

CSE 506 Operating Systems

Dongyoon Lee

About me

- Dongyoon Lee
- Assistant Professor in CS @ SBU
- Ph.D in CSE @ U of Michigan, Ann Arbor (2013)
- Email: dongyoon@cs.stonybrook.edu
- Office: 339 @ New Computer Science (NCS)
- Homepage: <https://www3.cs.stonybrook.edu/~dongyoon/>

Research interests: OS + Compiler + HW for

- Reliability
 - Concurrency bug [CGO'18][EuroSys'17][ASPLOS'17][ASPLOS'16]
 - Regular expression [FSE'19][ASE'19][ASE'19]
 - Transient fault (soft error) tolerance [MICRO'16][SC'16][LCTES'15]
- Security
 - Linux kernel permissions [USENIX Security'19]
 - Denial of service [S&P'21][USENIX Security'18][FSE'18][EuroSec'17]
 - Memory safety [ASPLOS'19][MICRO'18]

Research interests: OS + Compiler + HW for

- Performance
 - Edge stream processing [ATC'19]
 - Distributed key value stores [SC'18]

About this course

- CSE 506: Operating Systems
- This semester: Linux Kernel Programming
- Goals
 - Understand core subsystems of the Linux kernel in depth
 - Design, implement, and modify Linux kernel code and modules for these subsystems
 - Test, debug, and evaluate the performance of systems software in kernel or user space, using debugging, monitoring and tracing tools

What is the *Linux Kernel*?

- One of operating system kernel
 - e.g., Windows, FreeBSD, OSX, etc.
- What does an OS do for you?
 - **Abstract** the hardware for convenience and portability
 - **Multiplex** the hardware among multiple applications
 - **Isolate** applications to contain bugs
 - Allow **sharing** among applications

View: layered organization

- **User:** applications (e.g., vi and gcc)
- **Kernel:** file system, process, etc.
- **Hardware:** CPU, mem, disk, etc.

→ **Interface** between layers

View: core services

- Processes
- Memory
- File contents
- Directories and file names
- Security
- Many others: users, IPC, network, time, terminals, etc.

→ Abstraction for applications

Example: system calls

- Interface : applications talk to an OS via system calls
- Abstraction : process and file descriptor

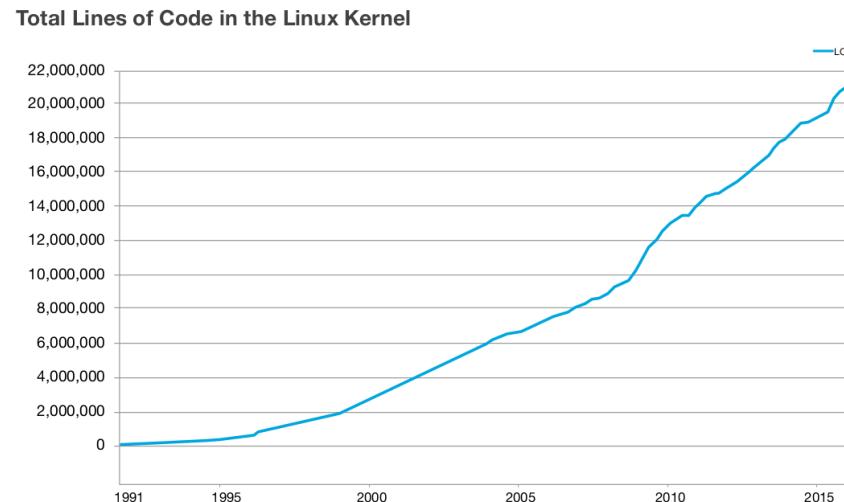
```
fd = open("out", 1);
write(fd, "hello\n", 6);
pid = fork();
```

Why is Linux kernel interesting?

- OS design deals with conflicting goals and trade-offs
 - Efficient yet portable
 - Powerful yet simple
 - Isolated yet interactable
 - General yet performant
- Open problems: multi-core and security
- How does a state-of-the-art OS deal with above issues?
 - **Hack the Linux kernel!**

Why is Linux kernel interesting?

- Extremely large software project
 - more than 25 million lines of code
 - 7,500 lines of code are added every day!



Why is Linux kernel interesting?

- Very fast development cycles
 - release about every 70 days
 - 13,000 patches / release
 - 273 ~~250~~ companies / release (or 1,600 developers / release)
- One of the most well-written/designed/maintained C code
- Ref: [Linux Foundation Kernel Report 2017](#)

Linux is eating the World

- 85.1% of smartphones and tables run Linux (Android)
 - iOS: 14.9%
- 98% of top 1 million web servers run Linux
- 99% of super computers run Linux
- SpaceX: [From Earth to orbit with Linux and SpaceX](#)
- Ref: [Usage share of OS](#)

It is good for your job search

- Contributions from unpaid developers had been in slow decline
 - 14.6% (2012) → 13.6% (2013) → 11.8% (2014) → 7.7% (2015)
- Why?
 - “There are many possible reasons for this decline, but, arguably, the most plausible of those is quite simple: **Kernel developers are in short supply, so anybody who demonstrates an ability to get code into the mainline tends not to have trouble finding job offers.**”
- Ref: [Linux Foundation Kernel Report 2017](#)

How to become a Linux kernel developer



- [Interview of Sarah Sharp](#)

Who should take this course?

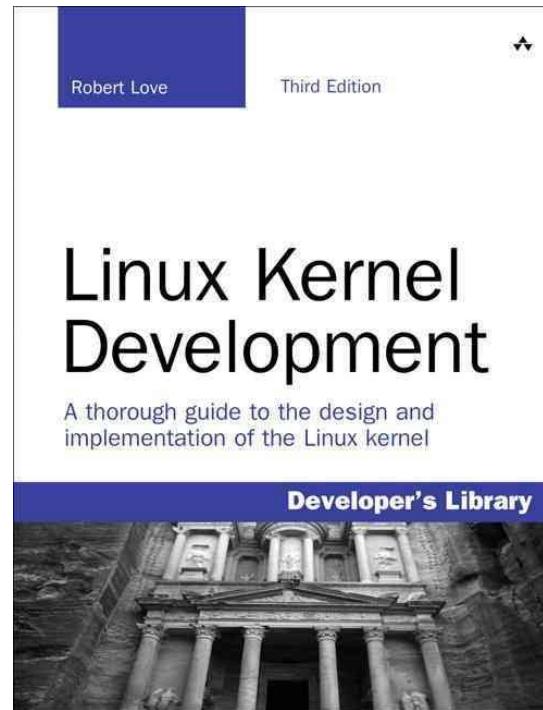
- Anyone wants to work on the above problems
- Anyone cares about what's going on under the hood
- Anyone has to build high-performance systems
- Anyone needs to diagnose bugs or security problems

Prerequisite

- Undergraduate Operating Systems (CSE 306)
- C programming
- Linux command line

Text book

- Robert Love, Linux Kernel Development, Addison-Wesley



Other useful sources

- [Understanding the Linux Kernel, O'Reilly Media](#)
- [Professional Linux Kernel Architecture, Wrox](#)
- [Linux Device Drivers, O'Reilly Media](#)
- [Understanding Linux Network Internals, O'Reilly Media](#)
- [Operating Systems: Three Easy Pieces](#)
- [Intel 64 and IA-32 Architectures Software Developer Manuals](#)

Communication

- [Zoom]
 - The meeting link can be found at Blackboard
 - Please do not distribute the zoom link.
 - All classes will be recorded in the cloud (via Blackboard).
 - Feel free to unmute yourself and ask questions as needed.
- [Course website](#)
 - Syllabus, schedule, etc.

Communication

- Blackboard
 - Primary way materials are *distributed*.
 - Lecture slides
 - Programming assignment submission
 - Grades posted
- Piazza
 - Please sign up
 - Use it to ask and answer questions (anonymous options available)

Office hours

- Dongyoon Lee
 - Tues and Thurs 9:30-10:30 AM (after classes)
 - Another Zoom meeting link can be found at Blackboard
 - Waiting Room. No Recording.
- GTA: Sergey Madaminov smadaminov@cs.stonybrook.edu
 - by appointment
- GTA: Ajay Paddayuru Shreepathi apaddayurush@cs.stonybrook.edu
 - by appointment

Grading policy (subject to change)

- Projects (75%)
 - P1. system call (10%)
 - P2. kernel data structure (10%)
 - P3. distributed shared memory (15%)
 - P4. cpu profiler (25%)
 - P5. file system (15%)
- Final exam (25%)
 - Online exam (via Blackboard, Honorlock)

About projects

- All programming projects are individual assignments. You **may discuss** the assignment details, designs, debugging techniques, or anything else with anyone you like in general terms, but you **may not provide, receive, or take code to or from anyone**. The code you submit must be your own work and only your own work. Any evidence that source code has been copied, shared, or transmitted in any way will be regarded as evidence of academic dishonesty.
- Each student should prepare a Linux virtual machine (Details will follow).

Submission policies (subject to change)

- Late submissions: No late project work will be assigned a grade.
- Wrong submissions: (Trivial) submission errors (e.g., a missing file, a wrong patch) are subject to at least 25% penalty. Students should provide an evidence (e.g., last modified time stamp) that the original source codes have not been modified after the due date.

“Hybrid” section students (CSE 506-02, Class#55714)

- The students who are registered in the hybrid section should attend at least three classes from NCS 120 to meet the on-campus attendance requirement and to obtain a passing grade.
- Attend three classes at NCS 120 during the weeks of Feb 15th and Feb 22nd, take the screenshots of your attendance in Zoom, and email me three screenshots.

Academic Integrity

Each student must pursue his or her academic goals honestly and be personally accountable for all submitted work. Representing another person's work as your own is always wrong. Faculty are required to report any suspected instances of academic dishonesty to the Academic Judiciary. For more comprehensive information on academic integrity, including categories of academic dishonesty, please refer to the [Academic Integrity](#) website.

Student Accessibility Support Center

If you have a physical, psychological, medical or learning disability that may impact your course work, please contact [Student Accessibility Support Center](#), ECC (Educational Communications Center) Building, room 128, (631) 632-6748. They will determine with you what accommodations, if any, are necessary and appropriate. All information and documentation is confidential.

Critical Incident Management

Stony Brook University expects students to respect the rights, privileges, and property of other people. Faculty are required to report to the Office of Judicial Affairs any disruptive behavior that interrupts their ability to teach, compromises the safety of the learning environment, or inhibits students' ability to learn.

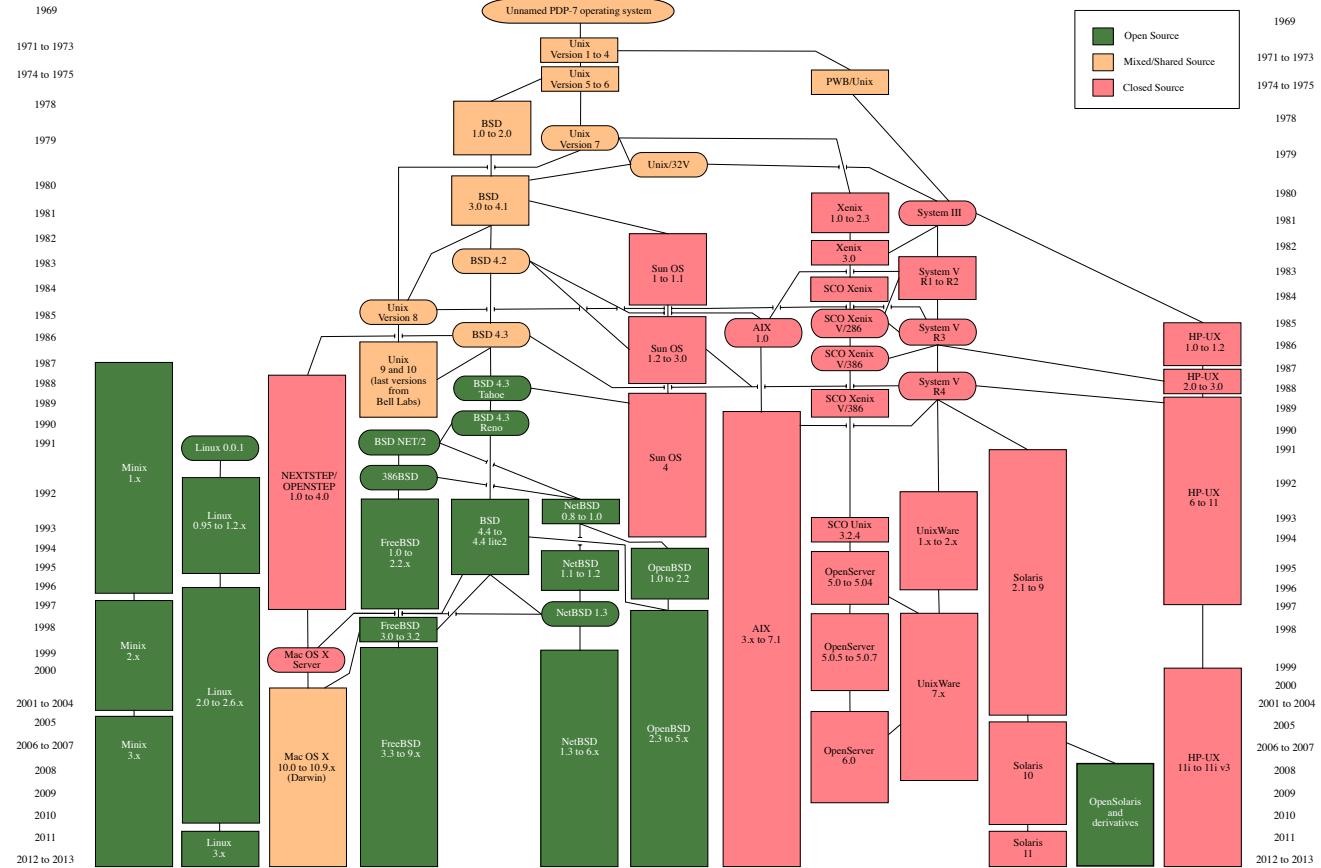
Acknowledgement

- This course reuses some of the material from:
 - VT's ECE 5984 by Dr. [Min](#) (main source)
 - SBU's CSE 506 by Drs. [Zadok](#), and [Ferdman](#)
 - GT's [CS 3210](#)
 - UW's [CSE 451](#) and [OSPP](#)
 - MIT's [6.828](#)

Today's agenda

- The history of Linux
- Linux open source model and community
- High level overview of the Linux kernel

History of UNIX ([Wikipedia](#))



Beginning of Linux

From: torvalds@klaava.Helsinki.FI (Linus Benedict Torvalds)
Newsgroups: comp.os.minix
Subject: What would you like to see most in minix?
Summary: small poll for my new operating system
Message-ID: <1991Aug25.205708.9541@klaava.Helsinki.FI>
Date: 25 Aug 91 20:57:08 GMT
Organization: University of Helsinki

Hello everybody out there using minix -

I'm doing a (free) operating system (just a hobby, won't be big and professional like gnu) for 386(486) AT clones. This has been brewing since april, and is starting to get ready. I'd like any feedback on things people like/dislike in minix, as my OS resembles it somewhat (same physical layout of the file-system (due to practical reasons) among other things).

I've currently ported bash(1.08) and gcc(1.40), and things seem to work. This implies that I'll get something practical within a few months, and Id like to know what features most people would want. Any suggestions are welcome, but I won't promise I'll implement them 😊

Linus (torvalds@kruuna.helsinki.fi)

PS. Yes - it's free of any minix code, and it has a multi-threaded fs. It is NOT protable (uses 386 task switching etc), and it probably never will support anything other than AT-harddisks, as that's all I have :-(.

Linux History

- 1991: First apparition, author: Linus Torvalds
- 1992: GPL License, first Linux distributions
- 1994: v1.0 - Single CPU for i386, then ported to Alpha, Sparc, MIPS
- 1996: v2.0 - Symmetric multiprocessing (SMP) support
- 1999: v2.2 - Big Kernel Lock removed
- 2001: v2.4 - USB, RAID, Bluetooth, etc.
- 2003: v2.6 - Physical Address Expansion (PAE), new architectures, etc.
- 2011: v3.0 - Incremental release of v2.6
- 2015: v4.0 - Livepatch → today's latest version: <http://www.kernel.org>

Linux open source model

- Linux is licensed under **GPLv2**

“

You may copy, distribute and modify the software as long as you track changes/dates in source files. Any modifications to or software including (via compiler) GPL-licensed code must also be made available under the GPL along with build & install instructions.

- Source code is freely available at <https://www.kernel.org/>
- Ref: [td;lrLegal, GPLv2](#)

Benefit of open source model

“

Given enough eyeballs, all bugs are shallow

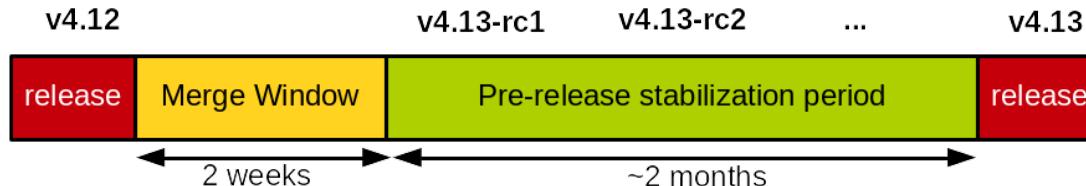
“

Given a large enough beta-test and co-developer base, almost every problem will be characterized quickly and the fix obvious to someone.

- **Linus's Law**
 - [The Cathedral & the Bazaar](#) by Eric S. Raymond
 - Security, stability, quality, speed of innovation, education, research, etc

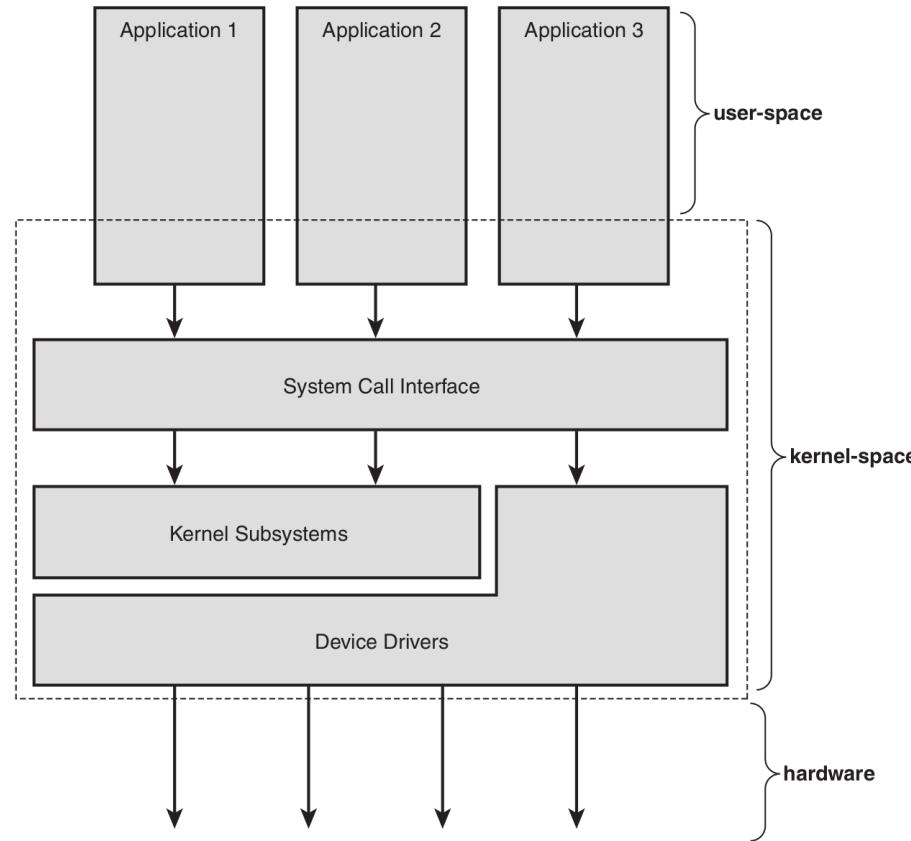
Kernel release cycle

- (major).(minor).(stable) → E.g., 5.8.3



- Prepatch or “RC” kernel release → for testing before the mainline release
- Mainline release → maintained by Linus with all new features
- Stable release → additional bug fixes after the mainline kernel release
- Long term support (LTS) for a subset of releases → e.g., 4.19, 4.14, 4.9

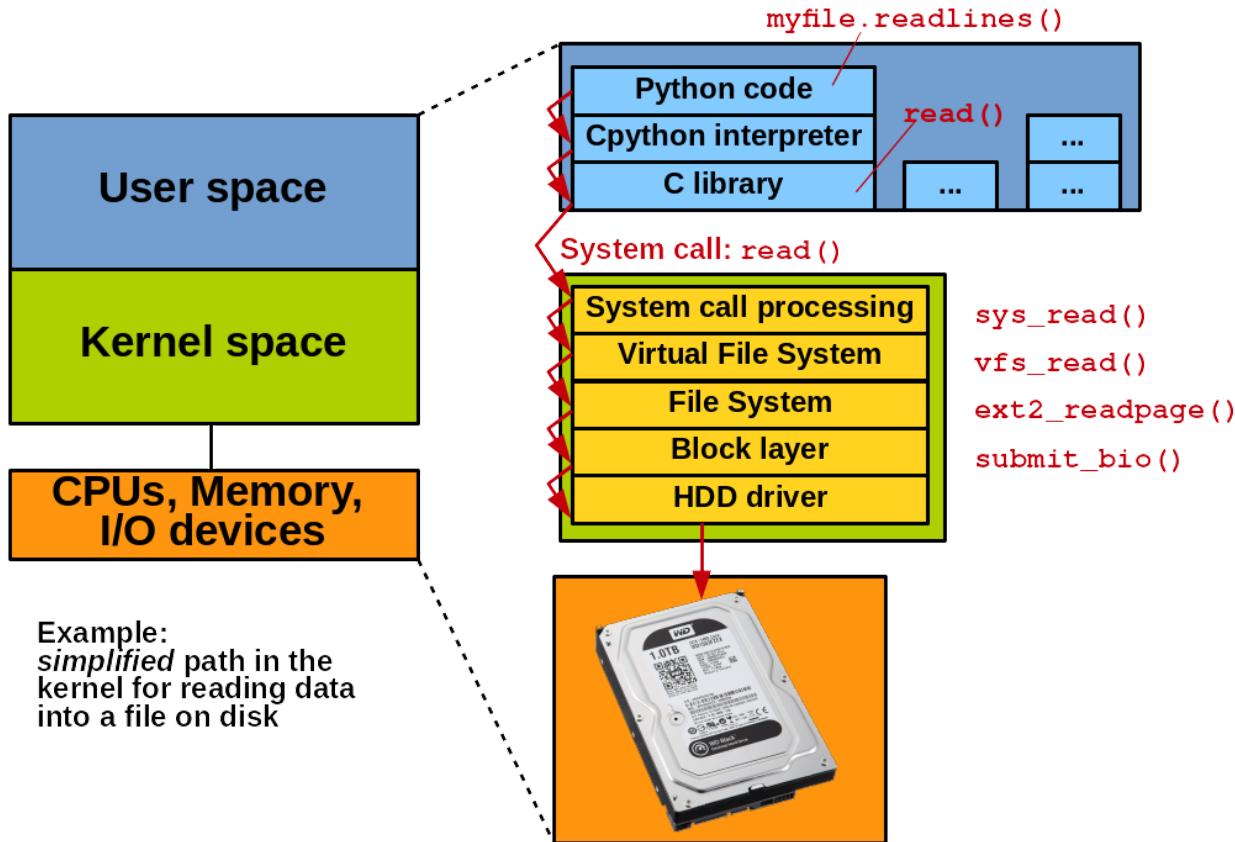
Overview of operating systems



User space vs. kernel space

- A CPU is executing in either of **user space** or in **kernel space**
- Only the kernel is allowed to perform **privileged operations** such as controlling CPU and IO devices
 - E.g., protection ring in x86 architecture
 - ring 3: user-space application
 - ring 0: operating system kernel
- An user-space application talks to the kernel space through **system call** interface
 - E.g., `open()`, `read()`, `write()`, `close()`

User space vs. kernel space



Linux is a *monolithic kernel*

- A traditional design: all of the OS runs in kernel, privileged mode
 - share the same address space
- Kernel interface ~= system call interface
- Good: easy for subsystems to cooperate
 - one cache shared by file system and virtual memory
- Bad: interactions are complex
 - leads to bugs, no isolation within kernel

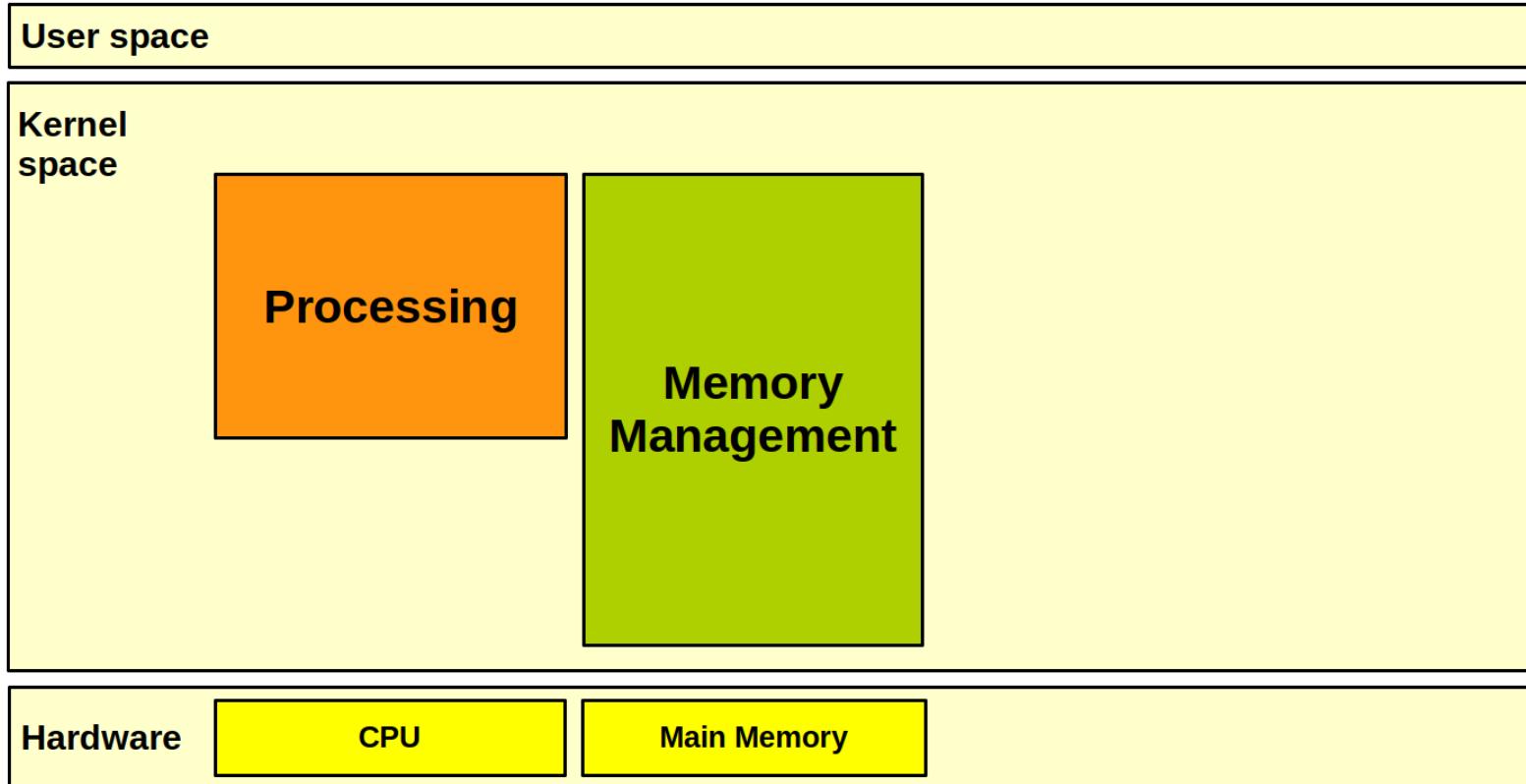
Alternative: *microkernel design*

- Many OS services run as ordinary user programs
 - e.g., file system in a file server
- Kernel implements minimal mechanism to run services in user space
 - IPC, virtual memory, threads
- Kernel interface != system call interface
 - applications talk to servers via IPCs
- Good: more isolation
- Bad: IPCs may be slow

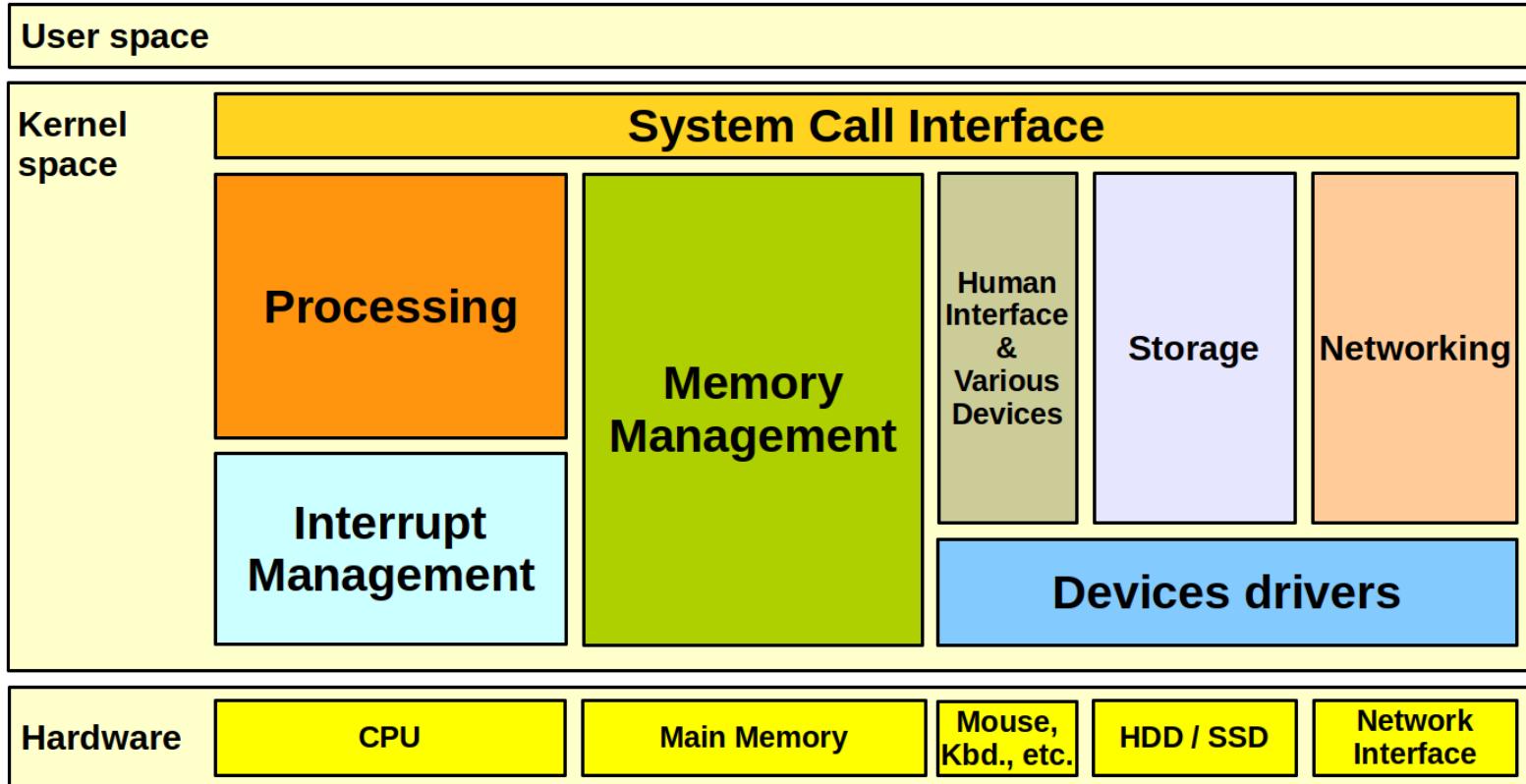
Debate

- Tanenbaum-Torvalds debate
- Most real-world kernels are mixed: Linux, OS X, Windows
 - e.g., X Window System

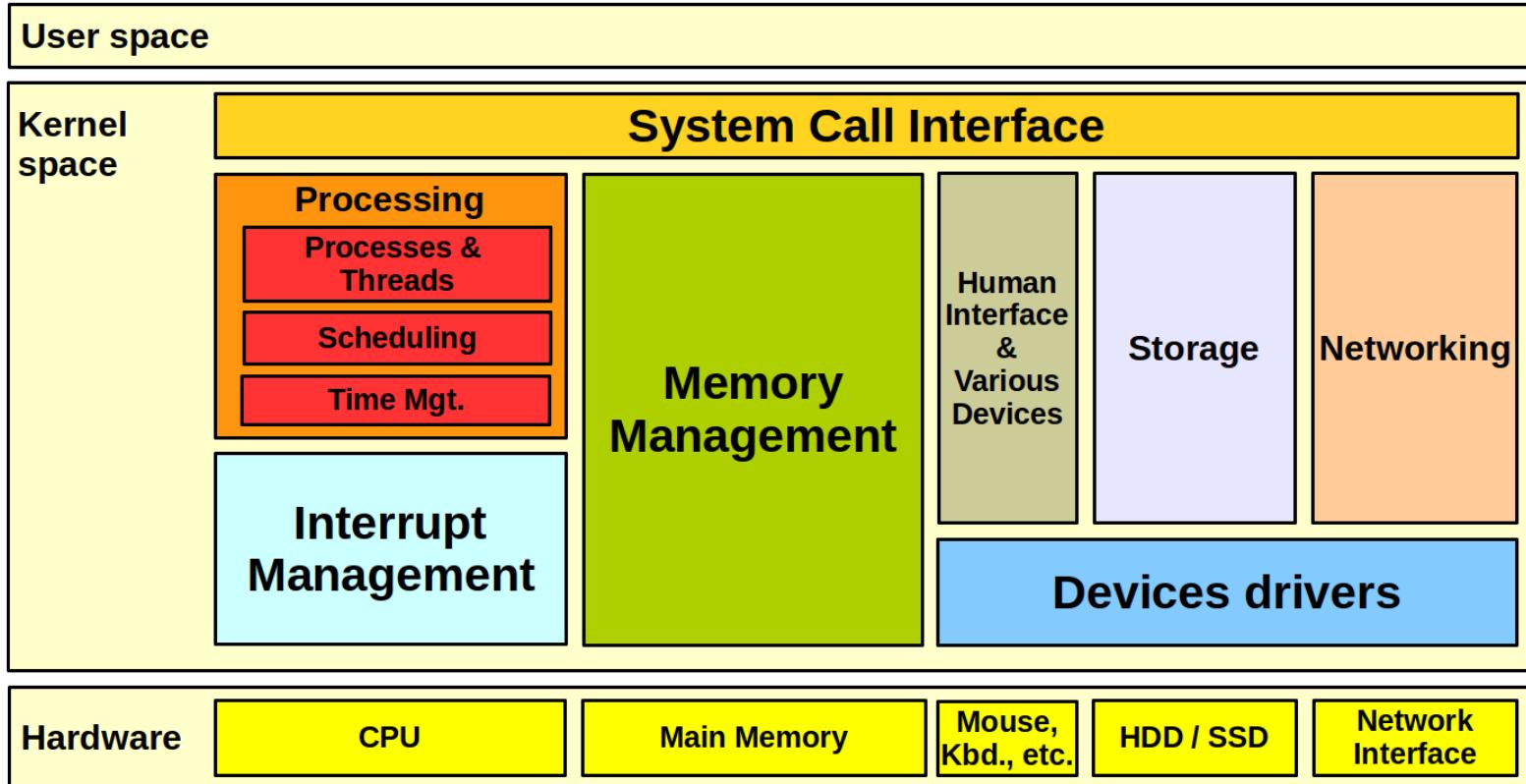
Kernel & course map



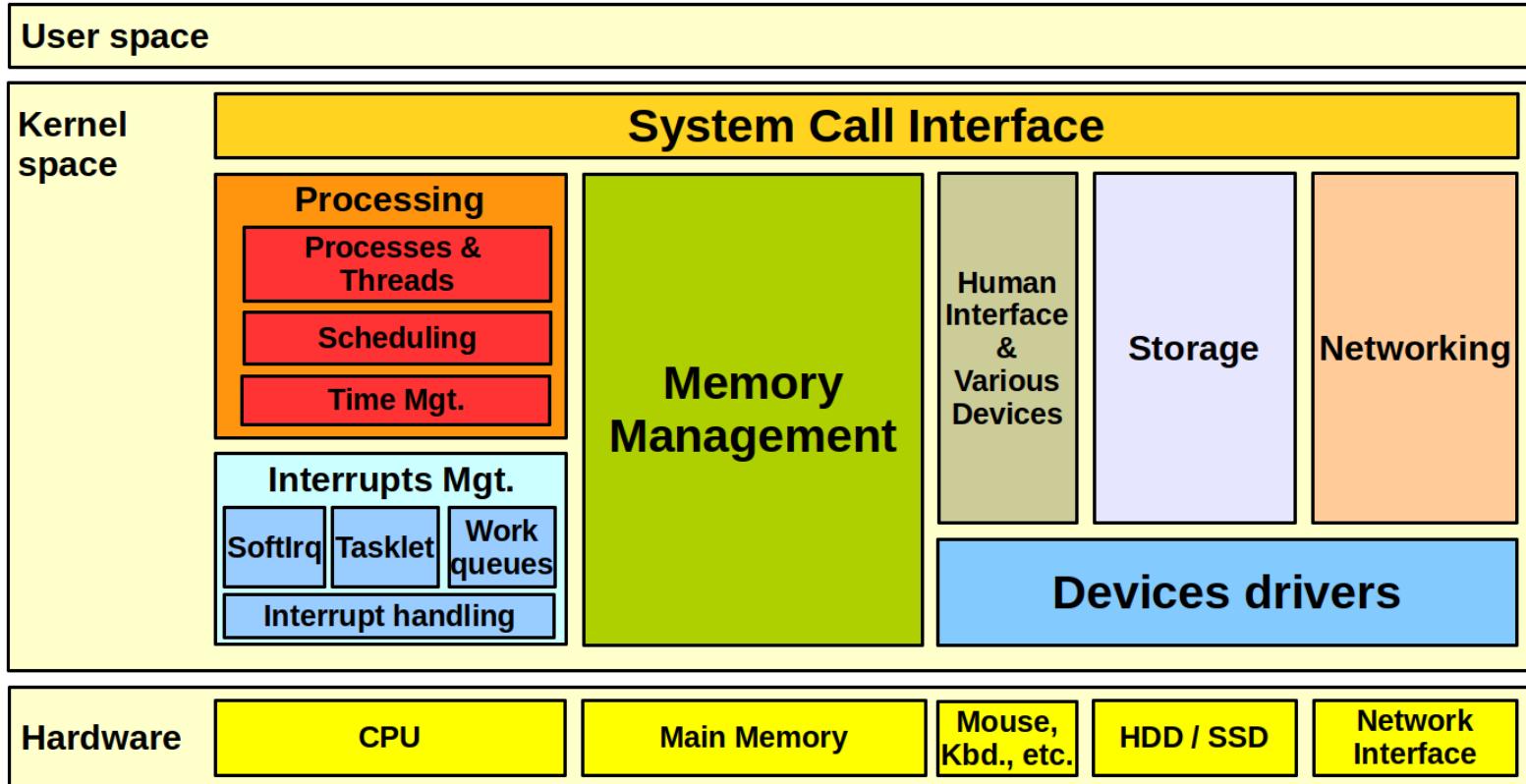
Kernel & course map



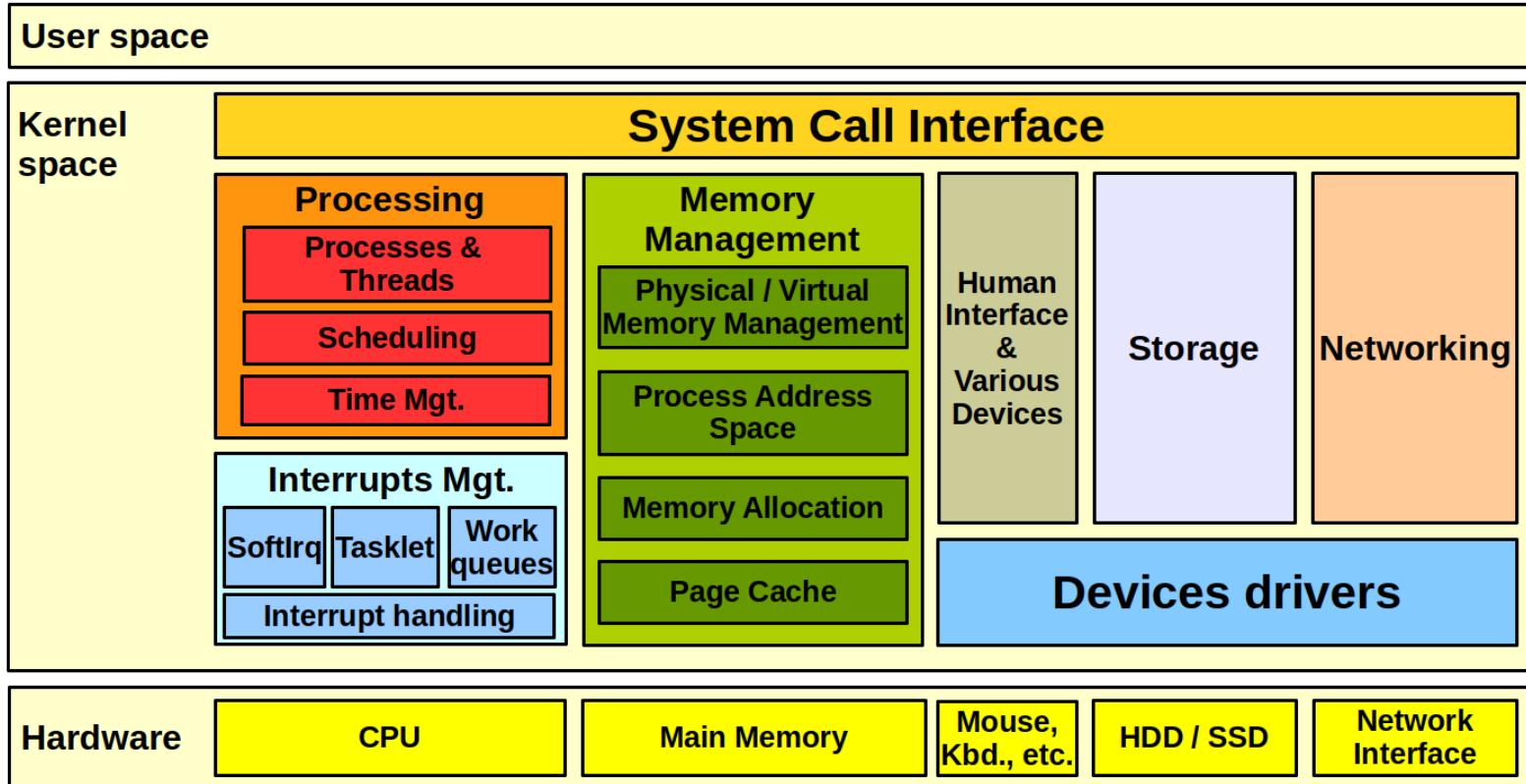
Kernel & course map



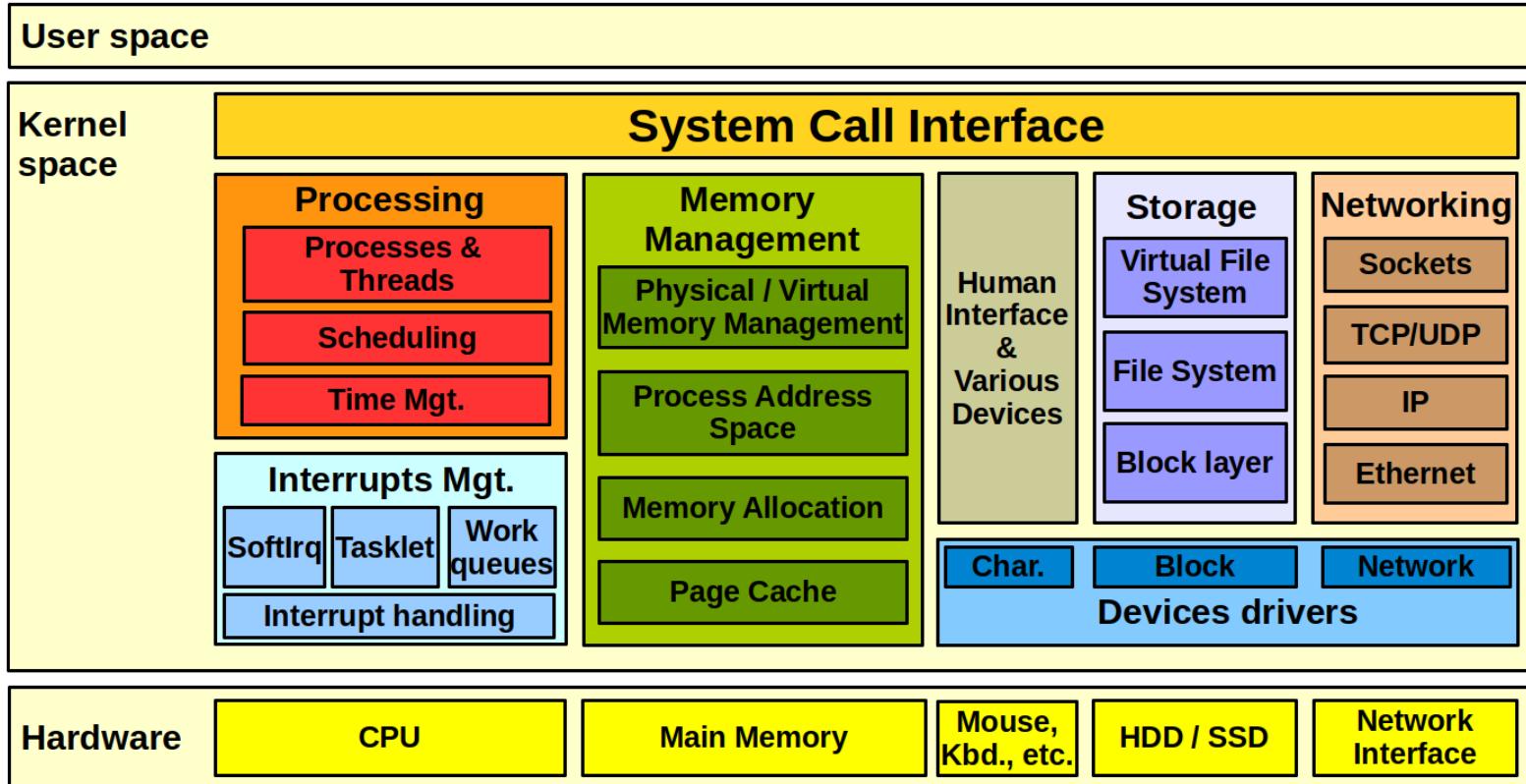
Kernel & course map



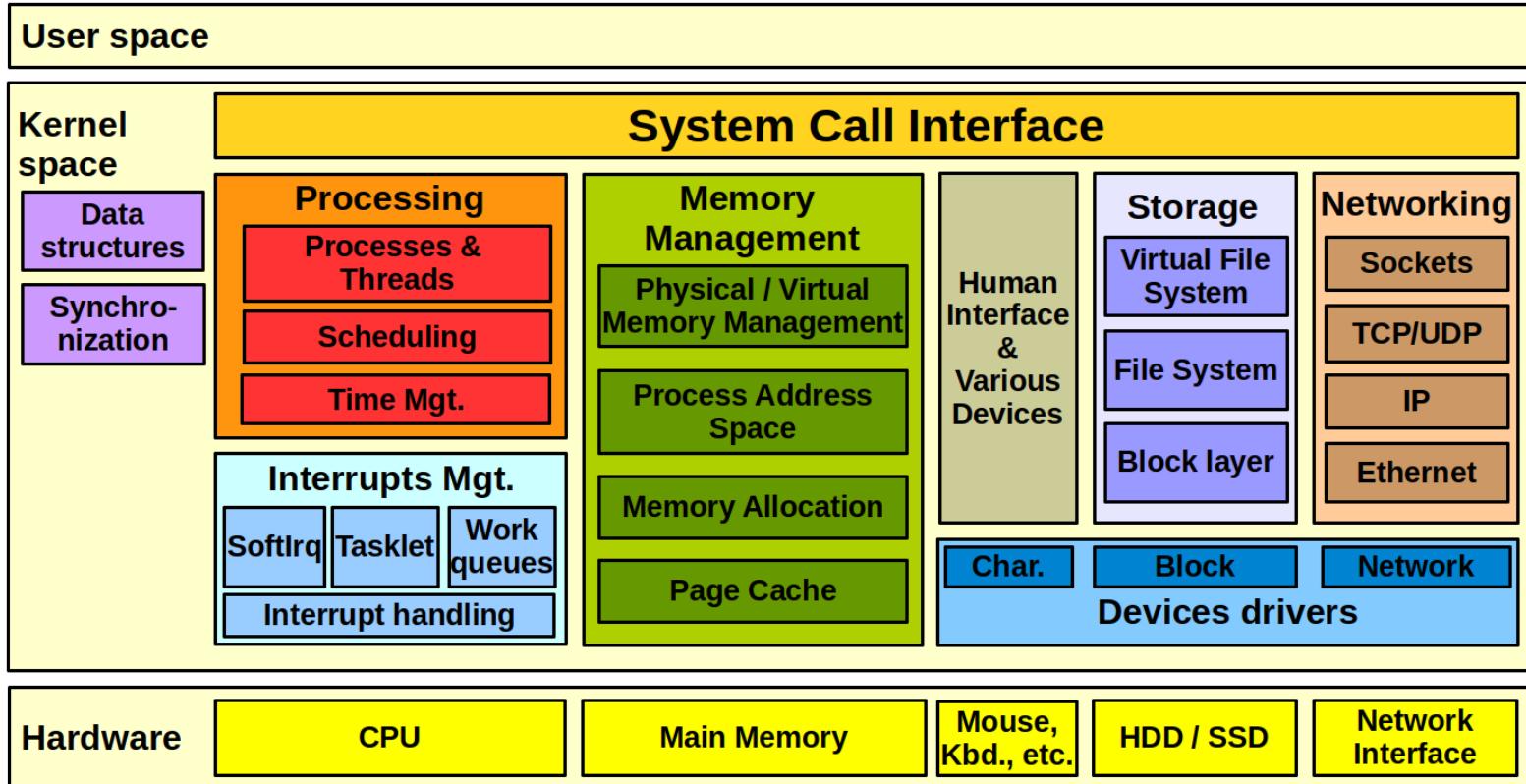
Kernel & course map



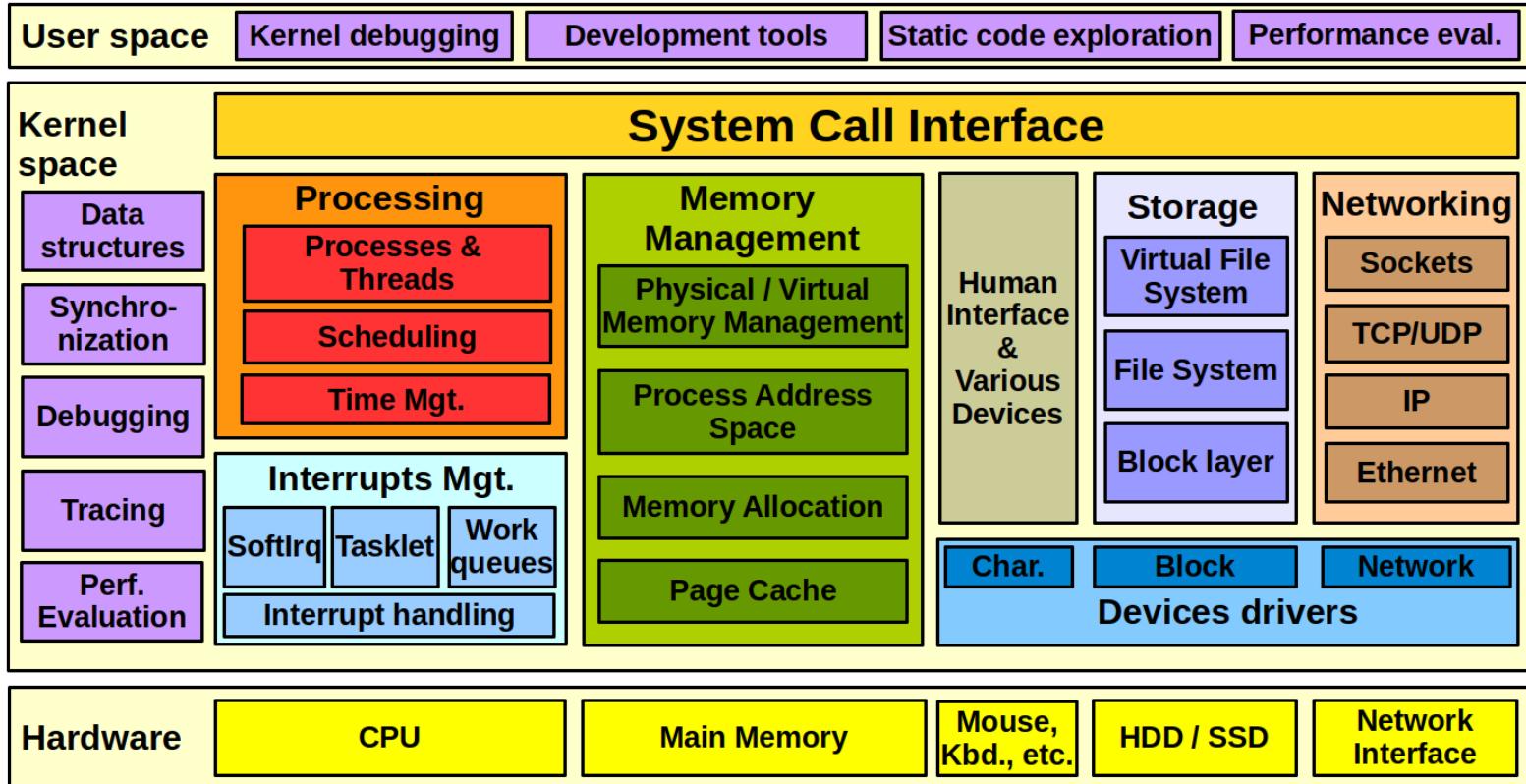
Kernel & course map



Kernel & course map



Kernel & course map



Kernel & course map

- Let's check [course schedule](#)
 - Will be further updated

Set up course environment

- VirtualBox to run Linux VM
 - Recommended setting
 - disk >= 64GB, RAM >= 4GB, # CPU >= 2
 - Add port forwarding rule
 - protocol: TCP, host IP: 127.0.0.1
 - host port: 2222, guest port: 22
 - Use **Shared folders** for file sharing between Linux VM and your host

Set up course environment

- Ubuntu 20.04.1 LTS Server for Linux distribution
 - Recommended disk space: 64 GB or more
 - Set up root password and create your user account
 - After login as a root user, add your account to `sudoers`
- SSH client on your laptop
 - Check whether you can `ssh` from host
 - `ssh -p 2222 {username}@localhost`
- Linux kernel: `v5.8` released at August 2nd, 2020

Next actions

- Finish to set up course environment
- If you are not familiar with Linux commands, learn followings:
 - `vim`, `ssh`, `scp`, `tmux`, `git`, and [more](#)
 - Check this awesome online lectures: [The Missing Semester of Your CS Education](#)
- Download the latest Linux kernel source inside your Linux VM

```
$ git clone https://github.com/torvalds/linux.git
```

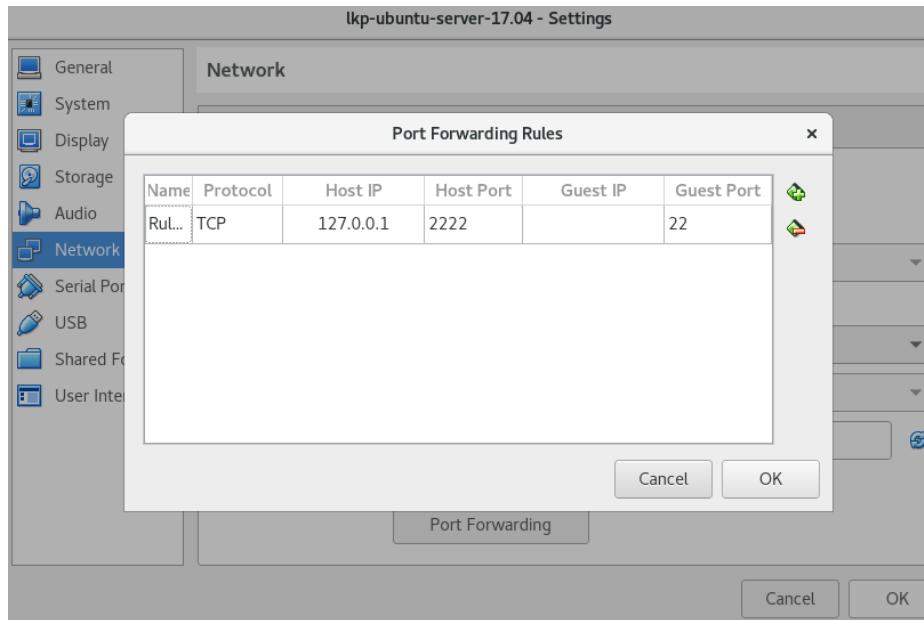
Next lecture

- Building and exploring Linux kernel

Building and exploring Linux kernel

Dongyoон Lee

Did you succeed in installing Linux on VirtualBox?



