- Assume X is a preemptive algorithm with context switch price = 0
- Then there exists a non-preemptive algorithm Y such that  
 $\text{avgResp}(Y \text{ /*batch*/}) \leq \text{avgResp}(X \text{ /*preemptive*/})$

- **Proof outline**

1. Let  $P_k$  be the last preempted process to finish computing



2. Compress all of  $P_k$ 's quanta to the “right” (assume time progresses left to right), such that the last quantum remains where it is, and all the rest of the quanta move to the right towards it



( $P_k$ 's response time didn't change, and the response time of the other processes improved or stayed the same)

3. Go to 1 (until no preempted processes are found)

# Aftermath

- **Corollary:**

**Based on (1) the previous slide and (2) the proof about SJF optimality from a few slides ago**

- SJF is also optimal relative to preemptive schedulers (that meet our assumptions)

$$\text{avgResponse(SJF)} \leq \text{avgResponse(batch or preemptive scheduler)}$$

rpt

- **So why do we use preemptive schedulers?**

# Connection between RR and SJF

- **Claim**
  - Assume:
    - 1-core system
    - All processes arrive together (and only use CPU, no I/O)
    - Quantum is identical to all processes + no context switch overhead
  - Then
    - $\text{avgResponse(RR)} \leq 2 \times \text{avgResponse(SJF)}$  // RR up to 2x “slower”
- **Notation / definitions:**
  - Process P1, P2, ..., P\_N
  - $R_i$  = runtime of  $P_i$ ;
  - $W_i$  = wait time of  $P_i$
  - $T_i = W_i + R_i$  = response time of  $P_i$
  - $\text{delay}(i,k) = \text{duration } P_i \text{ delayed } P_k \text{ in RR}$  (how long k waited due to i)
  - $\text{delay}(i,i) = R_i$
  - Note that  $T_k = \sum_{i=1}^N \text{delay}(i,k)$

# Connection between RR and SJF

- **Proof outline**

- For any scheduling algorithm A,  $N * \text{avgResponse}(A)$  is

$$= \sum_{k=1}^N T_k = \sum_{i=1}^N \sum_{j=1}^N \text{delay}_A(i, j)$$

$$= \sum_{i=1}^N R_i + \sum_{1 \leq i < j \leq N} [\text{delay}_A(i, j) + \text{delay}_A(j, i)]$$

- Thus, for A=SJF, we get

$$= \sum_{i=1}^N R_i + \sum_{1 \leq i < j \leq N} \min(R_i, R_j)$$

(assume  $P_i$  is shorter  $\Rightarrow R_i < R_j \Rightarrow \text{delay}(i, j) = R_i$  &  $\text{delay}(j, i) = 0$ )

- But for A=RR, we get

$$= \sum_{i=1}^N R_i + \sum_{1 \leq i < j \leq N} 2 \cdot \min(R_i, R_j)$$

(assume  $P_i$  is shorter, then  $P_i$  delays  $P_j$  by  $R_i$ , and since it's a "perfect" RR,  $P_j$  symmetrically delays  $P_i$  by  $R_i$ )

at most twice

# SRTF (Shortest-Remaining-Time First)

- Assume different jobs may arrive at different times
- SJF is not optimal
  - As it's not preemptive, and
  - A short job might arrive while a very long job is running  
=> recall: convoy effect
- SRTF is just like SJF but
  - Is allowed to use preemption
  - Hence, it's “optimal” (assuming a zero context-switch cost etc.)
- Whenever a new job arrives, or an old job terminates
  - SRTF schedules the job with the shortest remaining time
  - Thereby making an optimal decision

# Selfish RR

- **New processes wait in a FIFO queue**
  - Not yet scheduled
- **Older processes scheduled using RR**
- **New processes are scheduled when**
  1. No ready-to-run “old” processes exist
  2. “Aging” is being applied to new processes (some per-process counter is increased over time); when the counter passes a certain threshold, the “new” process becomes “old” and is transferred to the RR queue
- **Fast aging**
  - Algorithm resembles RR
- **Slow aging**
  - Algorithm resembles FCFS

Back to the context of general-purpose OSes

# **GENERAL-PURPOSE SCHEDULERS: PRIORITY-BASED & PREEMPTIVE**

# Scheduling using priorities

- **Every process is assigned a priority**
  - That reflects how “important” it is in that time instance
  - Can change over time
- **Processes with higher priority are favored**
  - Scheduled before processes with lower priorities
- **The priority concept can also be used for batch scheduler**
  - SJF: priority = runtime (smaller => higher)
  - FCFS: priority = arrival time (earlier => higher)

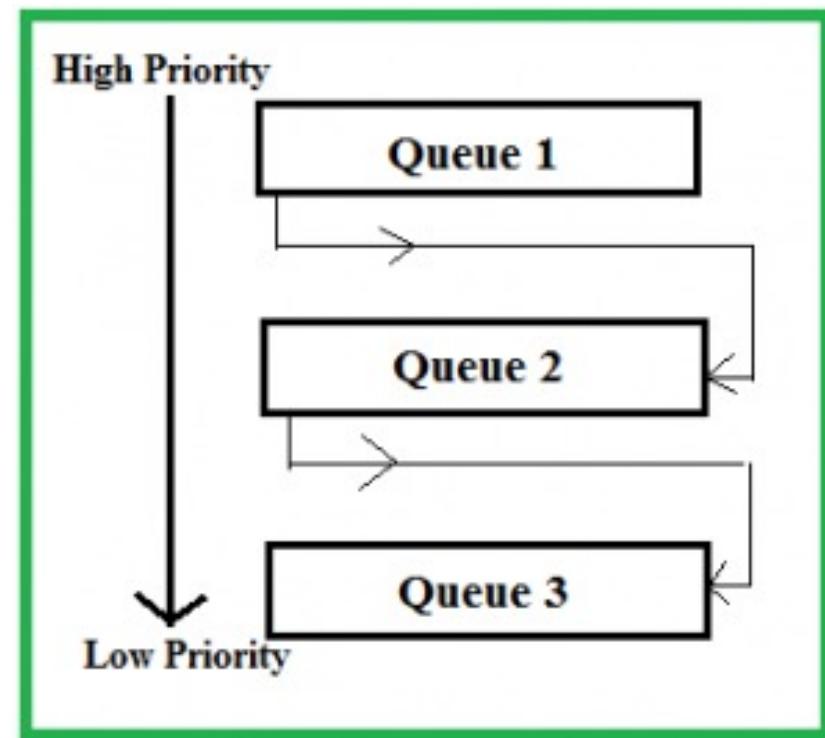
# Negative feedback principle

- The schedulers of all general-purpose OSes (Linux, Windows, ...) employ a negative feedback policy
  - Running reduces priority to run more
  - Not running increases priority to run
- How does this affect I/O-bound & CPU-bound processes?
  - I/O-bound processes (that seldom use the CPU) get higher priority
  - Which is why editors are responsive even if they run in the presence of CPU-bound processes like

```
while(1) {  
    sqrt( time() );  
}
```
- How about other interactive apps like videos with high-frame rate or 3D games?
  - Negative feedback doesn't help them (they consume lots of CPU)
  - Need other ways to identify/prioritize them (nontrivial)

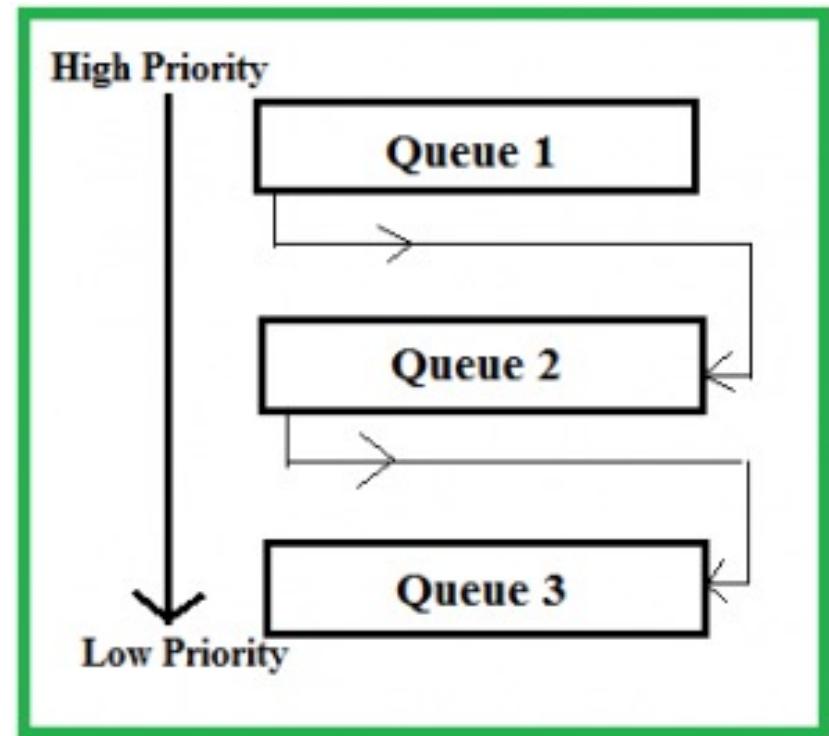
# Multi-level priority queue

- **General-purpose OSes**
  - Typically use some variant of multi-level priority queue
- **Consists of several RR queues**
  - Each associated with a priority
  - Higher-priority queues at the top
  - Lower-priority at the bottom
- **Processes migrate between queues**
  - So they have a dynamic priority
  - “Important” processes move up (e.g., I/O-bound or “interactive”)
  - “Unimportant” move down (e.g., CPU-bound or “non-interactive”)



# Multi-level priority queue

- rpt
- Priority is greatly affected by
    - CPU consumption of processes
      - I/O bound  $\Leftrightarrow$  move up
      - CPU-bound  $\Leftrightarrow$  move down
  - Quanta duration
    - Some schedulers allocate short quanta to higher priority queues
    - Some don't, or even do the opposite

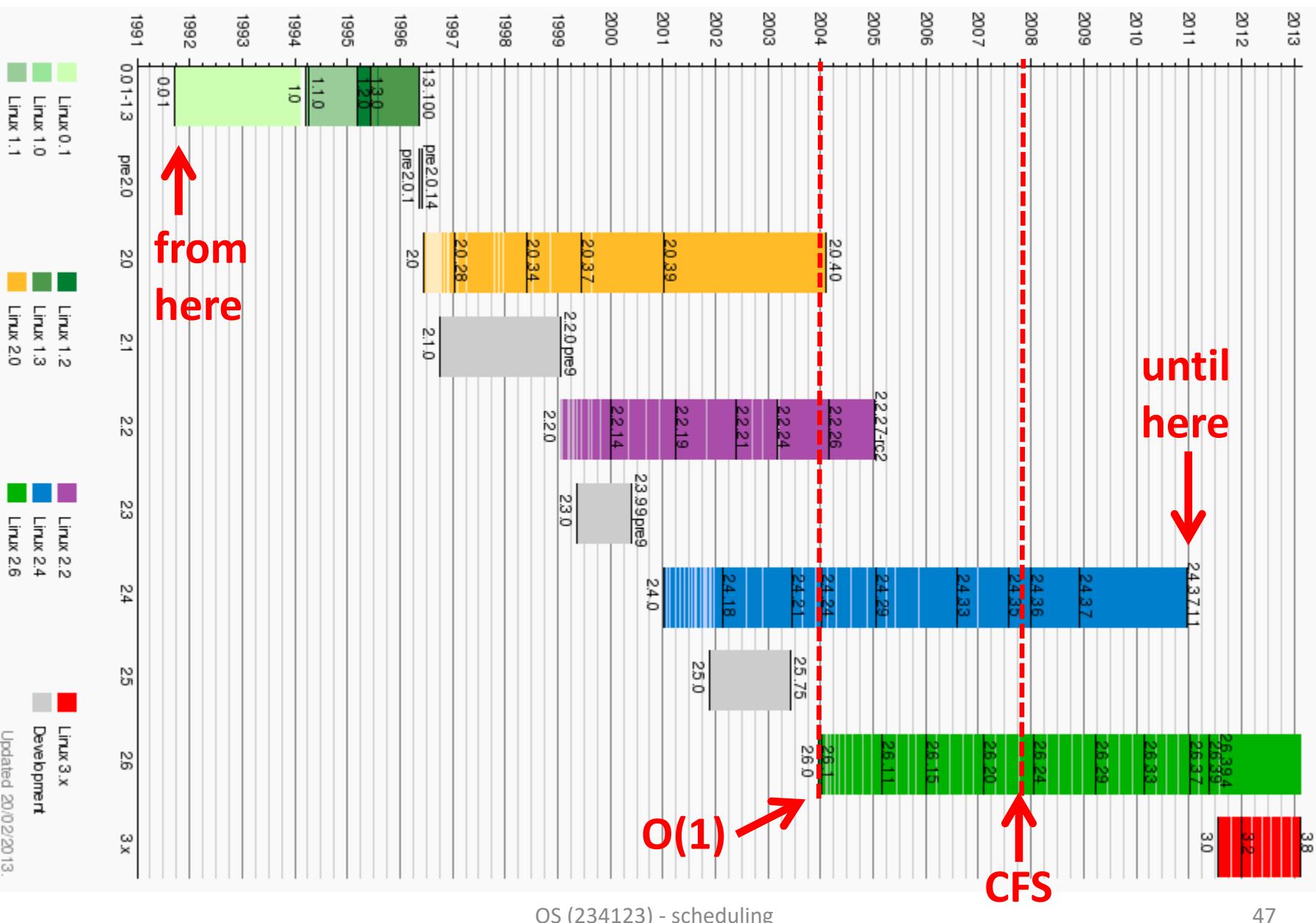


A comprehensive example

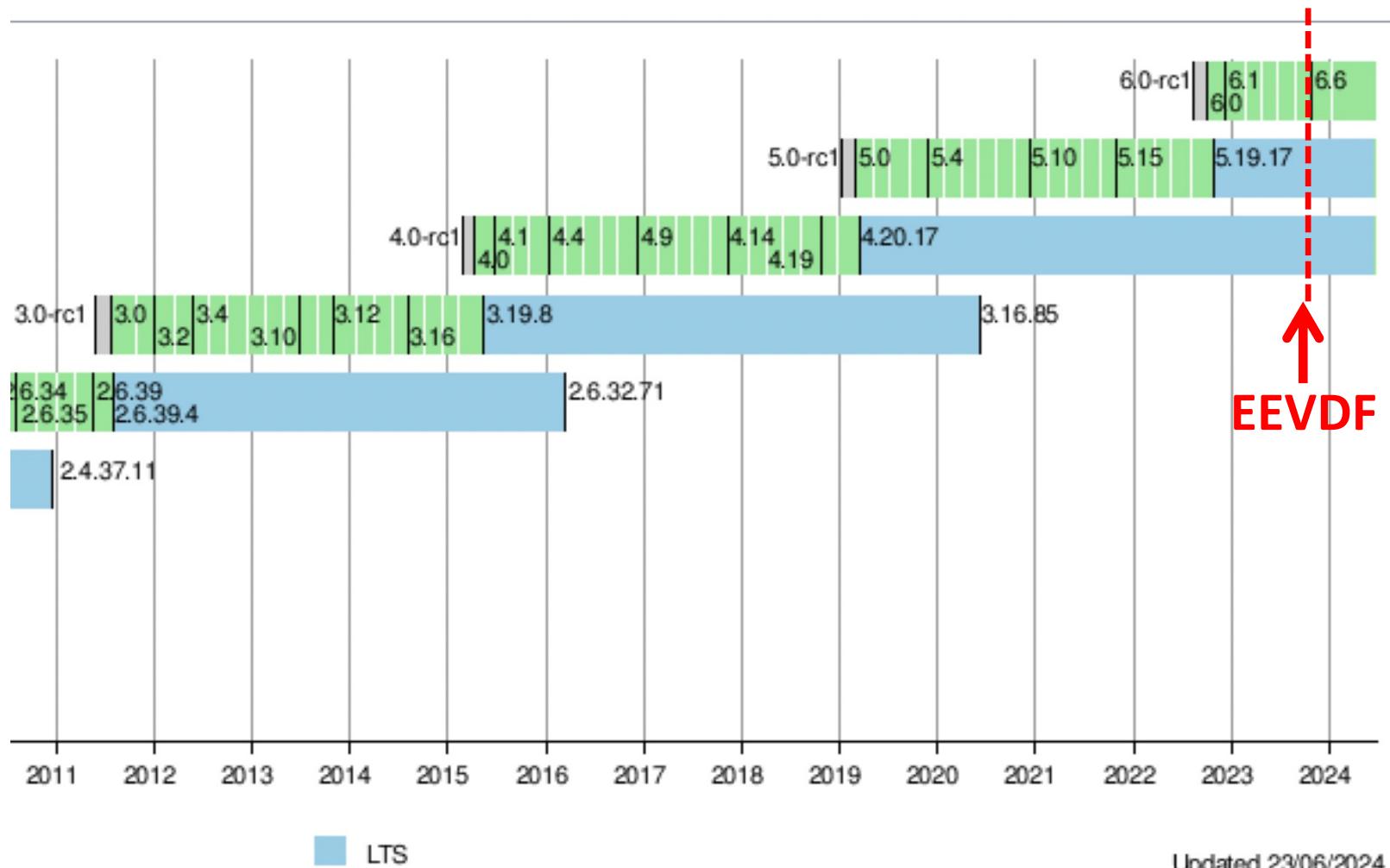
- Simple enough to see all its code, and
- Provides intuition about how all general-purpose schedulers work

## THE LINUX ≤2.4 SCHEDULER

# The <= 2.4 scheduler maintenance period



# Since Oct 2023 (Linux 6.6): EEVDF (Earliest eligible virtual deadline first scheduling)



[https://en.wikipedia.org/wiki/Earliest\\_eligible\\_virtual\\_deadline\\_first\\_scheduling](https://en.wikipedia.org/wiki/Earliest_eligible_virtual_deadline_first_scheduling)

**For the remainder of this presentations**

All definitions relate to the  $\leq 2.4$  scheduler

Not necessarily to other (Linux) schedulers

# **DEFINITIONS**

# Definition: task

- **Recall: within the Linux kernel**
  - Every process is called a “task”
  - Every thread is called a “task”

# Definition: standard POSIX scheduling policies

- **POSIX dictates that**
  - Each task is associated with one of three scheduling policies
    - “Realtime” policies: (1) SCHED\_RR (round robin),  
(2) SCHED\_FIFO (first-in, first-out), or
    - The default policy: (3) SCHED\_OTHER
      - Unlike realtime policies, POSIX defines that the meaning of SCHED\_OTHER (aka SCHED\_NORAML in Linux) is decided by OS
      - Typically employs some multilevel priority queue with the negative feedback loop
  - “Realtime” tasks are *always* favored by the scheduler, if exist
    - Must be admin to use SCHED\_RR & SCHED\_FIFO
  - Users can set policy using the sched\_setscheduler syscall; h.w.: browse
    - [https://man7.org/linux/man-pages/man2/sched\\_setscheduler.2.html](https://man7.org/linux/man-pages/man2/sched_setscheduler.2.html) (syscall)
    - <http://manpages.ubuntu.com/manpages/trusty/man8/schedtool.8.html> (shell utility)
- **Henceforth, focusing on SCHED\_OTHER, except to say...**

# Sidenote

- **POSIX scheduling borrowed the term “realtime”**
  - Which typically has a somewhat different meaning
    - Realtime describes various compute operations that must guarantee response times within a specified time (=deadline)
    - Must be fast enough to affect the environment in which it occurs
      - E.g., in the context of Iron Dome...
- **Commonly, “realtime scheduling” means something like**
  - Input
    - Set of tasks  $\langle \text{arrivaltime}_i, \text{runtime}_i, \text{deadline}_i \rangle$  ( $i=1\dots n$ )
  - Output
    - A schedule that honors all deadlines (=finish runtime by deadline)
    - Or a declaration: “unable to produce such a schedule”

Lecture  
ended here

# Definition: epoch

- (As mentioned earlier for RR)
- Recall that every runnable task gets allocated a quantum
  - CPU time the task is allowed to consume before it's stopped by the OS
- New epoch is started
  - Whenever all quanta of all *runnable* tasks become zero
  - In this case, allocate additional running time to *all tasks*
    - Runnable, or not

# Definition: static/dynamic priorities

- **Task's priority**
  - Every task is associated with an integer
  - Higher value indicates higher priority to run
  - Every task has two different kinds of priorities:
- **Task's static priority component**
  - Fixed unless user invokes the nice() or sched\_setscheduler syscalls
    - Homework: browse <https://linux.die.net/man/2/nice> (syscall) & <https://linux.die.net/man/1/nice> (shell utility)
  - Indirectly determines the maximal quantum for this task
- **Task's dynamic priority component**
  - The (i) **remaining time** for this task to run *and* (ii) its **current priority**
    - Decreases over time (while task is assigned a CPU & is running)
    - When reaches zero, OS forces task to yield CPU (until next epoch)
    - Reinitialized at start of every epoch according to static component

# Definition: HZ, resolution, ticks

- **Hz**
  - Linux gets a timer interrupt HZ times a second
  - Namely, it gets an interrupt every  $1/\text{HZ}$  second
  - ( $\text{HZ}=100$  for x86/Linux2.4)
- **Tick**
  - Two meanings (overloaded term):
    1. The time the elapses between two consecutive timer interrupts
      - Namely, tick =  $1/\text{HZ} = 10$  milliseconds by default for x86/Linux2.4
    2. The timer interrupt that fires every  $1/\text{HZ}$  (= 10 milliseconds)
  - Determines the scheduler timing **resolution**
    - The OS measures the passage of time by counting ticks
    - The **units** of the *dynamic* priority component are '**ticks**'

# Definition: per-task scheduling info

- **Every task is associated with**
  - A task\_struct
- **Every task\_struct has 5 fields used by the scheduler  
(to be further discussed in the next few slides)**
  1. nice (task's static priority)
  2. counter (task's dynamic priority)
  3. processor (identifies core on which task ran most recently)
  4. need\_resched (boolean)
  5. mm (identifies task's memory address space)

# Definition: task's nice (kernel vs. user)

- **Kernel's nice = the static priority component**
  - Between 1 ... 40, initialized to 20 by default
  - Can be changed by nice() and sched\_setscheduler()
  - *Higher* value indicates higher priority (in contrast to user's nice)
- **User's nice (POSIX parameter to the nice system call)**
  - Between -20 ... 19
  - *Smaller* value indicates higher priority (<0 requires superuser)
- **Conversion between kernel & user nice**
  - $\text{kernel\_nice} = 20 - \text{user\_nice}$

# Definition: task's counter

- **The dynamic component (time to run in epoch, & priority)**
  - Upon task creation (integer arithmetic)
    - $\text{child.counter} = \text{parent.counter}/2$ ;  $\text{parent.counter} -= \text{child.counter}$ ;
  - Upon a new epoch
    - $\text{task.counter} = \text{task.counter}/2 + \overbrace{\text{NICE\_TO\_TICKS}(\text{task.nice})}^{\alpha}$   
 $= \text{half of prev dynamic} + \text{convert\_to\_ticks(static)}$
  - When running, decremented each tick ( $\text{task.counter} -= 1$ ) until reaching 0
- **NICE\_TO\_TICKS**
  - Scales 20 (=DEF\_PRIORITY) to number of ticks comprising 50+ ms
  - Namely, scales 20 to 5+ ticks (recall that each tick is 10 ms by default):
    - `#define NICE_TO_TICKS(kern_nice) ((kern_nice)/4 + 1)`
- **Quantum range (without epoch accumulation) is therefore**
  - $(1/4 + 1) = 1$  tick = 10 ms (**min**)
  - $(20/4 + 1) = 6$  ticks = 60 ms (**default**)
  - $(40/4 + 1) = 11$  ticks = 110 ms (**max for SCHED\_OTHER**)

# Definition: processor, need\_resched, mm

- **Task's processor**
  - Logical ID of CPU core upon which task has executed most recently
  - If task is currently running
    - ‘processor’ = logical ID of core upon which the task executes *now*
- **Task's need\_resched**
  - Boolean checked by kernel just before switching back to user-mode
  - If set, check if there’s a “better” task than the one currently running
  - In which case, context switch to it
  - Since this flag is checked only for the currently running task
    - Can think of it as per-core – rather than per-task – variable
- **Task's mm**
  - A pointer to the task's memory address space

# **THE LINUX <=2.4 CODE**

# Scheduler is comprised of 4 functions

## 1. **goodness( task, cpu )**

- Given a task and a CPU, return how “desirable” it is for that CPU
- Compare tasks by this value to decide which will run next on CPU

## 2. **schedule()**

- Actual implementation of the scheduling algorithm
- Uses goodness to decide which task will run next on a given core

## 3. **\_\_wake\_up\_common( wait\_queue q )**

- Wake up task(s) when waited-for event has happened
- (Event may be, for example, completion of I/O)

## 4. **reschedule\_idle( task t )**

- Given a task, check whether it can be scheduled on some core
- Preferably on an idle core, but if there aren't any, by preempting a less desirable task (according to goodness)
- Used by both \_\_wake\_up\_common() and by schedule()

# How desirable is a task on a given core?

```
int goodness(task t, cpu this_cpu) { // bigger = more desirable
    g = t.counter
    if( g == 0 )
        // exhausted quantum, wait until next epoch
        return 0
    if( t.processor == this_cpu )
        // try to avoid migration between cores (why?)
        g += PROC_CHANGE_BONUS
    if( t.mm == this_cpu.current_task.mm )
        // prioritize threads sharing same address space
        // (why?)
        g += SAME_ADDRESS_SPACE_BONUS
    return g
}
```

# Wakeup blocked task(s)

```
void __wake_up_common(wait_queue q) {
    // the blocked tasks residing in q
    // are waiting for an event that has just happened,
    // so try to reschedule all of them
    foreach task t in q
        remove t from q
        add t to ready-to-run list // global, shared by all cores
        reschedule_idle(t)
}
```

# Try to schedule task on *some* core

```
void reschedule_idle(task t) {
    next_cpu = NIL
    if( t.processor is idle )           // t's most recent core is idle
        next_cpu = t.processor
    else if( there exists an idle core ) // some other core is idle
        next_cpu = least recently active idle core
    else                               // no core is idle; is t more desirable
   // than a currently running task?
        threshold = PREEMPTION THRESHOLD // =1
        foreach c in [all cores]       // find c where t is most desirable
            gdiff = goodness(t,c) - goodness( c.current_task,c )
            if( gdiff > threshold )
                threshold = gdiff
                next_cpu = c
    if( next_cpu != NIL )              // found a core for t
        prev_need = next_cpu.current_task.need_resched
        next_cpu.current_task.need_resched = true
        if( (prev_need == false) && (next_cpu != this_cpu) )
            interrupt next_cpu // "IPI" = inter-processor interrupt
}
```

# The heart of the scheduler

```
void schedule(cpu this_cpu) {
    // called when need_resched of this_cpu is on, when switching from
    // kernel mode back to user mode. need_resched can be set by, e.g.,
    // tick interrupt handler, or by I/O device drivers that initiate a slow op
    // (and hence move the associated tasks to a wait_queue)
    prev = this_cpu.current_task
START:
    if( prev is runnable )
        next = prev
        next_g = goodness(prev, this_cpu)
    else
        next_g = -1

    foreach task t in [runnable && not executing]
        // search for 'next' = the next task to run on this_cpu
        cur_g = goodness(t, this_cpu)
        if( cur_g > next_g )
            next = t
            next_g = cur_g
```

# The heart of the scheduler – continued

```
// ...continue function from previous slide...

if( next_g == -1 )          // no ready tasks
    go idle                 // schedule "idle task" (halts this_cpu)

else if( next_g == 0 )        // all quanta exhausted => start new epoch
    foreach task t
        t.counter = t.counter/2 + NICE_TO_TICKS(t.nice)
    goto START;

else if( next != prev )      // context switch
    next.processor = this_cpu
    next.need_resched = false           // 'next' will run next
    context_switch(prev, next);
    if( prev is still runnable )
        reschedule_idle(prev)      // perhaps on another core

// 'next' (which may be equal to 'prev') will run next...
}
```

# **CONCLUDING COMMENTS**

# Convergence of task's counter

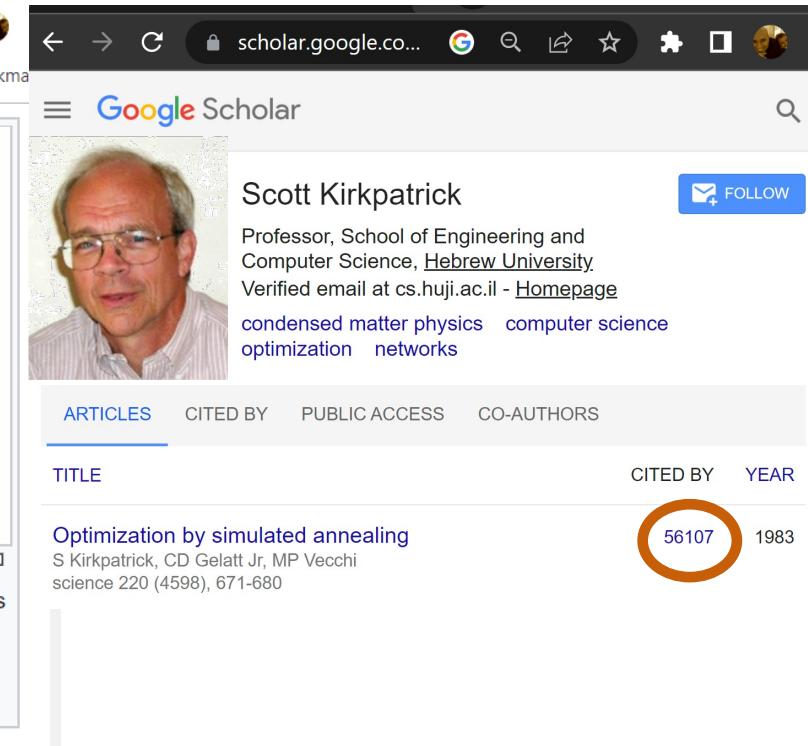
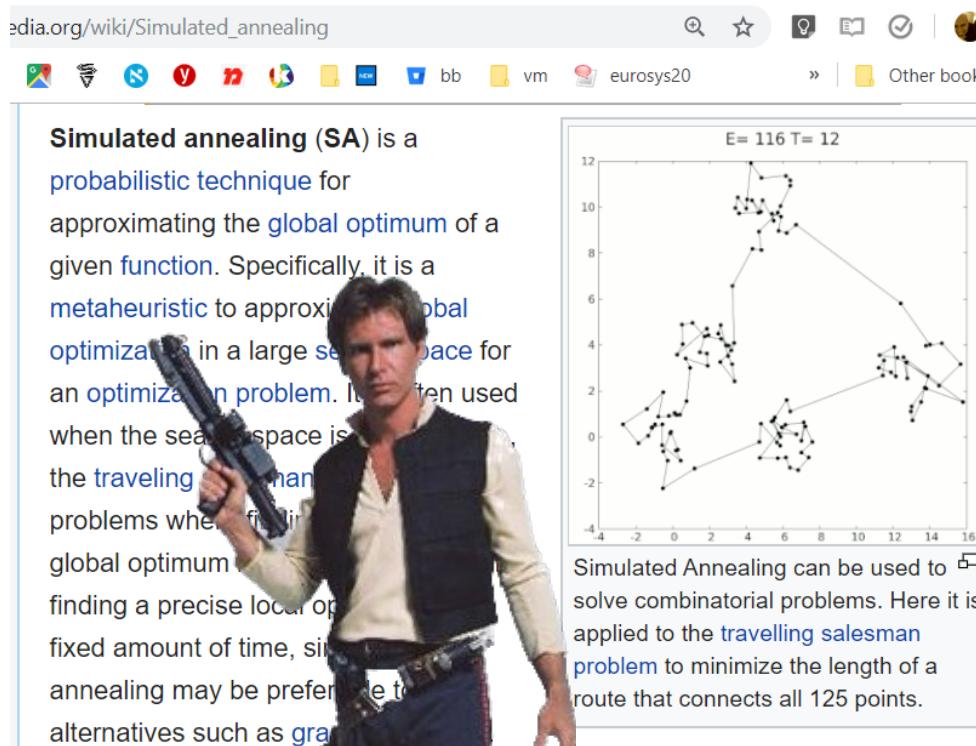
- **Recall that**
  - $\text{task.counter} = \text{task.counter}/2 + \overbrace{\text{NICE\_TO\_TICKS}(\text{task.nice})}^{\alpha}$   
= half of perv dynamic + convert\_to\_ticks(static)
- **Claim**
  - The counter value of an I/O-bound task will quickly converge to  $2\alpha$
  - (Homework: prove it)
- **Corollary**
  - By default, an I/O bound task will have a counter of 12 ticks (=120 ms)
  - (So long as it remains “I/O bound” that consumes negligible CPU time)
- **Notice**
  - As mentioned earlier, this is why text editors remain responsive
  - Even in the face of heavy CPU-bound background tasks
  - When awakened upon event (e.g., key press), usually immediately get the CPU, and run for a duration that’s too short to be “billed” by the tick handler (shorter than a tick)

# Many years ago...

edia.org/wiki/Simulated\_annealing

Simulated annealing (SA) is a probabilistic technique for approximating the global optimum of a given function. Specifically, it is a metaheuristic to approximate global optimization in a large search space for an optimization problem. It is often used when the search space is discrete, the traveling salesman problem being one such problem where finding the global optimum is equivalent to finding a precise local optimum in a fixed amount of time, since annealing may be preferable to other alternatives such as gradient descent.

The name and inspiration come from annealing in metallurgy, a technique involving heating and controlled cooling of a material to increase the size of its crystals and reduce their defects. Such effects are attributes of the material that depend on its thermodynamic free energy. By analogy, annealing and related effects can be used to find an approximate solution to an optimization problem with many variables. In 1983, three researchers, Kirkpatrick, Gelatt Jr., Vecchi, [1] for a solution to a Traveling Salesman Problem with 125 cities, proposed its current name, simulated annealing.

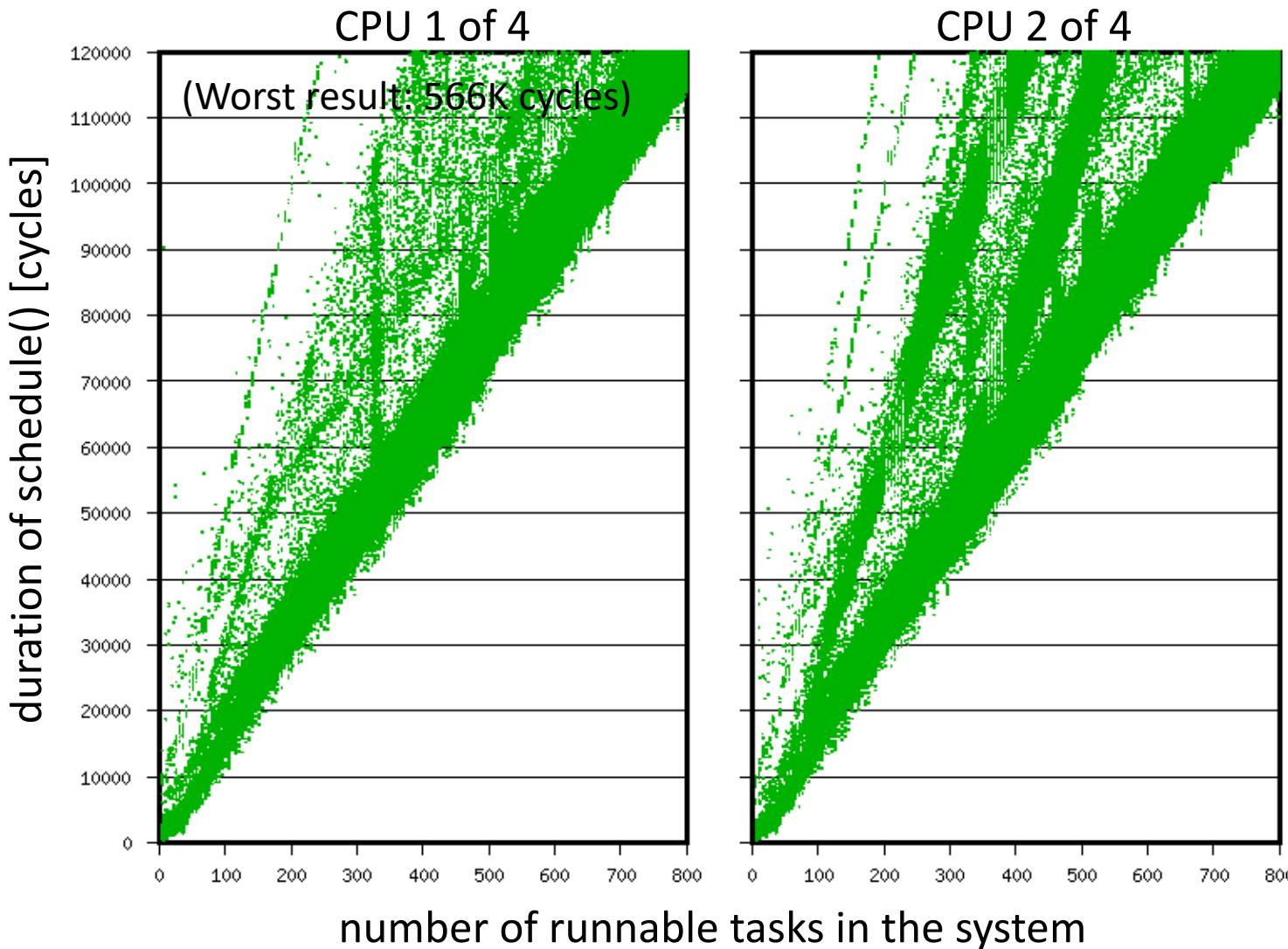


Circa 2000 – A machine named “hansolo”: IBM Netfinity 8500R(?) 4-way “SMP” (= 4 CPUs; no “multicores” back then) with 1GB memory



OS (234123) - scheduling

# Cost of schedule()'s linearity



Measured on  
CPUs 1 & 2 of  
the 4-way  
Netfinity server.

Motivated the  
“O(1) scheduler”  
of Linux, which  
replaced the  
linear scheme.

# Aftermath

- **<=2.4 is in fact a multi-level priority queue**
  - Every priority level integer is (logically) a RR queue
  - When counter is decreased/increased => task moves between queues
- **Drawback**
  - It's a linear scheduler...
  - Indeed, was replaced by “the O(1)” scheduler
  - Which was later replaced by the CFS scheduler, which is O(logn)

# **Operating Systems (234123)**

## *Synchronization*

Dan Tsafrir (2025-05-04, 2025-05-05)

Partially based on slides by Hagit Attiya

# Context

- **A set of threads (or processes) utilize shared resource(s) simultaneously**
  - For example, threads share the memory address space
  - Processes can also share address space (using the right system calls)
  - In this talk, we will use the terms thread/process interchangeably
- **Need to synchronize also in uni-core setup, not just multicore**
  - Userland: a thread can always be preempted in favor of another thread
  - Kernel: interrupts may occur, triggering a different context of execution
- **=> Need to learn about synchronization**
  - This is true for *any* parallel code that shares resources
  - An OS kernel happens to be such a code. (Some believe students should learn about parallelism from the very first course.)

# Example: withdraw money from bank

- **Assume**

- This is the code that runs whenever we make a withdrawal →

- Account holds \$50K

- Account has two owners

- Both owners make a withdrawal simultaneously from two ATMs

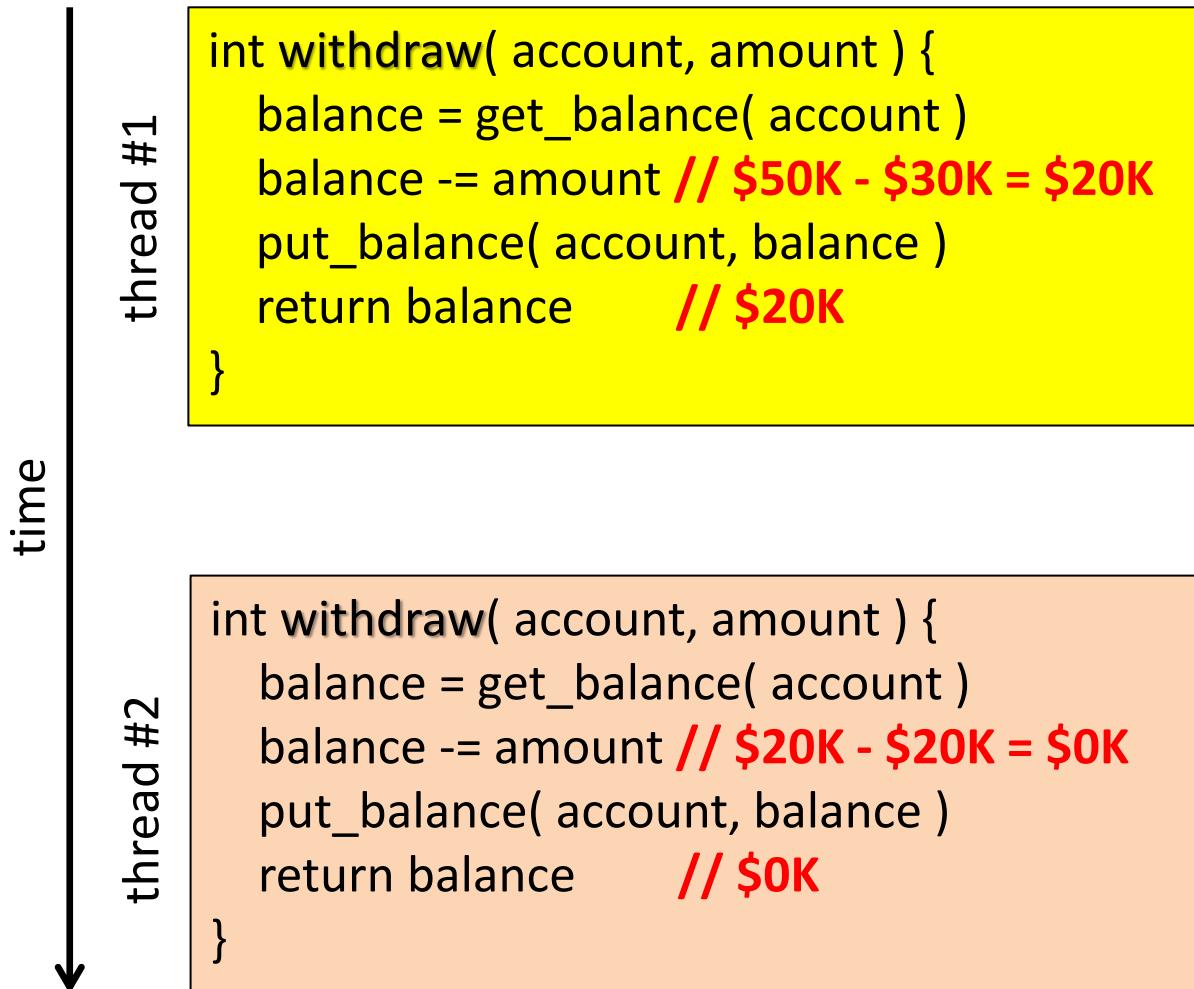
1. One takes out \$20K
2. The other takes out \$30K

- Every operation is done by a different thread on a different core

```
int withdraw( account, amount ) {  
    balance = get_balance( account )  
    balance -= amount  
    put_balance( account, balance )  
    return balance  
}
```

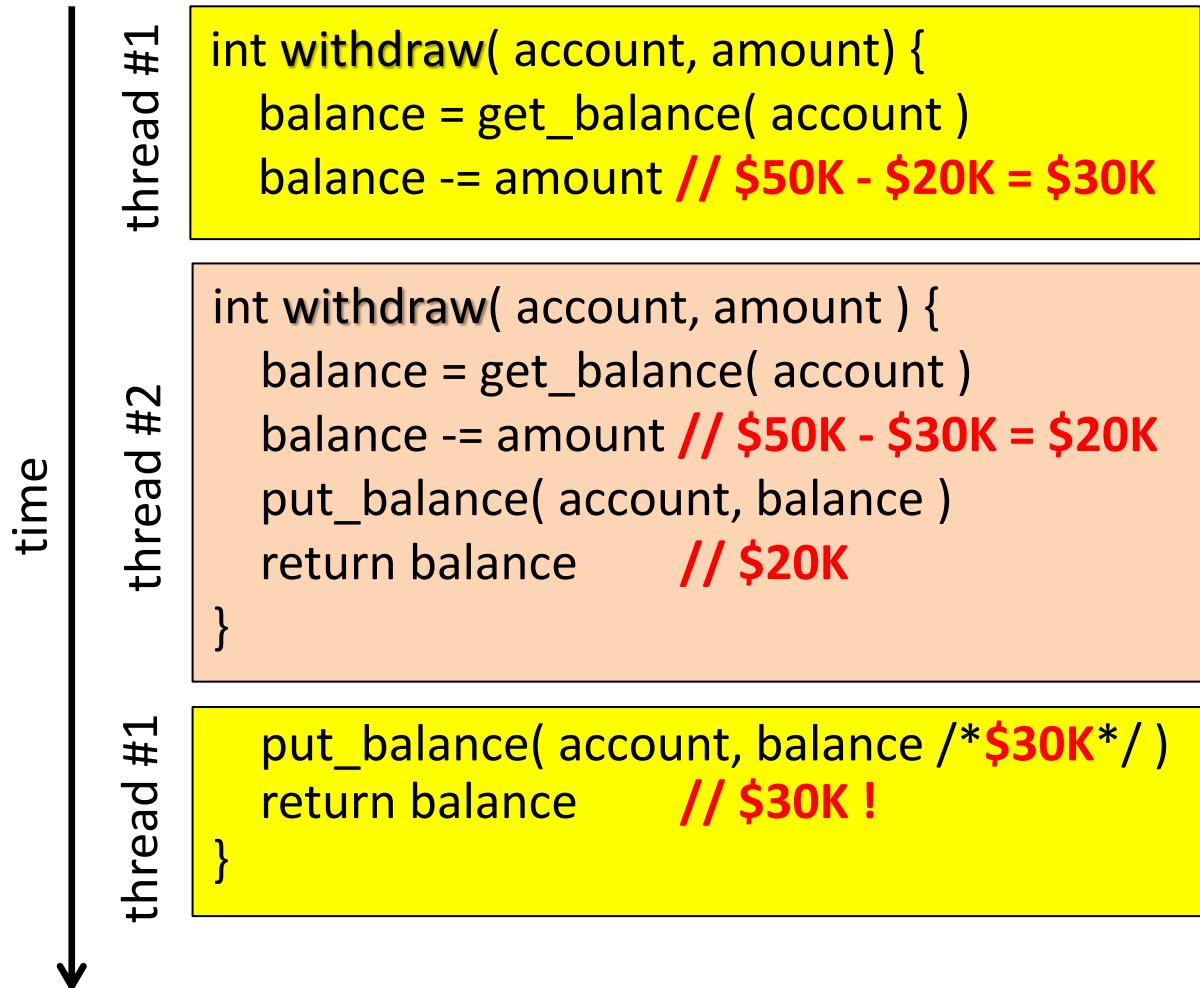
# Example: withdraw money from bank

- Best case scenario from bank's perspective
  - \$0 in account



# Example: withdraw money from bank

- A better scenario from *owners' perspective*
  - \$30K in account...
- We say that the program suffers from a **“race condition”**
  - Outcome is nondeterministic and depends on the timing of uncontrollable, unsynchronized events



# Example: too much milk

## *Slide used between 2025 – ?*

| time  | Beyoncé             | Jay-Z               |
|-------|---------------------|---------------------|
| 15:00 | checks fridge       |                     |
| 15:05 | goes to supermarket | checks fridge       |
| 15:10 | buys milk           | goes to supermarket |
| 15:15 | gets back home      | buys milk           |
| 15:20 | puts milk in fridge | gets back home      |
| 15:25 |                     | puts milk in fridge |



TRADER JOE'S



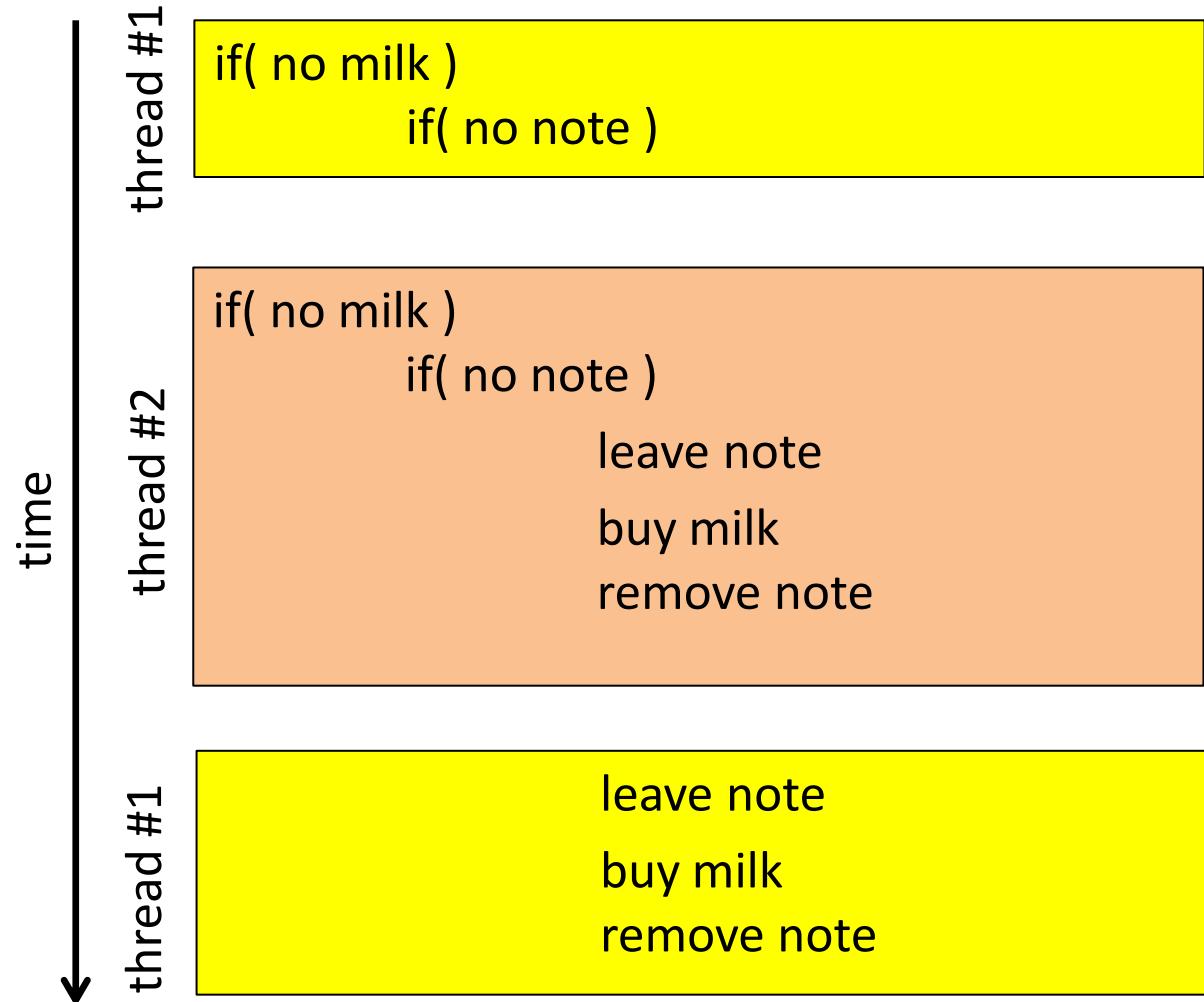
# Milk problem: solution #1?

- **Leave note on fridge before going to supermarket**
  - Probably works for humans
  - But for threads...

```
if( no milk )
    if( no note )
        leave note
        buy milk
        remove note
```

# Milk problem: solution #1 😠

- Too much milk,  
again...



# Milk problem: solution #2?

- First leave note, then check other note

## thread A:

```
leave note A  
if( ! note B )  
    if( no milk )  
        buy milk  
remove note A
```

## thread B:

```
leave note B  
if( ! note A )  
    if( no milk )  
        buy milk  
remove note B
```

# Milk problem: solution #2 😞

- Leave note, check other note, then check fridge!

thread A:

leave note A

if( ! note B ) // there is!  
    if( no milk )

~~buy milk~~  
remove note A

thread B:

leave note B

if( ! note A ) // there is!  
    if( no milk )

~~buy milk~~

remove note B

- => No milk...

# Milk problem: solution #3?

thread A (change ‘if’ to ‘while’)

```
leave note A  
while( note B ) do NOP  
if( no milk )  
    buy milk  
remove note A
```

thread B (same as before)

```
leave note B  
if( ! note A )  
    if( no milk )  
        buy milk  
remove note B
```

- **In the past, they'd have told you**

1. That it works due to the asymmetry! ☺ (= ‘A’ waits until ‘B’ is done)
2. But that
  - Asymmetry complicates things
  - It’s “unfair” (=> ‘A’ works harder: if 1<sup>st</sup> line executed exactly at the same time, it’ll be ‘A’ that will buy the milk)
  - Works for only two threads (what if there are more?)

# Milk problem: solution #3?

thread A (change ‘if’ to ‘while’)

leave note A

while( note B ) do NO

if( no milk )

but

remove

(waits before)

milk

But this  
doesn't work  
with modern  
hardware

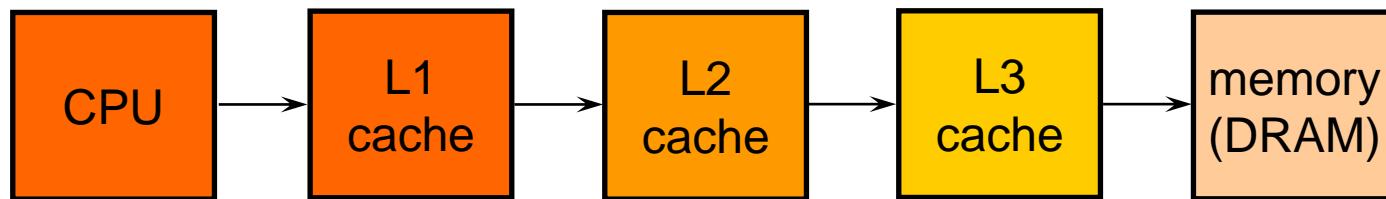
- In the notes:
  1. That’s fine (note A waits until B is done)
  2. But there are problems
    - Asymmetric race conditions
    - It’s “unfair” to the first thread harder: if 1<sup>st</sup> line executed exactly at the same time, it will buy the milk
    - Works for only two threads (what if there are more?)