- Add a port forwarding rule on VirtualBox

Why software tools are important?

- Linux source code is huge and evolves very fast
 - 27 million lines of code (LoC) ← 1,600 developers / release

```
✓ ~/workspace/research/linux [v5.8]
$ tree .
├── arch
│   ├── alpha
│   │   ├── boot
│   │   │   ├── bootloader.lds
│   │   │   └── bootp.c
│   ...
└── lib
    ├── irqbypass.c
    ├── Kconfig
    └── Makefile
    └── Makefile

7468 directories, 75689 files
```

Today's lecture

- Tools
 - Version control : `git` , `tig`
 - Configure, build, and install the kernel : `make`
 - Explore the code : `cscope` , `ctags`
 - Editor : `vim` , `emacs`
 - Screen : `tmux`
- Kernel vs. user programming

Obtaining the kernel source code

- Tar ball
 - <https://www.kernel.org/>
- Linus's git repository
 - <git://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git>
- Github mirror of Linus's git repository
 - <https://github.com/torvalds/linux.git>
- Let's explore above web sites!

Version control: `git`

- **Git** is a version control software
 - tracking changes in computer files
- Initially developed by Linus Torvalds for development of the Linux kernel
 - Extensively used in many other software development
 - Github <https://github.com/> is a `git` service provider
- Distributed revision control system
 - Every git directory on every computer is a full-fledged repository with complete history

Essential `git` commands

```
$ # 1. install and configure
$ sudo dnf install git # sudo apt-get install git
$ git config --global user.name "John Doe" # set your name and email for history
$ git config --global user.email johndoe@example.com

$ # 2. create a repository
$ git init                      # create a new local repo
$ git clone https://github.com/torvalds/linux.git # clone an existing repo

$ # 3. tags
$ git tag             # list all existing tags
$ git checkout v5.8   # checkout the tagged version

$ # 4. commit history (or use tig for prettier output)
$ git log            # show all commit history
$ git log <file>     # show changes over time for a file
$ git blame <file>   # who changed what and when in <file>
```

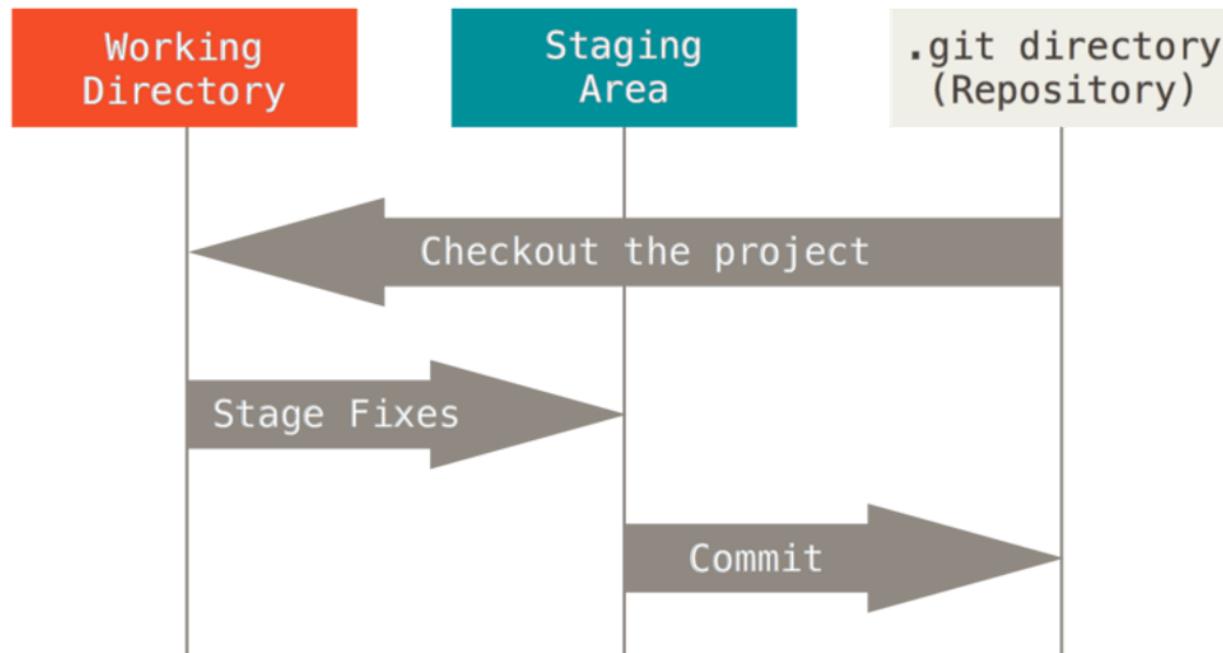
Essential **git** commands

```
$ # 5. local changes
$ git status          # show changed files
$ git diff            # show changed lines
$ git add <file>      # add <file> to the next commit
$ git commit          # commit previously staged files to my local repo

$ # 6. publish and update
$ git push            # publish a committed local changes to a remote repo
$ git pull            # update a local repo
```

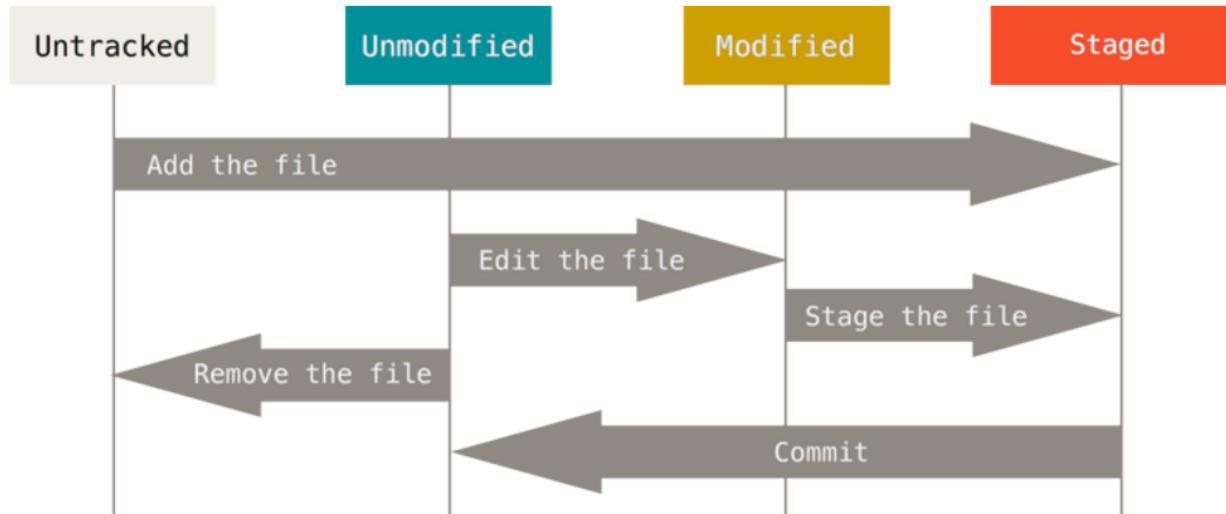
- Many useful git tutorials:
 - [Atlassian](#), [Github](#), [TutorialsPoint](#), [Linux kernel](#), [Pro Git](#)

git workflow



- Source: [Pro Git](#)

git workflow



- Source: [Pro Git](#)

The kernel source tree

```
$ git clone https://github.com/torvalds/linux.git # clone the kernel repo
$ cd linux; git checkout v5.8    # checkout v5.8
$  
$ tree -d -L 2      # list top two-level directories
├── arch          # * architecture dependent code
│   ├── arm        #   - ARM architecture
│   └── x86        #   - Intel/AMD x86 architecture
├── block         # * block layer: e.g., IO scheduler
├── Documentation # * design documents
├── drivers        # * device drivers
│   └── nvme       #   - NVMe SSD
├── fs            # * virtual file system (VFS)
│   ├── ext4        #   - ext4 file system
│   └── xfs         #   - XFS file system
├── include        # * include files
│   ├── linux       #   - include files for kernel
│   └── uapi         #   - include files for user-space tools
├── init          # * bootig: start_kernel() at main.c
└── ipc           # * IPC: e.g., semaphore
...  
...
```

The kernel source tree

```
...  
├── kernel      # * core features of the kernel  
│   ├── locking  # - locking: e.g., semaphore, mutex, spinlock  
│   └── sched     # - task scheduler  
└── lib         # * common library: e.g., red-black tree  
└── mm          # * memory management: e.g., memory allocation, paging  
└── net         # * network stack  
    ├── ipv4      # - TCP/IPv4  
    └── ipv6      # - TCP/IPv6  
└── security    # * security framework  
    └── selinux   # - selinux  
└── tools        # * user-space tools  
    └── perf      # - perf: performance profiling tool  
└── virt         # * virtualization  
    └── kvm       # - KVM type-2 hypervisor
```

615 directories

Build the kernel

1. Configuring the kernel

- Configuration file defining compilation options (~ 3700 for x86)

2. Compiling the kernel

- Compile and link the kernel source code

3. Installing the new kernel

- Install compiled new kernel image to a system
- `make help` to see other make options
- Ref: [Documentation/admin-guide/README.rst](#)

Configure the kernel

- `make menuconfig`
 - Need *libncurses*
 - `sudo dnf install ncurses-devel` # Fedora/CentOS/RedHat
 - `sudo apt-get install libncurses5-dev` # Debian/Ubuntu

Configure the kernel

- `make defconfig`
 - Generate the default configuration of running platform
 - `linux/arch/x86/configs/x86_64_defconfig`
- `make oldconfig`
 - Use the configuration file of running kernel
 - Will ask about new configurations
 - If you are not sure, choose default options
- `make localmodconfig`
 - Update current config disabling modules not loaded

Kernel configuration file: `.config`

- `.config` file is at the root of the kernel source
 - preprocessor flags in the source code

```
# linux/.config
# CONFIG_XEN_PV is not set
CONFIG_KVM_GUEST=y
CONFIG_XFS_FS=m

/* linux/arch/x86/kernel/cpu/hypervisor.c */
static const __initconst struct hypervisor_x86 * const hypervisors[] =
{
#ifndef CONFIG_XEN_PV
    &x86_hyper_xen_pv,
#endif
#ifndef CONFIG_KVM_GUEST
    &x86_hyper_kvm,
#endif
};
```

Compile the kernel

1. Compile the kernel: `make`

- Compile the kernel source code
- Compiled kernel image: `linux/arch.x86/boot/bzImage`

2. Compile modules: `make modules`

• Parallel `make`

- `make <target> -j<number of CPUs to use>`
- E.g., `make -j4`

Install the new kernel

```
# Install the new kernel modules (if you change modules)
$ sudo make modules_install
$ ls /lib/modules/

# Install the new kernel image
$ sudo make install
$ ls /boot/*5.8*
/boot/config-5.8.0      /boot/initrd.img-5.8.0
/boot/System.map-5.8.0   /boot/vmlinuz-5.8.0

# Reboot the machine
$ sudo reboot

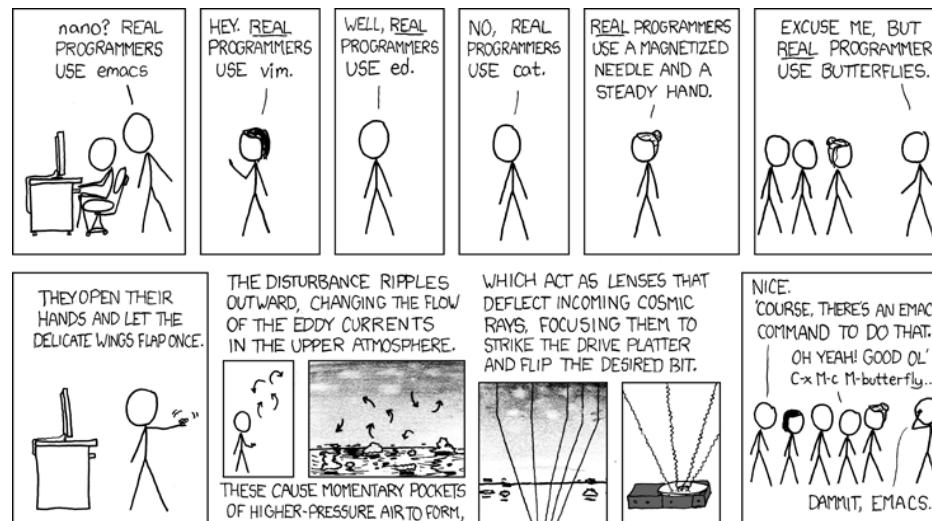
# Check if a system boots with the new kernel
$ uname -a
Linux dongyoон 5.8.0 #1 SMP Mon Jan 25 22:56:41 UTC 2021 x86_64 x86_64 x86_64 GNU/Linux

# See kernel log
$ dmesg
$ dmesg -w # wait for new kernel messages
```

Editor

- There are many good editors

- vim , emacs



- Source: <https://xkcd.com/378/>

Exploring the code

- [Linux Cross Reference \(LXR\)](#)
- `cscope`
- `vim` with `cscope` or `ctags`
- `emacs` with `cscope`
- ...

Linux Cross Reference (LXR)

- Code indexing tool with a web interface
 - Don't install it! One instance is running here:
 - <http://lxr.free-electrons.com/>
- Allows to:
 - Browse the code of different Linux versions
 - Search for identifiers (functions, variables, etc.)
 - Quickly lookup a function declaration/definition

Linux Cross Reference (LXR)

Linux Cross Reference

Free Electrons
Embedded Linux Experts

• Source Navigation • Identifier Search • Freetext Search •

Version: 2.0.40 2.2.26 2.4.37 3.12 3.13 3.14 3.15 3.16 3.17 3.18 3.19 4.0 4.1 4.2 4.3 4.4 4.5 4.6 4.7 4.8 4.9

[Linux/fs/](#)

Parent directory

- 9p/
- adfs/
- affs/
- afs/
- autofs4/
- befs/
- bfs/
- btrfs/
- cachefiles/
- ceph/
- cifs/
- coda/
- configs/
- cramfs/
- debuofs/

Linux Cross Reference (LXR)

Linux Cross Reference (LXR)

```
533             static_command_line, _start_param,
534             _stop_param - _start_param,
535             -1, -1, &unknown_bootoption);
536     if (!IS_ERR_OR_NULL(after_dashes))
537         parse_args("Setting init args", after_dashes, NULL, 0, -1, -1,
538                     set_init_arg);
539
540     jump_label_init();
541
542     /*
543      * These use large bootmem allocations and must precede
544      * kmem_cache_init()
545      */
546     setup_log_buf(0);
547     pidhash_init();
548     vfs_caches_init_early();
549     sort_main_extable();
550     trap_init();
551     mm_init();
552
553     /*
554      * Set up the scheduler prior starting any interrupts (such as the
555      * timer interrupt). Full topology setup happens at smp_init()
556      * time - but meanwhile we still have a functioning scheduler.
557      */
558     sched_init();
559
560     /*
561      * Disable preemption - early bootup scheduling is extremely
562      * fragile until we cpu_idle() for the first time.
563      */
564     preempt_disable();
565     if (WARN(!irqs_disabled(),
566             "Interrupts were enabled *very* early, fixing it\n"))
567         local_irq_disable();
568     idr_init_cache();
569     rcu_init();
570
571     /* trace_printk() and trace points may be used after this */
572     trace_init();
573
574     context_tracking_init();
575     radix_tree_init();
576     /* init some links before init_ISA_irqs() */
```

Click on a
function call to
search
for the function

Linux Cross Reference (LXR)

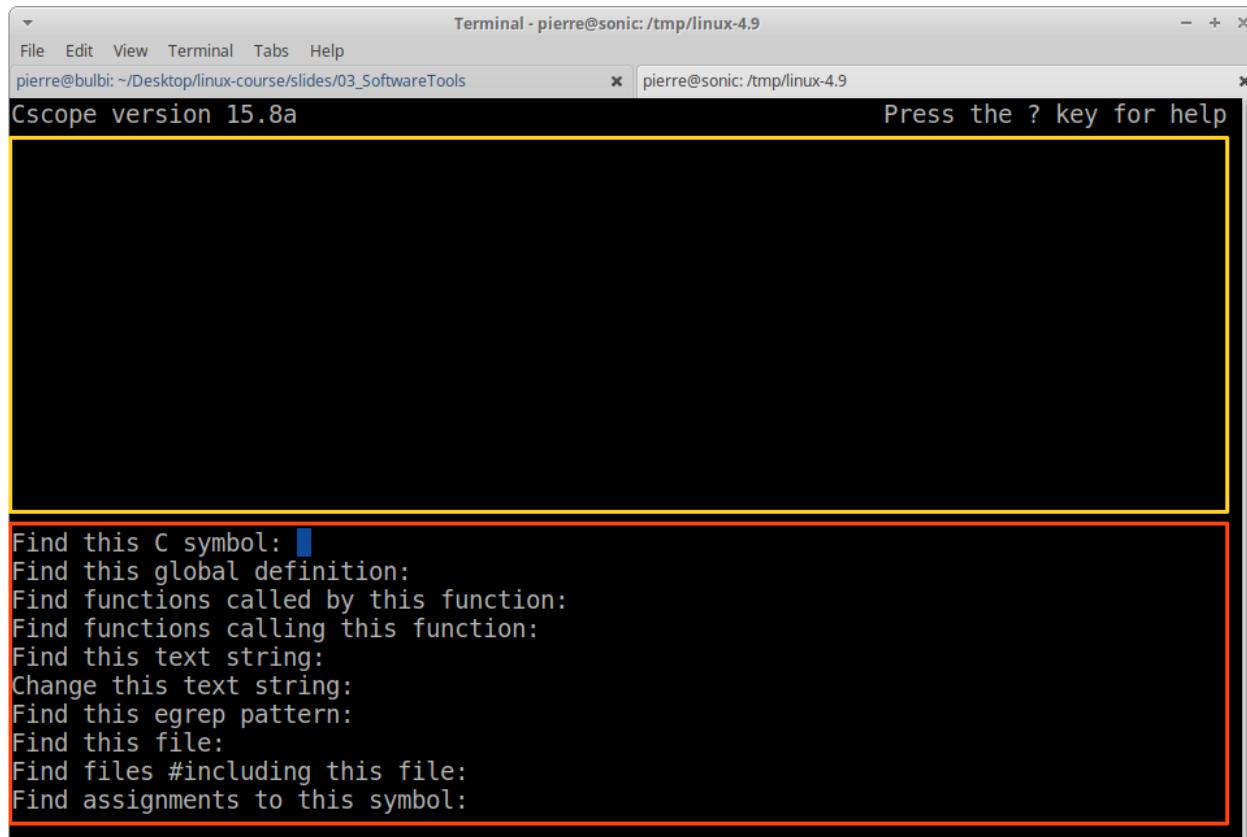
cscope

- Command line tool to browse (potentially large) C codebases
- Installation: `sudo {apt-get|dnf} install cscope`
- Build cscope database
 - `cd linux; KBUILD_ABS_SRCTREE=1 ARCH=x86 make cscope # only for x86`
 - Need to rebuild after code changes
- Although `cscope -R` is the common way to build cscope database, `make cscope` is optimized for the kernel source code

cscope

- Search for:
 - C identifier occurrences (variable name, function name, typedef/struct, label)
 - Functions/variables definitions
 - Functions called by/calling function f
 - Text string
- Terminating cscope: Ctrl-d

cscope



cscope

```
Terminal - pierre@sonic:/tmp/linux-4.9
File Edit View Terminal Tabs Help
pierre@bulbi:~/Desktop/linux-course/slides/03_SoftwareTools  × pierre@sonic: /tmp/linux-4.9
Cscope version 15.8a                                         Press the ? key for help

Find this C symbol: spin_lock
Find this global definition:
Find functions called by this function:
Find functions calling this function:
Find this text string:
Change this text string:
Find this egrep pattern:
Find this file:
Find files #including this file:
Find assignments to this symbol:
```

cscope

```
Terminal - pierre@sonic:/tmp/linux-4.9
File Edit View Terminal Tabs Help
pierre@bulbi: ~/Desktop/linux-course/slides/03_SoftwareTools  x  pierre@sonic: /tmp/linux-4.9  x
C symbol: spin_lock

      File          Function          Line
0 platsmp.c      <global>        287 spin_lock(&boot_lock);
1 bus.c          <global>        1049 spin_lock(&device_klis
                           t->k_lock);
2 platform.c     <global>        697 spin_lock(&drv->driver
                           .bus->p->klist_drivers
                           .k_lock);
3 runtime.c       <global>        279 spin_lock(&dev->power.
                           lock);
4 omap_gem.c      <global>        1256 spin_lock(&sync_lock);

* Lines 1-6 of 9743, 9738 more - press the space bar to display more *

Find this C symbol:
Find this global definition:
Find functions called by this function:
Find functions calling this function:
Find this text string:
Change this text string:
Find this egrep pattern:
Find this file:
Find files #including this file:
Find assignments to this symbol:
```

cscope

```
Terminal - pierre@sonic:/tmp/linux-4.9
File Edit View Terminal Tabs Help
pierre@bulbi:~/Desktop/linux-course/slides/03_SoftwareTools  × pierre@sonic: /tmp/linux-4.9
Cscope version 15.8a                                         Press the ? key for help

Find this C symbol:
Find this global definition: spin_lock
Find functions called by this function:
Find functions calling this function:
Find this text string:
Change this text string:
Find this egrep pattern:
Find this file:
Find files #including this file:
Find assignments to this symbol:
```

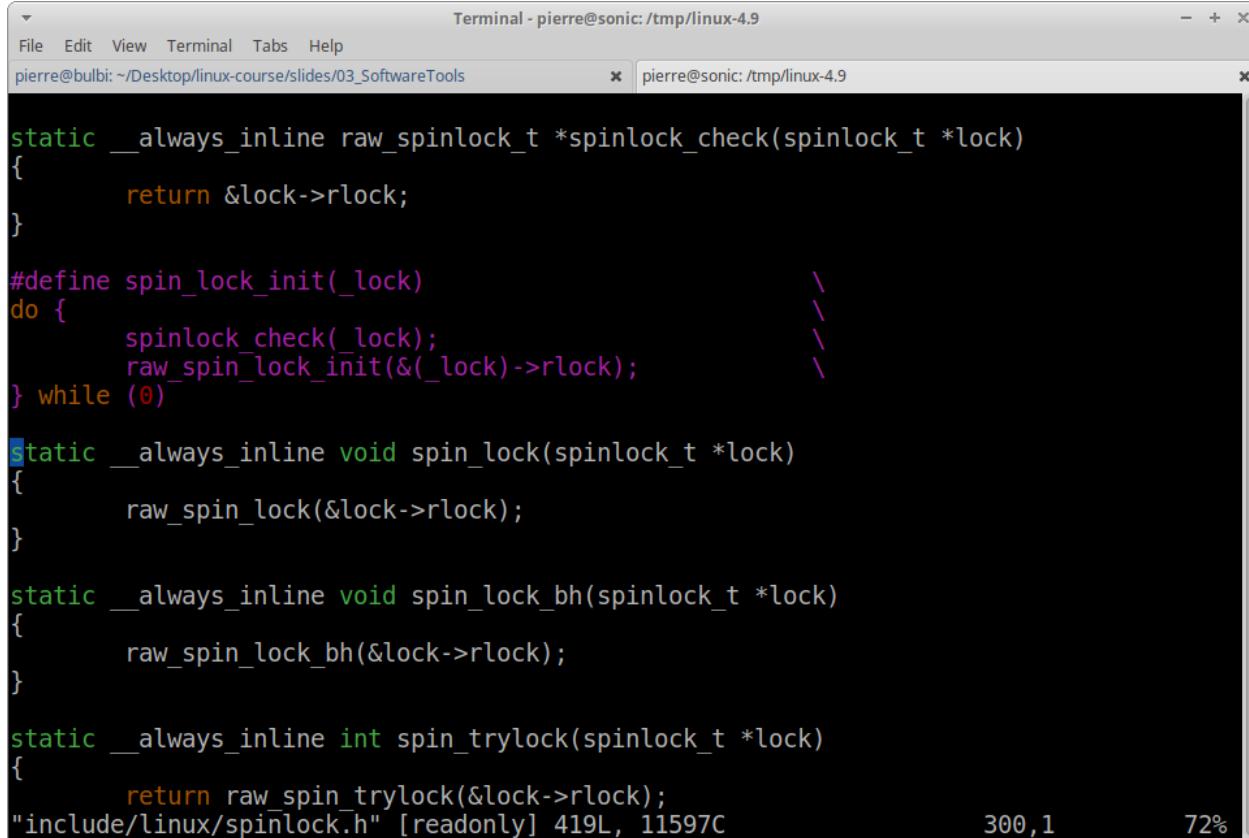
cscope

```
Terminal - pierre@sonic:/tmp/linux-4.9
File Edit View Terminal Tabs Help
pierre@bulbi:~/Desktop/linux-course/slides/03_SoftwareTools  × pierre@sonic: /tmp/linux-4.9  ×
Global definition: spin_lock

      File          Line
0 aic79xx_osm.h 352 spinlock_t spin_lock;
1 aic7xxx_osm.h 356 spinlock_t spin_lock;
2 comedidev.h    177 spinlock_t spin_lock;
3 spinlock.h     300 static  always inline void spin_lock(spinlock_t *lock)

Find this C symbol:
Find this global definition:
Find functions called by this function:
Find functions calling this function:
Find this text string:
Change this text string:
Find this egrep pattern:
Find this file:
Find files #including this file:
Find assignments to this symbol:
```

cscope



The screenshot shows a terminal window titled "Terminal - pierre@sonic:/tmp/linux-4.9". The window has two tabs: "pierre@bulbi: ~/Desktop/linux-course/slides/03_SoftwareTools" and "pierre@sonic: /tmp/linux-4.9". The content of the terminal is as follows:

```
static __always_inline raw_spinlock_t *spinlock_check(spinlock_t *lock)
{
    return &lock->rlock;
}

#define spin_lock_init(_lock)
do {
    spinlock_check(_lock);
    raw_spin_lock_init(&(_lock)->rlock);
} while (0)

static __always_inline void spin_lock(spinlock_t *lock)
{
    raw_spin_lock(&lock->rlock);
}

static __always_inline void spin_lock_bh(spinlock_t *lock)
{
    raw_spin_lock_bh(&lock->rlock);
}

static __always_inline int spin_trylock(spinlock_t *lock)
{
    return raw_spin_trylock(&lock->rlock);
}
"include/linux/spinlock.h" [readonly] 419L, 11597C
```

The status bar at the bottom right of the terminal window shows "300,1" and "72%".

vim with **cscope** or **ctags**

- **vim** can use the tag database of **cscope**, as well as **ctags**
 - `sudo apt-get install cscope exuberant-ctags`
 - `sudo dnf install cscope ctags`
- Generate the database
 - `cd linux; make cscope tags -j2`
- Launch vim
 - `vim init/main.c`

vim with **cscope** or **ctags**

- Search for function definition/variable declaration:
 - `:tag start_kernel` or `:cs find global start_kernel`
- Help for `ctags` and `cscope`
 - `:help tag` or `:help cs`
- Another way to find a function definition/variable declaration:
 - Put the cursor on the symbol name and press `Ctrl+]`
- To navigate back and forth between file:
 - `:bp` or `:bn`

Screen: tmux

- tmux is a tool to manage virtual consoles

The image shows two tmux sessions side-by-side. Both sessions have a title bar "changwoo::changwoo::changwoo".
Session 1 (Left): A terminal window displaying the Linux kernel source code for the main.c file. The code includes comments about the history of the kernel and defines the DEBUG macro. It also shows various #include directives for header files like <linux/types.h>, <linux/init.h>, and <linux/module.h>. The bottom of the window shows the command "When done with a buffer, type C-x #".
Session 2 (Right): A terminal window displaying the output of the command "grep -R CONFIG_KVM_GUEST /include/generated/autoconf.h". The results show multiple occurrences of the "#define CONFIG_KVM_GUEST 1" macro being defined across various kernel configuration headers, such as arch/x86/Kconfig and arch/mips/Kconfig. The session also shows the command "grep -R CONFIG_KVM_GUEST /include/generated/autoconf.h" again at the bottom.

- Ref: [A tmux Primer](#)

Essential tmux commands

- `tmux` : start a new tmux session
- `Ctrl-b %` : split a pane vertically
- `Ctrl-b "` : split a pane horizontally
- `Ctrl-b o` : move to the next pane
- `Ctrl-b z` : zoom (or unzoom) a pane
- `Ctrl-b c` : create a new window
- `Ctrl-b N` : go to window N (0~9)
- `Ctrl-b d` : detach from a session
- `tmux a` : attach to an existing session

Kernel vs. user programming

- No libc or standard headers
 - Instead the kernel implements lots of libc-like functions
- Examples
 - `#include <string.h>` → `#include <linux/string.h>`
 - `printf("Hello!")` → `printk(KERN_INFO "Hello!")`
 - `malloc(64)` → `kmalloc(64, GFP_KERNEL)`

Kernel vs. user programming

- Use GCC extensions
- Inline functions
 - `static inline void func()`
- Inline assembly: less than 2%
 - `asm volatile("rdtsc" : "=a" (l), "=d" (h));`
- Branch annotation: hint for better optimization
 - `if (unlikely(error)) {...}`
 - `if (likely(success)) {...}`

Kernel vs. user programming

- No (easy) use of floating point
- Small, fixed-size stack: 8 KB (2 pages) in x86
- No memory protection
 - **SIGSEGV → kernel panic (oops)**
- [An example of kernel oops](#)

Kernel vs. user programming

- Synchronization and concurrency
 - Multi-core processor → synchronization among tasks
 - A kernel code can execute on two or more processors
 - Preemptive multitasking → synchronization among tasks
 - A task can be scheduled and re-scheduled at any time
 - Interrupt → synchronization with interrupt handlers
 - Can occur in the midst of execution (e.g., accessing resource)
 - Need to synchronize with interrupt handler

Linux kernel coding style

- Indentation: 1 tab → 8-character width (not 8 spaces)
- No CamelCase use underscores: `SpinLock` → `spin_lock`
- Use C-style comments: `/* use this style */` // not this
- Line length: 80 column
- **Write code in a similar style with other kernel code**
- Ref: [Documentation/process/coding-style.rst](#)

Linux kernel coding style

```
/*
 * a multi-lines comment
 * (no C++ '//' !)
 */

struct foo {
    int member1;
    double member2;
}; /* no typedef ! */

#ifndef CONFIG_COOL_OPTION
int cool_function(void) {
    return 42;
}
#else
int cool_function(void) { }
#endif /* CONFIG_COOL_OPTION */
```

Linux kernel coding style

```
void my_function(int the_param, char *string, int a_long_parameter,
                 int another_long_parameter)
{
    int x = the_param % 42;

    if (!the_param)
        do_stuff();

    switch (x % 3) {
    case 0:
        do_some_stuff();
        cool_function();
        break;
    case 1:
        /* Fall through */
    default:
        do_other_stuff();
        cool_function();
    }
}
```

Summary of tools

- Version control : `git` , `tig`
- Configure the kernel : `make oldconfig`
- Build the kernel : `make -j8`; `make modules -j8`
- Install the kernel : `make install`; `make modules_install`
- Explore the code : `make cscope tags -j2`; `cscope` , `ctags`
- Editor : `vim` , `emacs`
- Screen : `tmux`

Other useful sources II

- [Documentation directory](#): the most up-to-date design documents
- [The Linux Kernel Documentation](#): the extensive documents extracted from kernel source
- [Linux Weekly News](#): easy explanation of recently added kernel features
- [Linux Inside](#): textbook-style description on kernel subsystems
- [Kernel newbies](#): useful information for new kernel developers
- [Linux Kernel API Manual](#)
- [Kernel Recipes](#)
- [kernel planet](#)

Next actions

- Master the essential tools, seriously
 - editor: `vim` , `emacs`
 - code navigation: `cscope` , `ctags`
 - version control: `git` , `tig`
 - terminal: `ssh` , `tmux`
- Useful lecture videos: [Vim](#), [tmux](#), [ssh](#), [Git](#)

Next lecture

- Isolation and system call
- Explore how following three system calls are implemented in the kernel

```
fd = open("out", 1);
write(fd, "hello\n", 6);
pid = fork();
```

Isolation and System Calls

Dongyoon Lee

Summary of last lectures

- Getting, building, and exploring the Linux kernel
 - `git`, `tig`, `make`, `make modules`, `make modules_install`, `make install`, `vim`, `emacs`, LXR, `cscope`, `ctags`, `tmux`
- *Don't try to master them at once. Instead gradually get used to them.*

How to read kernel code (top-down)

- **E.g., ext4 file system**
 1. General understanding on file systems in OS ← any [OS text book](#)
 2. File system in Linux kernel ← Ch. 13 in our text book
 3. Check [kernel Documentation](#) and [Ext4 on-disk layout](#)
 4. Read the ext4 kernel code
 - Module by module (e.g., dir, file, block management)
 - Start from a system call (e.g., how write() is implemented?)
 5. Search LWN to check the latest changes → E.g., [ext4 encryption support](#)

How to read kernel code (bottom-up)

- **Use function tracer**
 - ftrace : function tracer framework
 - perf tools : ftrace front end
- kernel/funcgraph
 - trace a graph of kernel function calls, showing children and times

```
# ./funcgraph -Htp 5363 vfs_read
Tracing "vfs_read" for PID 5363... Ctrl-C to end.
# tracer: function_graph
#
#      TIME          CPU    DURATION        FUNCTION CALLS
#      |          |    |    |
1728.478683 |  0)          |           | vfs_read() {
1728.478690 |  0)          |           |   rw_verify_area() {
1728.478691 |  0)          |           |     security_file_permission() {
1728.478692 |  0)          |           |       selinux_file_permission() {
```

How to navigate kernel code

```
$ KBUILD_ABS_SRCTREE=1 make ARCH=x86_64 cscope tags -j2
# KBUILD_ABS_SRCTREE=1 # use absolute path
# ARHC=x86_64          # select CPU architcture
# cscope                # build cscope database
# tags                  # build ctag database
# -j2                   # concurrently index source code using 2 CPUs

$ vim
# :tag <symbol>          # search symbol definition
# :cs find s <symbol>     # find uses of symbol
# Ctrl-]                  # search symbol definition on the cursor
# Ctrl-t                  # returning after a tag jump
# :bp :bn                 # nativate back and forth between files
```

Reference: [Vim Tips Wiki: Browsing programs with tag](#)

What's operating system (again)?

- OS design focuses on:
 - **Abstracting** the hardware for convenience and portability
 - **Multiplexing** the hardware among multiple applications
 - **Isolating** applications that might contain bugs
 - Allowing **sharing** among applications

Today: isolation and system calls

- How to isolate user applications from the kernel?
- How to safely access the kernel from user application?

The unit of isolation: “process”

- Prevent process X from wrecking or spying on process Y
 - (e.g., memory, cpu, FDs, resource exhaustion)
- Prevent a process from wrecking the operating system itself
 - (i.e. from preventing kernel from enforcing isolation)
- In the face of bugs or malice
 - (e.g. a bad process may try to trick the h/w or kernel)
- **Q: can we isolate a process from kernel?**

Isolation mechanisms in operating systems

1. User/kernel mode flag (aka ring)
2. Address spaces (later)
3. Timeslicing (later)
4. System call interface

Hardware isolation in x86 (aka ring)

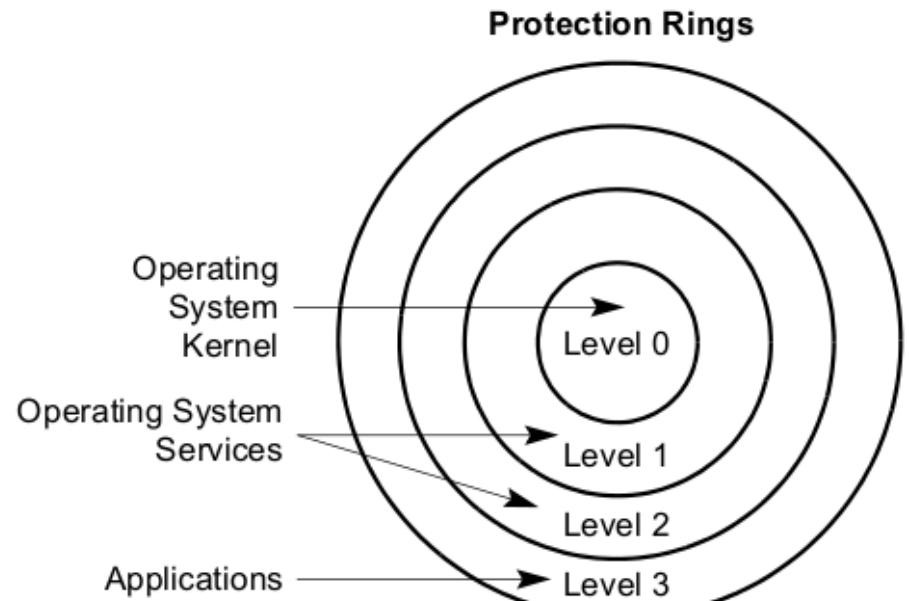


Figure 5-3. Protection Rings

- Q: How isolation is enforced in x86?

Segmentation in x86

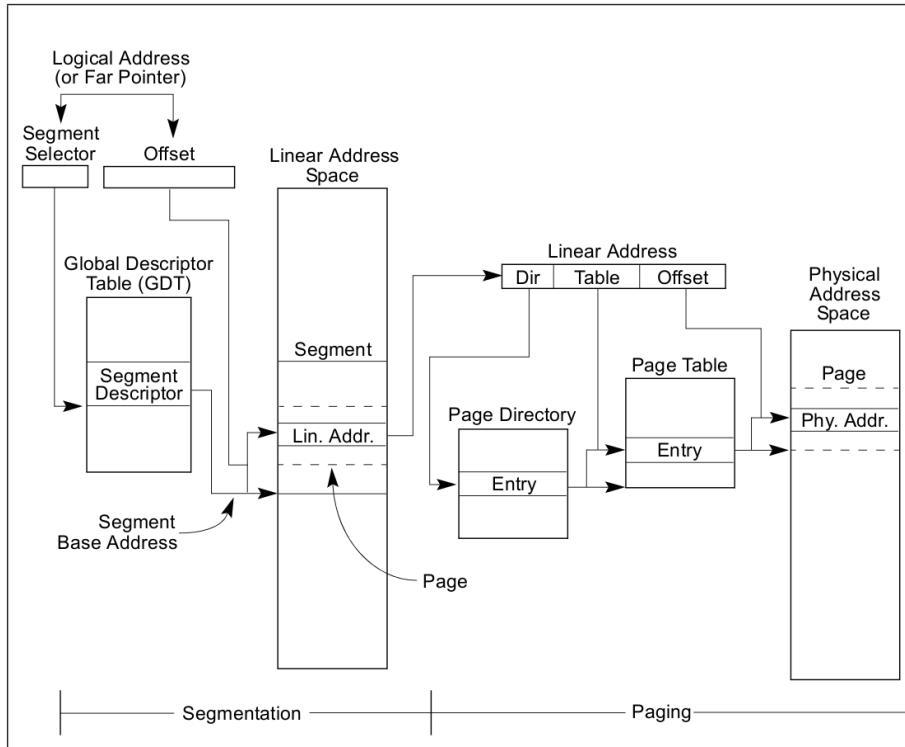


Figure 3-1. Segmentation and Paging

Segmentation in x86_64

3.2.4 Segmentation in IA-32e Mode

In IA-32e mode of Intel 64 architecture, the effects of segmentation depend on whether the processor is running in compatibility mode or 64-bit mode. In compatibility mode, segmentation functions just as it does using legacy 16-bit or 32-bit protected mode semantics.

In 64-bit mode, segmentation is generally (but not completely) disabled, creating a flat 64-bit linear-address space. The processor treats the segment base of CS, DS, ES, SS as zero, creating a linear address that is equal to the effective address. The FS and GS segments are exceptions. These segment registers (which hold the segment base) can be used as additional base registers in linear address calculations. They facilitate addressing local data and certain operating system data structures.

Note that the processor does not perform segment limit checks at runtime in 64-bit mode.

- Segmentation in 64-bit mode is generally disabled
- Two important features in segmentation are:
 - checking privilege level → ring 0, 3
 - implementing thread-local storage (TLS) → `fs`, `gs`

Privilege levels of a segment

- CPL (current privilege level)
 - the privilege level of currently executing program
 - bits 0 and 1 in the `%cs` register
- RPL (requested privilege level)
 - an override privilege level that is assigned to a segment selector
 - a segment selector is a part (16-bit) of segment registers (e.g., `ds` ,
`fs`), which is an index of a segment descriptor and RPL
- DPL (descriptor privilege level)
 - the privilege level of a segment

How isolation is enforced in x86?

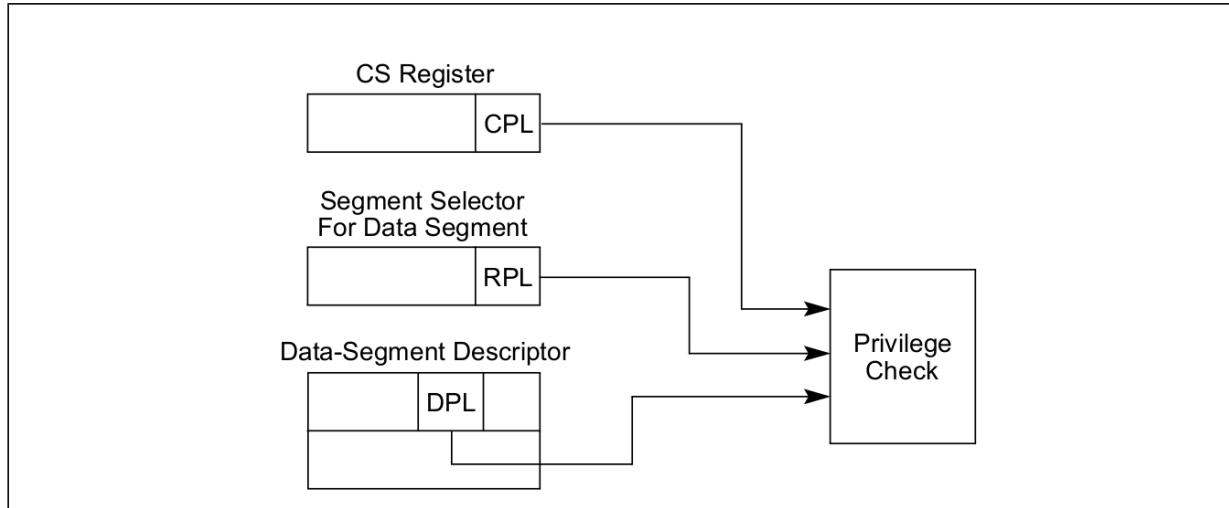


Figure 5-4. Privilege Check for Data Access

- Access is granted if $DPL \geq RPL$ and $DPL \geq CPL$

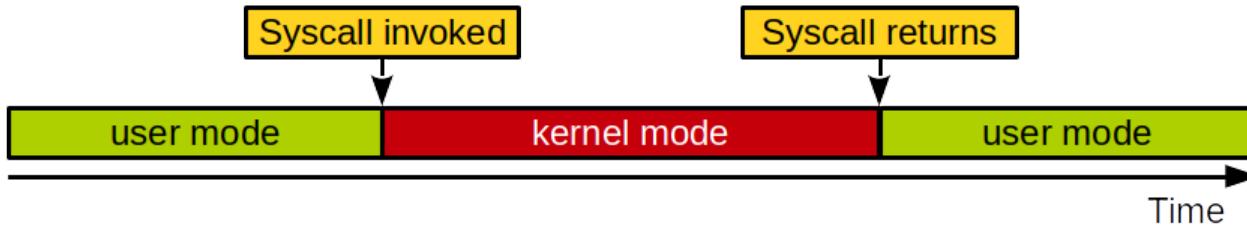
What does “ring 0” protect?

- Protects everything relevant to isolation
 - writes to `%cs` (to defend CPL)
 - every memory read/write
 - I/O port accesses
 - control register accesses (`eflags` , `%fs` , `%gs` ,...)

How to switch b/w rings (ring 0 ↔ ring 3)?