Memory (1) coherency & (2) consistency

# **COMPUTER ARCHITECTURE IN A NUTSHELL – PART #2**

# Reminder: memory tradeoffs

- Larger (denser) memories are slower
- Faster memories are smaller, more expensive, and consume more energy
- **Goal** – give the processor a feeling that it has a memory which is large (dense), fast, consumes low power, and cheap
- **Solution** – a hierarchy of memories, exploiting locality principle



|            |          |   |         |
|------------|----------|---|---------|
| speed..... | fastest  | → | slowest |
| size.....  | smallest | → | biggest |
| cost.....  | highest  | → | lowest  |
| power..... | highest  | → | lowest  |

# Reminder: memory hierarchy example

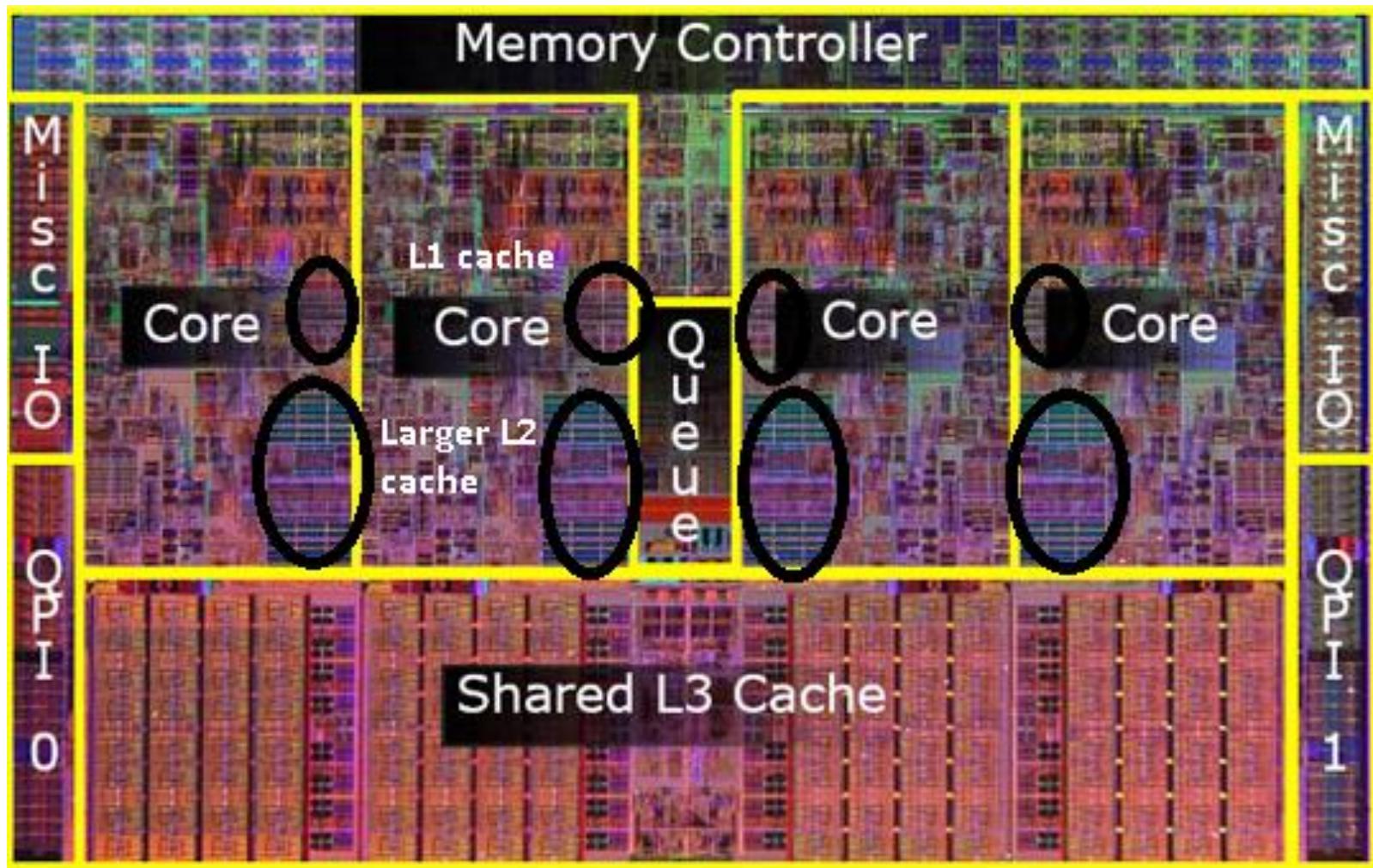
*Intel Core i7-6700 (Skylake), Aug 2015*

Base frequency : 3.40 GHz (cycle  $\approx$  1/3 nanosecond)

Turbo frequency: 4.00 GHz (cycle = 1/4 nanosecond)

| memory level                                  | size                                              | latency                                         |
|-----------------------------------------------|---------------------------------------------------|-------------------------------------------------|
| CPU registers                                 | 100s of bytes                                     | < cycle                                         |
| L1 cache                                      | 32 KB (i) + 32 KB (d)<br>(i=instructions; d=data) | 4–5 cycles                                      |
| L2 cache                                      | 256 KB                                            | 12 cycles                                       |
| L3 cache (called “LLC”<br>= last level cache) | 8 MB                                              | 42 cycles                                       |
| Max main memory size<br>(DRAM)                | 256 GB (for example)                              | 42 cycles + 51 nanosec<br>= 246 cycles (@ 4GHz) |

# Reminder: cores typically have private caches



(Nehalem, i7)

# Cache coherency (a.k.a. “memory coherency”)

*Relates to a **single** memory location*

| time | event              | cached value in core1 | cached value in core2 | memory contents for location X |
|------|--------------------|-----------------------|-----------------------|--------------------------------|
| 0    |                    |                       |                       | 1                              |
| 1    | core1 read(X)      | 1                     |                       | 1                              |
| 2    | core2 read(X)      | 1                     | 1                     | 1                              |
| 3    | core1 write(X) = 0 | 0 (new)               | 1 (old)               | ?                              |

This can't happen when **memory (hierarchy) is “coherent”**

1. Propagates the new value to other relevant cores “soon”
2. Ensures cores see writes in *some* order that makes sense
  - If one core observes writes to X in order:  $\text{write1}(X)$  before  $\text{write2}(X)$
  - Another core must *not* observe:  $\text{write2}(X)$  before  $\text{write1}(X)$

Stale value: next  $\text{read}(X)$  by core2 could be “1”!

Hardware achieves this using, e.g., the “MESI” cache coherency protocol, which is invalidation-based: each write invalidates copies in other caches (exact details beyond the scope of this course; see computer arch. course).

# Cache consistency – relates to *multiple locations*

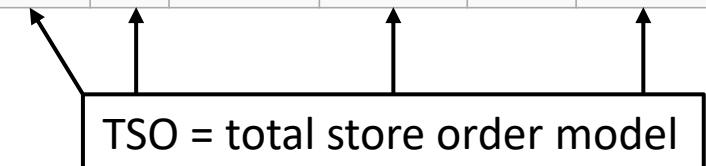
- **Assume**
  - Locations A & B cached by C1 & C2
- **If store operations are**
  1. Immediately seen by other cores
  2. Cannot be reordered with load ops
- **Then it's impossible for both “if” conditions to be true**
  - Reaching an “if” statements means either A or B holds 1
- **But on modern HW, both #1 and #2 can be violated due to performance considerations; e.g., non-blocking writes**
  1. Stored values can be temporarily placed in a local “store buffer”, temporarily remaining invisible to other cores, so...
  2. Cores might reorder load & store ops if there are no (apparent) dependencies between them (as in the per-core accesses of A & B)
- **Should this be allowed?**
  - Determined by the memory consistency model

| Core C1                                           | Core C2                                           |
|---------------------------------------------------|---------------------------------------------------|
| A = 0; // init                                    | B = 0; // init                                    |
| ...                                               | ...                                               |
| A = 1;                                            | B = 1;                                            |
| if ( B == 0 )<br>// I'm alone<br>// => fast path! | if ( A == 0 )<br>// I'm alone<br>// => fast path! |

# Consistency models

[https://en.wikipedia.org/wiki/Memory\\_ordering](https://en.wikipedia.org/wiki/Memory_ordering)

| Type                                | Alpha | ARMv7 | PA-RISC | POWER | SPARC RMO | SPARC PSO | SPARC TSO | x86 | x86 oostore | AMD64 | IA-64 | zSeries |
|-------------------------------------|-------|-------|---------|-------|-----------|-----------|-----------|-----|-------------|-------|-------|---------|
| Loads reordered after loads         | Y     | Y     | Y       | Y     | Y         |           |           |     | Y           |       | Y     |         |
| Loads reordered after stores        | Y     | Y     | Y       | Y     | Y         |           |           |     | Y           |       | Y     |         |
| Stores reordered after stores       | Y     | Y     | Y       | Y     | Y         | Y         |           |     | Y           |       | Y     |         |
| <b>Stores reordered after loads</b> | Y     | Y     | Y       | Y     | Y         | Y         | Y         | Y   | Y           | Y     | Y     | Y       |
| Atomic reordered with loads         | Y     | Y     |         | Y     | Y         |           |           |     |             |       | Y     |         |
| Atomic reordered with stores        | Y     | Y     |         | Y     | Y         | Y         |           |     |             |       | Y     |         |
| Dependent loads reordered           | Y     |       |         |       |           |           |           |     |             |       |       |         |



# Consistency – how to enforce ordering

- On x86, an explicit *memory fence* (a.k.a. *memory barrier*) would eliminate the problem
  - Makes all store-s that happened before the fence visible to all load-s after the fence
  - Notably, flushes the store buffer to the memory system
- Therefore, programmers must be aware of all of this

| Processor P1             | Processor P2             |
|--------------------------|--------------------------|
| A = 0; // init           | B = 0; // init           |
| ...                      | ...                      |
| A = 1;<br><b>mfence;</b> | B = 1;<br><b>mfence;</b> |
| if ( B == 0 ) ...        | if ( A == 0 ) ...        |

# **BACK TO SYNCHRONIZATION**

# Milk problem: solution #3 😞

thread A (change ‘if’ to ‘while’)

```
leave note A  
while( note B ) do NOP  
if( no milk )  
    buy milk  
remove note A
```

thread B (same as before)

```
leave note B  
if( ! note A )  
    if( no milk )  
        buy milk  
remove note B
```

- **For some consistency models, notably TSO**
  - Thread A might read stale value of note B (it's still in B's store buffer)
  - And vice versa
  - Meaning, once again, both would purchase milk
- **Fix**
  - With the help of explicit memory barriers=fences, placed after “leave note”

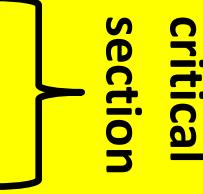
# Towards a solution: critical sections

- **The heart of the synchronization problem**
  - Unsynchronized access to data structure with “incomplete” changes
    - In our example: changing the “state” of the fridge
    - State could be involve
      - Linked lists, hash tables, search trees, ... , and their composition
      - (Regardless of whether parallelism is in kernel or user-level)
  - Doing only *part* of the work before another thread interferes
    - If only we could do multiple operations “atomically”...
- **A group of operations we need to do atomically is called a “critical section”**
  - Atomicity of the critical section would make sure
    - Other threads don’t see partial results
  - The critical section can be the same code across all threads
    - But can also be different

# Towards a solution: critical sections

- Example for the same code across threads

```
int withdraw( account, amount ) {  
    balance = get_balance( account )  
    balance -= amount  
    put_balance( account, balance ) } }  
    } }
```

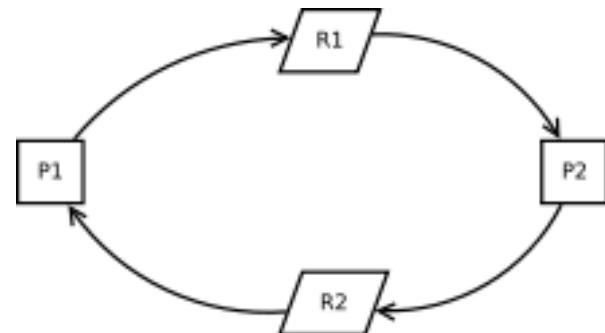


A brace groups the assignment and subtraction statements as a 'critical section'.

- Different code examples
  - If one thread increments (“++”) and the other decrements (“--”) a shared variable
  - Insertion/deletion of an element to/from a linked list

# Towards a solution: requirements

- **Mutual exclusion (“mutex”)**
  - To achieve atomicity
    - Threads execute an entire critical section one at a time
    - Never simultaneously
  - Thus, a critical section is a “**serialization point**”
- **Progress**
  - At least one thread gets to do the critical section at any time
  - No “**deadlock**”
    - Deadlock is a situation in which two or more competing actions are each waiting for the other to finish, and thus no one ever finishes
  - No “**livelock**”
    - Same as deadlock, except state changes
    - E.g., 2 people in narrow corridor, trying to be polite by moving aside to let the other pass, ending up swaying from side to side



# Towards a solution: optional requirements

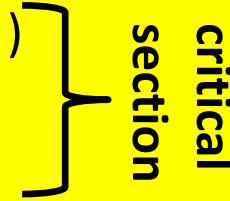
- **Fairness, which means...**
  - No starvation, namely, a thread that wants to do the critical section, would eventually succeed
  - Nice to have: bounded waiting
  - Nice to have: FCFS

# Solution: locks

- **Abstraction that supports two operations**
  - acquire(lock)
  - release(lock)
- **Semantics**
  - It's a memory **fence**
  - **Only one** thread can acquire at any given time
  - Other simultaneous **attempts to acquire are postponed**
    - Until the lock is released, at which point another thread will get it
  - Thus, at most one thread holds a lock at any given time
- **When a lock “protects” a critical section...**
  - Lock is acquired at the beginning of the critical section
  - Lock is released at the end of the critical section
- **...Then atomicity is guaranteed**

# Solution: locks

```
int withdraw( account, amount ) {  
    acquire( account->lock )  
    balance = get_balance( account )  
    balance -= amount  
    put_balance( account, balance )  
    release( account->lock )  
    return balance  
}
```



The code snippet illustrates a critical section. It begins with an **acquire** operation on a lock, followed by a series of operations (getting the balance, subtracting the amount, and putting the balance back) which are grouped together by a brace. This entire group is labeled "critical section". Finally, it ends with a **release** operation on the same lock.

# Solution: locks

- 2 threads make a withdrawal
- What happens when the pink tries to acquire?
- Is it okay to return outside the critical section?
  - Depends
  - Yes, if you want the balance at time of withdraw(), and you don't care if it changed since
  - Otherwise, need to acquire lock outside of withdraw(), rather than inside

```
int withdraw( account, amount ) {  
    acquire( account->lock )  
    balance = get_balance( account )  
    balance -= amount
```

```
int withdraw( account, amount ) {  
    acquire( account->lock )
```

```
    put_balance( account, balance )  
    release( account->lock )
```

```
    balance = get_balance( account )  
    balance -= amount  
    put_balance( account, balance )  
    release( account->lock )  
    return balance
```

```
return balance
```

# Implementing locks

- **When you try to implement a lock**
  - You quickly find that it involves a critical section...
  - Recursive problem
- **There are 2 ways to overcome the problem**
  1. Using SW only, no HW support
    - Possible (was once part of OS courses), but complex, error prone, wasteful, and nowadays completely irrelevant, because...
  2. Using HW support (all modern processors provide such support)
    - There are special instructions that ensure mutual exclusions (later)
    - Fairness typically not a problem within the kernel, because
      - Either critical sections are very short
      - Or we ensure fairness explicitly

spinlocks

# FINE-GRAINED SYNCHRONIZATION

# Possible kernel spinlock with x86's xchg

```
struct spinlock {  
    uint locked; // is the lock held? (0|1)  
};
```

```
inline uint  
xchg(volatile uint *addr, uint newval) {  
    uint oldval;  
    asm volatile("lock; xchgl %0, %1" :  
        "+m" (*addr), "=a" (oldval) :  
        "1" (newval) :  
        "cc");  
    // atomic[ oldval = *addr,  
    //         swap( *addr , newval ) ]  
    return oldval;  
}
```

(The ‘while’ is called “spinning”)

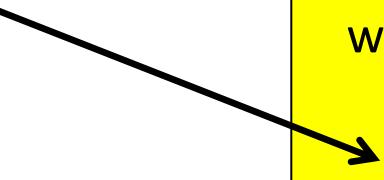
```
void  
acquire(struct spinlock *lk) {  
    disableInterrupts(); // kernel + this core  
    while( xchg(&lk->locked, 1) != 0 )  
        // xchg() is atomic, and the “lock”  
        // prefix adds a “fence” (so read/write  
        // ops after acquire aren’t reordered  
        // before it)  
        ;  
}
```

```
void  
release(struct spinlock *lk) {  
    xchg(&lk->locked, 0);  
    enableInterrupts(); //kernel; this core  
}
```

# Kernel spinlock issues

- **In our implementation, interrupts are disabled**

- While the lock is held
- And while the kernel is spinning
- If we want to allow interrupts while spinning...



```
void
acquire(struct spinlock *lk) {
    disable_interrupts();
    while( xchg(&lk->locked, 1) != 0 ) {
        enable_interrupts();
        disable_interrupts();
    }
}
```

- **Why?**

- Responsiveness
- (Kernel threads don't go to sleep with locks, and they hold locks for very short periods of time)

# Kernel spinlock issues

- **In the kernel, on a (uni)core, do we need to lock?**
  - Need to disable interrupts if handlers might access the same data structures; can be done via our previously defined acquire/release
- **On a multicore, do we care if *other* cores take interrupts?**
  - No: if they want the lock, they will acquire regardless
  - (In particular, within the interrupt handler)
- **User space can't disable interrupts, but...**
  - Is there something equivalent to interrupts that we need to address when considering synchronization?
    - Signals (we may want to block them for a while if the access the same data structures as the regular program)
  - Can we make sure we spin until we get the lock?
    - No, the OS might preempt us
    - But that's typically fine and acceptable

# Spinlock issues

- Other hardware-supported atomic ops to implement spinlocks
  - Compare-and-swap (“CAS”), which atomically does

```
bool cas(int *p, int old_val, int new_val) {  
    if( *p == old_val ) {  
        *p = new_val;  
        return true;  
    } else {  
        return false;  
    }
```

- Test-and-set, which atomically does

```
bool tas(bool *b) {  
    bool old_value = *b;  
    *b = true;  
    return old_value;  
}
```

Semaphores, conditional variables, monitors

# **COARSE-GRAINED SYNCHRONIZATION: BLOCK (SLEEP) WHILE WAITING**

# Spin or block? (applies to kernel & user)

- **Two basic alternatives**
  - Spin (busy wait)
    - Might waste too many cycles
  - Block (go to sleep) until the event you wait for has occurred
    - Free up the CPU for other useful work
  - OS offers blocking as a service; it doesn't offer spinning. If users want to spin, they can do it on their own, they don't need the kernel for it
- **When to spin?**
  - Rule of thumb:
    - Spin if we're going to get the lock very soon, sooner than a context switch takes
- **When to block?**
  - Rule of thumb:
    - Don't spin if we're going to get the lock later than the duration of a context switch

# Spin or block? (applies to kernel & user)

- Consider the following parallel program canonical structure

```
- for(i=0; i<N; i++) {  
    compute();           // duration is C cycles  
    sync();             // takes S cycles; spin? block?  
}
```

- Runtime is

- $N * (C + S) = N * C + N * S$
- $N * C$  is set by user; as an OS, we can't really do anything about it
- But with  $S$ , we may have a choice...
- What happens if  $C \ll S$ ?
  - Sync time dominates 😞
- If we have a fairly good idea that spinning would end much sooner than a context switch => should spin, or else runtime would explode
- This is how it's typically done within kernels (spinlocks are used to protect short critical sections)

# Coarse-grained synchronization

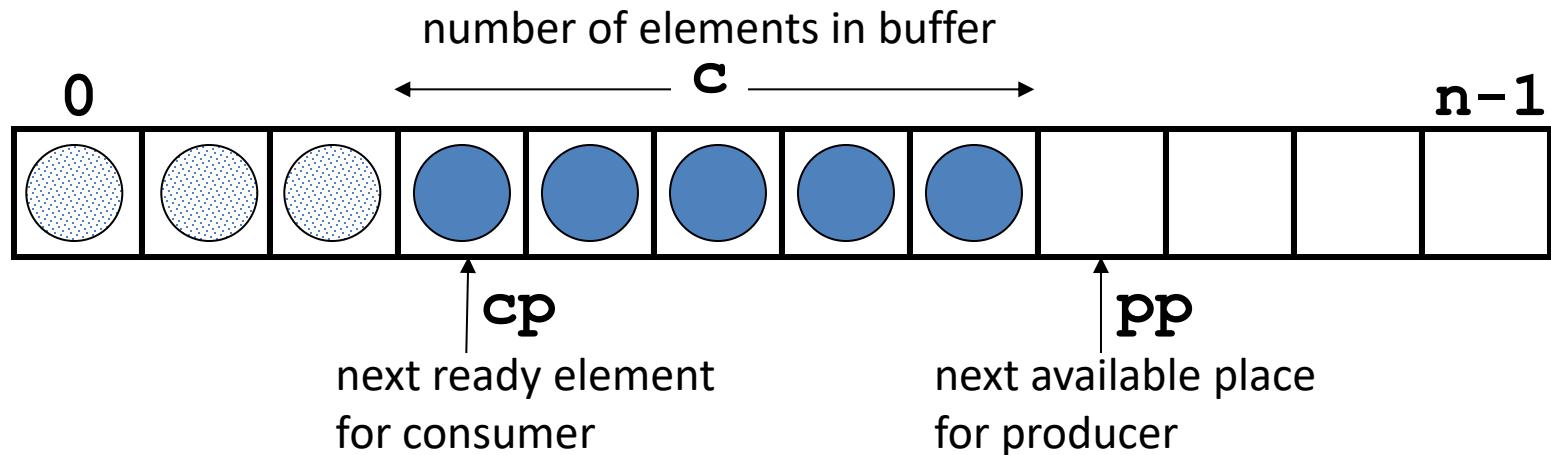
- **Waiting for a relatively long time**
  - Better relinquish the CPU:
    - Leave runnable queue
    - Move to sleep queue, waiting for an event
- **The OS provides several services that involve sleeping**
  - All system calls that block by default: read, write, select, setitimer, sigsuspend, pipe, sem\*, ...
- **Unlike spinning, user-land can't sleep on its own**
  - It must request the OS to sleep, since sleeping involves changing process states

Semaphores – motivating example

# **PRODUCER-CONSUMER RING**

# Producer/consumer – example problem

- **Problem**
  - Two threads share an address space
  - The “producer” produces elements
    - (E.g., decodes video frames to be displayed (element = frame))
  - The “consumer” consumes elements
    - (E.g., displays the decoded frames on screen)
- **Typically implemented using**
  - “Cyclic buffer” (indexed modulo n) a.k.a. “ring buffer”



# Producer/consumer – faulty solution

```
int c = 0; // shared var, counts  
           // produced & unconsumed  
producer:  
while( 1 )  
    busywait while (c==n);  
    buf[pp] = new item;  
    pp = (pp+1) mod n;  
    c += 1;
```

consumer:

```
while( 1 )  
    busywait while (c==0);  
    consume buf[cp];  
    cp = (cp+1) mod n;  
    c -= 1;
```

- **Might access ‘c’ simultaneously**
  - The “ $+=$ ” and “ $-=$ ” operations aren’t necessarily atomic
- **Busy waiting might be suboptimal**
  - Wasting valuable CPU cycles

# Producer/consumer – semaphore solution

```
semaphore_t free_space( n );
semaphore_t avail_items( 0 );
```

## producer:

```
while( 1 )
    wait( free_space );
    buf[pp] = new item;
    pp = (pp+1) mod n;
    signal( avail_items );
```

## consumer:

```
while( 1 )
    wait( avail_items );
    consume( buf[cp] );
    cp = (cp+1) mod n;
    signal( free_space );
```

- **Wait**
  1. Decrement semaphore's counter by 1
  2. If semaphore's counter <0, sleep until signal-ed
- **Signal**
  1. Increment semaphore's counter by 1
  2. If wait-ing threads exist, wake up the one that has been waiting longest

# Producer/consumer – semaphore solution

```
semaphore_t free_space( n );
semaphore_t avail_items( 0 );
```

## producer:

```
while( 1 )
    wait( free_space );
    buf[pp] = new item;
    pp = (pp+1) mod n;
    signal( avail_items );
```

## consumer:

```
while( 1 )
    wait( avail_items );
    consume( buf[cp] );
    cp = (cp+1) mod n;
    signal( free_space );
```

- **Works with only one consumer/producer pair**
  - Q: what if there are multiple producers/consumers?
    - For example, as the Apache webserver typically works
  - A: serialize access to the pp and cp pointers
    - E.g., spinlock right after ‘wait’ and unlock right before ‘signal’

Design & implementation

# **SEMAPHORES**

# Semaphore – concept

- **Proposed by Dijkstra**
  - @ 1968
- **Allows tasks to**
  - Coordinate using (several instances of) a resource
- **Use “flag” to announce**
  1. Either: I’m waiting for a resource
  2. Or: resource has just become available
- **A party who announces that it’s waiting for a resource**
  - Will get the resource if its available, or
  - Will go to sleep, otherwise
  - In which case it’ll be awakened when the resource becomes available to it



# Semaphore – fields

- **Value (integer)**
  - Nonnegative  
=> counting the number of resources currently available
  - Negative  
=> counting the number of tasks waiting for a resource
- **A queue of waiting tasks**
  - Waiting for the resource to become available
  - When value is negative,  $|value| = \text{queue.length}$

# Semaphore – interface

- **Wait(semaphore)**
  - value -= 1
  - If( value >= 0 ) // it was >= 1 before we decreased it
    - Task can continue to run (it has been assigned the resource)
  - Else
    - Place task in waiting queue
- **A.k.a.**
  - P() or proben

Babylon German-English  
dictionary

• **Probe (die)**

n. trial, test; trying experience; test period, probation, conditional release from jail during which a criminal is under supervision of a probation officer; rehearsal, practice session for a performance; assay

# Semaphore – interface

Lecture  
ended here

- **Signal(semaphore)**

- value += 1
- If( value <= 0 ) // it was <= -1 before we increased it,  
// which means somebody is waiting
  - Remove oldest-waiting task from wait-queue
  - Give it a resource instance
  - Wake it (make it runnable)

- **A.k.a.**

- V() or verhogen

Babylon Dutch-English



• **verhogen**

v. heighten, put up, make higher, raise, advance, increase, enhance, send up, lift, exalt, augment

# **HOW TO IMPLEMENT SEMAPHORES**

# How to implement Semaphores

```
struct semaphore_t { int value; wait_queue_t wq; };
```

```
void wait(semaphore_t *s) {  
  
    s->value -= 1;  
    if( s->value < 0 ) {  
        enqueue( self, &s->wq );  
  
        block( self );  
    }  
}
```

```
void signal(semaphore_t *s) {  
  
    s->value += 1;  
    if( s->value <= 0 ) {  
        p = dequeue( &s->wq );  
        wakeup( p );  
    }  
}
```

- **Doesn't work**
  - The semaphore\_t fields (value and wq) are accessed concurrently
  - (For starters...)

# How to implement Semaphores

```
struct semaphore_t { int value; wait_queue_t wq; lock_t l; };
```

```
void wait(semaphore_t *s) {
    lock( &s->l );
    s->value -= 1;
    if( s->value < 0 ) {
        enqueue( self, &s->wq );
        unlock( &s->l );
        block( self );
    }
    else
        unlock( &s->l );
}
```

```
void signal(semaphore_t *s) {
    lock( &s->l );
    s->value += 1;
    if( s->value <= 0 ) {
        p = dequeue( &s->wq );
        wakeup( p );
    }
    unlock( &s->l );
}
```

- **Doesn't work**
  - The well-known “problem of lost wakeup” (make sure you can find it)

# Semaphore – implementation ☺

- **In the kernel**
  - There's an interaction between **sleep** and **spinlock** and **context switch**
  - A task should be able to go to sleep (block) holding a lock...  
...and wake up holding that lock
  - The kernel does the magic of making it happen without deadlocking the system (freeing the lock in between)
  - To see how it's *really* done:

236376 – *Operating Systems Engineering (OSE)*  
<http://webcourse.cs.technion.ac.il/236376>

*Build an operating system from “scratch”  
(minimalistic, yet fully functional)*

- **Note that the semaphore does busy-waiting**
  - But that it has a very short critical section

# Semaphore vs. spinlock

|                      | spinlock                                                        | semaphore                                                                                                                        |
|----------------------|-----------------------------------------------------------------|----------------------------------------------------------------------------------------------------------------------------------|
| wait by              | spinning                                                        | sleeping                                                                                                                         |
| granularity          | fine                                                            | coarse: might wait for a long time                                                                                               |
| complexity           | lower                                                           | greater: typically uses lock                                                                                                     |
| expressiveness       | lower                                                           | greater: equivalent to lock if maximal value=1 (called “binary semaphore”); otherwise, counting how many resources are available |
| interface ordering   | strict: must acquire before release, must release after acquire | relaxed: may signal(s) without wait(s), may wait(s) without signal(s)                                                            |
| interface dependence | strict: release only invoked by locker                          | relaxed: signal may be invoked by threads that didn't previously wait and vice versa                                             |

- **Semaphore could be perceived as less “structured”**
  - Due to the relaxed interface’s ordering and dependence

# Semaphore – POSIX system calls

- A **semaphore** is a kernel object manipulated through
  - `semctl()`, `semget()`, `semop()`, `sem_close()`, `sem_destroy()`,  
`sem_getvalue()`, `sem_init()`, `sem_open()`, `sem_post()`, `sem_unlink()`,  
`sem_wait()`,
- **Homework**
  - Take a look at  
<http://pubs.opengroup.org/onlinepubs/009695399/functions/semop.html>

Using Semaphores

# **CONCURRENT READERS, EXCLUSIVE WRITER (CREW)**

# Concurrent readers, exclusive writer (CREW)

- **Multiple tasks want to read/write the same data element**
- **For consistency, need to enforce the following rules**
  - No problem for several tasks to read simultaneously
  - But when a task is writing, no other task is allowed to read or write
- **Table denoting if multiple access is allowed**

|        | reader | writer |
|--------|--------|--------|
| reader | ✓      | ✗      |
| writer | ✗      | ✗      |

# CREW ☹

```
int r = 0;                                // number of concurrent readers
semaphore_t sRead ( 1 );                  // defends (serializes) 'r'
semaphore_t sWrite ( 1 );                 // writers' mutual exclusion
```

## writer:

```
wait( sWrite )  
[write]  
signal( sWrite )
```

## reader:

```
wait( sRead )  
[read]  
signal( sRead )
```

- **Doesn't work**

- No mutual exclusions between readers and writers
- Only one reader at a time

# CREW ☹

```
int r = 0;                                // number of concurrent readers
semaphore_t sRead ( 1 );                  // defends (serializes) 'r'
semaphore_t sWrite ( 1 );                 // writers' mutual exclusion
```

## writer:

```
wait( sWrite )
[write]
signal( sWrite )
```

## reader:

```
wait( sWrite )
wait( sRead )
[read]
signal( sRead )
signal( sWrite )
```

- **Doesn't work**
  - Only one reader at a time

# CREW 😐

```
int r = 0;                                // number of concurrent readers
semaphore_t sRead ( 1 );                  // defends (serializes) 'r'
semaphore_t sWrite ( 1 );                 // writers' mutual exclusion
```

## writer:

```
wait( sWrite )
[write]
signal( sWrite )
```

## reader:

```
{ wait( sRead )
  r += 1
  if( r==1 )
    wait( sWrite )
    signal( sRead )
  [Read]
  { wait( sRead )
    r -= 1
    if( r==0 )
      signal(sWrite)
      signal( sRead ) }
```

- **Idea**
  - Only 1st reader waits for write semaphore
- **Works**
  - But might starve writers...
  - Think about how to fix
  - Solution:  
[http://en.wikipedia.org/wiki/Readers%20%93writers\\_problem](http://en.wikipedia.org/wiki/Readers%20%93writers_problem)

# Amdahl's law (1967)

- What's the maximal expected speed when parallelizing?
  - Let  $n$  be the number of threads
  - Let  $s$  be the fraction of the algorithm that is strictly serial ( $0 \leq s \leq 1$ )
  - Let  $T_n$  be the time it takes to run the algorithm with  $n$  threads
  - Then, optimally,

$$T_n = T_1 \times \left( s + \frac{1-s}{n} \right) \geq T_1 \times s$$

- And

$$\text{speedup} = \frac{T_1}{T_n} = \frac{1}{s + \frac{1-s}{n}} \leq \frac{1}{s}$$

Gene Amdahl



Gene Amdahl addressing a UW-Madison Alumni gathering, March 13, 2008

|             |                                                                                                 |
|-------------|-------------------------------------------------------------------------------------------------|
| Born        | November 16, 1922<br>Flandreau, South Dakota                                                    |
| Died        | November 10, 2015 (aged 92)<br>Palo Alto, California                                            |
| Nationality | American                                                                                        |
| Alma mater  | South Dakota State University<br>(B.S., 1948)<br>University of Wisconsin (M.S.;<br>Ph.D., 1952) |
| Known for   | founding Amdahl Corporation;<br>formulating Amdahl's law; IBM<br>360, 704                       |

# **Operating Systems (234123)**

## ***Deadlocks***

Dan Tsafrir (2025-05-18, and some of 2025-05-19)  
Partially based on slides by Hagit Attiya

# **Text @ OS notes book**

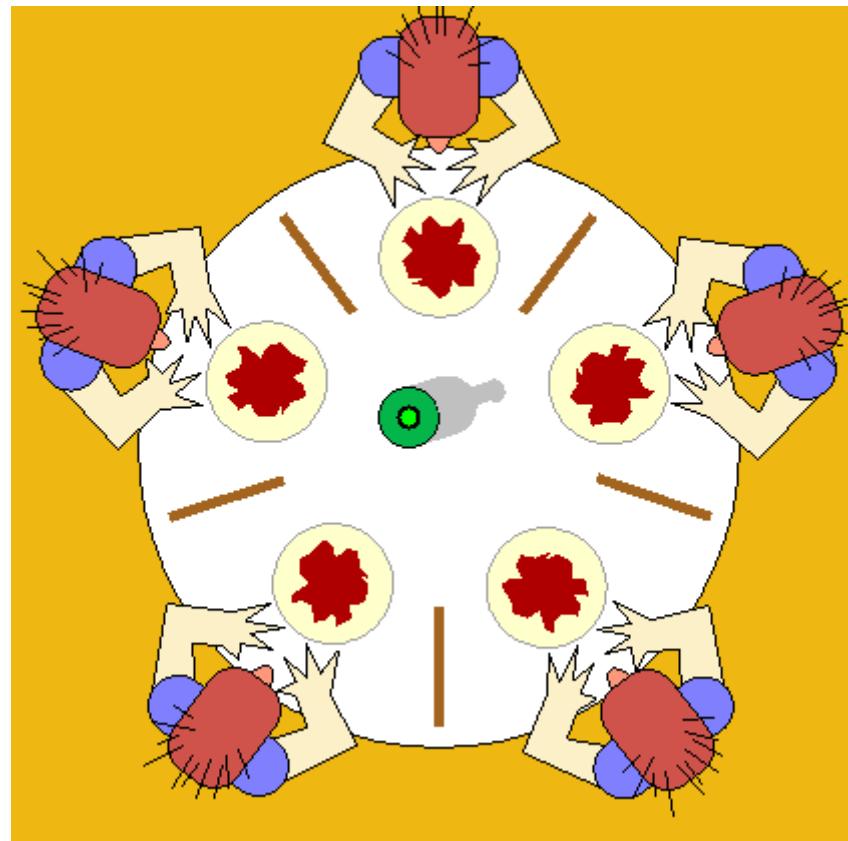
- **Much of the material appears in Section 3.2 in Feitelson's OS notes book**
  - Literature section in course homepage
- **In case this presentation and book conflicts**
  - As always, this presentation wins

# Intro

- **In the previous lecture**
  - We've talked about how to synchronize access to shared resources
- **When synchronizing, if we're not careful**
  - Our system might enter a deadlock state
- **The popular formal CS definition of a deadlock**
  - “A set of processes is deadlocked if each process in the set is waiting for an event that only a process in the set can cause”
- **Typically associated with synchronizing the use of resources**
  - Let's revise the definition accordingly
  - A set of processes is deadlocked if each process in the set is waiting for a resource held by another process in the set
- **“The dining philosophers problem”**
  - The canonical example in introductory OS lectures to demonstrate deadlocks

# Dining philosophers – rules

- Five philosophers are sitting around a round table, each with a bowl of Chinese food in front of him
- Between periods of meditation, they may start eating, whenever they want
- But there are only five chopsticks available, one between every pair of bowls -- and for eating Chinese food, one needs two chopsticks...
- When a philosopher wants to start eating, he *must* first pick up the chopstick to the **left** of his bowl; **then** the **right**

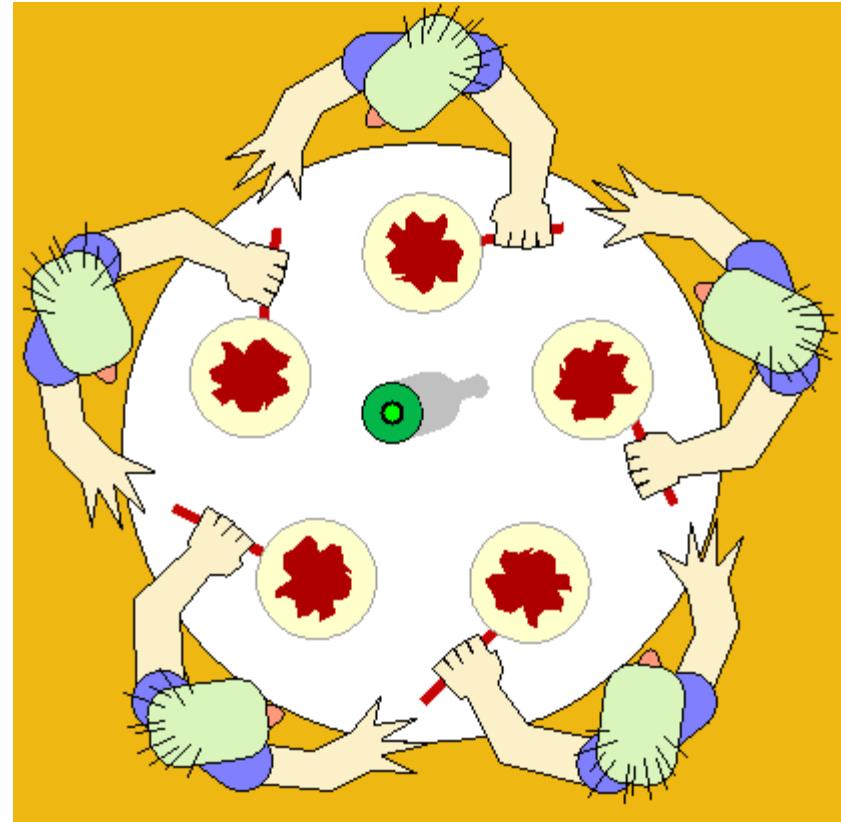


# Dining philosophers – naive solution

- **Semaphore for each chopstick**
  - semaphore\_t chopstick[5] // binary semaphores
  - (What if forks were places in the middle of the table and any philosopher would be able to grab any fork? Would we still need 5 semaphores?)
- **Naive (faulty) algorithm**
  - philosopher(i):  
while(1) do...
    - think for a while
    - wait( chopstick[i] ) // left
    - wait( chopstick[(i+1) % 5] ) // right
    - eat
    - signal( chopstick[(i+1) % 5] )
    - signal( chopstick[i] )

# Dining philosophers – problem

- All the philosophers become hungry at the exact same time
- They simultaneously pick up the chopstick to their left
- They then all try to pick up the chopstick to their right
- Only to find that those chopsticks have already been picked up (by the philosopher on their right)
- The philosophers then continue to sit there indefinitely, each holding onto one chopstick, glaring at his neighbor angrily
- They are deadlocked



# Resource type & instance(s)

- **Stuff that we may view as a **type** of a resource**
  - Physical: devices (or “portions” of devices), e.g.,
    - Disk / disk block
    - network controller (NIC)
    - Keyboard
    - Mouse
  - Logical: data structures, e.g.,
    - Process table / its individual task\_struct-s, node in a tree, ...
- **Sometimes, a resource type has only **one instance****
  - For example, typically, the keyboard
- **Sometimes, a resource type has **multiple instances****
  - Canonical example: each entry in the producer-consumer “ring”
    - E.g., resource type = image frame of video player
    - See previous lecture

# Resource type & instance(s)

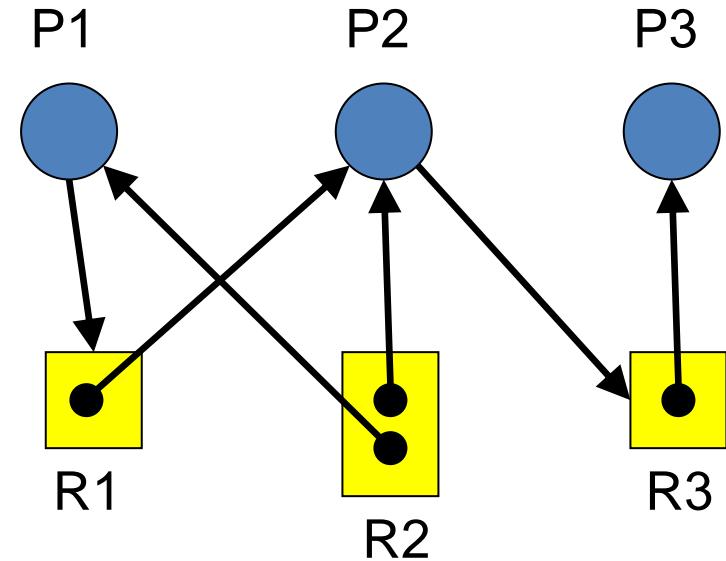
- **Assume we have**
  - N printers attached to a computer
- **Are they**
  - Instances of the same generic printer type?
  - Or N separate printer types?
  - Or something in between?
- **Depends on if their features differ**
  - Black & white vs. color
  - Duplex vs. simplex
  - ...

# Resource type & instance(s)

- **Lock is**
  - A synchronization construct that **reflects a single instance**
  - As by definition, it ensures mutual exclusion = only one task acquires it
- **Semaphore is**
  - A synchronization construct that reflects **single or multiple instances**
  - Depending on the maximal value of its ‘value’
    - Recall that a “binary semaphore” has semantics of a mutex lock

# Resource allocation graph

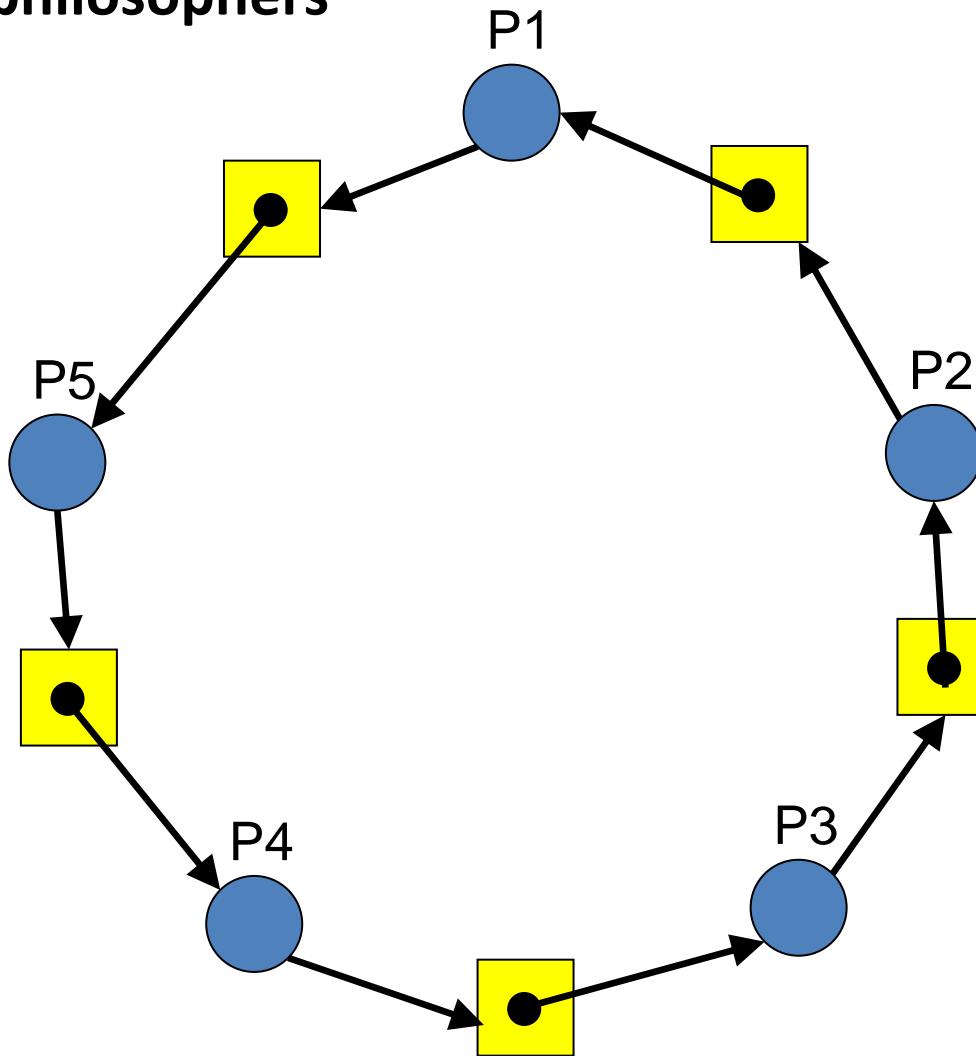
- When considering resource management
  - Convenient to represent system state with a directed graph
- 2 types of nodes
  - Process = round node
  - Resource type = square node
    - Within resource, each instance = a dot
- 2 types of edges
  - Request = edge from process to resource type
  - Allocation = edge from resource instance to a process



|           |                                              |
|-----------|----------------------------------------------|
| <b>P1</b> | Holds instance of R2.<br>Waits for R1.       |
| <b>P2</b> | Holds instances of R1 & R2.<br>Waits for R3. |
| <b>P3</b> | Holds instance of R3.                        |

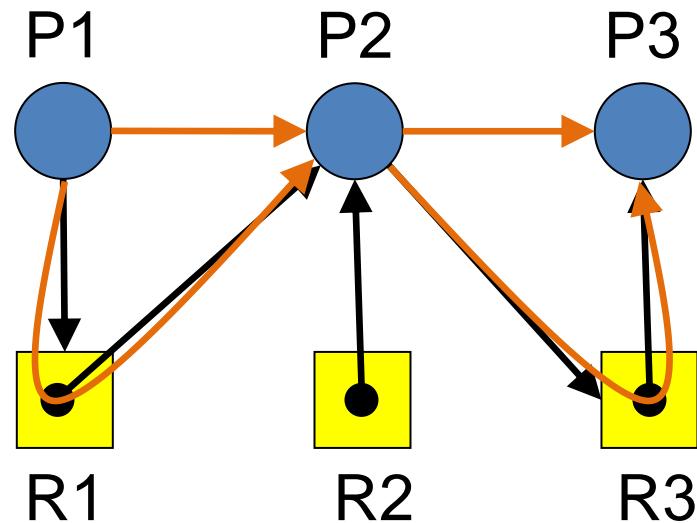
# Resource allocation graph

- Dining philosophers



# Resource allocation graph

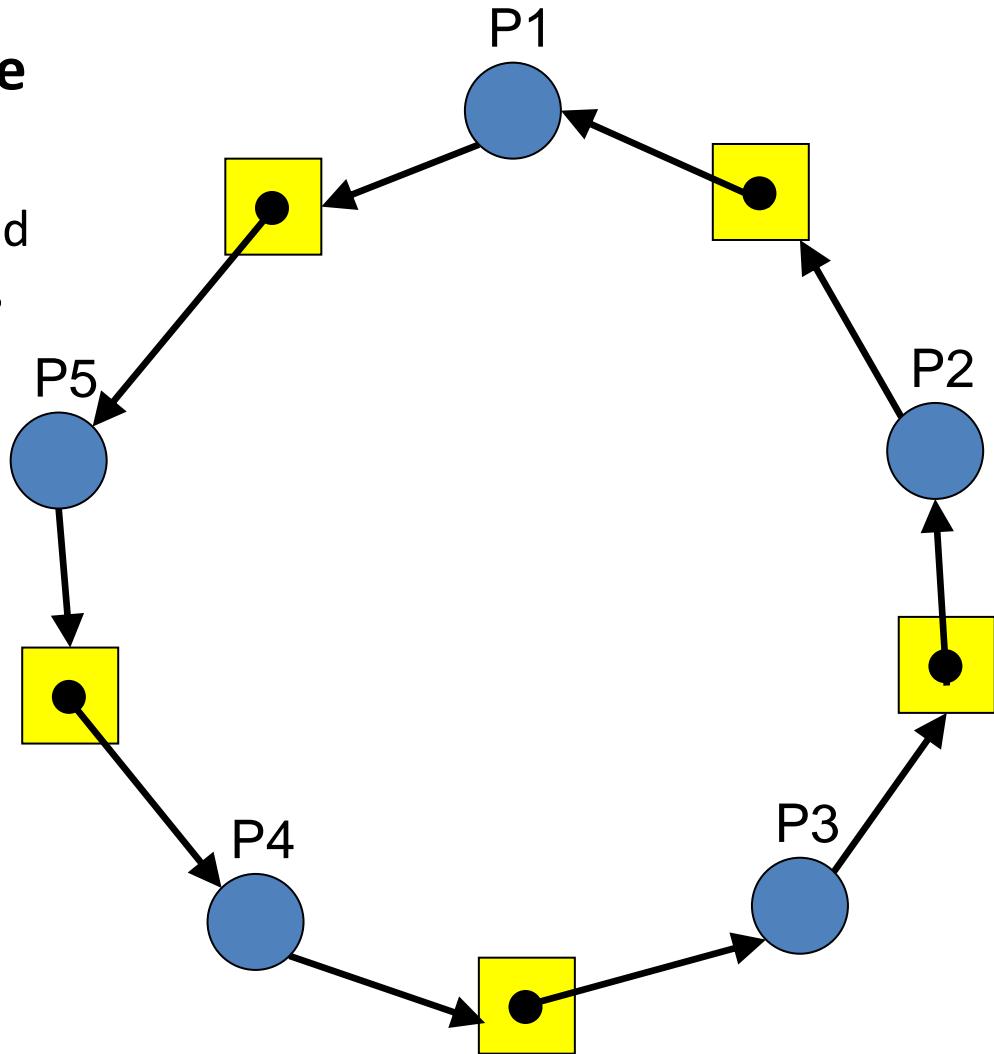
- When there's only one instance per resource type
  - Can simplify graph
  - By eliminating resources and only marking dependencies between processes



# Resource allocation graph

- When there's only one instance per resource type

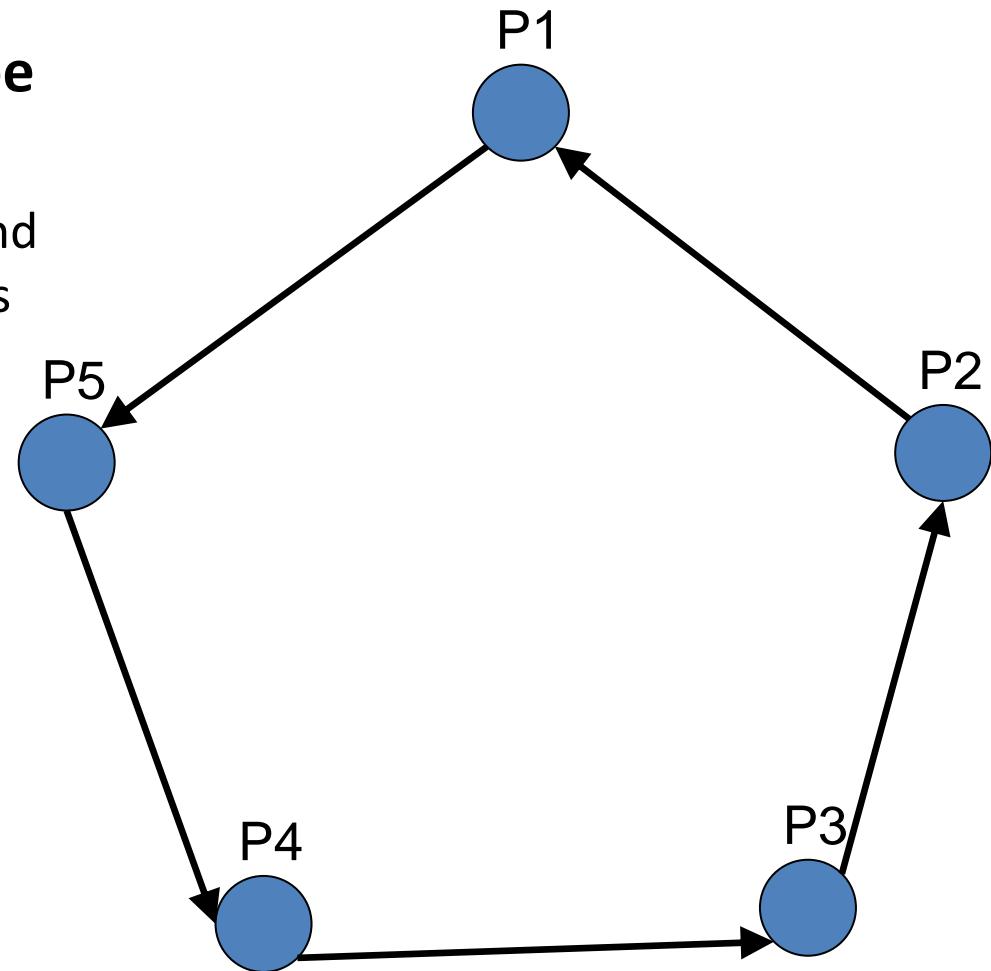
- Can simplify graph
- By eliminating resources and only marking dependencies between processes



# Resource allocation graph

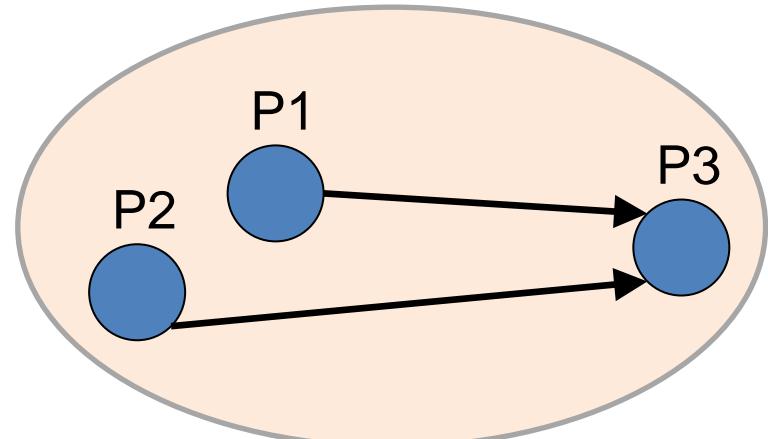
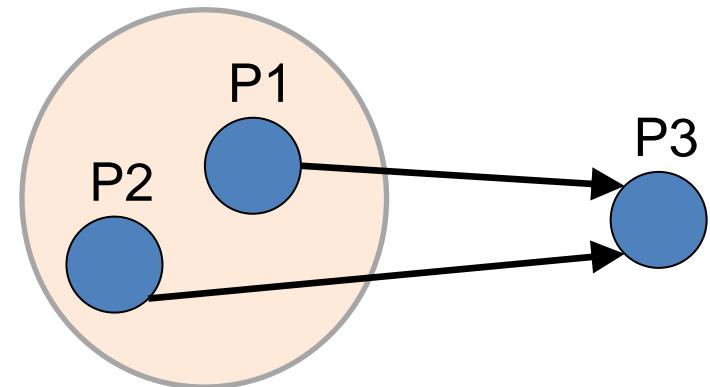
- When there's only one instance per resource type

- Can simplify graph
- By eliminating resources and only marking dependencies between processes



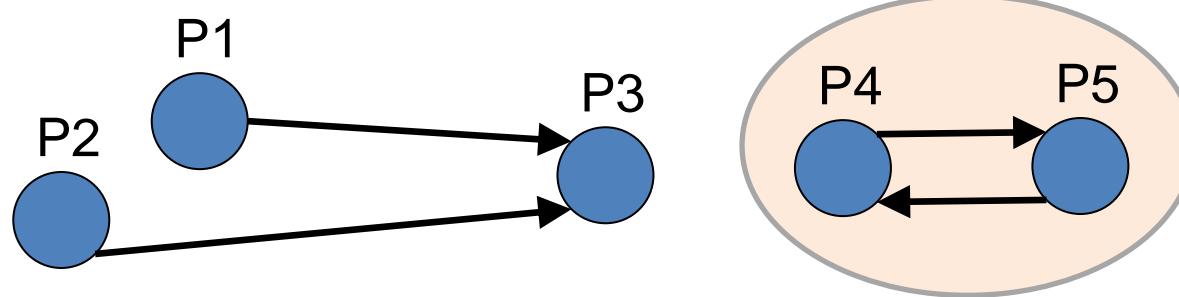
# Recall the formal definition of deadlock

- **Definition**
  - A set of processes is deadlocked if each process in the set is waiting for a resource held by another process in the set
- **Why “in the set”?**
  - No deadlock, even though every process in the set is waiting for a resource held by another process:
- **Why “each”?**
  - If including P3, then since P3 isn’t waiting for a resource held by another process => no deadlock:



# Recall the formal definition of deadlock

- **Definition**
  - A set of processes is deadlocked if each process in the set is waiting for a resource held by another process in the set
- **Can the set be a subset?**
  - Of course



# Necessary conditions for deadlock

- **All of these must hold for a deadlock to occur**
  1. Mutual exclusion
    - Some resource is (i) used by more than one process, but is (ii) exclusively allocated one process at a time (cannot be shared)
    - If used by only one process, or can be shared => can't deadlock
  2. Hold & wait
    - Processes may hold one resource and wait for another
    - If resources allocated atomically altogether => can't deadlock
  3. Circular wait
    - $P(i)$  waits for resource held by  $P((i+1) \% n)$  // some enumeration
    - Otherwise, recursively, there exists one process that need not wait
  4. No resource preemption
    - If resources held can be released (e.g., after some period of time), then can break circular wait

# **DEALING WITH DEADLOCKS**

# **Who's responsible for dealing with deadlocks?**

- **Typically, the OS doesn't do it for you**
- **Typically, it's you (the programmer)**
  - Who should be aware of the problem and deal with it

# 2 main approaches to cope with deadlocks

1. Design the system such that it is never allowed to enter into a deadlock situation
  - Example: this is how we'll acquire locks (soon)
2. Allow the system to experience deadlock, but use mechanisms to detect & recover
  - Example: memory exhaustion (assume no swap area)  
=> kill random process

Violate one of the four conditions

## “DEADLOCK PREVENTION”

# **“Prevention” = violate one of the 4 conditions**

- **All 4 conditions must hold for deadlock to occur**
  - So violating even one eliminates possibility of deadlock

# No “hold and wait” (#2)

lecture ended here

- **Instead of acquiring resources one by one**
  - Each process requests all resources it'll need at the outset
  - System can then either provide all resources immediately
  - Or block process until all requested resources are available
- **Con**
  - Processes will hold on to their resources for more time than they actually need them
  - Which limits concurrency and hence performance
- **Refinement**  
**(allow compute phases with different resources)**
  - Before a process issues a new (atomic) request for resources
  - It must release all resources it currently holds
    - (After bringing system to consistent state, of course)
  - Risking the resources will be allocated to other processes

# No “no resource preemption” (#4)

- **Under some circumstances, for some resources**
  - Can choose a victim process and release its resources
  - For example, if there isn't enough memory, can
    - Either write the victim's state to disk and release all its memory
    - Or kill it

# No “mutual exclusion” (#1)

- Resource is sharable, for example...
- Can implement some data structures (e.g., linked list)
  - Multiple threads can concurrently use of the data structure
  - Without using *any* form of explicit synchronization
    - No spinlocks, no semaphores, etc.
- How?
  - Using only HW-supported atomic operations (such as test-and-set)
  - These algorithms are sometimes (also) called “lock free”
    - Overloaded term – Not to be confused with another definition of “lock free” (= “some thread always makes progress”)
- Mature field
  - Books on how to do it (proving implementations are correct)
  - Ready libraries can be used w/o exposing programmer to the complexities
  - Major con: composition of operations is problematic

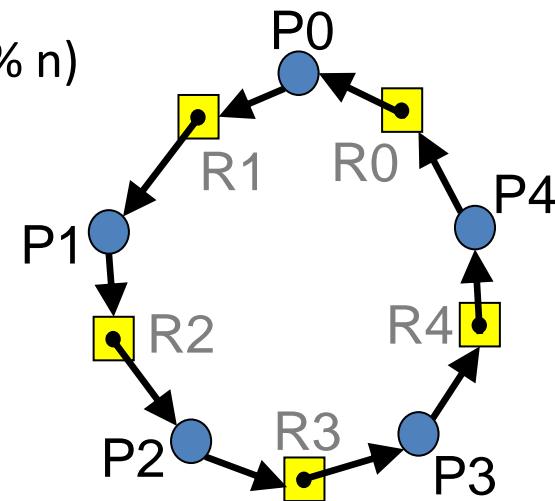
# No “circular wait” (#3)

- Arguably, the most usable flexible way to prevent deadlocks
- How it's done
  - All resources are numbered in one sequence
    - $Ord(\text{printer})=1, Ord(\text{scanner})=2, Ord(\text{lock\_x})=3, Ord(\text{lock\_y})=4, \dots$
  - Processes must request resources in increasing  $Ord()$  order
  - Namely, a process holding some resources can only request additional resources that have strictly higher numbers
  - A process that wishes to acquire a resource that has a lower order
    - Must first release all the resources it currently holds

# No “circular wait” (#3)

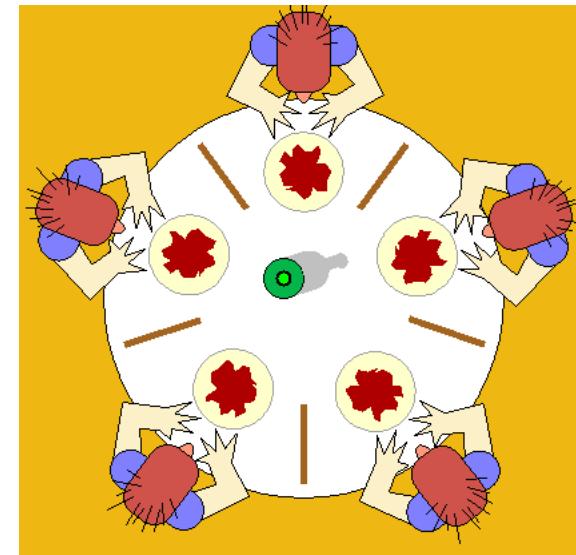
- **Proof that it works**

- Assume by contradiction that there exists a cycle
  - Without loss of generality, assume processes are numbered by this cycle
    - $P(i)$  waits for resource  $R((i+1) \% n)$  held by  $P((i+1) \% n)$  ( $i = 0, 1, \dots, n-1$ )
  - Let  $M(i)$  be
    - The maximal  $\text{Ord}()$  amongst the resources that  $P(i)$  holds
  - Thus, because
    - **Each  $P(i)$  acquires resources in order**, and
    - $P(i)$  waits for  $P((i+1)\%n)$ , which holds  $R((i+1) \% n)$
  - We get that
    - $M(i) < R((i+1)\%n) \leq M((i+1) \% n)$   
 $\Rightarrow M(i) < M((i+1) \% n)$   
 $\Rightarrow M(0) < M(1) < M(2) < \dots < M(n) < M(0)$   
 $\Rightarrow M(0) < M(0)$   
 $\Rightarrow$  contradiction



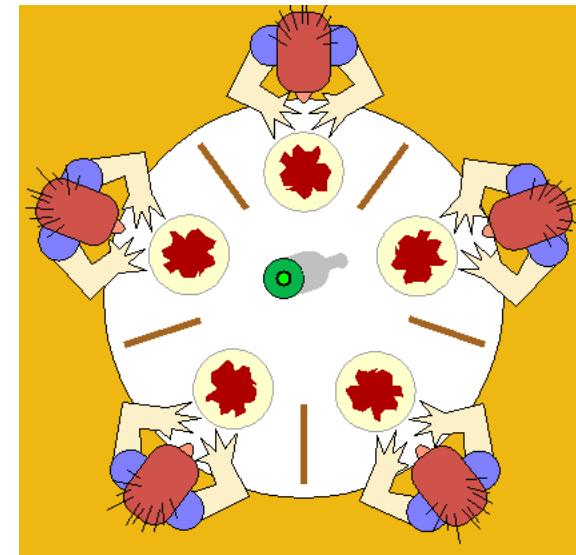
# No “circular wait” (#3)

- Let's solves the dining philosophers' problem in this way
- Recall that we numbered the chopsticks as 0...4; let's lock the chopsticks in this order
- Following is the code of Philosopher i
  - if (  $i < 4$  ) //  $i$  can be 0...3  
    wait( chopstick[ $i$ ] )  
    wait( chopstick[ $i+1$ ] )
  - else //  $i==4$   
    wait( chopstick[0] ) // smaller  
    wait( chopstick[4] ) // bigger
  - eat
  - if(  $i < 4$  )  
    signal( chopstick[ $i+1$ ] )  
    signal( chopstick[ $i$ ] )
  - else  
    signal( chopstick[4] )  
    signal( chopstick[0] )



# No “circular wait” (#3)

- Let's solves the dining philosophers' problem in this way
- Recall that we numbered the chopsticks as 0...4; let's lock the chopsticks in this order
- Following is the code of Philosopher i
  - ```
int lo = (i<4) ? i      : 0 // 4's lo==0
int hi = (i<4) ? i+1    : 4 // 4's hi==4
wait( chopstick[lo] )
wait( chopstick[hi] )
eat
signal( chopstick[hi] )
signal( chopstick[lo] )
```

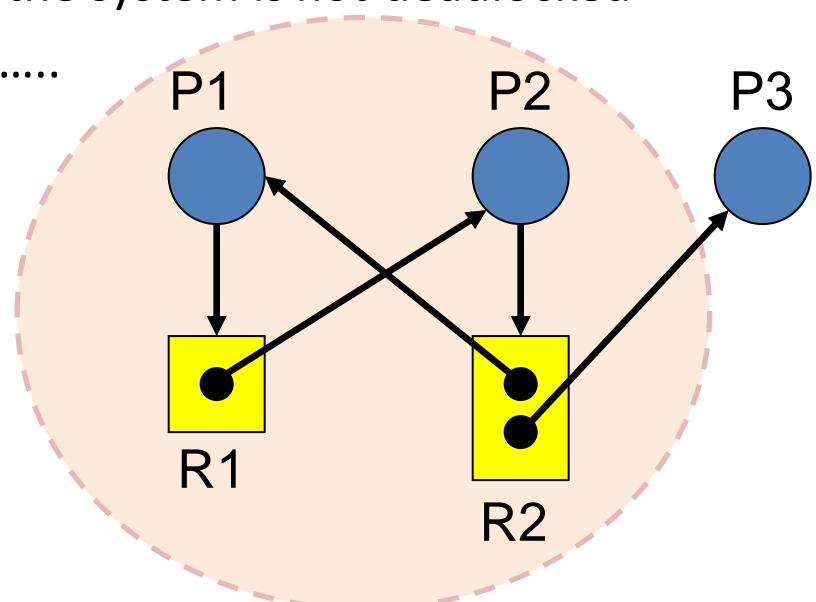


Finding & resolving deadlock via some resource-graph algorithm

# “DEADLOCK DETECTION & RECOVERY”

# Deadlock detection

- **If there's only one instance of each resource type**
  - Search for a cycle in the (simplified) resource allocation graph
    - Found  $\Leftrightarrow$  deadlock
- **In the general case, which allows multiple instances per type**
  - Necessary conditions for deadlock  $\neq$  sufficient conditions for deadlock
  - A graph can have a cycle while the system is *not* deadlocked
  - Example.....
- **Can nevertheless detect deadlocks in general case**
  - But algorithm outside of our scope



# Recovery from deadlock after detection

- After a deadlock has been detected (previous slide)
  - Need to somehow recover
- This could be done by terminating some of the processes
  - Until deadlock is resolved
  - Sometimes make sense, sometimes doesn't
- Or it could be done by preempting resources
  - Of deadlocked processes
  - Sometimes make sense, sometimes doesn't
- Finding a minimal (“optimal”) set of processes to terminate or resources to preempt is a hard problem

# Ambiguity between deadlock “prevention” & “detection & recovery”

- **In the literature**
  - “Prevention” means
    - System not allowed to deadlock by *always* violating 1 of 4
  - “Detection & recovery” means
    - Can deadlock, then detect, then recover
- **In practice, a somewhat problematic distinction, as**
  - We tend to use the same example – memory preemption – for both
    - “Detection & recovery”, and
    - “Resource preemption” (necessary condition #4 of “prevention”)
- **It seems**
  - “Detection & recovery” largely refers to the algorithmic problem
    - Of finding a minimal set to kill/preempt in the resource graph

Banker's algorithm (by Dijkstra)

# “DEADLOCK AVOIDANCE”

# Banker's algorithm

- **Rules**
  - $n$  processes
  - $k$  resource types (each type may have 1 or more instances)
  - Upon initialization, each process declares the maximal number of resource-instances it'll need for each resource type
  - While running, OS maintains how many resources are currently used by each process
  - And how many resource instances per type are currently free
- **Upon process resource allocation request**
  - OS will allocate iff allocation isn't “dangerous”, namely
    - If it knows for a fact that it'll be able to avoid deadlock in the future
  - Otherwise, the process will be blocked until a better (“safer”) time
  - Algorithm is thus said to be conservative, as there's a possibility for no deadlock even if allocation is made, but OS doesn't take that chance
- **Upon process termination (assume process do terminate)**
  - Process releases all its resources

# Banker's algorithm

- “Safe state”
  - System state whereby we’re sure that all processes can be executed, in a certain order, one after the other, such that each will obtain all the resources it needs to complete its execution
  - By ensuring such a sequence exists after each allocation  
=> we avoid deadlock
- Banker’s data structure
  - $\text{max}[p] = (m_1, m_2, \dots, m_k)$  = max resource requirements for process p
  - $\text{cur}[p] = (c_1, c_2, \dots, c_k)$  = current resource allocation for process p
  - $R = (r_1, r_2, \dots, r_k)$  = the current resource request (for some process p)
  - $\text{avail} = (a_1, a_2, \dots, a_k)$  = currently available (free) resources (global)
- Example
  - $\text{max}[p] = (3,0,1)$ ,  $\text{cur}[p] = (3,0,0)$
  - Note that  $\text{max}[p] \geq \text{cur}[p]$  always holds /\* compare by coordinates \*/

# Banker's algorithm

- Tentatively *assume* that request R was granted to process q
  - $\text{cur}[q] += R$  // vector addition
  - $\text{avail} -= R$  // vector subtraction
- Check if “safe state” (= can satisfy all processes in some order)
  - initialize P to hold all non-terminated process
  - while( P isn't empty ) {
    - found = false
    - for each p in P { // find one p that can be satisfied
      - if(  $\text{max}[p] - \text{cur}[p] \leq \text{avail}$  ) // p's biggest request
        - $\text{avail} += \text{cur}[p]$  // pretend p terminates
        - $P -= \{p\}$
        - found = true
    - }
    - if( ! found ) return FAILURE
  - }
  - return SUCCESS

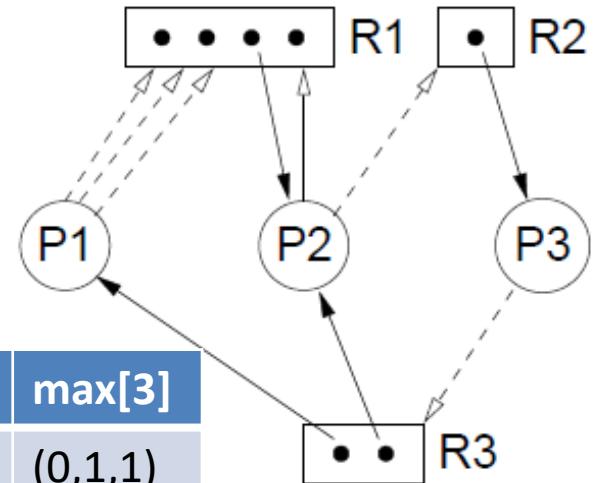
# Banker's algorithm – runtime complexity

- $O(n^2)$ 
  - Even though number of possible orders is  $n!$
  - Since resources increase monotonically as processes terminate
  - As long as it's possible to execute any set of processes
    - Execution order not important
    - (There is never any need to backtrack and try another order)

# Banker's algorithm – example

- Initial system state

cur[1]	cur[2]	cur[3]	avail	max[1]	max[2]	max[3]
(0,0,1)	(1,0,1)	(0,1,0)	(3,0,0)	(3,0,1)	(2,1,1)	(0,1,1)



- P1 requires instance of R1 [R = (1,0,0)]

- Granting the request yields

cur[1]	cur[2]	cur[3]	avail	max[1]	max[2]	max[3]
(1,0,1)	(1,0,1)	(0,1,0)	(2,0,0)	(3,0,1)	(2,1,1)	(0,1,1)

- Safe, because there are enough R1 instance so that P1's max additional request can be satisfied:  $\text{max}[1]-\text{cur}[1]=(2,0,0)$ ; so after P1's termination

	cur[2]	cur[3]	avail		max[2]	max[3]
	(1,0,1)	(0,1,0)	(3,0,1)		(2,1,1)	(0,1,1)

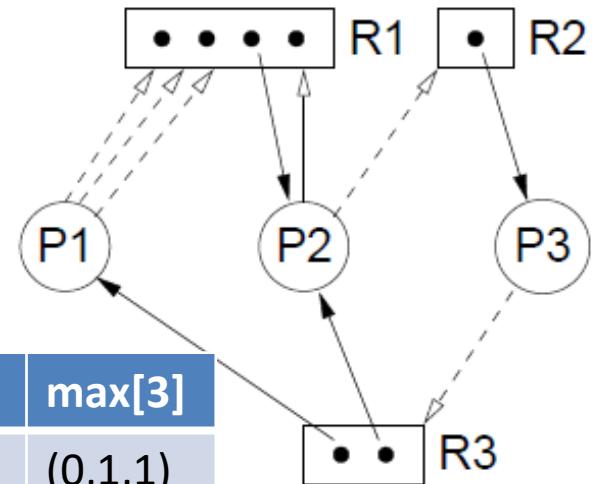
# Banker's algorithm – example

- Copied from previous slide

	cur[2]	cur[3]	avail		max[2]	max[3]
	(1,0,1)	(0,1,0)	(3,0,1)		(2,1,1)	(0,1,1)

- Not enough to satisfy P2 (why?), but can satisfy P3
  - $R3 = (0,1,1) - (0,1,0) = (0,0,1)$  ( $\leq \text{avail} = (3,0,1)$ )

	cur[2]	cur[3]	avail		max[2]	
	(1,0,1)		(3,1,1)		(2,1,1)	



# Summary: ways to deal with deadlocks

## 1. Deadlock “prevention”

- Violate one of the 4 conditions necessary for deadlock
- Deadlock can't happen
- E.g., by ordering resources & acquiring from smallest to biggest

## 2. Deadlock “avoidance”

- Banker's algorithm
- Requires full knowledge about available & requested resources
- System stays away from deadlocks by being careful on a per resource-allocation decision basis

## 3. Deadlock “detection & recovery”

- Allow system to enter deadlock state, but put in place mechanisms that can detect, and then recover from this situation
- Typically refer to the algorithmic resource-graph problem

# **Operating Systems (234123)**

## ***Networking***

Dan Tsafrir (2025-05-18 one hour; 2025-05-19)

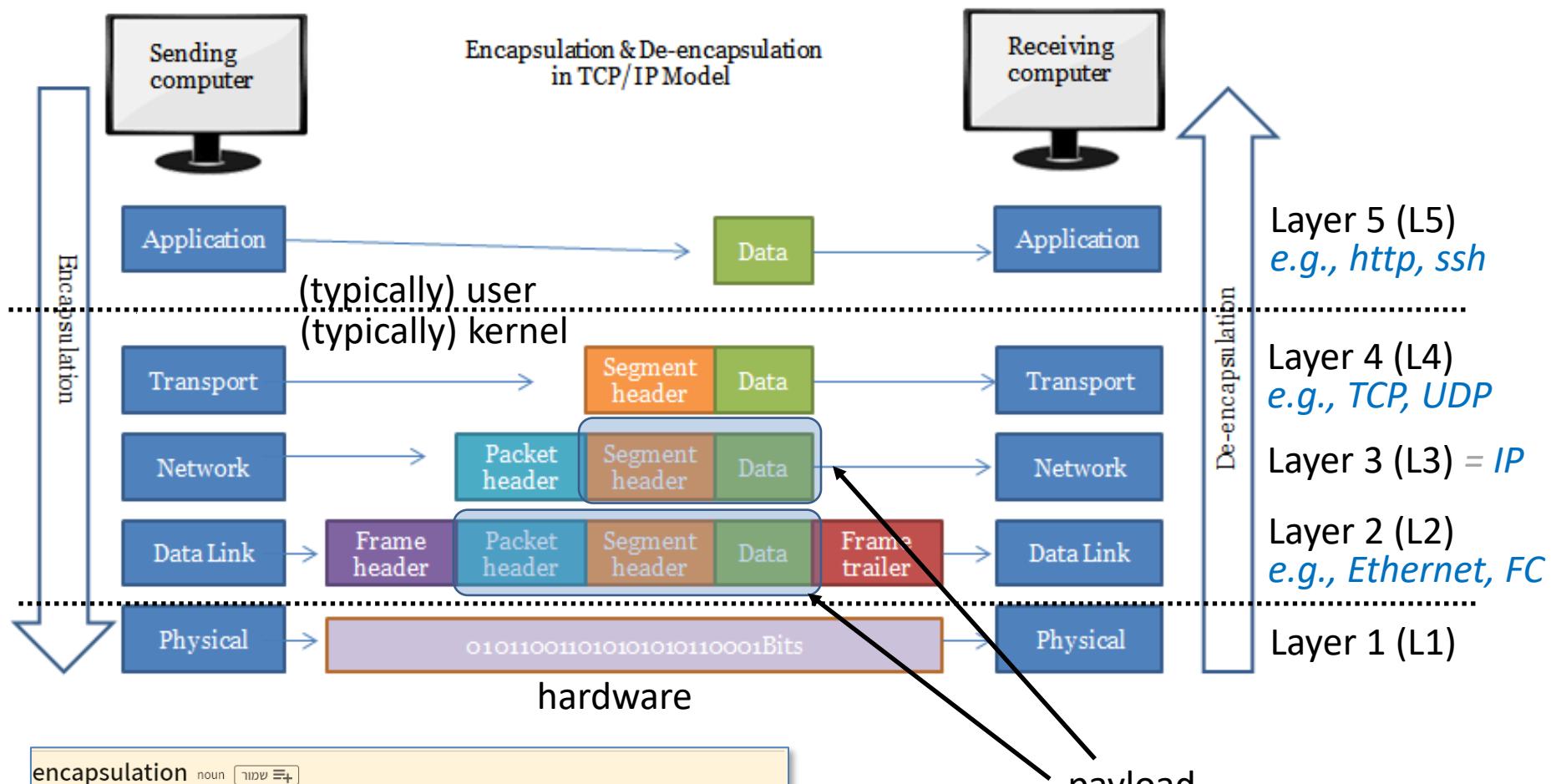
Preface

# **NETWORK PROTOCOLS**

# Protocol definition

- **Communicating parties are**
  - Host machines (computers) & processes
- **A network communication protocol is a set of rules defining**
  - Format & order of messages sent & received
  - Action taken upon message transmission & reception
- **All network communication activity**
  - Is governed by protocols

# Network protocol stack consists of layers



encapsulation noun + שמר

כמיהה, אטימה, סגירה (כברון קפיטולה); היהת סגור או אוטם; עטוף (עטיפת מסמך בשתי שכבות ניילון שקופות וחתוכמות); (מחשבים) כמוס, אנקפיטולציה, הقلלה נתונים משכבה פרוטוקול עליונה בתוך שכבה פרוטוקול ונמכה יותר

Higher-level view on

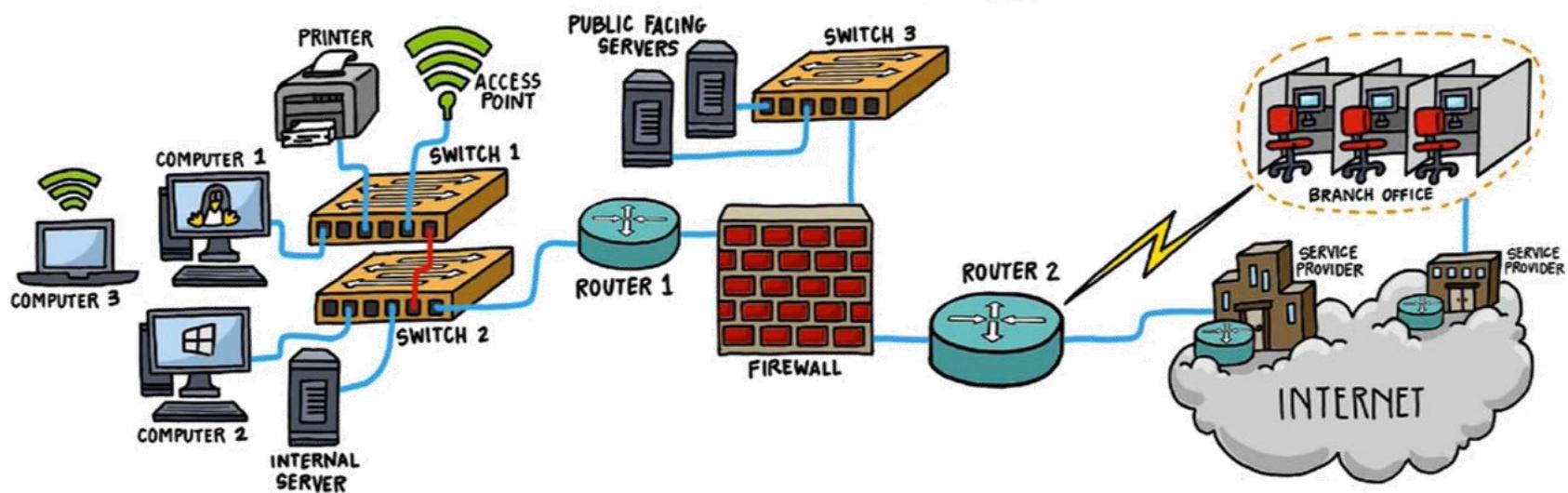
# **APPLICATION-LAYER (L5) & TRANSPORT-LAYER (L4) PROTOCOLS**

# Application-layer (L5) protocols

- **When the protocol is determined by the app**
  - Many examples, here are a few...
- **Standard** (protocol determined by multiple organizations)
  - HTTP, HTTPS (web browsing)
  - SMTP, IMAP, POP (email)
  - VoIP (voice)
  - iCalendar (scheduling)
  - NFS (distributed filesystem)
  - SSH (secure shell)
  - Bitcoin (cryptocurrency)
- **Proprietary** (single organization; still, can be open)
  - Microsoft Exchange (mail & scheduling)
  - Skype, Zoom (mostly video conferencing)
  - WhatsApp, Telegram (mostly text messaging)
- **And you can easily invent your own, as needed**

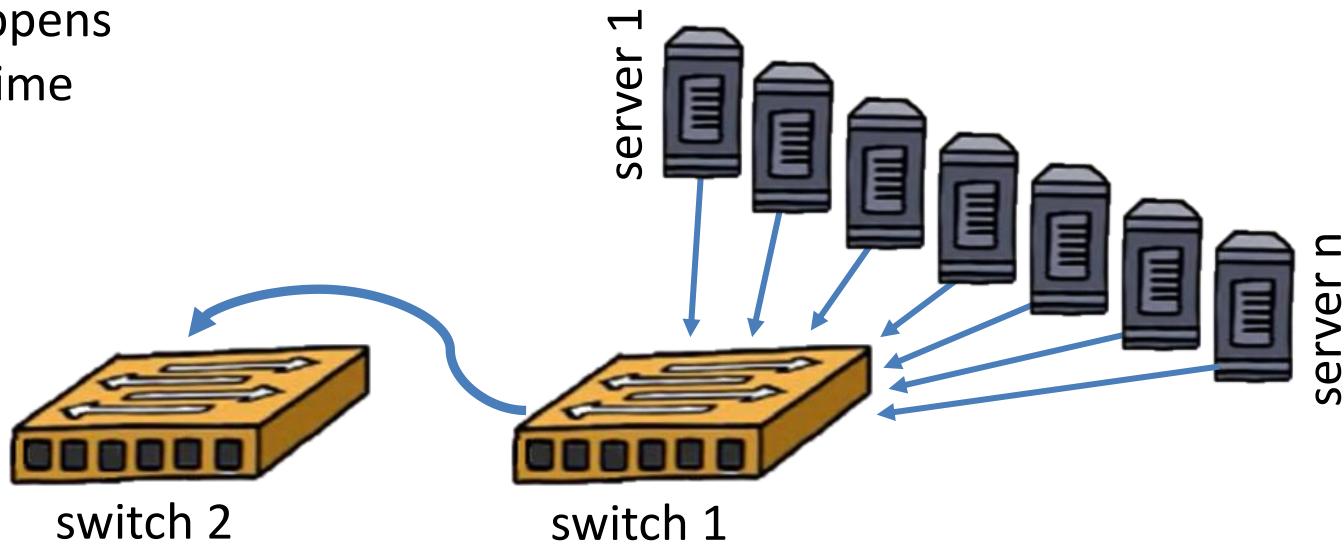
# Lossiness

- When considering protocols, we should be aware that
  - Networks are inherently lossy
  - What's sent won't necessarily reach its destination
- For example, because
  - Bits specifying destination get flipped due to electrical problems
  - Network elements (routers, switches, APs, endpoints) malfunction
  - Network cables get severed



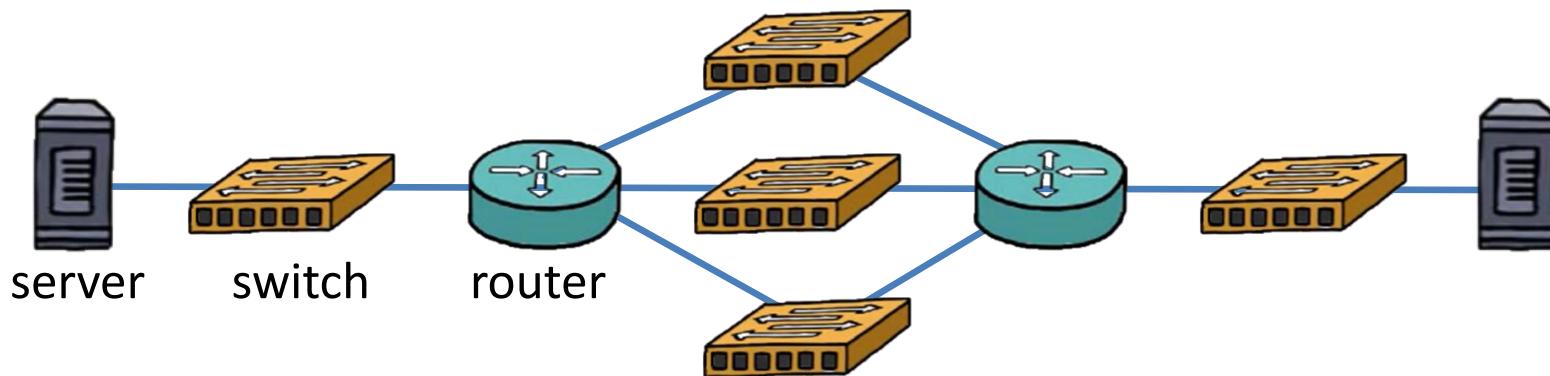
# Lossiness

- **But more frequently, loss occurs because**
  - Memory buffer space in network elements temporally runs out
- **For example, consider the incast problem**
  - Where many elements simultaneously send data to one element, beyond its processing / bandwidth capacity
- **Over the net**
  - Loss happens  
*All* the time



# Reordering

- **Also, when considering protocols, should be aware that**
  - Networks may mess up order of messages
- **For example, because**
  - Multiple paths between source & destination may exist
  - Messages may get sent through different paths



# Transport-layer (L4) protocols

- **Protocols in this layer**
  - Provide host-to-host communication service for processes
  - (Typically) implemented by the OS
    - Why would we want to user to implement it?
      - Example: DPDK (= Data Plane Development Kit)
    - Why would we want the NIC (hardware) to implement it?
      - Example: RDMA (= remoted DMA)
- **Two L4 protocols account for vast majority of internet traffic**
  - TCP (usually) & UDP (occasionally)
    - Implemented by all OSes

# TCP (transmission control protocol)

- **The protocol that cares...**
  - It **cares about data loss & reordering**  
(acronym should've been “Transmission that **Cares** Protocol” 😊 )
- **Provides**
  - Stream of bytes abstraction, namely
  - It ensures that all bytes arrives, in order, to the receiving app
- **Said to be “connection-oriented”**
  - A communication ‘session’ between the two parties must be negotiated/established before data transmission can begin
- **How does TCP implement its nice properties?**
  - Here’s an *oversimplified* explanation,  
to provide intuition,  
in a nutshell...

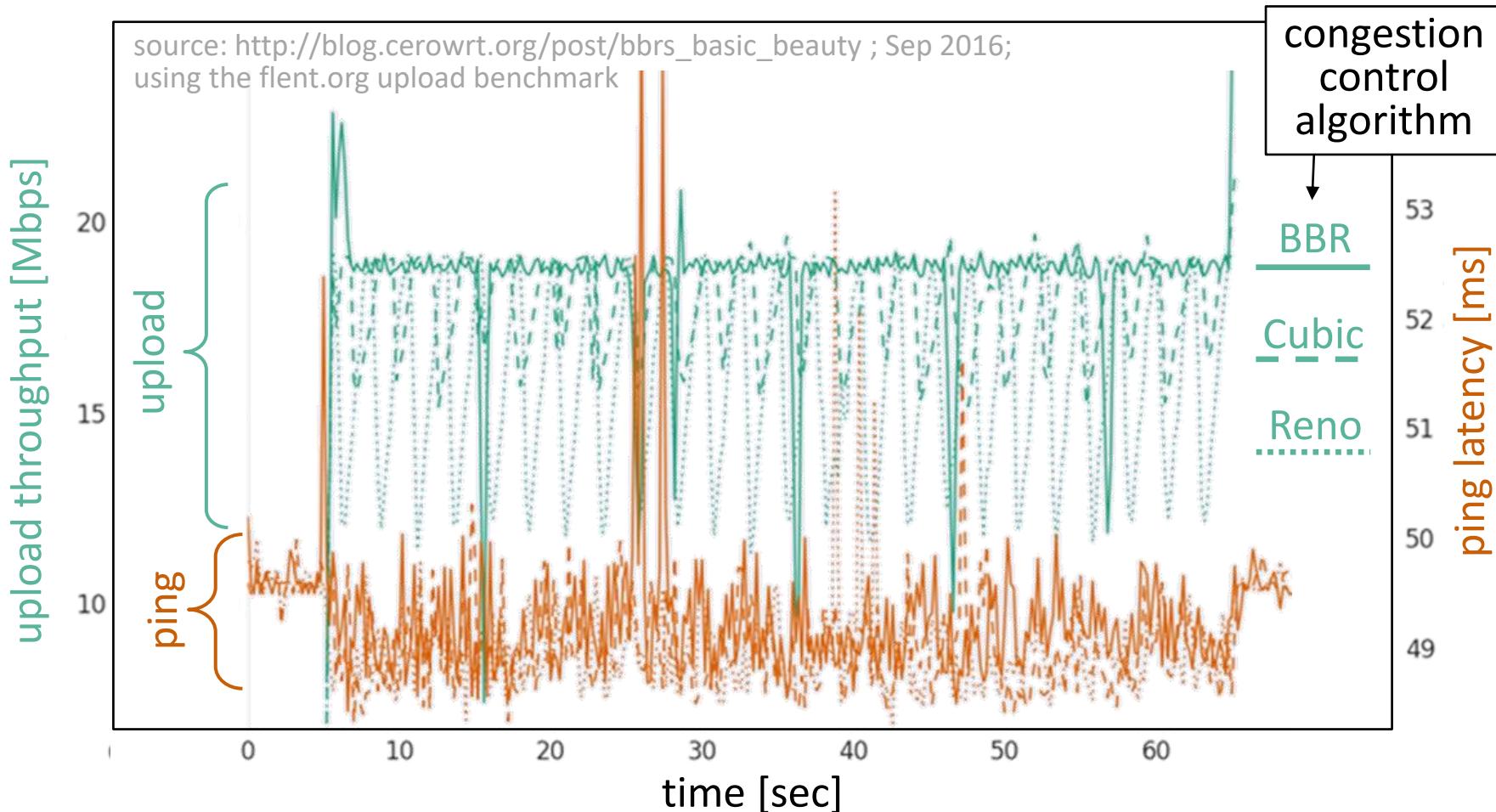
# TCP (transmission control protocol)

- **Sender**
  - Split bytes into contagious chunks (called “segments”)
    - Each chunk specifies which bytes [from ... to]
  - Keep chunks around
    - Until receiver acknowledges receiving them
  - Resend un-ack-ed chunks
    - That haven’t been ack-ed for some time, or
    - That the receiver says it’s missing
  - Slow down transmission rate
    - When noticing sent chunks don’t reach receiver
    - Gradually speed up otherwise
  - **Congestion control** = changing rate in response to drops
    - Tries to address buffering problem
      - [https://en.wikipedia.org/wiki/TCP\\_congestion\\_control](https://en.wikipedia.org/wiki/TCP_congestion_control)

# TCP (transmission control protocol)

- **Receiver**
  - Send ack messages
    - Upon received chunks
  - Ask sender to resend
    - Chunks that appear to be missing for some time
  - “Advertise” free buffer space
    - Sender is forbidden to send more
  - **Flow control:** advertising
    - Receiver **controls** sender transmission rate, thereby
    - Preventing its own buffer **overflow**
    - So that sender won’t transmit too much too fast
  - Deliver bytes in order to receiving user-level app

# TCP (transmission control protocol)



TCP “sawtooth” behavior with various congestion control algorithms; occurs because drops are used to sense congestion. (BBR proposed by Google in 2016.)

# UDP (user datagram protocol)

- **The protocol that doesn't care**
  - Neither about data loss nor about message reordering
  - Best effort service
- **Provides**
  - “Datagram” abstraction (as opposed to “data-stream” or just “stream”)
    - Datagram = chunk of bytes (still called “segment” here)
  - Chunks might get lost or be delivered out-of-order
    - But per-chunk bytes integrity is supported (with checksum)
  - Apps decide if they’re okay with that
- **Compared to TCP**
  - Simpler, lower latency, no congestion control (so can blast away)
- **Said to be “connectionless”**
  - No negotiation to establish communication ‘session’
  - Each chunk handled independently of others

Higher level view on

# **IP, THE NETWORK-LAYER (L3) PROTOCOL**

# Domain & host names

- Computers are associated with hierarchical human-readable “domain” names, sometimes referred to as host names
  - [www.cs.technion.ac.il](http://www.cs.technion.ac.il), [csa.cs.technion.ac.il](http://csa.cs.technion.ac.il), [csm.cs.technion.ac.il](http://csm.cs.technion.ac.il)
    - Each of these is a name of a *single* host machine
    - So, they’re indeed “host names”, but...
  - [www.google.com](http://www.google.com), [www.amazon.com](http://www.amazon.com), [www.cnn.com](http://www.cnn.com), [www.ynet.co.il](http://www.ynet.co.il)
    - Each of these is backed by multiple host machines, and...
  - [hagit.net.technion.ac.il](http://hagit.net.technion.ac.il), [benny.net.technion.ac.il](http://benny.net.technion.ac.il)
    - Each of these identifies a website in host net.technion.ac.il
    - So “domain” is a more general term
  - [https://en.wikipedia.org/wiki/Domain\\_name](https://en.wikipedia.org/wiki/Domain_name)
- In this lecture, we’ll typically use the term host name assuming
  - That it corresponds to a single host machine

# IP addresses

- **Domain names are only for humans**
  - They're not actually used to transmit data around
- **Instead, each host is mapped to a 32-bit IP address**
  - Obtained via a domain name resolution protocol (called [DNS](#); see later)
  - Caveat: for us, “IP” usually means [IPv4](#) = “Internet Protocol version #4”
    - There’s also [IPv6](#) (128-bit addresses), which we’ll mostly ignore
- **Each IP address correspond to a single host**
  - Unlike a domain name, which may correspond to multiple hosts
    - E.g., if its DNS is set to balance the load and resolve to multiple IPs
  - The opposite is *not* true
    - One host may be associated with multiple IPs (e.g., when it has multiple networking devices, such as wireless and wired)
  - But for simplicity, let’s ignore all that for the time being and
    - Assume that there’s a 1-to-1 host-to-IP mapping

# IP addresses

- Presented as four 8-bit decimal octets separated by dots
  - For example, the IP of [csa.cs.technion.ac.il](http://csa.cs.technion.ac.il) is

Notation	IP address
Decimal	2219057153 (not terribly convenient)
Binary ( $8 \times 4 = 32$ bits)	 10000100   01000100   00100000   00000001
Dot-decimal	<a href="http://132.68.32.1">132.68.32.1</a> (a bit more convenient)

- The protocol that implements IP addresses is... IP
  - (Recall that IP = internet protocol)
  - It's the network-layer (L3) protocol
  - On top of which TCP & UDP (L4) are implemented
    - IP identifies host machines
    - TCP/UDP identify individual communication channels of processes on the host, as we'll see next

lecture ended

How computers communicate – programmer's perspective

# **SOCKETS & PORTS (BACK TO L4), CLIENT-SERVER**

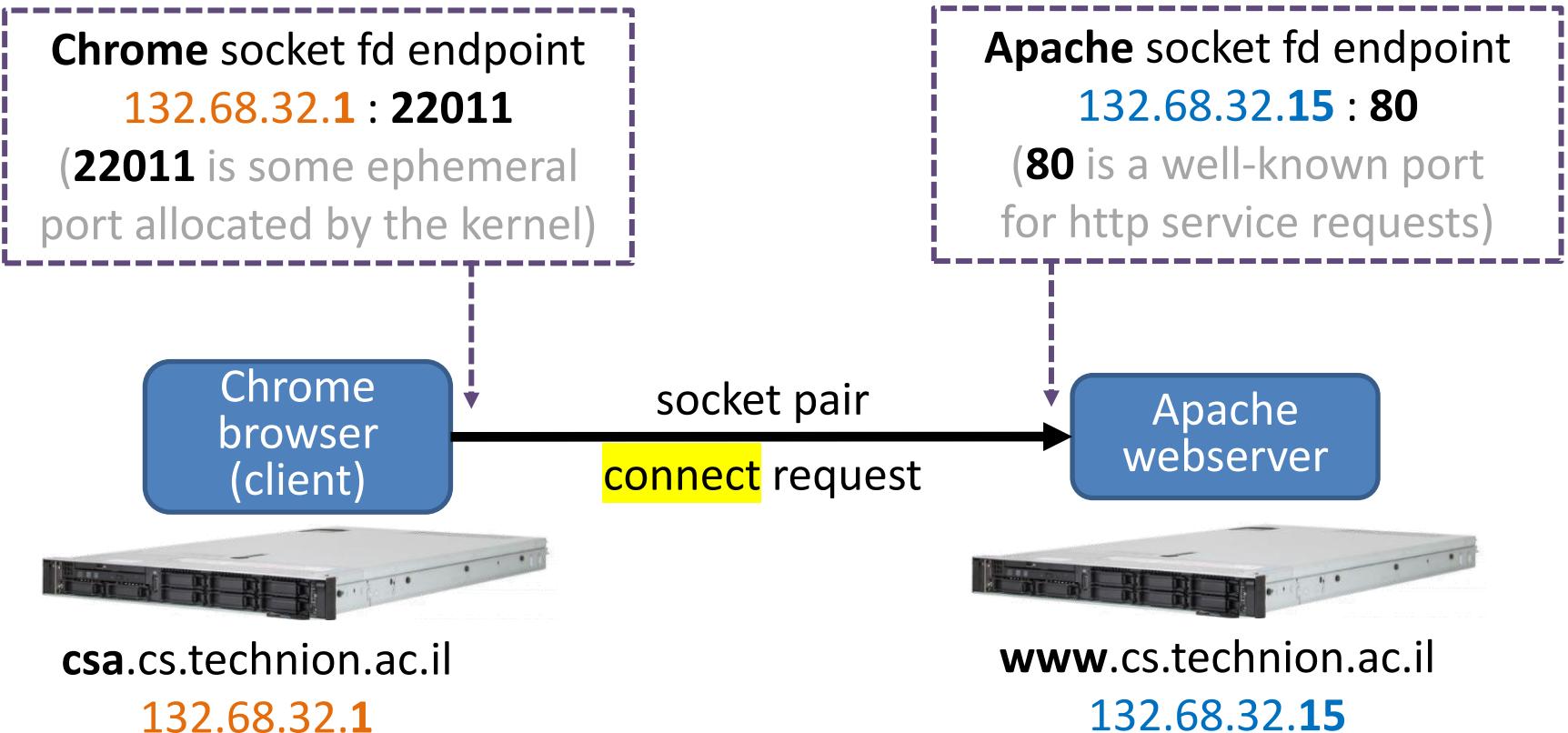
# Sockets

- **The parties that communicate across the network are**
  - Apps that run on hosts (e.g., browser, http server)
- **They communicate through**
  - Socket file descriptors, created via the `socket()` syscall, instead of `open()`
  - A `sockfd` constitutes a **communication endpoint**
- **read()-ing and write()-ing through socket fds**
  - Translate to receiving & sending data (duplex)
- **There's also more specific system calls (h/w: browse man)**
  - `ssize_t send (int sockfd, const void *buf, size_t len, int flags)`
  - `ssize_t recv (int sockfd, void *buf , size_t len, int flags)`
- **And their scatter-gather versions (h/w: browse man)**
  - `ssize_t sendmsg (int sockfd, const struct msghdr *msg, int flags)`
  - `ssize_t recvmsg (int sockfd, struct msghdr *msg , int flags)`

# Ports

- **On the same host, there can be**
  - Multiple communicating processes, each utilizing multiple sockfds
  - IP addresses aren't a sufficient identifier for transmitted data chunks
- **To disambiguate, each sockfd is associated with a**
  - Port, unsigned **16-bit integer** that identifies the sockfd
  - Every transmitted chunk is associated with IPaddress + port
- **Ports can be either**
  - Ephemeral (לטני; קוצר ימוי) = dynamically allocated by the kernel, or
  - Well-known = predetermined standard/known values
    - Ports  $\leq 1023$  are “reserved” (for privileged processes)
- **For example, http & https traffic flows via ports 80 & 443**
  - `http://www.google.com`  $\Leftrightarrow$  `http://www.google.com:80`
  - `https://www.google.com`  $\Leftrightarrow$  `https://www.google.com:443`
- **See more well-known ports in**
  - [https://en.wikipedia.org/wiki/List\\_of\\_TCP\\_and\\_UDP\\_port\\_numbers](https://en.wikipedia.org/wiki/List_of_TCP_and_UDP_port_numbers)

# Example



# Socket unique identifier: 5-tuple

- By, e.g., RFC 6146

<https://datatracker.ietf.org/doc/html/rfc6146#section-2>

“A 5-tuple

( *source IP address*,                    //1  
      *source port*,                    //2  
      *destination IP address*,      //3  
      *destination port*,              //4  
      *transport protocol* )          //5

*uniquely identifies a UDP/TCP [socket connection] session.*”

- Hence, multiple fd-s at the source or destination machines may be associated with the same port number

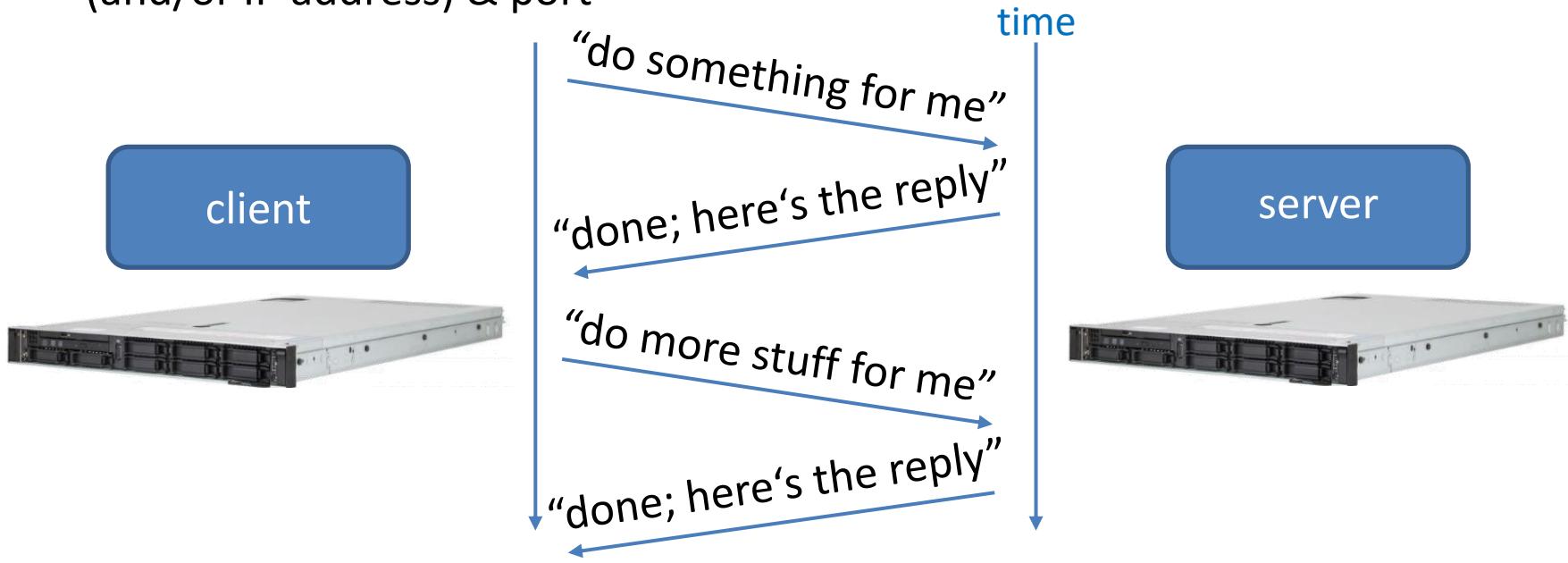
# Client-server model

## Server process (e.g., Apache)

- Always on (both host & process)
- Passively waits for clients to request service, and then reacts: “request-response” paradigm
- Has well-known domain name (and/or IP address) & port

## Client process (e.g., Chrome)

- Asynchronously / intermittently connects to server & sends request(s)
- May have dynamic IP
- May use ephemeral port



Simplistic echo server to exemplify the above

## **TOP-DOWN EXAMPLE**

Code available in

<http://www.cs.technion.ac.il/~dan/course/os/sock.c>

# Our echo application-layer protocol

	Simplistic echo client	Simplistic echo server
1	Connect to the server using TCP	Accept connection from a new client
2	Write (= send) a short byte-sequence message to server	Read (= receive) up to N characters from client
3	Read (= receive) from server at most N characters	Write (= send = echo) these characters back to the client
4	Print the received characters	Print client's host name
5	Exit	Go to 1

```
/*
 * Recall that, for simplicity, we're invoking syscalls
 * through DO_SYS, which exit()s upon failure with the
 * appropriate error message.
 *
 * There are advantages to reporting error via return
 * values, but we opt for simplicity here.
 */
#define DO_SYS(syscall) do { \
    if( (syscall) == -1 ) { \
        perror( #syscall ); \
        exit(EXIT_FAILURE); \
    } \
} while( 0 )
```

```

/*
 * Implement the protocol's server side. The host of this
 * server and the given port must be known to the client
 */
void echo_server(uint16_t port)
{
    const int N=256;
    char buf[N];
    int k, clifd, srvfd = tcp_establish(port);

    for(;;) {
        DO_SYS( clifd = accept(srvfd, NULL, NULL) );
        DO_SYS(      k = read( clifd, buf , N ) );
        DO_SYS(          write( clifd, buf , k ) );
        print_peer("client from:", clifd); // client's host
        DO_SYS( close(clifd) );
    }
}

```

```

/*
 * Implement the protocol's client side. The given host
 * and port must identify the server
 */
void echo_client(const char *host, uint16_t port)
{
    char buf[256], msg[] = "hello\n";
    int k, fd = tcp_connect(host, port);

    DO_SYS(      write(fd, msg, strlen(msg)) );
    DO_SYS( k = read (fd, buf, sizeof(buf)) );
    DO_SYS(      write(STDOUT_FILENO, buf, k) );
    DO_SYS(      close(fd) );

    exit(EXIT_SUCCESS);
}

```

# How were the TCP socket fds created?

- **Different for**
  - Client and server
- **But both need to**
  - Properly initialize a `struct addrinfo` via the `getaddrinfo()` syscall
  - Invoke `socket()` using values from `addrinfo`
    - Which returns the sockfd
- **addrinfo encapsulates all the required info, including**
  - Protocol family (IPv4 or IPv6)
    - Implies different size of address
    - Which the struct encapsulates
  - Protocol (TCP)
  - Socket type (stream)
  - Whether this should be a server or client