- Controlled transfer: system call
 - `int`, `sysenter` or `syscall` instruction set CPL to 0; change to KERNEL_CS and KERNEL_DS segments
 - set CPL to 3 before going back to user space; change to USER_CS and USER_DS segments
- **Q: How to systematically manage such interfaces?**

System call



- One and only way for user-space application to enter the kernel to request OS services and privileged operations such as accessing the hardware
 - A layer between the hardware and user-space processes
 - An abstract hardware interface for user-space
 - Ensure system security and stability

Examples of system calls

- Process management/scheduling: `fork`, `exit`, `execve`, `nice`,
`{get|set}priority`, `{get|set}pid`
- Memory management: `brk`, `mmap`
- File system: `open`, `read`, `write`, `lseek`, `stat`
- Inter-Process Communication: `pipe`, `shmget`
- Time management: `{get|set}timeofday`
- Others: `{get|set}uid`, `connect`
- **Q: Where are system call implementations in Linux kernel?**

Syscall table and syscall identifier

- The syscall table for x86_64 architecture
 - `linux/arch/x86/entry/syscalls/syscall_64.tbl`
- Syscall ID: unique integer ← sequentially assigned

```
# 64-bit system call numbers and entry vectors
#
# The format is:
# <number> <abi> <name> <entry point>
#
# The abi is "common", "64" or "x32" for this file.
0    common  read           sys_read
1    common  write          sys_write
2    common  open           sys_open
...
332   common statx         sys_statx
```

sys_call_table

- `syscall_64.tbl` is translated to an array of function pointers,
`sys_call_table`, upon kernel build
 - `linux/arch/x86/entry/syscalls/syscalltbl.sh`

```
asmlinkage const sys_call_ptr_t sys_call_table[__NR_syscall_max+1] = {  
    [0 ... __NR_syscall_max] = &sys_ni_syscall,  
    [0] = sys_read,  
    [1] = sys_write,  
    [2] = sys_open,  
    ...  
    ...  
    ...  
};
```

Syscall implementation (e.g., `read`)

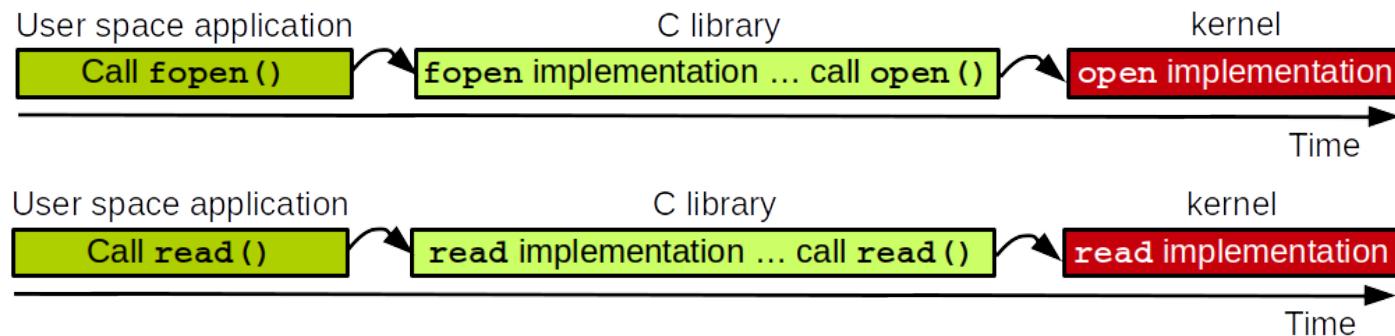
```
# linux/arch/x86/entry/syscalls/syscall_64.tbl
# 64-bit system call numbers and entry vectors
#
# The format is:
# <number> <abi> <name> <entry point>
0    common   read           sys_read
1    common   write          sys_write

/* linux/fs/read_write.c */
/* ssize_t write(int fd, const void *buf, size_t count); */
SYSCALL_DEFINE3(write, unsigned int, fd, const char __user *, buf,
               size_t, count)
{
    return ksys_write(fd, buf, count);
}
```

- **Q: How is a system call invoked?**

Invoking a syscall from a user space

- Syscalls are rarely invoked directly
 - Most of them are wrapped by the C library (`libc`, POSIX API)



Invoking a syscall from an application

- A syscall can be directly called through `syscall`
 - See `man syscall` → C library uses `syscall`

```
#include <unistd.h>
#include <sys/syscall.h> /* for SYS_xxx definitions */

int main(void)
{
    char    msg[] = "Hello, world!\n";
    ssize_t bytes_written;

    /* ssize_t write(int fd, const void *msg, size_t count);      */
    bytes_written = syscall(1, 1, msg, 14);
    /*           \ \                                         */
    /*           \   +- fd: standard output                 */
    /*           +- write syscall id (or SYS_write) */
    return 0;
}
```

Invoking a syscall from an application

- x86_64 architecture has a `syscall` instruction

```
.data

msg:
    .ascii "Hello, world!\n"
    len = . - msg

.text
.global _start

_start:
    mov $1, %rax      # syscall id: write
    mov $1, %rdi      # 1st arg: fd (standard output)
    mov $msg, %rsi     # 2nd arg: msg
    mov $len, %rdx     # 3rd arg: length of msg
    syscall           # switch from user space to kernel space

    mov $60, %rax      # syscall id: exit
    xor %rdi, %rdi     # 1st arg: 0
    syscall           # switch from user space to kernel space
```

Transition from a user space to kernel space

- x86 instruction for system call
 - int \$0x80: raise a software interrupt 128 (old)
 - sysenter: fast system call (x86_32)
 - syscall: fast system call (x86_64)
- Passing a syscall ID and parameters
 - syscall ID: %rax
 - parameters (x86_64): rdi, rsi, rdx, r10, r8 and r9

Handling the syscall interrupt

- The kernel syscall interrupt handler, system call handler
 - `entry_SYSCALL_64` at [linux/arch/x86/entry/syscall/entry_64.S](https://github.com/torvalds/linux/blob/v5.15/arch/x86/entry/syscall/entry_64.S)
(`do_syscall_64`)
 - `entry_SYSCALL_64` is registered at CPU initialization time
 - A handler of `syscall` is specified at a `IA32_LSTAR` MSR register
 - The address of `IA32_LSTAR` MSR is set to `entry_SYSCALL_64` at boot time: `syscall_init()` at [linux/arch/x86/kernel/cpu/common.c](https://github.com/torvalds/linux/blob/v5.15/arch/x86/kernel/cpu/common.c)

Handling the syscall interrupt

- entry_SYSCALL_64 invokes the entry function for the syscall ID
 - call do_syscall_64
 - regs->ax = sys_call_table[nr](regs);

```
# linux/arch/x86/entry/syscalls/syscall_64.tbl
# 64-bit system call numbers and entry vectors
#
# The format is:
# <number> <abi> <name> <entry point>
0    common  read           sys_read
1    common  write          sys_write
```

```
asmlinkage const sys_call_ptr_t sys_call_table[__NR_syscall_max+1] = {
    [0 ... __NR_syscall_max] = &sys_ni_syscall,
    [0] = sys_read,
    [1] = sys_write,
```

Returning from the syscall interrupt

- x86 instruction for system call
 - iret : interrupt return (x86-32 bit, old)
 - sysexit : fast return from fast system call(x86-32 bit)
 - sysret : return from fast system call (x86-64 bit)

Syscall example: `gettimeofday`

- `man gettimeofday`

NAME

`gettimeofday`, `settimeofday` - get / set time

SYNOPSIS

```
#include <sys/time.h>

int gettimeofday(struct timeval *tv, struct timezone *tz);

int settimeofday(const struct timeval *tv, const struct timezone *tz);
```

DESCRIPTION

The functions `gettimeofday()` and `settimeofday()` can get and set the time as well as a timezone. The `tv` argument is a `struct timeval` (as specified in `<sys/time.h>`):

```
struct timeval {
    time_t      tv_sec;      /* seconds */
    suseconds_t tv_usec;     /* microseconds */
};
```

Example C code

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/time.h>

int main(void)
{
    struct timeval tv;
    int ret;

    ret = gettimeofday(&tv, NULL);
    if(ret == -1)
    {
        perror("gettimeofday");
        return EXIT_FAILURE;
    }

    printf("Local time:\n");
    printf(" sec:%lu\n", tv.tv_sec);
    printf(" usec:%lu\n", tv.tv_usec);

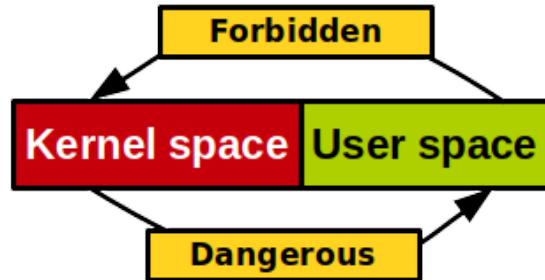
    return EXIT_SUCCESS;
}
```

Kernel implementation: **sys_gettimeofday**

```
/* linux/kernel/time/time.c */
/* SYSCALL_DEFINE2: a macro to define a syscall with two parameters */
SYSCALL_DEFINE2(gettimeofday, struct timeval __user *, tv,
               struct timezone __user *, tz) /* __user: user-space address */
{
    if (likely(tv != NULL)) { /* likely: branch hint */
        struct timespec64 ts;

        ktime_get_real_ts64(&ts);
        if (put_user(ts.tv_sec, &tv->tv_sec) ||
            put_user(ts.tv_nsec / 1000, &tv->tv_usec))
            return -EFAULT;
    }
    if (unlikely(tz != NULL)) {
        /* memcpy to usr-space memory */
        if (copy_to_user(tz, &sys_tz, sizeof(sys_tz)))
            return -EFAULT;
    }
    return 0;
}
```

User-space vs. kernel-space memory



- User space cannot access kernel memory
- Kernel code must never blindly follow a pointer into user-space
 - Accessing incorrect user address can make kernel crash!
- **Q: How to prevent a user-space access kernel-space memory?**
- **Q: How to safely access user-space memory?**

copy_{from|to}_user

```
/* copy user-space memory to kernel-space memory */
static inline
long copy_from_user(void *to, const void __user *from, unsigned long n);

/* copy kernel-space memory to user-space memory */
static inline
long copy_to_user(void __user *to, const void *from, unsigned long n);
```

- Is the provided user-space memory is legitimate?
 - If not, raise an illegal access error
- Does the user-space memory exist?
 - If swapped out, kernel accesses the user-space memory after swap in so the process can sleep

Implementing a new system call

1. Write your syscall function

- Add to the existing file or create a new file
- Add your new file into the kernel Makefile

2. Add it to the syscall table and assign an ID

- `linux/arch/x86/entry/syscalls/syscall_64.tbl`

3. Add its prototype in `linux/include/linux/syscalls.h`

4. Compile, reboot, and run

- Touching the syscall table will trigger the entire kernel compilation

Implementing a new system call

- Example: syscall implemented in linux sources in
linux/my_syscall/my_func.c
- Create a linux/my_syscall/Makefile

```
obj-y += my_func.o
```

- Add `my_syscall` in linux/Makefile

```
core-y      += kernel/ certs/ mm/ fs/ ipc/ security/ crypto/ block/ my_syscall/
```

Why not to implement a system call

- **Pros:** Easy to implement and use, fast
- **Cons:**
 - Needs an official syscall number
 - Interface cannot change after implementation
 - Must be registered for each architecture
 - Probably too much work for small exchanges of information
- **Alternative:**
 - Create a device node and `read()` and `write()`
 - Use `ioctl()`

Improving system call performance

- System call performance is critical in many applications
 - Web server: `select()`, `poll()`
 - Game engine: `gettimeofday()`
- **Hardware:** add a new fast system call instruction
 - `int 0x80` → `syscall`

Improving system call performance

- **Software:** vDSO (virtual dynamically linked shared object)
 - A kernel mechanism for exporting a kernel space routines to user space applications
 - No context switching overhead
 - E.g., `gettimeofday()`
 - the kernel allows the page containing the current time to be mapped read-only into user space
- **Software:** FlexSC: Exception-less system call, OSDI 2010

Project: common mistakes #1

- Direct operation on userspace string
 - **DONT** `strlen(user_str)`
 - **DO** `strlen_user`, `strncpy_from_user`
 - **DO** `copy_from_user`, `copy_to_user`

Project: common mistakes #2

- **DO** Use `SYSCALL_DEFINE` macro

```
/* !!! WRONG !!! DO NOT USE !!! */
asmlinkage long sys_my_syscall2(char *user_str)
{
    /* ... */
}

/* !!! CORRECT !!! */
SYSCALL_DEFINE1(my_syscall2, char __user *, user_str)
{
    /* ... */
}
```

Project: common mistakes #3

- Potential kernel stack overflow
- **DONT** use a stack (function local) array

```
/* !!! WRONG !!! DO NOT USE !!! */
SYSCALL_DEFINE1(my_syscall2, char __user *, user_str)
{
    int str_len = strlen_user(user_str);
    char str_buffer[str_len]; /* What happen if str_len is 16KB? */
    /* ... */
}
```

Next lecture

- Kernel Data Structures

Further readings

- LWN: Anatomy of a system call: [part 1](#) and [part2](#)
- [LWN: On vsyscalls and the vDSO](#)
- [Linux Inside: system calls](#)
- [Linux Performance Analysis: New Tools and Old Secrets](#)

Kernel Data Structure I

Dongyoon Lee

Summary of last lectures

- Tools
 - git, tig, make, cscope, ctags, vim, emacs, tmux, ssh, etc.
- System call
 - isolation, x86 ring architecture

Today: Kernel Data Structures

- Linked list
- Hash table
- Red-black tree
- Radix tree (next class)
- Bitmap (next class)

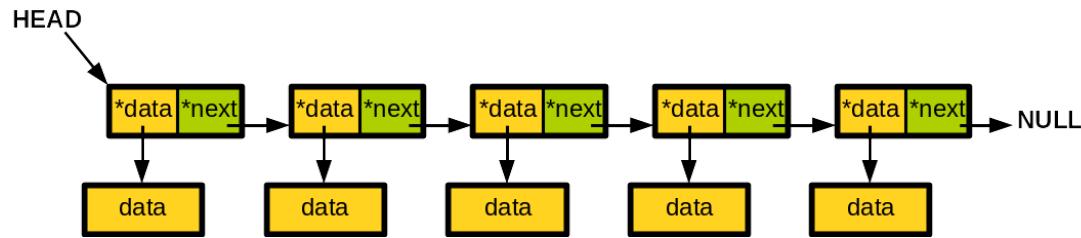
Why data structure is particularly important?

“

I will, in fact, claim that the difference between a bad programmer and a good one is whether he considers his code and his data structures more important. Bad programmers worry about the code, Good programmers worry about data structures and their relationships. - Linus Torvalds

Singly linked list (CS101)

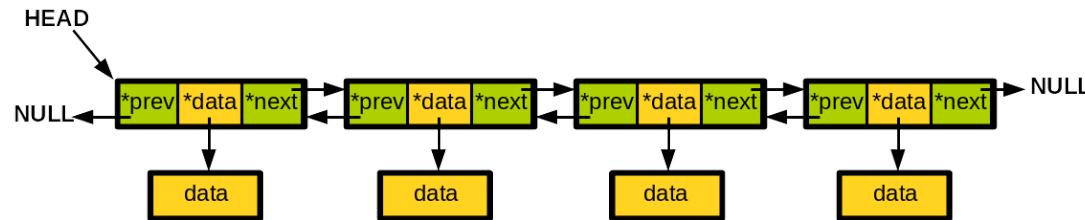
```
struct my_list_element {  
    void *data; /* void pointer to point on a generic data */  
    struct my_list_element *next; /* pointer to a next element */  
};
```



- Starts from HEAD and terminates at NULL
- Traverses forward only
- When empty, HEAD is NULL

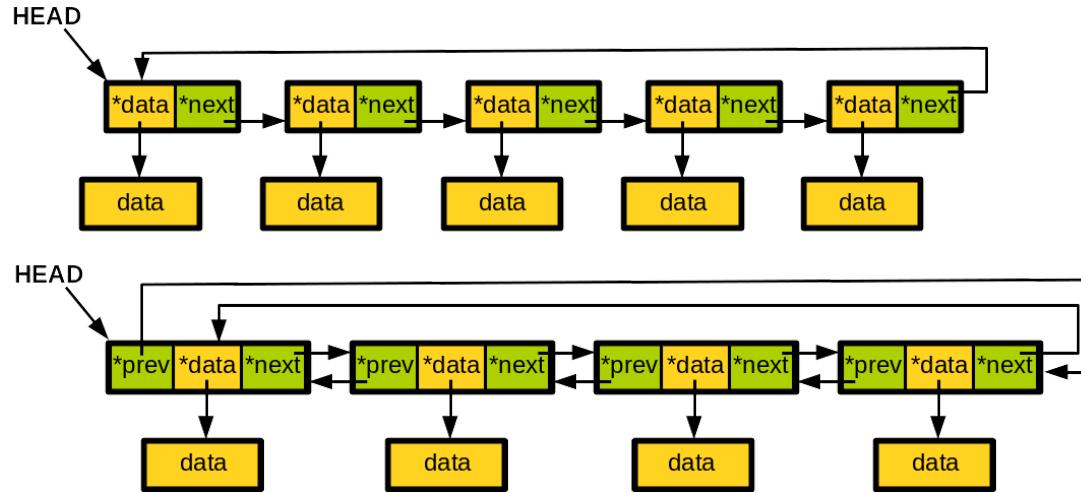
Doubly linked list (CS101)

```
struct my_list_element {  
    void *data; /* void pointer to point on a generic data */  
    struct my_list_element *prev; /* pointer to a previous element */  
    struct my_list_element *next; /* pointer to a next element */  
};
```



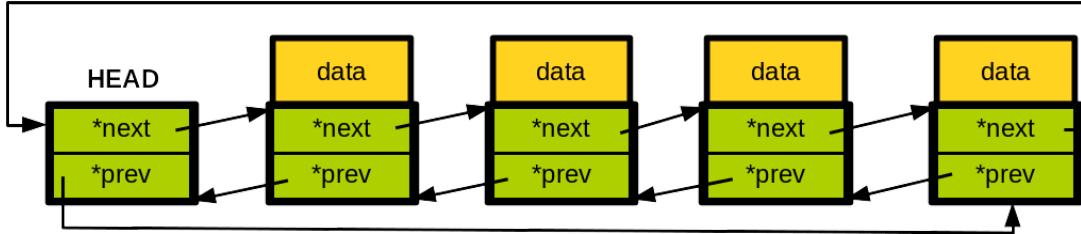
- Starts from HEAD and terminates at NULL
- Traverses forward and **backward**
- When empty, HEAD is NULL

Circular linked list (CS101)



- Starts from `HEAD` and terminates at `HEAD`
- When empty, `HEAD` is `NULL`
- **Easy to insert a new element at the end of a list**

Linux linked list



- Starts from `HEAD` and terminates at `HEAD`
- When empty, `HEAD` is **not** `NULL`
 - `prev` and `next` of `HEAD` points `HEAD`
 - `HEAD` is a sentinel node
- Easy to insert a new element at the end of a list
- **There is no exceptional case to handle `NULL`**

Linux linked list

- A circular doubly linked list
- Two differences from the typical design
 1. Embedding a linked list node in the structure
 2. Using a sentinel node as a list header
- `linux/include/linux/list.h`

Linux linked list

```
struct list_head {           /* kernel linked list data structure */
    struct list_head *next, *prev;
};

struct car {
    struct list_head list; /* add list_head instead of prev and next */
    unsigned int max_speed; /* put data directly */
    unsigned int drive_wheen_num;
    unsigned int price_in_dollars;
};

struct list_head my_car_list; /* HEAD is also list_head */
```

- `struct list_head` is the key data structure
- `list_head` is embedded in the data structure
- Start of a list is also `list_head`, `my_car_list` → sentinel node

Getting a data from its `list_head`

- How to get the pointer of `struct car` from its `list`
 - use `list_entry(ptr, type, member)`
 - just a pointer arithmetic

```
struct car *amazing_car = list_entry(car_list_ptr, struct car, list);

/**
 * list_entry - get the struct for this entry
 * @ptr:    the &struct list_head pointer.
 * @type:   the type of the struct this is embedded in.
 * @member: the name of the list_head within the struct.
 */
#define list_entry(ptr, type, member) container_of(ptr, type, member)
#define container_of(ptr, type, member) ({ \
    const typeof( ((type *)0)->member ) *_mptr = (ptr); \
    (type *)((char *)_mptr - offsetof(type,member));})
#define offsetof(TYPE, MEMBER) ((size_t)&((TYPE *)0)->MEMBER)
```

Defining a list

```
struct car *my_car = kmalloc(sizeof(*my_car), GFP_KERNEL);
my_car->max_speed = 150;
my_car->drive_wheel_num = 2;
my_car->price_in_dollars = 10000.0;
INIT_LIST_HEAD(&my_car->list); /* initialize an element */

struct car my_car {
    .max_speed = 150,
    .drive_wheel_num = 2,
    .price_in_dollars = 10000,
    .list = LIST_HEAD_INIT(my_car.list), /* initialize an element */
}

LIST_HEAD(my_car_list); /* initialize the HEAD of a list */
```

- Initializing a `list_head`
 - `list_head->prev = &list_head`
 - `list_head->next = &list_head`

Manipulating a list: O(1)

```
/* Insert a new entry after the specified head */
void list_add(struct list_head *new, struct list_head *head);

/* Insert a new entry before the specified head */
void list_add_tail(struct list_head *new, struct list_head *head);

/* Delete a list entry
 * NOTE: You still have to take care of the memory deallocation if needed */
void list_del(struct list_head *entry);

/* Delete from one list and add as another's head */
void list_move(struct list_head *list, struct list_head *head);

/* Delete from one list and add as another's tail */
void list_move_tail(struct list_head *list, struct list_head *head);

/* Tests whether a list is empty */
int list_empty(const struct list_head *head);

/* Join two lists (merge a list to the specified head) */
void list_splice(const struct list_head *list, struct list_head *head);
```

Iterating over a list: O(n)

```
/**  
 * list_for_each - iterate over a list  
 * @pos:    the &struct list_head to use as a loop cursor.  
 * @head:   the head for your list.  
 */  
#define list_for_each(pos, head) \  
    for (pos = (head)->next; pos != (head); pos = pos->next)  
  
/**  
 * list_for_each_entry - iterate over list of given type  
 * @pos:    the type * to use as a loop cursor.  
 * @head:   the head for your list.  
 * @member: the name of the list_head within the struct.  
 */  
#define list_for_each_entry(pos, head, member) \  
    for (pos = list_first_entry(head, typeof(*pos), member); \  
         &pos->member != (head); \  
         pos = list_next_entry(pos, member))
```

Iterating over a list: O(n)

```
/* Temporary variable needed to iterate: */
struct list_head p;

/* This will point on the actual data structures
 * (struct car)during the iteration: */
struct car *current_car;

list_for_each(p, &my_car_list) {
    current_car = list_entry(p, struct car, list);
    printk(KERN_INFO "Price: %lf\n", current_car->price_in_dollars);
}

/* Simpler: use list_for_each_entry */
list_for_each_entry(current_car, &my_car_list, list) {
    printk(KERN_INFO "Price: %lf\n", current_car->price_in_dollars);
}
```

- Backward iteration?

- `list_for_each_entry_reverse(pos, head, member)`

Iterating while removing

```
#define list_for_each_safe(pos, next, head) ...
#define list_for_each_entry_safe(pos, next, head, member) ...

/* This will point on the actual data structures
 * (struct car) during the iteration: */
struct car *current_car, *next;
list_for_each_entry_safe(current_car, next, my_car_list, list) {
    printk(KERN_INFO "Price: %lf\n", current_car->price_in_dollars);
    list_del(current_car->list);
    kfree(current_car); /* if dynamically allocated using kmalloc */
}
```

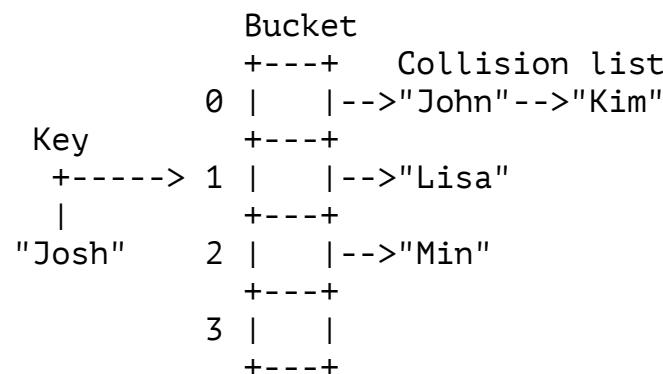
- For each iteration, `next` points to the next node
 - Can safely remove the current node
 - Otherwise, can cause a use-after-free bug

Usages of linked lists in the kernel

- Kernel code makes extensive use of linked lists:
 - a list of threads under the same parent PID
 - a list of superblocks of a file system
 - and many more

Linux hash table

- A simple fixed-size open chaining hash table
 - The size of bucket array is fixed at initialization as a 2^N
 - Each bucket has a singly linked list to resolve hash collision.
 - Time complexity: O(1)



Linux hash table

- A simple fixed-size chained hash table
 - The size of bucket array is fixed at initialization as a 2^N
 - Each bucket has a singly linked list to resolve hash collision.
 - Time complexity: O(1)

```
Bucket
+---+ Collision list
0 | -->"John"-->"Kim"
+---+
1 | -->"Josh"-->"Lisa"
+---+
2 | -->"Min"
+---+
3 | |
+---+
```

Linux hash table

```
/* linux/include/linux/hashtable.h, types.h */
/* hash bucket */
struct hlist_head {
    struct hlist_node *first;
};

/* collision list */
struct hlist_node {
    /* Similar to list_head, hlist_node is embedded
     * into a data structure. */
    struct hlist_node *next;
    struct hlist_node **pprev; /* &prev->next */
};

Bucket: array of hlist_head
+---+ Collision list: hlist_node
0 |   |-->"John"-->"Kim"
+---+
1 |   |-->"Josh"-->"Lisa"
+---+
2 |   |-->"Min"
+---+
3 |   |
+---+
```

Linux hash table API

```
/**  
 * Define a hashtable with 2^bits buckets  
 */  
#define DEFINE_HASHTABLE(name, bits) ...  
  
/**  
 * hash_init - initialize a hash table  
 * @hashtable: hashtable to be initialized  
 */  
#define hash_init(hashtable) ...  
  
/**  
 * hash_add - add an object to a hashtable  
 * @hashtable: hashtable to add to  
 * @node: the &struct hlist_node of the object to be added  
 * @key: the key of the object to be added  
 */  
#define hash_add(hashtable, node, key) ...
```

Linux hash table API

```
/**  
 * hash_for_each - iterate over a hashtable  
 * @name: hashtable to iterate  
 * @bkt: integer to use as bucket loop cursor  
 * @obj: the type * to use as a loop cursor for each entry  
 * @member: the name of the hlist_node within the struct  
 */  
#define hash_for_each(name, bkt, obj, member) ...  
  
        +---+  
0 |     |-->"John"-->"Kim"  
        +---+  
1 |     |-->"Josh"-->"Lisa"  
        +---+  
2 |     |-->"Min"  
        +---+  
3 |     |  
        +---+
```

Linux hash table API

```
/**  
 * hash_for_each_possible - iterate over all possible objects hashing to the  
 * same bucket  
 * @name: hashtable to iterate  
 * @obj: the type * to use as a loop cursor for each entry  
 * @member: the name of the hlist_node within the struct  
 * @key: the key of the objects to iterate over  
 */  
#define hash_for_each_possible(name, obj, member, key) ...  
  
      +---+  
      1 |    |-->"Josh"-->"Lisa"  
      +---+  
  
/**  
 * hash_del - remove an object from a hashtable  
 * @node: &struct hlist_node of the object to remove  
 */  
void hash_del(struct hlist_node *node);
```

Linux hash table example

- Transparent hugepage
 - finds physically consecutive 4KB pages
 - remaps consecutive 4KB pages to a 2MB page (huge page)
 - saves TLB entries and improves memory access performance by reducing TLB miss
 - maintains per-process memory structure, `struct mm_struct`

Linux hash table example

```
/* linux/mm/khugepaged.c */

#define MM_SLOTS_HASH_BITS 10
static DEFINE_HASHTABLE(mm_slots_hash, MM_SLOTS_HASH_BITS);

/* struct mm_slot - hash lookup from mm to mm_slot
 * @hash: hash collision list
 * @mm: the mm that this information is valid for
 */
struct mm_slot {
    struct hlist_node hash; /* hlist_node is embedded like list_head */
    struct mm_struct *mm;
};
```

Linux hash table example

```
/* add an mm_slot into the hash table
 * use the mm pointer as a key */
static void insert_to_mm_slots_hash(struct mm_struct *mm,
                                    struct mm_slot *mm_slot)
{
    mm_slot->mm = mm;
    hash_add(mm_slots_hash, &mm_slot->hash, (long)mm);
}

/* iterate the chained list of a bucket to find an entry */
static struct mm_slot *get_mm_slot(struct mm_struct *mm)
{
    struct mm_slot *mm_slot;

    hash_for_each_possible(mm_slots_hash, mm_slot, hash, (unsigned long)mm)
        if (mm == mm_slot->mm)
            return mm_slot;

    return NULL;
}
```

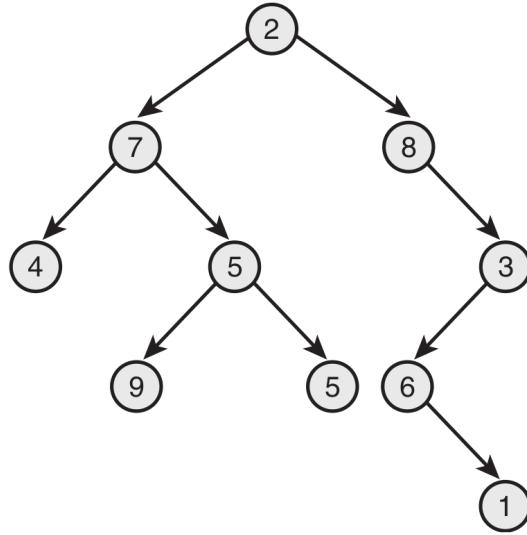
Linux hash table example

```
/* remove an entry after finding it */
void __khugepaged_exit(struct mm_struct *mm)
{
    struct mm_slot *mm_slot;

    spin_lock(&khugepaged_mm_lock);
    mm_slot = get_mm_slot(mm);
    if (mm_slot && khugepaged_scan.mm_slot != mm_slot) {
        hash_del(&mm_slot->hash);
        list_del(&mm_slot->mm_node);
        free = 1;
    }
    spin_unlock(&khugepaged_mm_lock);

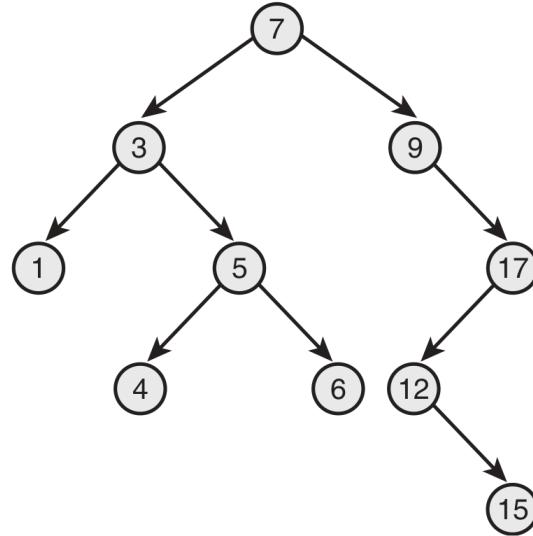
    clear_bit(MMF_VM_HUGEPAGE, &mm->flags);
    free_mm_slot(mm_slot);
    mmdrop(mm);
    /* ... */
}
```

Tree basics: **binary** tree



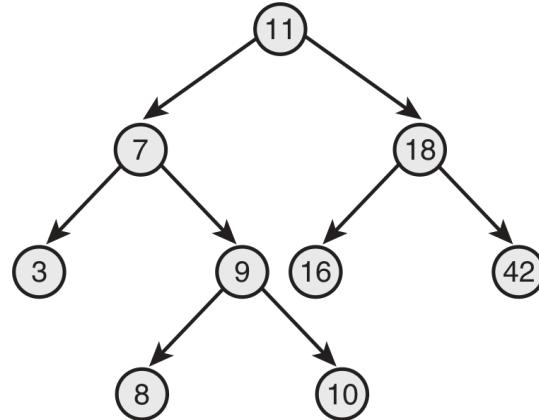
- Nodes have zero, one, or two children
- Root has no parent, other nodes have one

Tree basics: binary search tree



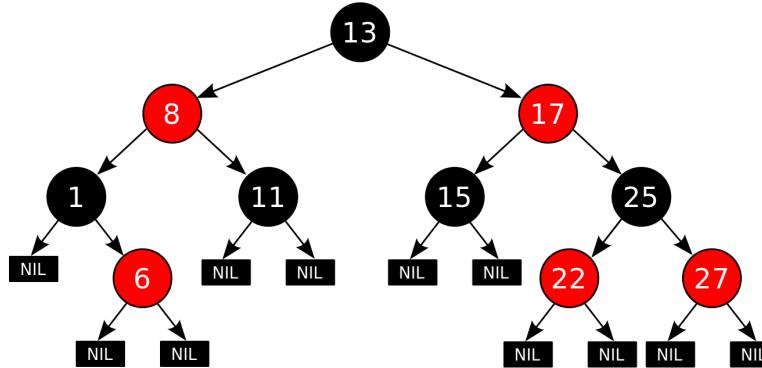
- Left children < parent
- Right children > parent
- Search and ordered traversal are efficient

Tree basics: **balanced** binary search tree



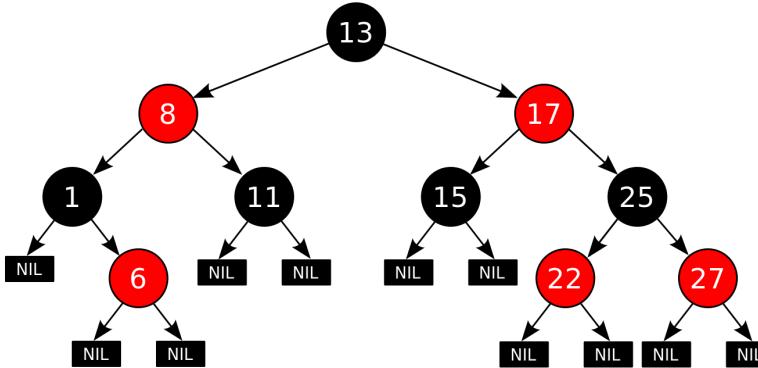
- Depth of all leaves differs by at most one
- Puts a boundary on the worst case operations

Tree basics: red-black tree



- A type of self-balancing binary search tree
 - Nodes: red or black
 - Leaves: black, no data

Tree basics: red-black tree



- Following properties are maintained during tree modifications:
 - The path from a node to one of its leaves contains the same number of black nodes as the shortest path to any of its other leaves.
- Fast search, insert, delete operations: $O(\log N)$

Linux red-black tree (or rbtree)

```
/* linux/include/linux/rbtree.h
 * linux/lib/rbtree.c */

/* Rbtree node, which is embedded to your data structure like
 * list_head and hlist node */
struct rb_node {
    unsigned long __rb_parent_color;
    struct rb_node *rb_right;
    struct rb_node *rb_left;
};

/* Root of a rbtree */
struct rb_root {
    struct rb_node *rb_node;
};

#define RB_ROOT (struct rb_root) { NULL, }

/* A macro to access data from rb_node */
#define rb_entry(ptr, type, member) container_of(ptr, type, member)
#define rb_parent(r) ((struct rb_node *)((r)->__rb_parent_color & ~3))
```

Linux red-black tree (or rbtree)

```
/* Find logical next and previous nodes in a tree */
struct rb_node *rb_next(const struct rb_node *);
struct rb_node *rb_prev(const struct rb_node *);
struct rb_node *rb_first(const struct rb_root *);
struct rb_node *rb_last(const struct rb_root *);

/* Insert a new node under a parent connected via rb_link */
void rb_link_node(struct rb_node *node, struct rb_node *parent,
                  struct rb_node **rb_link);

/* Re-balance an rbtree after inserting a node if necessary */
void rb_insert_color(struct rb_node *, struct rb_root *);

/* Delete a node */
void rb_erase(struct rb_node *, struct rb_root *);
```

Linux red-black tree example

- Completely Fair Scheduling (CFS)
 - Default task scheduler in Linux
 - Each task has `vruntime`, which presents how much time a task has run
 - CFS always picks a process with the smallest `vruntime` for fairness
 - Per-task `vruntime` structure is maintained in a rbtree

Linux red-black tree example

```
/* linux/include/linux/sched.h
 * linux/kernel/sched/fair.c, sched.h */

/* Define an rbtree */
struct cfs_rq {
    struct rb_root tasks_timeline; /* contains sched_entity */
};

/* Data structure of a task */
struct sched_entity {
    struct rb_node run_node; /* embed a rb_node */
    u64 vruntime; /* vruntime is the key of task_timeline */
};

/* Initialize an rbtree */
void init_cfs_rq(struct cfs_rq *cfs_rq)
{
    cfs_rq->tasks_timeline = RB_ROOT;
}
```

Linux red-black tree example

```
/* Enqueue an entity into the rb-tree: */
void __enqueue_entity(struct cfs_rq *cfs_rq, struct sched_entity *se)
{
    struct rb_node **link = &cfs_rq->tasks_timeline.rb_node; /* root node */
    struct rb_node *parent = NULL;
    struct sched_entity *entry;

    /* Traverse the rbtree to find the right place to insert */
    while (*link) {
        parent = *link;
        entry = rb_entry(parent, struct sched_entity, run_node);
        if (se->vruntime < entry->vruntime) {
            link = &parent->rb_left;
        } else {
            link = &parent->rb_right;
        }
    }

    /* Insert a new node */
    rb_link_node(&se->run_node, parent, link);
    /* Re-balance the rbtree if necessary */
    rb_insert_color(&se->run_node, &cfs_rq->tasks_timeline);
}
```

Linux red-black tree example

```
/* Delete a node */
void __dequeue_entity(struct cfs_rq *cfs_rq, struct sched_entity *se)
{
    rb_erase(&s->run_node, &cfs_rq->tasks_timeline);
}

/* Pick the first entity, which has the smallest vruntime,
 * for scheduling */
struct sched_entity *__pick_first_entity(struct cfs_rq *cfs_rq)
{
    return rb_first(&cfs_rq->tasks_timeline);
}
```

Design patterns of kernel data structures

- Embedding its pointer structure
 - `list_head`, `hlist_node`, `rb_node`
 - The programmer has full control of placement of fields in the structure in case they need to put important fields close together to improve cache utilization
 - A structure can easily be on two or more lists quite independently, simply by having multiple `list_head` fields
 - `container_of`, `list_entry`, and `rb_entry` are used to get its embedding data structure

Design patterns of kernel data structures

- Tool box rather than a complete solution for generic service
 - Sometimes it is best not to provide a complete solution for a generic service, but rather to provide a suite of tools that can be used to build custom solutions.
 - None of Linux list, hash table, and rbtree provides a `search` function.
 - You should build your own using given low-level primitives

Design patterns of kernel data structures

- Caller locks
 - When there is any doubt, choose to have the caller take locks rather than the callee. This puts more control in the hands of the client of a function.

Further readings

- [LWN: Linux kernel design patterns: part 2](#)
- [LWN: A generic hash table](#)
- [Hash Tables—Theory and Practice](#)
- [LWN: Relativistic hash tables](#)
- [LWN: Trees II: red-black trees](#)
- [Transparent Hugepage Support](#)
- [CFS Scheduler](#)

Next lecture

- More kernel data structures
 - radix tree, bitmap
- Kernel modules

Kernel Data Structures II

and kernel module

Dongyoon Lee

Summary of last lectures

- Essential kernel data structures
 - list, hash table, red-black tree
- Design patterns of kernel data structures
 - Embedding its pointer structure
 - Tool box rather than a complete solution for generic service
 - Caller locks

Today's agenda

- Memory allocation in the kernel
- More kernel data structures
 - Radix tree
 - XArray
 - Bitmap
- Kernel module

Memory allocation in kernel

- Two types of memory allocation functions are provided
 - `kmalloc(size, gfp_mask)` - `kfree(address)`
 - `vmalloc(size)` - `vfree(address)`
- `gfp_mask` is used to specify
 - which types of pages can be allocated
 - whether the allocator can wait for more memory to be freed
- Frequently used `gfp_mask`
 - `GFP_KERNEL` : a caller *might sleep*
 - `GFP_ATOMIC` : a caller *will not sleep* → higher chance of failure

kmalloc(size, GFP_MASK)

- Allocate virtually and *physically contiguous* memory
 - where physically contiguous memory is necessary
 - E.g., DMA, memory-mapped IO, performance in accessing
- The maximum allocatable size through one `kmalloc` is limited
 - 4MB on x86 (architecture dependent)

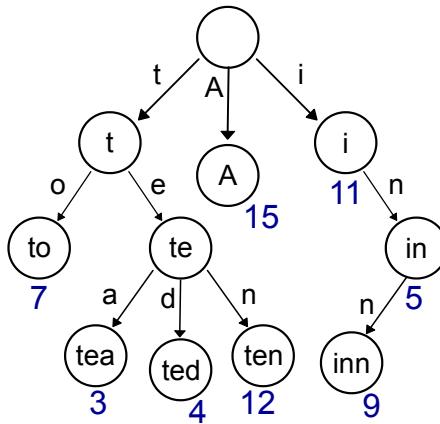
```
#include <linux/slab.h>
void my_function()
{
    char *my_string = (char *)kmalloc(128, GFP_KERNEL);
    my_struct my_struct_ptr = (my_struct *)kmalloc(sizeof(my_struct), GFP_KERNEL);
    /* ... */
    kfree(my_string);
    kfree(my_struct_ptr);
}
```

vmalloc(size)

- Allocate memory that is *virtually contiguous, but not physically contiguous*
- No size limit other than the amount of free RAM
 - Swapping is not supported for kernel memory
- Memory allocator might sleep to get more free memory
- Unit of allocation is a page (4KB)

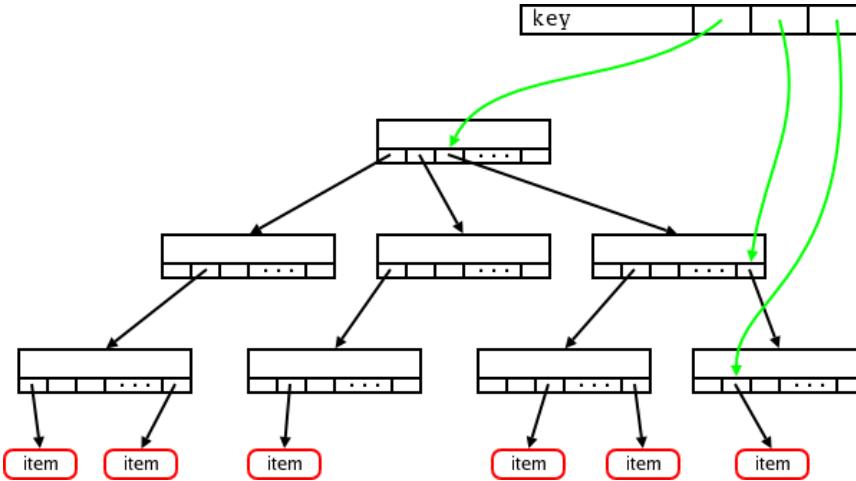
```
#include <linux/slab.h>
void my_function()
{
    char *my_string = (char *)vmalloc(128);
    my_struct my_struct_ptr = (my_struct *)vmalloc(sizeof(my_struct));
    /* ... */
    vfree(my_string);
    vfree(my_struct_ptr);
}
```

Radix tree (or trie, digital tree, prefix tree)



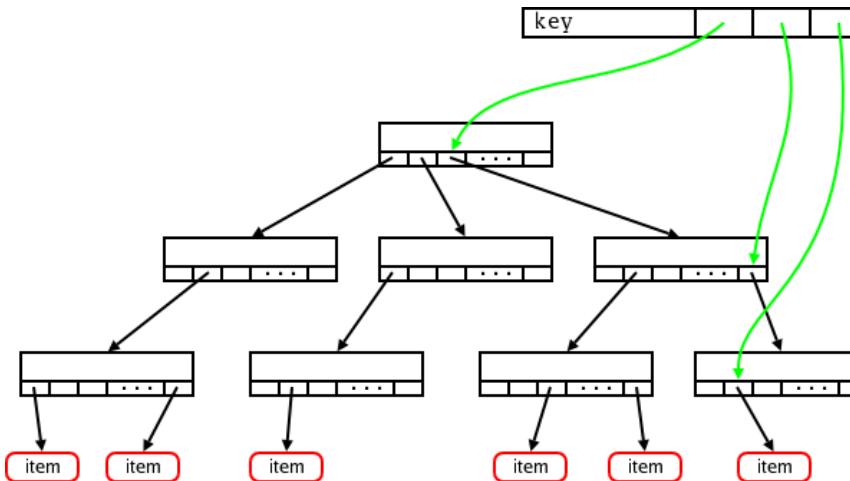
- The key at each node is compared chunk-of-bits by chunk-of-bits
- All descendants of a node have a common prefix
- Values are only associated with leaves
- Source: [Wikipedia](#)

Linux radix tree



- Mapping between `unsigned long` and `void *`
- Each node has 64 slots
- Slots are indexed by a 6-bit ($2^6=64$) portion of the key
- Source: [LWN](#)

Linux radix tree



- At leaves, a slot points to an address of data
- At non-leaf nodes, a slot points to another node in a lower layer
- Other metadata is also stored at each node:
 - **tags**, parent pointer, offset in parent, etc

Linux radix tree API

```
/* linux/include/linux/radix-tree.h, linux/lib/radix-tree.c */
#define RADIX_TREE_MAX_TAGS 3
#define RADIX_TREE_MAP_SIZE (1UL << 6)

/* Root of a radix tree */
struct radix_tree_root {
    gfp_t           gfp_mask; /* used to allocate internal nodes */
    struct radix_tree_node *rnode;
};

/* Radix tree internal node,
 * which is composed of slot and tag array */
struct radix_tree_node {
    unsigned char      offset;      /* Slot offset in parent */
    struct radix_tree_node *parent;   /* Used when ascending tree */
    void             *slots[RADIX_TREE_MAP_SIZE];
    unsigned long      tags[RADIX_TREE_MAX_TAGS][RADIX_TREE_TAG_LONGS];
    /* ... */
};
```

- Q: Is `radix_tree_node` embedded to user data like `list_head` ?

Linux radix tree API

```
/* Root of a radix tree */
struct radix_tree_root {
    gfp_t                  gfp_mask; /* used to allocate internal nodes */
    struct radix_tree_node *rnode;
};

/* Radix tree internal node,
 * which is composed of slot and tag array */
struct radix_tree_node {
    unsigned char           offset;      /* Slot offset in parent */
    struct radix_tree_node *parent;     /* Used when ascending tree */
    void                   *slots[RADIX_TREE_MAP_SIZE];
    unsigned long          tags[RADIX_TREE_MAX_TAGS][RADIX_TREE_TAG_LONGS];
    /* ... */
};
```

- Q: Is `radix_tree_node` embedded to user data like `list_head`?
 - It is dynamically allocated when inserting an item.

Linux radix tree API

```
/* Declare and initialize a radix tree
 * @gfp_mask: how memory allocations are to be performed
 *             (e.g., GFP_KERNEL, GFP_ATOMIC, GFP_FS, etc) */
RADIX_TREE(name, gfp_mask);

/* Initialize a radix tree at runtime */
struct radix_tree_root my_tree;
INIT_RADIX_TREE(my_tree, gfp_mask);

/* Insert an item into the radix tree at position @index.
 * @root:      radix tree root
 * @index:     index key
 * @item:      item to insert */
int radix_tree_insert(struct radix_tree_root *root,
                     unsigned long index, void *item);
```

- **Q: What happens if memory allocation fails?**

Linux radix tree API

- When failure to insert an item into a radix tree can be a significant problem, use `radix_tree_preload`

```
/* 1. Allocate sufficient memory (using the given gfp_mask) to guarantee
 * that the next radix tree insertion cannot fail. When successful,
 * it disables preemption so the pre-allocated memory can be used for
 * subsequent radix_tree_insert() operations. */
int radix_tree_preload(gfp_t gfp_mask);

/* 2. Insert an item into the radix tree at position @index.
 * @root: radix tree root
 * @index: index key
 * @item: item to insert */
int radix_tree_insert(struct radix_tree_root *root,
                     unsigned long index, void *item);

/* 3. Enable preemption again. */
void radix_tree_preload_end(void);
```

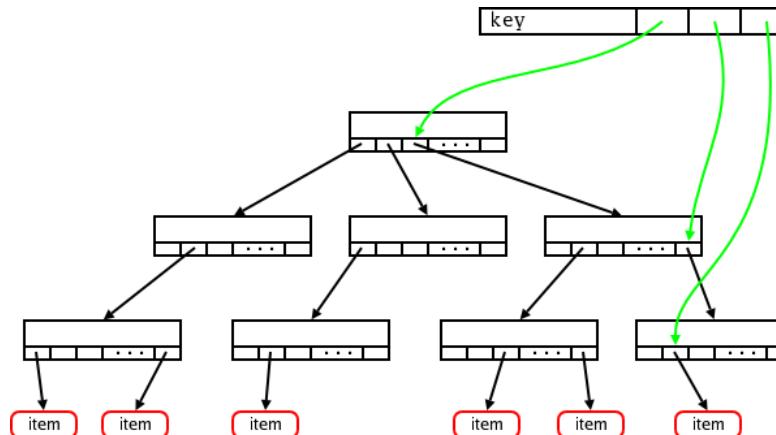
Linux radix tree API

Linux radix tree API

```
/* radix_tree_gang_lookup - perform multiple lookup on a radix tree
 * @root:          radix tree root
 * @results:       where the results of the lookup are placed
 * @first_index:   start the lookup from this key
 * @max_items:    place up to this many items at *results
 *
 * Performs an index-ascending scan of the tree for present items. Places
 * them at *@results and returns the number of items which were placed at
 * *@results. */
unsigned int
radix_tree_gang_lookup(const struct radix_tree_root *root, void **results,
                      unsigned long first_index, unsigned int max_items);
```

Linux radix tree API

- **tags:** specific bits can be set on items in the trees (0, 1, 2)
 - E.g., set the status of memory pages, which are dirty or under writeback



Linux radix tree API

```
/* radix_tree_tag_set - set a tag on a radix tree node
 * @root:      radix tree root
 * @index:     index key
 * @tag:       tag index
 *
 * Set the search tag (which must be < RADIX_TREE_MAX_TAGS)
 * corresponding to @index in the radix tree.  From
 * the root all the way down to the leaf node.
 *
 * Returns the address of the tagged item. */
void *radix_tree_tag_set(struct radix_tree_root *root,
                         unsigned long index, unsigned int tag);

/* radix_tree_tag_clear - clear a tag on a radix tree node
 *
 * Clear the search tag corresponding to @index in the radix tree.
 * If this causes the leaf node to have no tags set then clear the tag
 * in the next-to-leaf node, etc.
 *
 * Returns the address of the tagged item on success, else NULL. */
void *radix_tree_tag_clear(struct radix_tree_root *root,
                           unsigned long index, unsigned int tag);
```

Linux radix tree API

```
/* radix_tree_tag_get - get a tag on a radix tree node
 * @root:      radix tree root
 * @index:     index key
 * @tag:       tag index (< RADIX_TREE_MAX_TAGS)
 *
 * Return values:
 * 0: tag not present or not set
 * 1: tag set */
int radix_tree_tag_get(const struct radix_tree_root *root,
                      unsigned long index, unsigned int tag);

/* radix_tree_tagged - test whether any items in the tree are tagged
 * @root:      radix tree root
 * @tag:       tag to test */
int radix_tree_tagged(const struct radix_tree_root *root,
                      unsigned int tag);
```

Linux radix tree API

```
/* radix_tree_gang_lookup_tag - perform multiple lookup on a radix tree
 * based on a tag
 * @root:      radix tree root
 * @results:   where the results of the lookup are placed
 * @first_index: start the lookup from this key
 * @max_items: place up to this many items at *results
 * @tag:       the tag index (< RADIX_TREE_MAX_TAGS)
 *
 * Performs an index-ascending scan of the tree for present items which
 * have the tag indexed by @tag set. Places the items at *@results and
 * returns the number of items which were placed at *@results.
 */
unsigned int
radix_tree_gang_lookup_tag(const struct radix_tree_root *root, void **results,
                           unsigned long first_index, unsigned int max_items,
                           unsigned int tag);
```

Linux radix tree example

- The most important user is the page cache
 - Every time we look up a page in a file, we consult the radix tree to see if the page is already in the cache
 - Use tags to maintain the status of page (e.g.,
`PAGECACHE_TAG_DIRTY` or `PAGECACHE_TAG_WRITEBACK`)

Linux radix tree example

```
/* linux/include/linux/fs.h */

/* inode: a metadata of a file */
struct inode {
    umode_t          i_mode;
    struct super_block *i_sb;
    struct address_space *i_mapping;
};

/* address_space: a page cache of a file */
struct address_space {
    struct inode      *host;        /* owner: inode, block_device */
    struct radix_tree_root page_tree; /* radix tree of all pages
                                         * (i.e., page cache of an inode) */
    spinlock_t         tree_lock;   /* and lock protecting it */
};
```

Linux radix tree example

- Shared memory virtual file system
 - shared memory among process (`shmget()` and `shmat()`)
 - `tmpfs` memory file system

```
/* linux/fs/inode.c */
/* page_tree is initialized at associated address_space is initialized */
void address_space_init_once(struct address_space *mapping)
{
    INIT_RADIX_TREE(&mapping->page_tree, GFP_ATOMIC | __GFP_ACCOUNT);
}

/* linux/mm/shmem.c */
/* Radix operations are performed on page_tree for file system operations */
static int shmem_add_to_page_cache(struct page *page,
        struct address_space *mapping, pgoff_t index, void *expected)
{
    error = radix_tree_insert(&mapping->page_tree, index, page);
}
```

XArray

- A nicer API wrapper for linux radix tree (merged to 4.19)
- An automatically resizing array of pointers indexed by an unsigned long
- Entries may have up to three tag bits (get/set/clear)
- You can iterate over entries
- You can extract a batch of entries
- Embeds a spinlock
- Loads are store-free using RCU

XArray API

```
#include <linux/xarray.h>

/** Define an XArray */
DEFINE_XARRAY(array_name);
/* or */
struct xarray array;
xa_init(&array);

/** Storing a value into an XArray is done with: */
void *xa_store(struct xarray *xa, unsigned long index, void *entry,
               gfp_t gfp);

/** An entry can be removed by calling: */
void *xa_erase(struct xarray *xa, unsigned long index);

/** Storing a value only if the current value stored there matches old: */
void *xa_cmpxchg(struct xarray *xa, unsigned long index, void *old,
                  void *entry, gfp_t gfp);
```

XArray API

```
/** Fetching a value from an XArray is done with xa_load(): */
void *xa_load(struct xarray *xa, unsigned long index);

/** Up to three single-bit tags can be set on any non-null XArray
entry; they are managed with: */
void xa_set_tag(struct xarray *xa, unsigned long index, xa_tag_t tag);
void xa_clear_tag(struct xarray *xa, unsigned long index, xa_tag_t tag);
bool xa_get_tag(struct xarray *xa, unsigned long index, xa_tag_t tag);

/** Iterate over present entries in an XArray: */
xa_for_each(xa, index, entry) {
    /* Process "entry" */
}

/** Iterate over marked entries in an XArray: */
xa_for_each_marked(xa, index, entry, filter) {
    /* Process "entry" which marked with "filter" */
}
```

Linux XArray example (v5.8)

Linux bitmap

- A bit array that consumes one or more `unsigned long`
- Using in many places in kernel
 - a set of online/offline processors for systems which support hot-plug cpu (more about this you can read in the cpumasks part)
 - a set of allocated IRQs during initialization of the Linux kernel

Linux bitmap

```
/* linux/include/linux(bitmap.h
 * linux/lib(bitmap.c
 * arch/x86/include/asm/bitops.h */

/* Declare an array named 'name' of just enough unsigned longs to
 * contain all bit positions from 0 to 'bits' - 1 */
#define DECLARE_BITMAP(name,bits) \
    unsigned long name[BITS_TO_LONGS(bits)]

/* set_bit - Atomically set a bit in memory
 * @nr: the bit to set
 * @addr: the address to start counting from */
void set_bit(long nr, volatile unsigned long *addr);
void clear_bit(long nr, volatile unsigned long *addr);
void change_bit(long nr, volatile unsigned long *addr);

/* clear nbits from dst */
void bitmap_zero(unsigned long *dst, unsigned int nbits);
void bitmap_fill(unsigned long *dst, unsigned int nbits);
```

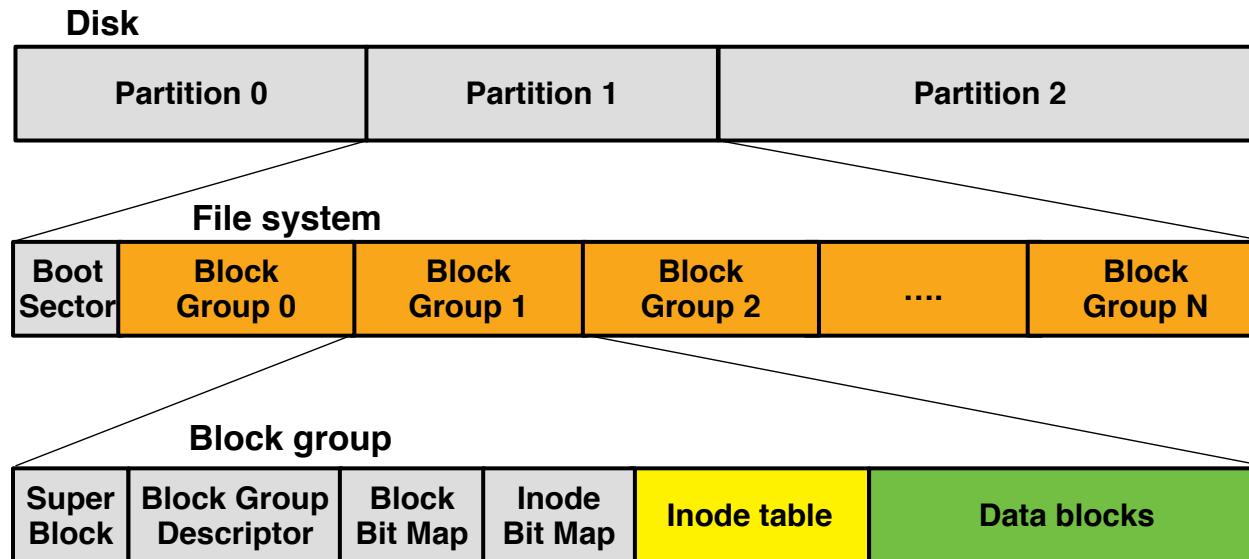
Linux bitmap

```
/* find_first_bit - find the first set bit in a memory region
 * @addr: The address to start the search at
 * @size: The maximum number of bits to search
 *
 * Returns the bit number of the first set bit.
 * If no bits are set, returns @size.
 */
unsigned long find_first_bit(const unsigned long *addr, unsigned long size);
unsigned long find_first_zero_bit(const unsigned long *addr, unsigned long size);

/* iterate bitmap */
#define for_each_set_bit(bit, addr, size) \
    for ((bit) = find_first_bit((addr), (size));          \
          (bit) < (size);                                \
          (bit) = find_next_bit((addr), (size), (bit) + 1))
#define for_each_set_bit_from(bit, addr, size) ...
#define for_each_clear_bit(bit, addr, size) ...
#define for_each_clear_bit_from(bit, addr, size) ...
```

Linux bitmap example

- Free inode/disk block management in ext2/3/4 file system



Kernel modules

- Modules are pieces of kernel code that can be **dynamically loaded and unloaded at runtime** → No need to reboot
- Appeared in Linux 1.2 (1995)
- Numerous Linux features can be compiled as modules
 - Selection in the configuration .config file

```
# linux/.config
# CONFIG_XEN_PV is not set
CONFIG_KVM_GUEST=y  # built-in to kernel binary executable, vmlinux
CONFIG_XFS_FS=m      # kernel module
```

Benefit of kernel modules

- No reboot → saves a lot of time when developing/debugging
- No need to compile the entire kernel
- Saves memory and CPU time by running on-demand
- No performance difference between module and built-in kernel code
- Help identifying buggy code
 - E.g., identifying a buggy driver compiled as a module by selectively running them

Writing a kernel module

- Module is linked against the entire kernel
- Module can access all of the kernel global symbols
 - `EXPORT_SYMBOL(function or variable name)`
- To avoid namespace pollution and involuntary reuse of variables names
 - Put prefix of your module name to symbols:
`my_module_func_a()`
 - Use `static` if a symbol is not global
- Kernel symbols list are at `/proc/kallsyms`

Writing a kernel module

```
#include <linux/module.h>      /* Needed by all modules */
#include <linux/kernel.h>      /* KERN_INFO */
#include <linux/init.h>        /* Init and exit macros */

static int answer = 42;

static int __init lkp_init(void)
{
    printk(KERN_INFO "Module loaded ...\\n");
    printk(KERN_INFO "The answer is %d ...\\n", answer);
    return 0; /* return 0 on success, something else on error */
}

static void __exit lkp_exit(void)
{
    printk(KERN_INFO "Module exiting ...\\n");
}

module_init(lkp_init); /* lkp_init() will be called at loading the module */
module_exit(lkp_exit); /* lkp_exit() will be called at unloading the module */

MODULE_LICENSE("GPL");
MODULE_AUTHOR("Dongyoon Lee <dongyoon@cs.stonybrook.edu>");
MODULE_DESCRIPTION("Sample kernel module");
```

Building a kernel module

- Source code of a module is out of the kernel source
- Put a Makefile in the module source directory
- After compilation, the compiled module is the file with `.ko` extension

```
# let's assume the module C file is named lkp.c
obj-m := lkp.o
# obj-m += lkp2.o # add multiple files if necessary

CONFIG_MODULE_SIG=n
KDIR := /path/to/kernel/sources/root/directory
# KDIR := /lib/modules/$(shell uname -r)/build
PWD := $(shell pwd)

all: lkp.c # add lkp2.c if necessary
    make -C $(KDIR) M=$(PWD) modules

clean:
    make -C $(KDIR) M=$(PWD) clean
```

Launching a kernel module

- Needs `root` privileges because you are executing kernel code!
- Loading a kernel module with `insmod`
 - `sudo insmod file.ko`
 - Module is loaded and init function is executed
- Note that a module is compiled against a specific kernel version and will not load on another kernel
 - This check can be bypassed through a mechanism called `modversions` but it can be dangerous

Launching a kernel module

- Remove the module with `rmmmod`
 - `sudo rmmod file`
 - or `sudo rmmod file.ko`
 - Module exit function is called before unloading
- `make modules_install` from the kernel sources installs the modules in a standard location
 - `/lib/modules/<kernel version>/`

Launching a kernel module

- These installed modules can be loaded using `modprobe`
 - `sudo modprobe <module name>` ← no need to give a file name
- Contrary to `insmod`, `modprobe` handles module dependencies
 - Dependency list generated in `/lib/modules/<kernel version>/modules.dep`
- Unload a module using `modprobe -r <module name>`
- Such installed modules can be loaded automatically at boot time by editing `/etc/modules` or the files in `/etc/modprobe.d`

Module parameters ~= command line arguments for module

```
#include <linux/module.h>
/* ... */
static int int_param = 42; /* default value */
static char *string_param = "default value";

module_param(int_param, int, 0);
MODULE_PARM_DESC(int_param, "A sample integer kernel module parameter");
module_param(string_param, charp, S_IRUSR | S_IWUSR | S_IRGRP | S_IROTH);
MODULE_PARM_DESC(string_param, "Another parameter, a string");

static int __init lkp_init(void)
{
    printk(KERN_INFO "Int param: %d\n", int_param);
    printk(KERN_INFO "String param: %s \n", string_param);
    /* ... */
}
```

- sudo insmod lkp.ko int_param=12 string_param="hello"

Getting module information

- `modinfo [module name | file name]`

```
modinfo my_module.ko
filename:      /tmp/test/my_module.ko
description:  Sample kernel module
author:       Dongyoon Lee <dongyoon@cs.stonybrook.edu>
license:      GPL
srcversion:   A5ADE92B1C81DCC4F774A37
depends:
vermagic:    4.8.0-34-generic SMP mod_unload modversions
parm:        int_param:A sample integer kernel module parameter (int)
parm:        string_param:Another parameter, a string (charp)
```

- `lsmod` : list currently running modules

Further readings

- [LWN: Trees I: Radix trees](#)
- [Bit arrays and bit operations in the Linux kernel](#)
- [LWN: The XArray data structure](#)
- [XArray API](#)
- [The design and implementation of the XArray](#)
- [LWN: How to get rid of mmap_sem](#)
- [2.6. Passing Command Line Arguments to a Module](#)
- [Building External Modules](#)

Next lecture

- Kernel debugging techniques

Kernel debugging

Dongyoон Lee

Summary of last lectures

- Kernel data structures
 - list, hash table, red-black tree
 - radix tree, XArray, bitmap array
- Kernel module

Today's agenda

- Kernel debugging
 - tools, techniques, and tricks

Kernel development cycle

- Write code → Build kernel/modules → Deploy → **Test and debug**
- *Debugging is the real bottleneck* even for experienced kernel developers due to limitations in kernel debugging
- It is important to get used to kernel debugging techniques to save your time and effort

Kernel debugging techniques

- Print debug message: `printf`
- Assert your code: `BUG_ON(c)` , `WARN_ON(c)`
- Analyze kernel panic message
- Debug with QEMU/gdb

Print debug message: `printf`

- Similar to `printf()` in C library
- Need to specify a log level (the default level is `KERN_WARNING` or `KERN_ERR`)

```
KERN_EMERG    /* 0: system is unusable          */
KERN_ALERT     /* 1: action must be taken immediately   */
KERN_CRIT      /* 2: critical conditions                 */
KERN_ERR        /* 3: error conditions                   */
KERN_WARNING    /* 4: warning conditions                 */
KERN_NOTICE     /* 5: normal but significant condition   */
KERN_INFO       /* 6: informational                      */
KERN_DEBUG      /* 7: debug-level messages               */
```

e.g., `printf(KERN_DEBUG "debug message from %s:%d\n", __func__, __LINE__);`

Print debug message: `printf`

- Prints out only messages, which log level is higher than the current level.