# Creating TCP socket descriptors: `getaddrinfo`

```
int getaddrinfo( const char*  
                const char*  
                const struct addrinfo*  
                struct addrinfo**  
                           node /*host (in our case)*/,  
                           service /*port (in our case)*/,  
                           hint /*input*/,  
                           res /*output, free with freeaddrinfo*/);
```

	argument	in server	in client
directly	node	null (=this host)	server's host name
	service		server's well-known port
input via hint	hint.ai_flags	AI_PASSIVE (=server)	0 (=client)
	hint.ai_family	AF_UNSPEC (IP version unspecified)	
	hint.ai_protocol		IPPROTO_TCP
	hint.ai_socktype		SOCK_STREAM
output via res	hint.ai_family		either IPv4 or IPv6
	hint.ai_addr	struct <code>sockaddr</code> *	, encapsulates IP+port
	hint.ai_addrlen		length of *ai_addr

- Thus, programmers can remain unaware of which IP version is being used by this particular host
  - v4 or v6
- And they don't need to worry about struct `sockaddr` allocation
  - Has different size for different IP version

# Creating TCP socket descriptors: `getaddrinfo`

```
int getaddrinfo( const char*  
                const char*  
                const struct addrinfo*  
                struct addrinfo**  
                           node /*host (in our case)*/,  
                           service /*port (in our case)*/,  
                           hint /*input*/,  
                           res /*output, free with freeaddrinfo*/);
```

	argument	in server	in client
directly	node	null (=this host)	server's host name
	service		server's well-known port
input via hint	hint.ai_flags	AI_PASSIVE (=server)	0 (=client)
	hint.ai_family	AF_UNSPEC (IP version unspecified)	
	hint.ai_protocol		IPPROTO_TCP
	hint.ai_socktype		SOCK_STREAM
output via res	hint.ai_family		either IPv4 or IPv6
	hint.ai_addr	struct <code>sockaddr</code> *	, encapsulates IP+port
	hint.ai_addrlen		length of *ai_addr

- Why do we need both `ai_protocol` and `ai_socktype`?
  - Other protocols may implement a stream abstraction too
  - Notably, non-default TCP versions specialized for data centers
    - LAN (local area network), as opposed WAN (wide area network)
    - Where latencies are lower (LAN=microseconds, WAN=milliseconds)

# Creating TCP socket descriptors: `getaddrinfo`

```
struct addrinfo*
alloc_tcp_addr(const char *host, uint16_t port, int flags)
{
    int err;    struct addrinfo hint, *a;    char ps[16];

    snprintf(ps, sizeof(ps), "%hu", port); // why string?
    memset(&hint, 0, sizeof(hint));
    hint.ai_flags = flags;
    hint.ai_family = AF_UNSPEC;
    hint.ai_socktype = SOCK_STREAM;
    hint.ai_protocol = IPPROTO_TCP;

    if( (err = getaddrinfo(host, ps, &hint, &a)) != 0 ) {
        fprintf(stderr,"%s\n", gai_strerror(err));
        exit(EXIT_FAILURE);
    }

    return a; // should later be freed with freeaddrinfo()
}
```

# Creating TCP socketfd: server

- A sequence of 4 syscalls

- `srvfd = socket(protocol family /*IP version*/, connection type /*stream in our example*/, protocol /*TCP in our example*/ );`
- `bind(srvfd, /*to*/ well-known port associated with server );`
- `listen(*on*/* srvfd /*for incoming requests directed at port, */ /*allowing*/ backlog /*of un-accept()ed pending requests, */ /*at the most; this syscall transforms srvfd to a server fd */ /*able to accept() new connections (= create clifd-s)*/ );`
- `clifd = accept(srvfd); /* new ephemeral fd for each client connect() request */`

# Creating+using TCP sockfd-s: both sides

server	message	client
srvfd = socket(...)		sockfd = socket(...)
bind( srvfd , port )		
listen( srvfd )		
<i>loop:</i>		
clifd = accept( srvfd )	request service (transport) ←	connect( sockfd , host+port )
read(clifd )	request (application) ←	write( sockfd )
write( clifd )	response (application) →	read( sockfd )

# Creating TCP socketfd: server

```
/*
 * Return server fd (on this host) that listen()s on port
 */
int tcp_establish(uint16_t port)
{
    int srvfd;
    struct addrinfo *a =
        alloc_tcp_addr(NULL/*host*/, port, AI_PASSIVE);

    DO_SYS( srvfd = socket( a->ai_family,
                            a->ai_socktype,
                            a->ai_protocol ) );
    DO_SYS( bind( srvfd,
                  a->ai_addr,
                  a->ai_addrlen ) );
    DO_SYS( listen( srvfd,
                   5/*backlog*/ ) );
    freeaddrinfo( a );
    return srvfd;
}
```

# Creating TCP socketfs: client

```
/*
 * Return client fd connect()ed to server @ host+port
 */
int tcp_connect(const char* host, uint16_t port)
{
    int clifd;
    struct addrinfo *a = alloc_tcp_addr(host, port, 0);

    DO_SYS( clifd = socket( a->ai_family,
                           a->ai_socktype,
                           a->ai_protocol ) );
    DO_SYS( connect( clifd,
                      a->ai_addr,
                      a->ai_addrlen ) );

    freeaddrinfo( a );
    return clifd;
}
```

# Getting information about other side

```
/*
 * Print hostname of peer associated with sockfd
 */
void print_peer(const char *msg_prefix, int sockfd)
{
    struct sockaddr_storage store;      // big enough for any sock
    socklen_t alen = sizeof(store);     // needed by getpeername
    char peer[HOST_NAME_MAX+1]={0};     // name of peer
    int mlen = strlen(msg_prefix)+8;   // 'msg' needs to be bigger
    char msg[sizeof(peer)+mlen]={0};   // use this for printing
    struct sockaddr *a = (struct sockaddr*)&store; // base class

    DO_SYS( getpeername(sockfd, a, &alen) ); // fills a+alen
    if((err=getnameinfo(a,alen,peer,sizeof(peer),NULL,0,0))) {
        fprintf(stderr,"%s\n", gai_strerror(err));
        exit(EXIT_FAILURE);
    }
    snprintf(msg, sizeof(msg), "%s %s\n", msg_prefix, peer);
    DO_SYS( write(STDOUT_FILENO, msg, strlen(msg)) );
}
```

# Example aftermath

- **Support very long per-client TCP messages?**
  - Easily: server reads + writes (=echoes) in a loop
  - Until client close()s its sockfd endpoint
    - When this happens, read(clifd) returns 0
- **Problem**
  - What if some clients finish quickly  
(echo short message)  
whereas other take a long time?  
(echo very long message, or send message chunks slowly)  
=> Convoy effect
- **Solution**
  - Concurrency: serve multiple clifd-s simultaneously, in RR order
- **But how will that work, technically?**
  - Need to use I/O multiplexing syscall: `select()` or `poll()` or `epoll()`
    - Get a set of fd-s; return a subset of ready fd-s that won't block
  - See, e.g., <https://devarea.com/linux-io-multiplexing-select-vs-poll-vs-epoll/>

# Example aftermath

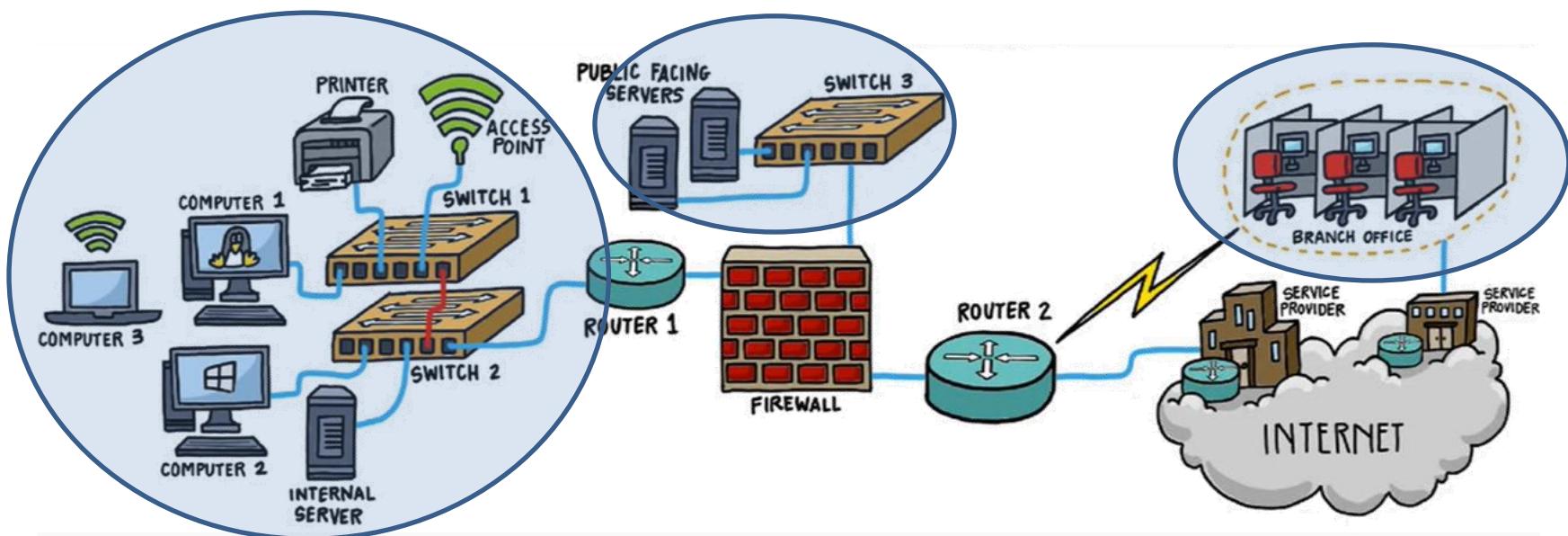
- **Can we use UDP for our simple echo server (instead of TCP)?**
  - If we assume each message fits into one segment
    - No need to handle reordering
  - But loss is an issue, so
    - Client should set up an alarm, and
    - Retransmit when it expires, if server response hasn't yet arrived
- **What if messages don't fit in one UDP segment?**
  - Need multiple segments (=messages), so
  - Requests & responses should be numbered due to possible reordering
  - Client must then handle out-of-order echo replies

lecture ended

# **PHYSICAL-LAYER (L1) & LINK-LAYER (L2) PROTOCOLS**

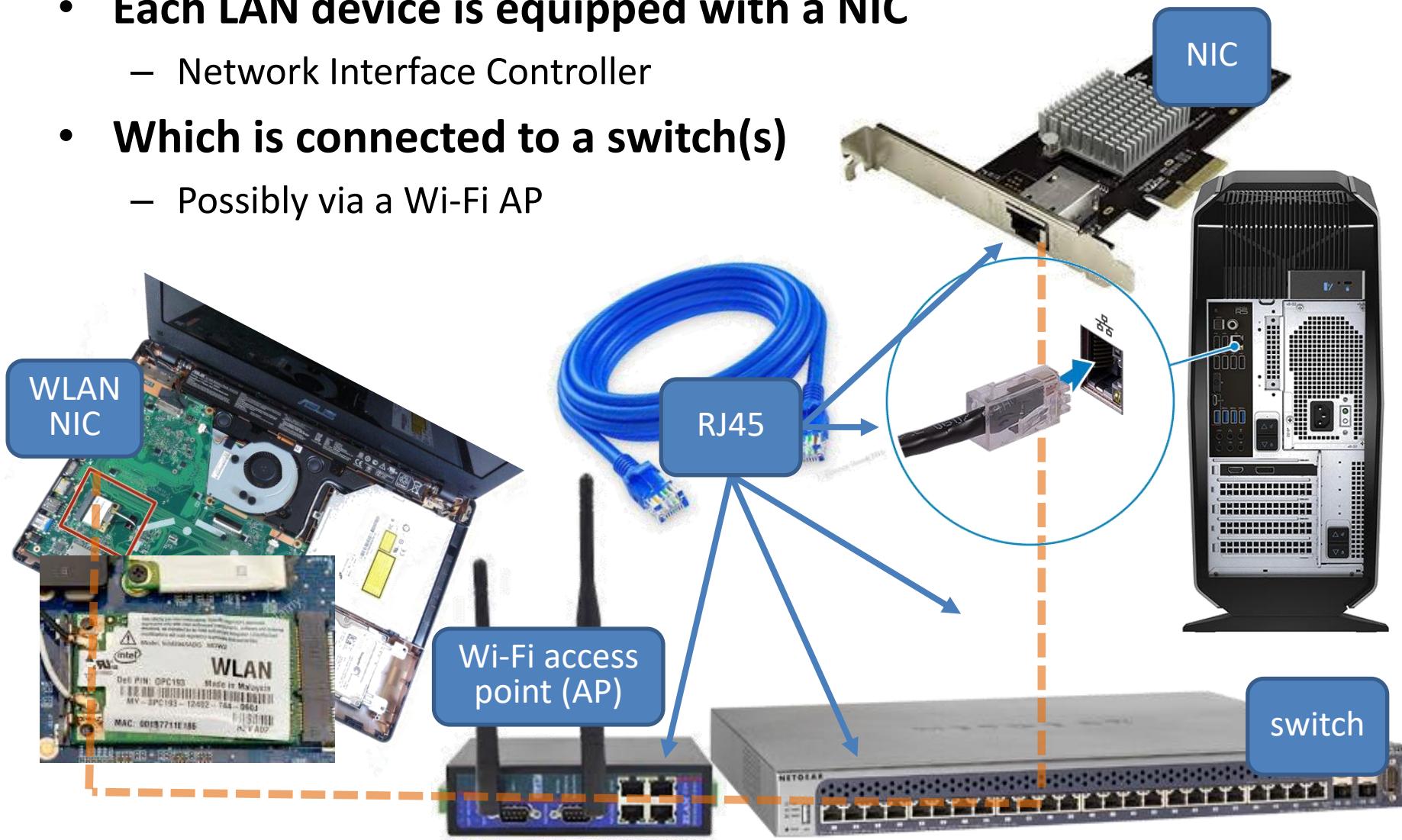
# LAN (local area network) connectivity

- **LAN examples**
  - Computers in your home
  - CS server room
  - Computers where you work
  - Computer in a data center
  - (Each may include multiple LANs)
- **Devices in the LAN are connected**
  - With wires, or wirelessly, or both



# LAN connectivity

- Each LAN device is equipped with a NIC
  - Network Interface Controller
- Which is connected to a switch(s)
  - Possibly via a Wi-Fi AP



# LAN connectivity

- **Question**
  - What's the protocol LAN nodes use to communicate?
    - Nodes = hosts, NICs, APs, switches, phones, printers, ...
  - What's the protocol that flows through the wires?
  - What's the protocol that flows on top of Wi-Fi?
  - What's the native “language” that all these components “speak”?
    - Both hardware components (nodes)
    - And software components (OS device drivers that speak to nodes)
- **Answer**
  - Most frequently, its Ethernet
  - We'll focus on it in this lecture

# Ethernet (IEEE 802.3 standard)

- **A combination of**
  - Hardware, firmware, and (OS) software
- **Most dominant wired LAN technology**
  - Simple, cheap, fast
    - 10 Gb/sec (Gbps) is probably most widely deployed in datacenters
    - 40 / 100 / 200 / 400 Gbps commodity
    - 800 Gbps around the corner
- **Ethernet is both a link-layer (L2) protocol**
  - Allows nodes (not processes) to communicate within the LAN
  - By sending “frames” (how byte-chunks are called in the link-layer)
- **And a physical-layer (L1) protocol**
  - Lowest protocol layer (EE realm)
  - Defines how raw bits are transmitted
  - Defines the transmission media
    - For Ethernet, there are several (twisted pairs, fiber, cable, ...)

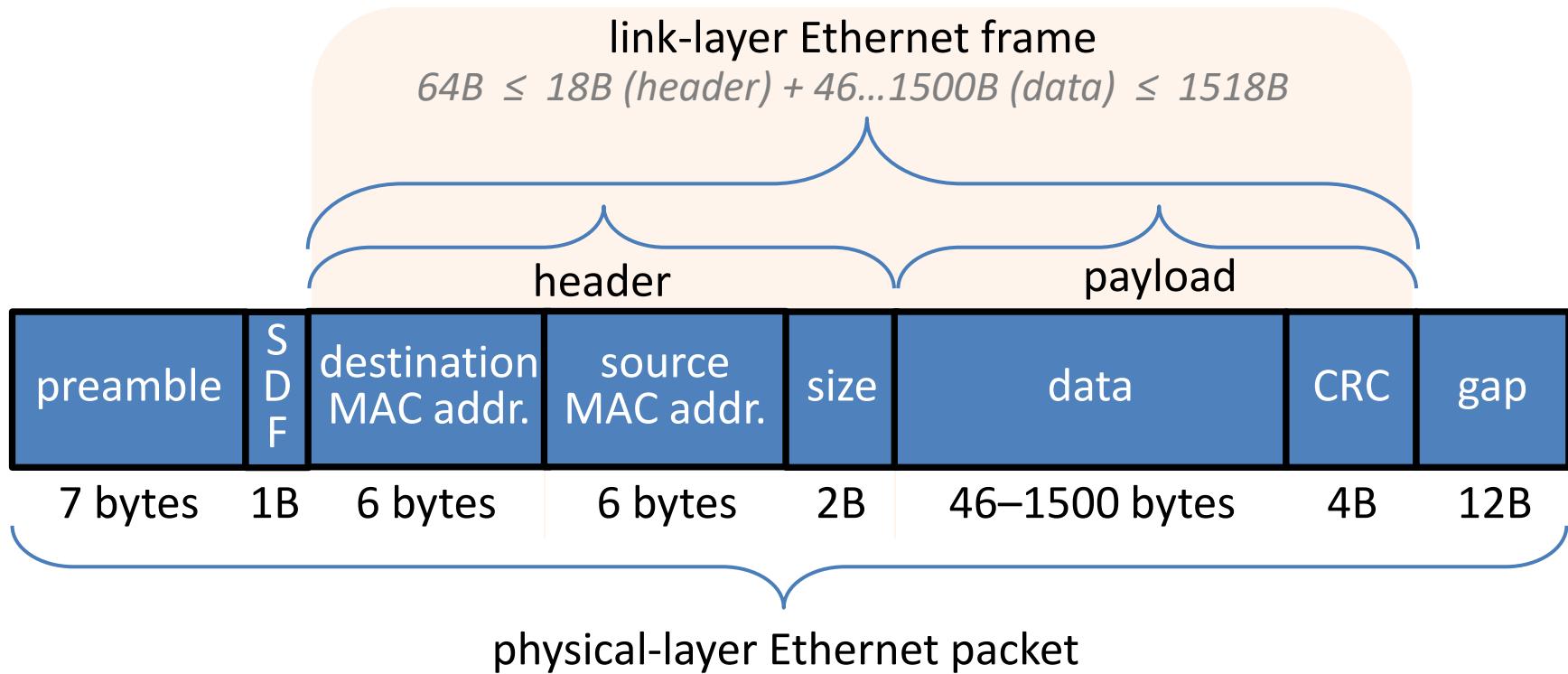
# Ethernet (IEEE 802.3 standard)

- **Network features**
  - Connectionless & unreliable (lossy)
  - Ethernet frames can fail to reach their destination
- **Relation with transport-layer protocols (TCP/UDP)?**
  - Within the LAN, hardware components speak Ethernet, not TCP/UDP
  - They don't typically understand TCP/UDP (oversimplified)
  - It is the OS that implements TCP/UDP on top of Ethernet
  - **For Ethernet nodes, TCP/UDP traffic appears as regular data**
  - More on that later

# MAC (media access control) address

- **A unique 48-bits (= 6-bytes) number**
  - Identifies each Ethernet component
  - Burned in ROM of NIC / Switch / AP
  - Used to switch Ethernet frames to their destination within the LAN
- **Notation**
  - 6 pairs of hex digits, e.g.,
    - C8:5B:76:EA:B8:A0
  - Of which the device manufacturer is allocated 3, exclusively, e.g.,
    - CC:46:D6 – Cisco
    - 3C:5A:B4 – Google
    - 00:9A:CD – HUAWEI
  - So, it's the job of manufacturers to ensure MAC address uniqueness

# Ethernet frame format (simplified)



- Using frame's header
  - A switch knows where to send the frame, and
  - Receiver can identify sender
- SDF = start frame delimiter
- CRC = cyclic redundancy check, an error-detecting code
  - Error => frame is dropped

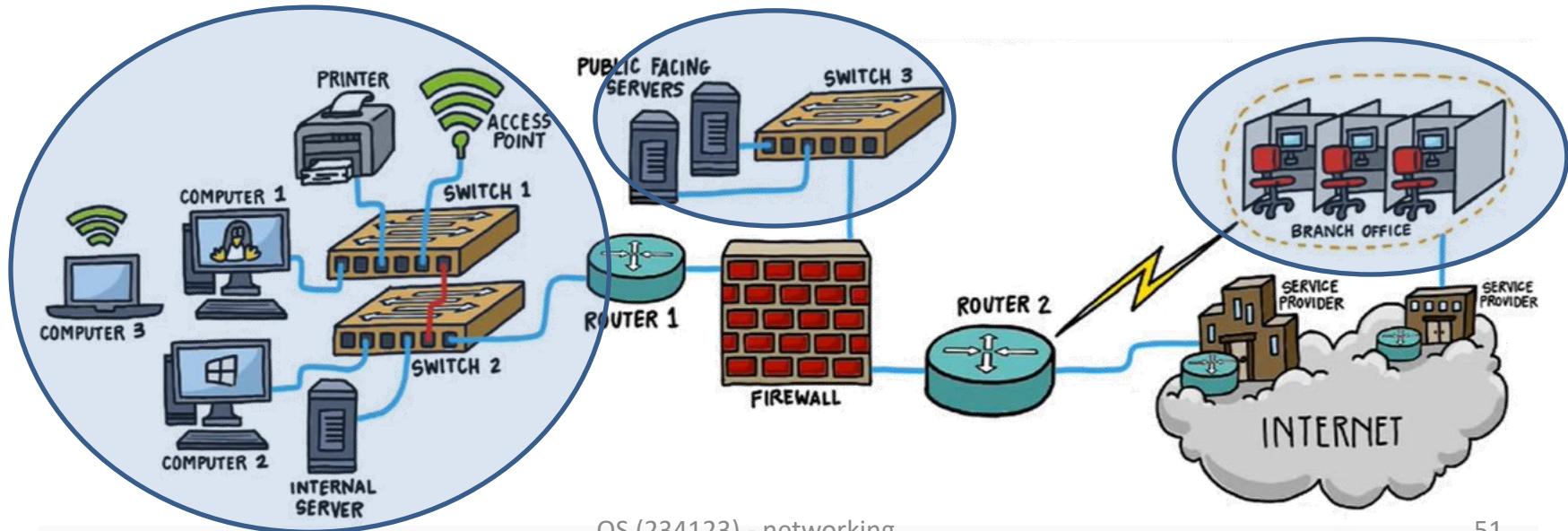
The role of IP, again

# **WHAT'S THE INTERNET, REALLY?**

# IP & Routers

- **Problem**

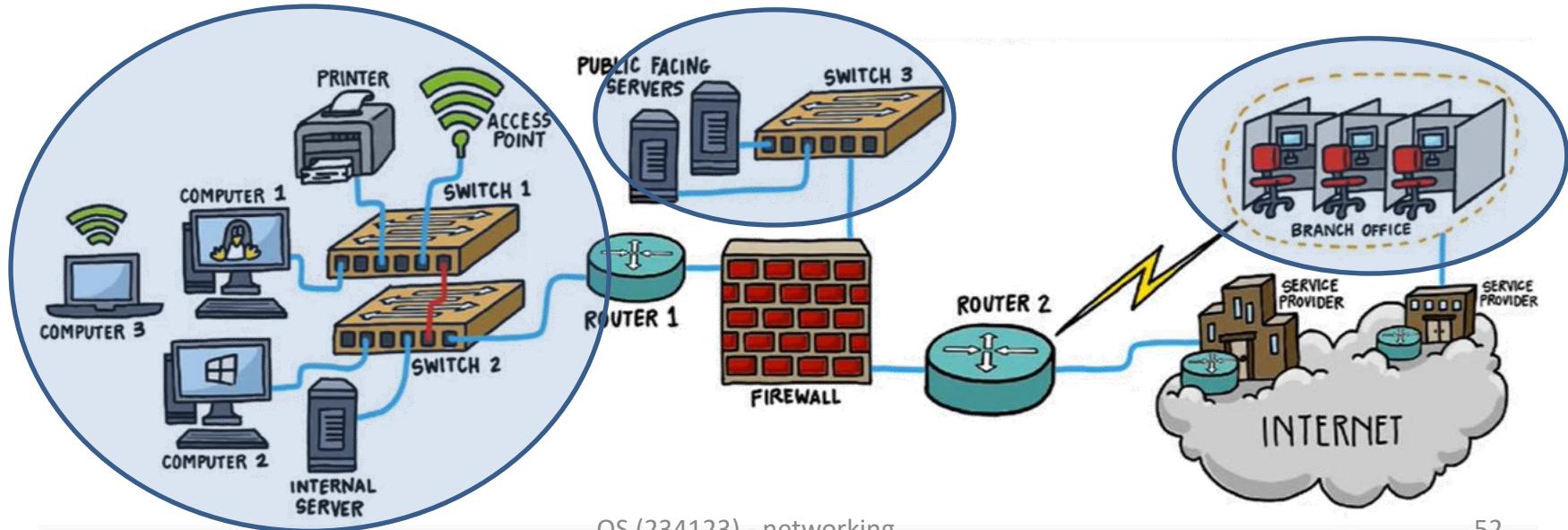
- LAN hardware components typically “speak” L2 (commonly Ethernet)
  - A “language” that works exclusively within the LAN
- How then, can node@LAN-A send messages to node@LAN-B?
  - Technically, switches@LAN-A can’t talk to switches@LAN-B
  - And in addition, note that...



# IP & Routers

- **Problem**

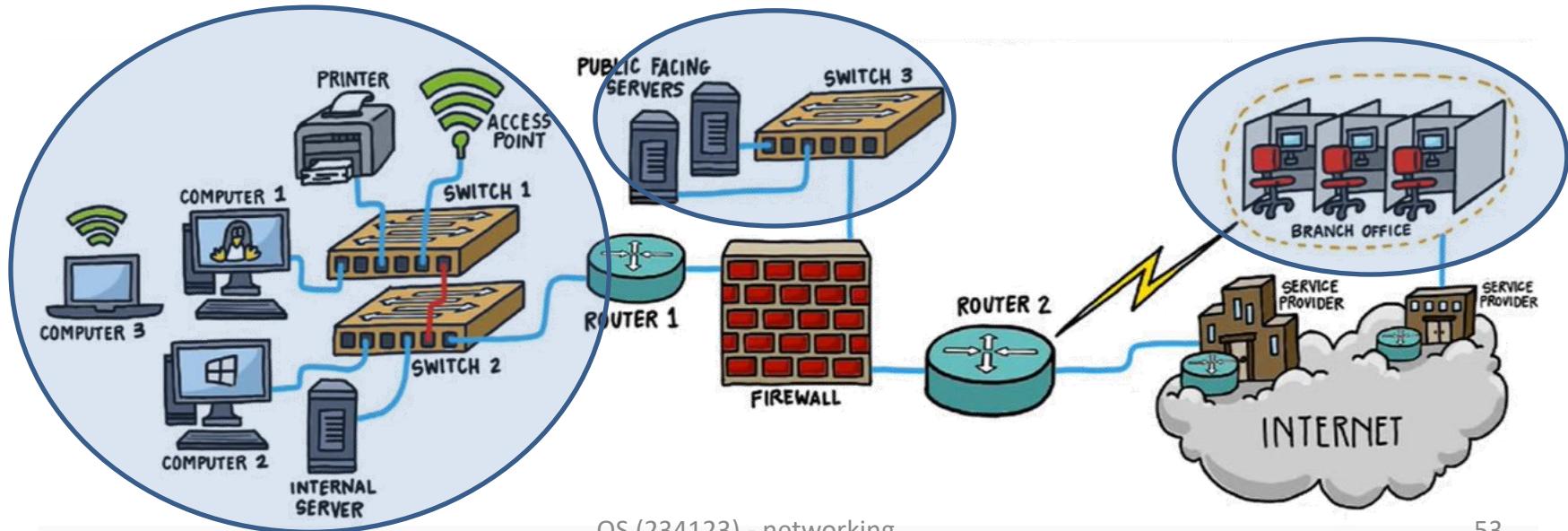
- LAN-B may use a **non-Ethernet protocol** (e.g., InfiniBand)
- LAN-A may be **continents apart** from LAN-B
- Getting from LAN-A to LAN-B may require using **multiple L1/L2-s**, e.g.,
  - Phone / cable / fiber / satellite / ... communication channels



# IP & Routers

- **Solution**

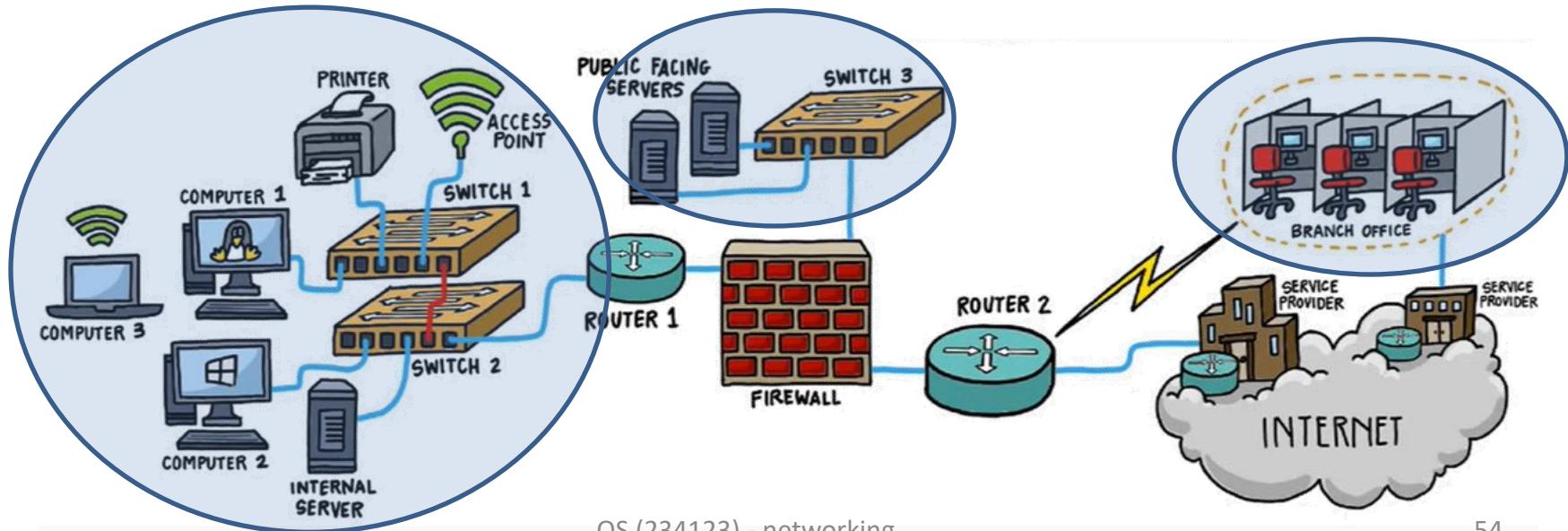
- Router = hardware component
  - Physically stands @ edge of **at least** two networks, LAN1 & LAN2,
  - And knows how to communicate with both
- So, the router can **speak** in the language of
  1. **L1/L2** of **LAN1 & LAN2**, and it also speaks
  2. IP = the network-layer (**L3**) protocol that implements IP addresses
    - On top of which transport-layer (**L4**) protocols are built



# IP & Routers

- **Solution**

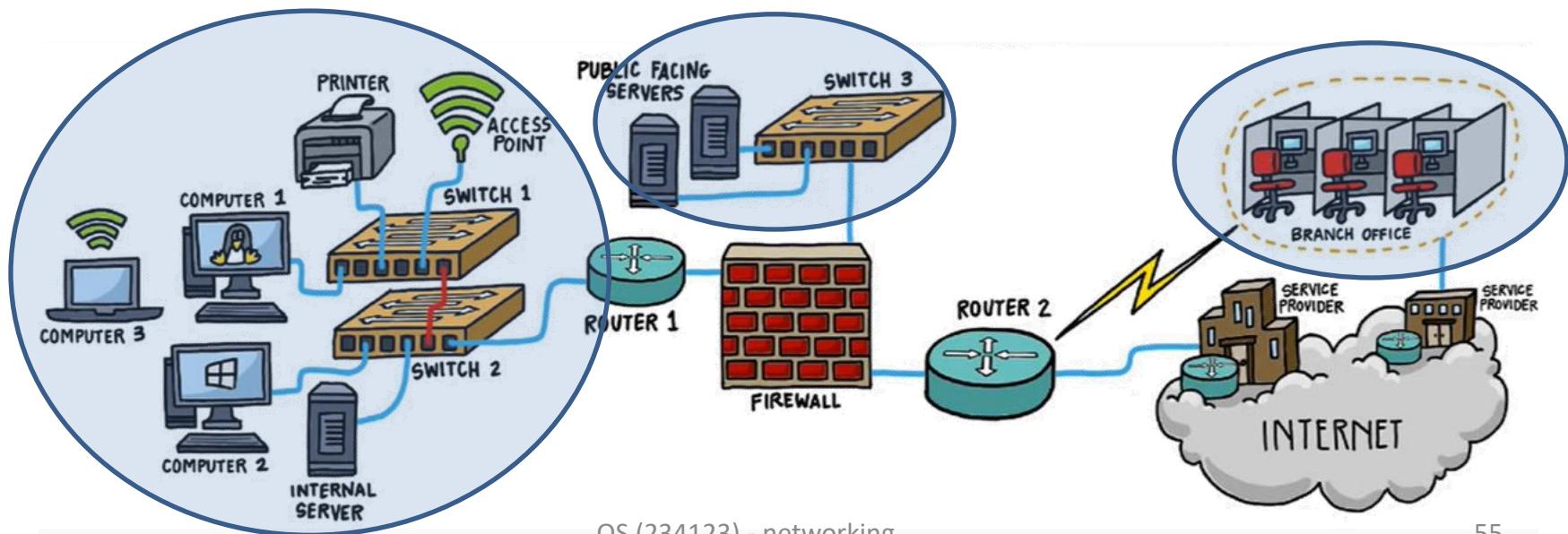
- Thus, the router can “peel off” L2-headers of LAN1 and re-encapsulate the corresponding data (= IP packets) within L2-headers of LAN2
- By forwarding IP packets (one setup at a time) between networks, routers eventually route the packets from source to destination



# IP & Routers

- **Solution**

- Thus, the “internet” is an “inter-network”... of networks!
  - That’s the origin of the name
  - Networks are vertices in the internet graph, and routers are edges
- Made possible by the IP (L3) protocol
  - Which provides a global address space for all hosts in the world
  - And routing tables (instructions how to route)



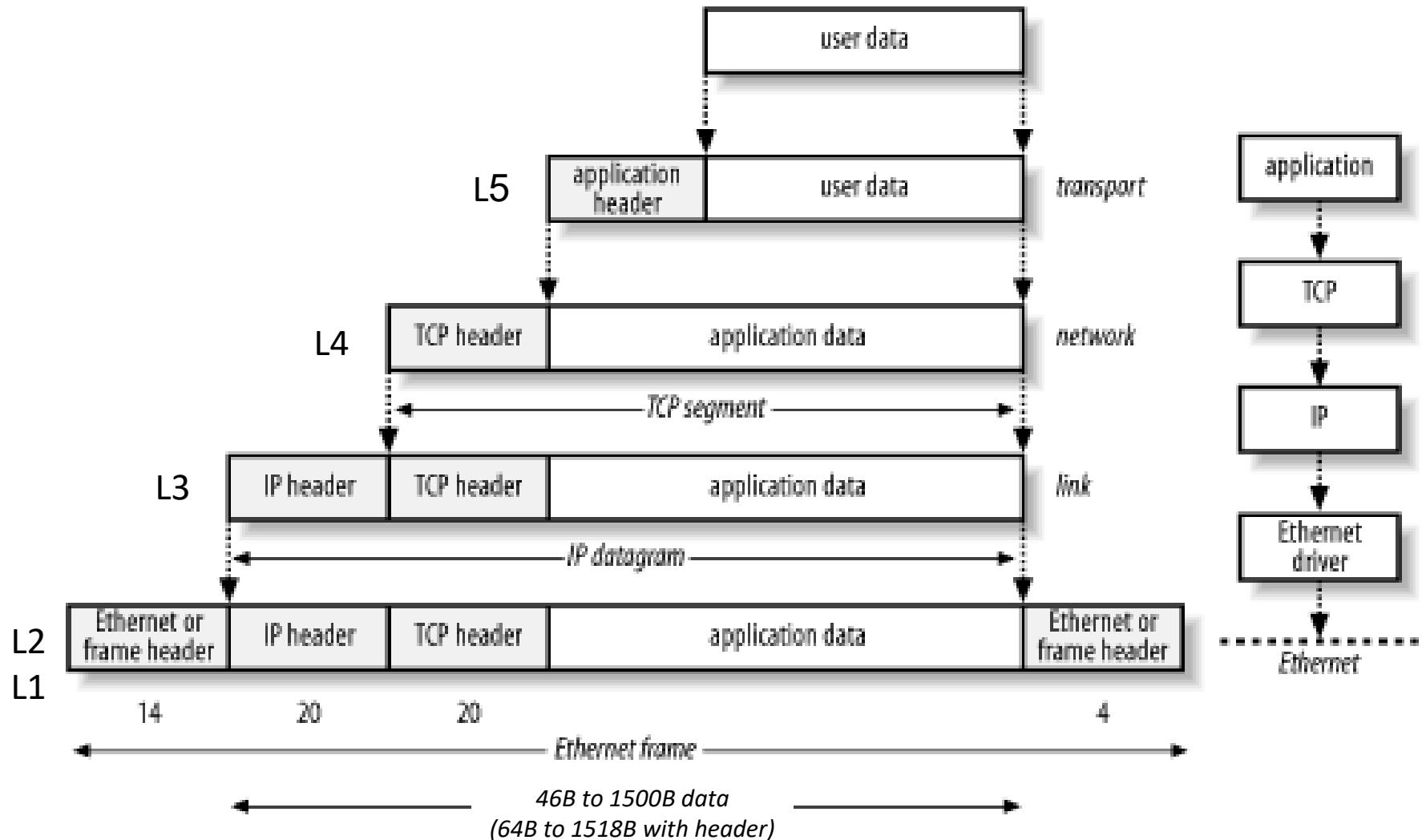
Closing loose ends &

# **PUTTING IT ALL TOGETHER**

# The “TCP/IP” protocol stack – recap

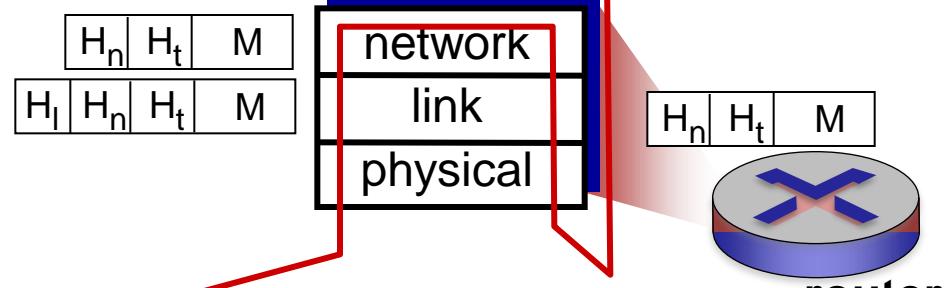
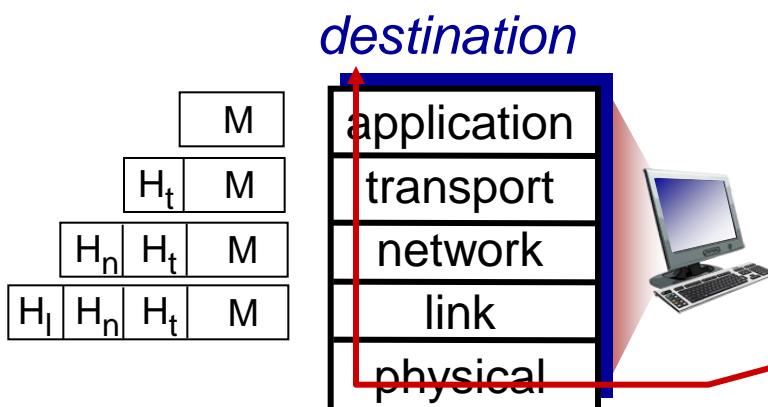
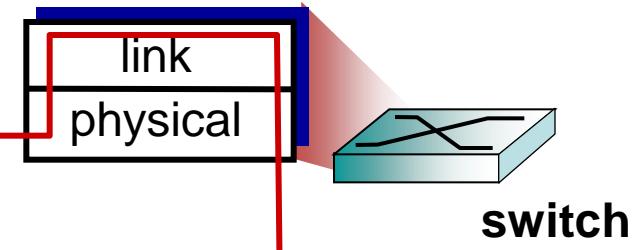
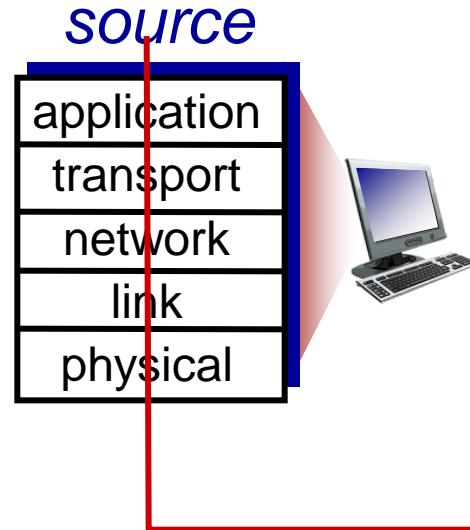
- **L1 = physical-layer**
  - How bits transmitted (EE level)
- **L2 = link-layer**
  - Ethernet (frequently)
  - Communication between hosts across a **LAN**, with MACs
- **L3 = network-layer**
  - IP, which provides global address space
  - Communication between hosts across the **WAN**, with IPs
- **L4 = transport-layer**
  - TCP & UDP (most frequently), stream abstraction
  - Communication between **processes** across the WAN
- **L5 = application-layer**
  - Numerous protocols that utilize L4

# Reminder: we started off with encapsulation

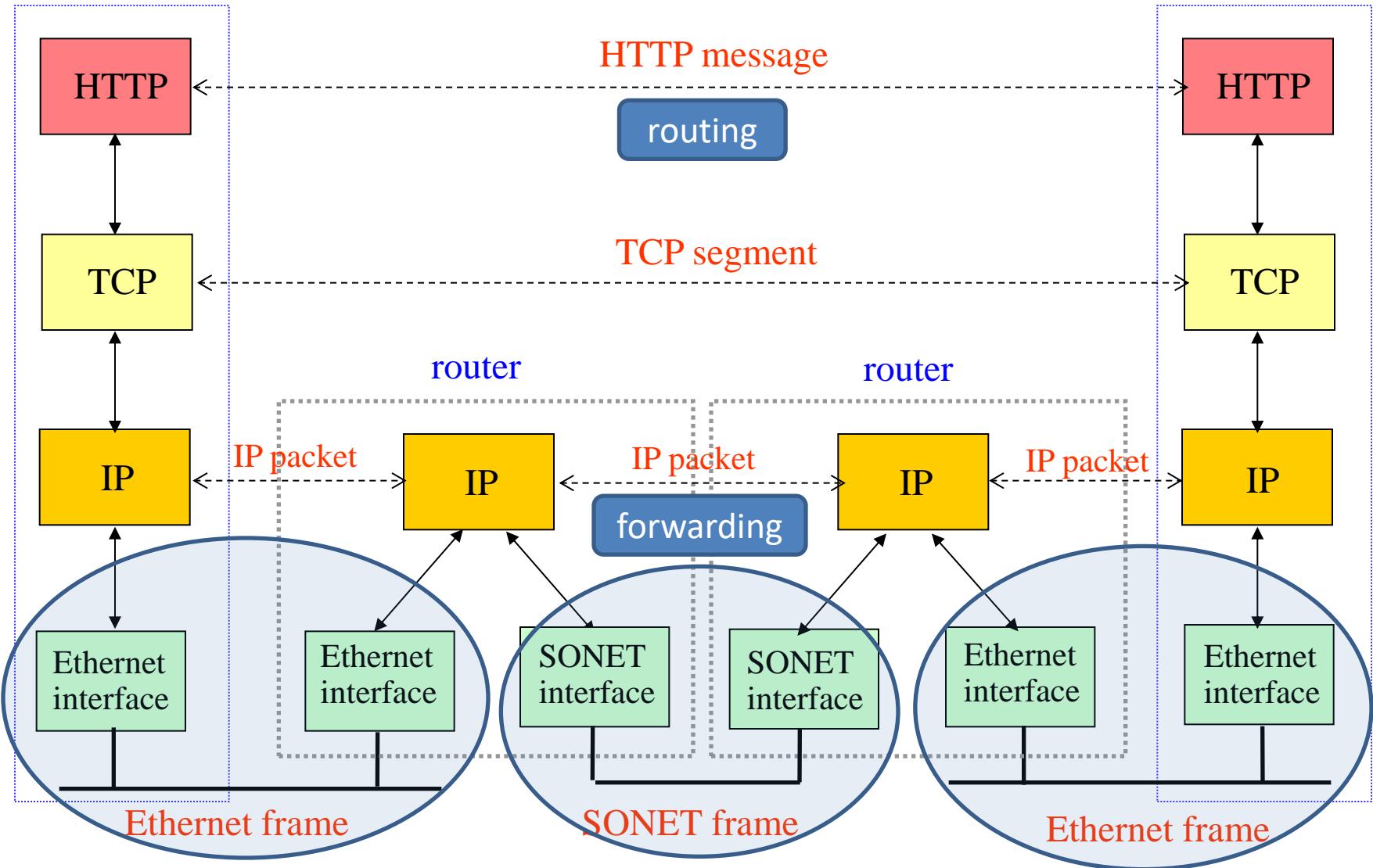


# Encapsulation

message	M
segment	H <sub>t</sub> M
packet	H <sub>n</sub> H <sub>t</sub> M
frame	H <sub>l</sub> H <sub>n</sub> H <sub>t</sub> M

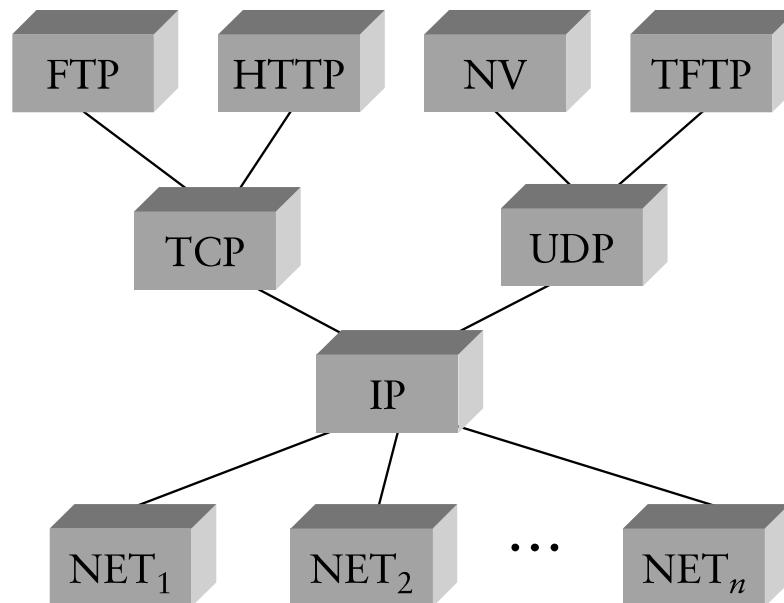


# Forwarding & routing



# Layering – IP as a narrow waist

- Many applications protocols on top of UDP & TCP
- IP works over many types of networks
- This is the “Hourglass” architecture of the Internet
  - If every network supports IP, applications may run over many different networks (cellular, ...)