```
# Check current kernel log level
$ cat /proc/sys/kernel/printk
    4      4      1      7
#   |      |      |      |
# current default minimum boot-time-default
# Enable all levels of messages:
$ echo 7 > /proc/sys/kernel/printk
```

- The kernel message buffer is a fixed-size circular buffer.
- If the buffer fills up, it warps around and *you can lose some message*.
- Increasing the buffer size would be helpful a little bit.
 - Add `log_buf_len=1M` to kernel boot parameters (power of 2)

Print debug message: `printf`

- Support additional format specifiers

```
/* function pointers with function name */
"%pF"      versatile_init+0x0/0x110 /* symbol+offset/length */
"%pf"      versatile_init

/* direct code address (e.g., regs->ip) */
"%pS"      versatile_init+0x0/0x110
"%ps"      versatile_init

/* direct code address in stack (e.g., return address) */
"%pB"      prev_fn_of_versatile_init+0x88/0x88

/* Example */
printf("Going to call: %pF\n", p->func);
printf("Faulted at %pS\n", (void *)regs->ip);
printf(" %s%pB\n", (reliable ? "" : "? "), (void *)*stack);
```

- Ref: [How to get printf format specifiers right](#)

BUG_ON(c), WARN_ON(c)

- Similar to `assert(c)` in userspace
- `BUG_ON(c)`
 - if `c` is false, kernel panics with its call stack
- `WARN_ON(c)`
 - if `c` is false, kernel prints out its call stack and keeps running

Kernel panic message

```
[ 174.507084] Stack:  
[ 174.507163]  ce0bd8ac 00000008 00000000 ce4a7e90 c039ce30 ce0bd8ac c0718b04 c07185a0  
[ 174.507380]  ce4a7ea0 c0398f22 ce0bd8ac c0718b04 ce4a7eb0 c037deee ce0bd8e0 ce0bd8ac  
[ 174.507597]  ce4a7ec0 c037dfe0 c07185a0 ce0bd8ac ce4a7ed4 c037d353 ce0bd8ac ce0bd8ac  
[ 174.507888] Call Trace:  
[ 174.508125]  [<c039ce30>] ? sd_remove+0x20/0x70  
[ 174.508235]  [<c0398f22>] ? scsi_bus_remove+0x32/0x40  
[ 174.508326]  [<c037deee>] ? __device_release_driver+0x3e/0x70  
[ 174.508421]  [<c037dfe0>] ? device_release_driver+0x20/0x40  
[ 174.508514]  [<c037d353>] ? bus_remove_device+0x73/0x90  
[ 174.508606]  [<c037bccf>] ? device_del+0xef/0x150  
[ 174.508693]  [<c0399207>] ? __scsi_remove_device+0x47/0x80  
[ 174.508786]  [<c0399262>] ? scsi_remove_device+0x22/0x40  
[ 174.508877]  [<c0399324>] ? __scsi_remove_target+0x94/0xd0  
[ 174.508969]  [<c03993c0>] ? __remove_child+0x0/0x20  
[ 174.509060]  [<c03993d7>] ? __remove_child+0x17/0x20  
[ 174.509148]  [<c037b868>] ? device_for_each_child+0x38/0x60
```

- Q: How can I find where `sd_remove+0x20/0x70` is at source code?

Analyze kernel panic message

1. Find where `sd_remove()` is (e.g., `linux/driver/scsi/sd.c`)
2. Load its object file with `gdb`
3. Use `gdb` command, `list *(function+0xoffset)` command

```
manjo@hungry:~/devel/kernel/build/build-generic/drivers/scsi$ gdb sd.o
This GDB was configured as "x86_64-linux-gnu".
(gdb) list *(sd_remove+0x20)
0x1650 is in sd_remove (/home/manjo/devel/linux/drivers/scsi/sd.c:2125).
2120     static int sd_remove(struct device *dev)
2121 {
2122         struct scsi_disk *sdkp;
2123
2124         async_synchronize_full();
```

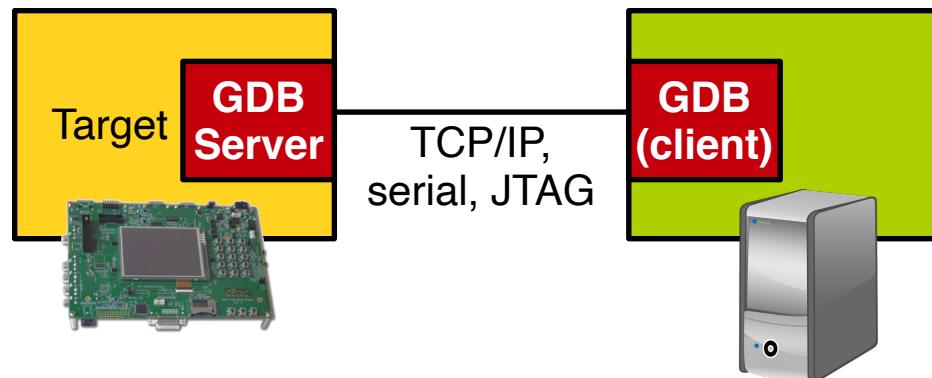
- **Q: Can I debug kernel using `gdb`? → It is possible using QEMU/gdb**

QEMU

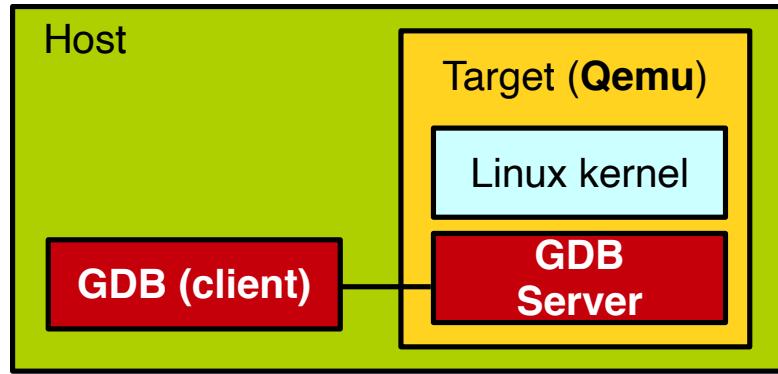
- Full system emulator: emulates an entire virtual machine
 - Using a software model for the CPU, memory, devices
 - Emulation is slow
- Can also be used in conjunction with hardware virtualization extensions to provide high performance virtualization
 - **KVM:** In-kernel support for virtualization + extensions to QEMU

GDB server

- Originally used to debug a program executing on a remote machine
- For example when GDB is not available on that remote machine
 - E.g., low performance embedded systems



Kernel debugging with QEMU/gdb



- Linux kernel runs in a virtual machine (KVM or emulated on QEMU)
- Hardware devices are emulated with QEMU
- GDB server runs at QEMU, emulated virtual machine, so it can fully control Linux kernel running on QEMU
- *It is fantastic for debugging and code exploration!*

Two ways for kernel debugging with QEMU/gdb

1. Running minimal Linux distribution

- Use a `debootstrap`-ed distribution
- Root file system is a directory in a host system
- Limited function in userspace applications

2. Running full Linux distribution

- Use a QEMU disk image (qcow2 or raw disk)
- Root file system is on the disk image
- Able to run full userspace applications

Build kernel for QEMU/gdb debugging

- Rebuild kernel with gdb script, 9p, and virtio enabled
- Following should be built-in not built as a kernel module.

```
$ cat .config
CONFIG_DEBUG_INFO=y      # debug symbol
CONFIG_GDB_SCRIPTS=y    # qemu/gdb support
CONFIG_E1000=y          # default network card

CONFIG_VIRTIO=y          # file sharing with host
CONFIG_NET_9P=y          # file sharing with host
CONFIG_NET_9P_VIRTIO=y   # file sharing with host
CONFIG_9P_FS=y           # file sharing with host
CONFIG_9P_FS_POSIX_ACL=y # file sharing with host
CONFIG_9P_FS_SECURITY=y  # file sharing with host
```

Build kernel for QEMU/gdb debugging

```
# or (use `'/ GDB_SCRIPTS` in `make menuconfig`
$ make menuconfig
    | Prompt: Provide GDB scripts for kernel debugging
    | Location:
    |     -> Kernel hacking
    |         -> Compile-time checks and compiler options
    |             -> Provide GDB scripts for kernel debugging
    |                 Compile the kernel with frame pointers

# then build the kernel
$ make -j8; make -j8 modules
```

- You don't need `make modules_install; make install` because all necessary features will be built-in to ease of deployment.

Debootstrap Linux distribution

- debootstrap.sh

```
#!/bin/bash

# debootstrap the latest, minimal debian linux
sudo debootstrap --no-check-gpg --arch=amd64 \
    --include=kmod,net-tools,iproute2,iputils-ping \
    --variant=minbase \ # minimal base packages
    focal \           # Ubuntu 20.04 or 'sid' for unstable (debian) distribution
    linux-chroot     # target directory

# copy the minimal initialization code to the target directory
sudo cp start.sh linux-chroot/

# allow gdb to load gdb script files
echo 'set auto-load safe-path ~/' > ~/.gdbinit
```

Run kernel with QEMU/gdb

- run-qemu-chroot.sh

```
#!/bin/bash
VMLINUX=~/.workspace/research/linux
VMLINUX=${VMLINUX}/arch/x86_64/boot/bzImage
ROOT_DIR=${PWD}/linux-chroot
sudo qemu-system-x86_64 -s \
    -S \
    -nographic \
    -fsdev local,id=root,path=${ROOT_DIR},security_model=passthrough \
    # set 9P file system
    -device virtio-9p-pci,fsdev=root,mount_tag=/dev/root \
    # use 9P file system as a boot device
    -append 'nokaslr root=/dev/root rw rootfstype=9p \
              rootflags=trans=virtio console=ttyS0 init=/bin/bash'
    # replace init process to /bin/bash
    -kernel ${VMLINUX} \
    # kernel binary
    -smp cpus=2 \
    # set num of CPUs
    -m 2G \
    # set memory size

# Ctrl-a x: terminating QEMU
```

Run kernel with QEMU/gdb

- QEMU options
 - `-kernel vmlinux` : path to the vmlinux of the kernel to debug
 - `-s` : enable the GDB server and open a port 1234
 - `-S` : (optional) pause on the first kernel instruction waiting for a GDB client connection to continue

Connect to the kernel running on QEMU/gdb

```
$ cd /path/to/linux-build  
$ gdb vmlinuz  
(gdb) target remote :1234
```

- You can use all *gdb commands* and *Linux-provided gdb helpers!*

- [b]reak <function name or filename:line# or *memory address>
- [hbreak] <start_kernel or any function name> # to debug boot code
- [d]elete <breakpoint #>
- [c]ontinue
- [b]ack[t]race
- [i]nfo [b]reak
- [n]ext
- [s]tep
- [p]rint <variable or *memory address>
- Ctrl-x Ctrl-a: TUI mode

Linux-provided gdb helpers

- Load module and main kernel symbols

```
(gdb) lx-symbols
loading vmlinux
scanning for modules in /home/user/linux/build
loading @0xfffffffffa002000: /home/user/linux/build/net/netfilter/xt_tcpudp.ko
loading @0xfffffffffa001600: /home/user/linux/build/net/netfilter/xt_pktype.ko
loading @0xfffffffffa000200: /home/user/linux/build/net/netfilter/xt_limit.ko
loading @0xfffffffffa00ca00: /home/user/linux/build/net/packet/af_packet.ko
loading @0xfffffffffa003c00: /home/user/linux/build/fs/fuse/fuse.ko
...
loading @0xfffffffffa000000:
/home/user/linux/build/drivers/ata/ata_generic.ko
```

Linux-provided gdb helpers

- Set a breakpoint on some not yet loaded module function, e.g.:

```
(gdb) b btrfs_init_sysfs
Function "btrfs_init_sysfs" not defined.
Make breakpoint pending on future shared library load? (y or [n]) y
Breakpoint 1 (btrfs_init_sysfs) pending.
```

- Continue the target:

```
(gdb) c
loading @0xfffffffffa006e000: /home/user/linux/build/lib/zlib_deflate/zlib_deflate.ko
loading @0xfffffffffa01b1000: /home/user/linux/build/fs/btrfs/btrfs.ko

Breakpoint 1, btrfs_init_sysfs () at /home/user/linux/fs/btrfs/sysfs.c:36
36          btrfs_kset = kset_create_and_add("btrfs", NULL, fs_kobj);
```

Linux-provided gdb helpers

- Dump the log buffer of the target kernel:

```
(gdb) lx-dmesg
[    0.00000] Initializing cgroup subsys cpuset
[    0.00000] Initializing cgroup subsys cpu
[    0.00000] Linux version 3.8.0-rc4-dbg+ (...
[    0.00000] Command line: root=/dev/sda2 resume=/dev/sda1 vga=0x314
[    0.00000] e820: BIOS-provided physical RAM map:
[    0.00000] BIOS-e820: [mem 0x0000000000000000-0x000000000009fbff] usable
[    0.00000] BIOS-e820: [mem 0x000000000009fc00-0x000000000009ffff] reserved
```

- Examine fields of the current task struct:

```
(gdb) p $lx_current().pid
$1 = 4998
```

Linux-provided gdb helpers

- Help

```
(gdb) apropos lx
function lx_current -- Return current task
function lx_module -- Find module by name and return the module variable
function lx_per_cpu -- Return per-cpu variable
function lx_task_by_pid -- Find Linux task by PID and return the task_struct variable
function lx_thread_info -- Calculate Linux thread_info from task variable
lx-dmesg -- Print Linux kernel log buffer
lx-lsmod -- List currently loaded modules
lx-symbols -- (Re-)load symbols of Linux kernel and currently loaded
modules
```

Tips for QEMU-gdb kernel debugging

- *(gdb) p my_var → \$1 = <value optimized out>*
 - `my_var` is optimized out
 - Since it is not possible to disable optimization for the entire kernel, we need to disable optimization for a specific file.

```
# linux/fs/ext4/Makefile
obj-$(CONFIG_EXT4_FS) += ext4.o

CFLAGS_bitmap.o = -O0  # disable optimization of bitmap.c

ext4-y := balloco.o bitmap.o dir.o file.o \
#...
```

Tips for QEMU-gdb kernel debugging

- Cursor disappears in qemu window?
 - Ctrl Alt (right)
- Always terminates QEMU with the `halt` command otherwise the disk image could be corrupted
- QEMU is too slow
 - Try KVM (`enable-kvm`)
 - It works only when your host is Linux.

Two ways for kernel debugging with QEMU/gdb

1. Running minimal Linux distribution

- Use a `debootstrap`-ed distribution
- Root file system is a directory in a host system
- Limited function in userspace applications

2. Running full Linux distribution

- Use a QEMU disk image (qcow2 or raw disk)
- Root file system is on the disk image
- Able to run full userspace applications

Build kernel for QEMU/gdb debugging

```
$ cat .config
# Compile-time checks and compiler options
#
CONFIG_DEBUG_INFO=y
CONFIG_GDB_SCRIPTS=y

# or (use `/ GDB_SCRIPTS` in `make menuconfig`
$ make menuconfig
    | Prompt: Provide GDB scripts for kernel debugging
    | Location:
    |     -> Kernel hacking
    |         -> Compile-time checks and compiler options
    |             -> Provide GDB scripts for kernel debugging
    |                 Compile the kernel with frame pointers

# then build the kernel
$ make -j8; make -j8 modules
```

Run kernel with QEMU/gdb

- run-qemu-qcow2.sh

```
#!/bin/bash
VMLINUX=~/.workspace/research/linux
VMLINUX=${VMLINUX}/arch/x86_64/boot/bzImage
VMIMAGE=${PWD}/linux-vm.qcow2
sudo qemu-system-x86_64 -s \
    -nographic \
    -hda ${VMIMAGE} \
    -kernel ${VMLINUX} \
    \          # boot parameter: no KASLR, set HDD, enable serial console
    -append "nokaslr root=/dev/sda1 console=ttyS0" \
    -smp cpus=2 \
    -device e1000,netdev=net0 \
        # forward TCP:5555 to 22 for ssh
    -netdev user,id=net0,hostfwd=tcp::5555-:22 \
    -m 2G \
        # set memory size

# Converting vmdk to qcow2:
#   qemu-img convert -O qcow2 linux-vm.vmdk linux-vm.qcow2
# Ctrl-a x: terminating QEMU
```

Further readings

- [Debugging by printing](#)
- [Kernel Debugging Tricks](#)
- [Kernel Debugging Tips](#)
- [Debugging kernel and modules via gdb](#)
- [gdb Cheatsheet](#)
- [Speed up your kernel development cycle with QEMU](#)
- [Installing new Debian systems with debootstrap](#)
- [Migrate a VirtualBox Disk Image \(.vdi\) to a QEMU Image \(.img\)](#)
- [The kernel's command-line parameters](#)

Next lecture

- Process management

Process Management

Dongyoon Lee

Summary of last lectures

- Getting, building, and exploring the Linux kernel
- System call: interface between applications and kernel
- Kernel data structures
- Kernel modules
- Kernel debugging techniques

Today's agenda

- Process management in Linux kernel
 - Process
 - The process descriptor: `task_struct`
 - Process creation
 - Threads
 - Process termination

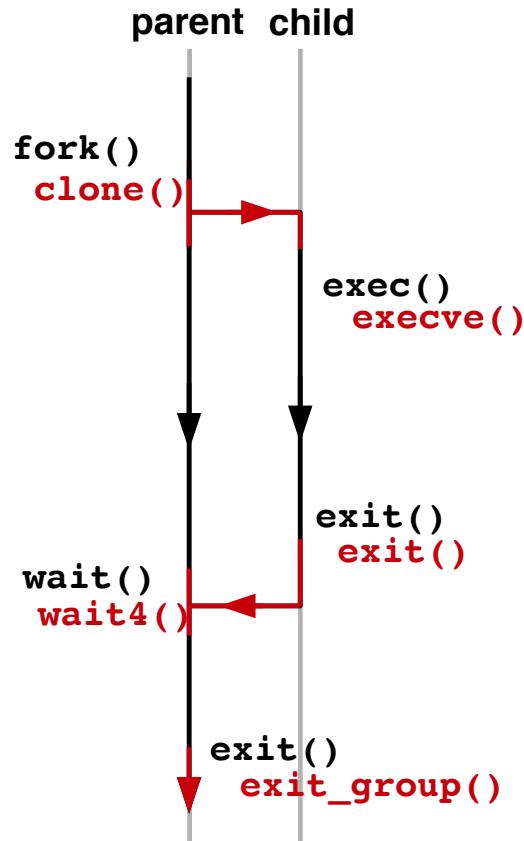
Process

- A program currently executing in the system
- A process is composed of
 - CPU registers
 - program code (i.e., *text section*)
 - state of memory segments (data, stack, etc)
 - kernel resources (open files, pending signals, etc)
 - threads
- Virtualization of processor and memory

Process from an user-space view

- `pid_t fork(void)`
 - creates a new process by duplicating the calling process
- `int execv(const char *path, const char *arg, ...)`
 - replaces the current process image with a new process image
- `pid_t wait(int *wstatus)`
 - wait for state changes in a child of the calling process
 - the child terminated; the child was stopped or resumed by a signal

Process from an user-space view



fork() example

```
int main(void)
{
    pid_t pid;
    int wstatus, ret;
    pid = fork(); /* create a child process */
    switch(pid) {
        case -1: /* fork error */
            perror("fork");
            return EXIT_FAILURE;
        case 0:   /* pid = 0: new born child process */
            sleep(1);
            printf("Noooooooo!\n");
            exit(99);
        default: /* pid = pid of child: parent process */
            printf("I am your father!: your pid is %d\n", pid);
            break;
    }
    /* A parent wait until the child terminates */
    ret = waitpid(pid, &wstatus, 0);
    if(ret == -1)
        return EXIT_FAILURE;
    printf("Child exit status: %d\n", WEXITSTATUS(wstatus));
    return EXIT_SUCCESS;
}
```

Let's check this example using `strace`

```
$ strace -f ./process
execve("./process", ["./process"], 0x7ffffb44f068 /* 64 vars */) = 0
...
clone(child_stack=NULL, flags=CLONE_CHILD_CLEARTID|CLONE_CHILD_SETTID|SIGCHLD, child_tidptr=0x
strace: Process 16888 attached
[pid 16887] fstat(1, {st_mode=S_IFCHR|0620, st_rdev=makedev(136, 3), ...}) = 0
[pid 16888] nanosleep({tv_sec=1, tv_nsec=0}, <unfinished ...>
[pid 16887] brk(NULL)                      = 0x164e000
[pid 16887] brk(0x166f000)                  = 0x166f000
[pid 16887] brk(NULL)                      = 0x166f000
[pid 16887] write(1, "I am your father!\n", 18I am your father!) = 18
[pid 16887] wait4(16888, <unfinished ...>
[pid 16888] <... nanosleep resumed> 0x7fffe1b4500) = 0
[pid 16888] fstat(1, {st_mode=S_IFCHR|0620, st_rdev=makedev(136, 3), ...}) = 0
[pid 16888] brk(NULL)                      = 0x164e000
[pid 16888] brk(0x166f000)                  = 0x166f000
[pid 16888] brk(NULL)                      = 0x166f000
[pid 16888] write(1, "Nooooooooo!\n", 11Nooooooooo!) = 11
[pid 16888] exit_group(0)                  = ?
[pid 16888] +++ exited with 0 +++
<... wait4 resumed> [{WIFEXITED(s) && WEXITSTATUS(s) == 0}], 0, NULL) = 16888
--- SIGCHLD {si_signo=SIGHLD, si_code=CLD_EXITED, si_pid=16888, si_uid=1000, si_status=0, si_
write(1, "Child exit status: 0\n", 21Child exit status: 0)  = 21
```

Processor descriptor: **task_struct**

```
/* linux/include/linux/sched.h */
struct task_struct {
    struct thread_info    thread_info; /* thread information */
    volatile long          state;        /* task status: TASK_RUNNING, etc */
    void                  *stack;       /* stack of this task */

    int                   prio;        /* task priority */
    struct sched_entity   se;         /* information for processor scheduler */
    cpumask_t             cpus_allowed; /* bitmask of CPUs allowed to execute */

    struct list_head      tasks;       /* a global task list */
    struct mm_struct      *mm;        /* memory mapping of this task */

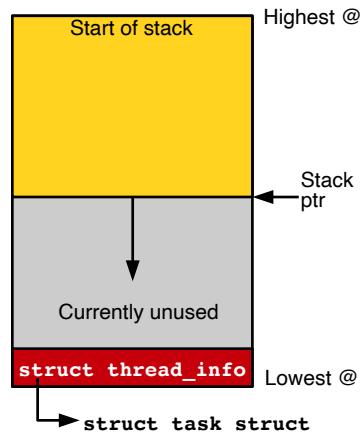
    struct task_struct    *parent;     /* parent task */
    struct list_head      children;   /* a list of child tasks */
    struct list_head      sibling;    /* siblings of the same parent */

    struct files_struct   *files;      /* open file information */
    struct signal_struct  *signal;    /* signal handlers */
    /* ... */

    /* NOTE: In Linux kernel, process and task are interchangably used. */
};
```

Processor descriptor: `task_struct`

- In old kernels, `task_struct` (or `thread_info` until v4.9) is allocated at the bottom of the kernel stack of each process
 - Getting current `task_struct` is just masking out the 13 least-significant bits the stack pointer



Processor descriptor: **task_struct**

- Since v4.9, `task_struct` is dynamically allocated at heap because of potential exploit when overflowing the kernel stack
- For efficient access of current `task_struct`, kernel maintains per-CPU variable, named `current_task`
 - Use `current` to get `current_task`

```
/* linux/arch/x86/include/asm/current.h */

DECLARE_PER_CPU(struct task_struct *, current_task);
static __always_inline struct task_struct *get_current(void)
{
    return this_cpu_read_stable(current_task);
}

#define current get_current()
```

Process Identifier (PID): **pid_t**

- Maximum is 32768 (int)
- Can be increased to 4 millions
- Wraps around when maximum reached

Process status: **task->state**

- **TASK_RUNNING**
 - A task is runnable (running or in a per-CPU scheduler run queue)
 - A task could be in user- or kernel-space

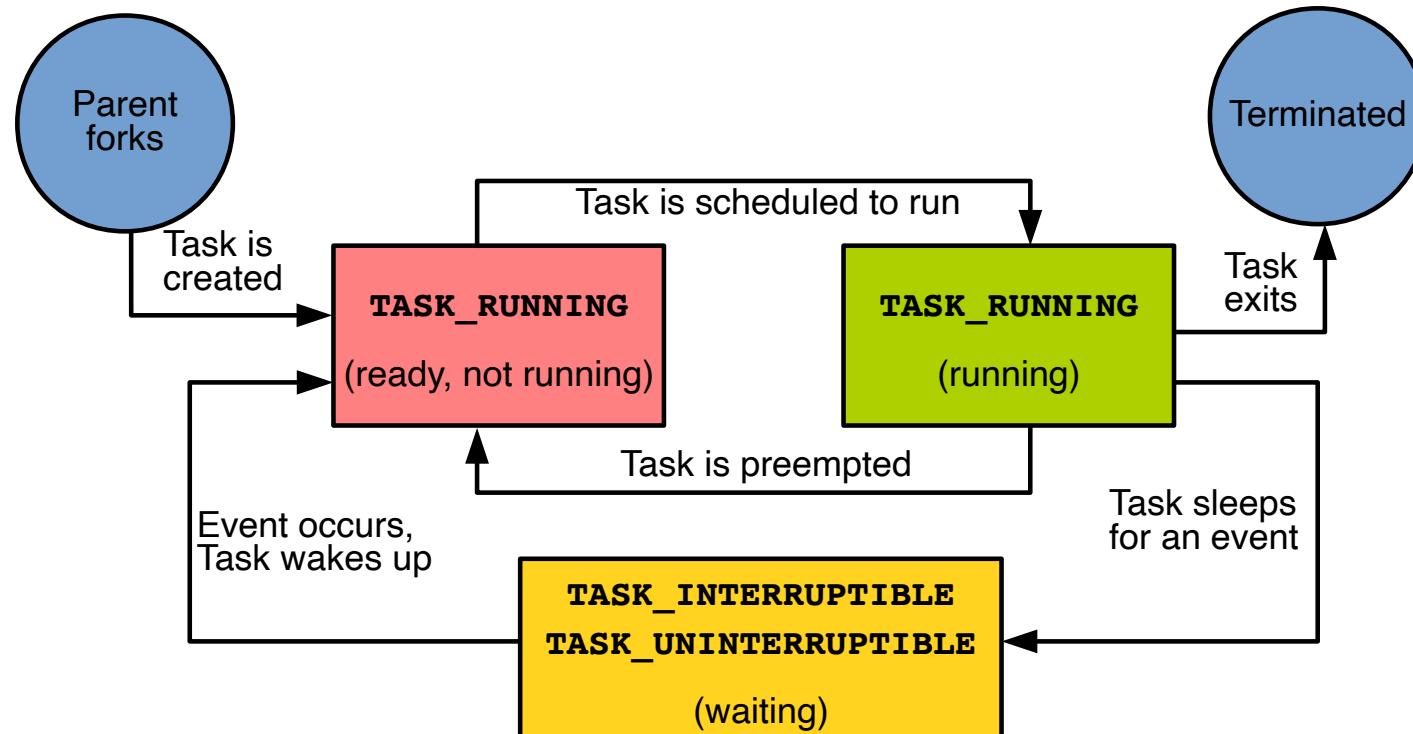
Process status: task->state

- `TASK_INTERRUPTIBLE`
 - Process is sleeping waiting for some condition
 - Switched to `TASK_RUNNING` when the waiting condition becomes true or a signal is received
- `TASK_UNINTERRUPTIBLE`
 - Same as `TASK_INTERRUPTIBLE` but does not wake up on signal

Process status: `task->state`

- `__TASK_TRACED`
 - Traced by another process (ex: debugger)
- `__TASK_STOPPED`
 - Not running nor waiting, result of the reception of some signals (e.g.,
`SIGSTOP`) to pause the process

Process status: task->state



Producer-consumer example

- Producer
 - generate an event and wake up a consumer
- Consumer
 - check if there is an event
 - if so, process all pending event in the list
 - otherwise, sleep until the producer wakes me up

Sleeping in the kernel

Producer task:

```
001 spin_lock(&list_lock);
002 list_add_tail(&list_head, new_event); /* append an event to the list */
003 spin_unlock(&list_lock);
004 wake_up_process(consumer_task); /* and wake up the consumer task */
```

Consumer task:

```
100 set_current_state(TASK_INTERRUPTIBLE); /* set status to TASK_INTERRUPTIBLE */
101 spin_lock(&list_lock);
102 if(list_empty(&list_head)) { /* if there is no item in the list */
103     spin_unlock(&list_lock);
104     schedule(); /* sleep until the producer task wakes this */
105     spin_lock(&list_lock); /* this task is waken up by the producer */
106 }
107 set_current_state(TASK_RUNNING); /* change status to TASK_RUNNING */
108
109 list_for_each(pos, list_head) {
110     list_del(&pos)
111     /* process an item */
112     /* ... */
113 }
114 spin_unlock(&list_lock);
```

Process context

- The kernel can executes in a **process context** or **interrupt context**
 - `current` is meaningful only when the kernel executes in a process context such as executing a system call
 - Interrupt has its own context

Process family tree

- `init` process is the root of all processes
 - Launched by the kernel as the last step of the boot process
 - Reads the system `initscripts` and executes more programs, such as daemons, eventually completing the boot process
 - Its `PID` is 1
 - Its `task_struct` is a global variable, named `init_task` (`linux/init/init_task.c`)

Let's check process tree using `pstree`

```
21:15 $ pstree
init─ apache2─ 2*[apache2─ 26*[{apache2}]</mark>
      └─collectl
      └─cron
      └─dbus-daemon
      └─6*[getty]
      └─irqbalance
      └─lxcfs─ 6*[{lxcfs}]
      └─mdadm
      └─memcached─ 5*[{memcached}]
      └─mosh-server─ bash─ tmux: client
      └─mpssd─ 10*[{mpssd}]
      └─netserver
      └─nullmailer-send─ smtp
      └─rpc.idmapd
      └─rpc.mountd
      └─rpc.statd
      └─rpcbind
      └─rsyslogd─ 3*[{rsyslogd}]
      └─sshd─ sshd─ sshd─ bash─ pstree
      └─systemd-logind
      └─systemd-udevd
      └─tmux: server─ bash─ vim─ bash
```

Process family tree

- `fork`-based process creation
 - my parent task: `current->parent`
 - my children tasks: `current->children`
 - siblings under the parent: `current->siblings`
 - list of all tasks in the system: `current->tasks`
 - macros to easy to explore:
 - `next_task(t)`, `for_each_process(t)`
- Let's check how these macros are implemented

Process creation

- Linux does not implement creating tasks from nothing (`spawn` or `CreateProcess`)
- `fork()` and `exec()`
 - `fork()` creates a child, copy of the parent process
 - Only PID, PPID and some resources/stats differ
 - `exec()` loads into a process address space a new executable

Copy-on-Write (CoW)

- On `fork()`, Linux duplicates the parent page tables and creates a new process descriptor
 - Change page table access bits to *read-only*
 - When a page is accessed *for write operations*, that page is copied and the corresponding page table entry is changed to *read-write*
- `fork()` is fast by delaying or altogether preventing copying of data
- `fork()` saves memory by sharing read-only pages among descendants

Forking

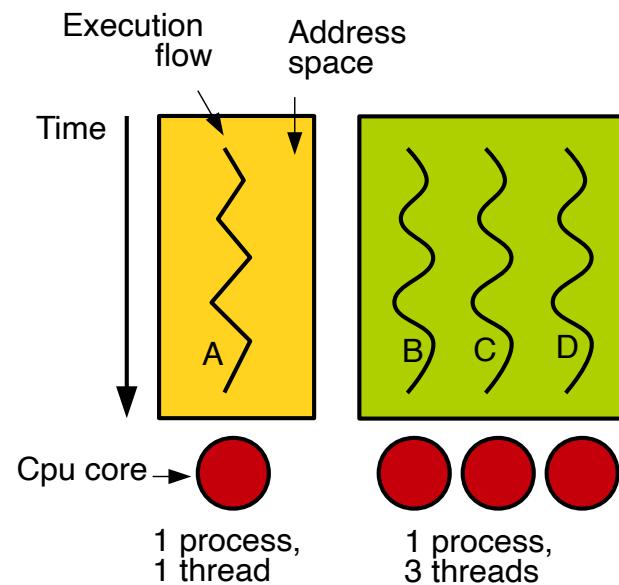
- `fork()` is implemented by the `clone()` system call
- `sys_clone()` calls `do_fork()`, which calls `copy_process()` and starts the new task
- `copy_process()`
 - `dup_task_struct()`, which duplicates kernel stack, `task_struct`, and `thread_info`
 - Checks that we do not overflow the processes number limit
 - Various members of the `task_struct` are cleared

Forking (cont'd)

- `copy_process()`
 - Calls `sched_fork()` to set the child `state` set to `TASK_NEW`
 - Copies parent information such as files, signal handlers, etc.
 - Gets a new PID using `alloc_pid()`
 - Returns a pointer to the new child `task_struct`
- Finally, `_do_fork()` calls `wake_up_new_task()`
 - The new child task becomes `TASK_RUNNING`

Thread

- Threads are concurrent flows of execution belonging to the same program *sharing the same address space*



Thread

- There is no concept of a thread in Linux kernel
 - No scheduling for threads
- Linux implements all threads as standard processes
 - A thread is just another process sharing some information with other processes so each thread has its own `task_struct`
 - Created through `clone()` system call with specific flags indicating sharing
 - `clone(CLONE_VM | CLONE_FS | CLONE_FILES | CLONE_SIGHAND, 0);`

Kernel thread

- Used to perform background operations in the kernel
- Very similar to user space threads
 - They are schedulable entities (like regular processes)
- However they do not have their own address space
 - `mm` in `task_struct` is `NULL`
- Kernel threads are all forked from the `kthreadd` kernel thread (PID 2)
- Use cases (`ps --ppid 2`)
 - Work queue (`kworker`)
 - Load balancing among CPUs (`migration`)

Kernel thread

- To create a kernel thread, use `kthread_create()`
- When created through `kthread_create()`, the thread is not in a runnable state
- Need to call `wake_up_process()` or use `kthread_run()`
- Other threads can ask a kernel thread to stop using `kthread_stop()`
 - A kernel thread should check `kthread_should_stop()` to decide to continue or stop

Kernel thread

```
/**  
 * kthread_create - create a kthread on the current node  
 * @threadfn: the function to run in the thread  
 * @data: data pointer for @threadfn()  
 * @namefmt: printf-style format string for the thread name  
 * @...: arguments for @namefmt.  
 *  
 * This macro will create a kthread on the current node, leaving it in  
 * the stopped state.  
 */  
#define kthread_create(threadfn, data, namefmt, arg...) ...  
  
/**  
 * wake_up_process - Wake up a specific process  
 * @p: The process to be woken up.  
 *  
 * Attempt to wake up the nominated process and move it to the set of runnable  
 * processes.  
 *  
 * Return: 1 if the process was woken up, 0 if it was already running.  
 */  
int wake_up_process(struct task_struct *p);
```

Kernel thread

```
/**  
 * kthread_run - create and wake a thread.  
 * @threadfn: the function to run until signal_pending(current).  
 * @data: data ptr for @threadfn.  
 * @namefmt: printf-style name for the thread.  
 *  
 * Description: Convenient wrapper for kthread_create() followed by  
 * wake_up_process(). Returns the kthread or ERR_PTR(-ENOMEM).  
 */  
#define kthread_run(threadfn, data, namefmt, ...) ...  
  
/**  
 * kthread_stop - stop a thread created by kthread_create().  
 * @k: thread created by kthread_create().  
 *  
 * Sets kthread_should_stop() for @k to return true, wakes it, and  
 * waits for it to exit. If threadfn() may call do_exit() itself,  
 * the caller must ensure task_struct can't go away.  
 */  
int kthread_stop(struct task_struct *k);
```

Kernel thread example

- Ext4 file system uses a kernel thread to finish file system initialization in the background

```
/* linux/fs/ext4/super.c */
static int ext4_run_lazyinit_thread(void)
{
    ext4_lazyinit_task = kthread_run(ext4_lazyinit_thread,
                                    ext4_li_info, "ext4lazyinit");
    /* ... */
}

static int ext4_lazyinit_thread(void *arg)
{
    while (true) {
        if (kthread_should_stop()) {
            goto exit_thread;
        }
        /* ... */
    }
}
```

Kernel thread example

```
static void ext4_destroy_lazyinit_thread(void)
{
    /* ... */
    kthread_stop(ext4_lazyinit_task);
}

static void __exit ext4_exit_fs(void)
{
    ext4_destroy_lazyinit_thread();
    /* ... */
}

module_exit(ext4_exit_fs)
```

Process termination

- Termination on invoking the `exit()` system call
 - Can be implicitly inserted by the compiler on `return` from `main()`
 - `sys_exit()` calls `do_exit()`
- `do_exit()` (linux/kernel/exit.c)
 - Calls `exit_signals()` which set the `PF_EXITING` flag in the `task_struct`
 - Set the exit code in the `exit_code` field of the `task_struct`, which will be retrieved by the parent

Process termination (cont'd)

- `do_exit()` (`linux/kernel/exit.c`)
 - Calls `exit_mm()` to release the `mm_struct` of the task
 - Calls `exit_sem()`. If the process is queued waiting for a semaphore, it is dequeued here.
 - Calls `exit_files()` and `exit_fs()` to decrement the reference counter of file descriptors and filesystem data, respectively. If a reference counter becomes zero, that object is no longer in use by any process, and it is destroyed.

Process termination (cont'd)

- Calls `exit_notify()`
 - Sends signals to parent
 - Reparents any of its children to another thread in the thread group or the init process
 - Set `exit_state` in `task_struct` to `EXIT_ZOMBIE`
- Calls `do_task_dead()`
 - Set the `state` to `TASK_DEAD`
 - Calls `schedule()` to switch to a new process. Because the process is now not scalable, `do_exit()` never returns.

Process termination (cont'd)

- At that point, what is left is `task_struct`, `thread_info` and kernel stack
- This is required to provide information to the parent
 - `pid_t wait(int *wstatus)`
- After the parent retrieves the information, the remaining memory held by the process is freed
- Clean up implemented in `release_task()` called from the `wait()` implementation
 - Remove the task from the task list and release remaining resources

Zombie (or parentless) process

- **Q: What happens if a parent task exits before its child?**
- A child task must be *reparented*
- `exit_notify()` calls `forget_original_parent()`, that calls
`find_new_reaper()`
 - Returns the `task_struct` of another task in the thread group if it exists, otherwise `init`
 - Then, all the children of the currently dying task are reparented to the reaper

Further readings

- [Kernel Korner - Sleeping in the Kernel](#)
- [Exploiting Stack Overflows in the Linux Kernel](#)
- [security things in Linux v4.9](#)

Next lecture

- Process scheduling

Process Scheduling

Dongyoon Lee

Summary of last lectures

- Tools: building, exploring, and debugging Linux kernel
- Core kernel infrastructure
 - syscall, module, kernel data structures
- Process management

Today's agenda

- What is processing scheduling?
- History of Linux CPU scheduler
- Scheduling policy
- Scheduler class in Linux

Processor scheduler

- Decides which process runs next, when, and for how long
- Responsible for making the best use of processor (CPU)
 - *E.g., Do not waste CPU cycles for waiting process*
 - *E.g., Give higher priority to higher-priority processes*
 - *E.g., Do not starve low-priority processes*

Multitasking

- Simultaneously interleave execution of more than one process
- Single core
 - The processor scheduler gives illusion of multiple processes running concurrently
- Multi-core
 - The processor scheduler enables true parallelism

Types of multitasking OS

- **Cooperative multitasking:** old OSes (e.g., Windows 3.1) and few language runtimes (e.g., Go runtime)
 - A process does not stop running until it decides to *yield CPU*
 - The operating system cannot enforce fair scheduling
- **Preemptive multitasking:** almost all modern OSes
 - The OS can interrupt the execution of a process (i.e., *preemption*)
 - after the process expires its *timeslice*,
 - which is decided by *process priority*

Cooperative multitasking vs. Preemptive multitasking

Process #100

```
long count = 0;  
void foo(void) {  
    while(1) {  
        count++;  
    }  
}
```

Process #200

```
long val = 2;  
void bar(void) {  
    while(1) {  
        val *= 3;  
    }  
}
```

Process #300

```
void baz(void) {  
    while(1) {  
        printf("hi");  
    }  
}
```

Operating system: scheduler

CPU0

- Q: how can the preemptive scheduler take the control of infinite loop?

Scheduling policy: I/O vs. CPU-bound tasks

- A set of rules determining *what runs when*
- **I/O-bound processes**
 - Spend most of their time waiting for I/O: disk, network, keyboard, mouse, etc.
 - Runs for only short duration
 - Response time is important
- **CPU-bound processes**
 - Heavy use of the CPU: MATLAB, scientific computations, etc.
 - Caches stay hot when they run for a long time

Scheduling policy: process priority

- **Priority-based scheduling**
 - Rank processes based on their worth and need for processor time
 - Processes with a higher priority run before those with a lower priority

Scheduling policy: Linux process priority

- **Linux has two priority ranges**
 - Nice value: ranges from -20 to +19 (default is 0)
 - High values of nice means lower priority
 - Real-time priority: ranges from 0 to 99
 - Higher values mean higher priority
 - Real-time processes always executes before standard (nice) processes
- `ps ax -eo pid,ni,rtprio,cmd`

Scheduling policy: timeslice

- How much time a process should execute before being preempted
- Defining the default timeslice in an absolute way is tricky:
 - Too long → bad interactive performance
 - Too short → high context switching overhead

Scheduling policy: timeslice in Linux CFS

- **Linux CFS does not use an absolute timeslice**
 - The timeslice a process receives is function of the load of the system
 - In addition, that timeslice is weighted by the process priority
- When a process **P** becomes runnable:
 - **P** will preempt the currently running process **C** if **P** consumed a smaller proportion of the CPU than **C**

Scheduling policy: example

- Two tasks in the system:
 - Text editor: I/O-bound, latency sensitive (interactive)
 - Video encoder: CPU-bound, background job
- Scheduling goal
 - Text editor: when ready to run, need to preempt the video encoder for good *interactive performance*
 - Video encoder: run as long as possible for better CPU cache usage

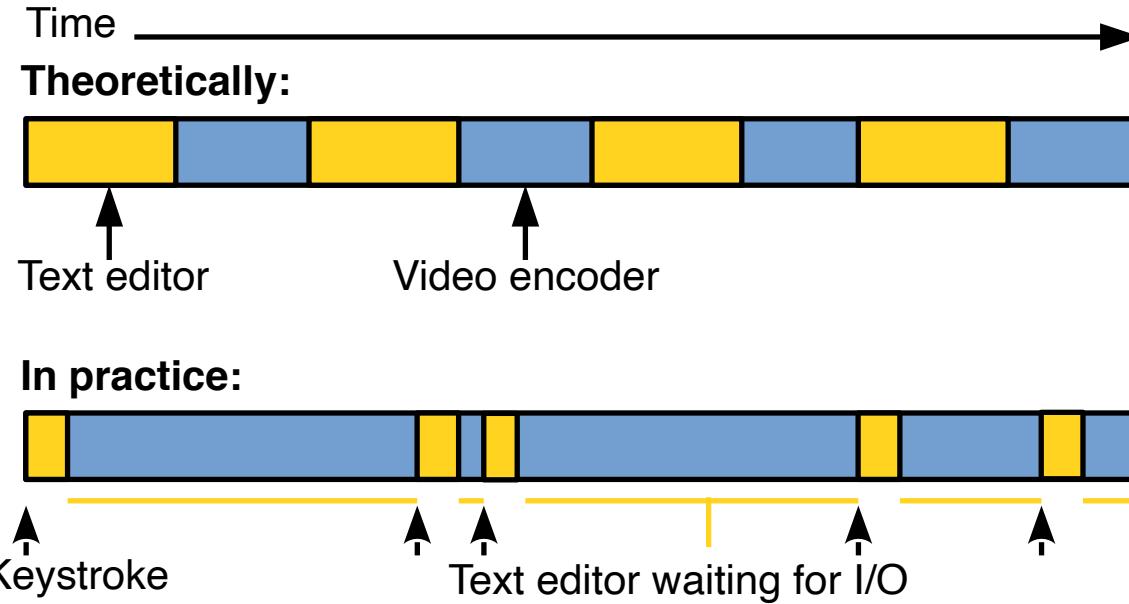
Scheduling policy: example in UNIX systems

- Gives higher priority to the text editor
- Not because it needs a lot of processor but because we want it to always have processor time available when it needs

Scheduling policy: example in Linux CFS

- CFS attempts to offer a fair proportion of CPU time
- CFS keeps track of the actual CPU time used by each program
- E.g., text editor : video encoder = 50% : 50%
 - The text editor mostly sleeps for waiting for user's input and the video encoder keeps running until preempted
 - When the text editor wakes up
 - CFS sees that text editor actually used less CPU time than the video encoder
 - The text editor preempts the video encoder

Scheduling policy: example in Linux CFS



- Good interactive performance
- Good background, CPU-bound performance

Linux CFS design

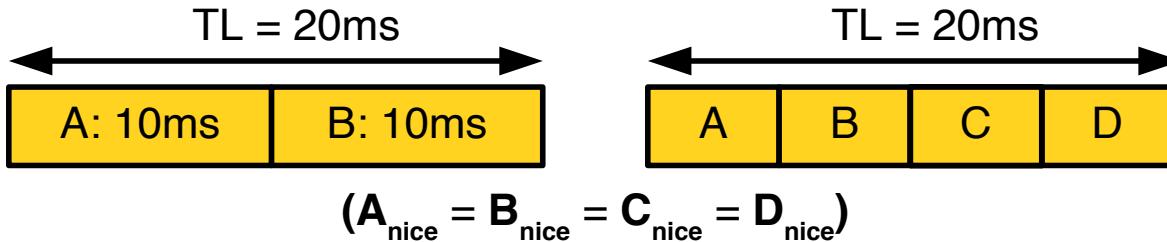
- Completely Fair Scheduler (CFS)
- Evolution of rotating staircase deadline scheduler (RSDL)
- At each moment, each process of the same priority has received an exact same amount of the CPU time
- If we could run n tasks in parallel on the CPU, give each $1/n$ of the CPU processing power
- CFS runs a process for some times, then swaps it for the runnable process that has run the least

Linux CFS design

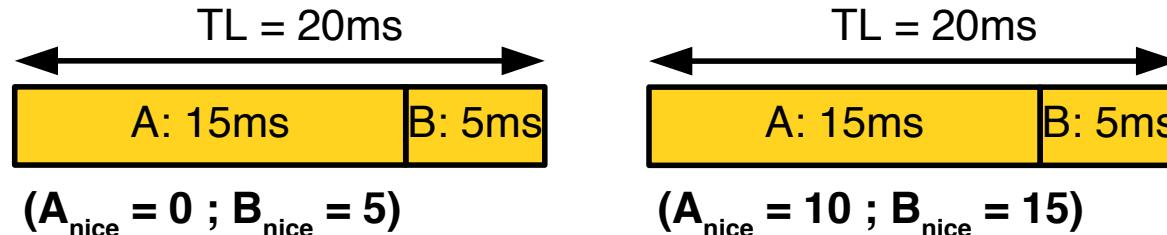
- No default timeslice, CFS calculates how long a process should run according to the number of runnable processes
 - That dynamic timeslice is weighted by the process priority (nice)
 - $\text{timeslice} = \text{weight of a task} / \text{total weight of runnable tasks}$
- To calculate the actual timeslice, CFS sets a **targeted latency**
 - Targeted latency: period during which all runnable processes should be scheduled at least once
 - Minimum granularity: floor at 1 ms (default)

Linux CFS design

- Example: processes with the same priority



- Example: processes with different priorities



Scheduler class design

- The Linux scheduler is *modular* and provides a *pluggable interface* for scheduling algorithms
 - Enables different scheduling algorithms co-exist, scheduling their own types of processes
- **Scheduler class** is a scheduling algorithm
 - Each scheduler class has a priority.
 - E.g., `SCHED_FIFO`, `SCHED_RR`, `SCHED_OTHER`
- The base scheduler code iterates over each scheduler in order of priority
 - `linux/kernel/sched/core.c`: `scheduler_tick()`, `schedule()`

Scheduler class design

- Time-sharing scheduling: `SCHED_OTHER`
 - `SCHED_NORMAL` in kernel code
 - Completely Fair Scheduler (CFS)
 - `linux/kernel/sched/fair.c`
- Real-time scheduling
 - `SCHED_FIFO` : First in-first out scheduling
 - `SCHED_RR` : Round-robin scheduling
 - `SCHED_DEADLINE` : Sporadic task model deadline scheduling

Scheduler class implementation

- `sched_class` : an abstract base class for all scheduler classes

```
/* linux/kernel/sched/sched.h */
struct sched_class {
    /* Called when a task enters a runnable state */
    void (*enqueue_task) (struct rq *rq, struct task_struct *p, int flags);
    /* Called when a task becomes unrunnable */
    void (*dequeue_task) (struct rq *rq, struct task_struct *p, int flags);
    /* Yield the processor (dequeue then enqueue back immediately) */
    void (*yield_task) (struct rq *rq);
    /* Preempt the current task with a newly woken task if needed */
    void (*check_preempt_curr) (struct rq *rq, struct task_struct *p, int flags);
    /* Choose a next task to run */
    struct task_struct * (*pick_next_task) (struct rq *rq,
                                           struct task_struct *prev,
                                           struct rq_flags *rf);
    /* Called periodically (e.g., 10 msec) by a system timer tick handler */
    void (*task_tick) (struct rq *rq, struct task_struct *p, int queued);
    /* Update the current task's runtime statistics */
    void (*update_curr) (struct rq *rq);
};
```

Scheduler class implementation

- Each scheduler class implements its own functions

```
/* linux/kernel/sched/fair.c */
const struct sched_class fair_sched_class = {
    .enqueue_task      = enqueue_task_fair,
    .dequeue_task      = dequeue_task_fair,
    .yield_task        = yield_task_fair,
    .check_preempt_curr = check_preempt_wakeup,
    .pick_next_task    = pick_next_task_fair,
    .task_tick          = task_tick_fair,
    .update_curr        = update_curr_fair, /* ... */
};

/* scheduler tick hitting a task of our scheduling class: */
static void task_tick_fair(struct rq *rq, struct task_struct *curr, int queued)
{
    struct cfs_rq *cfs_rq;
    struct sched_entity *se = &curr->se;
    for_each_sched_entity(se) {
        cfs_rq = cfs_rq_of(se);
        entity_tick(cfs_rq, se, queued);
    } /* ... */
}
```

Scheduler class implementation

- `task_struct` has scheduler-related fields.

```
/* linux/include/linux/sched.h */

struct task_struct {
    struct thread_info      thread_info;
    /* ... */
    const struct sched_class *sched_class; /* sched_class of this task */
    struct sched_entity       se; /* for time-sharing scheduling */
    struct sched_rt_entity    rt; /* for real-time scheduling */
    /* ... */
};
```

Scheduler class implementation

```
/* linux/include/linux/sched.h */

struct sched_entity {
    /* ... */
    struct rb_node      run_node;

    u64                exec_start;
    u64                sum_exec_runtime;
    u64                vruntime; /* how much time a process
                                * has been executed (ns) */
    struct cfs_rq      *cfs_rq; /* CFS run queue */
    /* ... */
};

struct cfs_rq {
    /* ... */
    struct rb_root_cached tasks_timeline; /* rb tree */
    /* ... */
};
```

Two scheduler entry points

- The base scheduler code triggers scheduling operations in two cases
 - when processing a timer interrupt (`scheduler_tick()`)
 - when the kernel calls `schedule()`

A timer interrupt calls *scheduler_tick()*

```
/* linux/kernel/sched/core.c */
/* This function gets called by the timer code, with HZ frequency. */
void scheduler_tick(void)
{
    int cpu = smp_processor_id();
    struct rq *rq = cpu_rq(cpu);
    struct task_struct *curr = rq->curr;
    struct rq_flags rf;

    /* call task_tick handler for the current process */
    sched_clock_tick();
    rq_lock(rq, &rf);
    update_rq_clock(rq);
    curr->sched_class->task_tick(rq, curr, 0); /* e.g., task_tick_fair in CFS */
    cpu_load_update_active(rq);
    calc_global_load_tick(rq);
    rq_unlock(rq, &rf);

    /* load balancing among CPUs */
    rq->idle_balance = idle_cpu(cpu);
    trigger_load_balance(rq);
    rq_last_tick_reset(rq);
}
```

The kernel calls *schedule()*

```
/* linux/kernel/sched/core.c */
/* __schedule() is the main scheduler function. */
static void __sched notrace __schedule(bool preempt)
{
    struct task_struct *prev, *next;
    struct rq_flags rf;
    struct rq *rq;
    int cpu;

    cpu = smp_processor_id();
    rq = cpu_rq(cpu);
    prev = rq->curr;

    /* pick up the highest-prio task */
    next = pick_next_task(rq, prev, &rf);

    if (likely(prev != next)) {
        /* switch to the new MM and the new thread's register state */
        rq->curr = next;
        rq = context_switch(rq, prev, next, &rf);
    }
    /* ... */
}
```

Scheduler class implementation

```
/* linux/kernel/sched/core.c */
/* Pick up the highest-prio task: */
static inline struct task_struct *
pick_next_task(struct rq *rq, struct task_struct *prev, struct rq_flags *rf)
{
    const struct sched_class *class;
    struct task_struct *p;

    /* ... */
again:
    for_each_class(class) {
        /* In CFS, pick_next_task_fair() will be called */
        p = class->pick_next_task(rq, prev, rf);
        if (p) {
            if (unlikely(p == RETRY_TASK))
                goto again;
            return p;
        }
    }

    /* The idle class should always have a runnable task: */
    BUG();
}
```

Next lecture

- Processing Scheduling II

Process Scheduling II

Dongyoon Lee

Summary of last lectures

- Tools: building, exploring, and debugging Linux kernel
- Core kernel infrastructure
 - syscall, module, kernel data structures
- Process management
- Process scheduling I

Today's agenda

- Linux Completely Fair Scheduler (CFS)
- Preemption and context switching
- Real-time scheduling policies
- Scheduling related system calls

Linux CFS design

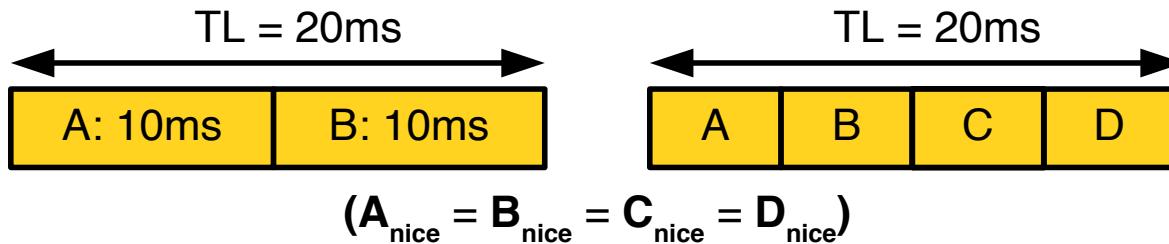
- Completely Fair Scheduler (CFS)
- Evolution of rotating staircase deadline scheduler (RSDL)
- At each moment, each process of the same priority has received an exact same amount of the CPU time
- If we could run n tasks in parallel on the CPU, give each $1/n$ of the CPU processing power
- CFS runs a process for some times, then swaps it for the runnable process that has run the least

Linux CFS design

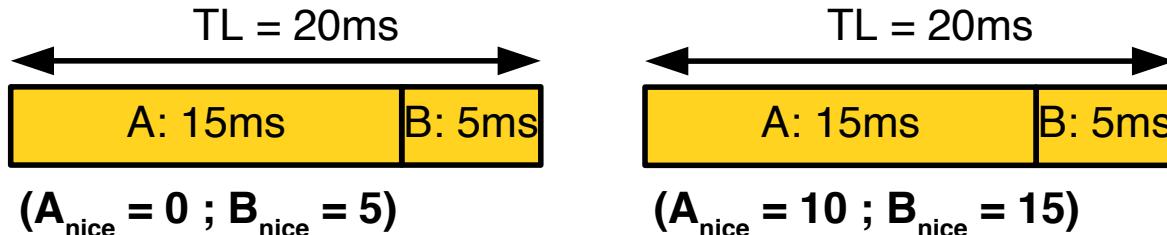
- No default timeslice , CFS calculates how long a process should run according to the number of runnable processes
 - That dynamic timeslice is weighted by the process priority (nice)
 - $\text{timeslice} = \text{weight of a task} / \text{total weight of runnable tasks}$
- To calculate the actual timeslice, CFS sets a **targeted latency**
 - Targeted latency: period during which all runnable processes should be scheduled at least once
 - Minimum granularity: floor at 1 ms (default)

Linux CFS design

- Example: processes with the same priority



- Example: processes with different priorities



CFS implementation

- Four main components of CFS
 - Time accounting
 - Process selection
 - Scheduler entry point: `schedule()`, `scheduler_tick()`
 - Sleeping and waking up

Time accounting in CFS

- **Virtual runtime:** how much time a process has been executed

```
/* linux/include/linux/sched.h */
struct task_struct {
    /* ... */
    const struct sched_class *sched_class; /* sched_class of this task */
    struct sched_entity      se; /* for time-sharing scheduling */
    struct sched_rt_entity   rt; /* for real-time scheduling */
    /* ... */
};

struct sched_entity {
    /* ... */
    struct rb_node          run_node;

    u64                      exec_start;
    u64                      sum_exec_runtime;
    u64                      vruntime; /* how much time a process
                                         * has been executed (ns) */
    struct cfs_rq            *cfs_rq; /* CFS run queue */
    /* ... */
};
```

Time accounting in CFS

- Upon every timer interrupt, CFS accounts the task's execution time
- `scheduler_tick()` → `task_tick_fair()` → `update_curr()`

```
/* linux/kernel/sched/fair.c */
static void update_curr(struct cfs_rq *cfs_rq)
{
    struct sched_entity *curr = cfs_rq->curr;
    u64 now = rq_clock_task(rq_of(cfs_rq));
    u64 delta_exec;

    if (unlikely(!curr))
        return;

    delta_exec = now - curr->exec_start; /* Step 1. calc exec duration */
    if (unlikely((s64)delta_exec < 0))
        return;

    curr->exec_start = now;
    /* continue in a next slide ... */
}
```

Time accounting in CFS

```
static void update_curr(struct cfs_rq *cfs_rq)
{
    /* continue from the previous slide ... */

    schedstat_set(curr->statistics.exec_max,
                  max(delta_exec, curr->statistics.exec_max));

    curr->sum_exec_runtime += delta_exec;
    schedstat_add(cfs_rq->exec_clock, delta_exec);

    /* update vruntime with delta_exec and nice value */
    curr->vruntime += calc_delta_fair(delta_exec, curr); /* CODE */
    update_min_vruntime(cfs_rq);

    if (entity_is_task(curr)) {
        struct task_struct *curtask = task_of(curr);

        trace_sched_stat_runtime(curtask, delta_exec, curr->vruntime);
        cpuacct_charge(curtask, delta_exec);
        account_group_exec_runtime(curtask, delta_exec);
    }

    account_cfs_rq_runtime(cfs_rq, delta_exec);
}
```

Time accounting in CFS

- CFS checks if the currently running task needs to be preempted (i.e., the tasks uses up the timeslice). If so, set the `TIF_NEED_RESCHED` flag
- `scheduler_tick()` → `check_preempt_tick()`

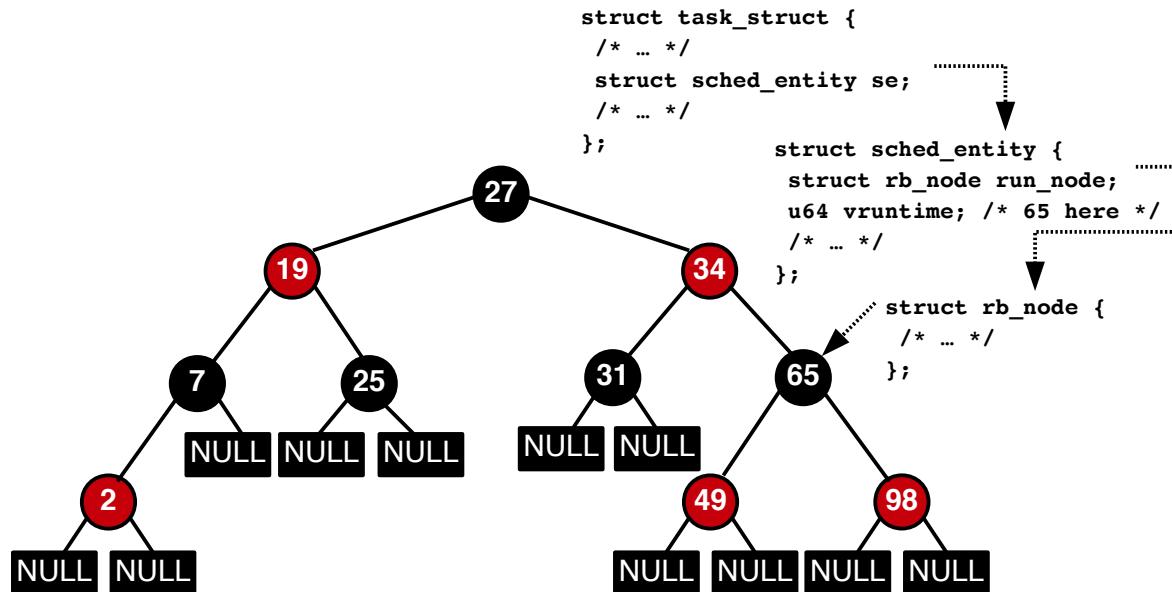
```
/* linux/kernel/sched/fair.c */
static void
check_preempt_tick(struct cfs_rq *cfs_rq, struct sched_entity *curr)
{
    /* ... */
    ideal_runtime = sched_slice(cfs_rq, curr);
    delta_exec = curr->sum_exec_runtime - curr->prev_sum_exec_runtime;
    if (delta_exec > ideal_runtime) {
        resched_curr(rq_of(cfs_rq));
        clear_buddies(cfs_rq, curr);
        return;
    }
    /* ... */
}
```

Time accounting in CFS: QEMU-gdb (Lec 6)

```
B+> 3092     void scheduler_tick(void)
3093     {
3094         int cpu = smp_processor_id();
3095         struct rq *rq = cpu_rq(cpu);
3096         struct task_struct *curr = rq->curr;
3097         struct rq_flags rf;
3098
3099         sched_clock_tick();
3100
3101         rq_lock(rq, &rf);
3102
3103         update_rq_clock(rq);
3104         curr->sched_class->task_tick(rq, curr, 0);
3105         cpu_load_update_active(rq);
3106         calc_global_load_tick(rq);
3107
3108         rq_unlock(rq, &rf);
3109
3110         perf_event_task_tick();
3111
remote Thread 2 In: scheduler_tick
Thread 2 hit Breakpoint 1, scheduler_tick () at kernel/sched/core.c:3093
(gdb) bt
#0  scheduler_tick () at kernel/sched/core.c:3093
#1  0xffffffff810bc302 in update_process_times (user_tick=0) at kernel/time/timer.c:1576
#2  0xffffffff810caacd in tick_sched_handle (regs=<optimized out>, ts=<optimized out>,
    ts=<optimized out>) at kernel/time/tick-sched.c:155
#3  0xffffffff810cafdb in tick_sched_timer (timer=0xfffff88007fd135c0) at kernel/time/tick-sched.c:1174
#4  0xffffffff810bcbe2 in __run_hrtimer (now=<optimized out>, timer=<optimized out>,
    base=<optimized out>, cpu_base=<optimized out>) at kernel/time/hrtimer.c:1212
#5  __hrtimer_run_queues (cpu_base=0xfffff88007fd13180, now=<optimized out>) at kernel/time/hrtimer.c:1276
#6  0xffffffff810bd23b in hrtimer_interrupt (dev=<optimized out>) at kernel/time/hrtimer.c:1310
#7  0xffffffff8103d9a3 in local_apic_timer_interrupt () at arch/x86/kernel/apic/apic.c:941
#8  0xffffffff8103e383 in smp_apic_timer_interrupt (regs=<optimized out>)
    at arch/x86/kernel/apic/apic.c:965
#9  0xffffffff8194f9c6 in apic_timer_interrupt () at arch/x86/entry/entry_64.S:701
#10 0xfffffc9000036fdf8 in ?? ()
#11 0x0000000000000000 in ?? ()
(gdb) ■
```

CFS maintains `cfs_rq` (runqueue) as a rbtree

- CFS maintains a rbtree of tasks indexed by `vruntime` (i.e., runqueue)
- Always pick a task with the smallest `vruntime`, the left-most node



Adding a task to a runqueue

- When a task is woken up or migrated, it is added to a runqueue

```
/* linux/kernel/sched/fair.c */
void enqueue_entity(struct cfs_rq *cfs_rq, struct sched_entity *se, int flags)
{
    bool renorm = !(flags & ENQUEUE_WAKEUP) || (flags & ENQUEUE_MIGRATED);
    bool curr = cfs_rq->curr == se;

    /* Update run-time statistics */
    update_curr(cfs_rq);

    update_load_avg(se, UPDATE_TG);
    enqueue_entity_load_avg(cfs_rq, se);
    update_cfs_shares(se);
    account_entity_enqueue(cfs_rq, se);
    /* ... */

    /* Add this to the rbtree */
    if (!curr)
        __enqueue_entity(cfs_rq, se);
    /* ... */
}
```

Adding a task to a runqueue

```
/* linux/kernel/sched/fair.c */
static void __enqueue_entity(struct cfs_rq *cfs_rq, struct sched_entity *se)
{
    struct rb_node **link = &cfs_rq->tasks_timeline.rb_node;
    struct rb_node *parent = NULL;
    struct sched_entity *entry;
    int leftmost = 1;
    /* Find the right place in the rbtree: */
    while (*link) {
        parent = *link;
        entry = rb_entry(parent, struct sched_entity, run_node);
        if (entity_before(se, entry)) {
            link = &parent->rb_left;
        } else {
            link = &parent->rb_right;
            leftmost = 0;
        }
    }
    /* Maintain a cache of leftmost tree entries (it is frequently used): */
    if (leftmost)
        cfs_rq->rb_leftmost = &se->run_node;
    rb_link_node(&se->run_node, parent, link);
    rb_insert_color(&se->run_node, &cfs_rq->tasks_timeline);
}
```

Removing a task from a runqueue

- When a task goes to sleep or is migrated, it is removed from a runqueue

```
/* linux/kernel/sched/fair.c */
void dequeue_entity(struct cfs_rq *cfs_rq, struct sched_entity *se, int flags)
{
    /* Update run-time statistics of the 'current'. */
    update_curr(cfs_rq);
    update_load_avg(se, UPDATE_TG);
    dequeue_entity_load_avg(cfs_rq, se);
    update_stats_dequeue(cfs_rq, se, flags);
    clear_buddies(cfs_rq, se);

    /* Remove this to the rbtree */
    if (se != cfs_rq->curr)
        __dequeue_entity(cfs_rq, se);
    se->on_rq = 0;
    account_entity_dequeue(cfs_rq, se);
    /* ... */
}
```

Removing a task from a runqueue

```
static void __dequeue_entity(struct cfs_rq *cfs_rq, struct sched_entity *se)
{
    if (cfs_rq->rb_leftmost == &se->run_node) {
        struct rb_node *next_node;

        next_node = rb_next(&se->run_node);
        cfs_rq->rb_leftmost = next_node;
    }

    rb_erase(&se->run_node, &cfs_rq->tasks_timeline);
}
```

Scheduler entry point: `schedule()`

```
/* linux/kernel/sched/core.c */
/* __schedule() is the main scheduler function. */
static void __sched notrace __schedule(bool preempt)
{
    struct task_struct *prev, *next;
    struct rq_flags rf;
    struct rq *rq;
    int cpu;

    cpu = smp_processor_id();
    rq = cpu_rq(cpu);
    prev = rq->curr;

    /* pick up the highest-prio task */
    next = pick_next_task(rq, prev, &rf);

    if (likely(prev != next)) {
        /* switch to the new MM and the new thread's register state */
        rq->curr = next;
        rq = context_switch(rq, prev, next, &rf);
    }
    /* ... */
}
```

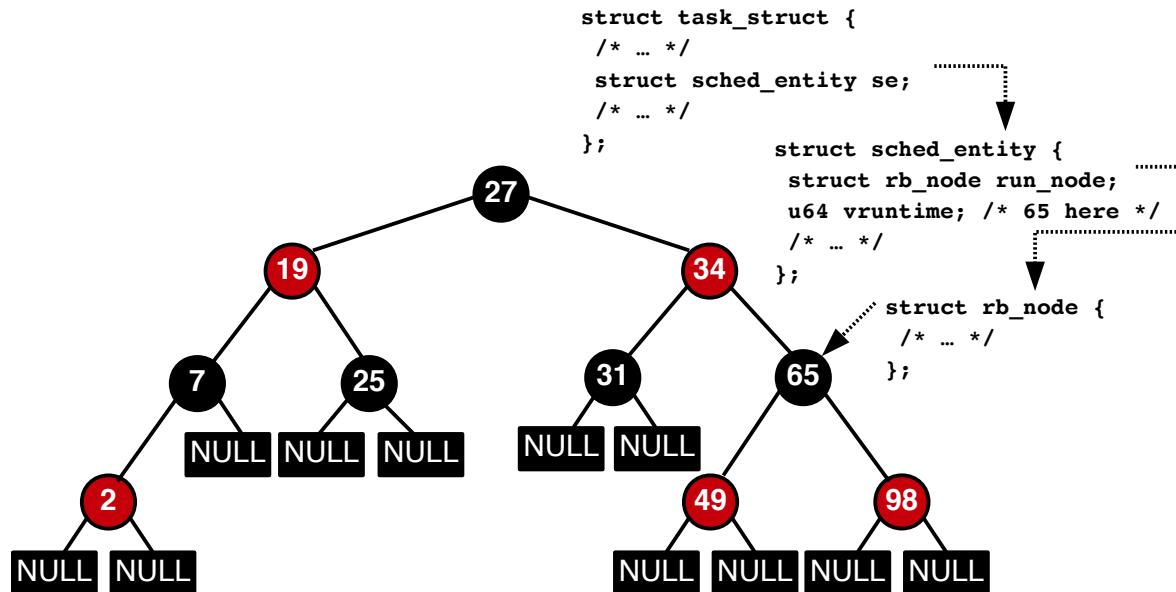
Scheduler entry point: `schedule()`

```
/* linux/kernel/sched/core.c */
/* Pick up the highest-prio task: */
static inline struct task_struct *
pick_next_task(struct rq *rq, struct task_struct *prev, struct rq_flags *rf)
{
    const struct sched_class *class;
    struct task_struct *p;

    /* ... */
again:
    for_each_class(class) {
        /* In CFS, pick_next_task_fair() will be called.
         * pick_next_task_fair() eventually calls __pick_first_entity() */
        p = class->pick_next_task(rq, prev, rf);
        if (p) {
            if (unlikely(p == RETRY_TASK))
                goto again;
            return p;
        }
    }
    /* The idle class should always have a runnable task: */
    BUG();
}
```

Process selection in CFS

- CFS maintains an rbtree of tasks indexed by `vruntime` (i.e., runqueue)
- Always pick a task with the smallest `vruntime`, the left-most node



Process selection in CFS

- `schedule()` calls `pick_next_task_fair()` in CFS
- `pick_next_task_fair()` calls `__pick_first_entity()` that returns the left-most node of the rbtree (runqueue)

```
/* linux/kernel/sched/fair.c */
struct sched_entity *__pick_first_entity(struct cfs_rq *cfs_rq) /* CODE */
{
    struct rb_node *left = cfs_rq->rb_leftmost;

    if (!left)
        return NULL;

    return rb_entry(left, struct sched_entity, run_node);
}
```

Sleeping and waking up

- Reasons for a task to sleep:
 - Specified amount of time, waiting for I/O, blocking on a mutex, etc.
- Steps to sleep
 - Mark a task sleeping
 - Put the task into a `waitqueue`
 - Dequeue the task from the rbtree of runnable tasks
 - The task calls `schedule()` to select a new task to run
- Waking up a task is the inverse of sleeping

Sleeping and waking up

- Two states associated with sleeping:
 - `TASK_INTERRUPTIBLE`
 - Wake up the sleeping task upon signal
 - `TASK_UNINTERRUPTIBLE`
 - Defer signal delivery until wake up

Wait queue: sleeping

- List of tasks waiting for an event to occur

```
/* linux/include/linux/wait.h */

struct wait_queue_entry {
    unsigned int          flags;
    void                 *private;
    wait_queue_func_t    func;
    struct list_head     entry;
};

struct wait_queue_head {
    spinlock_t            lock;
    struct list_head      head;
};

#define DEFINE_WAIT(name) ...

void add_wait_queue(struct wait_queue_head *wq_head, struct wait_queue_entry *wq_entry);
void prepare_to_wait(struct wait_queue_head *wq_head, struct wait_queue_entry *wq_entry, int s
void finish_wait(struct wait_queue_head *wq_head, struct wait_queue_entry *wq_entry);
```

Wait queue: sleeping

```
DEFINE_WAIT(wait); /* Initialize a wait queue entry */

/* 'q' is the wait queue that we wish to sleep on */
add_wait_queue(q, &wait); /* Add itself to a wait queue */
while (!condition) { /* event we are waiting for */
    /* Change process status to TASK_INTERRUPTIBLE */
    prepare_to_wait(&q, &wait, TASK_INTERRUPTIBLE); /* prevent the lost wake-up */
    /* Since the state is TASK_INTERRUPTIBLE, a signal can wake up the task.
     * If there is a pending signal, handle signals */
    if(signal_pending(current)) {
        /* This is a spurious wake up, not caused
         * by the occurrence of the waiting event */
        /* Handle signal */
    }
    /* Go to sleep */
    schedule();
    /* Now, the task is woken up.
     * Check condition if the event occurs */
}

/* Set the process status to TASK_RUNNING
 * and remove itself from the wait queue */
finish_wait(&q, &wait);
```

Wait queue: sleeping

- Or use one of `wait_event_*()` macros

```
/* linux/include/linux/wait.h */

/**
 * wait_event_interruptible - sleep until a condition gets true
 * @wq: the waitqueue to wait on
 * @condition: a C expression for the event to wait for
 *
 * The process is put to sleep (TASK_INTERRUPTIBLE) until the
 * @condition evaluates to true or a signal is received.
 * The @condition is checked each time the waitqueue @wq is woken up.
 */
#define wait_event_interruptible(wq, condition) \
({ \
    int __ret = 0; \
    might_sleep(); \
    if (!(condition)) \
        __ret = __wait_event_interruptible(wq, condition); \
    __ret; \
})
```

Wait queue: waking-up

- Waking up waiting tasks by `wake_up()`
 - By default, wake up *all* the tasks on a waitqueue
 - Exclusive tasks are added using `prepare_to_wait_exclusive()`

```
#define wake_up(x)           __wake_up(x, TASK_NORMAL, 1, NULL)

/* __wake_up() calls __wake_up_common() */
static void __wake_up_common(wait_queue_head_t *q, unsigned int mode,
                           int nr_exclusive, int wake_flags, void *key)
{
    wait_queue_t *curr, *next;
    list_for_each_entry_safe(curr, next, &q->task_list, task_list) {
        unsigned flags = curr->flags;
        if (curr->func(curr, mode, wake_flags, key) && /* wake-up function */
            (flags & WQ_FLAG_EXCLUSIVE) && !--nr_exclusive)
            break;
    }
}
```

Wait queue: waking-up

- A wait queue entry contains a pointer to a wake-up function

```
/* linux/include/linux/wait.h */

typedef struct wait_queue_entry wait_queue_entry_t;

typedef int (*wait_queue_func_t)(struct wait_queue_entry *wq_entry,
                                unsigned mode, int flags, void *key);
int default_wake_function(struct wait_queue_entry *wq_entry,
                           unsigned mode, int flags, void *key);

struct wait_queue_entry {
    unsigned int          flags;
    void                 *private;
    wait_queue_func_t    func;
    struct list_head     entry;
};

};
```

Wait queue: waking-up

- `default_wake_function()` calls `try_to_wake_up()`
 - calls `ttwu_queue()`
 - calls `ttwu_do_activate()`
 - puts the task back on runqueue
 - calls `ttwu_do_wakeup()`
 - calls `check_preempt_curr()`
 - sets the `TIF_NEED_RESCHED` flag (as needed)
 - sets the task state to `TASK_RUNNING`

CFS on multi-core machines

- Per-CPU runqueues (rbtrees)
 - To avoid costly accesses to shared data structures
- Runqueues must be kept balanced
 - E.g., dual-core with one long runqueue of high-priority processes, and a short one with low-priority processes
 - High-priority processes get less CPU time than low-priority ones
- A load balancer runs periodically based on priority and CPU usage

Preemption and context switching

- A **context switch** is the action of swapping the process currently running on the CPU to another one
- Performed by `context_switch()`, which is called by `schedule()`
 - Switch the address space through `switch_mm()`
 - Switch the CPU state (registers) through `switch_to()`

Preemption and context switching

- Then, when `schedule()` will be called?
 - A task can voluntarily relinquish the CPU by calling `schedule()`
 - A current task needs to be preempted if
 1. it runs long enough
 - by `scheduler_tick()`
 2. a task with a higher priority is woken up
 - by `try_to_wake_up()`

need_resched

- The `TIF_NEED_RESCHED` flag (in `thread_info`)
 - specifies whether a (preemptive) reschedule should be performed
 - `set_tsk_need_resched()`
 - `clear_tsk_need_resched()`
 - `need_resched()`
- `TIF_NEED_RESCHED` is set by
 - `scheduler_tick()` : the currently running task needs to be preempted
 - `try_to_wake_up()` : a process with higher priority wakes up

need_resched

- Then `TIF_NEED_RESCHED` flag is checked:
 - Upon returning to user space (from a syscall or an interrupt)
 - Upon returning from an interrupt
- If the flag is set, `schedule()` is called

need_resched

Process #100

```
long count = 0;  
void foo(void) {  
    while(1) {  
        count++;  
    }  
}
```

Process #200

```
long val = 2;  
void bar(void) {  
    while(1) {  
        val *= 3;  
    }  
}
```

Process #300

```
void baz(void) {  
    while(1) {  
        printf("hi");  
    }  
}
```

Operating system: scheduler

CPU0

- **Q: how can the preemptive scheduler take the control of infinite loop?**

Kernel preemption

- In most of UNIX-like operating systems, kernel code is non-preemptive
- In Linux, the kernel code is also preemptive
 - A task can be preempted in the kernel as long as execution is in a safe state without holding any lock
- `preempt_count` in the `thread_info` structure indicates the current lock depth
- If `need_resched && !preempt_count` then, it is safe to preempt
 - Checked when returning to the kernel from interrupt
 - Checked when releasing a lock

Kernel preemption

- Kernel preemption can occur:
 - On return from interrupt
 - When kernel code becomes preemptible again
 - If a task in the kernel blocks (e.g., mutex)

```
/* linux/include/linux/preempt.h */
#define preempt_disable() \
do { \
    preempt_count_inc(); \
    barrier(); \
} while (0)
#define preempt_enable() \
do { \
    barrier(); \
    if (unlikely(preempt_count_dec_and_test())) \
        __preempt_schedule(); \
} while (0)
```

Real-time scheduling policies

- Linux provides two *soft real-time* scheduling classes
 - SCHED_FIFO , SCHED_RR , SCHED_DEADLINE
 - Best effort, no guarantee
- Real-time task of any scheduling class will always run before non-realtime ones (CFS, SCHED_OTHER)
 - schedule() → pick_next_task() → for_each_class()

Real-time scheduling policies

- SCHED_FIFO
 - Tasks run until it blocks/yield
 - Only a higher priority RT task can preempt it
 - Round-robin for tasks of same priority
- SCHED_RR
 - Same as SCHED_FIFO, but with a fixed timeslice

Real-time scheduling policies

- **SCHED_DEADLINE**
 - Real-time policies mainlined in v3.14 enabling predictable RT scheduling
 - Early deadline first (EDF) scheduling based on a period of activation and a worst case execution time (WCET) for each task
 - Ref: [kernel Documentation](#)
- **SCHED_BATCH** : non-real-time, low priority background jobs
- **SCHED_IDLE** : non-real-time, very low priority background jobs

Scheduling related system calls

- `sched_getscheduler`, `sched_setscheduler`
- `nice`
- `sched_getparam`, `sched_setparam`
- `sched_get_priority_max`, `sched_get_priority_min`
- `sched_getaffinity`, `sched_setaffinity`
- `sched_yield`

Scheduling related system calls: example

```
#define _GNU_SOURCE

#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <unistd.h>
#include <sched.h>
#include <assert.h>

void handle_err(int ret, char *func)
{
    perror(func);
    exit(EXIT_FAILURE);
}

int main(void)
{
    pid_t pid = -1;
    int ret = -1;
    struct sched_param sp;
    int max_rr_prio, min_rr_prio = -42;
    size_t cpu_set_size = 0;
    cpu_set_t cs;
```

Scheduling related system calls: example

```
/* Get the PID of the calling process */
pid = getpid();
printf("My pid is: %d\n", pid);

/* Get the scheduling class */
ret = sched_getscheduler(pid);
if(ret == -1)
    handle_err(ret, "sched_getscheduler");
printf("sched_getscheduler returns: %d\n", ret);
assert(ret == SCHED_OTHER);

/* Get the priority (nice/RT) */
ret = sched_getparam(pid, &sp);
if(ret == -1)
    handle_err(ret, "sched_getparam");
printf("My priority is: %d\n", sp.sched_priority);

/* Set the priority (nice value) */
ret = nice(1);
if(ret == -1)
    handle_err(ret, "nice");
```

Scheduling related system calls: example

```
/* Get the priority again */
ret = sched_getparam(pid, &sp);
if(ret == -1)
    handle_err(ret, "sched_getparam");
printf("My priority is: %d\n", sp.sched_priority);

/* Switch scheduling class to FIFO and the priority to 99 */
sp.sched_priority = 99;
ret = sched_setscheduler(pid, SCHED_FIFO, &sp);
if(ret == -1)
    handle_err(ret, "sched_setscheduler");

/* Get the scheduling class */
ret = sched_getscheduler(pid);
if(ret == -1)
    handle_err(ret, "sched_getscheduler");
printf("sched_getscheduler returns: %d\n", ret);
assert(ret == SCHED_FIFO);
```

Scheduling related system calls: example

```
/* Get the priority again */
ret = sched_getparam(pid, &sp);
if(ret == -1)
    handle_err(ret, "sched_getparam");
printf("My priority is: %d\n", sp.sched_priority);

/* Set the RT priority */
sp.sched_priority = 42;
ret = sched_setparam(pid, &sp);
if(ret == -1)
    handle_err(ret, "sched_setparam");
printf("Priority changed to %d\n", sp.sched_priority);

/* Get the priority again */
ret = sched_getparam(pid, &sp);
if(ret == -1)
    handle_err(ret, "sched_getparam");
printf("My priority is: %d\n", sp.sched_priority);
```

Scheduling related system calls: example

```
/* Get the max priority value for SCHED_RR */
max_rr_prio = sched_get_priority_max(SCHED_RR);
if(max_rr_prio == -1)
    handle_err(max_rr_prio, "sched_get_priority_max");
printf("Max RR prio: %d\n", max_rr_prio);

/* Get the min priority value for SCHED_RR */
min_rr_prio = sched_get_priority_min(SCHED_RR);
if(min_rr_prio == -1)
    handle_err(min_rr_prio, "sched_get_priority_min");
printf("Min RR prio: %d\n", min_rr_prio);

cpu_set_size = sizeof(cpu_set_t);
CPU_ZERO(&cs); /* clear the mask */
CPU_SET(0, &cs);
CPU_SET(1, &cs);
/* Set the affinity to CPUs 0 and 1 only */
ret = sched_setaffinity(pid, cpu_set_size, &cs);
if(ret == -1)
    handle_err(ret, "sched_setaffinity");
```

Scheduling related system calls: example

```
/* Get the CPU affinity */
CPU_ZERO(&cs);
ret = sched_getaffinity(pid, cpu_set_size, &cs);
if(ret == -1)
    handle_err(ret, "sched_getaffinity");
assert(CPU_ISSET(0, &cs));
assert(CPU_ISSET(1, &cs));
printf("Affinity tests OK\n");

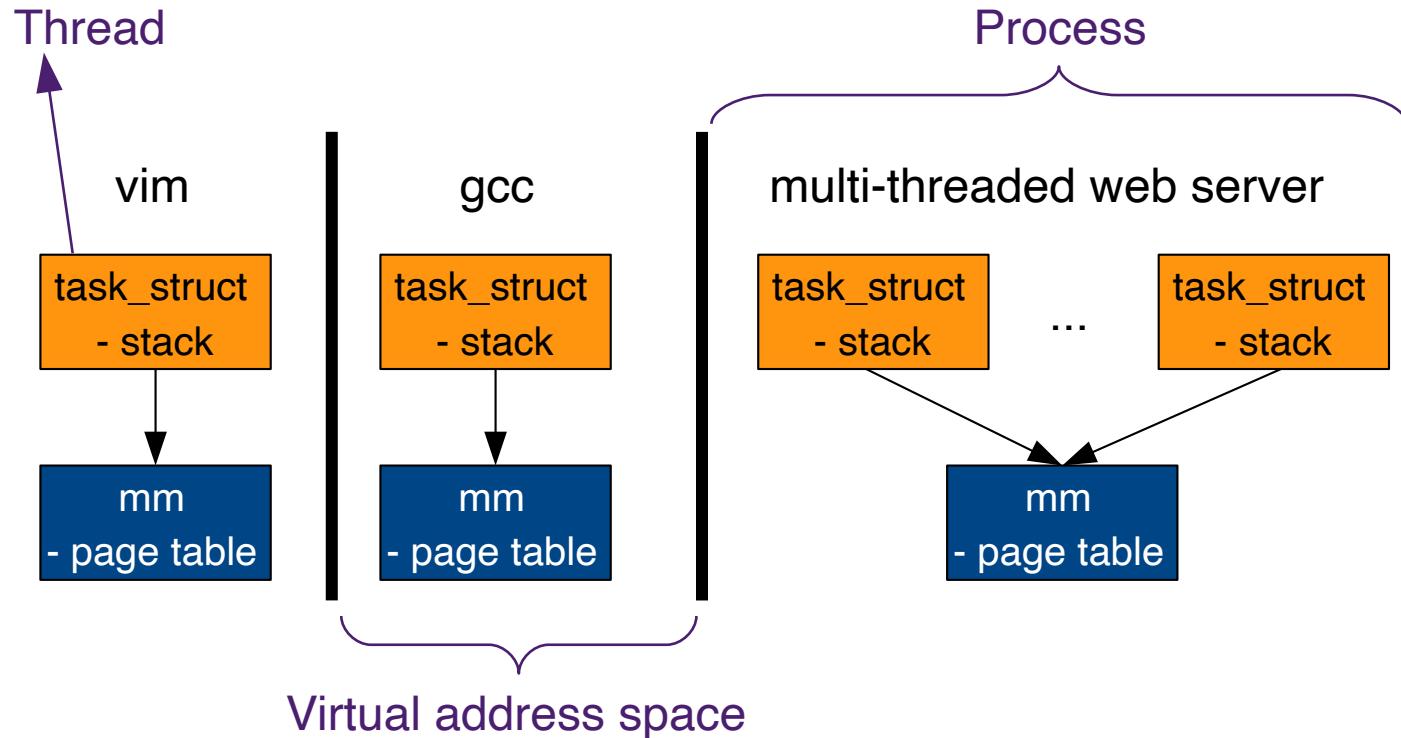
/* Yield the CPU */
ret = sched_yield();
if(ret == -1)
    handle_err(ret, "sched_yield");