# Fragmentations & reassembly

- **Link layer has MTU**
  - = Maximal transfer size = largest possible frame in network layer
  - Changes for different link types
  - For Ethernet: data  $\leq 1500B$ , which is smaller than..
- **Maximal IP packet**
  - 64 KB
- **So large IP packets are divided (fragmented)**
  - One packet becomes several packets
  - Each containing all headers of all higher-level protocols
- **IP packet**
  - Header contains fragment offset
  - Fragmented @ source
  - Reassembled @ final destination
  - NICs typically know how to fragment & reassemble (“HW acceleration”)
    - TSO (= TCP segmentation offload) & LRO (= large receive offload)

# Connection between IP & LANs

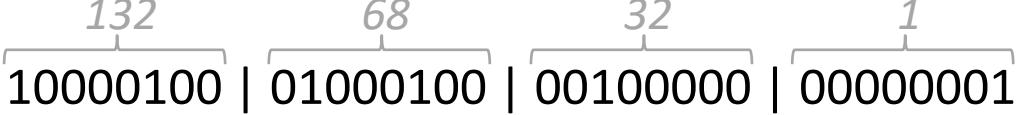
- Recall: IP presented as 4 x 8-bit decimal dot-separated octets
  - For example, the IP of [csa.cs.technion.ac.il](http://csa.cs.technion.ac.il) is

Notation	IP address
Decimal	2219057153 (not terribly convenient)
Binary (8x4 = 32 bits)	 10000100   01000100   00100000   00000001
Dot-decimal	<a href="http://132.68.32.1">132.68.32.1</a> (a bit more convenient)

- Actually, the IP address consist of two parts
  - Subnet part: high order bits
  - Host part: low order bits
- How is the division determined?
  - IP address is *always* coupled with a corresponding [subnet mask](#)

# Connection between IP & LANs

- For example, the subnet mask of
  - [csa.cs.technion.ac.il](http://csa.cs.technion.ac.il) is the 24 most significant bits

Notation	IP address
Decimal	2219057153 (not terribly convenient)
Binary (8x4 = 32 bits)	 10000100   01000100   00100000   00000001
Dot-decimal	<a href="http://132.68.32.1">132.68.32.1</a> (a bit more convenient)

- CIDR format
  - CIDR = classless inter-domain routing
  - a.b.c.d/x, where x is the number of bits in the subnet portion
  - For csa, this is [132.68.32.1/24](http://132.68.32.1/24)

# Connection between IP & LANs

- **Meaning of subnet**
  - All nodes in subnet can physically reach each other
    - Without intervening router
    - Through switches only
  - Namely, the IP subnet
    - Defines the boundaries of the Ethernet LAN

# ARP – address resolution protocol: IP => MAC

- **IP addresses of other machines are known**
  - Or can be discovered using domain names and DNS
- **X knows the IP of Y, with which it wants to connect**
  - How does X discover Y's MAC?
- **The network learns (plug & play) as follows**
  - X **broadcasts** ARP query packet
    - Contains Y's IP
    - Destination MAC = FF:FF:FF:FF:FF:FF
  - All nodes on LAN receive the query
  - Y, which knows its MAC address, sends it back to X (unicast)
- **If Y is outside the LAN**
  - X gets the MAC of the first-hop router,  
which can forward it onwards based on its IP

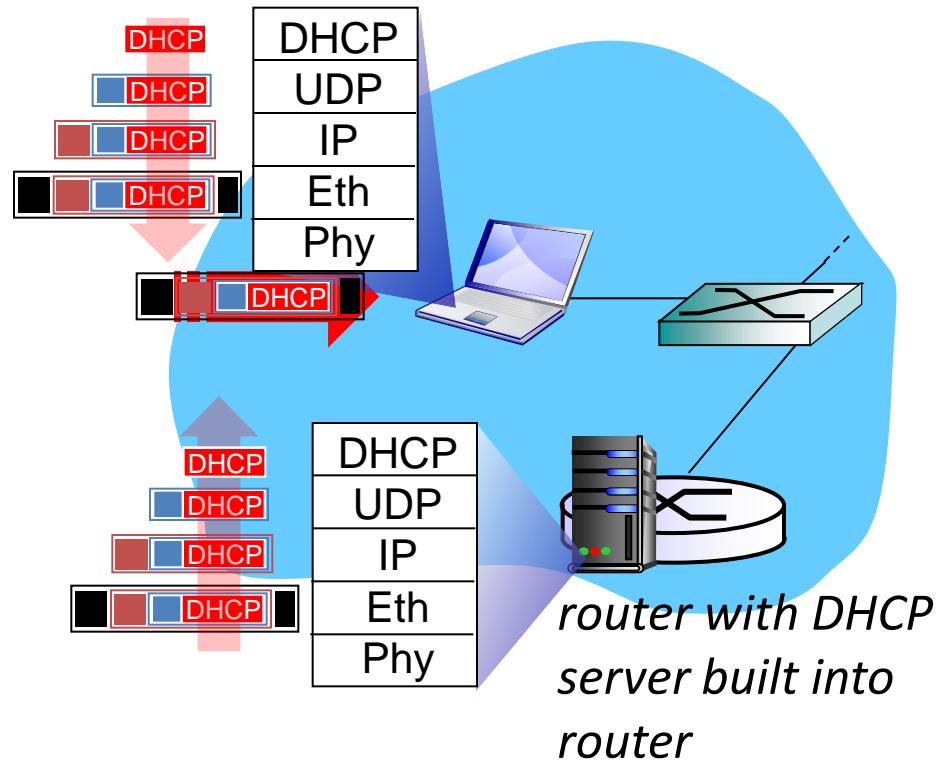
# NAT – network address translation

- **At home, for example, your router box**
  - (which is also a switch, an AP, and a modem)
  - Typically, has only one IP, allocated to you by your ISP
- **But you have many devices in your home LAN**
  - Which interact with WAN of the outside world (e.g., by browsing)
  - How can this be?
- **The IP standard defines a few millions of “private IP address”**
  - 10.0.0.0/8 (= 10.\*.\*.\*)
  - 172.16.0.0/12 (= 172.16.0.0 – 172.31.255.255)
  - 192.168.0.0/18 (= 192.168.\*.\*)
- **Your router box dynamically assigns such IPs**
  - To all your home LAN devices
- **And it jiggles the IP + port of outgoing/incoming segments**
  - Accordingly
  - (Other routers will refuse to forward private IPs)

# DHCP – dynamic host config. protocol: get an IP

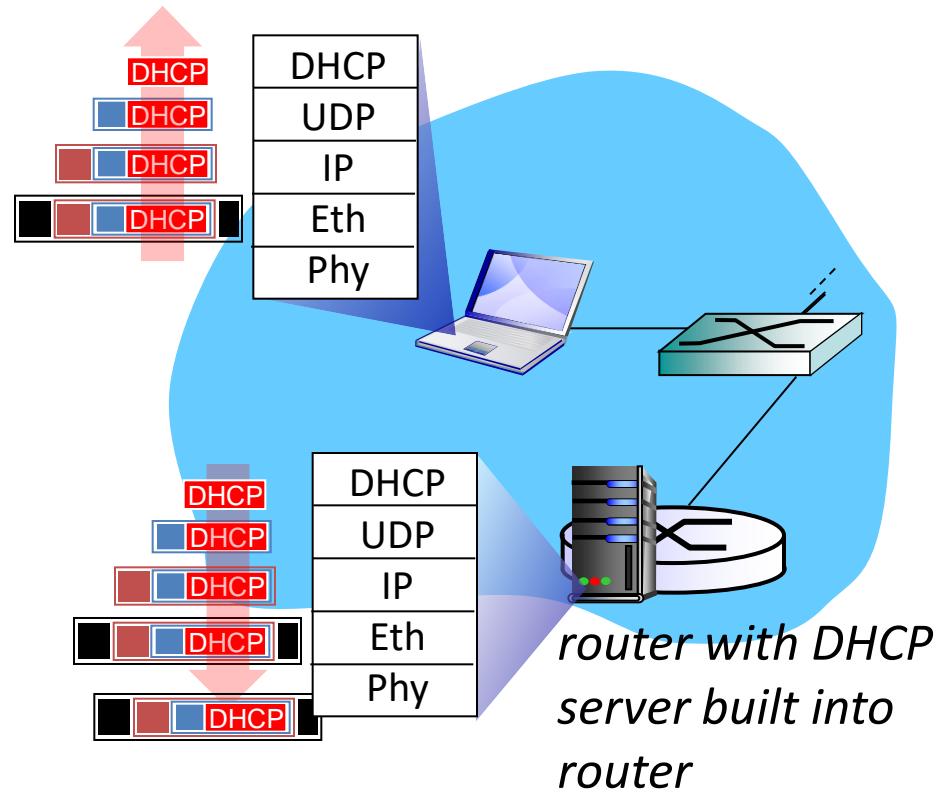
- **Layer-5 (application) protocol**
  - Implemented by the DHCP server (= the app)
    - In your home, typically, built into your router
- **Goal of DHCP server**
  - Allows client's host to dynamically get IP address when joining LAN
  - Can also provide
    - Subnet mask (indicating net vs. host portion of IP address)
    - IP of client's first-hop router (for IP destination outside LAN)
    - Name + IP of DNS server
- **Uses UDP**
  - Which provides the ability to broadcast to all hosts in LAN
  - (Which in turn uses Ethernet's ability to broadcast in LAN)

# DHCP – dynamic host configuration protocol



- Connecting laptop needs IP address (+ IPs of first-hop & DNS server)  
=> Acts as a DHCP client
- Client issues DHCP request  
“is there a DHCP server out there?”
  - encapsulated in UDP,  
encapsulated in IP,  
encapsulated in Ethernet
- Ethernet frame broadcast (FF:FF:FF:FF:FF) on LAN
  - Received at router running DHCP server
- Ethernet demuxed to IP  
demuxed to UDP demuxed to DHCP

# DHCP – dynamic host configuration protocol



- DHCP servers responds  
“Here’s the info you should use!”
  - New IP for client
  - IP of first-hop router for client
  - Name+IP of DNS server for client
- Encapsulation of DHCP server, frame forwarded to client, demuxing up to DHCP at client
- Client now knows its IP address, name & IP address of DNS server, IP address of its first-hop router

# **Operating Systems (234123)**

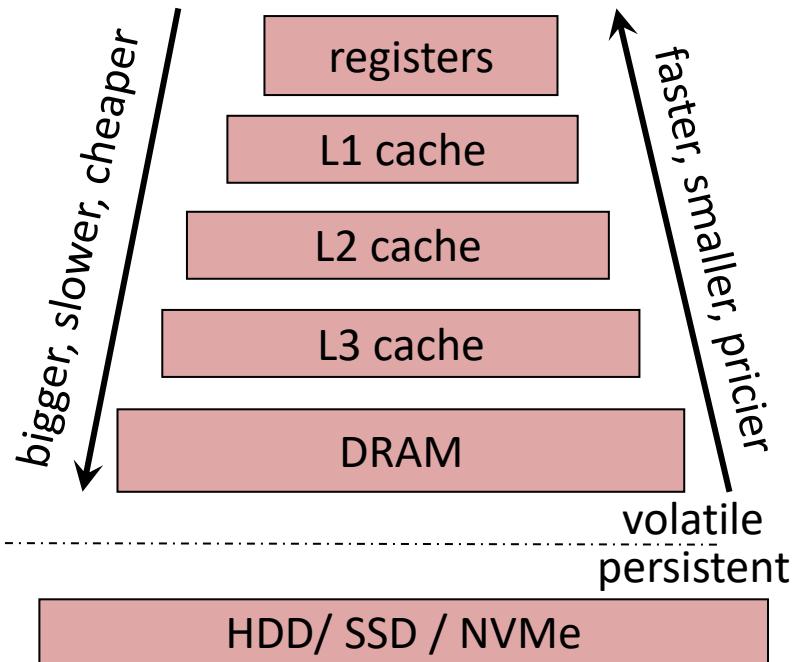
## ***Virtual memory – concepts***

Dan Tsafrir (2025-05-26 + ~15 mins)

Partially based on slides by Hagit Attiya

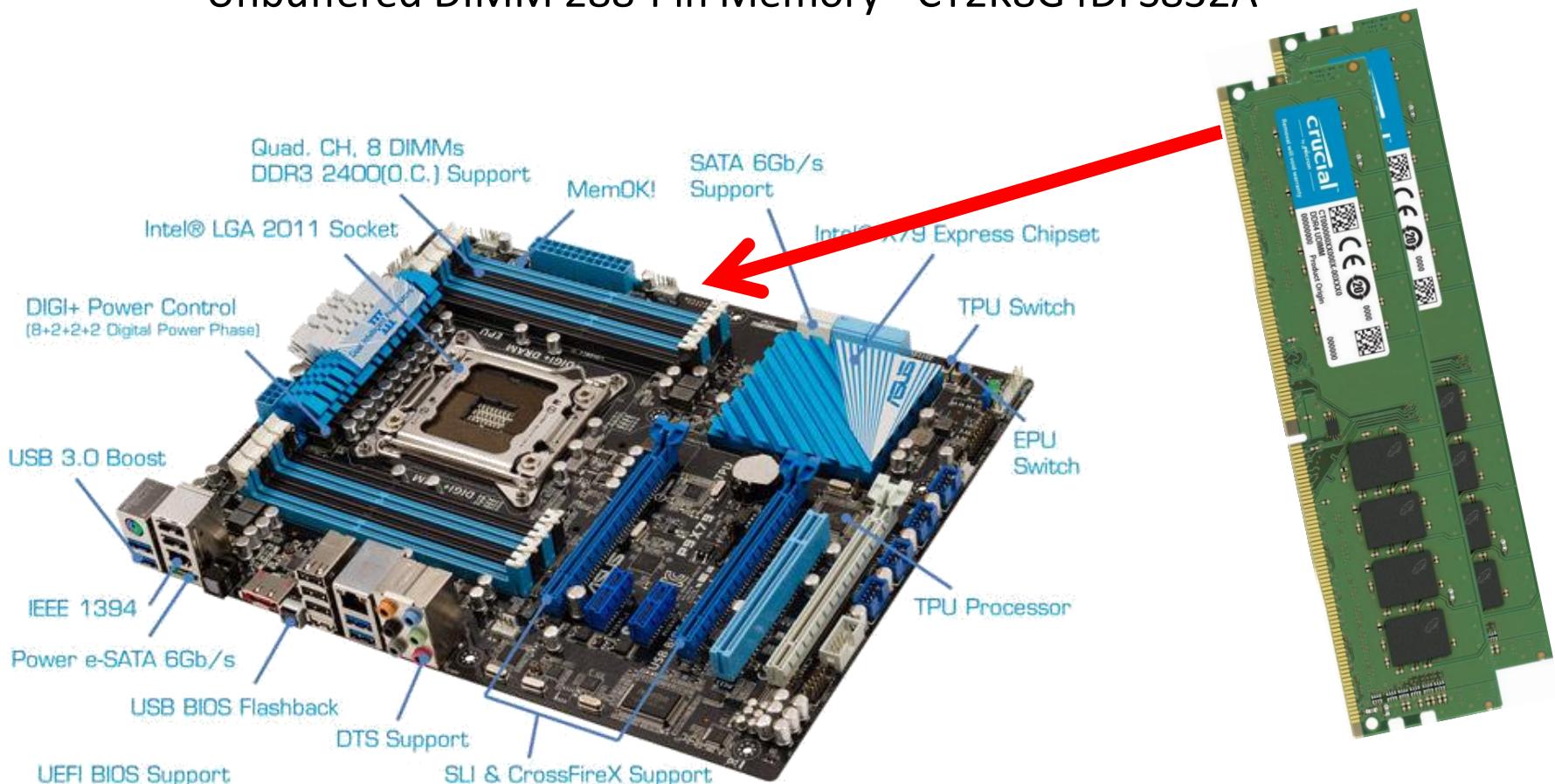
# In a previous lecture: “MAMAS in a nutshell”

- 1 – 32 cores @ 1–5 GHz
- **Memory hierarchy**
  - L1 = O(64 KB) per core
    - Latency = O(**2 cycles**)  
≈ O(1 nanosecond)
  - L2 = O(512 KB) per core
    - Latency = O(**10 cycles**)  
≈ O(5 nanoseconds)
  - L3 = LLC = O(8 MB) shared
    - Latency = O(**40 cycles**)  
≈ O(20 nanoseconds)
    - (LLC is “last-level cache”)
  - DRAM = O(1 – 1000 GBs) shared
    - Latency = O(**200 cycles**)  
≈ O(100 nanoseconds)
  - Persistent storage = HDD / SSD / NVMe (NVMe = $\sim$  PCIeSSD)
    - Latency = 10s to 100s of usec (SSD/NVMe) to a few ms (HDD)



# DRAM (dynamic random-access memory)

- This is the physical memory...
  - Crucial 16GB Kit (8GBx2) DDR4 3200 MT/S (PC4-25600) CL22 SR X8 Unbuffered DIMM 288-Pin Memory - CT2K8G4DFS832A

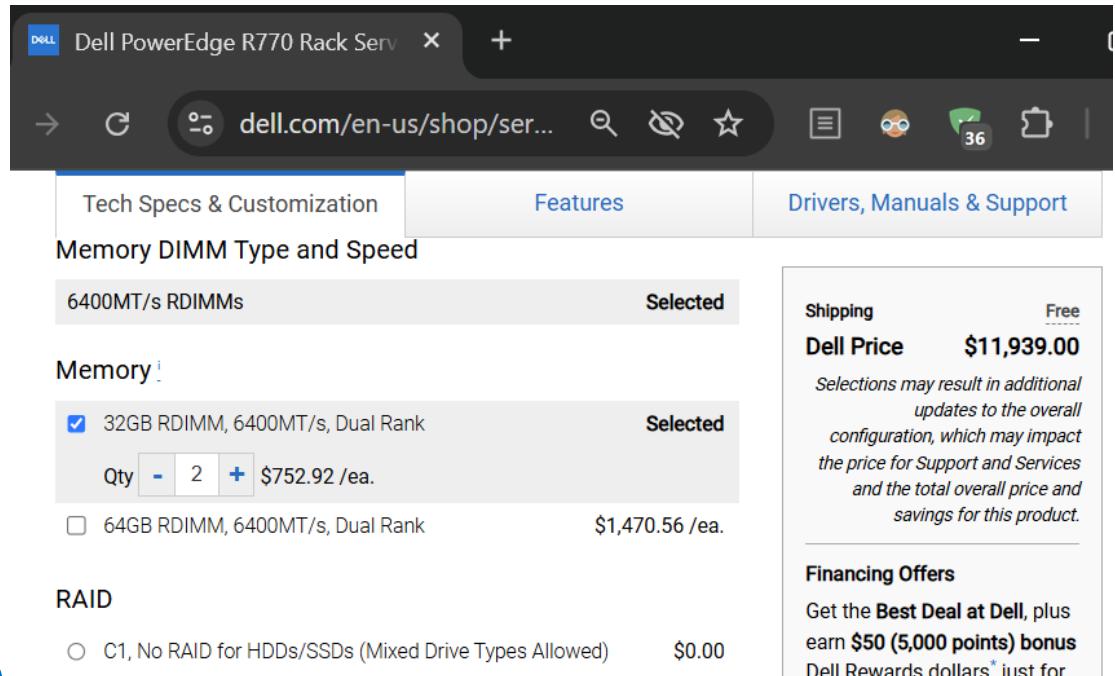


# DRAM (dynamic random-access memory)

- This is the physical memory...
  - Crucial 32GB Kit (16GBx2) DDR4 3200 MT/S (PC4-25600) CL22 SR X8 Unbuffered DIMM 288-Pin Memory - CT2K16G4DFD832A
  - Price this morning (Apr 2021) @ [Amazon](#)
    - 32GB @ \$174 (~ \$5 per 1 GB)
  - Price on May 2013, when I started making this slide
    - 16GB @ \$134 (~ \$8 per 1 GB) for 2x slower mem
  - Bandwidth
    - 3200 (MHz; transfers per sec)
    - x 64 (bit per bus transfer, for DDR4)
    - / 8 (bits per byte)
    - =  $(3200 \times 10^6) \times 64 / 8$  [B/sec]
    - = 25,600 MB/sec = 25.6 [GB/sec/module]  
(= meaning of the “PC4-25600”; GB=10<sup>9</sup>, in this case)
  - Latency (disregarding caching)
    - ≈ 100 ns = a few 100s of CPU cycles



# Server (“enterprise”) DRAM is costlier



The screenshot shows a Dell PowerEdge R770 Rack Server configuration page on dell.com. The 'Memory' section is selected, showing two options: '32GB RDIMM, 6400MT/s, Dual Rank' (selected) and '64GB RDIMM, 6400MT/s, Dual Rank'. The selected option is priced at \$752.92 per unit. The total price for 2 units is \$1,470.56. To the right, a summary box shows a Dell Price of \$11,939.00 with free shipping.

Shipping	Free
Dell Price	\$11,939.00

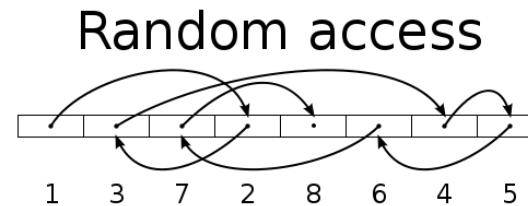
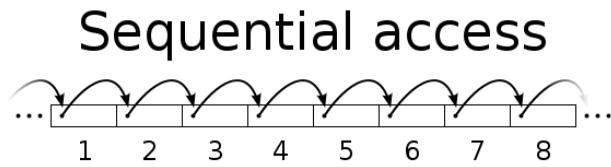
Selections may result in additional updates to the overall configuration, which may impact the price for Support and Services and the total overall price and savings for this product.

**32GB @ 6400 MHz  
this morning (2025-05)**

- **Enterprise @ Dell (of R770 PowerEdge Rack Server)**
  - \$750 (~\$24 per 1GB)
  - DRAM can become the costliest component in a server
- **Consumer @ Amazon**
  - \$95 (~\$3 per 1GB)

# DRAM vs. HDD (spinning hard disk drive) reading 4KB per operation

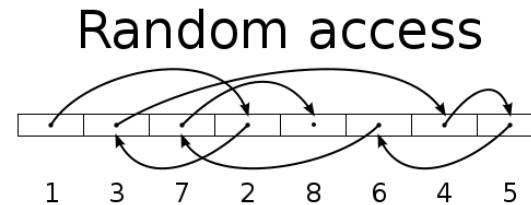
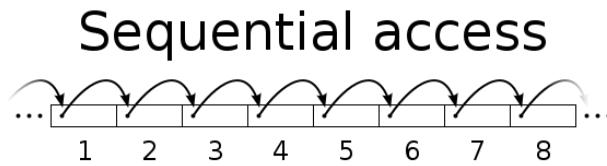
	HDD	DRAM	faster by
availability	persistent (non-volatile)	volatile	
typical sequential access bandwidth	~200 MB/s	~50 GB/s	~250x
typical random-access bandwidth	~1 MB/s	~50 GB/s	~25,000x
typical random-access latency	~5 ms	~100 ns	~50,000x



# DRAM vs. NVMe drive (PCIe SSD) reading 4KB per operation

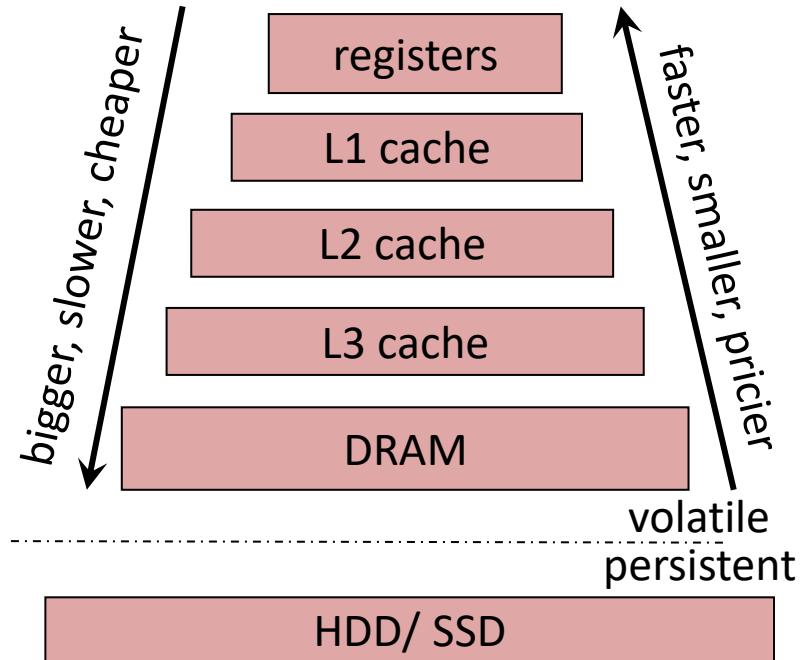
	NVMe drive	DRAM	faster by
availability	persistent (non-volatile)	volatile	
typical sequential access bandwidth	~5 GB/s	~50 GB/s	~10x
typical random-access bandwidth	~1 GB/s	~50 GB/s	~50x
typical random-access latency	~50 us	~100 ns	~500x

\*Unlike HDDs, NVMe drives don't involve mechanical moving parts

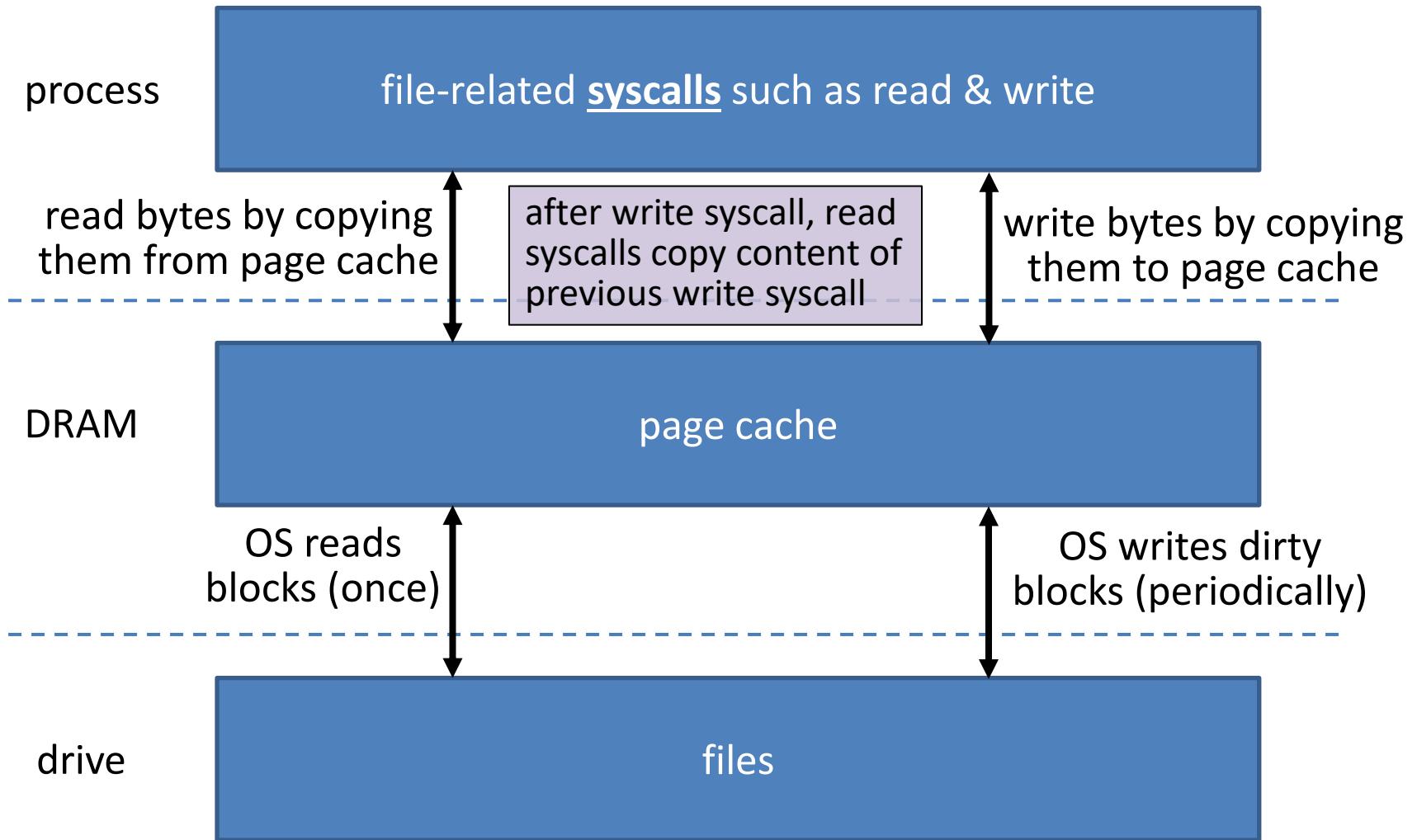


# Page cache

- **Because DRAM is much faster**
  - OS caches disk content in DRAM
- **If a process reads from disk**
  - Data brought to DRAM
  - Stays there until OS decides differently
- **If a process writes to disk**
  - First written to OS DRAM buffer
  - Later sync-ed to disk
- **The memory area that caches (synchronized) disk data**
  - Is called a “[page cache](#)”
  - It’s [managed by software](#) (= the OS)
  - As opposed to the CPU caches over DRAM, which are managed by hardware (the MMU)

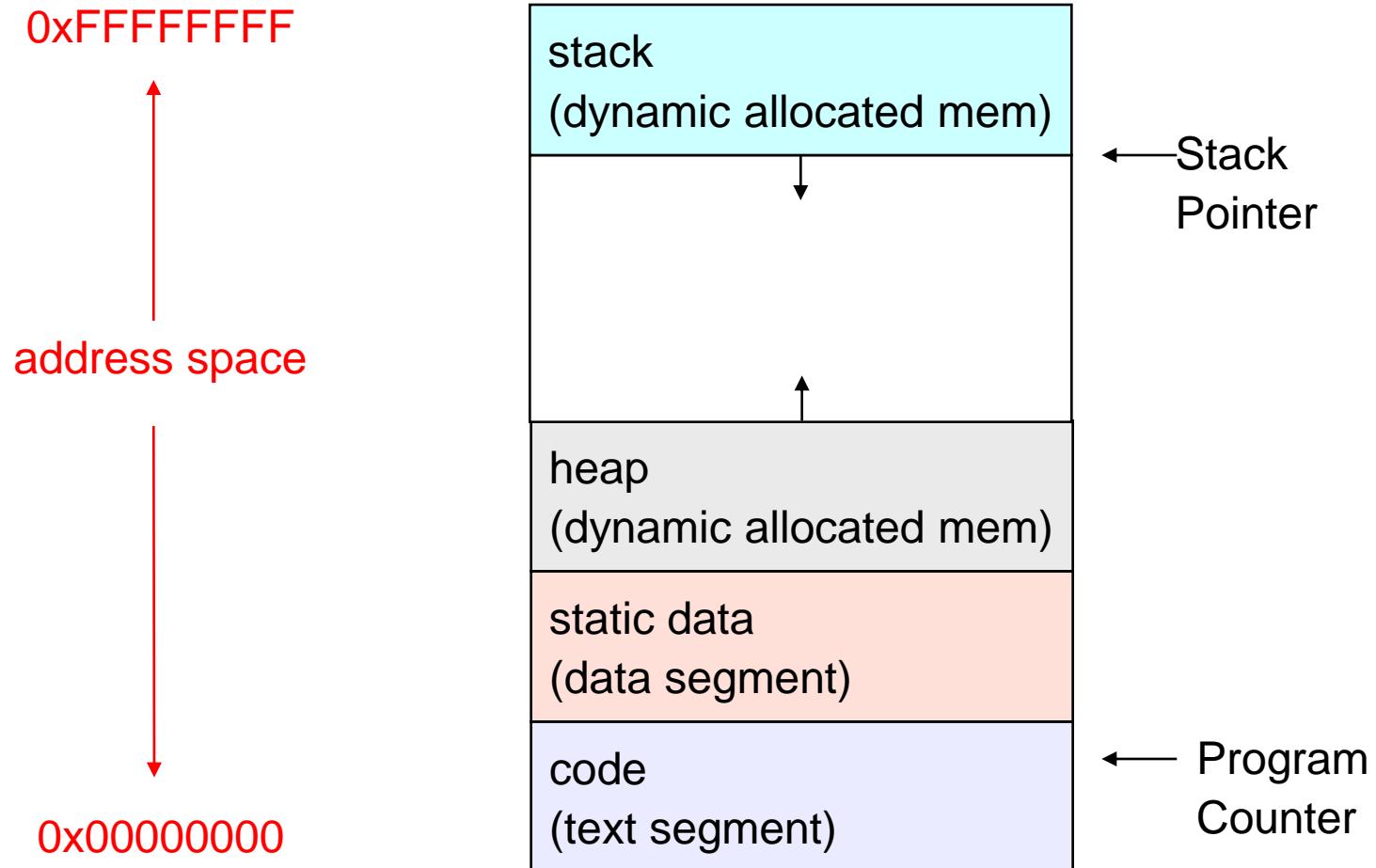


# Page cache



# Reminder

- This is how a program believes its memory looks like...



# But how can this be?

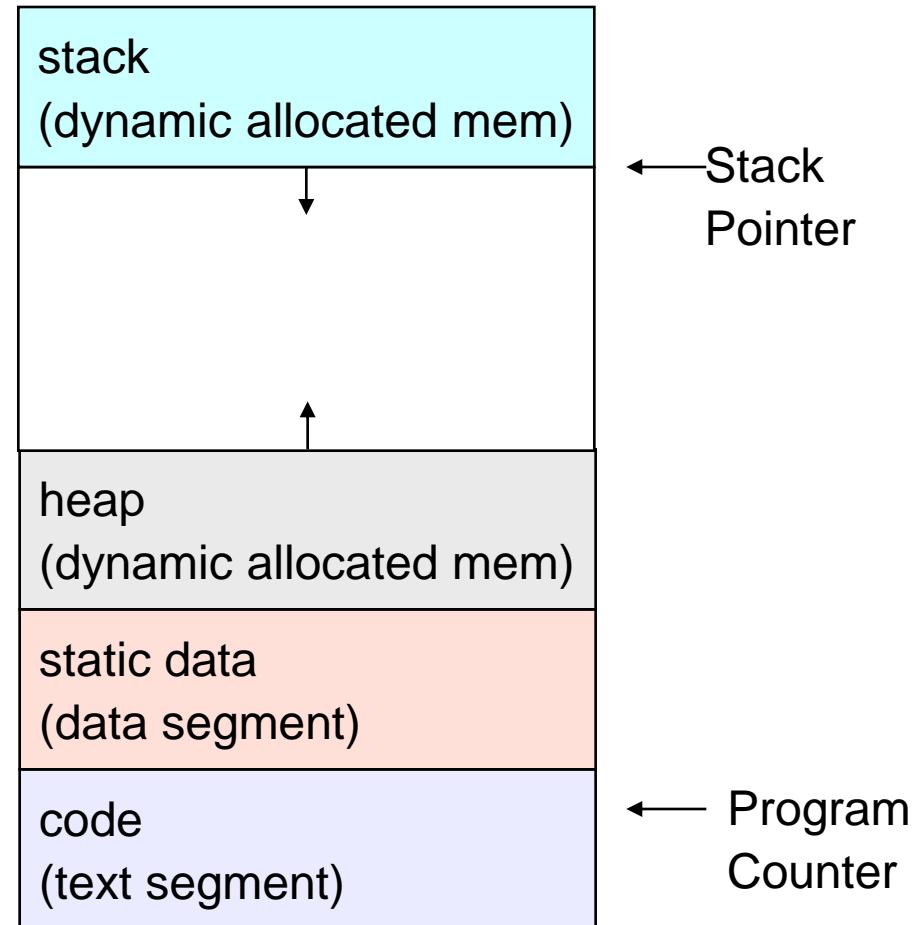
- It's impossible for all processes to share these addresses

⇒ They translate to different locations, so...

⇒ There's an abstraction layer

⇒ This abstraction is called “virtual memory”

- Provides this → illusion to all programs
- And more.  
Specifically... [next page]



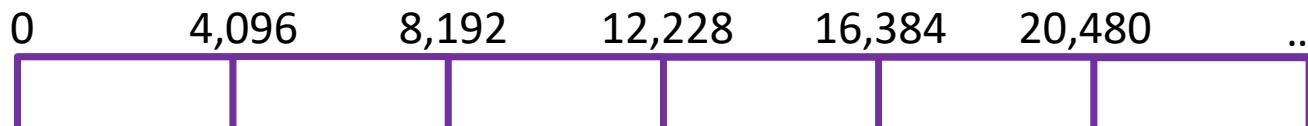
# Virtual memory (vmem) – motivation

- **Per-program Illusion of contiguous memory**
  - Programmers need not worry about *where* data is placed exactly
  - They use the simplistic, ideal address space from the previous slides
- **Isolation between processes**
  - Processes can concurrently run on the same processor
  - Yet vmem prevents them from accessing the memory of one another
  - (Although still allows for convenient sharing when required)
- **Dynamic growth**
  - Can add memory to process's heap/stack at runtime, as needed
- **Illusion of large memory => memory overcommitment**
  - DRAM is costly parts & often the bottleneck resource
  - Vmem size can be bigger than physical memory size
  - Allows for memory “overcommitment”
    - Sum of vmem spaces (across all processes) can be  $\geq$  physical
- **Access control**
  - Decide if individual memory chunks can be read / written / executed

# **HOW VIRTUAL MEMORY WORKS, IN PRINCIPLE**

# Virtual memory – terminology

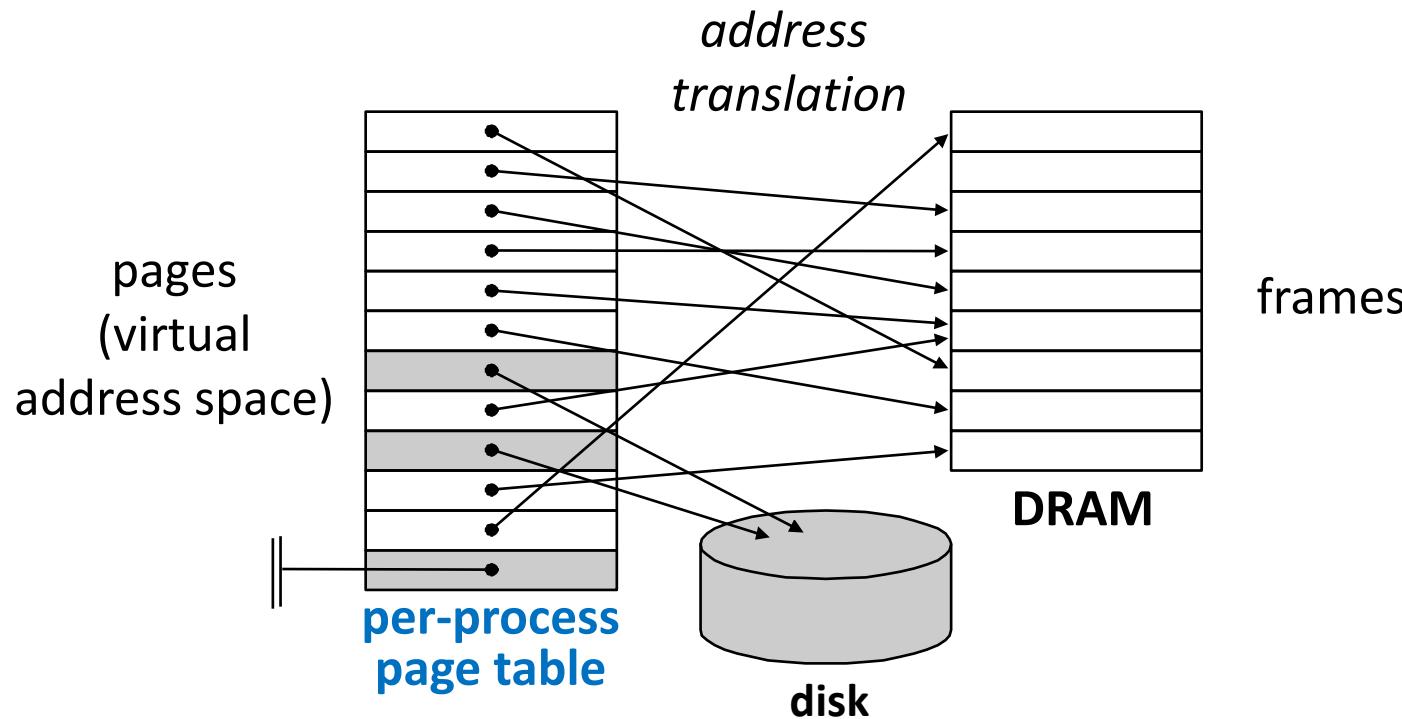
- **Virtual address (VA)**
  - Used by the program/programmer
  - “Ideal” = contiguous & as big as we’d like
- **Physical address (PA)**
  - The real, underlying physical memory address
  - Abstracted away by OS+HW => hidden from programs
- **Memory (virtual & physical) is divided into fixed size blocks**
  - “Page” = chunk of contiguous data (in virtual or physical space)
  - “Frame” = physical memory exactly big enough to hold one page
  - $|page| = |frame| = 2^k$  (bytes)
  - Typically,  $k = 12$ , namely a page (and frame) size is 4KB
- **Pages & frames are always aligned on  $2^k$  boundaries**
  - Both in physical and virtual memory spaces: 0, 4KB, 8KB, 12KB, 16KB, ...



# Virtual memory – basic idea

- “Map” (virtual) pages to frames, such that VA spaces are contiguous
  - Pages can be “mapped” into (associated with) arbitrary frames at arbitrary locations
- Pages can reside
  - Either in memory (used recently)
  - Or on disk (used not recently; they’ll be paged-in on demand)
  - Or be simply unallocated yet (& then allocated on-demand)
  - (We thus allow for the aforementioned memory overcommitment)
- All programs are written using VAs
  - And VAs are seamlessly translated into PAs
  - As we will see later, **translation is a HW/SW mechanism**, whereby
    - OS sets the VA=>PA mappings = governs the “control plane”
    - HW does on-the-fly translation from VA to PA = “data plane”

# Per-process virtual memory simplistic illustration

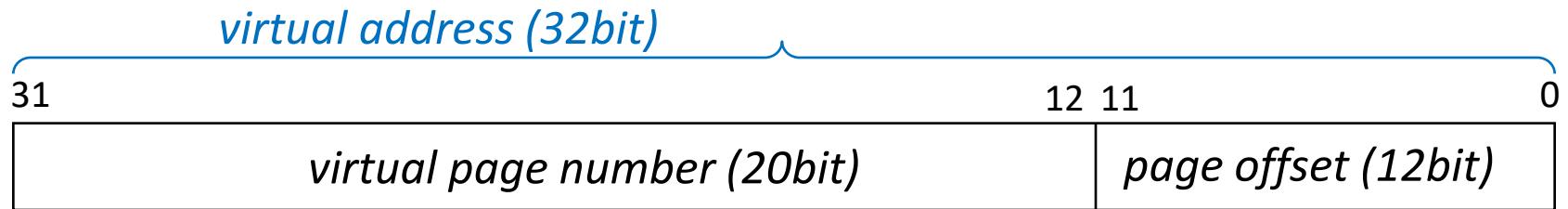


rpt

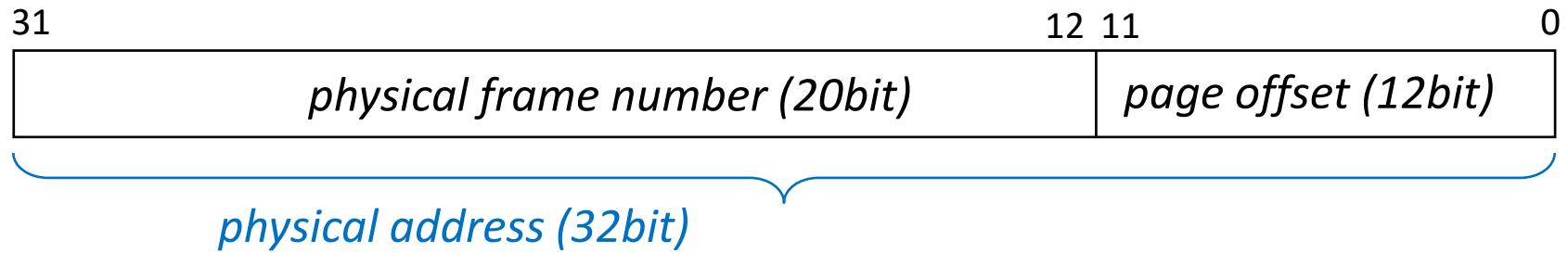
## • Immediate advantages

1. Illusion of contiguity & of having more physical memory
2. Actual physical location (of program and its data) unimportant
3. Dynamic growth, isolation, & sharing are easy to obtain
4. On-demand allocation

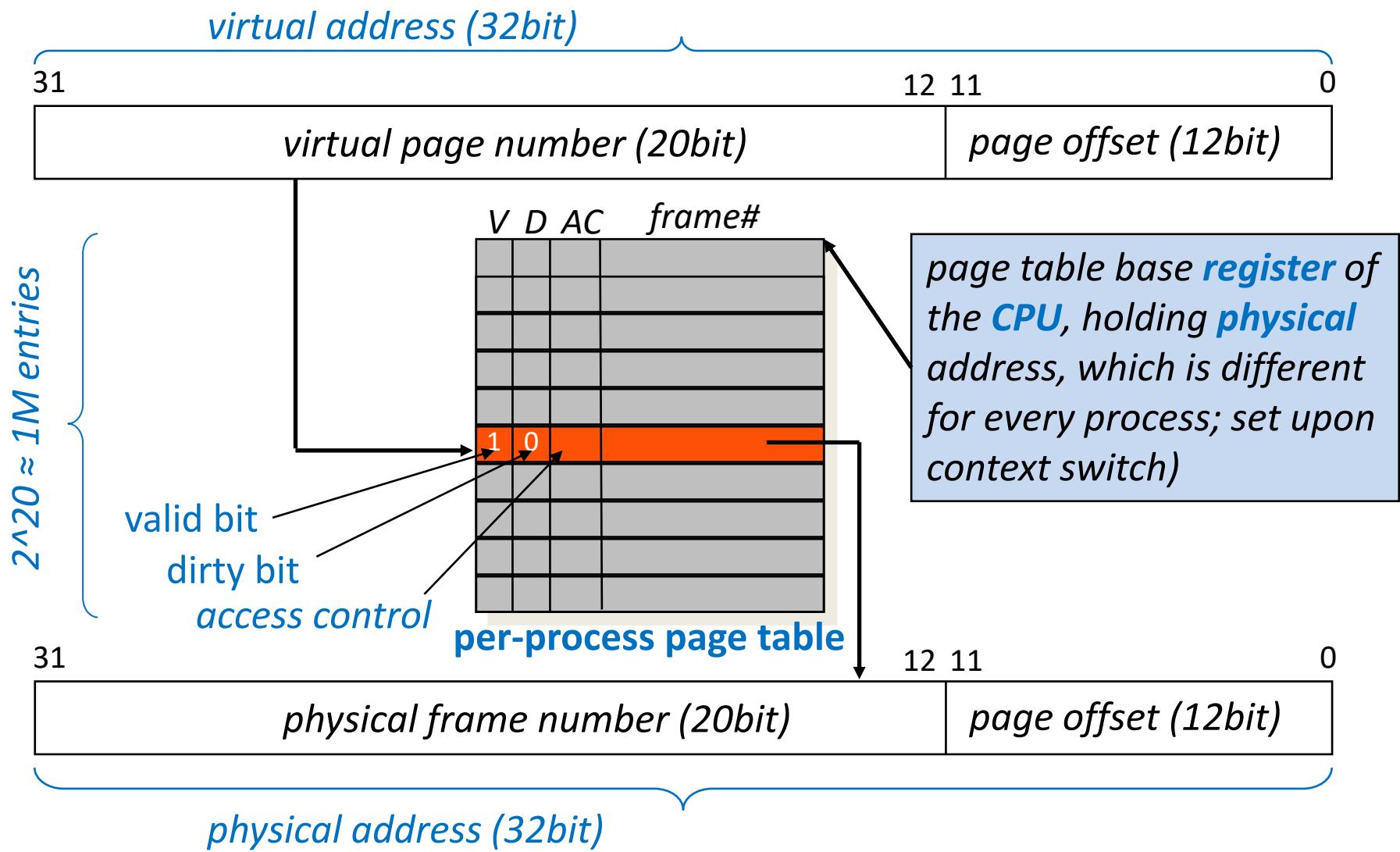
# How to map (assume 32bit address)?



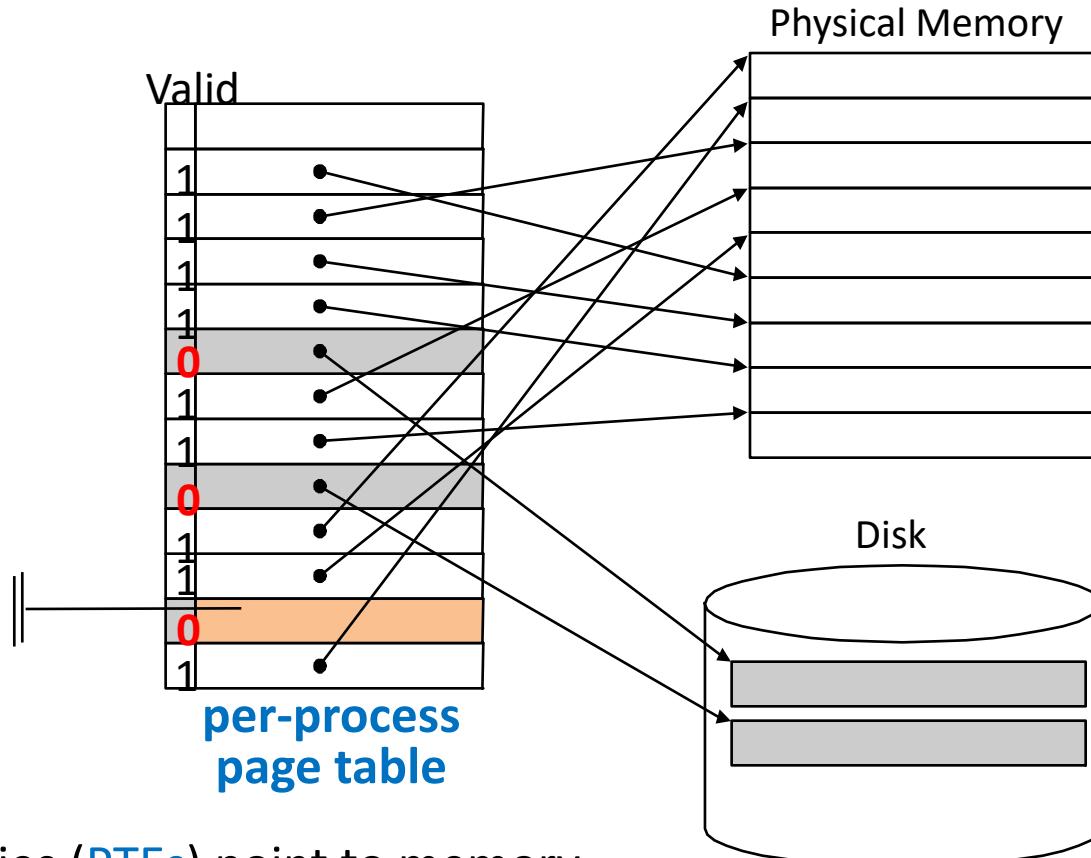
?



# Use per-process “page table” (in DRAM)



# Valid bit



Page table entries ([PTEs](#)) point to memory frame addresses, or disk addresses, or nothing

# Upon (each & every) memory access

- **If ( valid == 1 ) // otherwise HW generates page fault interrupt**
  - Page is in main memory @ PA stored in table => data can be used
- **Else if ( page resides on disk ) // “major” page fault**
  - OS suspends process
  - Fetches page from disk and adds VA=>PA mapping to page table
  - Resumes process, which will re-execute faulting instruction
- **Else if ( should be but still isn't allocated ||  
in page cache but not in page table ) // “minor” page fault**
  - OS allocates page if needed & adds VA=>PA mapping to page table
  - (Q: can we always avoid going to disk upon minor page faults?)
- **Else if ( some other legal situations for which OS is responsible )**
  - Resolve (example: CoW; see later)
- **Else // process performed an illegal memory ref**
  - OS (which handles the page fault interrupts) delivers a signal (SIGSEGV/SIGBUS) to the offending process
  - The matching signal handler terminates process by default

# Upon (each & every) memory access

- **Access Control:** R=read-only, R/W=read/write, X=execute
  - If access type incompatible with specified access rights
    - ⇒ protection violation fault
    - ⇒ interrupt
    - ⇒ resolve if OS's fault (as in CoW),  
or deliver signal if process's fault

# Major page fault flow

- Major => need to retrieve page from disk drive
  - 1. CPU detects the situation (valid bit = 0)
    - It cannot remedy the situation on its own
      - CPU doesn't communicate with drives
      - Moreover, CPU has no say regarding page table contents
  - 2. CPU generates interrupt and transfers control to the OS
    - Invoking the OS page-fault handler
  - 3. OS regains control, realizes page is on drive, initiates I/O read ops
    - To read missing page from drive to DRAM
    - Possibly need to write victim page(s) to drive (if no room & dirty)
  - 4. OS suspends process & context switches to another process
    - It might take a few milliseconds for I/O ops to complete
  - 5. Upon read completion, OS makes suspended process runnable again
    - It'll soon be chosen for execution
  - 6. When process is resumed, faulting operation is re-executed
    - Now it will succeed because the page is there

# Minor page fault flow

- **No need to go to drive to resolve the fault**
  - Instead, we can resolve fault quickly
  - No need to suspend process => it remains runnable
- **Examples**
  1. **CoW (copy-on-write)**
    - Used to, e.g., implementing fork()
      - Map child pages to parent frames + make pages read-only
      - When child writes => page fault => create a private copy for it
  2. **Demand-based (lazy) memory allocation**
    - OS allocates memory “lazily” – pages are truly allocated only when actually used for the first time
  3. **Reading a file content that is already found in the “page cache”**
    - OS caches previously read files, possibly by other processes
    - All read/write ops go through the page cache

# Terminology

- “**Page in**” & “**page out**” a chunk of data
  - Page in  $\Leftrightarrow$  copy page from disk to DRAM (= read)
  - Page out  $\Leftrightarrow$  copy page from DRAM to disk (=write)

# The mmap system call

- **Prototype**
  - ```
void *mmap(void *p, size_t len, int prot, int flags, int fd, off_t offset);
// prot = PROT_{EXEC, READ, WRITE, NONE}
// flags = MAP_{SHARED, PRIVATE, ANONYMOUS, LOCKED, POPULATE, ...}
```
- **Map a given file to a given virtual address range in the process's memory space**
  - A file becomes an “array of bytes” (backed by disk), and gets a VA
  - With the right mmap() flags, reading/writing from/to the array (more or less) translates to reading/writing from/to the file
  - Writes will be made durable on disk “soon” = every few secs, in batch
  - If we want to make writes durable now, use the fsync(fd) system call
  - A mmap()ed file is said to be a “[memory-mapped file](#)”
- **Implementation leverages the page cache mechanism**
  - Recall: all disk load/store ops are cached in memory by page cache
  - By default, all I/O goes through page cache, not directly to disk
  - Mmap sets process's PTEs to point to the page cache pages

# The mmap system call

- **Pros of mmap()** (relative to read()ing and write()ing)
  - Can make programming easier
  - Saves overhead of read/write system calls = user/kernel **switches**
  - Which means we get “zero copy” I/O
    - Saves **copying** between user and kernel buffers
    - Which may reduce memory pressure

# The mmap system call

- **Anonymous pages**
  - Heap/stack pages; not mmap-ed files
- **Can be allocated by mmap with flag=MAP\_ANONYMOUS**
  - Namely, can implement malloc with mmap, which is invoked whenever malloc runs out of memory
  - Alternatively, malloc can be implemented using the system calls brk() and sbrk()
  - Homework: browse through the man page of mmap and (s)brk
- **Named pages**
  - Backed by a file (via mmap-ing)
- **“Shared” and “private” mmap-ing**
  - **MAP\_SHARED** = changes affect the actual file (visible to other processes)
  - **MAP\_PRIVATE** = changes don't affect the actual file; rather, there will be a CoW for the process (other processes will not see changes)

# The mmap system call

- **Cons of mmap()**
  - Mapped buffer must be page-aligned
  - There's a subtle **consistency issue** arising from an interaction between `mmap(MAP_PRIVATE)` and `write(fd)`:
    - POSIX doesn't specify whether changes made to the file via `fd`, [after the `mmap\(\)` call](#), are visible in the mapped region
    - Namely,
      - P1: `arr = mmap(MAP_PRIVATE, some_file)`
      - P2: `fd = open(some_file);`  
`write(fd, "...");`
      - What's `arr[0]`?
      - Is it the information there before the write? After?
      - Unspecified.

# The mmap system call

- “mapped” <> `mmap()`ed
  - According to our definition of “mapping” from a few slides ago
    - Mapping a page to address space of a process is the act of connecting a virtual page to a physical frame in the process’s page table

# Q/A

- Q: do we need to page out named pages?  
A: only if they are “dirty”
- Q: when we read() a file’s page (with the read syscall), is it named?  
A: seemingly no, but actually depends on the buffer pointer passed to the read system call
- Q: is reading (via read() or mmap-ed buffer) always slow as disk?  
A: depends on whether or not the corresponding area being read already resides in the page cache
- Q: is writing (via write() or mmap-ed buffer) slow as disk?  
A: no, writing is asynchronous (explicit fsync-ing, however, *will* make it slow by making the process wait until the data is actually written to disk)
- Q: can anonymous pages reside on disk?  
A: yes [see next slide]

# Swapping

- **Swap space**
  - Disk area (file) where anonymous pages are written, if the OS decides they have no room in DRAM
  - Page is said to be “swapped out” when this occurs (and “swapped in” for the reverse operation)
  - Swap area contains anonymous pages (including mmap anonymity)
- **Swapping vs. paging**
  - In the olden days, “swapping in/out” sometimes referred to the entire memory of a process (not just to a certain page(s))
  - Nowadays people often use “paging” and “swapping” interchangeably
  - <http://en.wikipedia.org/wiki/Paging> makes the following distinction
    - Page in = transfer page from anywhere on disk to DRAM
    - Swap in = transfer a page from swap-space to DRAM
    - But we typically do not use this distinction

# On-demand paging & readahead

- **On-demand paging**

rpt

- OS maps / reads page from disk into DRAM only if/when the process attempts to access it for the first time (and, hence, a page fault occurs)
- That is, OS pages-in data only via page faults (+ prefetching; see below)
- Thus, a process begins execution with most pages “unmapped”, possibly not residing in physical memory (executable code included), and page faults occur until its pages are placed in in DRAM & mapped
- Also called “lazy” loading
- What’s the benefit? Reading/mapping is costly, & with demand-paging we only read what we need

- **Notice: on-demand allocation also works**

rpt

- For anonymously mmap-ed / (s)brk-ed memory

- **Readahead prefetching (anticipatory paging)**

- `read()` does prefetching when identifying sequential access
- The page fault handler does the same when doing on-demand paging
  - Complements demand-paging in an attempt to minimize page faults

# Working set

- **Informal definition:** pages that a process currently works on
  - For some definition of “currently”
- **Formal definition:**  $WS_p(k)$ 
  - Pages accessed by process (or thread) P in the last k accesses

| time          | 1        | 2        | 3 | 4 | 5        | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 |
|---------------|----------|----------|---|---|----------|---|---|---|---|----|----|----|----|----|----|----|----|
| page accessed | 2        | 6        | 1 | 5 | 7        | 7 | 7 | 7 | 5 | 1  | 6  | 2  | 3  | 4  | 4  | 4  | 3  |
| WS( $k=3$ )   | <b>2</b> | <b>6</b> | 1 | 5 | <b>7</b> | 7 | 7 | 7 | 5 | 1  | 6  | 2  | 3  | 4  | 4  | 3  | 4  |
|               |          | 2        | 6 | 1 | 5        | 5 |   |   | 7 | 5  | 1  | 6  | 2  | 3  | 3  | 2  |    |
|               |          | 2        | 6 | 1 | 1        | 7 |   |   | 7 | 7  | 5  | 1  | 6  | 2  | 3  | 2  |    |

- For a fixed value of k, smaller  $WS_p(k)$  indicates more locality
- During time 1–17, there are 7 pages that p accesses
  - We informally call these pages p’s current “working set”
- What happens if the current working set is too big to reside in DRAM (and/or if memory is currently tight such that there isn’t enough of it to house the current working set)

# Thrashing

- **When we've overcommitted too much memory**
  - There isn't enough physical memory to back the currently-in-active-use virtual memory
- **The system might enter a state of “thrashing”, that is**
  - Virtual memory is in a constant state of paging, rapidly exchanging data between memory & disk
  - Nearly nothing else is done in the system (causes performance to degrade/collapse)

# Virtual memory: did we achieve our goals?

- **Illusion of contiguous memory**
  - Yes: virtual memory is contiguous by definition
- **Illusion of large memory (possibly bigger than physical mem)**
  - Yes: chunks that don't fit into physical memory may reside on disk and on-demand allocation also helps
- **Dynamic growth**
  - Yes: heap & stack can grow at runtime by mapping more VAs to PAs, in order
- **Isolation between processes**
  - Yes: same VAs point to different PAs; as long as we keep PAs disjoint on a per-process basis, the processes are isolated
  - Still, sharing memory is possible and easy (how?)
- **Memory overcommitment**
  - Yes: using the disk,  $\sum_{i=1}^n vmem_i$  (for n processes) can be > physical size
- **Access control**
  - Yes: HW enforces PTE (= page table entry) bits; e.g., it will reject a write to a page that is marked 'read-only'

TLB (translation-lookaside buffer) & page reclamation

## **VIRTUAL MEMORY PERFORMANCE**

# But how does it perform?

- **(1) Temporal locality helps**
  - Typically, during a given time interval, a process often uses only a fraction of its memory, over and over again
  - So it's fine to keep currently unused parts on disk
  - As long as the working set is in memory most of the time
- **(2) Asynchronous nature of OS, when doing I/O, helps**
  - Writes are non-blocking by default: when writing a page to disk, we don't need to block the associated process
  - When reading stuff, we can run other processes
- **(3) Demand-allocation and paging help**
  - Pages fetched from secondary memory only upon the first page fault, rather than, e.g., upon file open – we bring only what we need
  - Likewise, page allocation is done only when needed

# But how does it perform?

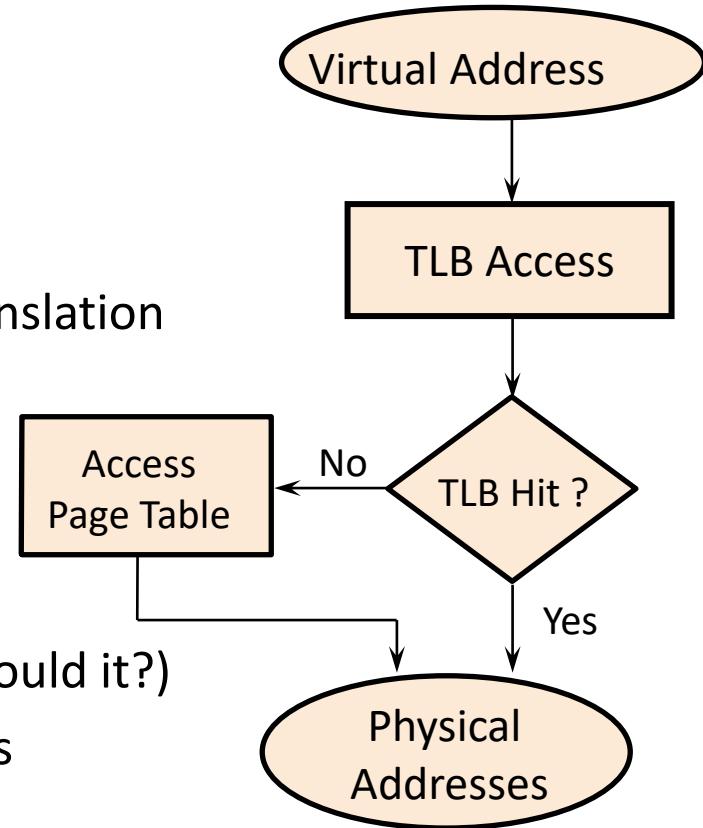
- **(4) Special locality helps**
  - First, we work in page resolution = compatible with special locality
  - Second, when reading page P from disk, we may employ the **readahead** optimization = bring in some additional file pages from immediately after/before P, even though they weren't accessed yet
  - rpt As we've learned, “**locality principle**” suggests these pages would be used soon
    - Locality = special locality + temporal locality
    - Locality principle contends that most programs exhibit special and temporal locality when utilizing the memory
  - So some pages don't induce major page faults when accessed for the first time; they are simply “there”

# But how does it perform?

- **(5) Making VA=>PA translations fast helps**
  - The TLB (translation lookaside buffer) is a small, fast HW structure that caches recently used VA=>PA mappings
    - Size of TLB L1 can be, e.g., 32-64-128
    - Size of TLB L2 can be hundreds of entries (1536 in Intel chips)
  - Given a VA, HW first searches for its translation in the TLB; only if it's not there HW access the in-memory page table
  - Accessing the TLB is fast (similarly to L1 & L2 or faster)
  - Even though the TLB is relatively small, locality principle typically ensures it is rather effective
    - Special locality: since we work in 4KB page granularity, lots of nearby accesses fall in the same page
    - Temporal locality: same pages are used repeatedly
  - Lots of workloads (though certainly not all) approach 100% TLB hit rate

# VA => PA translation with TLB (Translation Lookaside Buffer)

- **Page table resides in memory**
  - Each translation requires a memory access
  - Required for *each* load/store!
- **TLB**
  - Cache recently used PTEs => speeds up translation
  - Typically:
    - HW fills TLB automatically by reading the page table on its own (no SW involvement)
    - OS can invalidate TLB entries (when should it?)
      - But must synchronize TLBs of cores
      - Called “TLB shootdown”
    - Processes are completely unaware of the TLB



# But how does it perform?

- **(6) Using an intelligent page replacement policy helps**
  - When we need to evict a page from memory to disk
    - The page replacement policy decides which page it'll be
    - Chosen page called the “victim” page.
  - Also called “page reclamation policy”.
  - Goal
    - Minimize number of future page faults
    - Minimize price of paging (evicting dirty pages costs more; why?)
  - Typically done via a daemon process (“swapper”) that runs in the background
    - Start: when number of free frames drops below some threshold
    - Stop: when number of free frames exceeds some threshold

# Page reclamation algorithms

- **Belady – optimal, but theoretical**
  - Greedily page out page accessed furthest in the future
  - An “off-line” algorithm = we know the future memory accesses
- **LRU (least recently used) – not theoretical, but still impractical**
  - Leverages the locality principle
  - But typically too wasteful to be used in practice
    - OS data structures must be updated up on *each* mem ref
- **Clock – a practical LRU approximation**
  - Pages are cyclically ordered
  - Maintain a per-frame bit: “was it referenced since I last checked?”
    - Cleared every n seconds (a linear pass “zeroing round”)
    - Set upon first access since zeroing round (implemented in SW or HW)
  - “Current” iterator points to last examined page frame
  - Search clockwise for first page with bit=0 – this is the victim page
  - Set bit=0 for pages with bit=1

# Page reclamation algorithms

- **NRU (not recently used) – a more sophisticated LRU approx.**
  - Can be based on ‘clock’
  - Underlying principles (order is important):
    - Prefer keeping referenced over unreferenced
    - Prefer keeping modified (dirty) over unmodified
  - HW or SW maintain per-page ‘referenced’ & ‘dirty’ bits
    - Periodically (every n seconds), SW turns ‘referenced’ off
  - Replacement algorithm partitions pages to
    - Class 3: referenced, dirty
    - Class 2: referenced, not dirty
    - Class 1: not referenced, dirty
    - Class 0: not referenced, not dirty
  - Choose victim page from the numerically smallest class

# Page reclamation algorithms

end of 1st lecture

- **More recent algorithms**
  - Consider frequency
  - Not just recency
  - **Homework:** see, for example, ARC = adaptive replacement cache
    - [https://en.wikipedia.org/wiki/Adaptive\\_replacement\\_cache](https://en.wikipedia.org/wiki/Adaptive_replacement_cache)

# **Operating Systems (234123)**

## ***Virtual memory – implementation***

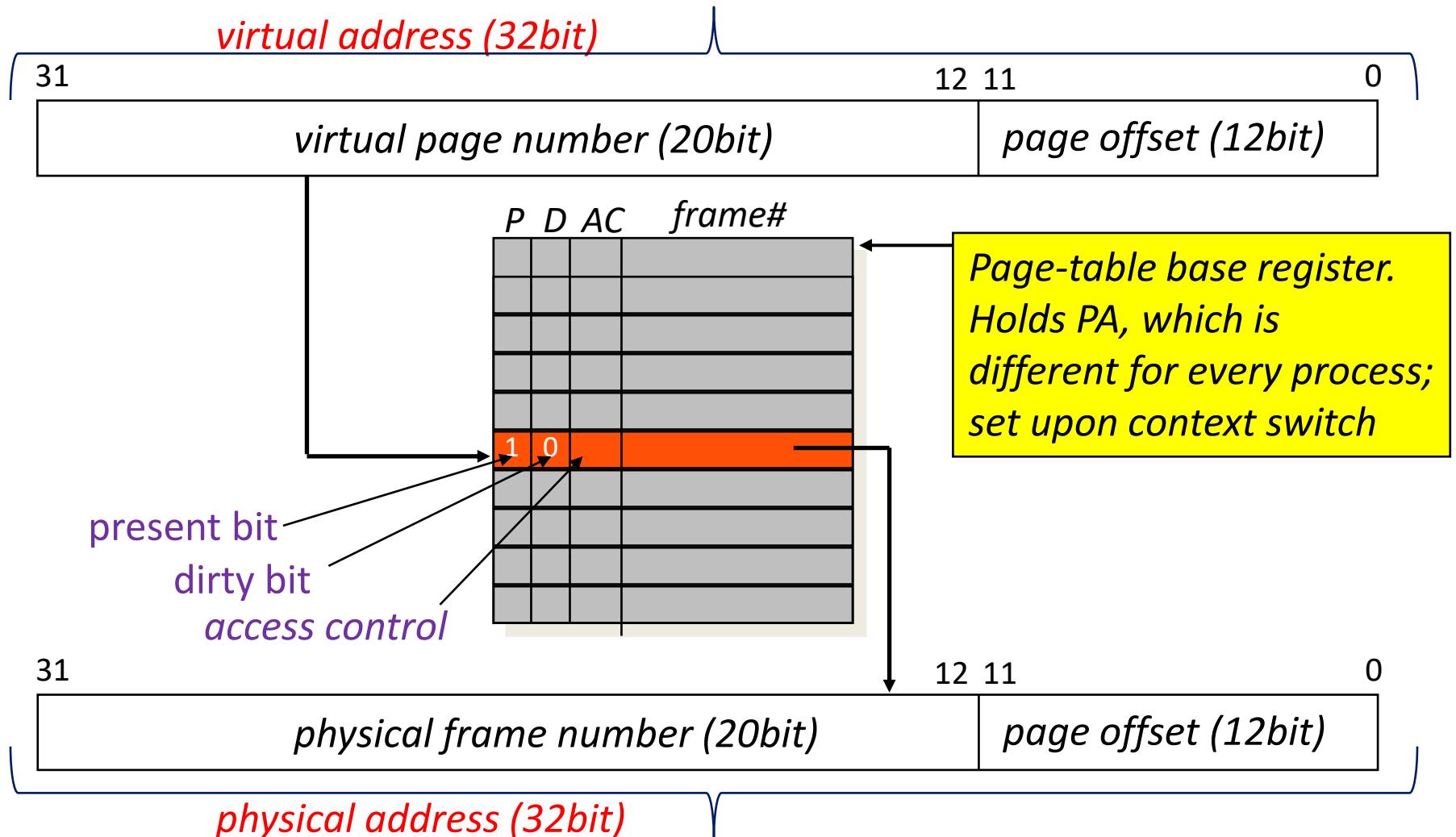
Dan Tsafrir (2025-06-09)

Partially based on slides by Hagit Attiya

Hierarchical translation (radix tree)

# **HOW IT WORKS IN X86**

# Reminder: 32bit address translation

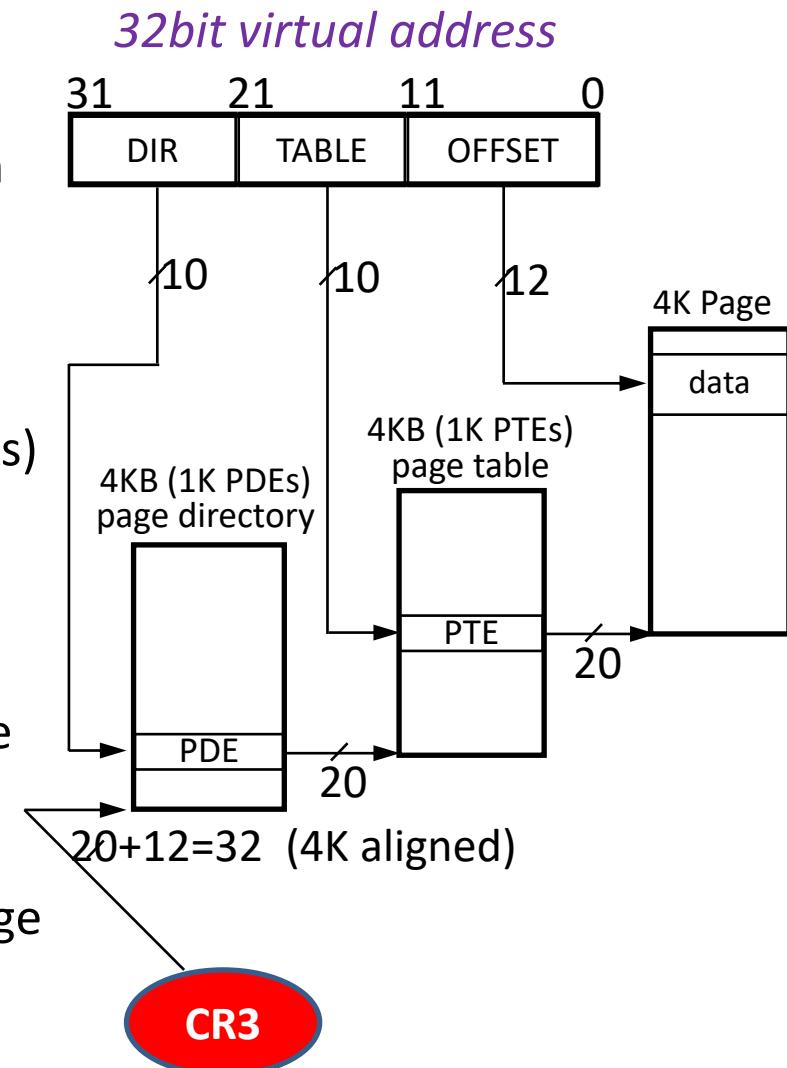


# 32bit x86 address translation

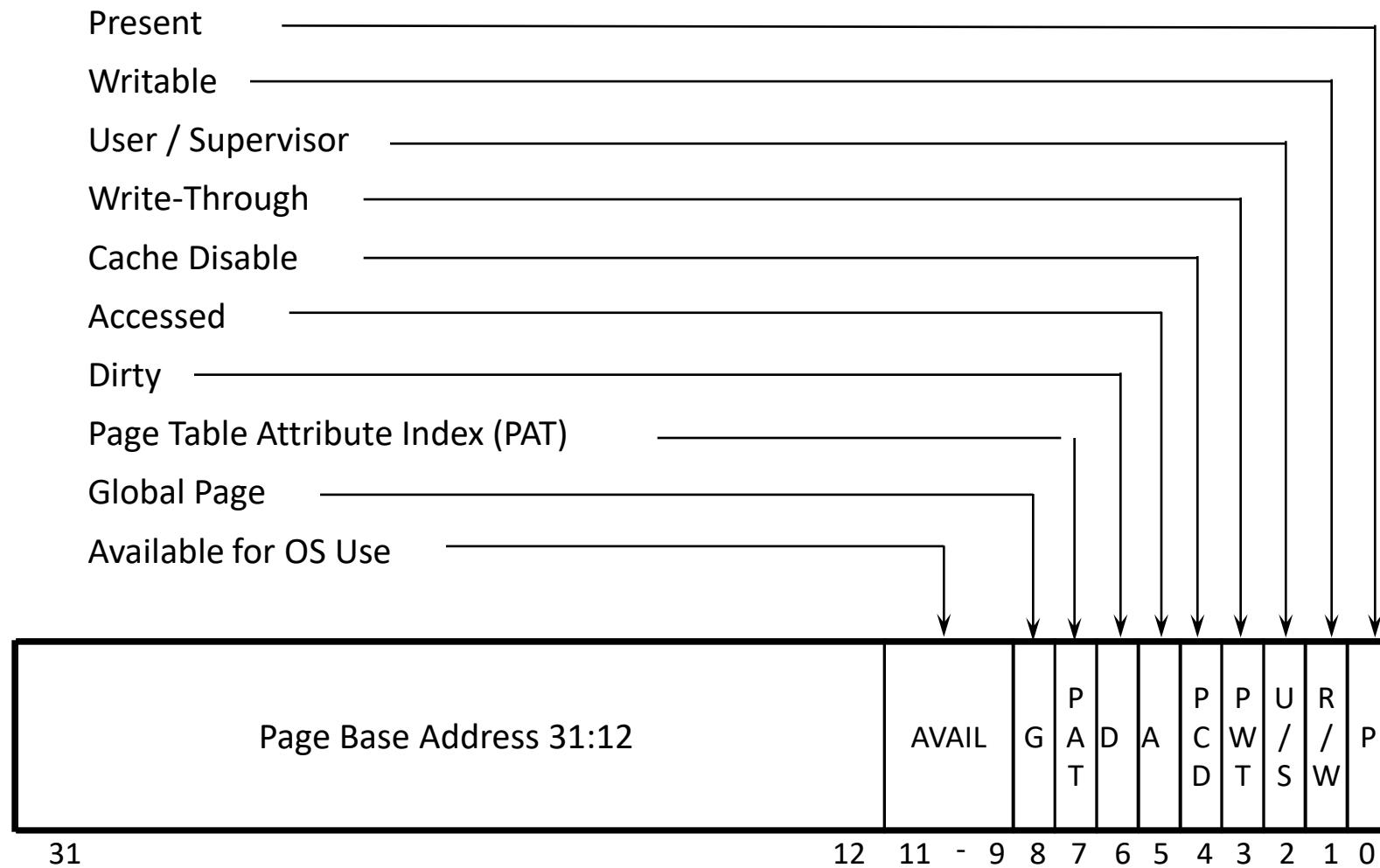
- **32bit address means  $2^{32} = 4\text{GB}$  address space**
  - There are  $2^{20} = 1\text{MB}$  pages ( $1\text{MB}$  pages  $\times$   $4\text{KB}$  per-page =  $4\text{GB}$ )
- **The job of the x86 virtual memory subsystem**
  - Translate 20 bits (= virtual page #) to 20 bits (= physical frame #)
  - Pages are  $4\text{KB}$ -aligned, so it's enough to identify them with 20bits
- **Every process has a “page directory” =  $4\text{KB}$  page**
  - Holds 1024 PDEs (page-directory entries)
    - Each PDE is comprised of 4 bytes (= 32 bits;  $1024 \times 4 = 4\text{KB}$ )
  - Each PDE contains
    - Bits: Present? Accessed? Dirty? Writable? User? Cache Disabled? ...
    - 20bit physical page frame number:
      - Where to find the corresponding “page table”
- **Every PDE points to a  $4\text{KB}$  “page table”**
  - Holds 1024 PTEs (page-table entries) with same bits + 20bit frame#
  - Frame# points to a program's memory page

# 2-level hierarchy

- **CR3 register points to page-directory**
  - Physical address set on context switch
  - Per process (but threads share it)
- **DIR (10 bits)**
  - Index of PDE in page-directory array (there are 1024 PDEs =>  $|PDE| = 4\text{bytes}$ )
  - Each PDE holds 20bit of 4KB-aligned physical frame# of a 4KB page table
- **TABLE (10 bits)**
  - Index of PTE in page-table array (there are 1024 PTEs =>  $|PTE| = 4\text{bytes}$ )
  - Each PTE holds 20 bit of 4KB-aligned physical frame# of a 4KB “regular” page
- **OFFSET (12 bits)**
  - Offset within the selected 4KB page



# 4KB-page PTE format



# Combining user/supervisor & global page bits

- **Problem: when a system call happens...**
  - OS must run and use its own internal data structures
  - But we don't want every system call to induce a context switch
- **Solution: map OS space to address space of all processes**
  - Use the “user/supervisor bit” to indicate that only the OS (“ring 0”) can access this memory area, whereas user code (“ring 3”) cannot
  - Further, set the “global page” bit in the PTEs associated with the OS memory, which would then leave the corresponding TLB translations valid across context switches
- **The “meltdown” attack negates the above on vulnerable CPUs**
  - On vulnerable CPUs, OSes employ PTI = page table isolation
    - Kernel has its own page table, making syscalls costlier (a full C.S.)
  - The PCID (process context ID) x86 hardware feature alleviates this cost
    - Each process has its own PCID, and TLB entries are tagged with PCIDs
    - Only TLB entries of the current PCID are in effect at any given time
    - So no need to flush TLB content upon context switch

# Why hierarchical?

- **The alternative: one linear page table**
  - Requires  $2^{20}$  (PTEs)  $\times$  4 (bytes per PTE) = 4MB
  - Could be wasteful (recall, it's a per-process overhead)
  - And of course, this approach will be more wasteful ( $\Leftrightarrow$  to the point of being impossible) for 64bit addresses (discussed later)
- **The hierarchical translation allows us to avoid wasting memory**
  - Page tables (2<sup>nd</sup>-level) are allocated on-demand only
  - Less “internal fragmentation” when memory space is sparse
    - “Internal fragmentation”  
= when parts of an allocation unit remain unused
    - Q: Is there no internal fragmentation when using hierarchical translation?
    - What’s “external fragmentation”?

# HW/OS cooperate

- **HW defines data structures**
  - Structure of hierarchy, PTE bits, etc.
- **OS determines most of the content of page directories & page tables**
  - It explicitly sets the values of the PDEs and PTEs
- **HW does the “table walk” automatically**
  - If VA=>PA translation not found in the TLB
    - Called “TLB miss”
  - HW knows where to find the root page directory (using CR3)
  - HW walks the tables, hierarchically, until it reaches the data page
  - It inserts the VA=>PA translation to the TLB
    - When page-faulting operation resumes, there will be a TLB hit
- **HW also responsible for setting bits**
  - Accessed & dirty bits (Why? Recall that OS can emulate this behavior)
  - OS is responsible for turning these bits off

# HW/OS cooperate

- **HW populates TLB**
  - Whenever TLB miss occurs
- **OS invalidates TLB entries**
  - E.g., upon munmap() or context switch (if HW doesn't support PCID in TLB; nowadays it typically does, but the number of PCIDs is limited)
- **OS is also responsible for synchronizing between TLBs**
  - Each core has its own TLB
  - **TLB shootdown**: when, e.g., one core invalidates PTE or changes access, OS must sync TLBs on other cores

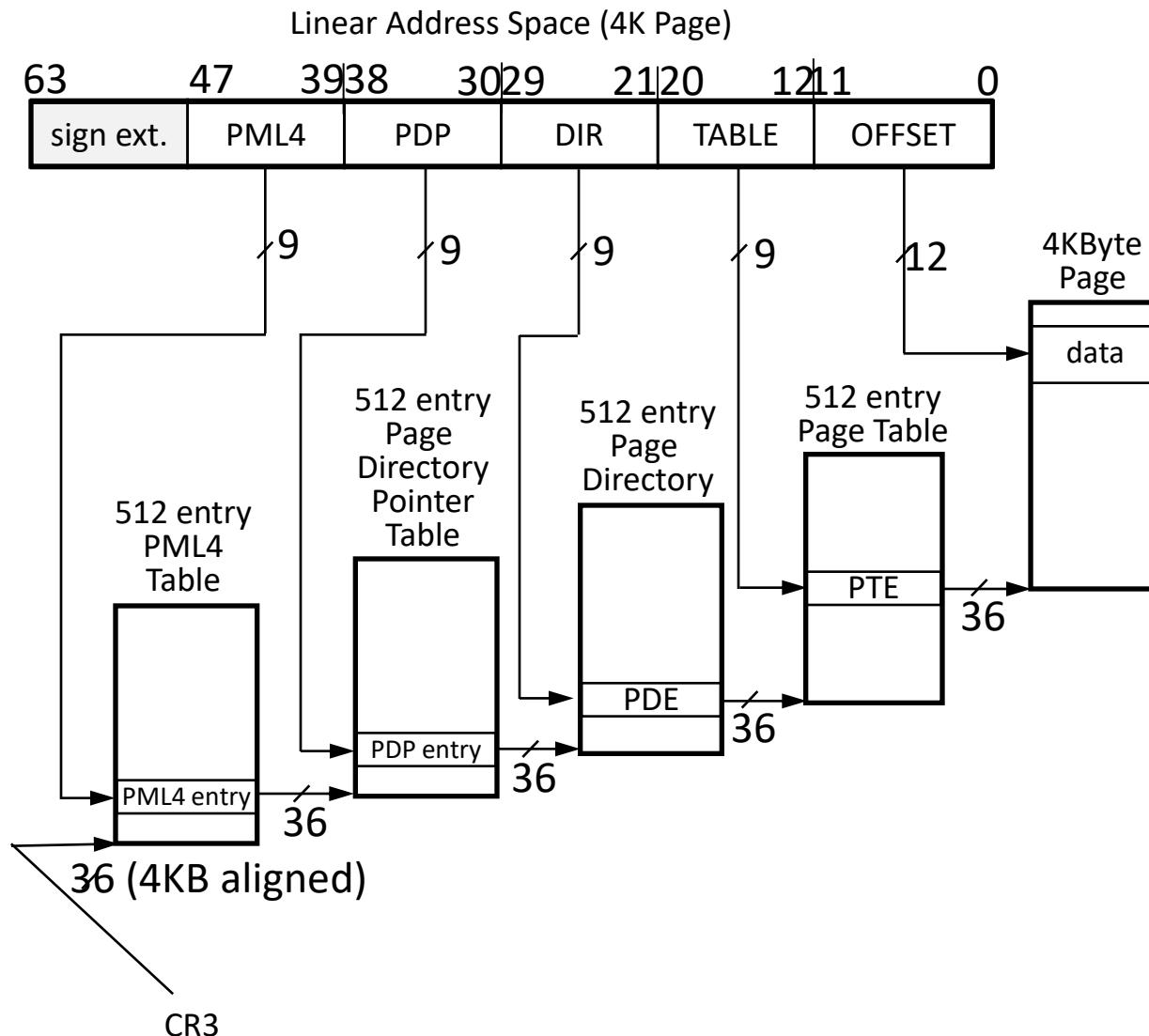
# Locality helps address translation

- **PTE “prefetching”**
  - CPU reads data from DRAM in cache-line resolution
    - Cache-line size = 32 bytes in x86 32bit
  - So, whenever the CPU reads one PTE (= 4 bytes)
    - It actually inserts another 7 PTEs to the cache (to L3 and L2 and possibly also to L1, depending on model)
      - $32\text{bytes} / 4\text{bytes} = 8 \text{ PTEs}$
    - The eight PTEs reflect contiguous virtual space (special locality)
    - If/when the core encounters a TLB miss on one of the 7, it'll find them in the cache and so will not have to read it from DRAM
- **For x86 64bit**
  - PTE is twice as long (8 bytes)
  - But cache line is twice as long too (64 bytes)

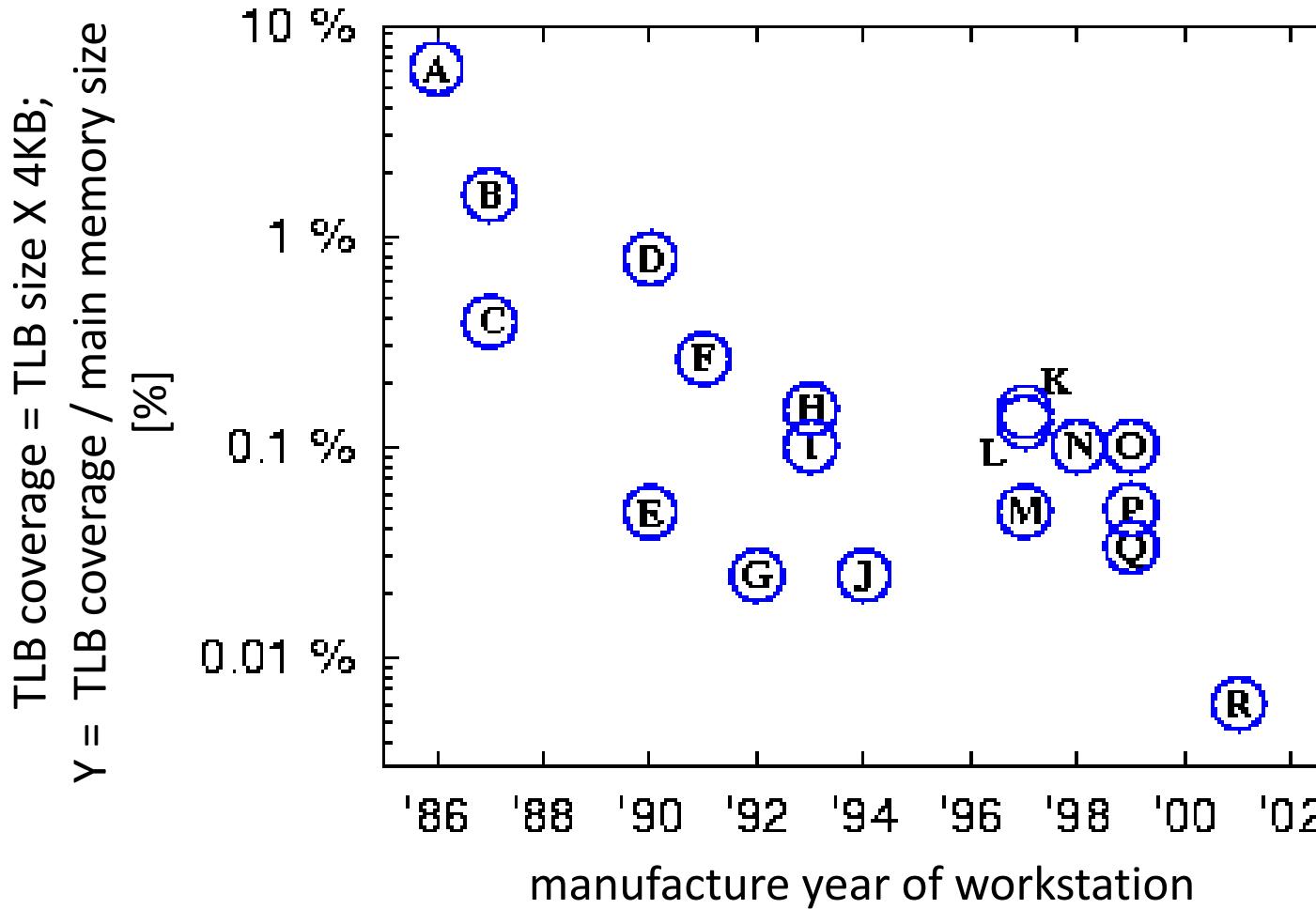
# 64bit x86 address translation

- **64bit address means  $2^{64}$  = 16 Exabyte address space**
  - $2^{20} \approx 10^6$  = Megabyte
  - $2^{30} \approx 10^9$  = Gigabyte
  - $2^{40} \approx 10^{12}$  = Terabyte
  - $2^{50} \approx 10^{15}$  = Petabyte
  - $2^{60} \approx 10^{18}$  = Exabyte (= a billion GBs)
  - DRAM typically can't be that big; 64 bits are currently more than needed
- **x86\_64 HW usually uses “only” 48 bits => 256 TB**
  - Since IceLake (2019), can do 57 bits (5-level hierarchy => 128 PiB)
- **48bit address reflects a 4-level hierarchy, divides into 5 parts**
  - 9bits X 4 levels + 12bits offset
  - Each PTE is 8 bytes (rather than 4, to be able to hold the wider address)
- **The job of the x86 64bit virtual memory subsystem**
  - Translate 36 bits (= virtual page #) to 36 bits (= physical frame #)
  - As before, pages are 4KB-aligned

# 4-level hierarchy



# TLB (L1) coverage drops exponentially



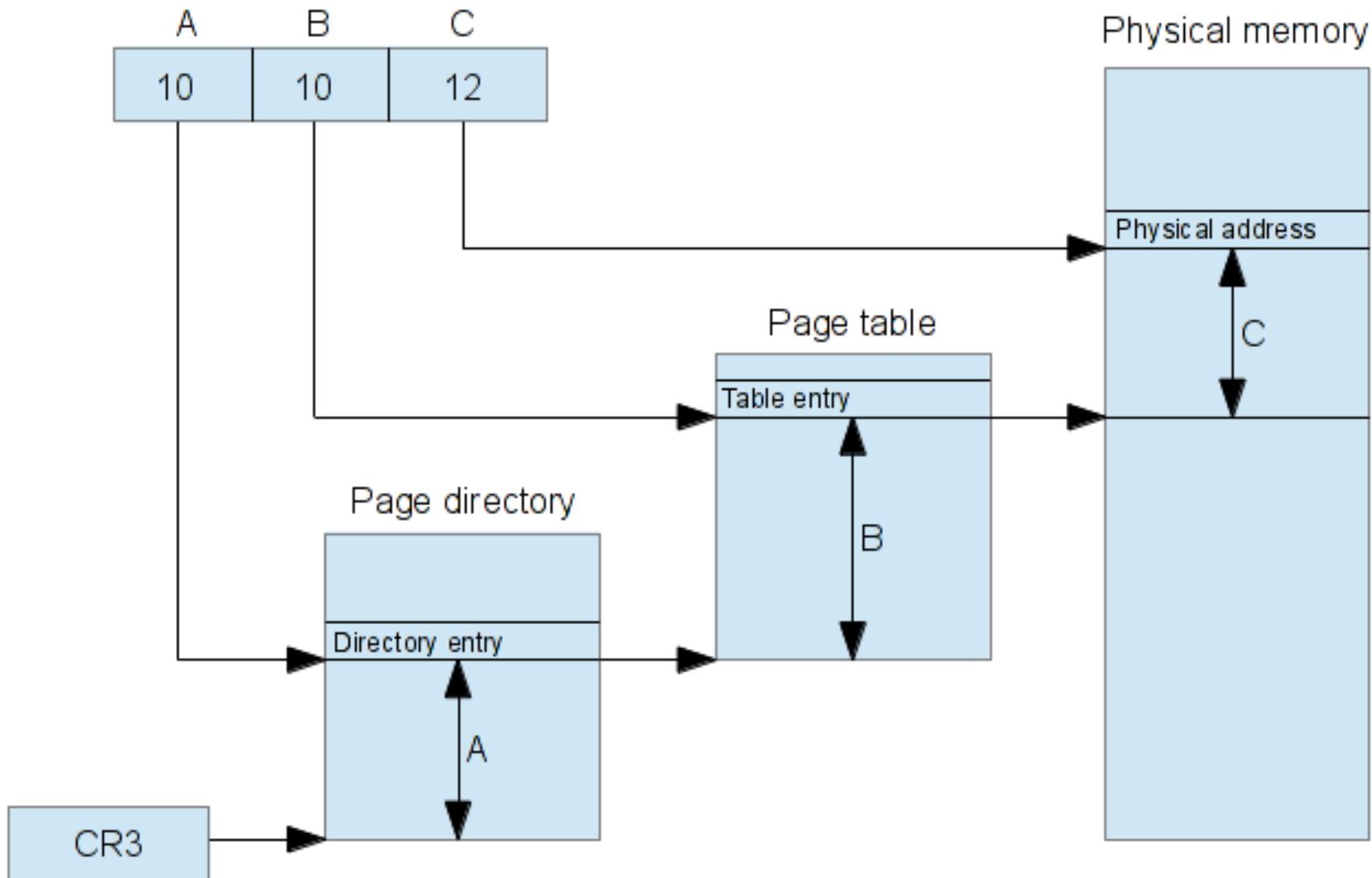
<http://static.usenix.org/event/osdi02/tech/navarro.html>

# How to increase TLB converge?

- **Superpages, a.k.a. **hugepages**, of page size > 4KB**
  - Different sizes supported by different architectures; for Intel/AMD:
    - 32bit architecture\* (“x86”): 4MB (why?)
      - \* With PAE (= page address extension) mode disabled
    - 64bit architecture (“x86-64”): 2MB, 1GB (why?)
  - In Linux, THP (= transparent hugepages) is typically on by default
    - The khugepaged daemon dynamically + opportunistically creates hugepages in the background
    - Homework: think about how khugepaged works
- **TLB hierarchy**
  - Similarly to regular caches: L1, L2, ...,
  - Architectures can also support TLB-L1 (size = a few dozens), TLB-L2 (size = a few hundreds; e.g., 1536 in Intel Skylake, 2048 in Icelake)
  - As usual in a caching hierarchy, TLB-L2 is bigger but slower

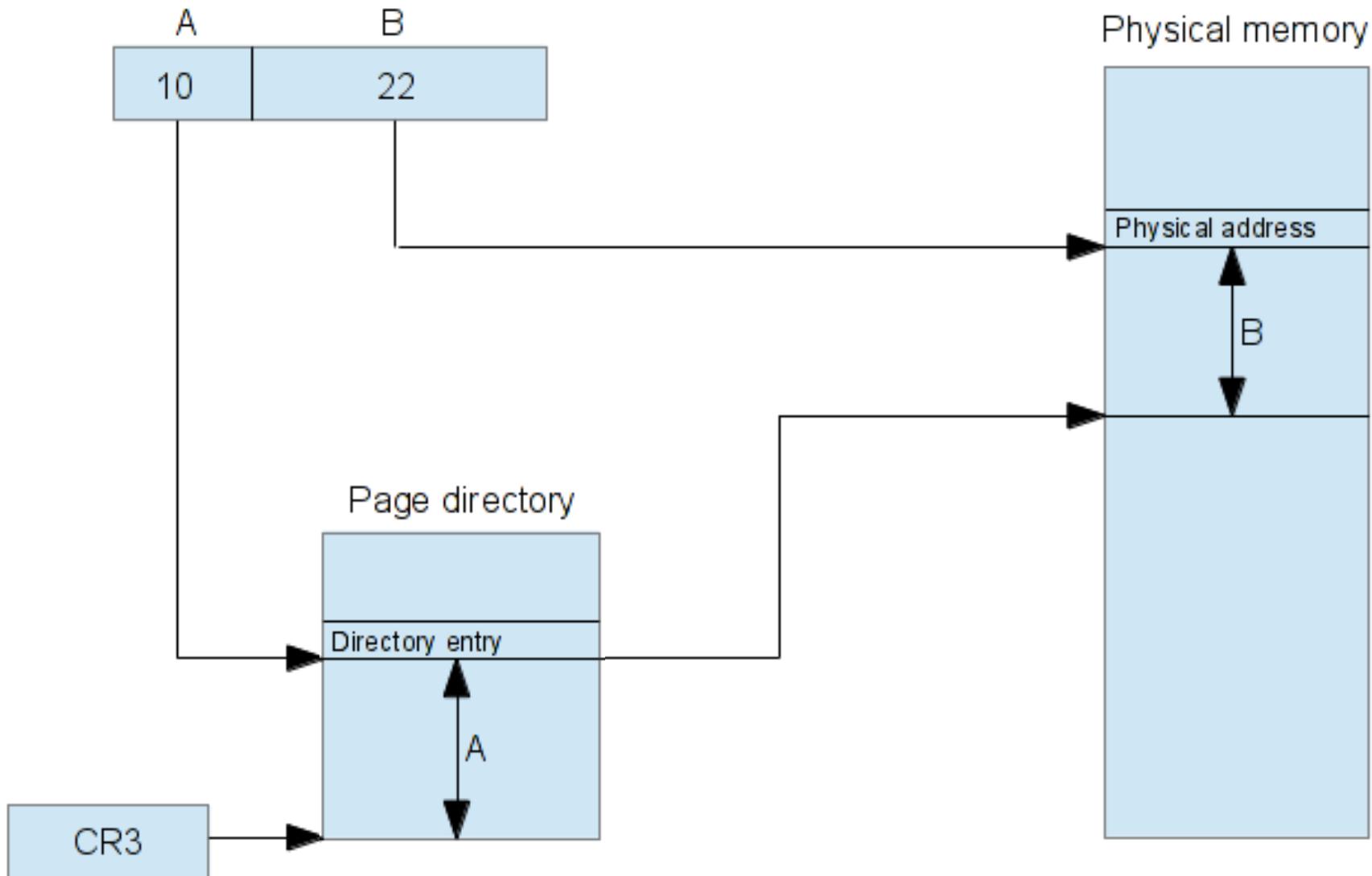
# 4KB pages translation in x86/32bit

32-bits Virtual address



# 4MB pages translation in x86/32bit

32-bits Virtual address



Last time finished here

More details:

<http://www.cs.technion.ac.il/~dan/course/os/ppc64vm-zhang2009.pdf>

<http://www.cs.technion.ac.il/~dan/course/os/ppc64vm-peng1995.pdf>

A different HW/SW virtual memory implementation

# **64-BIT POWER-PC (PPC) VIRTUAL MEMORY**

# HW-support for virtual memory can be....

- **Implemented completely differently**
  - Radix tree design not carved in stone
  - It's the HW vendor that decides
- **Intel/AMD defined it in one way for the x86 architecture**
  - Radix tree page table
- **IBM defined it differently for the POWER architecture**
  - Hashed page table
  - Which we're going to describe next
  - Since POWER9 (2017), also supports radix page tables (for Linux)

J. Jann et al., "IBM POWER9 system software," in IBM Journal of Research and Development, vol. 62, no. 4/5, pp. 6:1-6:10, 1 July-Sept. 2018, <http://doi.org/10.1147/JRD.2018.2846959>

# 3 address types: effective => virtual => real

- **Effective**
  - Each process uses **64-bit** “effective” addresses
  - Effective addresses are **not unique** per-process
  - More or less equivalent to x86 “virtual” addresses
  - Get translated to PPC “virtual” addresses
- **Virtual**
  - A huge **80-bit** address space
  - All processes live in and share this (single) space => **unique** addresses
  - Namely, if two processes have a page with the same virtual address
    - Then it's the same page (= a shared page)
  - *Not* equivalent to x86 virtual addresses
  - Get translated into physical (“real”) address
- **Physical, a.k.a. “real”**
  - **62-bit**

# PPC Segments

- Effective & virtual spaces are partitioned into contiguous segments of 256MB ( $= 2^{28} = 2^{16} \cdot 2^{12} = 64K \cdot 4KB$  pages)
  - Each segment is contiguous in the per-process effective memory space
  - Each segment is contiguous in the single, huge virtual space
  - Segments are 256MB-aligned and can be private or shared (why?)
- How many segments can there be in an effective space?
  - Effective space size is  $2^{64}$ ; so can be  $2^{(64-28)} = 2^{36}$  segments

| $ESID = \text{effective segment ID (36b)}$ | $\text{page number (16b)}$ | $\text{off (12b)}$                     |
|--------------------------------------------|----------------------------|----------------------------------------|
| 63                                         | 28 27                      | 12 11 0<br><i>segment offset (28b)</i> |

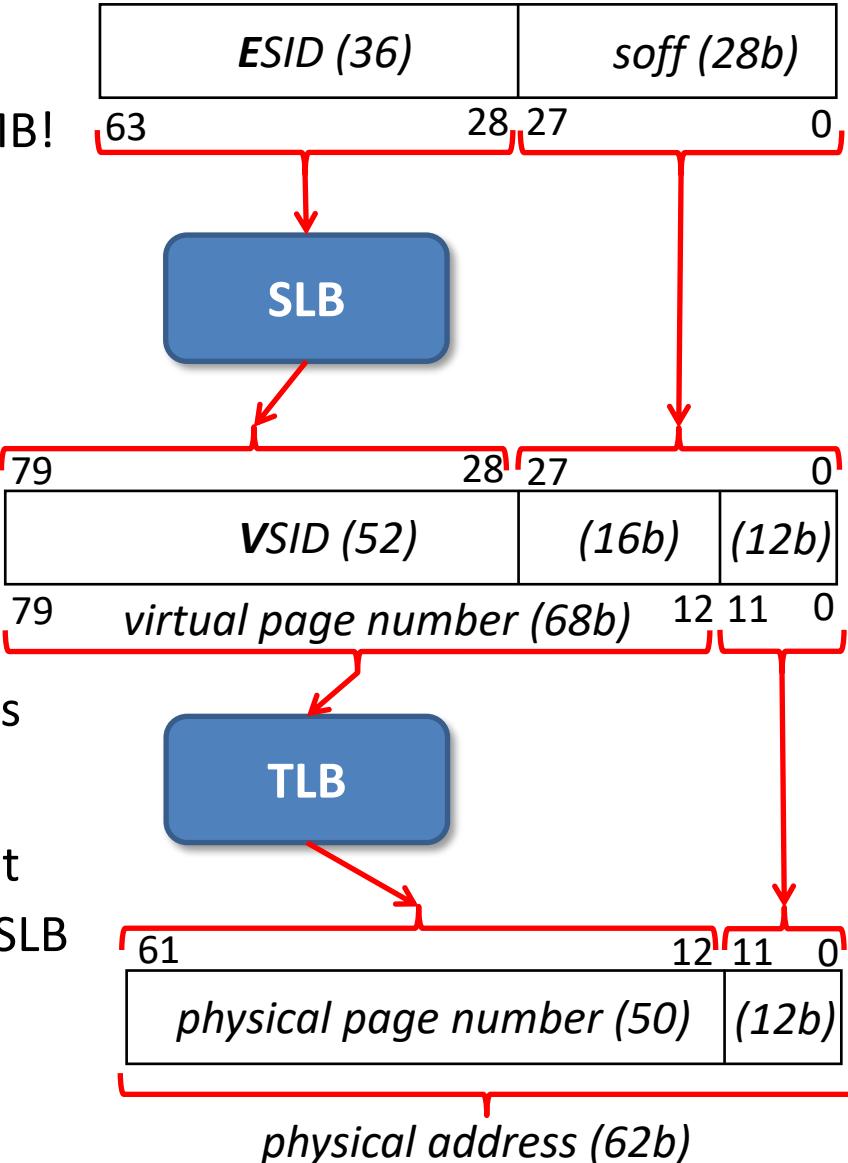
- How many segments can there be in the virtual space?

- Effective space size is  $2^{80}$ ; so can be  $2^{(80-28)} = 2^{52}$  segments

| $VSID = \text{virtual segment ID (52b)}$ | $\text{page number (16b)}$ | $\text{off (12b)}$                     |
|------------------------------------------|----------------------------|----------------------------------------|
| 79                                       | 28 27                      | 12 11 0<br><i>segment offset (28b)</i> |

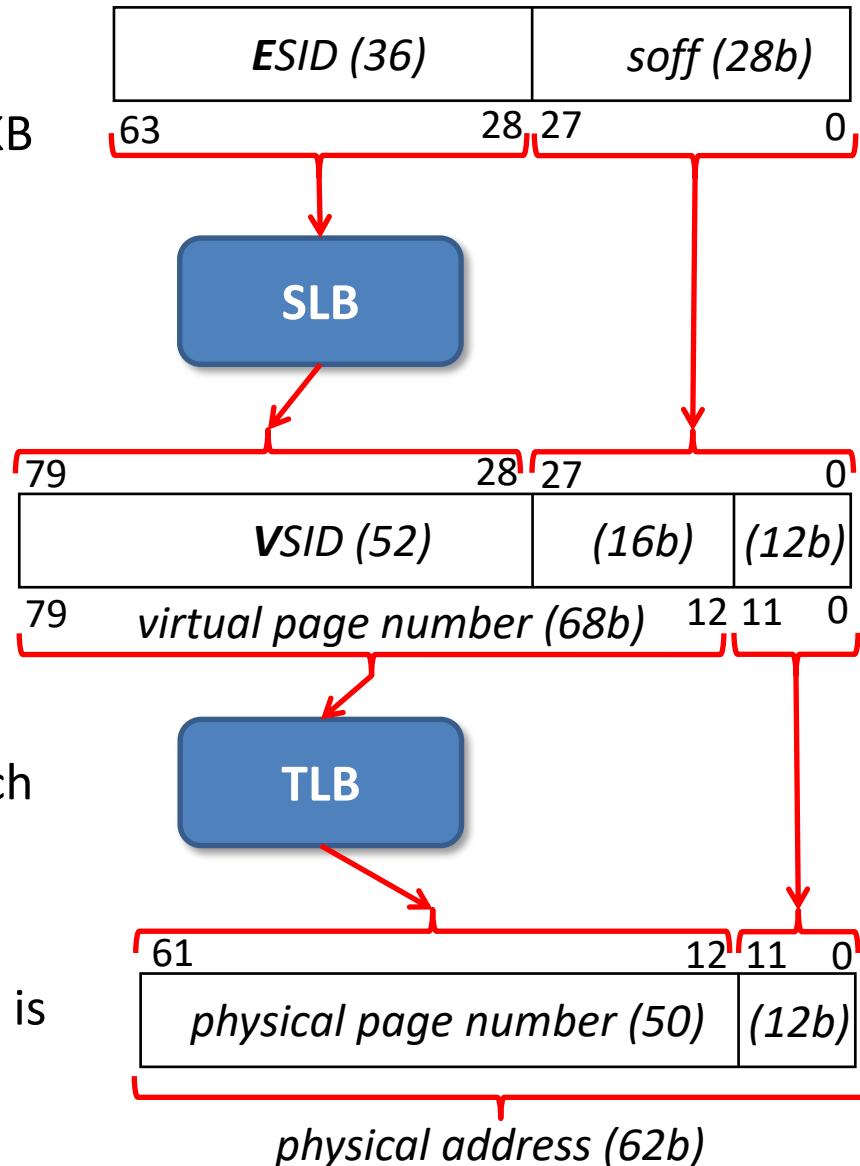
# PPC SLB (segment lookaside buffer)

- **SLB is a fast, small HW cache**
  - O(32) entries, each points to 256MB!
  - O(1) cycles to access
- **HW searches SLB**
  - To find ESID=>VSID mapping
  - Each STE (segment table entry) contains ESID & VSID info
- **OS explicitly manages SLB**
  - Maintains E=>V segment table
  - Includes all segments of the process
- **Upon SLB miss**
  - HW raises “segment fault” interrupt
  - OS will then insert the right STE to SLB
- **Upon context switch**
  - OS invalidates SLB (not shared)



# PPC TLB (translation lookaside buffer)

- **TLB is a (less) fast HW cache**
  - O(1024) entries, each points to 4KB
  - O(10) cycles to access
- **HW searches TLB**
  - To find VPN=>PPN mapping
  - Each PTE (page table entry) contains VPN & PPN info
- **Shared by all processes**
  - Since virtual space is shared
  - Unlike in x86
  - Not invalidated upon context switch
- **Managed by HW & OS**
  - Upon miss, HW populates TLB
  - By “walking the page table”, which is in fact the “HTAB” ...



# PPC HTAB – a hash table

- **PPC PTE (page table entry)**
  - Contains PPN & VPN (the “tag”; why do we need it?)
    - Unlike x86 PTE that only contains PPN
  - $|PTE| = 16 \text{ bytes} = 128 \text{ bit} > |VPN (68 bit)| + |PPN (50 bit)| = 118 \text{ bit}$
- **PTEG (page table entry group)**
  - Contains 8 PTEs
  - $8 (\text{PTEs}) * 16 \text{ bytes} = 128 \text{ bytes} = |\text{PPC cache line}|$
  - Each VPN=>PPN translation may reside in “primary” or “secondary” PTEGs
- **HTAB**
  - At boot time, OS allocates in DRAM the “HTAB” array
    - HTAB size is configurable (we’ll discuss which size to use later)
    - OS saves HTAB’s size & base in “SDR1” (storage description register)
  - Primary & secondary hash functions defined in processor spec
    - $\text{Hash}_i(\text{VPN}) = \text{PTEG number } (i=1,2)$

# PPC HTAB – a hash table

- **Upon TLB miss**
  - Foreach hash  $h$  in <primary, secondary>
    - HW computes  $h(\text{VPN})$  modulo  $k$  (= num of PTEGs in HTAB)
    - HW searches VPN in 8 PTEs populating the corresponding PTEG
    - If found, HW puts  $\text{VPN} \Rightarrow \text{PPN}$  in TLB and re-executes operation
  - If not found, HW triggers page-fault interrupt
    - OS will then resolve the page fault, and
    - Place the appropriate PTE in one of the two associated PTEGs
    - After interrupt is handled, HW will re-execute the above ‘foreach’
      - Now it’ll find the VPN in one of the 2 PTEGs
- Parallelism
  - Primary & secondary can be searched in parallel
  - All 8 PTEs in PTEG can be compared in parallel

# PPC HTAB – a hash table

- **HW searches for VPN=>PPN translation as follows**
  - if( VPN found in TLB )
    - Get PPN from TLB
  - else if( VPN found in primary\_PTEG = HTAB[ hash1(VPN) % k ] )
    - Get PPN from primary\_PTEG
  - else if( VPN found in secondary\_PTEG = HTAB[ hash2(VPN) % k ] )
    - Get PPN from secondary\_PTEG
  - else
    - Generate page-fault interrupt
    - OS will place appropriate PTE in either primary or secondary PTEGs
    - Next time, the faulting operation will succeed to load the mapping from the HTAB

# PPC ERAT (effective to real translation)

- A small, fast HW cache
  - $O(128)$  entries
  - $O(< 1\text{-cycle})$  to access (on the critical path)
  - Translates from effective to real (physical)
  - Analogous to x86 TLB (L1)
  - Updated to hold the most recent effective $\Rightarrow$ physical mappings used
    - On an LRU basis
  - If hit, don't need to go through the SLB/TLB process
    - Typically, obviates the need to do costly SLB $\Rightarrow$ TLB $\Rightarrow$ HTAB translations

# Aftermath

- In fact, PPC uses TLB hierarchy
  - ERAT
    - Is “TLB” L1
  - TLB
    - Is TLB L2
  - HTAB
    - Is “TLB” L3
  - Only if not found in all 3 levels
    - Go to OS, which has all information
  - Similarly, Intel & AMD
    - Introduced a (bigger, slower) TLB 2 (> 1000 entries)
  - The reason...
    - DRAM is getting bigger
    - TLB L1 remains roughly the same size

rpt

rpt

# End

# Aftermath

- Why three spaces (effective, virtual, real)?  
Why one HTAB for all processes (rather than per-process)?
  1. Limits hash table (HTAB) load factor => no need to dynamically resize!
    - Lots of memory pages are shared (threads, page cache, CoW, ...)
    - But unlike x86, sharing doesn't imply more <key, data> hash pairs
      - Because 'virtual' shared addresses are identical across processes
    - Therefore, if  $| \text{physical mem} | = K \text{ pages}$   
and we'd like  $| \text{load factor} | = f$  (e.g.,  $f=0.5$ )  
then allocate  $| \text{HTAB} | = K/f \text{ entries}$
    - Example: if we'd like  $f=\frac{1}{2}$  for  $| \text{DRAM} | = 128\text{GB}$ , then  $| \text{HTAB} |$  should be
      - $16\text{B} (=| \text{PTE} |) * (128\text{GB} / 4\text{KB}) / f = 1\text{GB} < 0.8\% \text{ of phys mem}$
  2. Efficiently support different page sizes
    - Different 256MB segments can be associated with different page sizes
  3. Technically, allows a single HTAB to service all processes
    - Effective – but not virtual – addresses of different processes reused

Material we assume you learned in (the new version of) ATAM, which

- (1) is considered part of this course, and which
- (2) we're skipping due to the above assumption

# Operating Systems (234123)

## *Interrupts*

Dan Tsafrir (2025-06-16)

Partially based on Section 1.1 – 1.3 in Feitelson’s “OS notes”

# **Reminder: the OS is a reactive “program”**

- **Usually, the OS spends most of its time**
  - Waiting for events to occur (e.g., a keyboard key press)
- **Whenever an event happens**
  - OS handles quickly & gives CPU back to running process (if such exists)
- **The goal of the OS**
  - Run as little as possible, handle events as quickly as possible, and let apps run most of the time
- **OS pseudo code**
  - (Doesn't look like the real thing, but does capture the essence)

```
while( 1 ) {  
    event_t e = wait_for_event(); /*sleep*/  
    handle_event(e);  
    resume_process() }
```
- **But what are these events really?**
  - Interrupts! That is it. There's \*nothing\* else

# Terminology note

- Our “interrupts” are Intel’s “interrupts and exceptions” such that
  - Intel’s “exceptions”  $\Leftrightarrow$  our “internal (synchronous) implicit interrupts”
  - Intel’s “interrupts”  $\Leftrightarrow$  all the rest

# Who generates interrupts?

- **Two sources (two interrupt “types”)**
  - External / asynchronous
    - Triggered by devices external to the core’s compute circuitry
    - “Asynchronous” because they can occur at any time
    - (“between CPU operations”)
  - Internal / synchronous
    - Triggered by the core’s compute circuitry (CPU instructions)
      - Well, actually, it's the software that runs on the core...
      - Which is why such interrupts are also sometimes called
        - » “Software interrupts”
    - (“while a CPU operation occurs”)

# External (asynchronous) interrupts

| external devices | when they want to say, for example,                                      | comment                                                                                                |
|------------------|--------------------------------------------------------------------------|--------------------------------------------------------------------------------------------------------|
| NIC              | packet(s) arrived!<br>packet(s) sent! ( <b>DMA</b> )                     | network interface controller (or network interface card)                                               |
| disk controller  | finished reading a block!<br>finished writing a block!<br>( <b>DMA</b> ) | HDD, SDD, floppy, DVD, ...<br>(called “block device”, as opposed to byte-addressable)                  |
| periodic clock   | another tick has elapsed!                                                |                                                                                                        |
| one-shot timer   | timer expired!                                                           | clock                                                                                                  |
| another core     | I want you to do something!                                              | IPI = inter-processor interrupt<br>(recall that we've seen it used in the Linux 2.4 scheduler lecture) |
| keyboard         | key was pressed!                                                         |                                                                                                        |
| mouse            | I was moved/clicked!                                                     |                                                                                                        |
| microphone       | I heard something!                                                       |                                                                                                        |
| many more...     |                                                                          |                                                                                                        |

# Internal (synchronous) interrupts – explicit

- The software that runs on the core can generate interrupts
- Some interrupts are deliberate = explicitly invoked by SW
  - Execute the ‘int’ or ‘syscall’ operation, which trap into the OS
    - System calls to request some service from OS
      - Same interrupt for all system calls (why?)
    - Debugging (software-)breakpoints
  - Ptrace-ing
    - Allows userland to implement single-step debugging, see
      - <http://en.wikipedia.org/wiki/Ptrace> and
      - <http://linux.die.net/man/2/ptrace>

# Internal (synchronous) interrupts – implicit

- Some interrupts occur implicitly, indicating an exception condition
  - The processor can't complete the current instruction for some reason
  - So it transfers responsibility to the OS
- There are 2 types of exceptions:
  - An error condition tells the OS that the app did something illegal
    - Divide by 0, execute privileged instruction, ...
    - By default, OS kills such misbehaving processes
      - By sending a signal & setting default signal-handler that kills the process (discussed later)
  - A temporary problem
    - Memory currently unmapped (next lecture)
    - This is the OS's fault; it can & should handle it
  - Who decides if it's temporary?
    - OS

# Interrupt handling (x86/Linux)

- **Each interrupt is associated with a**
  - Number (called “interrupt vector”)
    - There are 256 interrupt vectors, numbered 0...255
  - Handling routine (called “interrupt handler”)
    - Need different routine because, for example, keyboard key press is handled differently than incoming network packet
    - Can associate several devices with same interrupt vector when some additional info is made available to OS to identify source device
- **Handlers pointed to by array[256] of pointer-to-functions**
  - Array[j] holds pointer to the interrupt handler of vector j
  - When an interrupt fires, HW invokes the corresponding function
  - Array content is initially set by BIOS
    - (For example, to be able to use keyboard before OS is up)
  - Later, at boot time, array is reinitialized by OS with its own routines

# Interrupt handling (x86/Linux)

- **The interrupt handler**
  - Save state of currently running process P
  - Execute handler
  - Return to user mode, resuming either P or another process
    - Q: When will it be another process? When the same?
- **Context type**
  - If **asynchronous** interrupts, kernel is said to be in “interrupt context”
    - It doesn’t directly serve the process that has just been stopped
  - If **synchronous**, kernel is in “process context”
    - It needs to provide some service to the process that invoked it (whether explicitly or implicitly)

# Interrupt handlers are not schedulable entities

- **When an interrupt fires**
  - Its handler runs immediately (scheduler not involved!), that is
  - Unless the interrupt is currently “masked”
  - In which case handler will run when the interrupt is unmasked
- **Interrupt masking**
  - An interrupt handlers constitutes a different (OS) thread of execution
  - So need to worry about synchronization (races, mutex, deadlocks, ...)
  - Thus, HW supports mask[256] bitmap, and it blocks those with set bits
- **What happens to masked interrupts?**
  - If interrupt vector K should fire but is masked
  - x86 HW remembers at most 2 instances of K (the rest get lost, if exist)
- **Since handlers run immediately when their interrupts fire**
  - The OS scheduler doesn’t control them even if what’s currently running is “more important”, so...