return EXIT_SUCCESS;
}
```

Summary: **task = process | thread**

- **struct task_struct**
 - a process or a thread
- **struct mm**
 - a virtual address space

task = process | thread

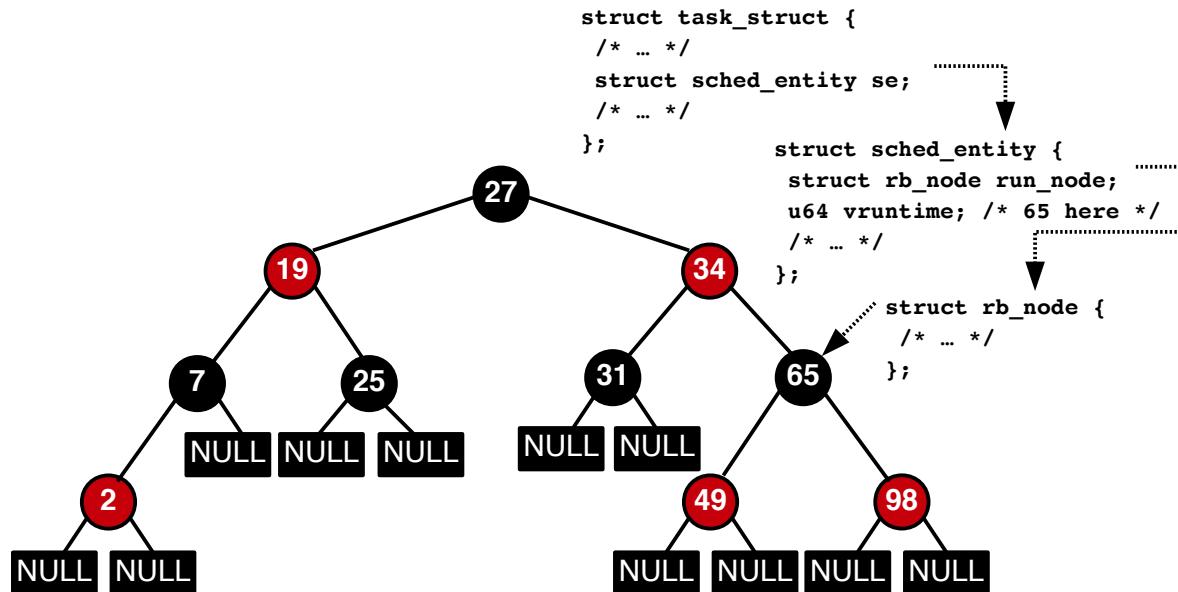


Summary: Completely Fair Scheduling (CFS)

- `struct sched_entity`
 - embedded to `task_struct`
 - has `vruntime` value
- `struct cfs_rq`
 - a queue of runnable tasks in a `TASK_RUNNING` status
 - has `struct rb_root tasks_timeline`

Summary: Completely Fair Scheduling (CFS)

- CFS maintains an rbtree of tasks indexed by `vruntime` (i.e., runqueue)
- Always pick a task with the smallest `vruntime`, the left-most node



Next lecture

- Interrupt Handler: Top Half

Further readings

- [The Battle of the Schedulers: FreeBSD ULE vs. Linux CFS, USENIX ATC18](#)
- [The Rotating Staircase Deadline Scheduler](#)

Interrupt Handler: Top Half

Dongyoon Lee

Summary of last lectures

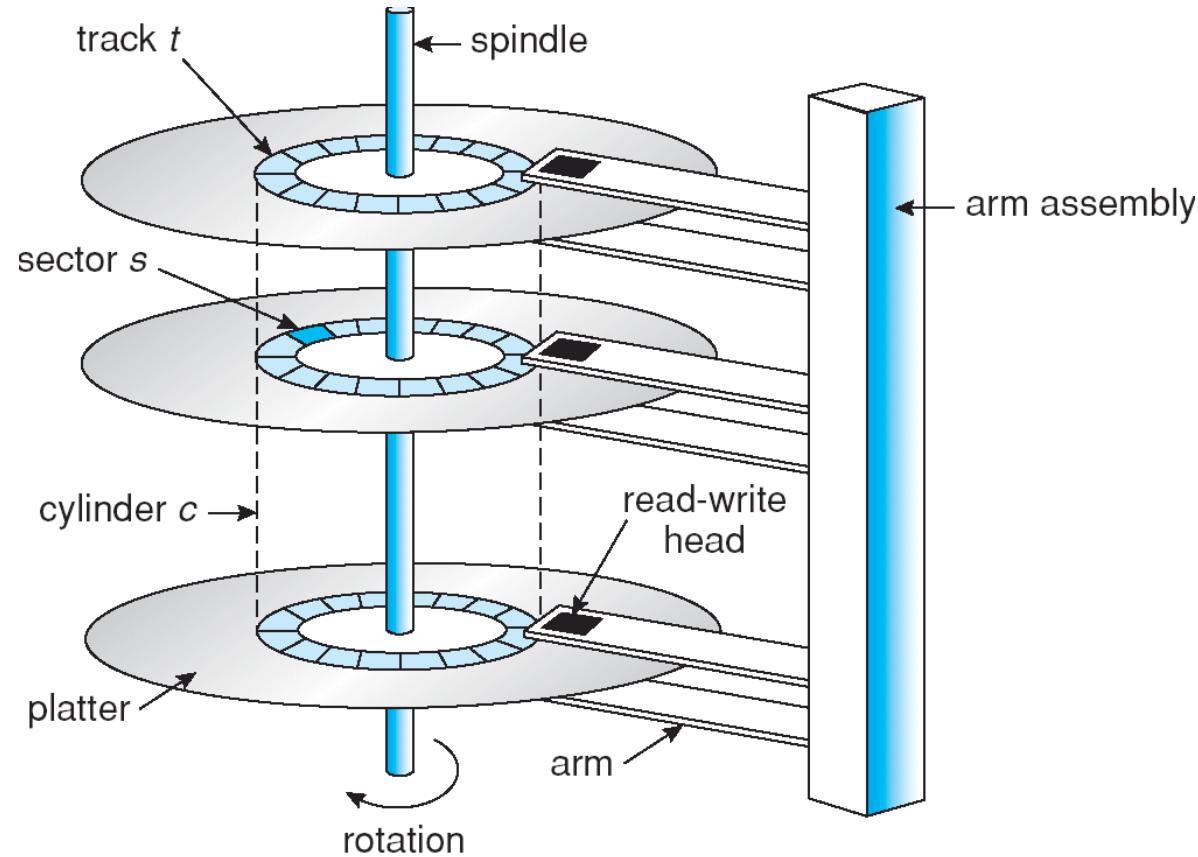
- Tools: building, exploring, and debugging Linux kernel
- Core kernel infrastructure
 - syscall, module, kernel data structures
- Process management & scheduling

Today: “interrupt”

- A mechanism to implement abstraction and multiplexing
- Interrupt: asking for a service to the kernel
 - by software (e.g., `int`) or by hardware (e.g., keyboard)
- Interrupt handling in Linux
 - top half + bottom half

Yeah! New hard disk drive!

HDD architecture



How fast is my new HDD?

- HDD access time = seek time + rotational latency
- Seek time
 - The time to move the disk head to the track that contains data
 - Average seek time: 4 ~ 10 msec
- Rotational latency
 - The delay for the rotation of the disk to bring the required disk sector under the read-write mechanism
 - 7200 RPM: 4.16 msec
- **Access time of your new HDD: about 10 msec**

Interrupt

- **Compared to the CPU, devices are slow to respond (e.g., 10 msec)**
 - The kernel must be free to go and handle other work, dealing with the hardware only after that hardware has completed its work
- **How to know the completion of a hardware operation**
 - **Polling:** the kernel periodically checks the status of hardware
 - **Interrupt:** the hardware signals its completion to the processor
- Interrupt examples
 - Completion of disk read
 - Key press on a keyboard, network packet arrival

Interrupt controller



- Interrupts are electrical signals multiplexed by the interrupt controller
 - Sent on a specific pin of the CPU
- Once an interrupt is received, a dedicated function is executed
 - **Interrupt handler**
- The kernel/user space can be interrupted at (nearly) any time to process an interrupt

Advanced PIC (APIC, I/O APIC)

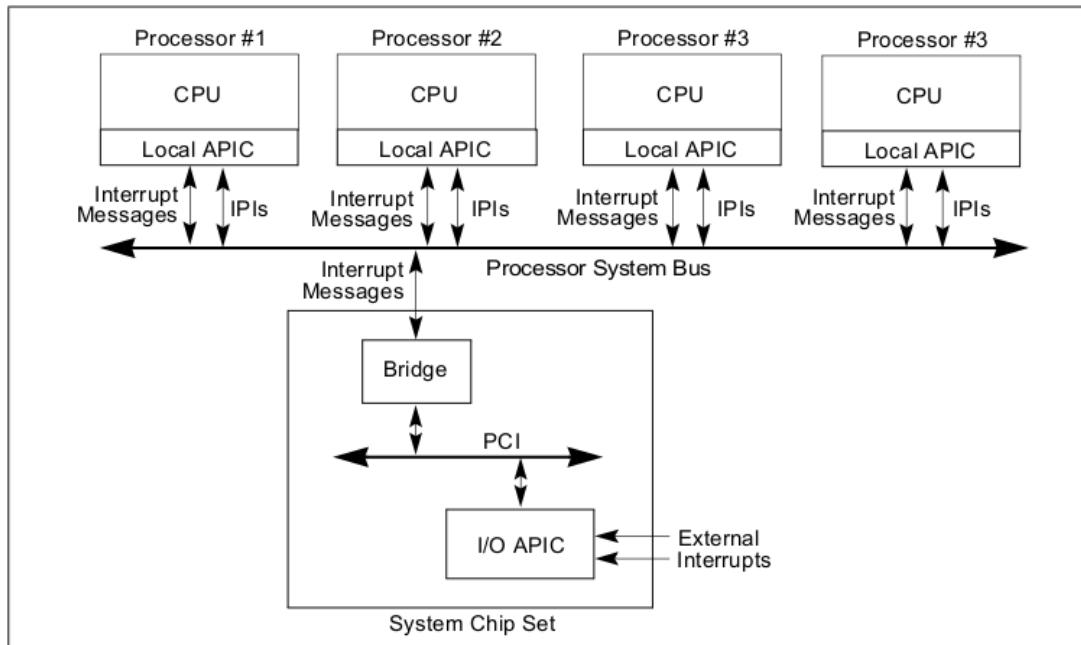
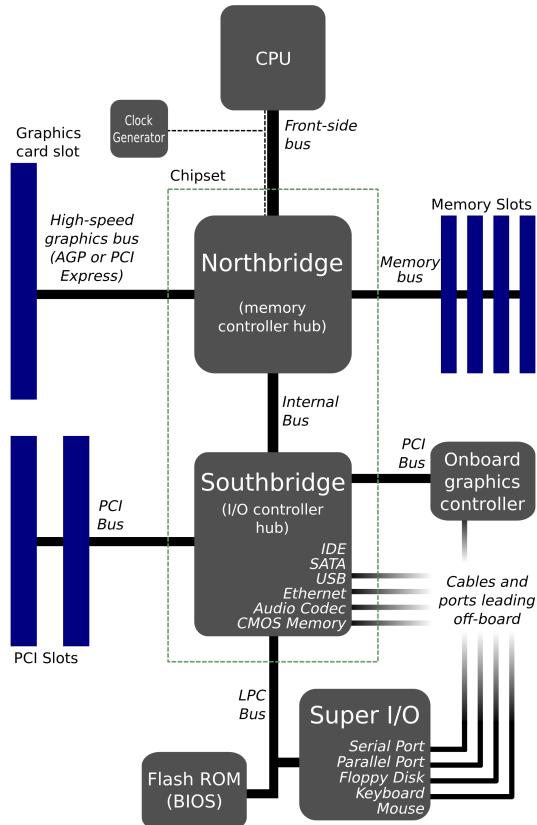


Figure 10-2. Local APICs and I/O APIC When Intel Xeon Processors Are Used in Multiple-Processor Systems

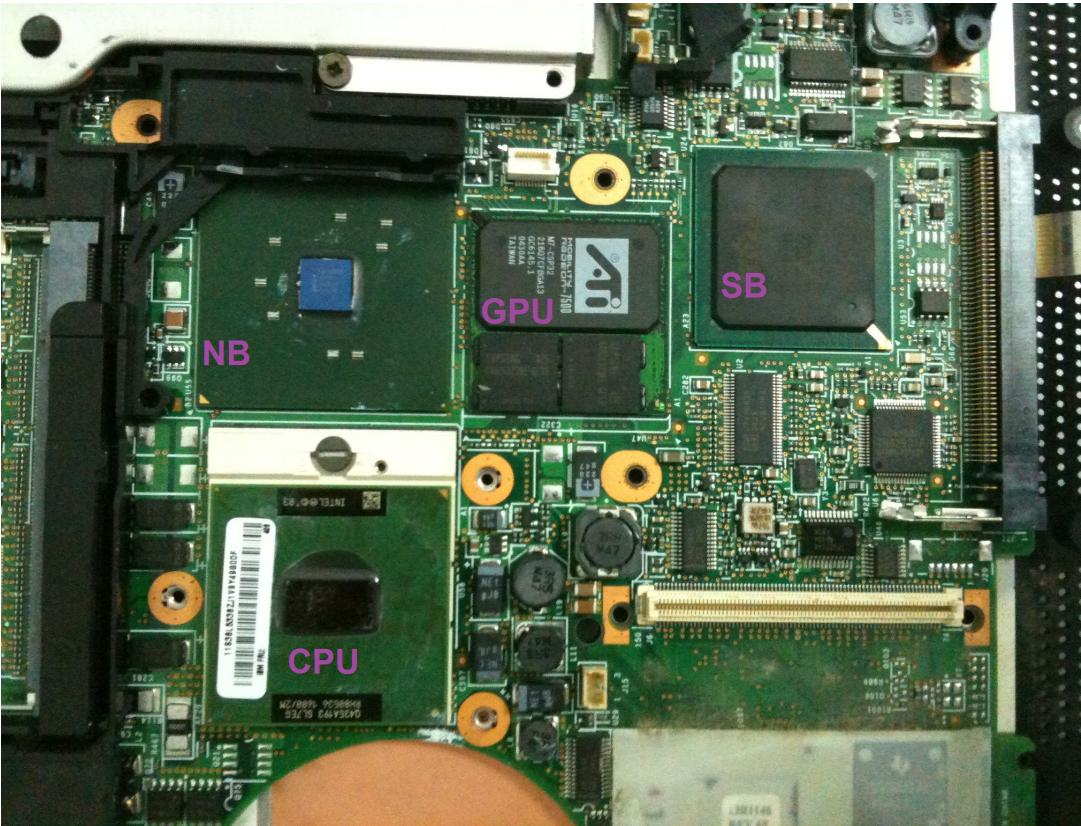
Advanced PIC (APIC, I/O APIC)

- I/O APIC
 - system chipset (or south bridge)
 - redistribute interrupts to local APICs
- Local APIC
 - inside a processor chip
 - has a timer, which raises timer interrupt
 - issues a IPI (inter-process interrupt)

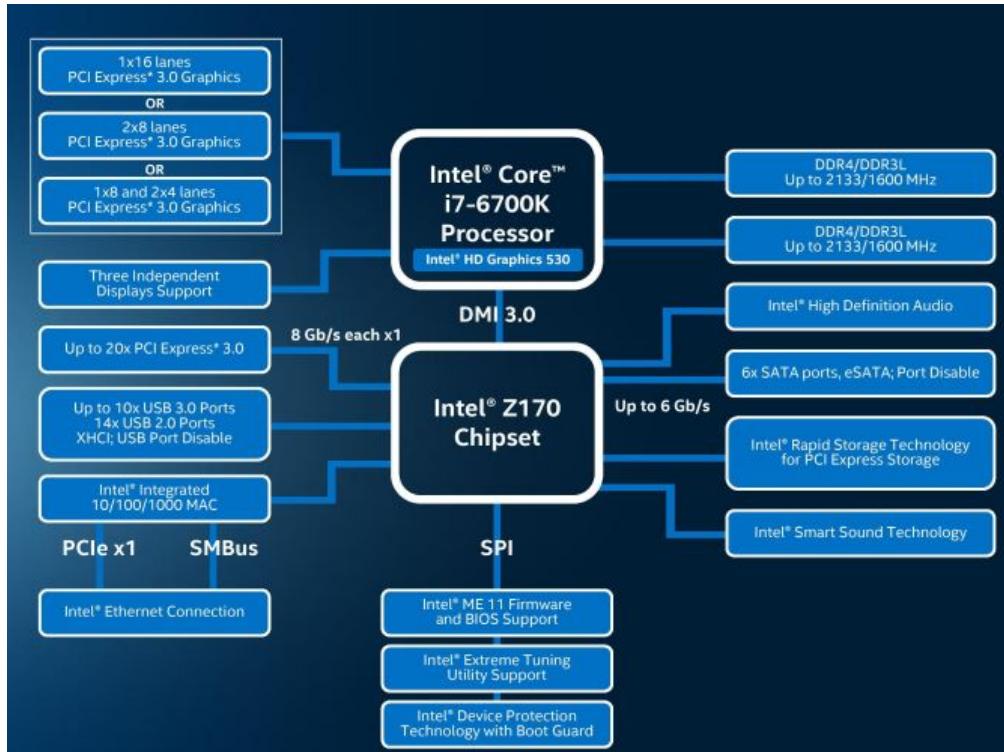
A bigger picture



A real motherboard (T42)



Intel Z170 chipset



- Ref: [The Intel 6th Gen Skylake Review](#)

Interrupt request (IRQ)

- **Interrupt line** or **interrupt request (IRQ)**
 - device identifier
- E.g., 8259A interrupt lines
 - IRQ 0: system timer, IRQ 1: keyboard controller
 - IRQ 3, 4: serial port, IRQ 5: terminal
- Some interrupt lines can be shared among several devices
 - True for most modern devices (PCIe)

Exception

- **Exception** are interrupt issued by the CPU executing some code
 - **Software interrupts**, as opposed to hardware ones (devices)
 - Examples:
 - **Program faults**: divide-by-zero, page fault, general protection fault, etc
 - **Voluntary exceptions**: `int` assembly instruction, for example for syscall invocation
- Exceptions are managed by the kernel the same as hardware interrupts

Interface to hardware interrupt

- **Non-Maskable Interrupt (NMI)**
 - Never ignored, e.g., power failure, memory error
 - In x86, vector 2, prevents other interrupts from executing.
- **Maskable interrupt**
 - Ignored when `IF` in `EFLAGS` is 0
 - Enabling/disabling:- `sti` : set interrupt - `cli` : clear interrupt
- **INTA**
 - Interrupt acknowledgement
 - EOI (end of interrupt)

“Software” interrupt: INT

- Intentionally interrupts
 - x86 provides the `INT` instruction
 - Invokes the interrupt handler for the vector (0-255)
- Entering: `int N`
- Exiting: `iret`

Interrupt vector

- Telling `int N`
 - N-th interrupt handler (e.g., `int 0 → vector 0`)

Interrupt descriptor table

- **IDT**
 - Table of 256 8-byte entries (similar to GDT)
 - Located anywhere in memory
- **IDTR register**
 - Stores current IDT
- **lidt instruction**
 - Loads IDTR with address and size of the IDT
 - Takes in a linear address

Interrupt descriptor table

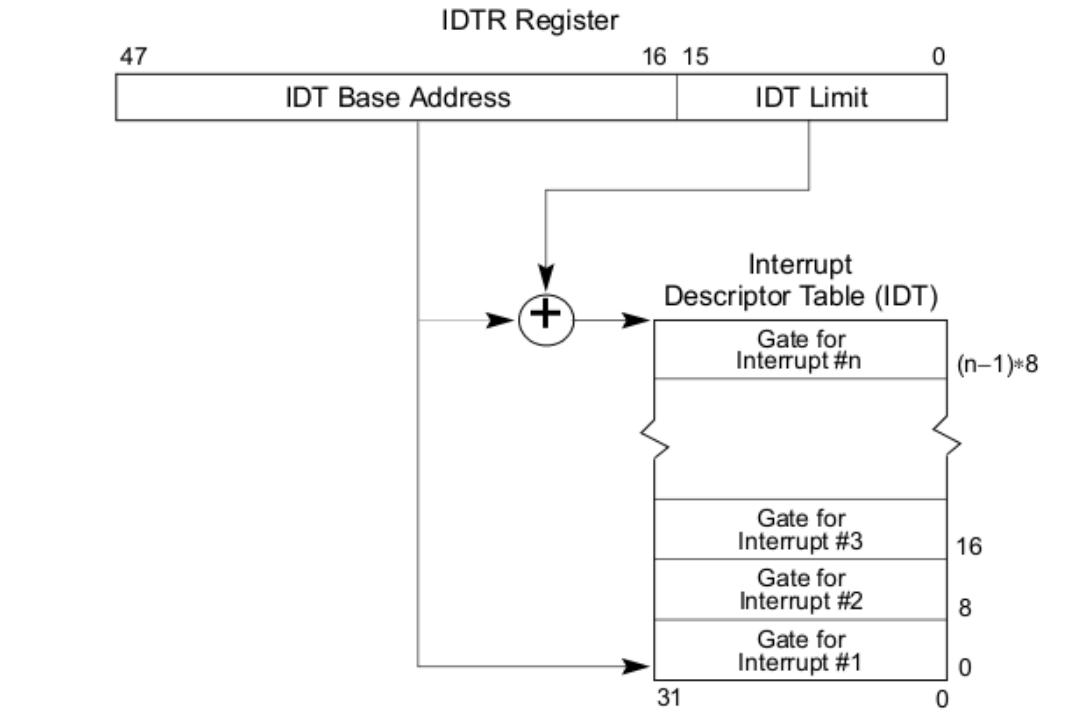


Figure 6-1. Relationship of the IDTR and IDT

Interrupt descriptor entry

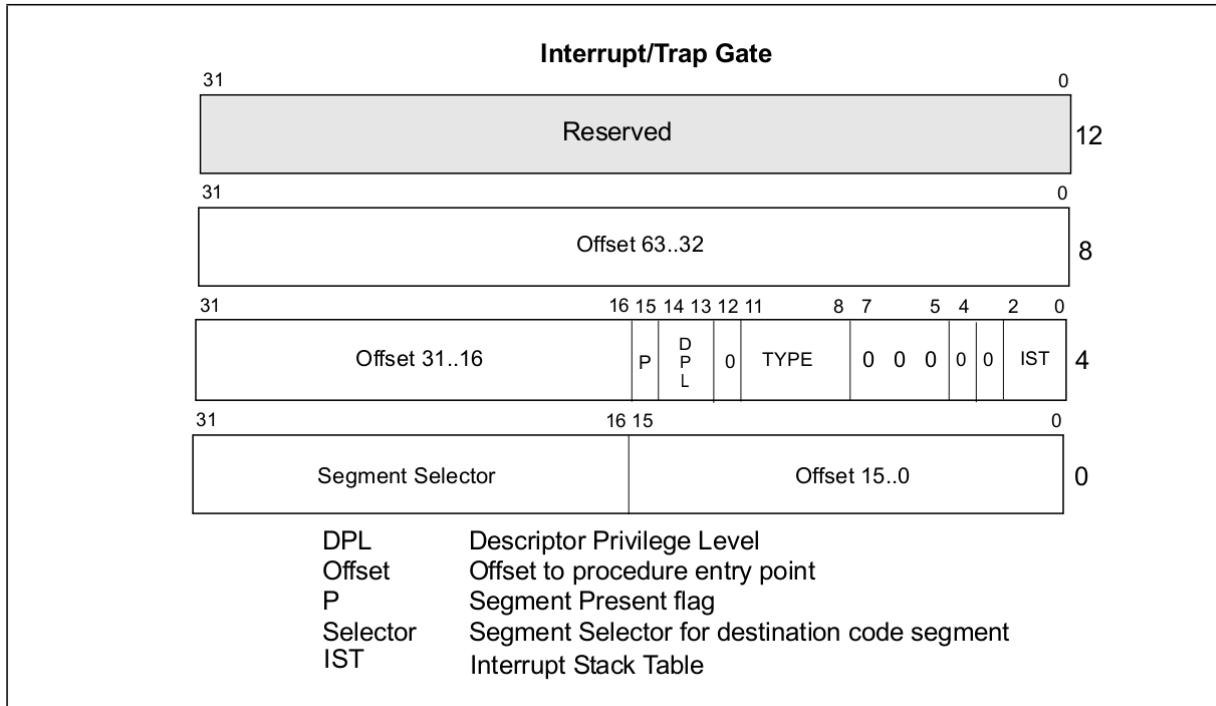


Figure 6-7. 64-Bit IDT Gate Descriptors

Interrupt descriptor entry

- Offset is a 32-bit value split into two parts pointing to the destination IP or EIP
- Segment selector points to the destination CS in the kernel
- Present flag indicates that this is a valid entry
- Descriptor Privilege Level indicates the minimum privilege level of the caller to prevent users from calling hardware interrupts directly
- Size of gate can be 32 bits or 16 bits

Interrupt descriptor table

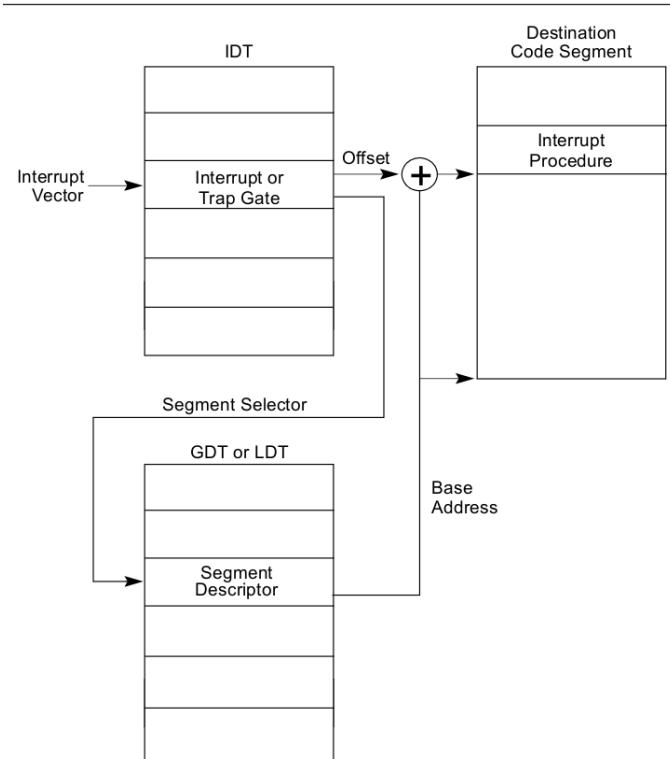


Figure 6-3. Interrupt Procedure Call

Predefined interrupt vectors

- 0 : Divide Error
- 1 : Debug Exception
- 2 : Non-Maskable Interrupt
- 3 : Breakpoint Exception (e.g., `int 3`)
- 4 : Invalid Opcode
- 13 : General Protection Fault
- 14 : Page Fault
- 18 : Machine Check (abort)
- 32–255 : User Defined Interrupts

The INT instruction (1/2)

- Decide the vector number: e.g., `int 0x80`
- Fetch the interrupt descriptor for vector 0x80 from the `IDT`. The CPU finds it by taking the 0x80'th 8-byte entry starting at the physical address that the `IDTR` CPU register points to.
- Check if $CPL \leq DPL$ in the descriptor.
- Save ESP and SS in a CPU-internal register.

The INT instruction (2/2)

- Load SS and ESP from TSS (Task State Segment)
- Push user SS
- Push user ESP
- Push user EFLAGS
- Push user CS
- Push user EIP
- Clear some EFLAGS bits
- Set CS and EIP from IDT descriptor's segment selector and offset

Interrupt service routine (ISR)

- **Interrupt handler or Interrupt Service Routine (ISR)**
 - function executed by the CPU in response to a specific interrupt
- Runs in **interrupt context (or atomic context)**
 - Opposite to process context (system call)
 - A task cannot sleep in an ISR because an interrupt context is not a schedulable entity

Two conflicting goals of ISR

1. Interrupt processing must be fast

- We are indeed interrupting user processes executing (user/kernel space)
- Some other interrupts may need to be disabled while processing an interrupt

2. Sometimes it requires significant amount of work

- So it will take time
- E.g., processing a network packet from the network card

Top half vs. bottom half

- In many modern OS including Linux, an interrupt processing is split into two parts:
- **Top-half:** run immediately upon receipt of the interrupt
 - Performs only the time-critical operations
 - E.g., Acknowledging receipt of the interrupt
 - resetting the hardware
- **Bottom-half:** less critical & time-consuming work
 - Run later with other interrupts enabled

Example: network packet processing

- **Top-half: interrupt service routine**
 - Acknowledges the hardware
 - Copies the new network packets into main memory
 - Makes the network card ready for more packets
 - Time critical because the packet buffer on the network card is limited in size → packet drop
- **Bottom-half: processing the copied packets**
 - Softirq, tasklet, work queue
 - Similar to thread pool in user-space

Registering an interrupt handler

```
/* linux/include/linux/interrupt.h */

/***
 * This call allocates interrupt resources and enables the
 * interrupt line and IRQ handling.
 *
 * @irq: Interrupt line to allocate
 * @handler: Function to be called when the IRQ occurs.
 *           Primary handler for threaded interrupts
 * @irqflags: Interrupt type flags
 *           IRQF_SHARED - allow sharing the irq among several devices
 *           IRQF_TIMER - Flag to mark this interrupt as timer interrupt
 *           IRQF_TRIGGER_* - Specify active edge(s) or level
 * @devname: An ascii name for the claiming device
 * @dev_id: A cookie passed back to the handler function
 *          Normally the address of the device data structure
 *          is used as the cookie.
 */
int request_irq(unsigned int irq, irq_handler_t handler,
                unsigned long irqflags, const char *devname, void *dev_id);
```

Freeing an interrupt handler

```
/* linux/include/linux/interrupt.h */

/***
 *  Free an interrupt allocated with request_irq
 *
 *  @irq: Interrupt line to free
 *  @dev_id: Device identity to free
 *
 *  Remove an interrupt handler. The handler is removed and if the
 *  interrupt line is no longer in use by any driver it is disabled.
 *  On a shared IRQ the caller must ensure the interrupt is disabled
 *  on the card it drives before calling this function. The function
 *  does not return until any executing interrupts for this IRQ
 *  have completed.
 *
 *  Returns the devname argument passed to request_irq.
 */
const void *free_irq(unsigned int irq, void *dev_id);
```

Writing an interrupt handler

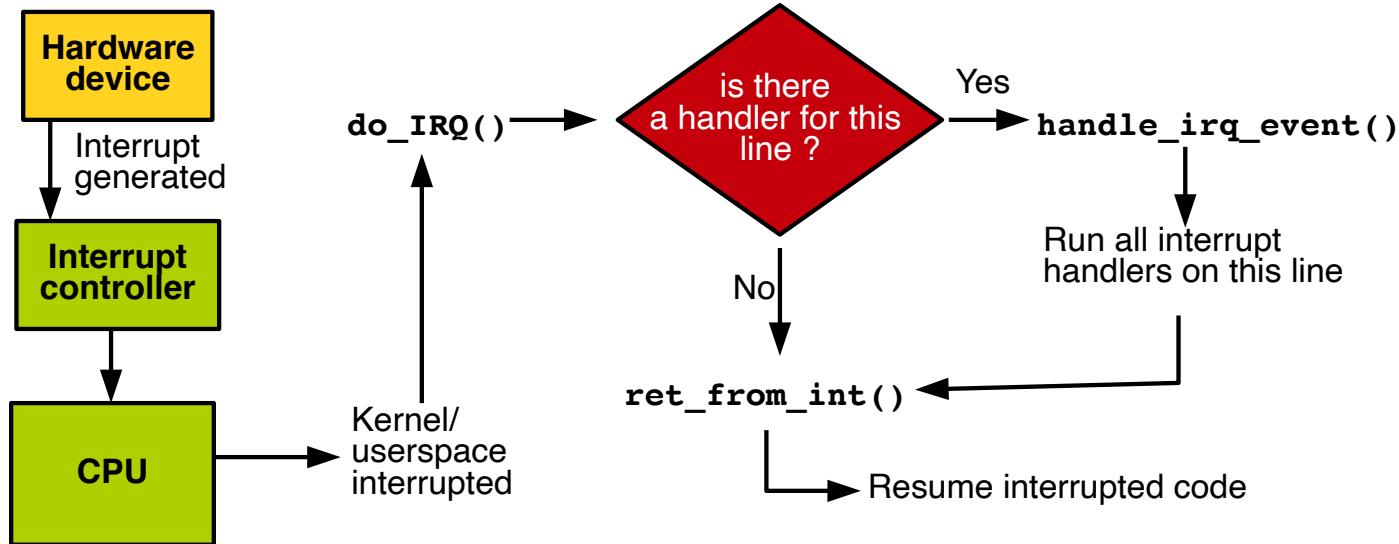
```
/* linux/include/linux/interrupt.h */

/***
 * Interrupt handler prototype
 *
 * @irq: the interrupt line number that the handler is serving
 * @dev_id: a generic pointer that was given to request_irq()
 *          when the interrupt handler is registered
 *
 * Return value:
 *      IRQ_NONE: the interrupt is not handled (i.e., the expected
 *                device was not the source of the interrupt)
 *      IRQ_HANDLED: the interrupt is handled (i.e., the hanlder was
 *                  correctly invoked)
 * #define IRQ_RETVAL(x)  ((x) ? IRQ_HANDLED : IRQ_NONE)
 *
 * NOTE: interrupt handlers need not be reentrant (tread-safe)
 *       - When a given interrupt handler is executing, the corresponding
 *         interrupt line is disabled on all cores while.
 *       - Normally all other interrupts are enables, os other interrupts
 *         are serviced.
 */
typedef irqreturn_t (*irq_handler_t)(int irq, void *dev_id);
```

Interrupt context

- Process context: normal task execution, syscall, and exception
- Interrupt context: interrupt service routine (ISR)
 - **Sleeping/blocking is not possible** because the ISR is not a schedule entity
 - No `kmalloc(size, GFP_KERNEL)` : use `GFP_ATOMIC`
 - No blocking locking (e.g., mutex): use spinlock
 - No `printk` : use `trace_printk`
- Small stack size
 - Interrupt stack: one page (4KB)

Interrupt handling internals in Linux



Interrupt handling internals in Linux

- Specific entry point for each interrupt line
 - Saves the interrupt number and the current registers
 - Calls `do_IRQ()`
- `unsigned int do_IRQ(struct pt_regs *regs)`
 - Acknowledge interrupt reception and disable the line
 - Calls architecture specific functions

Interrupt handling internals in Linux

- Call chain ends up by calling `generic_handle_irq_desc()`
 - Call the handler if the line is not shared
 - Otherwise iterate over all the handlers registered on that line
 - Disable interrupts on the line again if they were previously enabled
- `do_IRQ()` returns to entry point that call `ret_from_intr()`
 - Checks if reschedule is needed (`need_resched`)
 - Restore register values

do_IRQ() in QEMU/gdb

```
--arch/x86/kernel/irq.c
198     }
199
200     u64 arch_irq_stat(void)
201     {
202         u64 sum = atomic_read(&irq_err_count);
203         return sum;
204     }
205
206
207     /*
208      * do_IRQ handles all normal device IRQ's (the special
209      * SMP cross-CPU interrupts have their own specific
210      * handlers).
211      */
212     __visible unsigned int __irq_entry do_IRQ(struct pt_regs *regs)
B+> 213     {
214         struct pt_regs *old_regs = set_irq_regs(regs);
215         struct irq_desc * desc;
216         /* high bit used in ret_from_code */
217         unsigned vector = ~regs->orig_ax;
218
219         /*
220          * NB: Unlike exception entries, IRQ entries do not reliably
221          * handle context tracking in the low-level entry code. This is
222          * because syscall entries execute briefly with IRQs on before
223          * updating context tracking state, so we can take an IRQ from
224          * kernel mode with CONTEXT_USER. The low-level entry code only
225          * updates the context if we came from user mode, so we won't
226          * switch to CONTEXT_KERNEL. We'll fix that once the syscall
227          * code is cleaned up enough that we can cleanly defer enabling
228          * IRQs.
229         */
remote Thread 1 In: do_IRQ
(gdb) bt
#0  do_IRQ (regs=0xfffffc9000031fc38) at arch/x86/kernel/irq.c:213
#1  0xffffffff8194f746 in common_interrupt () at arch/x86/entry/entry_64.S:512
#2  0xfffffc9000031fc38 in ?? ()
#3  0x0000000000000000 in ?? ()
(gdb) L213 PC: 0xffffffff8101e170
```

Initializing IDT

```
/* linux/arch/x86/include/asm/desc_defs.h */
struct gate_struct {
    u16      offset_low;
    u16      segment;
    struct idt_bits bits;
    u16      offset_middle;
#ifdef CONFIG_X86_64
    u32      offset_high;
    u32      reserved;
#endif
} __attribute__((packed));

typedef struct gate_struct gate_desc;

/* linux/arch/x86/kernel/traps.c */
DECLARE_BITMAP(system_vectors, NR_VECTORS);
```

Initializing IDT

```
/* linux/arch/x86/include/asm/idtentry.h
/*
 * Build the entry stubs with some assembler magic.
 * We pack 1 stub into every 8-byte block.
 */
.align 8
SYM_CODE_START(irq_entries_start)
    vector=FIRST_EXTERNAL_VECTOR
    .rept (FIRST_SYSTEM_VECTOR - FIRST_EXTERNAL_VECTOR)
    UNWIND_HINT_IRET_REGS
0 :
    .byte    0x6a, vector
    jmp asm_common_interrupt
    nop
    /* Ensure that the above is 8 bytes max */
    . = 0b + 8
    vector = vector+1
    .endr
SYM_CODE_END(irq_entries_start)
```

- Let's see how `gate_desc` is initialized during boot.

Initializing IDT

```
/* linux/init/main.c */

asmlinkage __visible void __init start_kernel(void)
{
    /* ... */
    early_irq_init();
    init_IRQ();
    /* ... */
}

/* linux/arch/x86/kernel/irqinit.c */
void __init init_IRQ(void)
{
    int i;
    for (i = 0; i < nr_legacy_irqs(); i++)
        per_cpu(vector_irq, 0)[ISA_IRQ_VECTOR(i)] = irq_to_desc(i);

    BUG_ON(irq_init_percpu_irqstack(smp_processor_id()));

    x86_init.irqs.intr_init();
}
```

Initializing IDT

```
/* linux/arch/x86/kernel/irqinit.c */
void __init native_init_IRQ(void)
{
    /* Execute any quirks before the call gates are initialised: */
    x86_init.irqs.pre_vector_init();

    idt_setup_apic_and_irq_gates();
    lapic_assign_system_vectors();

    if (!acpi_ioapic && !of_ioapic && nr_legacy_irqs())
        setup_irq(2, &irq2);
}
```

Initializing IDT

```
/* linux/arch/x86/kernel/idt.c */
void __init idt_setup_apic_and_irq_gates(void)
{
    int i = FIRST_EXTERNAL_VECTOR;
    void *entry;

    idt_setup_from_table(idt_table, apic_idts, ARRAY_SIZE(apic_idts), true);

    for_each_clear_bit_from(i, system_vectors, FIRST_SYSTEM_VECTOR) {
        entry = irq_entries_start + 8 * (i - FIRST_EXTERNAL_VECTOR);
        set_intr_gate(i, entry);
    }
}
```

Initializing IDT

```
/* linux/arch/x86/kernel/idt.c */
static void set_intr_gate(unsigned int n, const void *addr)
{
    struct idt_data data;

    BUG_ON(n > 0xFF);

    memset(&data, 0, sizeof(data));
    data.vector = n;
    data.addr = addr;
    data.segment = __KERNEL_CS;
    data.bits.type = GATE_INTERRUPT;
    data.bits.p = 1;

    idt_setup_from_table(idt_table, &data, 1, false);
}
```

/proc/interrupts

```
$ cat /proc/interrupts
# Int line
# |           Num of occurrence per CPU
# |           |
# |           Int controller
# |           |
# |           Edge/level
# |           |
# |           |
# |           Device name
# |           |
# |           |
CPU0          CPU1
 0:      34          0  IO-APIC   2-edge    timer
 1:      26          8  IO-APIC   1-edge    i8042
 8:      0          0  IO-APIC   8-edge    rtc0
 9:      0          0  IO-APIC   9-fasteoi acpi
12:     156          0  IO-APIC  12-edge    i8042
14:      0          0  IO-APIC  14-edge   ata_piix
15:     116         24  IO-APIC  15-edge   ata_piix
19:      5         68  IO-APIC  19-fasteoi virtio0
21:    3142        533  IO-APIC  21-fasteoi ahci[0000:00:0d.0], snd_intel8x0
22:     27          0  IO-APIC  22-fasteoi ohci_hcd:usb1
NMI:      0          0  Non-maskable interrupts
LOC:    5031       4373  Local timer interrupts
PMI:      0          0  Performance monitoring interrupts
TLB:     27         89  TLB shootdowns
```

Interrupt control

- Kernel code sometimes needs to disable interrupts to ensure atomic execution of a section of code
 - By disabling interrupts, you can guarantee that an interrupt handler will not preempt your current code.
 - Moreover, disabling interrupts also disables kernel preemption.
- Note that disabling interrupts does not protect against concurrent access from other cores
 - Need locking, often used in conjunction with interrupts disabling
- The kernel provides an API to disable/enable interrupts

Disabling interrupts on the local core

- Disable and enable IRQ

```
local_irq_disable();
/* interrupts are disable ... */
local_irq_enable();
```

- What happen if local_irq_disable() is called twice?

```
local_irq_disable(); /* interrupt is disabled */
local_irq_disable(); /* no effect */
/* ... */
local_irq_enable(); /* interrupt is eanbled! */
local_irq_enable(); /* no effect */
```

Disabling interrupts on the local core

- Use `local_irq_save()`

```
unsigned long flags;
local_irq_save(flags);      /* disables interrupts if needed */
/* ... */
local_irq_restore(flags); /* restore interrupt status to the previous */

/* nesting is okay */
unsigned long flags;
local_irq_save(flags);
{
    unsigned long flags;
    local_irq_save(flags);
    /* ... */
    local_irq_restore(flags);
}
local_irq_restore(flags);
```

Disabling a specific interrupt line

```
/** disable_irq - disable an irq and wait for completion
 * @irq: Interrupt to disable
 *
 * Disable the selected interrupt line. Enables and Disables are
 * nested.
 * This function waits for any pending IRQ handlers for this interrupt
 * to complete before returning. If you use this function while
 * holding a resource the IRQ handler may need you will deadlock.
 *
 * This function may be called - with care - from IRQ context. */
void disable_irq(unsigned int irq);

/** disable_irq_nosync - disable an irq without waiting
 * @irq: Interrupt to disable */
void disable_irq_nosync(unsigned int irq);

/** enable_irq - enable handling of an irq
 * @irq: Interrupt to enable
 *
 * Undoes the effect of one call to disable_irq(). If this
 * matches the last disable, processing of interrupts on this
 * IRQ line is re-enabled. */
void enable_irq(unsigned int irq);
```

Status of the interrupt system

```
/* linux/include/linux/preempt.h */

/*
 * Are we doing bottom half or hardware interrupt processing?
 *
 * in_irq()          - We're in (hard) IRQ context
 * in_interrupt()    - We're in NMI,IRQ,SoftIRQ context or have BH disabled
 * in_nmi()          - We're in NMI context
 * in_softirq()      - We have BH disabled, or are processing softirqs
 * in_serving_softirq() - We're in softirq context
 * in_task()          - We're in task context
 */
#define in_irq()          (hardirq_count())
#define in_interrupt()    (irq_count())
#define in_nmi()          (preempt_count() & NMI_MASK)
#define in_softirq()      (softirq_count())
#define in_serving_softirq() (softirq_count() & SOFTIRQ_OFFSET)
#define in_task()          (! (preempt_count() & \
                           (NMI_MASK | HARDIRQ_MASK | SOFTIRQ_OFFSET)))
```

Further readings

- [LWN: Debugging the kernel using Ftrace - part 1](#)
- [0xAx: Interrupts and Interrupt Handling](#)

Next lecture

- Interrupt handler: bottom half

Interrupt Handler: Bottom Half

Dongyoon Lee

Summary of last lectures

- Tools: building, exploring, and debugging Linux kernel
- Core kernel infrastructure
 - syscall, module, kernel data structures
- Process management & scheduling
- Interrupt & interrupt handler (top half)

Interrupt controller



- Interrupts are electrical signals multiplexed by the interrupt controller
 - Sent on a specific pin of the CPU
- Once an interrupt is received, a dedicated function is executed
 - **Interrupt handler**
- The kernel/user space can be interrupted at (nearly) any time to process an interrupt

Advanced PIC (APIC, I/O APIC)

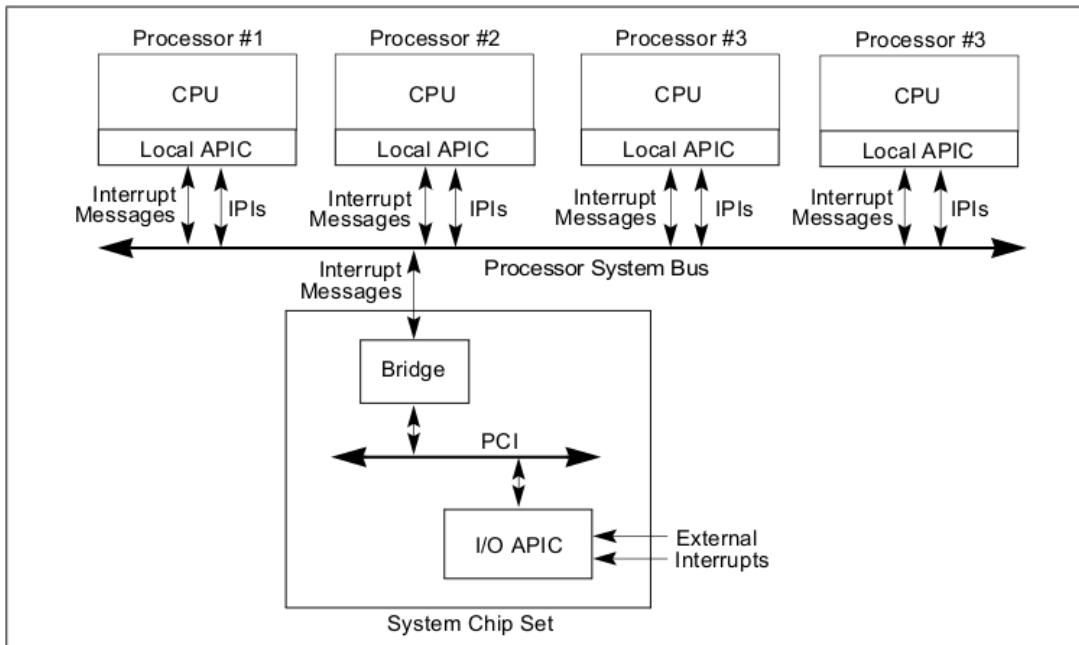


Figure 10-2. Local APICs and I/O APIC When Intel Xeon Processors Are Used in Multiple-Processor Systems

Interrupt descriptor table

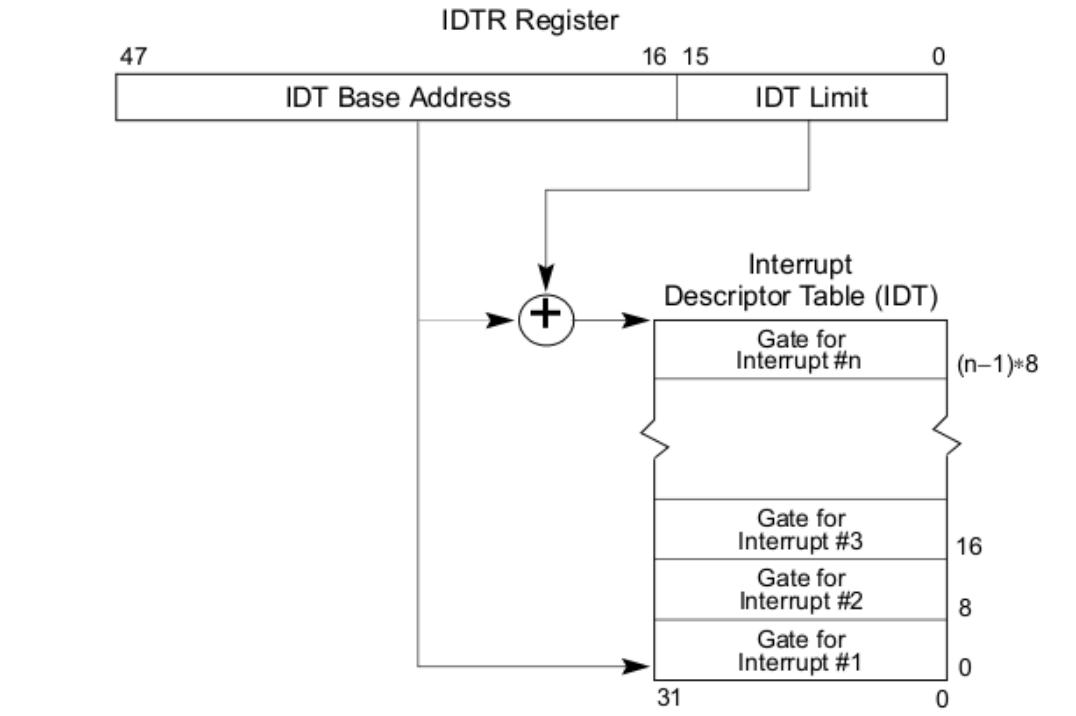


Figure 6-1. Relationship of the IDTR and IDT

Interrupt descriptor table

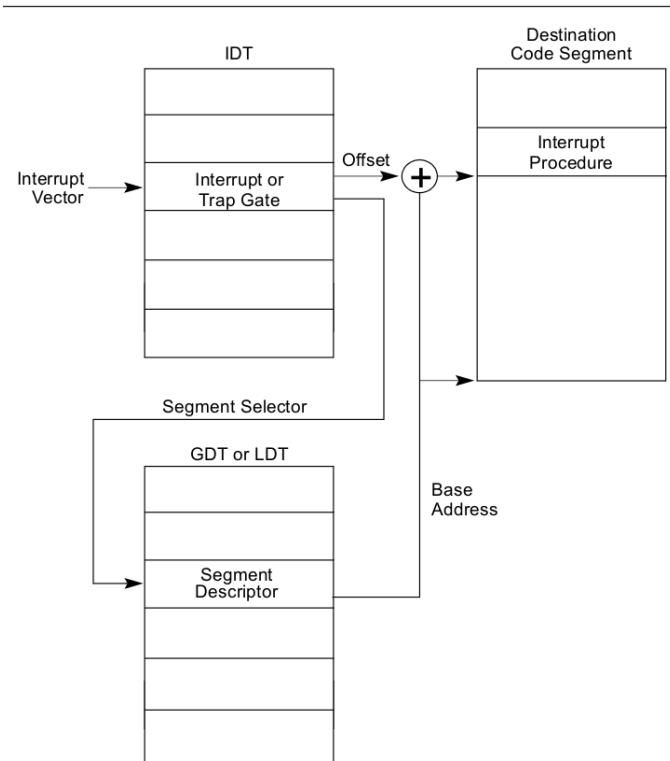
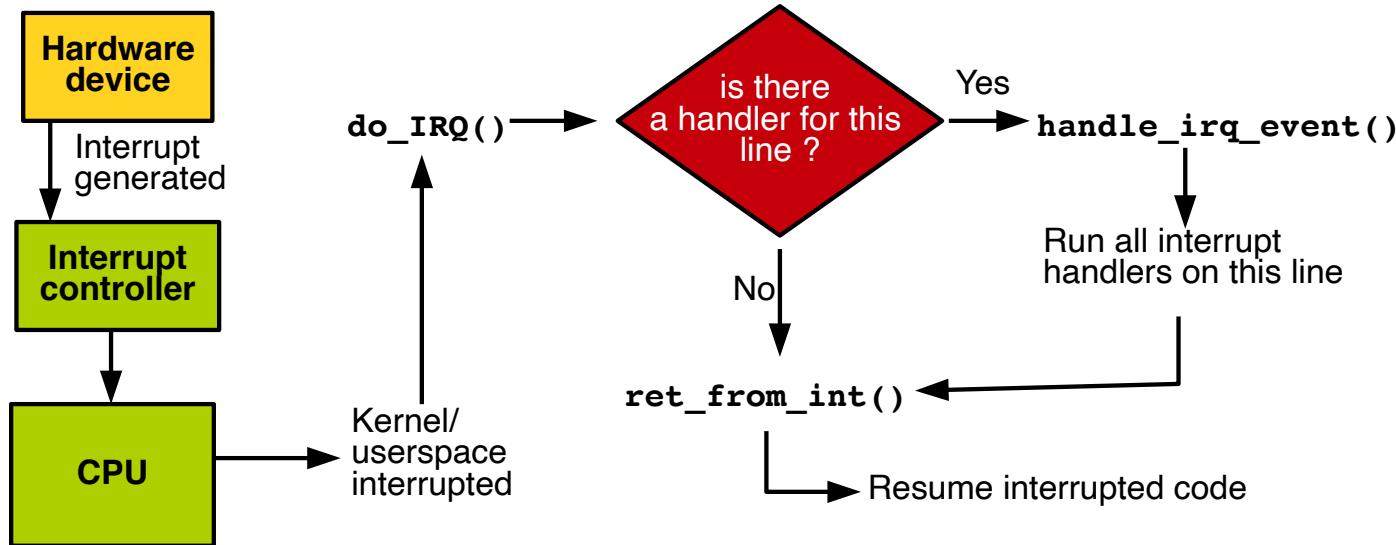


Figure 6-3. Interrupt Procedure Call

Interrupt handling internals in Linux



Today: interrupt handler

- **Top-halves (interrupt handlers)** must run as quickly as possible
 - They are interrupting other kernel/user code
 - They are often timing-critical because they deal with hardware
 - They run in interrupt context: they cannot block
 - One or all interrupt lines are disabled
- Defer the less critical part of interrupt processing to a **bottom-half**

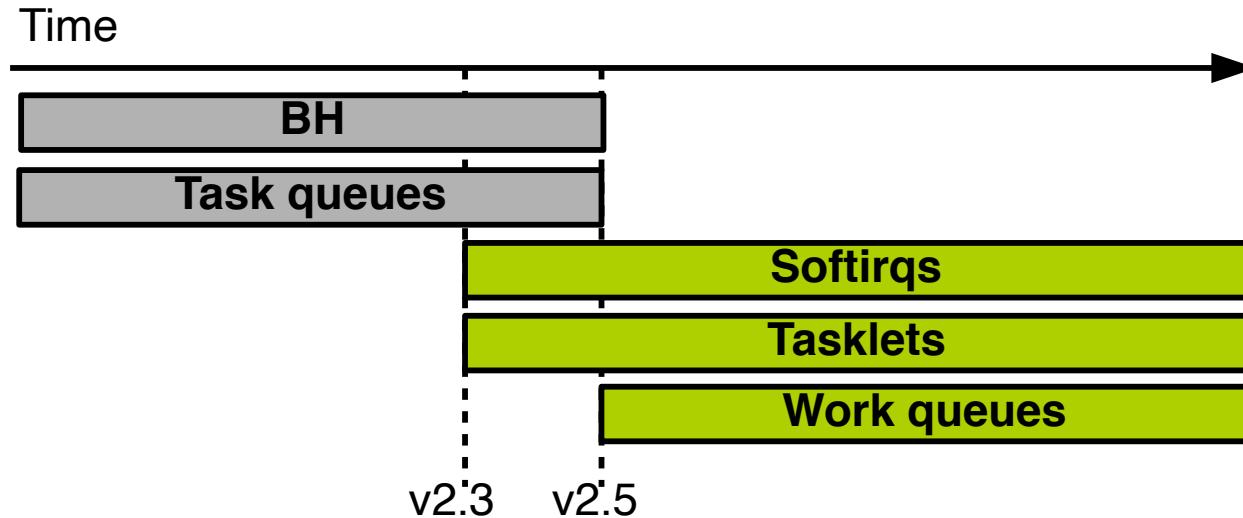
Top-halves vs. bottom-halves

- **When to use top halves?**
 - Work is time sensitive
 - Work is related to controlling the hardware
 - Work should not be interrupted by other interrupt
 - The top half is quick and simple, and runs with **some or all interrupts disabled**
- **When to use bottom halves?**
 - Everything else
 - The bottom half runs later with **all interrupts enabled**

The history of bottom halves in Linux

- “Top-half” and “bottom-half” are generic terms not specific to Linux
- Old “Bottom-Half” (BH) mechanism
 - a statistically created list of 32 bottom halves
 - globally synchronized
 - easy-to-use yet inflexible and a performance bottleneck
- Task queues: queues of function pointers
 - still too inflexible
 - not lightweight enough for performance-critical subsystems (e.g., networking)

The history of bottom halves in Linux



- BH → Softirq, tasklet
- Task queue → work queue

Today's bottom halves in Linux



- All bottom-half mechanisms run with all interrupts enabled
- **Softirqs and tasklets** run in interrupt context
 - Softirq is rarely used directly
 - Tasklet is a simple and easy-to-use softirq (built on softirq)
- **Work queues** run in process context
 - They can block and go to sleep

Softirq

```
/* linux/kernel/softirq.c */
/* Softirq is statically allocated at compile time */
static struct softirq_action softirq_vec[NR_SOFTIRQS]; /* softirq vector */

/* linux/include/linux/interrupt.h */
enum {
    HI_SOFTIRQ=0,          /* [highest priority] high-priority tasklet */
    TIMER_SOFTIRQ,         /* timer */
    NET_TX_SOFTIRQ,        /* send network packets */
    NET_RX_SOFTIRQ,        /* receive network packets */
    BLOCK_SOFTIRQ,         /* block devices */
    IRQ_POLL_SOFTIRQ,      /* interrupt-poll handling for block device */
    TASKLET_SOFTIRQ,       /* normal priority tasklet */
    SCHED_SOFTIRQ,         /* scheduler */
    HRTIMER_SOFTIRQ,       /* unused */
    RCU_SOFTIRQ,           /* [lowest priority] RCU locking */
    NR_SOFTIRQS            /* the number of defined softirq (< 32) */
};

struct softirq_action {
    void    (*action)(struct softirq_action *);      /* softirq handler */
};
```

Executing softirqs

- **Raising the softirq**
 - Mark the execution of a particular softirq is needed
 - Usually, a top-half marks its softirq for execution before returning
- **Pending softirqs are checked and executed in following places:**
 - In the return from hardware interrupt code path
 - In the `ksoftirqd` kernel thread
 - In any code that explicitly checks for and executes pending softirqs

Executing softirqs: **do_softirq()**

- Goes over the softirq vector and executes the pending softirq handler

```
/* linux/kernel/softirq.c */
/* do_softirq() calls __do_softirq() */
void __do_softirq(void) /* much simplified version for explanation */
{
    u32 pending;
    pending = local_softirq_pending(); /* 32-bit flags for pending softirq */
    if (pending) {
        struct softirq_action *h;

        set_softirq_pending(0); /* reset the pending bitmask */
        h = softirq_vec;
        do {
            if (pending & 1)
                h->action(h); /* execute the handler of the pending softirq */
            h++;
            pending >>= 1;
        } while (pending);
    }
}
```

Using softirq: assigning an index

```
/* linux/include/linux/interrupt.h */
enum {
    HI_SOFTIRQ=0,          /* [highest priority] high-priority tasklet */
    TIMER_SOFTIRQ,         /* timer */
    NET_TX_SOFTIRQ,        /* send network packets */
    NET_RX_SOFTIRQ,        /* receive network packets */
    BLOCK_SOFTIRQ,         /* block devices */
    IRQ_POLL_SOFTIRQ,      /* interrupt-poll handling for block device */
    TASKLET_SOFTIRQ,        /* normal priority tasklet */
    SCHED_SOFTIRQ,          /* scheduler */
    HRTIMER_SOFTIRQ,        /* unused */
    RCU_SOFTIRQ,            /* [lowest priority] RCU locking */

    YOUR_NEW_SOFTIRQ, /* TODO: add your new softirq index */

    NR_SOFTIRQS          /* the number of defined softirq (< 32) */
};
```

Using softirq: registering a handler

```
/* linux/kernel/softirq.c */
/* register a softirq handler for nr */
void open_softirq(int nr, void (*action)(struct softirq_action *))
{
    softirq_vec[nr].action = action;
}

/* linux/net/core/dev.c */
static int __init net_dev_init(void)
{
    /* ... */
    /* register softirq handler to send messages */
    open_softirq(NET_TX_SOFTIRQ, net_tx_action);

    /* register softirq handler to receive messages */
    open_softirq(NET_RX_SOFTIRQ, net_rx_action);
    /* ... */
}

static void net_tx_action(struct softirq_action *h)
{
    /* ... */
}
```

Using softirq: softirq handler

- Run with interrupts enabled and cannot sleep
- The key advantage of softirq over tasklet is scalability
 - **If the same softirq is raised again while it is executing, another processor can run it simultaneously**
- **This means that any shared data needs proper locking**
 - To avoid locking, most softirq handlers resort to per-processor data (data unique to each processor and thus not requiring locking)

Using softirq: raising a softirq

- Softirqs are most often raised from within interrupt handlers (i.e., top halves)
 - The interrupt handler performs the basic hardware-related work, raises the softirq, and then exits

```
/* linux/include/linux/interrupt.h */
/* Disable interrupt and raise a softirq */
extern void raise_softirq(unsigned int nr);

/* Raise a softirq. Interrupt must already be off. */
extern void raise_softirq_irqoff(unsigned int nr);

/* linux/net/core/dev.c */
raise_softirq(NET_TX_SOFTIRQ);
raise_softirq_irqoff(NET_TX_SOFTIRQ);
```

Tasklet

- Built on top of softirqs
 - HI_SOFTIRQ : high priority tasklet
 - TASKLET_SOFTIRQ : normal priority tasklet
- Running in an interrupt context (i.e., cannot sleep)
 - Like softirq, all interrupts are enabled
- Restricted concurrency than softirq
 - The same tasklet cannot run concurrently

tasklet_struct

```
/* linux/linux/include/interrupt.h */
struct tasklet_struct
{
    struct tasklet_struct *next; /* next tasklet in the list */
    unsigned long state;        /* state of a tasklet
        * - TASKLET_STATE_SCHED: a tasklet is scheduled for execution
        * - TASKLET_STATE_RUN: a tasklet is running */
    atomic_t count;             /* disable counter
        * != 0: a tasklet is disabled and cannot run
        * == 0: a tasklet is enabled */
    void (*func)(unsigned long); /* tasklet handler function */
    unsigned long data;          /* argument of the tasklet function */
};
```

Scheduling a tasklet

- Scheduled tasklets are stored in two per-processor linked list:
 - `tasklet_vec`, `tasklet_hi_vec`

```
/* linux/kernel/softirq.c*/
struct tasklet_head {
    struct tasklet_struct *head;
    struct tasklet_struct **tail;
};

/* regular tasklet */
static DEFINE_PER_CPU(struct tasklet_head, tasklet_vec);
/* high-priority tasklet */
static DEFINE_PER_CPU(struct tasklet_head, tasklet_hi_vec);
```

Scheduling a tasklet

```
/* linux/include/linux/interrupt.h, linux/kernel/softirq.c */

/* Schedule a regular tasklet
 * For high-priority tasklet, use tasklet_hi_schedule() */
static inline void tasklet_schedule(struct tasklet_struct *t)
{
    if (!test_and_set_bit(TASKLET_STATE_SCHED, &t->state))
        __tasklet_schedule(t);
}

void __tasklet_schedule(struct tasklet_struct *t)
{
    unsigned long flags;
    local_irq_save(flags);      /* disable interrupt */
    /* Append this tasklet at the end of list */
    t->next = NULL;
    *_this_cpu_read(tasklet_vec.tail) = t;
    _this_cpu_write(tasklet_vec.tail, &(t->next));
    /* Raise a softirq */
    raise_softirq_irqoff(TASKLET_SOFTIRQ); /* tasklet is a softirq */
    local_irq_restore(flags); /* enable interrupt */
}
```

Tasklet softirq handlers

```
/* linux/kernel/softirq.c*/
void __init softirq_init(void)
{
    /* ... */
    /* Tasklet softirq handlers are registered at initializing softirq */
    open_softirq(TASKLET_SOFTIRQ, tasklet_action);
    open_softirq(HI_SOFTIRQ, tasklet_hi_action);
}

static __latent_entropy void tasklet_action(struct softirq_action *a)
{
    struct tasklet_struct *list;

    /* Clear the list for this processor by setting it equal to NULL */
    local_irq_disable();
    list = __this_cpu_read(tasklet_vec.head);
    __this_cpu_write(tasklet_vec.head, NULL);
    __this_cpu_write(tasklet_vec.tail, this_cpu_ptr(&tasklet_vec.head));
    local_irq_enable();
```

Tasklet softirq handlers (cont'd)

```
/* For all tasklets in the list */
while (list) {
    struct tasklet_struct *t = list;
    list = list->next;
    /* If a tasklet is not processing and it is enabled */
    if (tasklet_trylock(t) && !atomic_read(&t->count)) {
        /* and it is not running */
        if (!test_and_clear_bit(TASKLET_STATE_SCHED, &t->state))
            BUG();
        /* then execute the associate tasklet handler */
        t->func(t->data);
        tasklet_unlock(t);
        continue;
    }
    tasklet_unlock(t);
}
local_irq_disable();
t->next = NULL;
*_this_cpu_write(tasklet_vec.tail) = t;
__this_cpu_write(tasklet_vec.tail, &(t->next));
__raise_softirq_irqoff(TASKLET_SOFTIRQ);
local_irq_enable();
}
```

Using tasklet: declaring a tasklet

```
/* linux/include/linux/interrupt.h */

/* Static declaration of a tasklet with initially enabled */
#define DECLARE_TASKLET(tasklet_name, handler_func, handler_arg) \
struct tasklet_struct tasklet_name = { NULL, 0, \
                                      ATOMIC_INIT(0) /* disable counter */, \
                                      handler_func, handler_arg }

/* Static declaration of a tasklet with initially disabled */
#define DECLARE_TASKLET_DISABLED(tasklet_name, handler_func, handler_arg) \
struct tasklet_struct tasklet_name = { NULL, 0, \
                                      ATOMIC_INIT(1) /* disable counter */, \
                                      handler_func, handler_arg }