Skipping over material you've already learned in  
ATAM ends here

Remaining interrupt material

# **RESUME**

# Interrupt “top half” & “bottom half”

- **Handling an interrupt might take a long time**
  - But we want to allow OS to decide if it has more important stuff to do
  - Also, x86 HW waits for OS to say “EOI” (end of interrupt) before it allows other, equal or lower priority interrupts to fire
- **Solution: split the handler into two**
  - Fast/Hard/First-Level Interrupt handler (FLIH)
    - What we *must* do when receiving an interrupt, and nothing else
    - Not interruptible => as small and as quick as possible
  - Slow/Soft/Second-Level interrupt handler(SLIH)
    - All the rest (lower priority)
    - Interruptible; will be executed later by some kernel thread
- **In Linux**
  - FLIH = “top-halve”
  - SLIH = “bottom-halve”

# Example

- **When receiving a packet from the network**
  - Possible top half  
(not schedulable, occurs immediately)
    - Put a pointer to the received packet in a queue for later processing
    - Give the NIC a new free memory buffer, such that it will have room to save subsequent incoming packets
  - Possible bottom half  
(will happen later when the scheduler decides)
    - Hand packet to network TCP/IP stack
    - Copy the content of the packet to the memory space of the corresponding user application
    - Notify the user app that a new packet has arrived, e.g., by returning from a read() syscall

# Resume an interrupted process - where?

- **If asynchronous (external) interrupt // e.g., timer**
  - Resume in the next process instruction,
  - Right after the last instruction that was executed
- **If synchronous(=internal) + explicit // e.g., syscall, ptrace**
  - Likewise, resume in the next instruction
  - Right after the last instruction that triggered the trap
- **If synchronous(=internal) + implicit // =exception, e.g., page fault**
  - Resume in the same instruction that the HW failed to execute
    - Called “restartable instruction”
  - This makes sense for temporal problems (like unmapped memory)
    - Thus, OS doesn’t send signal to app
  - It makes less sense for error conditions (divide by 0 etc.)
    - Thus, signal is sent to app (default signal handler kills app)
    - If app changed the default & doesn’t fix things => endless loop

# Interrupt handling in multicores – where?

- **Internal=synchronous interrupts (system call, divide by 0, ...)**
  - Each core handles on its own
- **External=async interrupts (generated by external devices)**
  - Can do static partition of interrupts between cores
    - E.g., all interrupts go to core 0
    - Or, all interrupts in range x–y go to core j
  - Can do dynamic partition of interrupts between cores
    - E.g., round robin
    - Or, send interrupt to least loaded core
  - Network typically hash 5-tuples to determine destination core (RSS)
    - To preserve packet order
- **Homework**
  - In a Linux shell do
    - `watch -n1 cat /proc/interrupts`

# Polling

- **With I/O-intensive devices (like 100Gb/s NICs)**
  - OS might be overwhelmed by very many interrupts, telling it what it already knows: that there are lots of incoming packets to handle
  - This is called an “interrupt storm”
- **Handling that many interrupts consumes lots of cycles**
  - Cycles that could be used more productively for actually receiving & processing the packets
- **Polling is an alternative to interrupt-based processing**
  - OS turns off interrupts and instead polls the NIC every once in a while
  - Effective when OS is mostly busy handling the packets
- **Cons**
  - Polling too frequently might waste CPU cycles if not enough packets
  - Polling too infrequently increases latency & may result in packet loss
- **Compromise**
  - Interrupt coalescing (מיזג) (next slide)

# Interrupt coalescing

- **By HW**
  - Devices are mindful not to create “interrupt storm”, so
  - By default, they typically coalesce several interrupts into one
  - For example, NICs typically let a few microseconds to elapse before emitting another interrupt
  - (Sometimes the coalescing interval is configurable by SW)
- **By SW**
  - Example: Linux’s NAPI (“new API” :-/)
  - Linux kernel uses the interrupt-driven mode by default
  - Temporally switches to polling mode when the flow of incoming packets exceeds a certain threshold, in process (schedulable) context
  - See [http://en.wikipedia.org/wiki/New\\_API](http://en.wikipedia.org/wiki/New_API) for more details

# DMA & interrupts

- **Recall: DMA = direct memory access**
  - When an I/O device accesses memory directly, without CPU involvement
  - OS requests I/O devices to transfer data
  - Devices do it on their own, with DMA operations (read or write)
- **Conceptually, when disregarding coalescing, devices fire an interrupt when a DMA operation is finished**
  - Not only when receiving (reading) data
  - But also, when sending (writing) data
  - Because need to notify OS when the send buffer becomes free

# Example – blocking read from disk

- **process**
  - => read()
  - => OS
  - => process is suspended (give CPU to others)
  - => find blocks from which data should be read
  - => initiate DMA operation
  - => DMA done
  - => device fires interrupt to notify DMA is done
  - => OS top-half adds relevant bottom half to relevant list
  - => later, bottom half scheduled to run & copies data to user
  - => makes process runnable
  - => process scheduled and can now read the data

# **Operating Systems (234123)**

## ***Storage***

Dan Tsafrir (2025-06-16, 2025-06-23)

Partially based on: Chapter #5 in OS notes; Silberschatz; slides by Hagit Attiya, Idit Keidar, Kaustubh Joshi, Michael Swift; VSFS stuff taken from “Operating Systems: Three Easy Pieces” by Remzi & Andrea Arpaci-Dusseau (<http://pages.cs.wisc.edu/~remzi/OSTEP/>)

# Why disk drives? (Memory not enough?)

- **Disk drive = secondary storage**

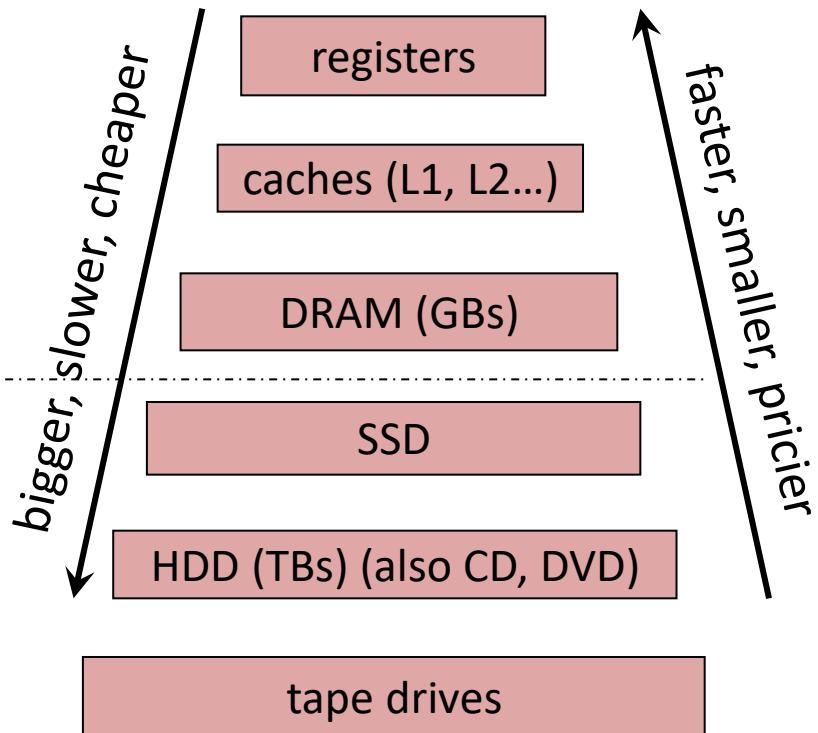
- Persistent (non-volatile)
- Bigger
- Cheaper (per byte)

- **Recall that**

- CPU (or rather, its MMU = memory management unit), can't access disk drives directly. It's all done by the OS

volatile  
persistent

can't access disk drives directly. It's all done by the OS



- **We'll typically think of HDDs**

- Spinning hard disk drives
- Still being used in data centers (albeit rapidly replaced by SSDs)
- Spinning, moving head negatively affects random access

# Goals: when using disks, we want...

- **Persistence**
  - Outlive lifetime of a process, tolerate power outage
- **Support ownership**
  - Users, groups; who can access? (read/write)
- **Robustness**
  - In the face of failures
- **Performance**
  - As performant as possible
- **Concurrency support**
  - What happens if we access the information concurrently
- **Storage drive abstraction**
  - Access similar to HDD, SSD, DVD, CD, tape = all are “block devices”
    - Block device = IO is done in block (=several bytes) resolution

# Achieve our goals through *filesystems*

- **Provides abstractions**
  - To help us organize our information conveniently
  - Such that we could easily find & access our data
- **Main abstractions**
  - File
  - “Directory” (in UNIX) = “folder” (in Windows)
  - “Soft link” (in UNIX) = “shortcut” (in Windows)
  - Hard link (same term used in Windows)
  - Standardized by POSIX
- **We will discuss**
  - The filesystem API
  - The abstractions
  - And their implementation

# **FILES AND DIRECTORIES**

# File

- **A logical unit of information**
  - ADT (abstract data type)
  - Has a name
  - Has content
    - Typically, a sequence of 0 or more bytes  
(we'll exclusively focus on that)
    - But could be, e.g., a stream of records (database)
  - Has metadata/attributes (creation date, size, ...)
  - Can apply operations to it (read, rename, ...)
- **Persistent (non-volatile)**
  - Survives power outage, outlives processes
  - Process can use file, die, then another process can use file

# POSIX file concurrency semantics

- **Reading concurrently from a file**
    - Isn't a problem
    - For example, if a file is an executable (a program), it can serve as the text of multiple processes simultaneously
  - **Writing concurrently to a file**
    - Local filesystem ensures **sequential consistency**
      - *“...the result of any execution is the same as if the operations of all the processors were executed in some sequential order, and the operations of each individual processor appear in this sequence in the order specified by its program.”*
- Leslie Lamport, 1979

|                       |                                                                                                                                                                 |
|-----------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>Leslie Lamport</b> |                                                                                                                                                                 |
| <b>Born</b>           | Leslie B. Lamport<br>February 7, 1941 (age 78)<br>New York City, New York                                                                                       |
| <b>Alma mater</b>     | Massachusetts Institute of Technology (BSc)<br>Brandeis University (PhD)                                                                                        |
| <b>Known for</b>      | LaTeX<br>Sequential consistency<br>Atomic Register Hierarchy<br>Lamport's bakery algorithm<br>Byzantine fault tolerance<br>Paxos algorithm<br>Lamport signature |
| <b>Awards</b>         | Dijkstra Prize (2000, 2005, and 2014)<br>IEEE Emanuel R. Piore Award (2004)<br>IEEE John von Neumann Medal (2008)<br>Turing Award (2013)                        |

# File metadata (a.k.a. attributes)

- Examples
  - Size
  - Owner
  - Permissions
    - Readable? Writable? Executable?
    - Who can read? Who can write? Who can execute?
  - Timestamps, e.g.,
    - Creation time
    - Last time **content** was modified
    - Last time **metadata** was modified
  - Physical location on disk
    - Recall that a disk is a “block device”
    - Where do the file’s blocks reside on disk?
  - Type (means various things in various filesystems)
    - E.g., regular file vs. directory

# POSIX file descriptors (FDs)

- A successful `open`<“file name”> of a file returns a FD
  - A nonnegative integer
  - An index to a per-process array called the “file descriptor table”
  - Each entry in the array saves, e.g., the current `offset`
  - Threads share the array (and hence the offset)
  - C’s `FILE` structure encapsulates a FD

rpt

- FD or filename?
  - Some file-related POSIX system `calls operate on FDs`
    - `read`, `write`, `fchmod`, `fchown`, `fchdir`, `fstat`, `ftruncate`...
  - Others `operate on file names`
    - `chmod`, `chown`, `chdir`, `stat`, `truncate`
  - Has security implications: FD versions are more secure in some sense
    - Because association of FD to underlying file is immutable
      - Once an FD exists, it will *always* point to the same file
    - Whereas association between file & its name is mutable
      - So they can lead to TOCTTOU (time of check to time of use) races

# Canonical POSIX file operations

- **Creation**    (**syscalls**: `creat`, `open`;                    **C**: `fopen`)
    - Associate with a name; allocate physical space (at least for metadata)
  - **Open**            (**open**;                                            **C**: `fopen`, `fdopen`)
    - Load required metadata to allow process to access file
  - **Deletion**    (**unlink**, **rmdir**;                            **C**: `remove`)
    - Remove name/file association & (possibly) release physical content
  - **Close**            (**close**;                                    **C**: `fclose`)
    - Mark end of access; release associated process resources
  - **Rename**        (**rename**;                                    -)
    - Change associated name
  - **Stat**            (**stat**, **Istat**, **fstat**;                    -)
    - Get the file's metadata (timestamps, owner, etc.)
  - **Chmod**        (**chmod**, **fchmod**;                            -)
    - Change readable, writable, executable properties

# Canonical POSIX file operations

# Canonical POSIX file operations

- **Sync**      (`sync<fs>, fsync<fd>;`)      <> C: `fflush`)
  - Recall that
    - All disk I/O goes through OS “page cache”, which caches the disk
    - OS sync-s dirty pages to disk periodically (every few seconds)
  - Use this operation if we want the sync now
  - ‘sync’ is for all the filesystem, and ‘fsync’ is just for a given FD
  - Sync <> fflush; the latter flushes user-space (`fprintf`, `cout`) buffers to the kernel
- **Lock**      (`flock, fcntl;`)      C: `flockfile`)
  - “Advisory” lock (= processes can ignore it, if they wish)
  - There exists mandatory locking support  
(in Linux and other OSes)
    - E.g., every open implicitly locks; and can’t open more than once  
<https://www.kernel.org/doc/Documentation/filesystems/mandatory-locking.txt>
    - But that’s not POSIX

# Reminder: man sections (in Linux)

- To see exact semantics of file (and other) ops, use ‘man’ command
  - Synopsis: man [options] [section] <name>
- Section 1: user (shell) commands
  - man sync                  man 1 sync                  man 1 read                  man 1 printf
- Section 2: system calls
  - man fsync                  man 2 fsync                  man 2 read                  man 2 write
- Section 3: C library functions
  - man fflush                  man 3 fflush                  man fread                  man 3 printf
- Section 4: devices & special files
  - man 4 null (for /dev/null)                  man 4 zero (for /dev/zero)
- Section 5: file formats
  - man fstab                  man 5 fstab                  man crontab                  man 5 crontab
- Section 6: games et al.
  - (man intro = intro to Linux, ...)
- Section 7: miscellanea
  - (pthreads, ascii, ...)
- Section 8: sys admin & daemons
  - (httpd, hdparm, fsck, ...)

# File types

- **Some systems (not Unix/POSIX) distinguished between**
  - Text & binary
- **Some older systems decided type by name extensions**
  - In DOS, executables have the extensions: com, exe, bat
- **In UNIX (POSIX), types are**
  - Regular file
  - Directory
  - Symbolic link (= shortcut), a.k.a. soft link
  - FIFO (named pipe)
  - Socket
  - Device file
  - See: [http://en.wikipedia.org/wiki/Unix\\_file\\_types](http://en.wikipedia.org/wiki/Unix_file_types)

# Magic numbers in the UNIX family

- A semi-standard Unix way to tell the type of a file
  - Store a "magic number" inside the file itself
  - Originally, first two 2-byte => only  $2^{16}$  => not enough
  - Nowadays, much a more complex scheme
- Examples
  - Every GIF file starts with the ASCII strings: *GIF87a* or *GIF89a*
  - Every PDF file starts with the ASCII string: *%PDF*
  - Script files start with a “shebang” (`#!`), followed by an executable name, which identifies the interpreter of the script; the shell executes the interpreter and feeds the script to it as input (“#” indicates comment for the interpreter, so it ignores this line)
    - `#!/usr/bin/perl -w`  
# Perl code here...
    - `#!/usr/bin/py`  
# Python code here...

# Magic numbers in the UNIX family

- **Pros**
  - File's content, rather than metadata, determines what this file is
  - (Metadata like the file's name might be altered independently of the content, potentially erroneously)
- **Cons**
  - Magic logic became fairly complex
  - Somewhat inefficient because
    - Need to check against entire magic database, and
    - Need to read file content rather than just metadata
- **More details**
  - [http://en.wikipedia.org/wiki/File\\_format#Magic\\_number](http://en.wikipedia.org/wiki/File_format#Magic_number)
- **Helpful – the ‘file’ utility**
  - A shell utility that, given a file, identifies its type
  - <http://linux.die.net/man/1/file>

# POSIX file protection: ownership & mode

- **Motivation**
  - In a multiuser system, not everyone is allowed to access a given file
  - Even if they're allowed to “access”, we don't necessarily want to allow them to perform every conceivable operation on that file
- **For each file, POSIX divides users into 3 classes**
  - “User” (the owner of the file), “group”, and “all” the rest
  - (Users can belong to several groups; see: man 2 getgroups)
- **POSIX associates 3 capabilities with each class**
  - Read, write, and execute
- **Hence, each file is associated with**
  - $3 \times 3 = 9$  class/capabilities => this is called the “file mode”
  - Mode is controlled by the (f)chmod syscall
  - Ownership (user & group) is controlled by the (f)chown syscall

# POSIX file protection: ownership & mode

- Stat returns these 9, and ‘ls -l’ displays them
  - 10 “bits” are displayed
  - Leftmost is the “type”
  - Remaining 9 bits are read/write/execute X user/group/all

|            |   |         |       |         |        |       |               |
|------------|---|---------|-------|---------|--------|-------|---------------|
| brw-r--r-- | 1 | unixguy | staff | 64, 64  | Jan 27 | 05:52 | block         |
| crw-r--r-- | 1 | unixguy | staff | 64, 255 | Jan 26 | 13:57 | character     |
| -rw-r--r-- | 1 | unixguy | staff | 290     | Jan 26 | 14:08 | compressed.gz |
| -rw-r--r-- | 1 | unixguy | staff | 331836  | Jan 26 | 14:06 | data.ppm      |
| drwxrwx--x | 2 | unixguy | staff | 48      | Jan 26 | 11:28 | directory     |
| -rwxrwx--x | 1 | unixguy | staff | 29      | Jan 26 | 14:03 | executable    |
| prw-r--r-- | 1 | unixguy | staff | 0       | Jan 26 | 11:50 | fifo          |
| lrwxrwxrwx | 1 | unixguy | staff | 3       | Jan 26 | 11:44 | link -> dir   |
| -rw-rw---- | 1 | unixguy | staff | 217     | Jan 26 | 14:08 | regularfile   |

# Access control lists (ACLs)

- **OSes can support a much finer, more detailed protection**
  - Who can do what
- **Most OSes/filesystems support some form of ACLs**
  - Many groups/users can be associated with a file
  - Each group/user can be associated with the 3 attributes (r/w/x)  
<http://static.usenix.org/events/usenix03/tech/freenix03/gruenbacher.html>
  - Or more, finer attributes (“can delete”, “can rename”, etc.)
- **Con: not part of POSIX**
  - Effort to standardize ACLs abounded in Jan 1998
    - Participating parties couldn’t reach an agreement...
  - Hence, it’s hard to make programs that use ACLs portable
    - (Recall: a program is “portable” across a set of OSes if it works on all of them without having to change its source code; in particular, a program that adheres to POSIX works unchanged in all OSes that comply with POSIX = the UNIX family: Linux, AIX, Solaris, FreeBSD, Mac OS, ...)

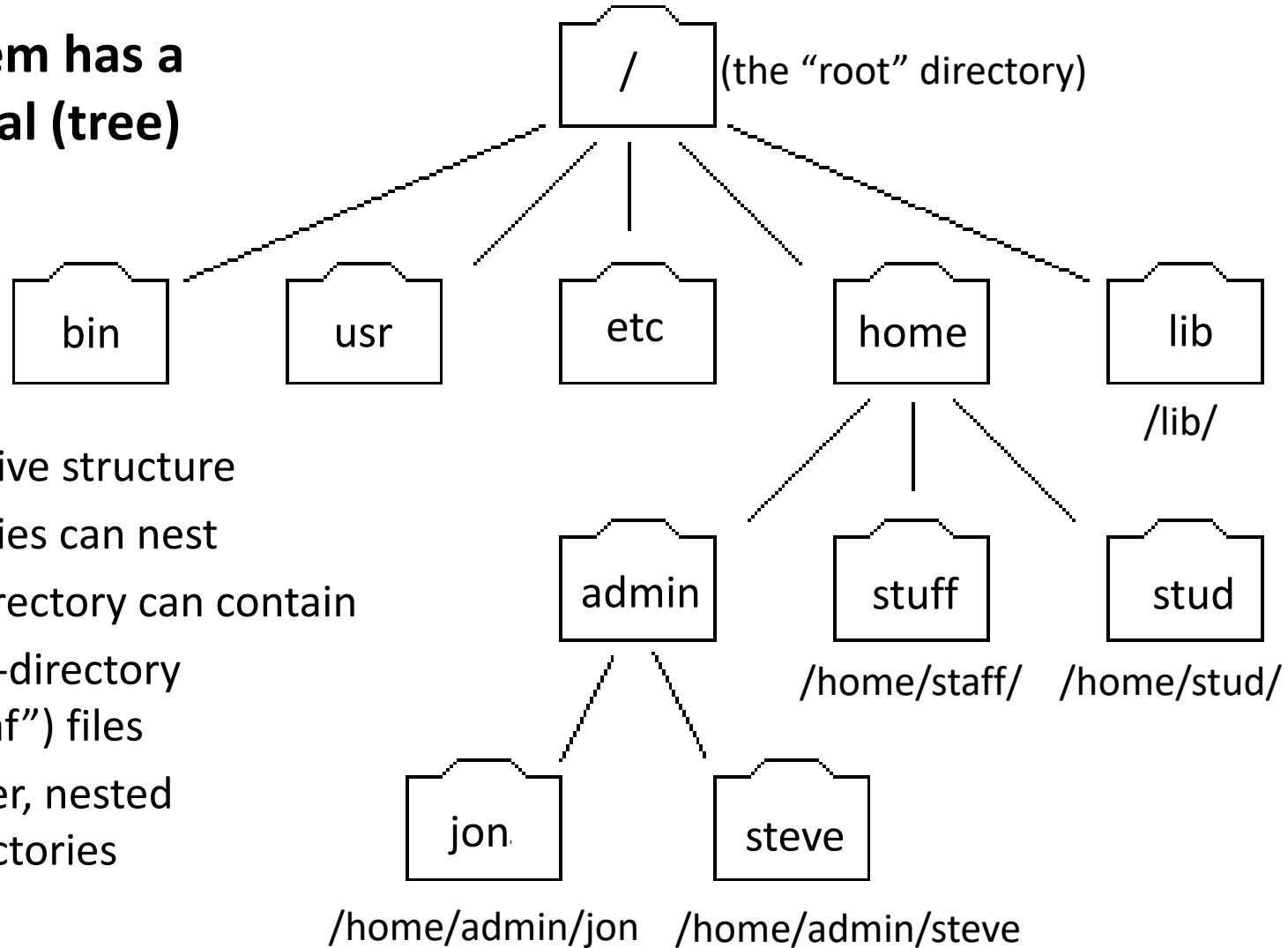
# Filesystem building blocks

- **Abstractions**
  - Directories,
  - Hard links, and
  - Symbolic links
- **Implementation**
  - Inodes, and
  - Dirents

# Directories

- A filesystem has a hierarchical (tree) structure

- A recursive structure
- Directories can nest
- Every directory can contain
  - Non-directory (“leaf”) files
  - Other, nested directories



# Absolute & relative file paths

- **Terminology**
  - File name = file path = path
- **Every process has its own “working directory” (WD)**
  - In the shell
    - Can print it with ‘pwd’ (= print WD)
    - Can move between WDs with ‘cd’ (= change directory)
  - System calls to change WD
    - chdir, fchdir
- **A path is absolute if it starts with “/” (the root directory)**
  - E.g., /users/admin/jon
- **A path is relative (to the WD) otherwise**
  - If WD is “/”, then “/home/admin/jon” = “home/admin/jon”
  - If WD is “/home/” then “/home/admin/jon” = “admin/jon”
  - If WD is “/home/admin/” then “/home/admin/jon” = “jon”

# POSIX directory operations

- **mkdir(2) – create empty directory**
- **rmdir(2) – remove empty directory (fails if nonempty)**
- **Special directory names (relative paths that always exist)**
  - Current directory “.”
  - Parent directory “..” (root is its own parent)
  - Hidden by default by ‘ls’, as start with a “.”
- **Dir content traversal – opendir(3), readdir(3), closedir(3)**
  -

```
struct dirent *de;
DIR *p = opendir( "." );
while( (de = readdir(p)) != NULL )
    printf("%s\n", de->d_name);
closedir( p );
```

- What do we need to do to make it recursive?

# Hard links – intro

- **File != file name**
  - They're *not* the same thing
  - In fact, the name is *not* really part of the file's metadata
  - A file can have many names, which appear in unrelated places in the filesystem hierarchy
  - Creating another name => creating another “hard link”
- **System calls**
  - `link( srcpath, dstpath )`
  - `unlink( path )`
- **Shell**
  - `ln <srcpath> <dstpath>`
  - `rm <path>`

# Hard links – example

```
<0>dan@csa:~$ echo "hello" > f1
```

```
<0>dan@csa:~$ cat f1
```

hello

```
<0>dan@csa:~$ ls -l f1
```

-rw-r--r-- 1 dan 6 Jun 10 06:48 f1

```
<0>dan@csa:~$ ln f1 tmp/f2;
```

```
<0>dan@csa:~$ cat tmp/f2;
```

hello

```
<0>dan@csa:~$ ls -l f1;
```

-rw-r--r-- 2 dan 6 Jun 10 06:48 f1

```
<0>dan@csa:~$ echo "goodbye" > f1;
```

```
<0>dan@csa:~$ cat tmp/f2;
```

goodbye

# “1” = how many links to the file

# ln <src> <dst> creates the link

# f1 & f2 are links to same file

# so they have the same content

# ‘ls -l’ reveals how many links:

# 2 links

# override content of f1

# content of f2 also changes

# Hard links – when is a file deleted?

- **Every file has a “reference count” associated with it**
  - `link()`  $\Leftrightarrow$  `ref_count++`
  - `unlink()`  $\Leftrightarrow$  `ref_count--`
- **if( `ref_count == 0` )**
  - The file has no more names
  - It isn’t pointed to from any node within the file hierarchy
  - So it can finally be deleted
- **What if an open file is deleted? (its `ref_count==0`)**
  - Can we still access the file through the open FD(s)?
    - Yes
  - If  $>=1$  processes have the file open when the last link is removed
    - The link shall be removed before `unlink()` returns
    - But the removal of the file contents shall be postponed until all references (file descriptors) to the file are `close()`-ed
    - Have you seen files names that begin with “.nfs”?

# Hard links – what they do to the hierarchy

- Before hard links: tree graph
- After: any graph
- Hard links to directories?
  - Hard links to directories are typically disallowed & unsupported by the filesystem (though POSIX does allow directory hard links)
  - => Acyclic graph (no circles)
  - What's the benefit?
- Notable exception
  - HFS+, the Mac OS filesystem (since ~2007)
  - For time-machine (allows for cheap filesystem snapshots (why?))
    - [http://appleinsider.com/articles/07/10/12/road\\_to\\_mac\\_os\\_x\\_leopard\\_time\\_machine/page/3](http://appleinsider.com/articles/07/10/12/road_to_mac_os_x_leopard_time_machine/page/3)
  - But not APFS (= apple filesystem), successor of HFS+ (since ~2017);
    - Natively supports file “clone” and snapshots

# Directory hard links

- **A noted, most filesystems don't support directory hard links**
  - HFS+ is an exception
- **Still, all filesystems that adhere to POSIX provide at least some support to directory hard links**
  - Due to the special directory names “.” and “..”
  - What's the minimum number of hard links for directory?
    - 2 (due to “.”)
  - What's the maximum?
    - Depends on how many subdirectories nest in it (due to “..”)

# Symbolic links (soft links)

- **Unlike hard links**
  - Which point to the actual underlying file object
- **Symlinks (“shortcuts” in Windows terms)**
  - Point to a *name* of a “target” file (their content is typically this name)
  - They’re not counted in the file’s ref count
  - They can be “broken” / “dangling” (point to a nonexistent path)
  - They can refer to a directory (unlike hard links in most cases)
  - They can refer to files outside of the filesystem / mount point (whereas hard links must point to files within the same filesystem)
- **When applying a system call to a symlink**
  - The syscall would be seamlessly applied to the target file
  - For example, open(), stat(), chmod(), chown(), ...

# Symbolic links (soft links)

- **Exception: the unlink(2) syscall (and thus the ‘rm’ utility)**
  - Will remove the symlink, not the target file
- **Symlink-specific system calls**
  - symlink(target, linkPath) **creates** a symlink to the specified target file
  - readlink(sympath,buf,bufsiz) **reads** “content” of symlink = its target file
    - Q: what does read(2) do when applied to a symlink?
    - A: it’s a trick question; read(2) operates on a file descriptor, which is returned by open(2), which, as noted, applies to the target file
- **Shell**
  - ln –s <srcpath> <dstpath>

# Symbolic links – example

```
<0>dan@csa:~$ echo hey > f1
```

```
<0>dan@csa:~$ cat f1;                                # content of f1  
hey
```

```
<0>dan@csa:~$ ln -s f1 f2;                            # f2 is a symlink to f1
```

```
<0>dan@csa:~$ ls -l f2  
lrwxrwxrwx 1 dan 2 Jun 10 07:56 f2 -> f1          # notice arrow & perms
```

```
<0>dan@csa:~$ cat f2;                                # content of f1  
hey
```

```
<0>dan@csa:~$ rm -f f1;                            # f1 no longer exists  
<0>dan@csa:~$ cat f2;                            # so the 'cat' fails  
cat: f2: No such file or directory
```

# Implementation – inode

- **The OS data structure that represents the file**
  - Each file has its own (single) inode (= short for “index node”)
  - You can think of the inode as the true representative of “the file” or “the file object”
  - Internally, **file names “point” to inodes**
    - This inode is determined via the path-resolution algorithm (later)
- **The inode contains all the metadata of the file**
  - Timestamps, owner, permissions, ...
  - Pointers to the actual physical blocks, on the drive

# inodes & \*stat syscalls

- **stat(2) & fstat(2) retrieve metadata held in inode**
  - POSIX promises that at least the following fields are found in the stat structure and have meaningful values
    1. st\_dev ID of device containing file
    2. st\_ino inode number, unique for st\_dev
    3. st\_mode specifies file type & permissions (S\_ISREG(m), ...)
    4. st\_nlink number of hard links to the file
    5. st\_size file size in bytes
    6. st\_uid user ID of owner
    7. st\_gid group ID of owner
    8. st\_ctime last time metadata (=inode) **or data changed**
    9. st\_mtime last time **data changed**
    10. st\_atime last time **metadata (=inode) or data accessed**
    11. st\_blksize block size of this file object
    12. st\_blocks number of blocks allocated for this file object
  - Why do we need st\_size as well as st\_blksize & st\_blocks?

# inodes & \*stat syscalls

- **Istat(2)**
  - Exactly the same as stat(2) if applied to a hard link
  - But if applied to a symlink, would return the information of this symlink (*not* to the target of the symlink)
  - In this case, POSIX says that the only fields within the stat structure that you can portably use are:
    - st\_mode which will specify that the file is a symlink
    - st\_size symlink content length (= length of target filepath)
  - The value of the rest of the fields *could* be valid, but it is not specified by POSIX
  - Notably, it is not specified if a symlink has a corresponding inode
    - Will be discussed shortly

# Implementation – directory file

- A simple “flat” file comprised of directory entries

- For example, it could be a sequence of (Linux’s dirent):

```
struct dirent {  
    ino_t    d_ino;           /* POSIX: inode number      */  
    off_t    d_off;          /* offset to next dirent     */  
    short    d_reclen;        /* length of this record     */  
    char     d_type;          /* type of file              */  
    char    *d_name[NAME_MAX+1]; /* POSIX: null-term fname */  
    /* Must we always use NAME_MAX+1 chars? No! */  
};
```

- Importantly, note that

- The name of a file is **not** saved in the inode  
(recall: there can be many names/hardlinks associated with one file)
  - Rather, it is stored in the directory file as a simple string, in `d_name`
  - If the file is a symbolic link
    - The target file can be retrieved from the contents of the symlink

# Simple example

Ended here

d\_reclen = n1 bytes

| d_ino | d_off | d_reclen | d_type | d_name    |
|-------|-------|----------|--------|-----------|
| 133   | n1    | n1       | <reg>  | first.txt |

d\_reclen = n2 bytes

| d_ino | d_off | d_reclen | d_type | d_name           |
|-------|-------|----------|--------|------------------|
| 89    | n1+n2 | n2       | <reg>  | second.something |

d\_reclen = n3 bytes

| d_ino | d_off              | d_reclen | d_type | d_name |
|-------|--------------------|----------|--------|--------|
| 1002  | $\sum_{k=1}^3 n_k$ | n3       | <reg>  | README |

```
$ ls  
first.txt  
second.something  
README  
...
```

Comment: note that, unlike in this example:

- records may have “holes” between them (why?), and
- their order may be arbitrary (why?)

...

# Implementation – symlink

- **POSIX doesn't specify whether a symlink should have an inode**
  - In principle, symlinks could be implemented as directory entries and nothing else
- **In practice, filesystems typically define per-symlink inode-s**
  - In that case, if the name of the target file is short enough
  - Then the target file name can be saved in the inode itself (rather than in a data block pointed-to by the inode)
    - Such a target file is said to be “inlined”
    - Q: What's the benefit?
  - Example: ext4 (the default filesystem of Linux)
    - “The target of a symbolic link will be stored in [inode.i\_block] if the target string is less than 60 bytes long.”
    - <https://www.kernel.org/doc/html/latest/filesystems/ext4/ifork.html>

# Path resolution process

- **Resolving a path**
  - Get a file path
  - Return the inode that corresponds to the path
- **Syscalls that get a file name as an argument**
  - Need to resolve the path
- **To this end, syscalls use one algorithm**
  - Typically called “namei” (internally, within the kernel)
- **The namei algorithm consists of  $O(n)$  steps, where...**
  - $n$  = number of atom name components that comprise the file path
    - “Atom”
      - = a filepath that doesn’t contain a slash (“/”)
      - = a single name component
  - Including the components that comprise the symlinks along the path
    - Recursively speaking

# Path resolution process

- Assume the file path is /x/y/z
  - n=4 if x and y are directories and z is a regular file (4 because of "/")
  - n=5 if z is a relative-path symlink to a regular file w
    - $z \Rightarrow w$  (that is, w resides under /x/y/, and the target of the symlink z is the relative path "w")
  - n=8 if x is a directory, y = absolute-path-symlink to "/r/p/q/" (such that r, p, and q are directories), and z is a regular file
    - In this case, /x/y/z  $\Leftrightarrow$  /r/p/q/z
    - Because the path resolution process traverses the following components: (1) "/" (2) "x" (3) "y" (4) "/" (5) "r" (6) "p" (7) "q" (8) "z"
- Permissions are checked for each non-symlink atom in path
  - A user must be able to "search" all directories that lead to "z"
    - A directory is "searchable", if its "x" bit is on
  - The content of a "readable" (= "r" bit is on) directory can be listed (with 'ls')
  - A file can be added to a "writeable" (= "w" bit is on) directory
- Definition: the “terminal point” of a path
  - Is the last non-symlink atom in the path ("z" in our case)

# Path resolution process

Simplistic pseudo code version of a user-mode component-by-component open, which emulates ‘namei’

```
#define SYS( call ) if( (call) == -1 ) return -1

int my_open( char * fname ) {
    if( fname is absolute ) chdir( "/" ) + make fname relative
    foreach atom in fname do // e.g., atoms of "x/y" are "x" and "y"
        if( is symlink ) SYS( fd = my_open( atom's symlink target ) )
        else             SYS( fd =     open( atom /*checks perm!*/ ) )
    if( ! terminal ){ SYS( fchdir( fd )
                           SYS( close (fd)
                           break
    }
    return fd
}
```

# Path resolution process

- Q: n is a lower bound on the complexity of name; why?  
A: Because finding each individual directory component along the path may also be a linear process
  - Recall the procedure to print all files in a directory-file:

```
struct dirent *de;
DIR *p = opendir( "." );
while( (de = readdir(p)) != NULL )
    printf("%s\n", de->d_name);
closedir( p );
```
- We can speed up the name resolution process within the kernel by caching directory entries
  - Learn(ed) this in the Tutorial
- All the details of the (Linux) path resolution process:
  - [http://man7.org/linux/man-pages/man7/path\\_resolution.7.html](http://man7.org/linux/man-pages/man7/path_resolution.7.html)

# Path resolution process

Simplistic pseudo code version of a user-mode component-by-component open, which emulates ‘namei’

```
#define SYS( call ) if( (call) == -1 ) return -1

int my_open( char * fname ) {
    if( fname is absolute ) chdir( "/" ) + make fname relative
    foreach atom in fname do // e.g., atoms of "x/y" are "x" and "y"
        if( is symlink ) SYS( fd = my_open( atom's symlink target ) )
        else             SYS( fd =     open( atom /*checks perm!*/ ) )
        if( ! terminal ) SYS( fchdir( fd ) ) +  SYS( close( fd ) )
        else             break
    return fd
}
```

# Path resolution process

- Q: n is a lower bound on the complexity of name; why?  
A: Because finding each individual directory component along the path may also be a linear process
  - Recall the procedure to print all files in a directory-file:

```
struct dirent *de;
DIR *p = opendir( "." );
while( (de = readdir(p)) != NULL )
    printf("%s\n", de->d_name);
closedir( p );
```
- We can speed up the name resolution process within the kernel by caching directory entries
  - Learn(ed) this in the Tutorial
- All the details of the (Linux) path resolution process:
  - [http://man7.org/linux/man-pages/man7/path\\_resolution.7.html](http://man7.org/linux/man-pages/man7/path_resolution.7.html)

**Hard Disk Drives**

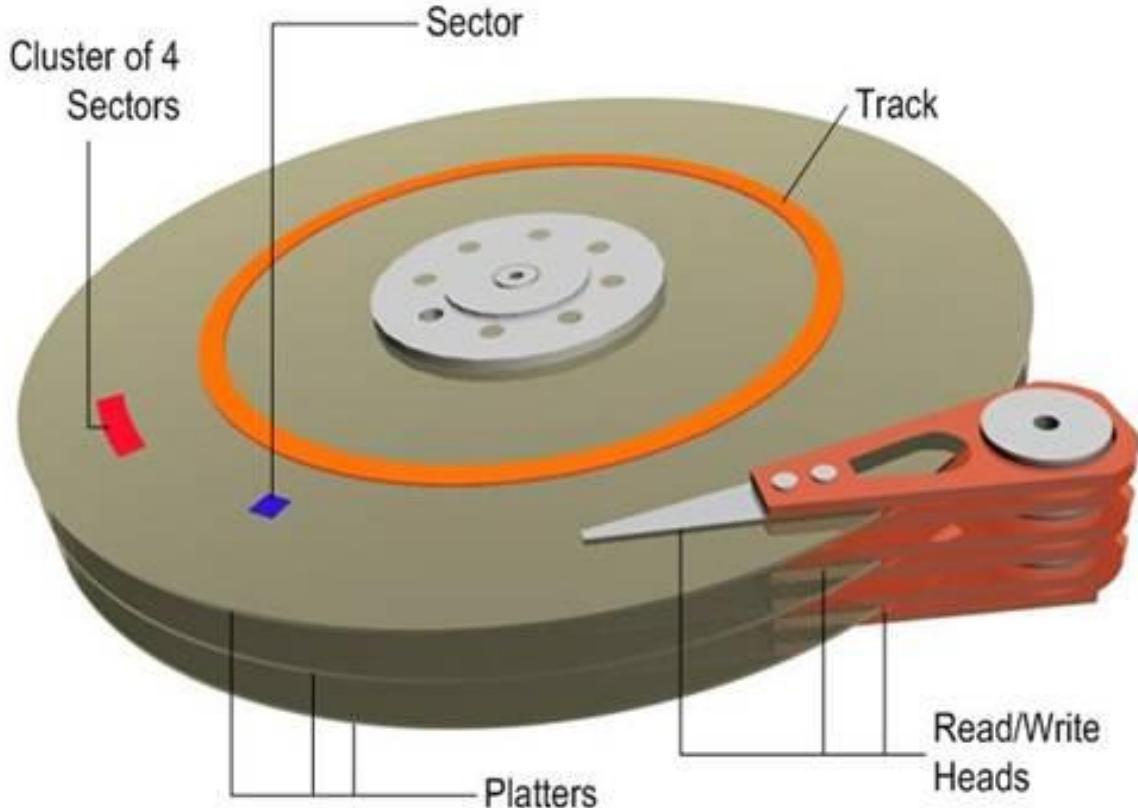
**HDD-S**

# HDDs



# HDDs

- **HDD latency is the sum of**
  - *Seek time*
    - Time of head to move to right track
  - Rotational latency
    - Delay caused by waiting for disk rotation to bring required sector under head
- **Performance**
  - A few milliseconds latency, say, ~5ms (how many IOPs if always seeking?)
  - Faster sequential accesses (~100MB/s), slower random access (~1MB)



# Example – enterprise level (mid 2023)

- **Seagate Exos X18:**

- “Enterprise highest-capacity HDD [...] **5yr warranty**”
  - Interface: SATA (6 Gb/s) or SAS (12 Gb/s),  
SAS is dual port
  - Price: \$299 (@ amazon.com)
  - Spindle speed 7.2K RPM (rounds per minute)
  - Capacity: 18 TB
  - Cache: 256 MB (DRAM)
  - Form factor: 3.5 inch (laptops ~~have had~~ 2.5" or 1.8" HDDs)
  - Max throughput: 270 MB/s = 258 MiB/s
  - Avg latency: 4.16 ms
  - 4KiB IOPS: 170 (read), 550 (write)
  - Sector size (bytes): 512 / 520 / 528 / 4096 / 4160 / 4224 (configurable)
  - MTBF: 2.5M hours (mean time between failures)
  - Error rate:  $< 1 \text{ in } 10^{15}$



# Example – enterprise level (mid 2013)

- **Seagate Cheetah 15K.7 SAS; by spec:**

- “Highest reliability rating in industry”
- Interface: **SAS** (Serial Attached SCSI; rotates faster and is pricier than SATA’s 5.4K/7.2K RPM HDDs)
- Price: \$260 (@ amazon.com)
- Spindle speed 15K (RPM = rounds per minute)
- Capacity: 600 GB
- Cache: 16MB (DRAM)
- Form factor: 3.5 inch
- Throughput: min=122 MB/s – max=204 MB/s
- Avg rotational lat.: 2.0 ms
- Typical avg seek time 3.4 ms (read) 3.9 ms (write)  
Typical single track seek 0.2 ms (read) 0.44 ms (write)  
Typical full stroke seek 6.6 ms (read) 7.4 ms (write)
  - “Full stroke” = head moves from outer to inner portion of the disk (max head motion); “single track” = move from track n to n±1
- Error rate < 1 in 10<sup>21</sup>



# Example – consumer grade (mid 2014)

- Western Digital 4 TB Green SATA III 5400 RPM  
64 MB Cache Bulk/OEM Desktop Hard Drive WD40EZRX

| 500GB | 1TB  | 1.5TB | 2TB  | 3TB   | 4TB   |
|-------|------|-------|------|-------|-------|
| \$54  | \$58 | \$75  | \$76 | \$110 | \$147 |

- Interface: SATA 6 Gb/s
- Form factor: 3.5"
- Max throughput: 150 MB/s
- (No data in spec about latency)
- Non-recoverable read errors per bits read < 1 in  $10^{14}$



# Western Digital HDDs – color lines

(generated by ChatGPT this morning: 2025-06-23)

| Color            | Intended use                        | Typical size†     | Amazon “street” price‡ |
|------------------|-------------------------------------|-------------------|------------------------|
| Blue             | Everyday desktop / general PC       | 1 TB (WD10EZEX)   | ≈ US \$49.50           |
| Black            | High-performance / gaming           | 2 TB (WD2003FZEX) | ≈ US \$92.99           |
| Purple           | Video-surveillance DVR/NVR          | 4 TB (WD42PURZ)   | ≈ US \$96.89           |
| Red (CMR / Plus) | 24×7 NAS up to 8 bays               | 4 TB (WD40EFZX)   | ≈ US \$149.99          |
| Red Pro          | Higher-workload NAS (up to 24 bays) | 8 TB (WD8005FFBX) | ≈ US \$229.99          |
| Gold             | Enterprise / data-center            | 8 TB (WD8004FRYZ) | ≈ US \$298.50          |

† “Typical size” = most commonly stocked capacity for that line on Amazon as of June 2025.

‡ Prices fluctuate with promos and stock; figures shown are the lowest “Buy New” offer visible on Amazon US at time of lookup (23 Jun 2025).

# (Anecdote: why “3.5 inch form-factor”?)

- **3.5" = 8.89 cm**
- **But standard dimensions of a 3.5 inch disk are**

- Depth : 5.79"  $\approx$  14.7 cm
- Height: 1.03"  $\approx$  2.61 cm
- Width: 4.00" = 10.16 cm



# (Anecdote: why “3.5 inch form-factor”?)

- $3.5" = 8.89 \text{ cm} = \sim 90\text{mm}$
- But standard dimensions of a 3.5 inch disk are

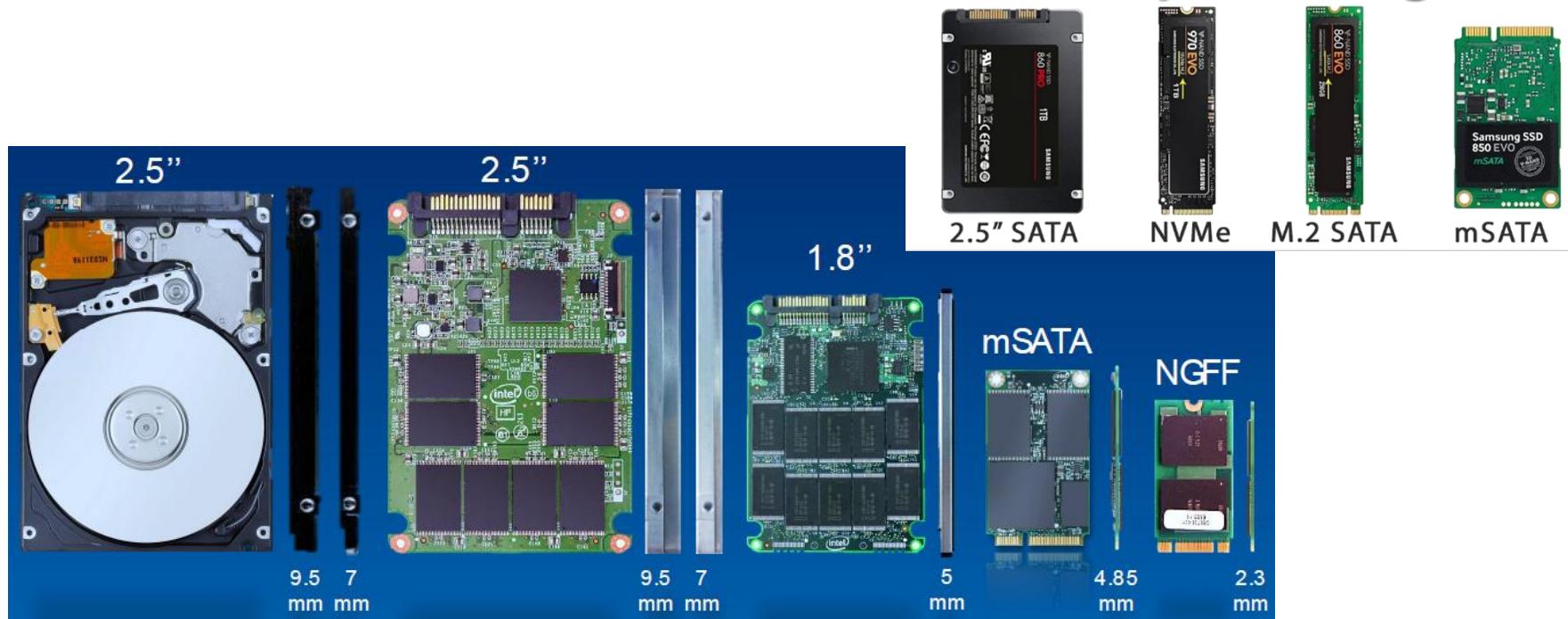
- Depth :    5.79"     $\approx 14.7 \text{ cm}$
- Height:   1.03"     $\approx 2.61 \text{ cm}$
- Width:    4.00"     $= 10.16 \text{ cm}$



- Historical reason
  - Capable of holding a platter that resides within a 3.5" ( $\approx 90\text{mm}$  width) floppy disk drive



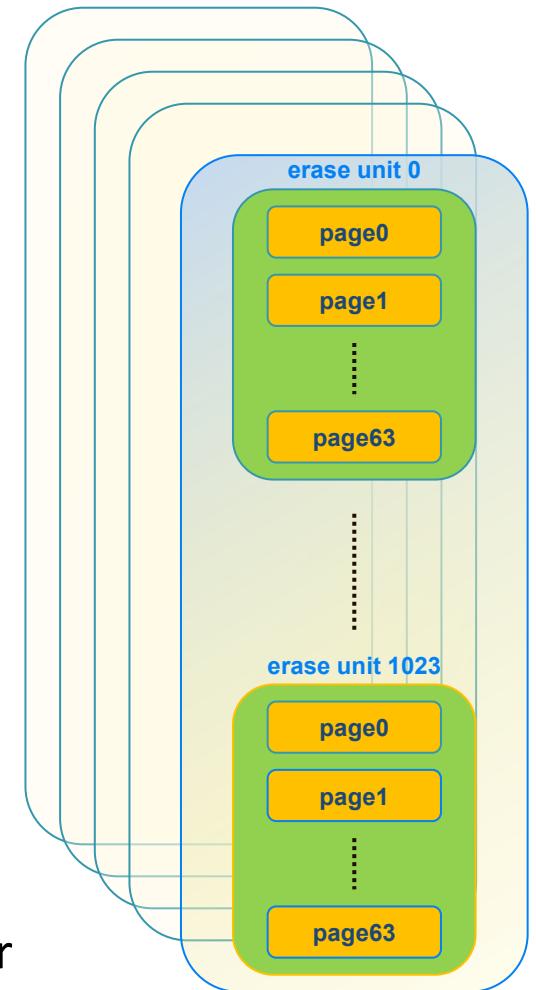
# SSDs = Solid State Drives



- Flash memory (often with DRAM cache, similarly to HDDs)
- No mechanical movement (unlike HDDs)
  - Still, sequential access is typically faster due to parallelism
    - Multiple flash chips in SSD, so SSD can prefetch to DRAM
- NVMe = non-volatile memory [for PCI] express
- See much more info in Storage Systems Course (by Gala)

# SSDs = Solid State Drives

- **Latency**
  - Read 100s to 10s of microseconds
  - Write can be the same or slower; it depends...
- **Layout**
  - Pages, organized in “erase units”
    - Cannot (re)write in-place
      - Must write the entire unit
    - Limited num of erase cycles (thousands)
  - Therefore, log-structured: write only sequentially to the end, and LBA internally virtualized, using a map (LBA => physical address)
  - Requires garbage collection
- **Read acceleration**
  - Multiple flash chips, so can parallelize
  - Full read bandwidth can be obtained by
    - Multiple outstanding random read requests, or
    - Sequential access (prefetching)



(Drawing from Gala's course)

# **FILESYSTEM LAYOUT ON DISK**

# Which blocks on disk constitute a file?

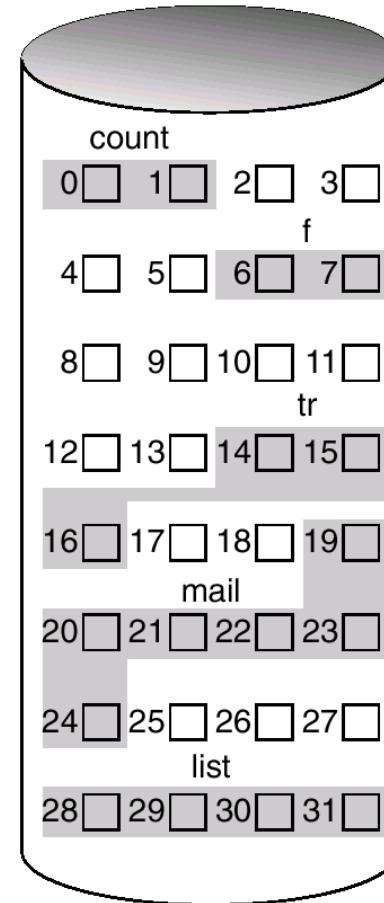
- Recall that HDDs are “block devices”
  - They are accessed in resolution of sectors
    - (Once, 512 bytes per sector; more recently, 4KB)
    - (Sequential access faster than random access)
  - Sectors are identified by their LBA (logical block address: 0,1,2, ...)
    - Such that adjacent LBAs imply physical contiguity on disk  
=> To accommodate fast sequential access
- However
  - Which sectors together constitute a given file?
  - And how do we find the right sector if we want to access the file at a particular offset?