/* Dynamic initialization of a tasklet */
extern void tasklet_init(struct tasklet_struct *tasklet_name,
                        void (*handler_func)(unsigned long), unsigned long handler_arg);
```

Using tasklet: tasklet handler

- Run with interrupts enabled and cannot sleep
 - If your tasklet shared data with an interrupt handler, you must take precautions (e.g., disable interrupt or obtain a lock)
- Two of the same tasklets never run concurrently
 - Because `tasklet_action()` checks `TASKLET_STATE_RUN`
- But two different tasklets can run at the same time on two different processors

Using tasklet: scheduling a tasklet

```
/* linux/include/linux/interrupt.h */

void tasklet_schedule(struct tasklet_struct *t);
void tasklet_hi_schedule(struct tasklet_struct *t);

/* Disable a tasklet by increasing the disable counter */
void tasklet_disable(struct tasklet_struct *t)
{
    tasklet_disable_nosync(t);
    tasklet_unlock_wait(t); /* and wait until the tasklet finishes */
    smp_mb();
}

void tasklet_disable_nosync(struct tasklet_struct *t)
{
    atomic_inc(&t->count);
    smp_mb__after_atomic();
}

/* Enable a tasklet by descreasing the disable counter */
void tasklet_enable(struct tasklet_struct *t)
{
    smp_mb__before_atomic();
```

Processing overwhelming softirqs

- System can be flooded by softirqs (and tasklets)
 - Softirq might be raised at high rates (e.g., heavy network traffic)
 - While running, a softirq can raise itself so that it runs again
- How to handle such overwhelming softirqs
 - **Solution 1:** Keep processing softirqs as they come in
 - User-space application can starve
 - **Solution 2:** Process one softirq at a time
 - Should wait until the next interrupt occurrence
 - Sub-optimal on an idle system

ksoftirqd

- Per-processor kernel thread to aid processing of softirq
- If the number of softirqs grows excessive, the kernel wakes up

ksoftirqd with normal priority (nice 0)

- No starvation of user-space application
- Running a softirq has the normal priority (nice 0)

```
22:33 $ ps ax -eo pid,nice,stat,cmd | grep ksoftirqd
      7  0 S    [ksoftirqd/0]
     18  0 S    [ksoftirqd/1]
     26  0 S    [ksoftirqd/2]
     34  0 S    [ksoftirqd/3]
```

Work queues

- Work queues defer work into a kernel thread
 - Always runs in process context
 - Thus, work queues are schedulable and can therefore sleep
- By default, per-cpu kernel thread is created, `kworker/n`
 - You can create additional per-CPU worker thread, if needed
- Workqueues users can also create their own threads for better performance and lighten the load on default threads

Work queue implementation: data structure

```
/* linux/kernel/workqueue.c */
struct worker_pool {
    spinlock_t      lock;          /* the pool lock */
    int            cpu;           /* I: the associated cpu */
    int            node;          /* I: the associated node ID */
    int            id;            /* I: pool ID */
    unsigned int    flags;         /* X: flags */

    struct list_head worklist;   /* L: list of pending works */
    int             nr_workers;  /* L: total number of workers */
    /* ... */
};

/* linux/include/workqueue.h */
struct work_struct {
    atomic_long_t data;
    struct list_head entry;
    work_func_t func;
};

typedef void (*work_func_t)(struct work_struct *work);
```

Work queue implementation: work thread

- Worker threads execute the `worker_thread()` function
- Infinite loop doing the following:
 1. Check if there is some work to do in the current pool
 2. If so, execute all the `work_struct` objects pending in the pool `worklist` by calling `process_scheduled_works()`
 - Call the `work_struct` function pointer `func`
 - `work_struct` objects removed
 3. Go to sleep until a new work is inserted in the work queue

Using work queues: creating work

```
/* linux/include/workqueue.h */

/* Statically creating a work */
DECLARE_WORK(work, handler_func);

/* Dynamically creating a work at runtime */
INIT_WORK(work_ptr, handler_func);

/* Work handler prototype
 * - Runs in process context with interrupts are enabled
 * - How to pass a handler-specific parameter
 * : embed work_struct and use container_of() macro */
typedef void (*work_func_t)(struct work_struct *work);

/* Create/destory a new work queue in addition to the default queue
 * - One worker thread per process */
struct workqueue_struct *create_workqueue(char *name);
void destroy_workqueue(struct workqueue_struct *wq);
```

Using work queues: scheduling work

```
/* Put work task in global workqueue (kworker/n) */
bool schedule_work(struct work_struct *work);
bool schedule_work_on(int cpu,
                      struct work_struct *work); /* on the specified CPU */

/* Queue work on a specified workqueue */
bool queue_work(struct workqueue_struct *wq, struct work_struct *work);
bool queue_work_on(int cpu, struct workqueue_struct *wq,
                   struct work_struct *work); /* on the specified CPU */
```

Using work queues: finishing work

```
/* Flush a specific work_struct */
int flush_work(struct work_struct *work);
/* Flush a specific workqueue: */
void flush_workqueue(struct workqueue_struct *);
/* Flush the default workqueue (kworkers): */
void flush_scheduled_work(void);

/* Cancel the work */
void flush_workqueue(struct workqueue_struct *wq);
/* Check if a work is pending */
work_pending(struct work_struct *work);
```

Work queue example

```
#include <linux/module.h>
#include <linux/kernel.h>
#include <linux/init.h>
#include <linux/slab.h>
#include <linux/workqueue.h>

struct work_item {
    struct work_struct ws;
    int parameter;
};

struct work_item *wi, *wi2;
struct workqueue_struct *my_wq;

static void handler(struct work_struct *work)
{
    int parameter = ((struct work_item *)container_of(
                      work, struct work_item, ws))->parameter;
    printk("doing some work ...\\n");
    printk("parameter is: %d\\n", parameter);
}
```

Work queue example

```
static int __init my_mod_init(void)
{
    printk("Entering module.\n");

    my_wq = create_workqueue("lkp_wq");
    wi = kmalloc(sizeof(struct work_item), GFP_KERNEL);
    wi2 = kmalloc(sizeof(struct work_item), GFP_KERNEL);

    INIT_WORK(&wi->ws, handler);
    wi->parameter = 42;
    INIT_WORK(&wi2->ws, handler);
    wi2->parameter = -42;

    schedule_work(&wi->ws);
    queue_work(my_wq, &wi2->ws);

    return 0;
}
```

Work queue example

```
static void __exit my_mod_exit(void)
{
    flush_scheduled_work();
    flush_workqueue(my_wq);
    kfree(wi);
    kfree(wi2);
    destroy_workqueue(my_wq);
    printk(KERN_INFO "Exiting module.\n");
}

module_init(my_mod_init);
module_exit(my_mod_exit);
MODULE_LICENSE("GPL");
```

Choosing the right bottom-half

Bottom half	Context	Inherent serialization
Softirq	Interrupt	None
Tasklet	Interrupt	Against the same tasklet
Work queue	Process	None

- All of these generally run with interrupts enabled
- If there is a shared data with an interrupt handler (top-half), need to disable interrupts or use locks

Disabling softirq and tasklet processing

```
/* Disable softirq and tasklet processing on the local processor */  
void local_bh_disable();  
  
/* Enable softirq and tasklet processing on the local processor */  
void local_bh_enable();
```

- The calls can be nested
 - Only the final call to `local_bh_enable()` actually enables bottom halves
- These calls do not disable workqueues processing

Further readings

- [0xA: Interrupts and Interrupt Handling](#)
- [Modernizing the tasklet API](#)
- [Moving interrupts to threads](#)

Next lecture

- Kernel synchronization

Kernel Synchronization I

Dongyoon Lee

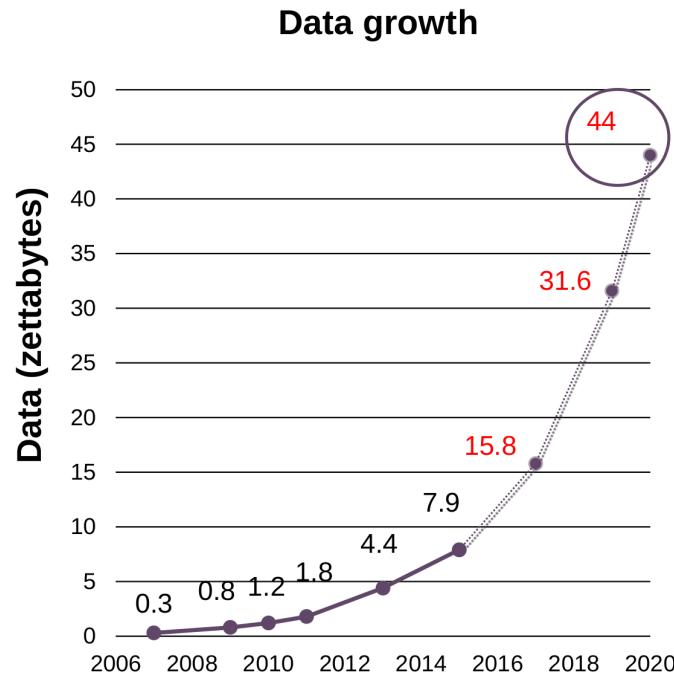
Summary of last lectures

- Tools: building, exploring, and debugging Linux kernel
- Core kernel infrastructure
 - syscall, module, kernel data structures
- Process management & scheduling
- Interrupt & interrupt handler

Today: kernel synchronization

- Kernel synchronization I
 - Background on multicore processing
 - Introduction to synchronization
- Kernel synchronization II
 - Synchronization mechanisms in Linux Kernel
- Kernel synchronization III
 - Read-Copy-Update (RCU)

Data growth is already exponential

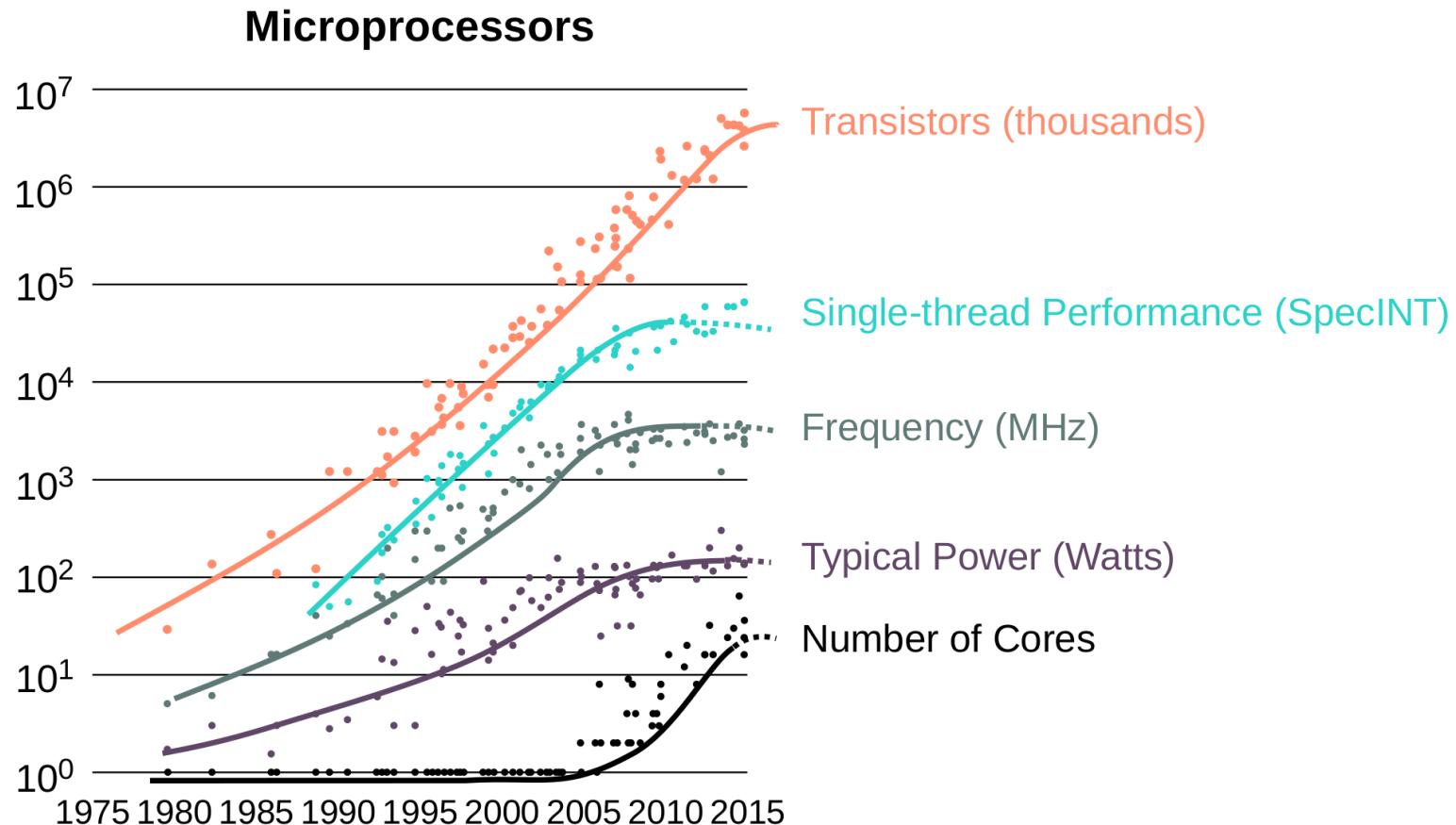


- 1 zettabyte = 10^9 terabytes

Data growth is already exponential

- Data nearly doubles every two years (2013-20)
- By 2020
 - 8 billion people
 - 20 billion connected devices
 - 100 billion infrastructure devices
- Need more processing power

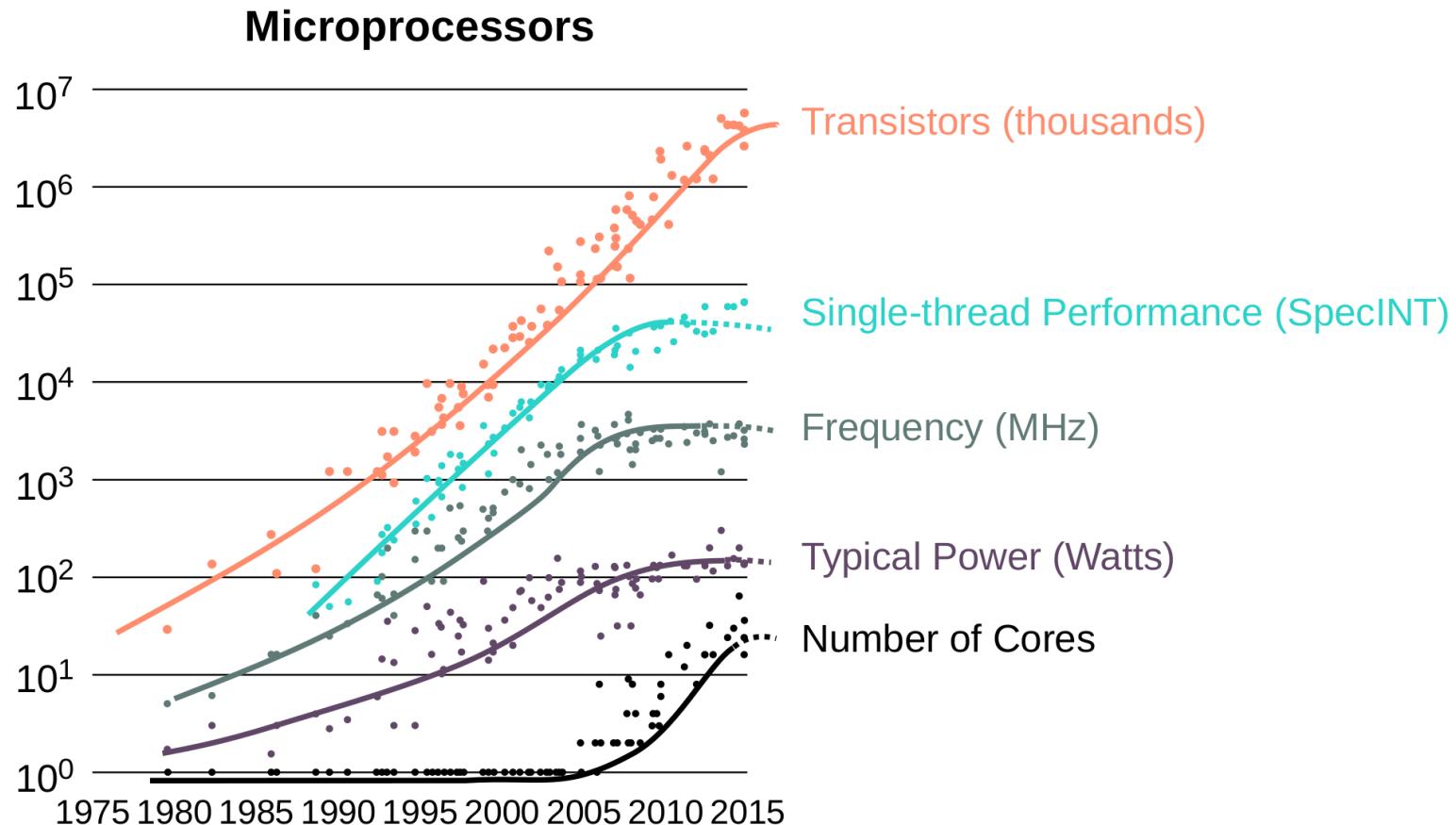
Single-core performance scaling stopped



Single-core performance scaling stopped

- **Increasing clock frequency is not possible anymore**
 - Power consumption: higher frequency → higher power consumption
 - Wire delay: range of a wire in one clock cycle
- **Limitation in Instruction Level Parallelism (ILP)**
 - 1980s: more transistors → superscalar → pipeline
 - 10 CPI (cycles per instruction) → 1 CPI
 - 1990s: multi-way issue, out-of-order issue, branch prediction
 - 1 CPI → 0.5 CPI

The new normal: multi-core processors



The new normal: multi-core processors

- **Moore's law:** the observation that the number of transistors in a dense integrated circuit doubles approximately every two years
- **Q: Where to use such a doubled transistors in processor design?**
- **~ 2007: make a single-core processor faster**
 - deeper processor pipeline, branch prediction, out-of-order execution, etc.
- **2007 ~: increase the number of cores in a chip**
 - multi-core processor

The new normal: multi-core processors

Example: Intel Xeon 8180M processor

Essentials

Product Collection	Intel® Xeon® Scalable Processors
Code Name	Products formerly Skylake
Vertical Segment	Server
Processor Number	8180M
Status	Launched
Launch Date ?	Q3'17
Lithography ?	14 nm
Recommended Customer Price ?	\$13011.00

Performance

# of Cores ?	28
# of Threads ?	56
Processor Base Frequency ?	2.50 GHz
Max Turbo Frequency ?	3.80 GHz
Cache ?	38.5 MB L3
# of UPI Links ?	3
TDP ?	205 W

- Support up to 8 sockets: $28 \times 8 = 224$ cores (or 448 H/W threads)

Example: AMD Ryzen Threadripper 2



AMD

**AMD 2nd Gen RYZEN Threadripper 2990WX
32-Core, 64-Thread, 4.2 GHz Max Boost (3.0
GHz Base), Socket sTR4 250W
YD299XAZAFWOF Desktop Processor**

(9) Write a Review See 15 Questions | 28 Answers

SHARE

In stock. Limit 5 per customer. Ships from United States.

Sold and Shipped by Newegg

Options: Ryzen Threadripper 2990WX

Ryzen Threadripper 2990WX

Ryzen Threadripper 2950X



- 2nd Gen Threadripper
- AMD SenseMI Technology
- AMD Ryzen Master Utility
- 32 Cores / 64 Threads
- 4.2 GHz Max Boost Clocks, 3.0 GHz Base
- Socket sTR4
- DDR4 Support
- 12nm
- Unlocked for Overclocking
- Cooling device not included - Processor Only

★ FREE SHIPPING

Sold and Shipped by: **Newegg**

\$1,749.99

P REMIER - FREE 3 DAY or faster shipping

ASROCK: BEYOND EXPECTATION

+

<input checked="" type="radio"/> ASRock X399 Taichi sTR4 AMD X399 SATA 6Gb/s USB 3.1 ATX AMD Motherboard	\$319.99
<input type="radio"/> ASRock Fatal1ty X399 Professional Gaming sTR4 AMD X399 SATA 6Gb/s USB 3.1 ATX	\$389.99

1 **ADD TO CART ▶**

ADD TO COMPARE PRICE ALERT

ADD TO WISH LIST

Found on 47 wish lists

SHOPRUNNER get it by Mon Oct. 8 with free 2-day shipping [SIGN UP FREE ▶](#) | [sign in](#)

- Support up to 2 sockets: $32 \times 2 = 64$ cores (or 128 H/W threads)

Example: Cavium's ThunderX2 (ARM server)

Assessing Cavium's ThunderX2: The Arm Server Dream Realized At Last

by [Johan De Gelas](#) on May 23, 2018 9:00 AM EST

Posted in [CPUs](#) [Arm](#) [Enterprise](#) [SoC](#) [Enterprise CPUs](#) [ARMv8](#) [Cavium](#) [ThunderX](#) [ThunderX2](#)

SIZING THINGS UP: SPECIFICATIONS COMPARED

Sizing Things Up: Specifications Compared

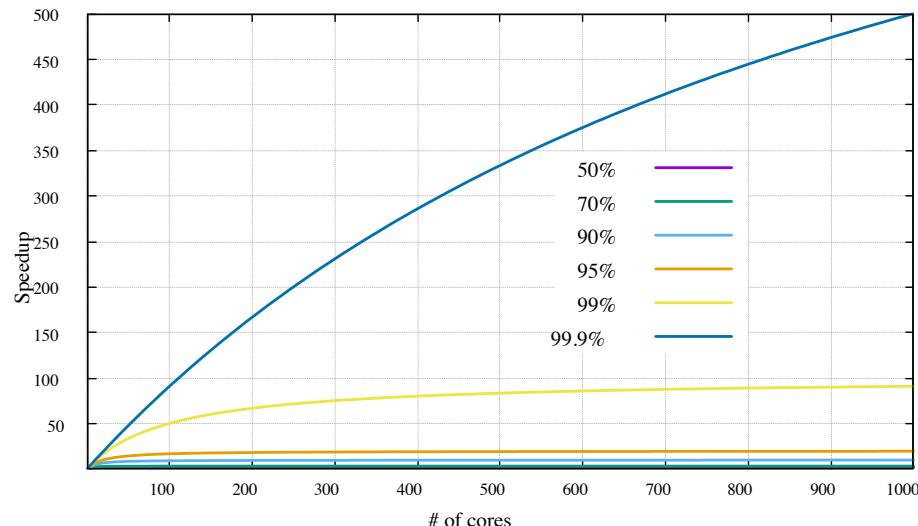
Thirty-two high-IPC cores in one package sounds promising. But how does the best ThunderX2 compare to what AMD, Qualcomm and Intel have to offer? In the table below we compare the high level specifications of several top server SKUs.

Comparison of Major Server SKUs					
AnandTech.com	Cavium ThunderX2 9980-2200	Qualcomm Centriq 2460	Intel Xeon B176	Intel Xeon 6148	AMD EPYC 7601
Process Technology	TSMC 16 nm	Samsung 10 nm	Intel 14 nm	Intel 14 nm	Global Foundries 14 nm
Cores	32 Ring bus	48 Ring bus	28 Mesh	20 Mesh	4 dies x 8 cores MCM
Threads	128	48	56	40	64
Max. number of sockets	2	1	8	4	2
Base Frequency	2.2 GHz	2.2 GHz	2.2 GHz	2.4 GHz	2.2 GHz
Turbo Frequency	2.5 GHz	2.6 GHz	3.8 GHz	3.7 GHz	3.2 GHz
L3 Cache	32 MB	60 MB	38.5 MB	27.5 MB	8x8 MB
DRAM	8-Channel DDR4-2667	6-Channel DDR4-2667	6-Channel DDR4-2667	6-Channel DDR4-2667	8-Channel DDR4-2667
PCIe 3.0 lanes	56	32	48	48	128
TDP	180W	120 W	165W	150W	180W
Price	\$1795	\$1995	\$8719	\$3072	\$4200

- Support up to 2 sockets: $32 \times 2 = 64$ cores (or 256 H/W threads)

Small sequential part does matter

- **Amdhal's Law: theoretical speedup of the execution of a task**
 - Speedup = $1 / (1 - p + p/n)$
 - p : parallel portion of a task, n : the number of CPU core



Where are such sequential parts?

- Applications: sequential algorithm
- Libraries: memory allocator (buddy structure)
- Operating system kernel
 - Memory management: VMA (virtual memory area)
 - File system: file descriptor table, journaling
 - Network stack: receive queue
 - **Your application may not scale even if its design and implementation is scalable**

Introduction to kernel synchronization

- The kernel is programmed using the shared memory model
- **Critical section (also called critical region)**
 - Code paths that access and manipulate shared data
 - Must execute **atomically** without interruption
 - Should not be executed in parallel on SMP → *sequential part*
- **Race condition**
 - Two threads concurrently executing the same critical region → *Bug!*

The case of concurrent data access in kernel

- Real concurrent access on multiple CPUs
 - Same as user-space thread programming
- Preemption on a single core
 - Same as user-space thread programming
- **Interrupt**
 - Only in kernel-space programming
 - *Is a data structure accessed in an interrupt context, top-half or bottom-half?*

Why do we need protection?

- Withdrawing money from an ATM

```
01: int total = get_total_from_account();      /* total funds in account */
02: int withdrawal = get_withdrawal_amount(); /* amount asked to withdrawal */
03:
04: /* check whether the user has enough funds in her account */
05: if (total < withdrawal) {
06:     error("You do not have that much money!")
07:     return -1;
08: }
09:
10: /* OK, the user has enough money:
11: * deduct the withdrawal amount from her total */
12: total -= withdrawal;
13: update_total_funds(total);
14:
15: /* give the user their money */
16: spit_out_money(withdrawal);
```

Concurrent withdrawal from an ATM

- *What happen if two transactions are happening nearly at the same time?*
 - Shared credit card account with your spouse
- Suppose that
 - total == 105
 - withdrawal1 == 100
 - withdrawal2 == 50
- Either of one transaction should fail because $(100 + 50) > 105$

One possible incorrect scenario

1. Two threads check that $100 < 105$ and $50 < 105$ (Line 5)
 2. Thread 1 updates (Line 13)
 - $\text{total} = 105 - 100 = 5$
 3. Thread 2 updates (Line 13)
 - $\text{total} = 105 - 50 = 55$
-
- **Total withdrawal = 150 but there is 55 left on the account!**
 - **Must lock the account during certain operations, make each transaction atomic**

Updating a single variable

```
int i;

void foo(void)
{
    i++;
}
```

- Q: What happens if two threads concurrently execute `foo()` ?
- Q: What happens if two threads concurrently update `i` ?
- Q: Is incrementing `i` atomic operation?

Updating a single variable

- A single C statement

```
/* C code */  
01: i++;
```

- It can be translated into multiple machine instructions

```
/* Machine instructions */  
01: get the current value of i and copy it into a register  
02: add one to the value stored in the register  
03: write back to memory the new value of i
```

- Now, check what happens if two threads concurrently update `i`

Updating a single variable

- Two threads are running. Initial value of `i` is 7

Thread 1	Thread 2
get <code>i</code> (7)	—
increment <code>i</code> (7 -> 8)	—
write back <code>i</code> (8)	—
—	get <code>i</code> (8)
—	increment <code>i</code> (8 -> 9)
—	write back <code>i</code> (9)

- As expected, 7 incremented twice is 9

Updating a single variable

- Two threads are running. Initial value of `i` is 7

Thread 1	Thread 2
get <code>i</code> (7)	get <code>i</code> (7)
increment <code>i</code> (7 -> 8)	—
—	increment <code>i</code> (7 -> 8)
write back <code>i</code> (8)	—
—	write back <code>i</code> (8)

- If both threads of execution read the initial value of `i` before it is incremented, both threads increment and save the same value.

Solution: using an *atomic instruction*

Thread 1	Thread 2
increment & store i (7 -> 8)	—
—	increment & store i (8 -> 9)

Or conversely

Thread 1	Thread 2
—	increment & store (7 -> 8)
increment & store (8 -> 9)	—

- It would never be possible for the two atomic operations to interleave.
- The processor would physically ensure that it was impossible.

x86 example of an atomic instruction

- XADD DEST SRC
- Operation
 - $\text{TEMP} = \text{SRC} + \text{DEST}$
 - $\text{SRC} = \text{DEST}$
 - $\text{DEST} = \text{TEMP}$
- LOCK XADD DEST SRC
- This instruction can be used with a LOCK prefix to allow the instruction to be executed **atomically**.

Wait! Then what is a **volatile** for?

- Operations on **volatile** variables are **not** atomic
- They shall not be **optimized out** or **reordered** by compiler optimization

```
/* C code */
int i;
void foo(void)
{
    /* ... */
    i++;
    /* ... */
}

/* Compiler-generated machine instructions */
/* Non-volatile variables can be optimized out without
 * actually accessing its memory location */
(01: get the current value of i and copy it into a register) <- optimized out
    02: add one to the value stored in the register
(03: write back to memory the new value of i)           <- optimized out
```

Wait! Then what is a **volatile** for?

- They shall not be **optimized out** or **reordered** by compiler optimization

```
/* C code */
int j, i;
void foo(void)
{
    i++;
    j++;
}

/* Compiler-generated machine instructions */
/* Non-volatile variables can be reordered
 * by compiler optimization */
(01/j: get the current value of j and copy it into a register)
(01/i: get the current value of i and copy it into a register)
  02/j: add one to the value stored in the register for j
  02/i: add one to the value stored in the register for i
(03/j: write back to memory the new value of j)
(03/i: write back to memory the new value of i)
```

Wait! Then what is a **volatile** for?

- Operations on **volatile** variables are guaranteed not optimized out or reordered → **disabling compiler optimization**

```
/* C code */
volatile int j, i;
void foo(void)
{
    i++;
    j++;
}

/* Compiler-generated machine instructions */
/* Volatile variables can be optimized out or reordered
 * by compiler optimization */
01/i: get the current value of i and copy it into a register
02/i: add one to the value stored in the register for i
03/i: write back to memory the new value of i
01/j: get the current value of j and copy it into a register
02/j: add one to the value stored in the register for j
03/j: write back to memory the new value of j
```

When we should use **volatile**?

- Memory location can be modified by other entity
 - Other threads for a memory location
 - Other processes for a shared memory location
 - IO devices for an IO address

Locking

- Atomic operations are not sufficient for protecting shared data in long and complex critical regions
 - E.g., `page_tree` of an `inode` (page cache)
- What is needed is a way of making sure that only one thread manipulates the data structure at a time
 - A mechanism for preventing access to a resource while another thread of execution is in the marked region. → **lock**

Linux radix tree example

```
/* linux/include/linux/fs.h */

struct inode {          /** metadata of a file */
    umode_t           i_mode;        /* permission: rwxrw-r-- */
    struct super_block *i_sb;       /* a file system instance */
    struct address_space *i_mapping; /* page cache */
};

struct address_space { /** page cache of an inode */
    struct inode      *host;        /* owner: inode, block_device */
    struct radix_tree_root page_tree; /* radix tree of all pages
                                         * - i.e., page cache of an inode
                                         * - key:   file offset
                                         * - value: cached page */
    spinlock_t         tree_lock;   /* lock protecting it */
};
```

Linux radix tree example

Thread 1	Thread 2
=====	=====
Try to lock the tree_lock	Try to lock the tree_lock
Succeeded: acquired the tree_lock	Failed: waiting...
Access page_tree	Waiting...
...	Waiting...
Unlock the tree_lock	Succeeded: acquired the tree_lock
...	Access page_tree...
	...
	Unlock the tree_lock

- Locks are entirely a programming construct that the programmer must take advantage of → No protection generally ends up in data corruption
- Linux provides various locking mechanisms
 - Non-blocking (or spinning) locks, blocking (or sleeping) locks

Causes of concurrency

- **Symmetrical multiprocessing (true concurrency)**
 - Two or more processors can execute kernel code at exactly the same time.
- **Kernel preemption (pseudo-concurrency)**
 - Because the kernel is preemptive, one task in the kernel can preempt another.
 - Two things do not actually happen at the same time but interleave with each other such that they might as well.

Causes of concurrency

- **Sleeping and synchronization with user-space**
 - A task in the kernel can sleep and thus invoke the scheduler, resulting in the running of a new process.
- **Interrupts**
 - An interrupt can occur asynchronously at almost any time, interrupting the currently executing code.
- **Softirqs and tasklets**
 - The kernel can raise or schedule a softirq or tasklet at almost any time, interrupting the currently executing code.

Concurrency safety

- **SMP-safe**
 - Code that is safe from concurrency on symmetrical multiprocessing machines
- **Preemption-safe**
 - Code that is safe from concurrency with kernel preemption
- **Interrupt-safe**
 - Code that is safe from concurrent access from an interrupt handler

What to protect?

- **Protect data not code**
 - `page_tree` is protected by `tree_lock`

```
/* linux/include/linux/fs.h */

struct inode {          /** metadata of a file */
    umode_t           i_mode;        /* permission: rwxrw-r-- */
    struct super_block *i_sb;       /* a file system instance */
    struct address_space *i_mapping; /* page cache */
};

struct address_space { /** page cache of an inode */
    struct inode      *host;        /* owner: inode, block_device */
    struct radix_tree_root page_tree; /* radix tree of all pages
                                         * - i.e., page cache of an inode
                                         * - key:   file offset
                                         * - value: cached page */
    spinlock_t         tree_lock;   /* lock protecting it */
};
```

Questionnaire for locking

- Is the data global?
- Can a thread of execution other than the current one access it?
- Is the data shared between process context and interrupt context?
- Is it shared between two different interrupt handlers?
- If a process is preempted while accessing this data, can the newly scheduled process access the same data?
- If the current process sleep on anything, in what state does that leave any shared data?
- What happens if this function is called again on another processor?

Deadlocks

- Situations in which one or several threads are waiting on locks for one or several resources that will never be freed
 - None of the threads can continue
- **Self-deadlock**
 - **NOTE: Linux does not support recursive locks**

```
acquire lock
acquire lock, again
wait for lock to become available
...
...
```

Deadlocks

- **Deadly embrace (ABBA deadlock)**

Thread 1	Thread 2
acquire lock A	acquire lock B
try to acquire lock B	try to acquire lock A
wait for lock B	wait for lock A

Deadlock prevention: lock ordering

- Nested locks must *always* be obtained in the *same order*.
 - This prevents the deadly embrace deadlock.

```
/* linux/mm/filemap.c */
/*
 * Lock ordering:
 *
 * ->i_mmap_rwsem      (truncate_pagecache)
 *   ->private_lock     (_free_pte->_set_page_dirty_buffers)
 *     ->swap_lock      (exclusive_swap_page, others)
 *       ->mapping->tree_lock
 *
 * ->i_mutex
 *   ->i_mmap_rwsem    (truncate->unmap_mapping_range)
 * ...
 */
```

Deadlock prevention: lock ordering

```
/* linux/fs/namei.c */
struct dentry *lock_rename(struct dentry *p1, struct dentry *p2)
{
    struct dentry *p;
    if (p1 == p2) {
        inode_lock_nested(p1->d_inode, I_MUTEX_PARENT);
        return NULL;
    }
    mutex_lock(&p1->d_sb->s_vfs_rename_mutex);
    p = d_ancestor(p2, p1);
    if (p) {
        inode_lock_nested(p2->d_inode, I_MUTEX_PARENT);
        inode_lock_nested(p1->d_inode, I_MUTEX_CHILD);
        return p;
    }
    p = d_ancestor(p1, p2);
    if (p) {
        inode_lock_nested(p1->d_inode, I_MUTEX_PARENT);
        inode_lock_nested(p2->d_inode, I_MUTEX_CHILD);
        return p;
    }
    inode_lock_nested(p1->d_inode, I_MUTEX_PARENT);
    inode_lock_nested(p2->d_inode, I_MUTEX_PARENT2);
```

Contention and scalability

- **Lock contention:** a lock currently in use but that another thread is trying to acquire
- **Scalability:** how well a system can be expanded with a large number of processors
- **Coarse- vs fine-grained locking**
 - Coarse-grained lock: bottleneck on high-core count machines
 - Fine-grained lock: overhead on low-core count machines
- **Start simple and grow in complexity only as needed. Simplicity is key.**

Further readings

- [Memory-Driven Computing](#)
- [Wikipedia: Moore's Law](#)
- [Wikipedia: Amdahl's Law](#)
- [Intel 64 and IA-32 Architectures Software Developer's Manual](#)
- [Intel Xeon Platinum 8180M Processor](#)

Kernel Synchronization II

Dongyoon Lee

Summary of last lectures

- Tools: building, exploring, and debugging Linux kernel
- Core kernel infrastructure
 - syscall, module, kernel data structures
- Process management & scheduling
- Interrupt & interrupt handler
- Kernel synchronization: concepts

Today: kernel synchronization II

- Atomic operations
- Spinlock, reader-writer spinlock (RWLock)
- Semaphore, mutex
- Sequential lock (seqlock)
- Completion variable

Atomic operations

- Provide instructions that execute *atomically without interruption*
- Non-atomic update: `i++`

Thread 1	Thread 2
get i (7)	get i (7)
increment i (7 -> 8)	—
—	increment i (7 -> 8)
write back i (8)	—
—	write back i (8)

Atomic operations

- Atomic update: `atomic_inc(&i)`

Thread 1	Thread 2
increment & store i (7 -> 8)	—
—	increment & store i (8 -> 9)

Or conversely

Thread 1	Thread 2
—	increment & store (7 -> 8)
increment & store (8 -> 9)	—

Atomic operations

- Examples
 - `fetch-and-add` : atomic increment
 - `test-and-set` : set a value at a memory location and return the previous value
 - `compare-and-swap` : modify the content of a memory location only if the previous content is equal to a given value
- Linux provides two APIs:
 - Integer atomic operations
 - Bitwise atomic operations

Atomic integer operations

```
/* Type definition: linux/include/linux/types.h */
typedef struct {
    int counter;
} atomic_t;

typedef struct {
    long counter;
} atomic64_t;

/* API definition: linux/include/linux/atomic.h */
/* Usage example */
atomic_t v;                      /* define v */
atomic_t u = ATOMIC_INIT(0); /* define and initialize u to 0 */

atomic_set(&v, 4);           /* v = 4 (atomically) */
atomic_add(2, &v);          /* v = v + 2 == 6 (atomically) */
atomic_inc(&v);            /* v = v + 1 == 7 (atomically) */
```

Atomic integer operations (32-bit)

Atomic integer operations (32-bit)

int atomic_add_negative(int i, atomic_t *v)	Atomically add i to v and return true if the result is negative; otherwise false.
int atomic_add_return(int i, atomic_t *v)	Atomically add i to v and return the result.
int atomic_sub_return(int i, atomic_t *v)	Atomically subtract i from v and return the result.
int atomic_inc_return(int i, atomic_t *v)	Atomically increment v by one and return the result.
int atomic_dec_return(int i, atomic_t *v)	Atomically decrement v by one and return the result.
int atomic_dec_and_test(atomic_t *v)	Atomically decrement v by one and return true if zero; false otherwise.
int atomic_inc_and_test(atomic_t *v)	Atomically increment v by one and return true if the result is zero; false otherwise.

Atomic integer operations (64-bit)

Atomic Integer Operation	Description
ATOMIC64_INIT(long i)	At declaration, initialize to i.
long atomic64_read atomic64_t *v)	Atomically read the integer value of v.
void atomic64_set(atomic64_t *v, int i)	Atomically set v equal to i.
void atomic64_add(int i, atomic64_t *v)	Atomically add i to v.
void atomic64_sub(int i, atomic64_t *v)	Atomically subtract i from v.
void atomic64_inc(atomic64_t *v)	Atomically add one to v.
void atomic64_dec(atomic64_t *v)	Atomically subtract one from v.
int atomic64_sub_and_test(int i, atomic64_t *v)	Atomically subtract i from v and return true if the result is zero; otherwise false.
int atomic64_add_negative(int i, atomic64_t *v)	Atomically add i to v and return true if the result is negative; otherwise false.

Atomic integer operations (64-bit)

long atomic64_add_return(int i, atomic64_t *v)	Atomically add i to v and return the result.
long atomic64_sub_return(int i, atomic64_t *v)	Atomically subtract i from v and return the result.
long atomic64_inc_return(int i, atomic64_t *v)	Atomically increment v by one and return the result.
long atomic64_dec_return(int i, atomic64_t *v)	Atomically decrement v by one and return the result.
int atomic64_dec_and_test(atomic64_t *v)	Atomically decrement v by one and return true if zero; false otherwise.
int atomic64_inc_and_test(atomic64_t *v)	Atomically increment v by one and return true if the result is zero; false otherwise.

Atomic integer operations: usage example

```
#include <linux/module.h>
#include <linux/kernel.h>
#include <linux/init.h>
#include <linux/slab.h>
#include <linux/delay.h>
#include <linux/kthread.h>
#include <linux/sched.h>
#include <linux/types.h>

#define PRINT_PREF "[SYNC_ATOMIC] "
atomic_t counter; /* shared data: */
struct task_struct *read_thread, *write_thread;

static int writer_function(void *data)
{
    while(!kthread_should_stop()) {
        atomic_inc(&counter);
        msleep(500);
    }
    do_exit(0);
}
```

Atomic integer operations: usage example

```
static int read_function(void *data)
{
    while(!kthread_should_stop()) {
        printk(PRINT_PREF "counter: %d\n", atomic_read(&counter));
        msleep(500);
    }
    do_exit(0);
}

static int __init my_mod_init(void)
{
    printk(PRINT_PREF "Entering module.\n");

    atomic_set(&counter, 0);

    read_thread = kthread_run(read_function, NULL, "read-thread");
    write_thread = kthread_run(writer_function, NULL, "write-thread");

    return 0;
}
```

Atomic integer operations: usage example

```
static void __exit my_mod_exit(void)
{
    kthread_stop(read_thread);
    kthread_stop(write_thread);
    printk(KERN_INFO "Exiting module.\n");
}

module_init(my_mod_init);
module_exit(my_mod_exit);

MODULE_LICENSE("GPL");
```

Atomic bitwise operations

```
/* API definition: include/linux/bitops.h */

/* Usage example */
unsigned long word = 0; /* 32 / 64 bits according to the system */

set_bit(0, &word);      /* bit zero is set atomically */
set_bit(1, &word);      /* bit one is set atomically */
printf("&ul\n", word); /* print "3" */
clear_bit(1, &word);    /* bit one is unset atomically */
change_bit(0, &word);   /* flip bit zero atomically (now unset) */

/* set bit zero and return its previous value (atomically) */
if(test_and_set_bit(0, &word)) {
    /* not true in the case of our example */
}

/* you can mix atomic bit operations and normal C */
word = 7;
```

Atomic bitwise operations

Atomic Bitwise Operation

```
void set_bit(int nr, void *addr)
```

```
void clear_bit(int nr, void *addr)
```

```
void change_bit(int nr, void *addr)
```

```
int test_and_set_bit(int nr, void *addr)
```

```
int test_and_clear_bit(int nr, void *addr)
```

```
int test_and_change_bit(int nr, void *addr)
```

```
int test_bit(int nr, void *addr)
```

Description

Atomically set the *nr-th* bit starting from *addr*.

Atomically clear the *nr-th* bit starting from *addr*.

Atomically flip the value of the *nr-th* bit starting from *addr*.

Atomically set the *nr-th* bit starting from *addr* and return the previous value.

Atomically clear the *nr-th* bit starting from *addr* and return the previous value.

Atomically flip the *nr-th* bit starting from *addr* and return the previous value.

Atomically return the value of the *nr-th* bit starting from *addr*.

Atomic bitwise operations

- Non-atomic bitwise operations are also provided
 - prefixed with double underscore
 - Example: `test_bit()` vs `__test_bit()`
- If you do not require atomicity (say, for example, because a lock already protects your data), these variants of the bitwise functions might be faster.

Spinlocks

- The most common lock used in the kernel
- When a thread tries to acquire an already held lock, it spins while waiting for the lock become available.
 - Wasting processor time when spinning is too long
 - Spinlocks can be used in interrupt context, which a thread cannot sleep → **Kernel provides special spinlock API for data structures shared in interrupt context**
- In process context, do not sleep while holding a spinlock
 - Kernel preemption is disabled

Spinlocks: usage

```
/* linux/include/linux/spinlock.h */  
DEFINE_SPINLOCK(my_lock);  
  
spin_lock(&my_lock);  
/* critical region */  
spin_unlock(&my_lock);
```

- `spin_lock()` is not recursive! → self-deadlock
- Lock/unlock methods disable/enable kernel preemption and acquire/release the lock
- Lock is compiled away on uniprocessor systems
 - Still needs do disabled/re-enable preemption to prevent interleaving of task execution

Quiz #1: find a deadlock

```
01: /* WARNING!!! THIS CODE HAS A DEADLOCK!!! WARNING!!! */
02: DEFINE_HASHTABLE(global_hashtbl, 10);
03: DEFINE_SPINLOCK(hashtbl_lock);
04:
05: irqreturn_t irq_handler(int irq, void *dev_id)
06: {
07:     /* Interrupt handler running in interrupt context */
08:     spin_lock(&hashtbl_lock);
09:     /* access global_hashtbl */
10:     spin_unlock(&hashtbl_lock);
11: }
12:
13: int foo(void)
14: {
15:     /* A function running in process context */
16:     spin_lock(&hashtbl_lock);
17:     /* What happen if an interrupt occurs
18:      * while a task executing here? -> Deadlock */
19:     spin_unlock(&hashtbl_lock);
20: }
```

Quiz #2: find a deadlock

```
01: /* WARNING!!! THIS CODE HAS A DEADLOCK!!! WARNING!!! */
02: DEFINE_HASHTABLE(global_hashtbl, 10);
03: DEFINE_SPINLOCK(hashtbl_lock);
04:
05: irqreturn_t irq_handler_1(int irq, void *dev_id)
06: {
07:     /* Interrupt handler running in interrupt context */
08:     spin_lock(&hashtbl_lock);
09:     /* access global_hashtbl */
10:     spin_unlock(&hashtbl_lock);
11: }
12:
13: irqreturn_t irq_handler_2(int irq, void *dev_id)
14: {
15:     /* Interrupt handler running in interrupt context */
16:     spin_lock(&hashtbl_lock);
17:     /* What happen if an interrupt 1 occurs
18:      * while executing here? -> Deadlock */
19:     spin_unlock(&hashtbl_lock);
20: }
```

Spinlocks: usage in interrupt handlers

- Spin locks do not sleep so it is safe to use them in interrupt context
- **If a lock is used in an interrupt handler, you must also disable local interrupts before obtaining the lock.**
- Otherwise, it is possible for an interrupt handler to interrupt kernel code while the lock is held and attempt to reacquire the lock.
- The interrupt handler spins, waiting for the lock to become available. The lock holder, however, does not run until the interrupt handler completes.
→ **double-acquire deadlock**

Spinlocks: usage in interrupt handlers

- Conditional enabling/disabling local interrupt

```
DEFINE_SPINLOCK(mr_lock);
unsigned long flags;

/* Saves the current state of interrupts, disables them locally, and
then obtains the given lock */
spin_lock_irqsave(&mr_lock, flags);

/* critical region ... */

/* Unlocks the given lock and returns interrupts to their previous state */
spin_unlock_irqrestore(&mr_lock, flags);
```

Spinlocks: usage in interrupt handlers

- Unconditional enabling/disabling local interrupt
 - If you always know before the fact that interrupts are initially enabled, there is no need to restore their previous state.