# Need to understand...

- **Data structures**
  - On-disk structures that organize the data/metadata
    - Array?, linked list?, tree?,...
- **Access methods**
  - How to map system calls to the above structure
    - open(), read(), write(), ...
- **Unlike virtual memory, above decisions made purely in SW**
  - => Lots of flexibility
  - => Lots of filesystems,  
Literally, from AFS (Andrew FS) to ZFS (Sun's Zettabyte FS)
- **We'll focus on the simplistic VSFS = Very Simple File System**
  - A toy example to help us understand some of the concepts

# But before, let's rule out contiguous block allocation

- **Pros**
  - Simple file representation
  - Perfect sequential access when reading a file sequentially
  - Minimal seek for random access
    - Just one seek for each contiguous access
- **Cons**
  - External fragmentation
    - Deletions might leave holes of unusable sizes
  - Problem: how to append to a growing file?
    - Need to guess size in advance



| directory |       |        |
|-----------|-------|--------|
| file      | start | length |
| count     | 0     | 2      |
| tr        | 14    | 3      |
| mail      | 19    | 6      |
| list      | 28    | 4      |
| f         | 6     | 2      |

# VSFS – layout on disk

- Assume we have a really small disk of N=64 blocks (0,1,... 63)
  - Block size = sector size = 4 KB ( $\Rightarrow$  overall size = 4 KB x 64 block = 256 KB)

BBBBBBBB BBBB BBBB BBBB BBBB BBBB BBBB BBBB  
01234567 8 15 16 23 24 31 32 39 40 47 48 55 56 63

- VSFS layout

|----- The Data Region -----|  
S idIIIII DDDDDDDD DDDDDDDD DDDDDDDD DDDDDDDD DDDDDDDD DDDDDDDD DDDDDDDD  
01234567 8 15 16 23 24 31 32 39 40 47 48 55 56 63

- D = data blocks (56 out of the 64); all the rest is metadata:
  - I = inode table (on-disk array, 5 blocks out of the 8)
    - Assume 128 B inode  $\Rightarrow$  At most:  $5 \times 4 \text{ KB} / 128 \text{ B} = 5 \times 32 = 160$  files
  - d = bitmap of allocated data blocks (at most 56 bits are on)
    - (An entire block for ‘d’ and ‘i’ is too big for VSFS, as we only need 56 and 160 bits, respectively; but we do it for simplicity)
- S = “superblock” = filesystem information

# Filesystem superblock

- **Serves as a sort of “base class” for filesystems; contains information about this particular filesystem**
  - Magic number = identifies the filesystem (VSFS, there are many others)
  - The inode of the root directory (“/”)
  - How many inodes? (VSFS: 160)
  - How many data blocks? (VSFS: 56)
  - Start of inode table (VSFS: block 3)
  - [... and everything else needed to work with the filesystem]
- **Location of the superblock (of any FS) must be known**
  - For when “mounting” the system (next slide)

# Partitions, fdisk, mkfs, mount

- A disk can be subdivided into several “partitions” using ‘fdisk’
  - Partition = contiguous disjoint part of the disk that can host a filesystem
- Disks are typically named
  - /dev/sda, /dev/sdb, ...
- Disk partitions are typically named
  - /dev/sda1, /dev/sda2, ...
  - Listing partitions on disk /dev/sda:

```
<0>dan@pomela1:~$ sudo fdisk -l /dev/sda
```

```
Disk /dev/sda: 1979.1 GB, 1979120025600 bytes
255 heads, 63 sectors/track, 240614 cylinders, total 3865468800 sectors
Units = sectors of 1 * 512 = 512 bytes
```

| Device    | Boot | Start  | End        | Blocks     | Id | System    |
|-----------|------|--------|------------|------------|----|-----------|
| /dev/sda1 | *    | 2048   | 499711     | 248832     | 83 | Linux     |
| /dev/sda2 |      | 501758 | 3865466879 | 1932482561 | 5  | Extended  |
| /dev/sda5 |      | 501760 | 3865466879 | 1932482560 | 8e | Linux LVM |

# Partitions, fdisk, mkfs, mount

- **Every filesystem implements a mkfs (“make FS”) utility**
  - Creates an empty filesystem of that type on a given partition
  - (What’s the implementation of mkfs for VSFS?)
- **After an FS is created, need to “mount” it to make it accessible**
  - `mount` = עלה, הפעיל; רכיב; ה
    - תכלית, ארגן, הרכיב, התקיין
  - For example
    - `mount -t ext3 /dev/sda1 /home/users`  
Mounts the filesystem that resides in the partition `/dev/sda1` onto the directory `/home/users`; the `sda1` filesystem is of the type `ext3`
  - From now on
    - `ls /home/users`  
will list the content of the root directory of the said `ext3` filesystem
  - The ‘`mount`’ shell utility utilizes the ‘`mount`’ system call
  - Obviously, `mount` makes use of the superblock of the filesystem
  - Reverse of `mount`: `umount`

# Partitions, fdisk, mkfs, mount

- Running **mount** in shell with no args prints all mounts, e.g.

```
proc      on /proc      type proc (rw)
sysfs    on /sys       type sysfs (rw)
tmpfs     on /dev/shm   type tmpfs (rw)
/dev/sda1 on /          type ext3 (rw)
/dev/sda2 on /usr       type ext3 (rw)
/dev/sda3 on /var       type ext3 (rw)
/dev/sda6 on /tmp       type ext3 (rw)
```

- For more details see

- “Learn Linux, 101: Hard disk layout”
  - <https://www.ibm.com/developerworks/library/l-lpic1-v3-102-1>
- “Learn Linux, 101: Create partitions and filesystems”
  - <http://www.ibm.com/developerworks/library/l-lpic1-v3-104-1>

# VSFS – finding an inode

- **VSFS layout**

|                             |      |          |          |          |          |          |          |          |          |          |          |
|-----------------------------|------|----------|----------|----------|----------|----------|----------|----------|----------|----------|----------|
| ----- The Data Region ----- |      |          |          |          |          |          |          |          |          |          |          |
| Sid                         | IIII | DDDDDDDD |
| 01234567                    | 8    | 15 16    | 23 24    | 31 32    | 39 40    | 47 48    | 55 56    | 63       |          |          |          |

- **Given inumber (=index of inode in table), find inode's sector:**

- sector = (inodeStartAddr + (inumber x sizeof(inode\_t)) / blockSize
  - |--- in bytes ---|
  - |--- in bytes --|

- **Once found, the inode tells us all the file's metadata**

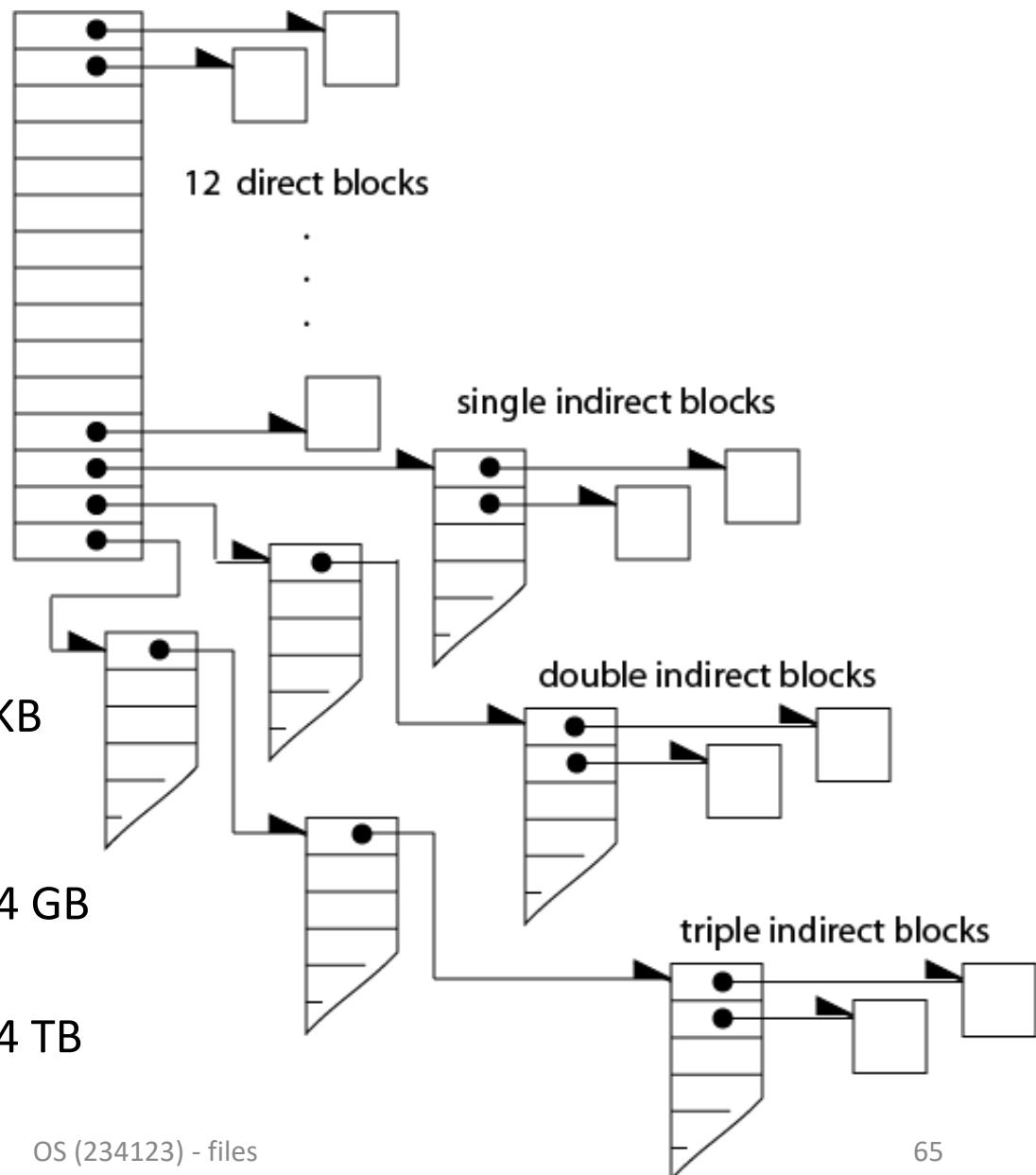
- Number of allocated blocks, protection, times, ..., and
- Pointer(s) to where data is stored

- **Pointers could be, for example,**

1. Direct
  - Point directly to all data blocks =>  $|file| \leq \text{pointerNum} \times \text{blockSize}$
2. Indirect (multi-level)
3. Linked list

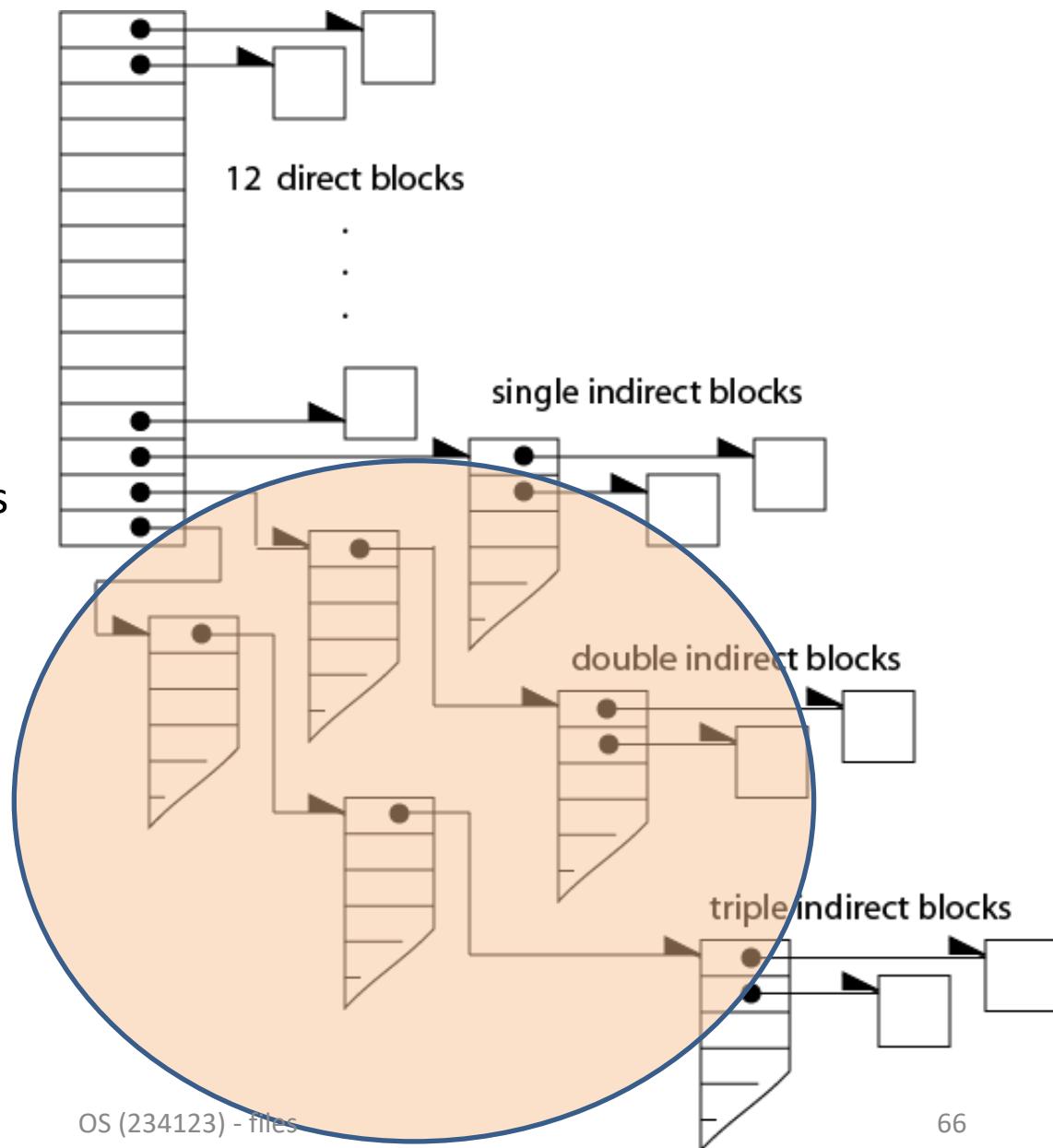
# Multi-level index (in the classic Unix FS)

- Assume each block pointer is 4 bytes long, and each block is 4KB
- 12 pointers in inode point directly to data blocks
  - Consumes  $4 \times 12 = 48$  B
- Single-indirect pointer points to a block completely comprised of pointers to data blocks
  - 1024 data blocks
  - $(12 + 1024) \times 4\text{KB} = 4144\text{ KB}$   
 $= \sim 4\text{MB}$
- Double-indirect adds
  - $1024^2 \Rightarrow 4\text{KB} \times 2^{20} = 4\text{ GB}$
- Triple-indirect adds
  - $1024^3 \Rightarrow 4\text{KB} \times 2^{30} = 4\text{ TB}$



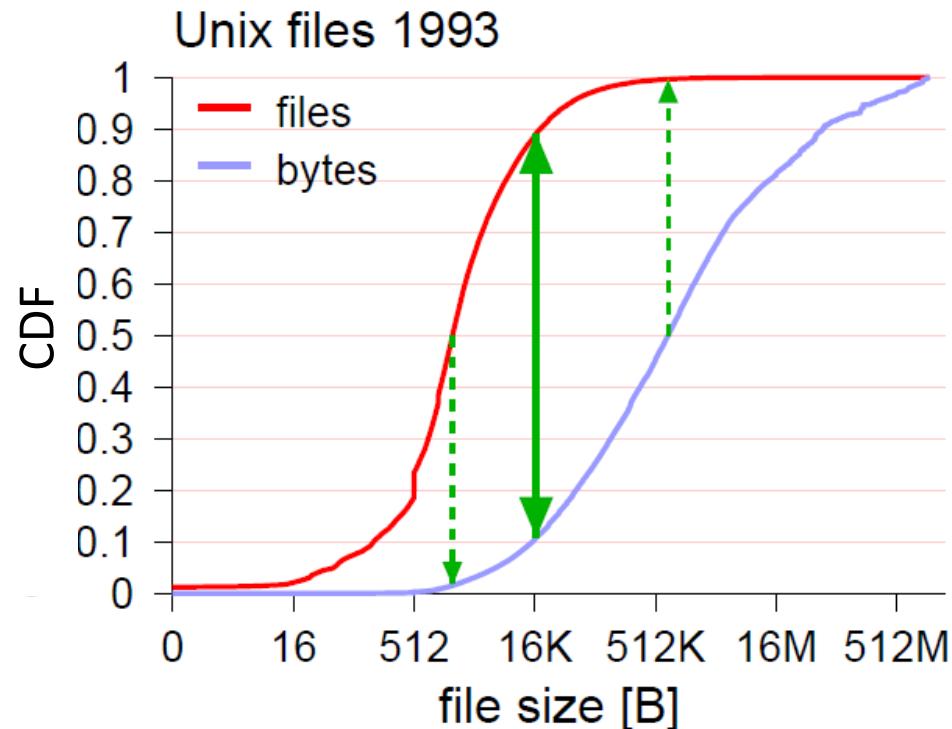
# Multi-level index (in the classic Unix FS)

- The indirect meta-data blocks are blocks like “ordinary” data blocks
- They do not reside in the inode
- The filesystem allocates them exactly as it allocates “ordinary” blocks
- They are not included in the “size” of the file
- But they are included in `st_blocks` in the `stat` structure



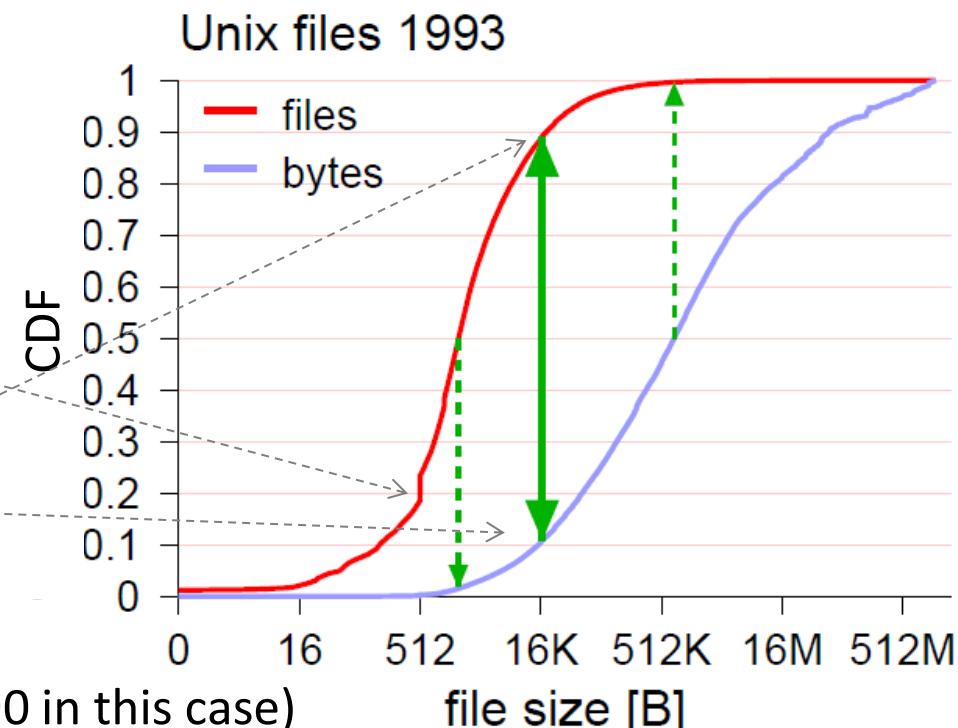
# Why is hierarchy imbalanced like so?

- Data from
  - 12 million files from over 1000 Unix filesystems
- Y axis
  - CDF = cumulative distribution function = how many items [in %] are associated with the corresponding X value, or are smaller
- ‘Files’ curve
  - Percent of existing files whose size is  $\leq x$
- ‘Bytes’ curve
  - Percent of stored bytes that belong to a file whose size is  $\leq x$



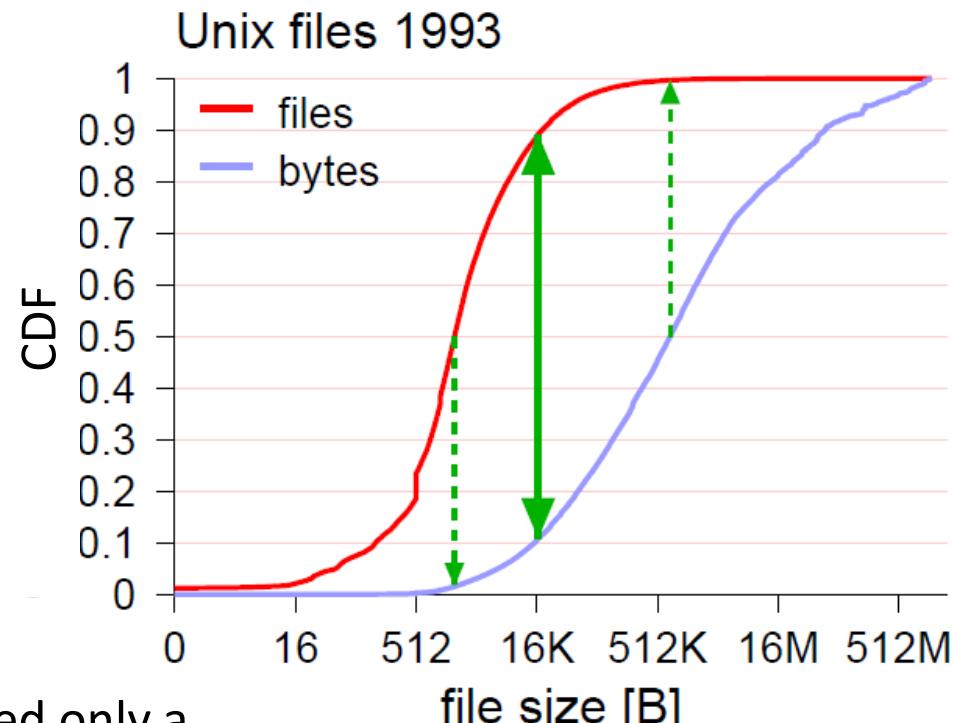
# Why is hierarchy imbalanced like so?

- Data from
  - 12 million files from over 1000 Unix filesystems
- Observations
  - ~20% of the files  $\leq 512$  B
  - ~90% of the files  $\leq 16$  KB
    - ~10% of stored bytes belong to files that are up to 16 KB
- Vertical arrows
  - Middle = “joint ratio” ( $= 10/90$  in this case)
    - Determined by where sum of the two curves = 1
    - 90% of the files (are so small they) account for 10% of all stored bytes
    - 10% of the files (are so big they) account for 90% of all stored bytes
  - Left = half of the files are so small they account for only  $\sim 2\%$  of bytes
  - Right = half of stored bytes belong to (only)  $\sim 0.3\%$  of the files



# Why is hierarchy imbalanced like so?

- This phenomenon is commonly observed in computer systems
  - A typical file is small, but a typical byte belongs to a large file
  - A typical job/process is short, but a typical cycle belongs to a long job/process
  - A typical youtube clip is viewed only a few times, but a typical view is associated with a clip that has been viewed very many times
- The phenomenon has been named “mass-count disparity”
  - A small number of items account for the majority of mass, whereas all small items together only account for negligible mass
  - (Disparity = נבדלות, שוני; in our example: mass=bytes & count=files)



# Extents

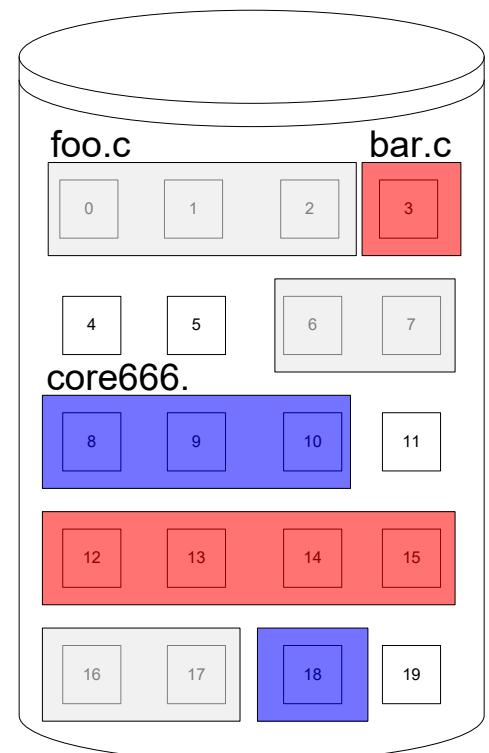
- **Motivation**

- Hierarchical block structure could mess up sequential access
- Also, nowadays, many more files size = O(MB)
- Dynamic append/truncate/delete I/O ops might hinder sequential access (= every leaf block might reside someplace else)

| Catalog  |       |        |        |
|----------|-------|--------|--------|
| foo.c    | (0,3) | (6,2)  | (16,2) |
| bar.c    | (3,1) | (12,4) |        |
| core666. | (8,3) | (18,1) |        |

- **Solution: extent-based allocation**

- Extent = contiguous area comprised of variable number of blocks
- inode saves a list of extents = (pointer, size) pairs



# Extents

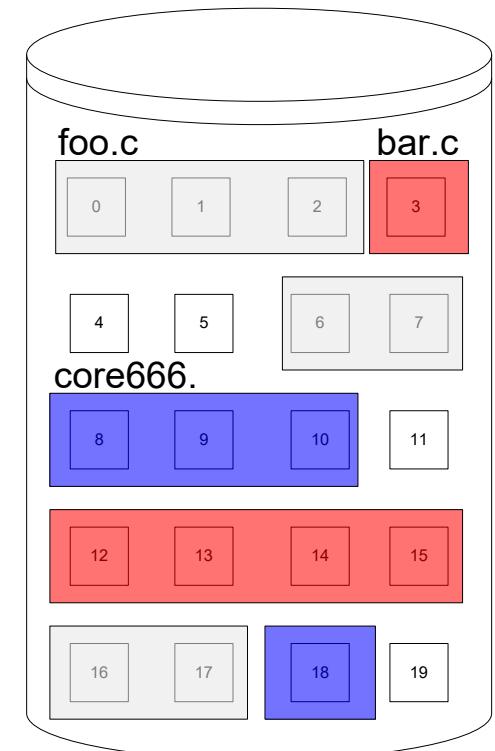
- **Pros**

- Promotes sequentiality (less pointer chasing, simplified hierarchy)
- Efficient seek (one I/O op when extent found)

- **Cons**

- Harder management: like memory management in the face of dynamic deletes/appends
- Need to maintain holes with usable sizes
- Need to predict size of file
  - Over-prediction => internal fragmentation
  - Under-prediction => allocate another extent => more seeks when searching for a given offset

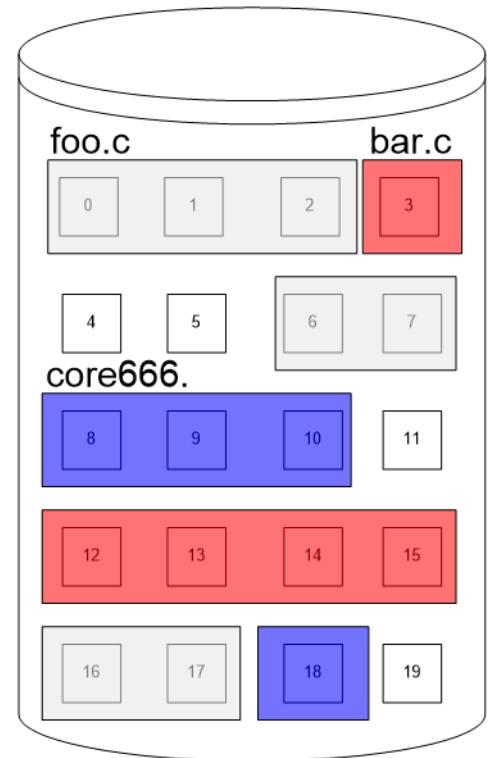
| Catalog  |       |        |        |
|----------|-------|--------|--------|
| foo.c    | (0,3) | (6,2)  | (16,2) |
| bar.c    | (3,1) | (12,4) |        |
| core666. | (8,3) | (18,1) |        |



# Extents

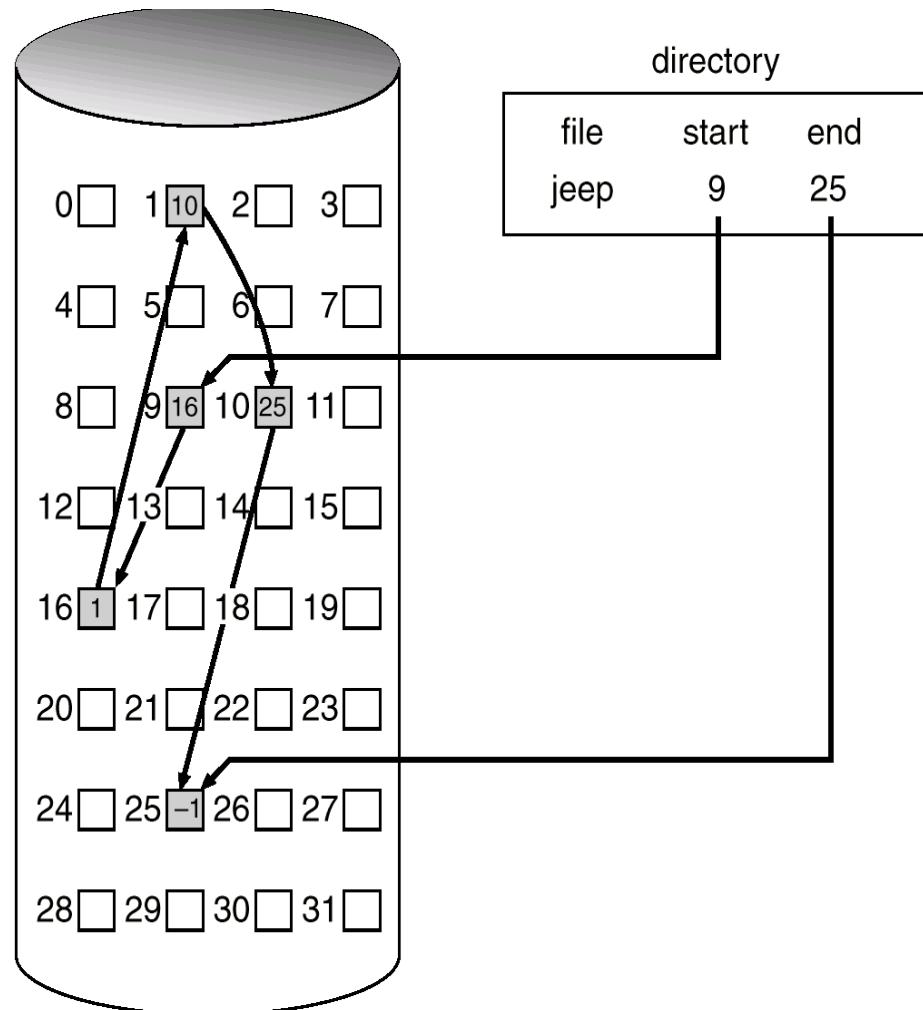
- **In ext4 (current default FS for Linux)**
  - With 4KB block, size of a single extent can be between: 4KB ... 128MB (32K blocks)
  - Up to 4 <base,size> extents stored in inode
  - Hierarchical if more than 4
    - Indexed by an “Htree” (similar to Btree but with constant depth of 1 or 2)
    - Allows for efficient seek/offset search
- **Nowadays supported by most filesystems**
  - ntfs, ext4, hfs+/apfs, btrfs, reiser4, xfs, ...
- **Comments**
  - ext2 = the ‘extended file system’ for Linux
  - ext3 = ext2 + journaling (“journal” keeps track of uncommitted changes)
  - ext4 = ext3 + extents (=> “ext” name unrelated to “extent”)
  - ext4 is backward compatible: ext2 and ext3 can be mounted as ext4

| Catalog  |       |        |        |
|----------|-------|--------|--------|
| foo.c    | (0,3) | (6,2)  | (16,2) |
| bar.c    | (3,1) | (12,4) |        |
| core666. | (8,3) | (18,1) |        |



# Linked list block allocation

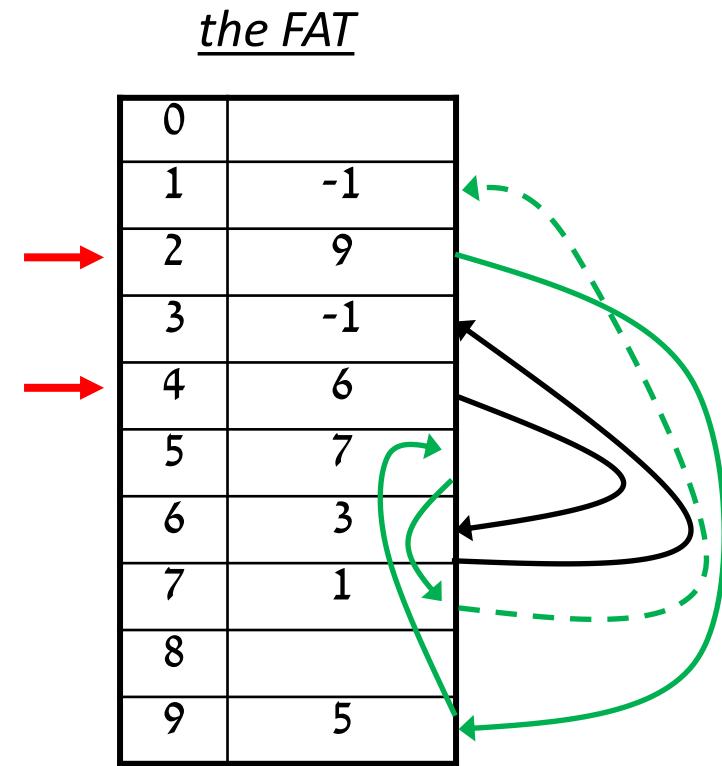
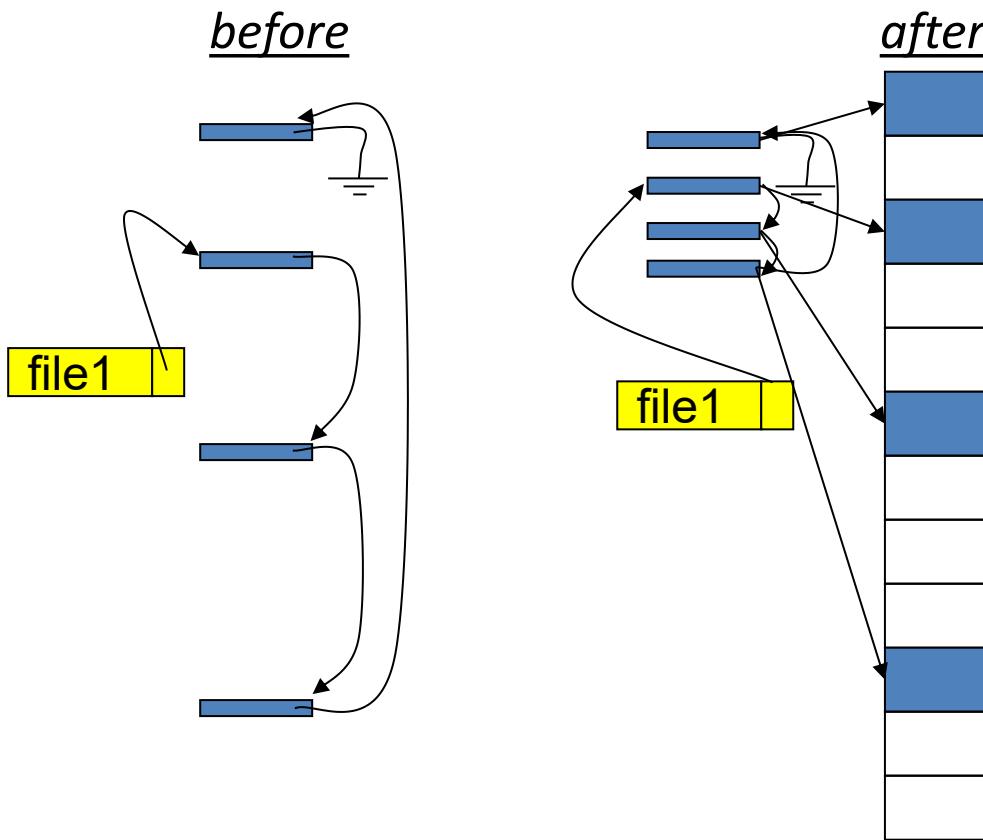
- A file could be a linked list
  - Such that every block points to the next
  - Each directory entry points to 1<sup>st</sup> and last file blocks of file
    - (Need last for quick append)
- Performance overhead
  - Because seek operations require linear number of ops
  - Each op is a random disk read
- Optimization
  - Move pointers out of blocks
  - Put them in external table that can be cached in memory
  - Rings a bell?



# FAT filesystem

- **FAT = file allocation table**
  - The DOS filesystem, still lives
  - Named after its main data structure

| name |     | start block |
|------|-----|-------------|
| foo  | ... | 2           |
| bar  | ... | 4           |



# FAT

- **Layout**
  - One table for all files
  - One table entry for every disk block
  - -1 (0xFFFF) marks “the end”
  - Directory file: content of each entry points to start of file
- **FAT (table) is copied to (cached in) memory**
  - But of course, it must also be saved on disk (why?)
  - Solves random access problem of file pointers
- **Pros**
  - Simple to implement:
    - File append, free block allocation management, easy allocation
  - No external fragmentation
  - Fast random access since table is in memory (for block pointers, not necessarily for the blocks themselves)

# FAT

- **Cons**
  - File contiguity can be lost (this is why extents were invented)
  - Table can be huge
    - $32\text{ GB (disk)} / 4\text{KB (block)} = 2^5 \times 2^{30} / 2^{12} = 2^{23} = 8\text{ M}$
    - Assuming 4B pointer, this means 32 MB table
    - Exercise: what's the size of FAT for a 4TB disk?

Redundant Array of Independent Disks

**RAID**

# Motivation

- **Problem #1**
  - Sometimes disks fail
- **Problem #2**
  - Disk transfer rates can be much slower than CPU performance
- **Solution – part A: striping**
  - We can use multiple disks to improve performance
  - By *striping* files across multiple disks (placing parts of each file on a different disk), we can use parallel I/O to improve the throughput
- **But striping makes problem #1 even worse**
  - Reduces reliability: 100 disks have 1/100th the MTBF (=mean time between failures)
- **Solution – part B: redundancy**
  - Add redundant data to the disks (in addition to striping)

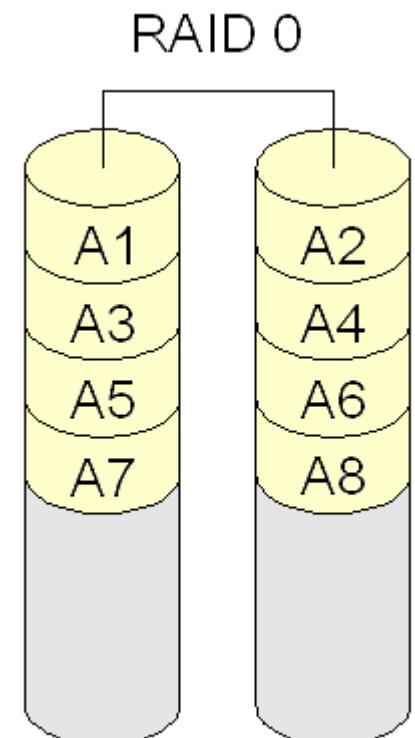
# RAID (Redundant Array of Inexpensive Disks)

- **Individual disks are small and relatively cheap**
  - So it's easy to put lots of disks (10s, 100s) in one box/rack for
    - Increased storage, performance, and reliability
- **Data plus some redundant information are striped across the disks in some way**
  - How striping is done is key to performance & reliability
  - We decide on how to do it depending on the level of redundancy and performance required
  - Called "RAID levels"; standard RAID levels:
    - RAID-0, RAID-1, RAID-4, RAID-5, RAID-6
- **Proposed circa 1987**
  - By David Patterson, Garth Gibson, and Randy Katz (@ Berkeley)



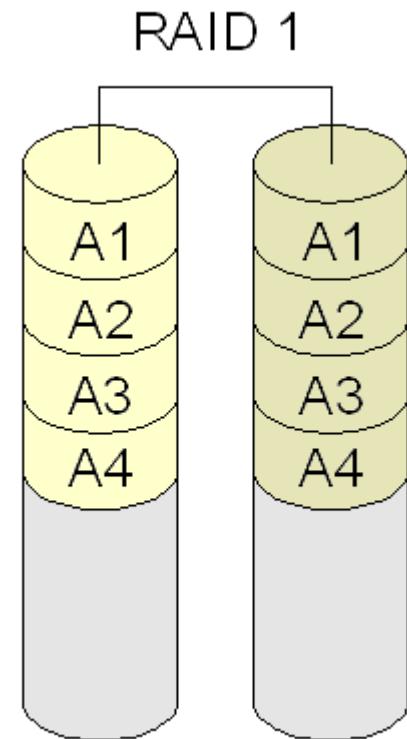
# RAID-0

- **Non-redundant disk array**
  - Files striped evenly across  $N \geq 2$  disks
  - Done in block resolution: in principle, technically, a block can be as small as 1 byte; but is a multiple of drive's sector size in most standard RAID levels taught today (0,1,4,5,6)
- **Pros**
  - High read/write throughput
    - Potentially  $N$  times faster
    - Best write throughput (relative to other RAID levels)
  - Faster aggregated seek time, that is
    - The seek time of one disk remains the same
    - But if we have  $N$  independent random seeks, potentially, we can do it  $N$  times faster, on avg
- **Con**
  - Any disk failure results in data loss



# RAID-1

- **Mirrored disks**
  - Files are striped across half the disks
  - Data written to 2 places: data disk & mirror disk
- **Pros**
  - On failure, can immediately use surviving disk
  - Read performance: similar to RAID 0. Why?
  - How about write performance?
- **Cons**
  - Wastes half the capacity
- **Related: nowadays, in the cloud / data center**
  - Replicating storage systems are prevalent
  - Often 3 replicas of each “hot data chunk” (chunk sizes are usually in MBs, to amortize seeks; hot = used a lot)
  - Replicas get “randomly” assigned to disks to balance the load
  - That is, not using classical RAID 1 disk pairs; rather, every disk is striped across many (potentially all) other disk



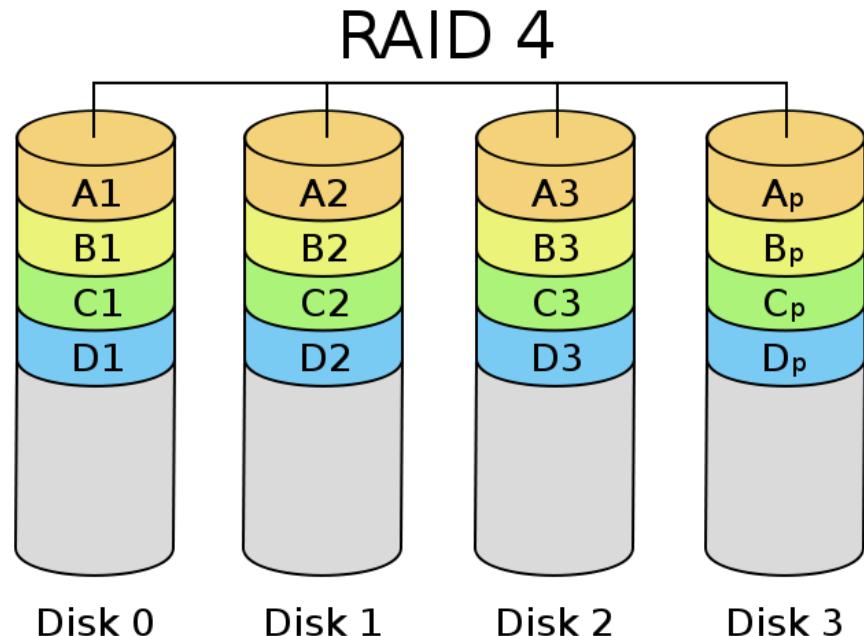
# Reminder: parity/xor

- **Modulo-2 computations**
  - 0 = even number of 1s
  - 1 = odd number of 1s
- **When one sequence bit is lost**
  - Can reconstruct it using the parity
- **In storage systems**
  - Parity computed on a block level

| # | bit sequence | parity/xor |
|---|--------------|------------|
| 1 | 000          | 0          |
| 2 | 001          | 1          |
| 3 | 010          | 1          |
| 4 | 011          | 0          |
| 5 | 100          | 1          |
| 6 | 101          | 0          |
| 7 | 110          | 0          |
| 8 | 111          | 1          |

# RAID-4

- **Use parity disks**
  - Each block (= multiple of sector) on the parity disk is a parity function (=xor) of the corresponding blocks on all the N-1 other disks
- **Read ops**
  - Access data disks only
- **Write ops**
  - Access data disks & parity disk
  - Update parity
- **Failure => “degraded read”**
  - Read remaining disks plus parity disk to compute missing data
- **Pros**
  - In terms of capacity, less wasteful than RAID-1 (wastes only 1/N)
  - Read performance similar to RAID-0
  - How about write performance? Next slide...

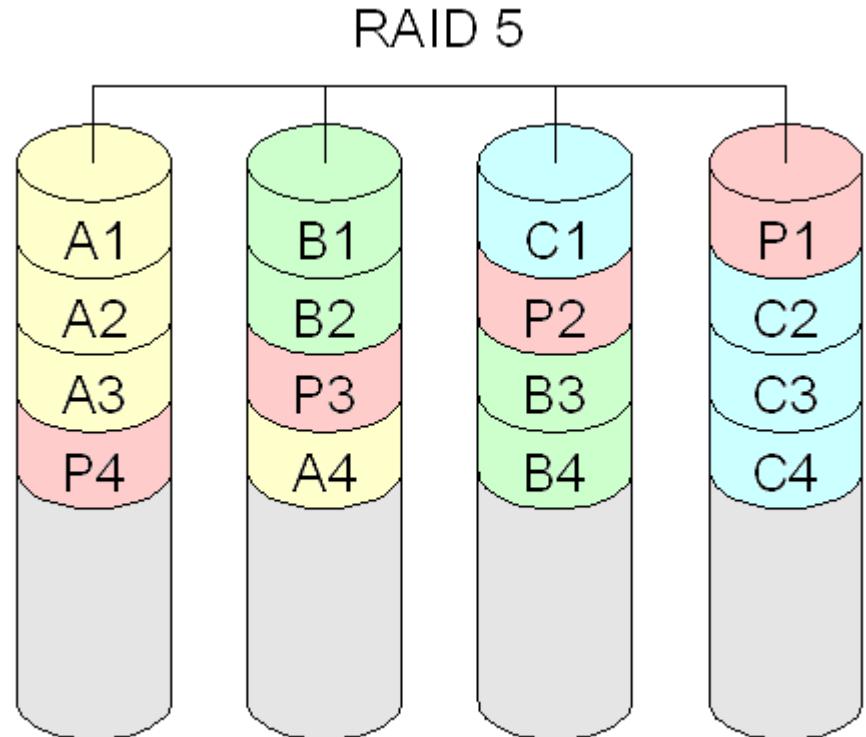


# RAID “small write problem”

- **Applicable to all RAIDs that utilize “erasure codes” (like parity)**
  - Write is a slower op
  - As it requires to also update the parity
- **When performing a small write (e.g., one block), we need to**
  1. Read the block of the data
  2. Read the block of the parity
  3. Compute difference between old and new data
  4. Write-update parity block accordingly
  5. Write-update data block
- **We thus need to perform 4 IOs**
  - Instead of one!
- **Why is this just a \*small\* write problem?**
  - When writing an entire stripe (A1–A3), we don’t need to read anything
  - Instead, we can compute their parity and write it to Ap

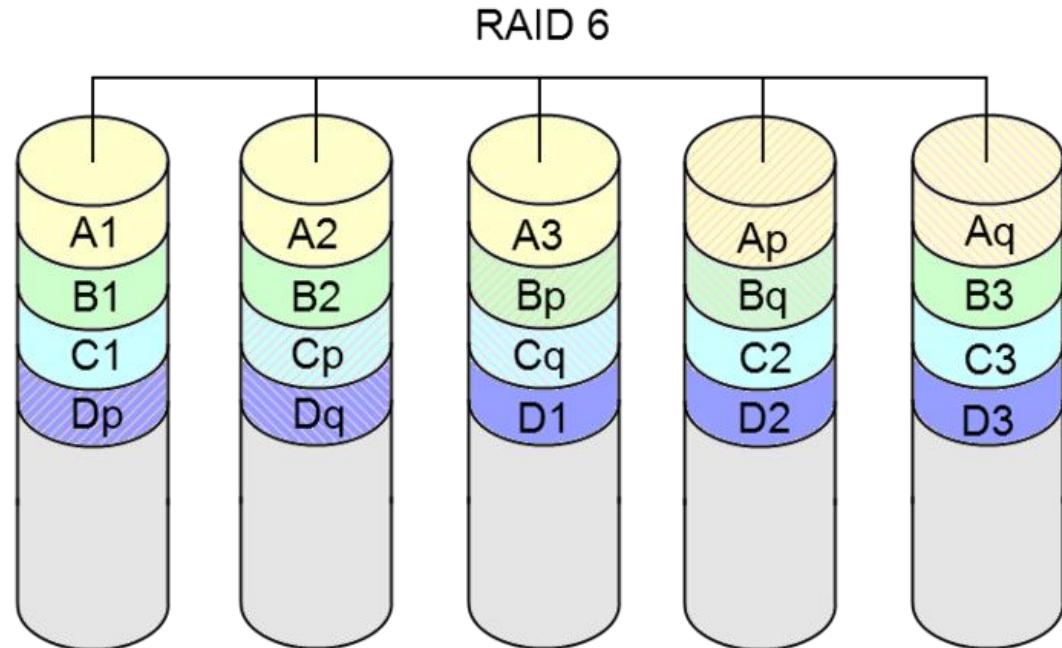
# RAID-5

- **Similar to RAID-4, but uses block interleaved distributed parity**
  - Distribute parity info across all disks
  - A “stripe” is a concurrent series of blocks ( $A_1B_1C_1, A_2B_2C_2, \dots$ )
  - The parity of each stripe resides on a different disk
- **Pros**
  - Like RAID-4, but better because it eliminates the hot spot disk
  - E.g., when performing two small writes in RAID-4
    - They must be serialized
  - Not necessarily so in RAID-5
    - => better performance



# RAID-6

- Extends RAID-5 by adding an additional “parity” block
  - $A_p$  and  $A_q$  must be independent of each other, algebraically speaking
- Pros & cons relative to RAID-5
  - Can withstand 2 disk failures (2 equations, can find 2 variables)
  - But wastes  $2/N$  rather than  $1/N$  of the capacity



# FYI: RAID-2 & RAID-3

- **Like RAID-4**
  - But in bit and byte resolution, respectively
- **Requires “lockstep”**
  - All disks spin synchronously => almost never used

# Notation: RAID n+k

- **We have n+k disks and can tolerate k failures**
  - n = blocks of regular data
  - k = blocks that provide redundancy, using erasure codes
- **Example**
  - Previous slide: 3+2
  - If we have 12 disks and we'd like to be able to tolerate 3 failures: 9+3

# Erasure vs. replication in a data center

end lecture

- **Erasure ( $n+k$ )**
  - Pros: optimizing capacity, “wasting” only  $k/(k+n)$  on redundancy
    - RAID-5:  $1/(1+n)$ ; RAID-6:  $2/(2+n)$ ; typically:  $< 1/2$  (as  $k < n$ )
  - Cons
    - Small write problem (solution: make “block” much bigger)
    - Degraded reads = occasional high read latency due to having to reconstruct data by reading  $n$  blocks to compute a missing block
    - Repair traffic = having to wire said  $n$  blocks through the network to a machine that computes the missing block
    - Potentially indirectly hurting throughput & latency
      - To reconstruct a failed disk need to read  $n$  disks (lots of storage IO) and send their data through the network (lots of network IO)
- **Replication ( $r$ )**
  - Pros: none of the above cons
  - Cons: wasting  $(r-1)/r$  of the capacity
- **Encoding to accommodate hot/cold data: active research field**