```
DEFINE_SPINLOCK(mr_lock);

/* Disable local interrupt and acquire lock */
spin_lock_irq(&mr_lock);

/* critical section ... */

/* Unlocks the given lock and enable local interrupt */
spin_unlock_irq(&mr_lock);
```

- Let's check the code

Bug fix for usage #1

```
01: /* NOTE: BUG-FIXED VERSION OF USAGE #1 */
02: DEFINE_HASHTABLE(global_hashtbl, 10);
03: DEFINE_SPINLOCK(hashtbl_lock);
04:
05: irqreturn_t irq_handler(int irq, void *dev_id)
06: {
07:     /* Interrupt handler running in interrupt context */
08:     spin_lock(&hashtbl_lock);
09:     /* It is okay not to disable interrupt here
10:      * because this is the only interrupt handler access
11:      * the shared data and this particular interrupt is
12:      * already disabled. */
13:     spin_unlock(&hashtbl_lock);
14: }
15:
16: int foo(void)
17: {
18:     /* A function running in process context */
19:     unsigned long flags;
20:     spin_lock_irqsave(&hashtbl_lock, flags);
21:     /* Interrupt is disabled here */
22:     spin_unlock_irqrestore(&hashtbl_lock, flags);
23: }
```

Bug fix for usage #2

```
01: /* NOTE: BUG-FIXED VERSION OF USAGE #2 */
02: DEFINE_HASHTABLE(global_hashtbl, 10);
03: DEFINE_SPINLOCK(hashtbl_lock);
04:
05: irqreturn_t irq_handler_1(int irq, void *dev_id)
06: {
07:     /* Interrupt handler running in interrupt context */
08:     spin_lock_irq(&hashtbl_lock);
09:     /* Need to disable interrupt here
10:         * to prevent irq_handler_1 from accessing the shared data */
11:     spin_unlock_irq(&hashtbl_lock);
12:
13: irqreturn_t irq_handler_2(int irq, void *dev_id)
14: {
15:     /* Interrupt handler running in interrupt context */
16:     spin_lock_irq(&hashtbl_lock);
17:     /* Need to disable interrupt here
18:         * to prevent irq_handler_1 from accessing the shared data */
19:     spin_unlock_irq(&hashtbl_lock);
```

Spinlock API

Table 10.4 Spin Lock Methods

Method	Description
<code>spin_lock()</code>	Acquires given lock
<code>spin_lock_irq()</code>	Disables local interrupts and acquires given lock
<code>spin_lock_irqsave()</code>	Saves current state of local interrupts, disables local interrupts, and acquires given lock
<code>spin_unlock()</code>	Releases given lock
<code>spin_unlock_irq()</code>	Releases given lock and enables local interrupts
<code>spin_unlock_irqrestore()</code>	Releases given lock and restores local interrupts to given previous state
<code>spin_lock_init()</code>	Dynamically initializes given <code>spinlock_t</code>
<code>spin_trylock()</code>	Tries to acquire given lock; if unavailable, returns nonzero
<code>spin_is_locked()</code>	Returns nonzero if the given lock is currently acquired, otherwise it returns zero

Spinlocks and bottom halves

- `spin_lock_bh()` / `spin_unlock_bh()`
 - Obtains the given lock and disables all bottom halves
- Because a bottom half might preempt process context code, if data is shared between a bottom-half process context, you must protect the data in process context with both a lock and the disabling of bottom halves.
- Likewise, because an interrupt handler might preempt a bottom half, if data is shared between an interrupt handler and a bottom half, you must both obtain the appropriate lock and disable interrupts.

Quiz #3: interrupt context?

- Top-half: Interrupt handler
- Bottom-half: Softirq, Tasklet
- KProbe handler, timer handler
- Any handler
 - Ask whether it runs in interrupt context
 - If so ask which interrupts are disabled

Spinlock usage example

```
#include <linux/module.h>
#include <linux/kernel.h>
#include <linux/init.h>
#include <linux/slab.h>
#include <linux/delay.h>
#include <linux/spinlock.h>
#include <linux/kthread.h>
#include <linux/sched.h>
#define PRINT_PREF "[SYNC_SPINLOCK] "

unsigned int counter; /* shared data: */
DEFINE_SPINLOCK(counter_lock);
struct task_struct *read_thread, *write_thread;

static int writer_function(void *data)
{
    while(!kthread_should_stop()) {
        spin_lock(&counter_lock);
        counter++;
        spin_unlock(&counter_lock);
        msleep(500);
    }
    do_exit(0);
}
```

Spinlock usage example

```
static int read_function(void *data)
{
    while(!kthread_should_stop()) {
        spin_lock(&counter_lock);
        printk(PRINT_PREF "counter: %d\n", counter);
        spin_unlock(&counter_lock);
        msleep(500);
    }
    do_exit(0);
}

static int __init my_mod_init(void)
{
    printk(PRINT_PREF "Entering module.\n");
    counter = 0;

    read_thread = kthread_run(read_function, NULL, "read-thread");
    write_thread = kthread_run(writer_function, NULL, "write-thread");

    return 0;
}
```

Spinlock usage example

```
static void __exit my_mod_exit(void)
{
    kthread_stop(read_thread);
    kthread_stop(write_thread);
    printk(KERN_INFO "Exiting module.\n");
}

module_init(my_mod_init);
module_exit(my_mod_exit);

MODULE_LICENSE("GPL");
```

Reader-writer spinlock (RWLock)

- Reader-writer spinlock allows multiple concurrent readers
- When entities accessing a shared data can be clearly divided into readers and writers
- Example: list updated (write) and searched (read)
 - When updated, no other entity should update nor search
 - When searched, no other entity should update
 - **Safe to allow multiple readers in parallel**
 - **Can improve scalability by allowing parallel readers**
- Reader-write lock == shared-exclusive lock == concurrent-exclusive lock

Reader-writer spinlock (RWLock)

```
#include <linux/spinlock.h>

/* Define reader-writer spinlock */
DEFINE_RWLOCK(mr_rwlock);

/* Reader */
read_lock(&mr_rwlock);
/* critical section (read only) ... */
read_unlock(&mr_rwlock);

/* Writer */
write_lock(&mr_rwlock);
/* critical section (read and write) ... */
write_unlock(&mr_lock);

/* You cannot upgrade a read lock to a write lock.
 * Following code has a deadlock: */
read_lock(&mr_rwlock);
write_lock(&mr_lock); /* It will wait forever until there is no reader */
```

Reader-writer spinlock (RWLock)

- Linux reader-writer spinlocks favor readers over writers
 - If the read lock is held and a writer is waiting for exclusive access, readers that attempt to acquire the lock continue to succeed.
 - Therefore, a sufficient number of readers can starve pending writers.

Reader-writer spinlock API

Table 10.5 Reader-Writer Spin Lock Methods

Method	Description
<code>read_lock()</code>	Acquires given lock for reading
<code>read_lock_irq()</code>	Disables local interrupts and acquires given lock for reading
<code>read_lock_irqsave()</code>	Saves the current state of local interrupts, disables local interrupts, and acquires the given lock for reading
<code>read_unlock()</code>	Releases given lock for reading
<code>read_unlock_irq()</code>	Releases given lock and enables local interrupts
<code>read_unlock_irqrestore()</code>	Releases given lock and restores local interrupts to the given previous state
<code>write_lock()</code>	Acquires given lock for writing
<code>write_lock_irq()</code>	Disables local interrupts and acquires the given lock for writing
<code>write_lock_irqsave()</code>	Saves current state of local interrupts, disables local interrupts, and acquires the given lock for writing

Reader-writer spinlock API

<code>write_unlock()</code>	Releases given lock
<code>write_unlock_irq()</code>	Releases given lock and enables local interrupts
<code>write_unlock_irqrestore()</code>	Releases given lock and restores local interrupts to given previous state
<code>write_trylock()</code>	Tries to acquire given lock for writing; if unavailable, returns nonzero
<code>rwlock_init()</code>	Initializes given <code>rwlock_t</code>

Reader-writer spinlock: usage example

```
#include <linux/module.h>
#include <linux/kernel.h>
#include <linux/init.h>
#include <linux/slab.h>
#include <linux/delay.h>
#include <linux/spinlock.h>
#include <linux/kthread.h>
#include <linux/sched.h>
#define PRINT_PREF "[SYNC_RWSPINLOCK]: "
/* shared data: */
unsigned int counter;
DEFINE_RWLOCK(counter_lock);
struct task_struct *read_thread1, *read_thread2, *read_thread3, *write_thread;
static int writer_function(void *data)
{
    while(!kthread_should_stop()) {
        write_lock(&counter_lock);
        counter++;
        write_unlock(&counter_lock);

        msleep(500);
    }
    do_exit(0);
}
```

Reader-writer spinlock: usage example

```
static int read_function(void *data)
{
    while(!kthread_should_stop()) {
        read_lock(&counter_lock);
        printk(PRINT_PREF "counter: %d\n", counter);
        read_unlock(&counter_lock);

        msleep(500);
    }
    do_exit(0);
}

static int __init my_mod_init(void)
{
    printk(PRINT_PREF "Entering module.\n");
    counter = 0;

    read_thread1 = kthread_run(read_function, NULL, "read-thread1");
    read_thread2 = kthread_run(read_function, NULL, "read-thread2");
    read_thread3 = kthread_run(read_function, NULL, "read-thread3");
    write_thread = kthread_run(writer_function, NULL, "write-thread");

    return 0;
}
```

Reader-writer spinlock: usage example

```
static void __exit my_mod_exit(void)
{
    kthread_stop(read_thread3);
    kthread_stop(read_thread2);
    kthread_stop(read_thread1);
    kthread_stop(write_thread);
    printk(KERN_INFO "Exiting module.\n");
}

module_init(my_mod_init);
module_exit(my_mod_exit);

MODULE_LICENSE("GPL");
```

Semaphore

- Sleeping locks → not usable in interrupt context
- When a task attempts to acquire a semaphore that is unavailable, the semaphore places the task onto a wait queue and puts the task to sleep.
 - The processor is then free to execute other code.
- When a task releases the semaphore, one of the tasks on the wait queue is awakened so that it can then acquire the semaphore.
- Semaphores are not optimal for locks that are held for short periods because the overhead of sleeping, maintaining the wait queue, and waking back up can easily outweigh the total lock hold time.

Semaphore

- Semaphores allow multiples holders
- `counter` initialized to a given value
 - Decremented each time a thread acquire the semaphore
 - The semaphore becomes unavailable when the counter reaches 0
- In the kernel, most of the semaphores used are **binary semaphores** (or mutex)

Semaphore: usage example

```
struct semaphore *sem1;

sem1 = kmalloc(sizeof(struct semaphore), GFP_KERNEL);
if(!sem1)
    return -1;

/* counter == 1: binary semaphore */
sema_init(&sema, 1);

down(sem1);
/* critical region */
up(sem1);

/* Binary semaphore static declaration */
DECLARE_MUTEX(sem2);

if(down_interruptible(&sem2)) {
    /* signal received, semaphore not acquired */
}

/* critical region */

up(sem2);
```

Semaphore API

Method	Description
<code>sema_init(struct semaphore *, int)</code>	Initializes the dynamically created semaphore to the given count
<code>init_MUTEX(struct semaphore *)</code>	Initializes the dynamically created semaphore with a count of one
<code>init_MUTEX_LOCKED(struct semaphore *)</code>	Initializes the dynamically created semaphore with a count of zero (so it is initially locked)
<code>down_interruptible (struct semaphore *)</code>	Tries to acquire the given semaphore and enter interruptible sleep if it is contended
<code>down(struct semaphore *)</code>	Tries to acquire the given semaphore and enter uninterruptible sleep if it is contended
<code>down_trylock(struct semaphore *)</code>	Tries to acquire the given semaphore and immediately return nonzero if it is contended
<code>up(struct semaphore *)</code>	Releases the given semaphore and wakes a waiting task, if any

Reader-writer semaphores

- Reader-writer flavor of semaphore like reader-writer spinlock

```
#include <linux/rwsem.h>

/* declare reader-writer semaphore */
static DECLARE_RWSEM(mr_rwsem); /* or use init_rwsem(struct rw_semaphore *) */

/* attempt to acquire the semaphore for reading ... */
down_read(&mr_rwsem);
/* critical region (read only) ... */
/* release the semaphore */
up_read(&mr_rwsem);

/* ... */

/* attempt to acquire the semaphore for writing ... */
down_write(&mr_rwsem);
/* critical region (read and write) ... */
/* release the semaphore */
up_write(&mr_sem);
```

Reader-writer semaphores

- `down_read_trylock()`, `down_write_trylock()`
 - try to acquire read/write lock
 - returns 1 if successful, 0 if contention
- `downgrade_write()`
 - atomically converts an acquired write lock to a read lock

Reader-writer semaphore: usage example

```
#include <linux/module.h>
#include <linux/kernel.h>
#include <linux/init.h>
#include <linux/slab.h>
#include <linux/delay.h>
#include <linux/kthread.h>
#include <linux/sched.h>
#include <linux/rwsem.h>
#define PRINT_PREF "[SYNC_RWSEM] "

/* shared data: */
unsigned int counter;
struct rw_semaphore *counter_rwsemaphore;
struct task_struct *read_thread, *read_thread2, *write_thread;
```

Reader-writer semaphore: usage example

```
static int writer_function(void *data)
{
    while(!kthread_should_stop()) {
        down_write(counter_rwsemaphore);
        counter++;
        downgrade_write(counter_rwsemaphore);
        printk(PRINT_PREF "(writer) counter: %d\n", counter);
        up_read(counter_rwsemaphore);
        msleep(500);
    }
    do_exit(0);
}

static int read_function(void *data)
{
    while(!kthread_should_stop()) {
        down_read(counter_rwsemaphore);
        printk(PRINT_PREF "counter: %d\n", counter);
        up_read(counter_rwsemaphore);
        msleep(500);
    }
    do_exit(0);
}
```

Reader-writer semaphore: usage example

```
static int __init my_mod_init(void)
{
    printk(PRINT_PREF "Entering module.\n");
    counter = 0;

    counter_rwsemaphore = kmalloc(sizeof(struct rw_semaphore), GFP_KERNEL);
    if(!counter_rwsemaphore)
        return -1;

    init_rwsem(counter_rwsemaphore);

    read_thread = kthread_run(read_function, NULL, "read-thread");
    read_thread2 = kthread_run(read_function, NULL, "read-thread2");
    write_thread = kthread_run(writer_function, NULL, "write-thread");

    return 0;
}
```

Reader-writer semaphore: usage example

```
static void __exit my_mod_exit(void)
{
    kthread_stop(read_thread);
    kthread_stop(write_thread);
    kthread_stop(read_thread2);

    kfree(counter_rwsemaphore);

    printk(KERN_INFO "Exiting module.\n");
}

module_init(my_mod_init);
module_exit(my_mod_exit);

MODULE_LICENSE("GPL");
```

Mutex

- Mutexes are binary semaphore with stricter use cases:
 - Only one thread can hold the mutex at a time
 - A thread locking a mutex must unlock it
 - No recursive lock and unlock operations
 - A thread cannot exit while holding a mutex
 - A mutex cannot be acquired in interrupt context
 - A mutex can be managed only through the API
- Semaphore vs Mutex?
 - Start with a mutex and move to a semaphore only if you have to

Mutex

```
#include <linux/mutex.h>

DEFINE_MUTEX(mut1); /* static */

struct mutex *mut2 = kmalloc(sizeof(struct mutex), GFP_KERNEL); /* dynamic */
if(!mut2)
    return -1;

mutex_init(mut2);

mutex_lock(&mut1);
/* critical region */
mutex_unlock(&mut1);
```

Mutex API

Method	Description
<code>mutex_lock(struct mutex *)</code>	Locks the given mutex; sleeps if the lock is unavailable
<code>mutex_unlock(struct mutex *)</code>	Unlocks the given mutex
<code>mutex_trylock(struct mutex *)</code>	Tries to acquire the given mutex; returns one if successful and the lock is acquired and zero otherwise
<code>mutex_is_locked (struct mutex *)</code>	Returns one if the lock is locked and zero otherwise

Mutex: usage example

```
#include <linux/module.h>
#include <linux/kernel.h>
#include <linux/init.h>
#include <linux/slab.h>
#include <linux/delay.h>
#include <linux/kthread.h>
#include <linux/sched.h>
#include <linux/mutex.h>

#define PRINT_PREF "[SYNC_MUTEX]: "
/* shared data: */
unsigned int counter;
struct mutex *mut;
struct task_struct *read_thread, *write_thread;
```

Mutex: usage example

```
static int writer_function(void *data)
{
    while(!kthread_should_stop()) {
        mutex_lock(mut);
        kfree(mut);          /* !!! */
        counter++;
        mutex_unlock(mut);
        msleep(500);
    }
    do_exit(0);
}

static int read_function(void *data)
{
    while(!kthread_should_stop()) {
        mutex_lock(mut);
        printk(PRINT_PREF "counter: %d\n", counter);
        mutex_unlock(mut);
        msleep(500);
    }
    do_exit(0);
}
```

Mutex: usage example

```
static int __init my_mod_init(void)
{
    printk(PRINT_PREF "Entering module.\n");
    counter = 0;
    mut = kmalloc(sizeof(struct mutex), GFP_KERNEL);
    if(!mut)
        return -1;
    mutex_init(mut);
    read_thread = kthread_run(read_function, NULL, "read-thread");
    write_thread = kthread_run(writer_function, NULL, "write-thread");
    return 0;
}

static void __exit my_mod_exit(void)
{
    kthread_stop(read_thread);
    kthread_stop(write_thread);
    kfree(mut);
    printk(KERN_INFO "Exiting module.\n");
}

module_init(my_mod_init);
module_exit(my_mod_exit);
MODULE_LICENSE("GPL");
```

Spinlock vs mutex

Requirement	Recommended lock
Low overhead locking	Spin lock is preferred
Short lock hold time	Spin lock is preferred
Long lock hold time	Mutex is preferred
Need to lock from interrupt context	Spin lock is required
Need to sleep while holding lock	Mutex is required

Completion variable

- Completion variables are used when one task needs to signal to the other that an event has occurred.

```
#include <linux/completion.h>

/* Declaration / initialization */
DECLARE_COMPLETION(comp1); /* static */
struct completion *comp2 = kmalloc(sizeof(struct completion), GFP_KERNEL);
if(!comp2)
    return -1;
init_completion(comp2);

/* Thread 1 */
/* signal event: */
complete(comp1);

/* Thread 2 */
/* wait for signal: */
wait_for_completion(comp1);
```

Completion variable API

Method	Description
<code>init_completion(struct completion *)</code>	Initializes the given dynamically created completion variable
<code>wait_for_completion(struct completion *)</code>	Waits for the given completion variable to be signaled
<code>complete(struct completion *)</code>	Signals any waiting tasks to wake up

Completion variable: usage example

```
#include <linux/module.h>
#include <linux/kernel.h>
#include <linux/init.h>
#include <linux/slab.h>
#include <linux/delay.h>
#include <linux/kthread.h>
#include <linux/sched.h>
#include <linux/completion.h>

#define PRINT_PREF "[SYNC_COMP]: "

/* shared data: */
unsigned int counter;
struct completion *comp;

struct task_struct *read_thread, *write_thread;

static int writer_function(void *data)
{
    while(counter != 1234)
        counter++;
    complete(comp);
    do_exit(0);
}
```

Completion variable: usage example

```
static int read_function(void *data)
{
    wait_for_completion(comp);
    printk(PRINT_PREF "counter: %d\n", counter);

    do_exit(0);
}

static int __init my_mod_init(void)
{
    printk(PRINT_PREF "Entering module.\n");
    counter = 0;

    comp = kmalloc(sizeof(struct completion), GFP_KERNEL);
    if(!comp)
        return -1;

    init_completion(comp);

    read_thread = kthread_run(read_function, NULL, "read-thread");
    write_thread = kthread_run(writer_function, NULL, "write-thread");

    return 0;
}
```

Completion variable: usage example

```
static void __exit my_mod_exit(void)
{
    kfree(comp);
    printk(KERN_INFO "Exiting module.\n");
}

module_init(my_mod_init);
module_exit(my_mod_exit);

MODULE_LICENSE("GPL");
```

Sequential lock (seqlock)

- A simple mechanism for reading and writing shared data
- Works by maintaining a sequence counter (or version number)
- Whenever the data in question is written to, a lock is obtained and a sequence number is incremented.
- Prior to and after reading the data, the sequence number is read. If the values are the same, a write did not begin in the middle of the read.
- Further, if the values are even, a write is not underway. (Grabbing the write lock makes the value odd, whereas releasing it makes it even because the lock starts at zero.)

Sequential lock (seqlock)

```
/* define a seq lock */
seqlock_t my_seq_lock = DEFINE_SEQLOCK(my_seq_lock);

/* Write path */
write_seqlock(&my_seq_lock);
/* critical (write) region */
write_sequnlock(&my_seq_lock);

/* Read path */
unsigned long seq;
do {
    seq = read_seqbegin(&my_seq_lock);
    /* read data here ... */
} while(read_seqretry(&my_seq_lock, seq));
```

- Seq locks are useful when
 - There are many readers and few writers
 - Writers should be favored over readers

Preemption disabling

- When a spin lock is held preemption is disabled
- Some situations need to disable preemption without involving spin locks
- Example: manipulating per-processor data:

```
task A manipulates per-processor variable foo, which is not protected by a lock
task A is preempted
task B is scheduled
task B manipulates variable foo
task B completes
task A is rescheduled
task A continues manipulating variable foo
```

Preemption disabling

Function	Description
<code>preempt_disable()</code>	Disables kernel preemption by incrementing the preemption counter
<code>preempt_enable()</code>	Decrement the preemption counter and checks and services any pending reschedules if the count is now zero
<code>preempt_enable_no_resched()</code>	Enables kernel preemption but does not check for any pending reschedules
<code>preempt_count()</code>	Returns the preemption count

Preemption disabling

- For per-processor data

```
int cpu;  
  
/* disable kernel preemption and set "cpu" to the current processor */  
cpu = get_cpu();  
  
/* manipulate per-processor data ... */  
  
/* reenable kernel preemption, "cpu" can change and so is no longer valid */  
put_cpu();
```

Next lecture

- Kernel synchronization III
 - Memory ordering
 - Read-Copy-Update (RCU)

Kernel Synchronization III

Dongyoon Lee

Summary of last lectures

- Tools: building, exploring, and debugging Linux kernel
- Core kernel infrastructure
 - syscall, module, kernel data structures
- Process management & scheduling
- Interrupt & interrupt handler
- Kernel synchronization: concepts and key APIs

Today: kernel synchronization III

- Memory ordering and memory barriers
- Read-copy-update (RCU)

Ordering and barriers

- Memory reads (`load`) and write (`store`) operations can be reordered for performance reasons
 - by the compiler at compile time: compiler optimization
 - by the CPU at run time: TSO (total store ordering), PSO (partial store ordering)

Ordering and barriers

```
/* Following code can be reordered
 * by a compiler (optimization) or a processor (out-of-order execution)
 *
 * Your code          Compiled code
 * =====            ===== */
a = 4;           b = 5;
b = 5;           a = 4;

/* Following code will never be reordered because
 * there is a dependency between a and b.
 *
 * Your code          Compiled code
 * =====            ===== */
a = 1;           a = 1;
b = a;           b = a;
```

Memory barriers

- Instruct a compiler or a processor not to reorder instructions around a given point
- **barrier()** (a.k.a., compiler barrier)
 - Prevents the compiler from reordering stores or loads across the barrier

```
/* Compiler does not reorder store operations of a and b
 * However, a processor may reorder the store operations for performance */
a = 4;
barrier();
b = 5;
```

Memory barrier instructions

- `rmb()` : prevents loads from being reordered across the barrier
- `wmb()` : prevents stores from being reordered across the barrier
- `mb()` : prevents loads and stores from being reordered across the barrier
- `read_barrier_depends()` : prevent data-dependent loads to be reordered across the barrier
 - On some architectures, it is much faster than `rmb()` because it is not needed and is, thus, a `noop`

Memory barrier example

- Initial value: `a = 1` and `b = 2`

Thread 1	Thread 2
<code>a = 3;</code>	—
<code>mb();</code>	—
<code>b = 4;</code>	<code>c = b;</code>
—	<code>rmb();</code>
—	<code>d = a;</code>

- `mb()` ensures that `a` is written before `b`
- `rmb()` ensures that `b` is read before `a`

Memory barrier example

- Initial value: `a = 1`, `b = 2`, and `p = &b`

Thread 1	Thread 2
<code>a = 3;</code>	—
<code>mb();</code>	—
<code>p = &a;</code>	<code>pp = p;</code>
—	<code>read_barrier_depends();</code>
—	<code>b = *pp;</code>

- `mb()` ensures that `a` is written before `p`
- `read_barrier_depends()` is sufficient because the load of `*pp` depends on the load of `p`

Memory barrier API for multi-processor

Barrier	Description
<code>smp_rmb()</code>	Provides an <code>rmb()</code> on SMP and on UP provides a <code>barrier()</code>
<code>smp_read_barrier_depends()</code>	Provides a <code>read_barrier_depends()</code> on SMP, and provides a <code>barrier()</code> on UP
<code>smp_wmb()</code>	Provides a <code>wmb()</code> on SMP, and provides a <code>barrier()</code> on UP
<code>smp_mb()</code>	Provides an <code>mb()</code> on SMP, and provides a <code>barrier()</code> on UP
<code>barrier()</code>	Prevents the compiler from optimizing stores or loads across the barrier

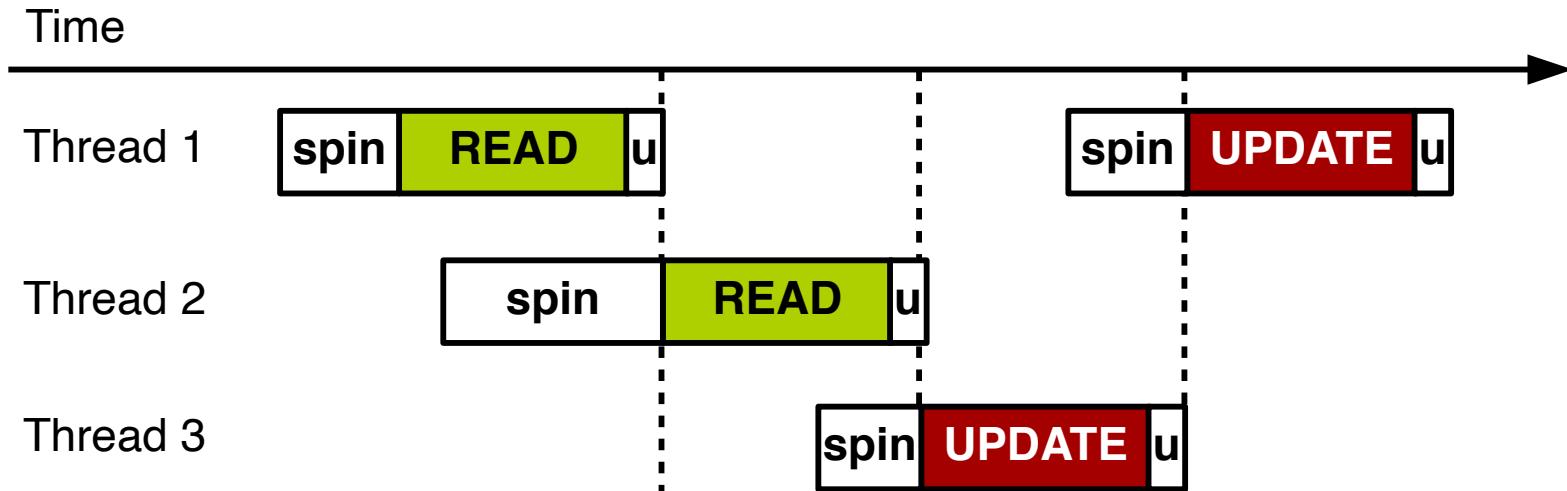
- On SMP kernel they are defined as the usual memory barriers
- On UP kernel they are defined only as a compiler barrier

Recap: Synchronization primitives

- Protect shared data from concurrent access
- Non-sleeping (non-blocking) synchronization primitives
 - atomic operation, spinlock, reader-write lock (rwlock), sequential lock (seqlock)
- Sleeping (blocking) synchronization primitives
 - semaphore, mutex, completion variable, wait queue

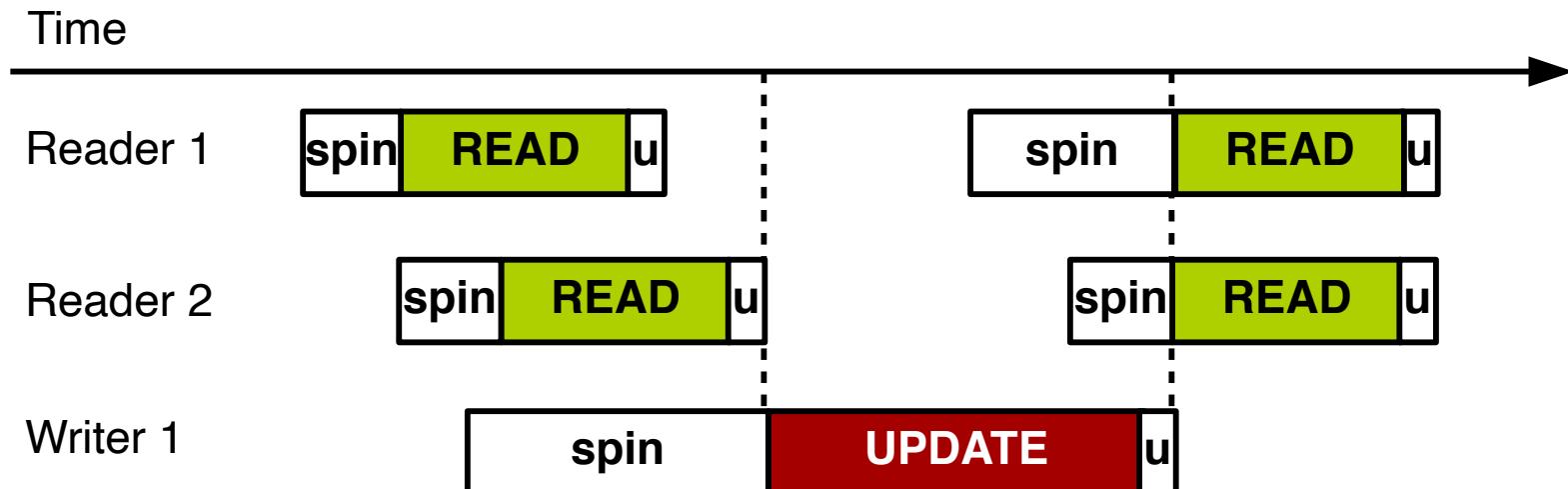
Recap: spinlock

- Implement by mutual exclusion



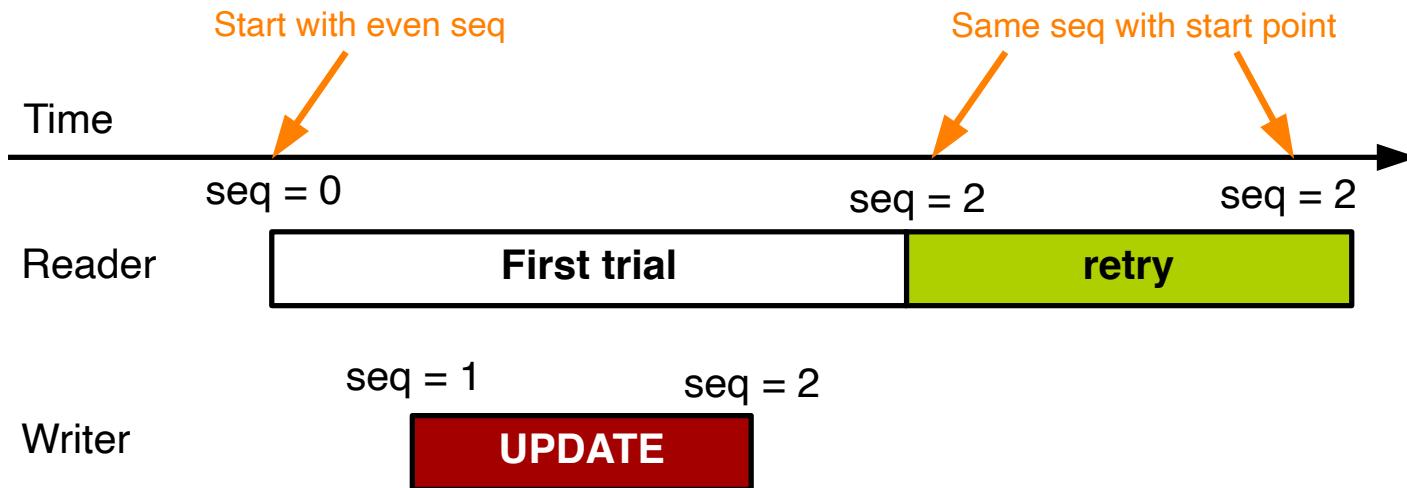
Recap: rwlock

- Allow multiple readers
- Mutual exclusion between readers and a writer
- Linux rwlock is a reader-preferred algorithm



Recap: seqlock

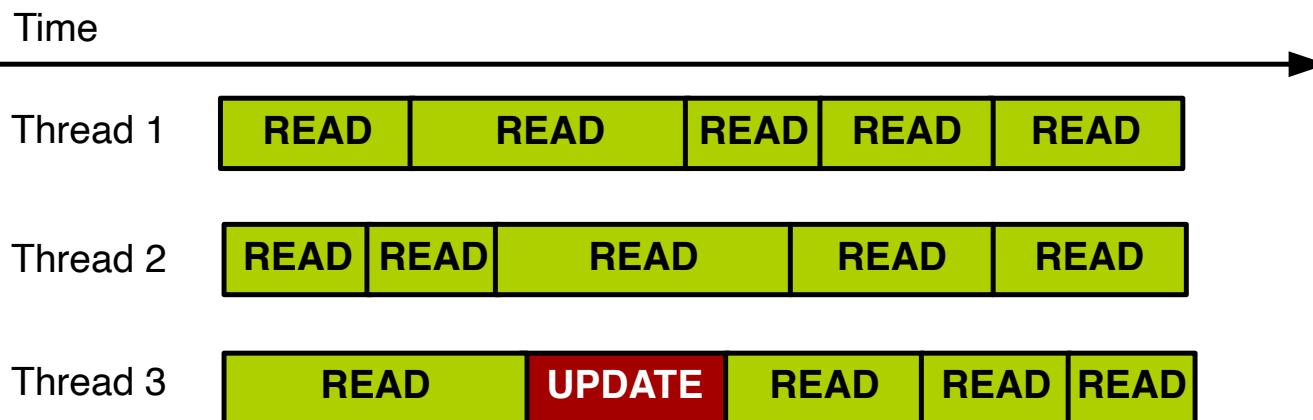
- Consistent mechanism without starving writers



Read-Copy-Update (RCU) design principle

- RCU supports concurrency between multiple readers a single writer.

- A writer does not block readers!
- Allow multiple readers with almost zero overhead
- Optimize for reader performance



Read-Copy-Update (RCU)

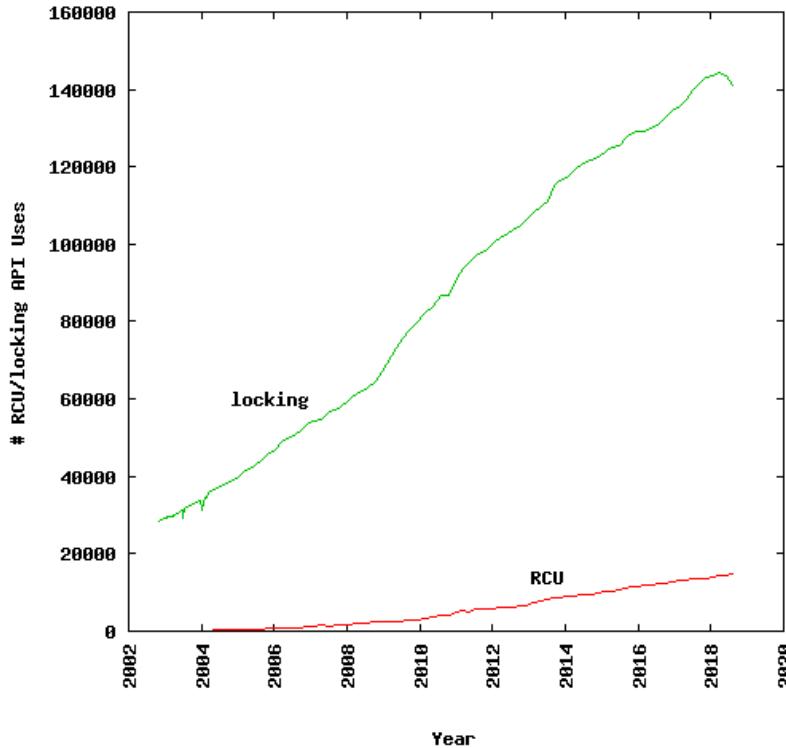
- Only require locks for writes; carefully update data structures so readers see consistent views of data all the time
- RCU ensures that reads are coherent by maintaining multiple version of objects and ensuring that they are not freed up until all pre-existing read-side critical sections complete.
- Widely-used for read-mostly data structures
 - Directory entry cache, DNS name database, etc.

Who developed RCU?

- Paul McKenney @ Facebook



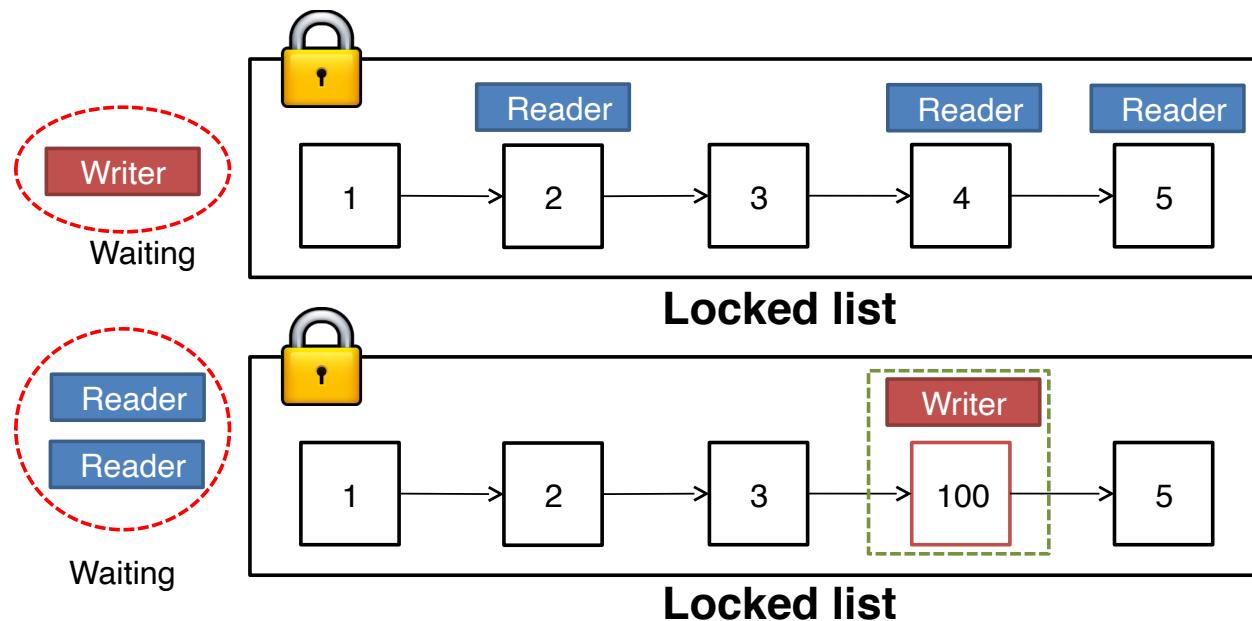
How much is RCU used in the Linux kernel?



- Source: [RCU Linux Usage](#)

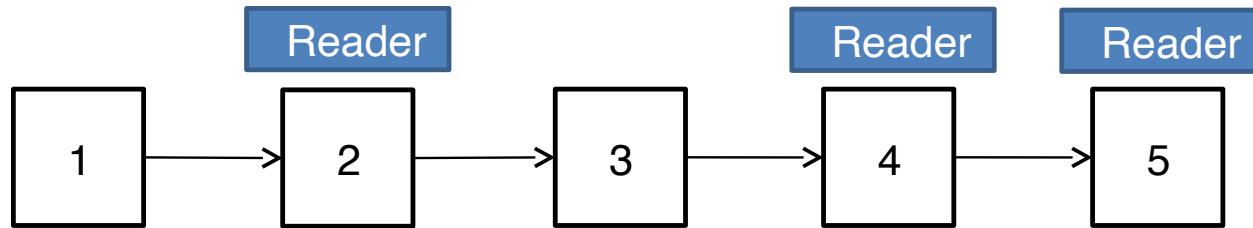
RWLock-based linked list

- Even using a scalable rwlock, readers and a writer cannot concurrently access the list



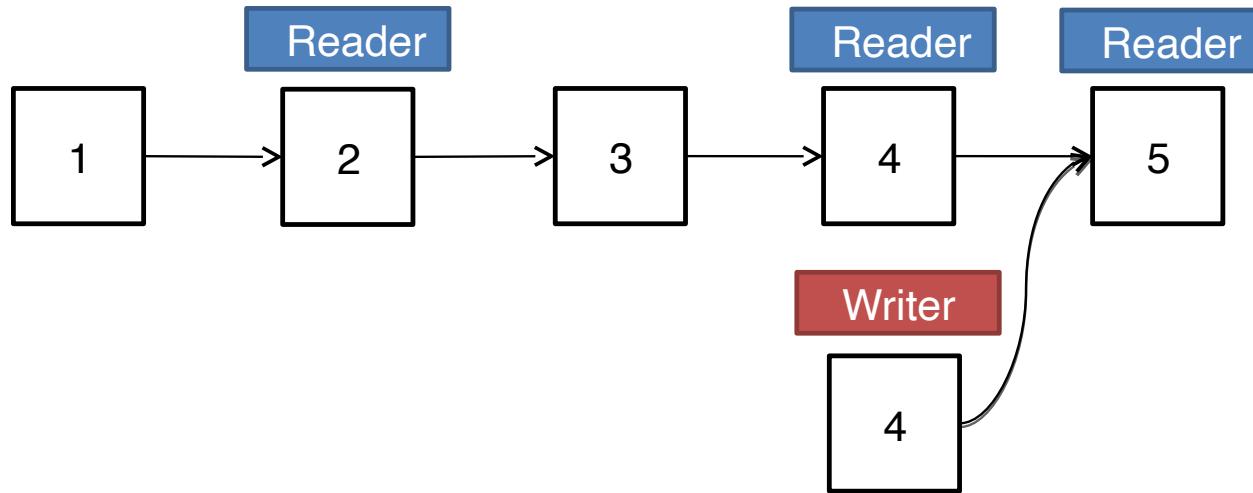
RCU-based linked list

- Allow concurrent access of readers



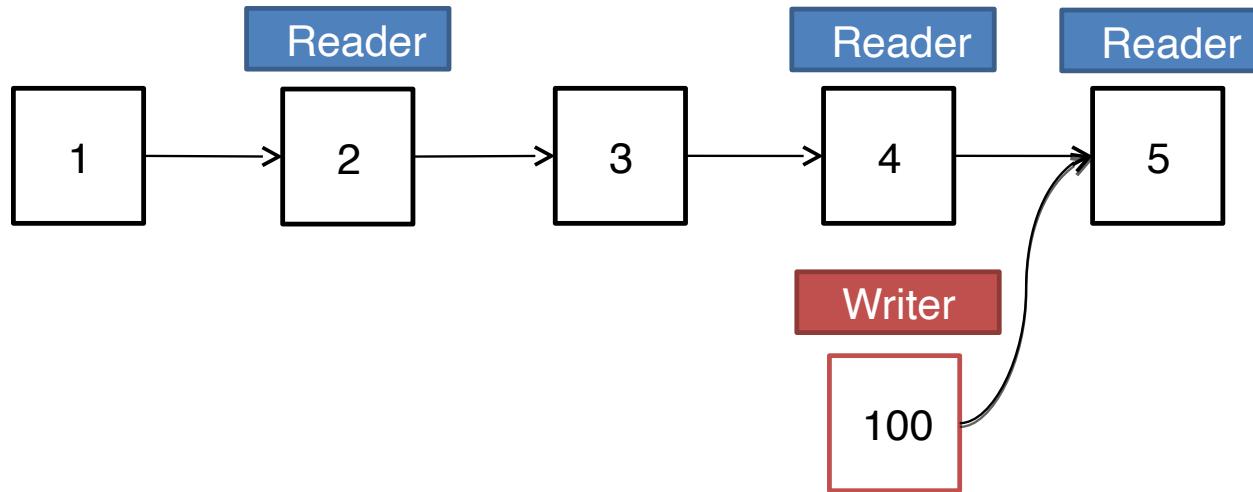
RCU-based linked list

- A writer copies an element first



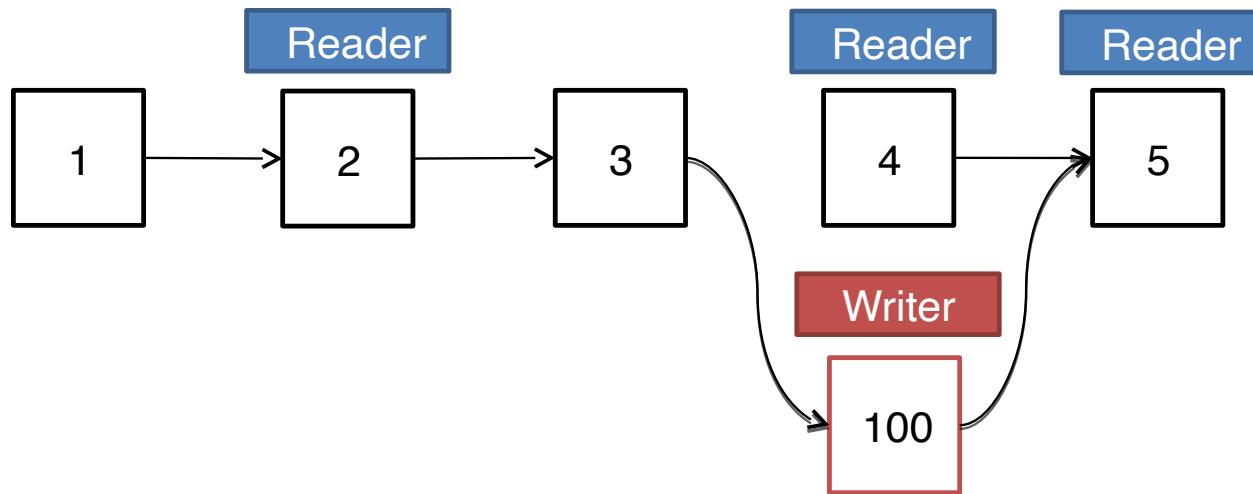
RCU-based linked list

- And then it updates the element



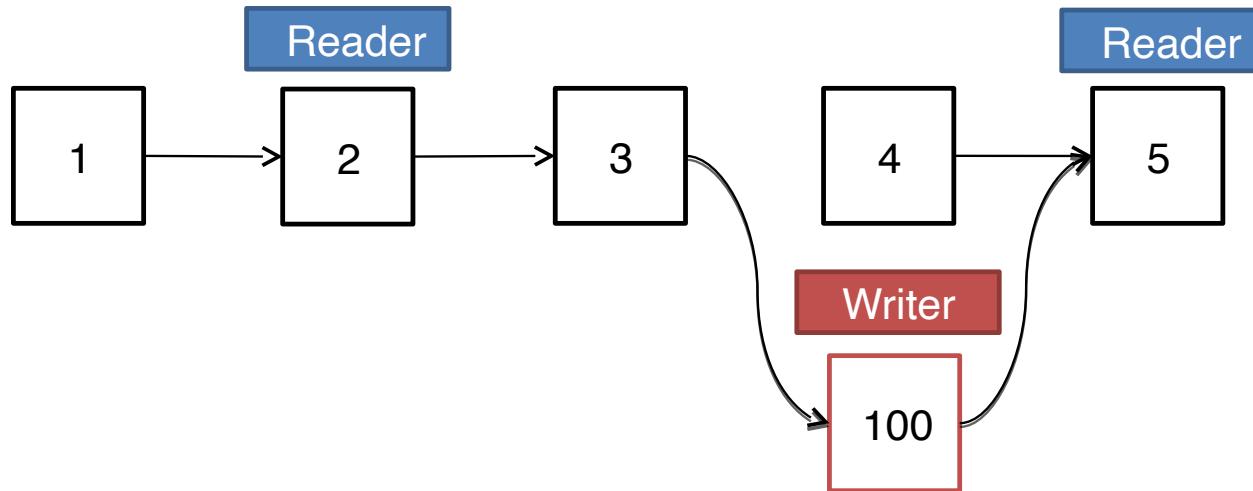
RCU-based linked list

- And then it makes its change public by updating the next pointer of its previous. → New readers will traverse 100 instead of 4



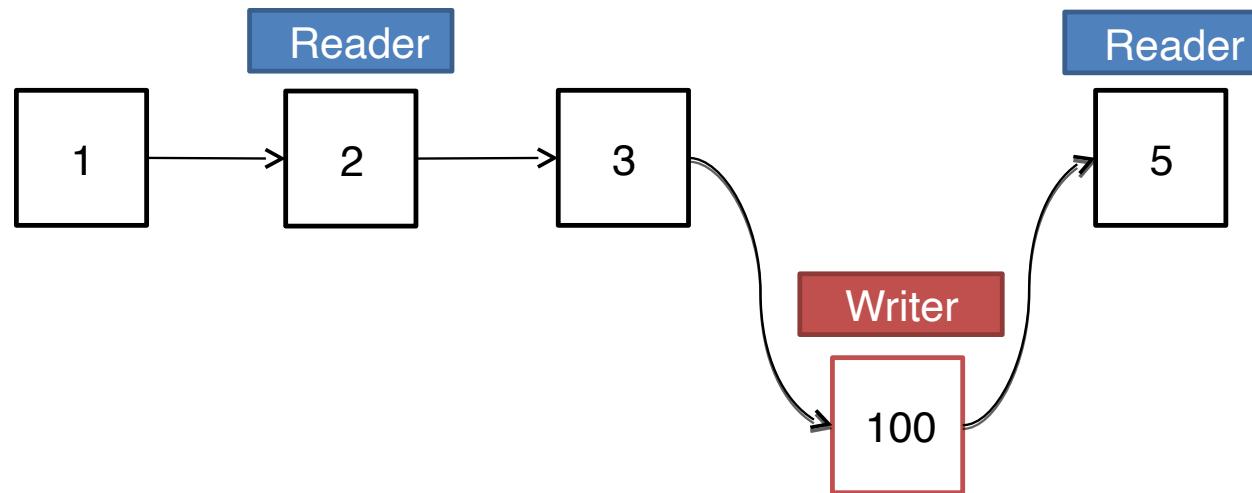
RCU-based linked list

- Do not free the old node, 4 , until any reader accesses it.



RCU-based linked list

- When it is guaranteed that there is no reader accessing the old node, free the old node.



RCU API

```
/* linux/include/linux/rcupdate.h */

/* Mark the beginning of an RCU read-side critical section */
void rCU_read_lock(void);

/* Mark the end of an RCU read-side critical section */
void rCU_read_unlock(void);

/* Assign to RCU-protected pointer: p = v
 * @p: pointer to assign to
 * @v: value to assign (publish) */
#define rCU_assign_pointer(p, v) ..

/* Fetch RCU-protected pointer for dereferencing
 * @p: The pointer to read, prior to dereferencing */
#define rCU_dereference(p) ...

/* Queue an RCU callback for invocation after a grace period.
 * @head: structure to be used for queueing the RCU updates.
 * @func: actual callback function to be invoked after the grace period */
void call_rcu(struct rCU_head *head, rCU_callback_t func);

/* Wait until quiescent states */
void synchronize_rcu(void);
```

Replace rwlock by RCU

```
/* RWLock */
1 struct el {
2     struct list_head lp;
3     long key;
4     int data;
5     /* Other data fields */
6 };
7 DEFINE_RWLOCK(listlock);
8 LIST_HEAD(head);
```

```
/* RCU */
1 struct el {
2     struct list_head lp;
3     long key;
4     int data;
5     /* Other data fields */
6 };
7 DEFINE_SPINLOCK(listlock);
8 LIST_HEAD(head);
```

Replace rwlock by RCU

```
/* RLock */
1 int search(long key, int *result)
2 {
3     struct el *p;
4
5     read_lock(&listlock);
6     list_for_each_entry(p,&head,lp) {
7         if (p->key == key) {
8             *result = p->data;
9             read_unlock(&listlock);
10            return 1;
11        }
12    }
13    read_unlock(&listlock);
14    return 0;
15 }
```

```
/* RCU */
1 int search(long key, int *result)
2 {
3     struct el *p;
4
5     rcu_read_lock();
6     list_for_each_entry_rcu(p,&head,lp) {
7         if (p->key == key) {
8             *result = p->data;
9             rcu_read_unlock();
10            return 1;
11        }
12    }
13    rcu_read_unlock();
14    return 0;
15 }
```

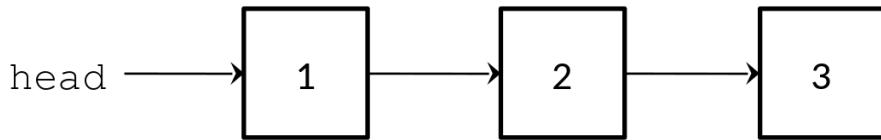
Replace rwlock by RCU

```
/* RLock */
1 int delete(long key)
2 {
3     struct el *p;
4
5     write_lock(&listlock);
6     list_for_each_entry(p, &head, lp) {
7         if (p->key == key) {
8             list_del(&p->lp);
9             write_unlock(&listlock);
10            kfree(p);
11            return 1;
12        }
13    }
14    write_unlock(&listlock);
15    return 0;
16 }

/* RCU */
1 int delete(long key)
2 {
3     struct el *p;
4
5     spin_lock(&listlock);
6     list_for_each_entry(p, &head, lp) {
7         if (p->key == key) {
8             list_del_rcu(&p->lp);
9             spin_unlock(&listlock);
10            synchronize_rcu();
11            kfree(p);
12            return 1;
13        }
14    }
15    spin_unlock(&listlock);
16    return 0;
17 }
```

RCU Primer

Lock-free reads + Single pointer update + Delayed free

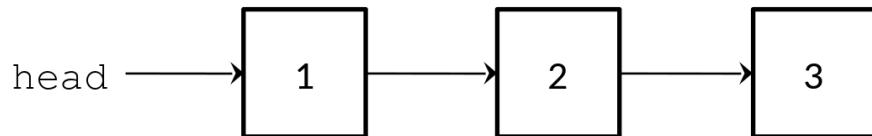


```
length() {
    rcu_read_lock();
    p=rcu_dereference(head); //p=head
    for(i=0;p;p=p->next,i++);
    } rcu_read_unlock();
    return i;
}
```

```
pop_n(n) {
    for(p=head;p&&n;p=p->next,n--)
        call_rcu(free, p);
    rcu_assign_pointer(head,p); //head=p
}
```

RCU Primer

Lock-free reads + Single pointer update + Delayed free



```

length() {
    rCU_read_lock();
    p=rcu_dereference(head); //p=head
    for(i=0;p;p=p->next,i++) ;
} rCU_read_unlock();
return i;
}

```

```

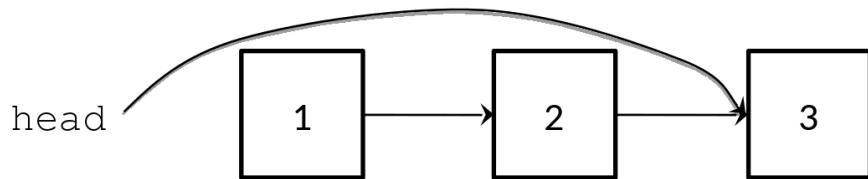
pop_n(n) {
    for(p=head;p&&n;p=p->next,n--)
        call_rcu(free, p);
    rCU_assign_pointer(head,p); //head=p
}

```

- No locks, no barriers
- `rcu_read_lock()` just sets the status of a thread “reading” RCU data.

RCU Primer

Lock-free reads + Single pointer update + Delayed free



```

length() {
    rcu_read_lock();
    p=rcu_dereference(head); //p=head
    for(i=0;p;p=p->next, i++);
    } rcu_read_unlock();
    return i;
}
  
```

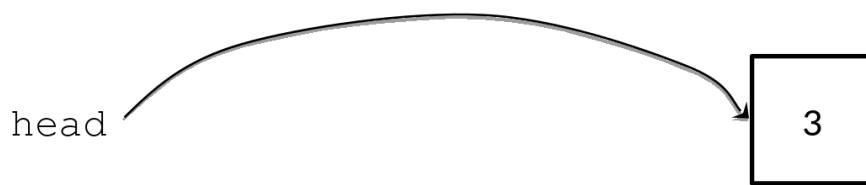
```

pop_n(n) {
    for(p=head;p&&n;p=p->next, n--)
        call_rcu(free, p);
    rcu_assign_pointer(head,p); //head=p
}
  
```

- No locks, no barriers
- `rcu_read_lock()` just sets the status of a thread “reading” RCU data.
- Update exactly one pointer, which is atomic.

RCU Primer

Lock-free reads + **Single pointer update** + **Delayed free**

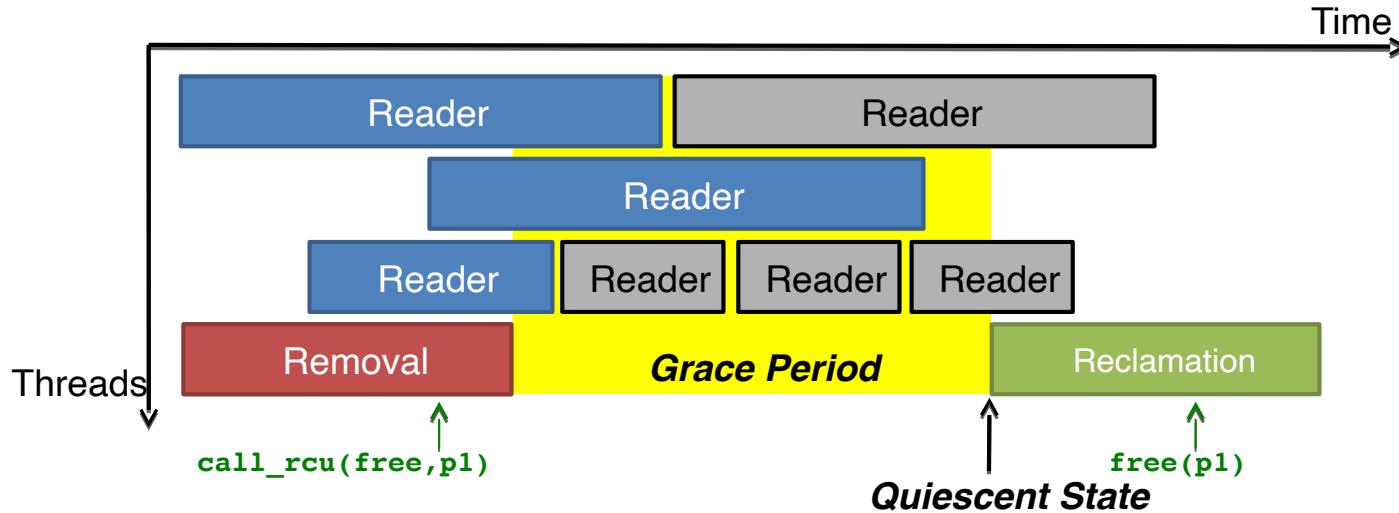


```
length() {
    rCU_read_lock();
    p=rcu_dereference(head); //p=head
    for(i=0;p;p=p->next, i++);
    } rCU_read_unlock();
    return i;
}
```

```
pop_n(n) {
    for(p=head;p&&n;p=p->next, n--)
        call_rcu(free, p);
    rCU_assign_pointer(head,p); //head=p
}
```

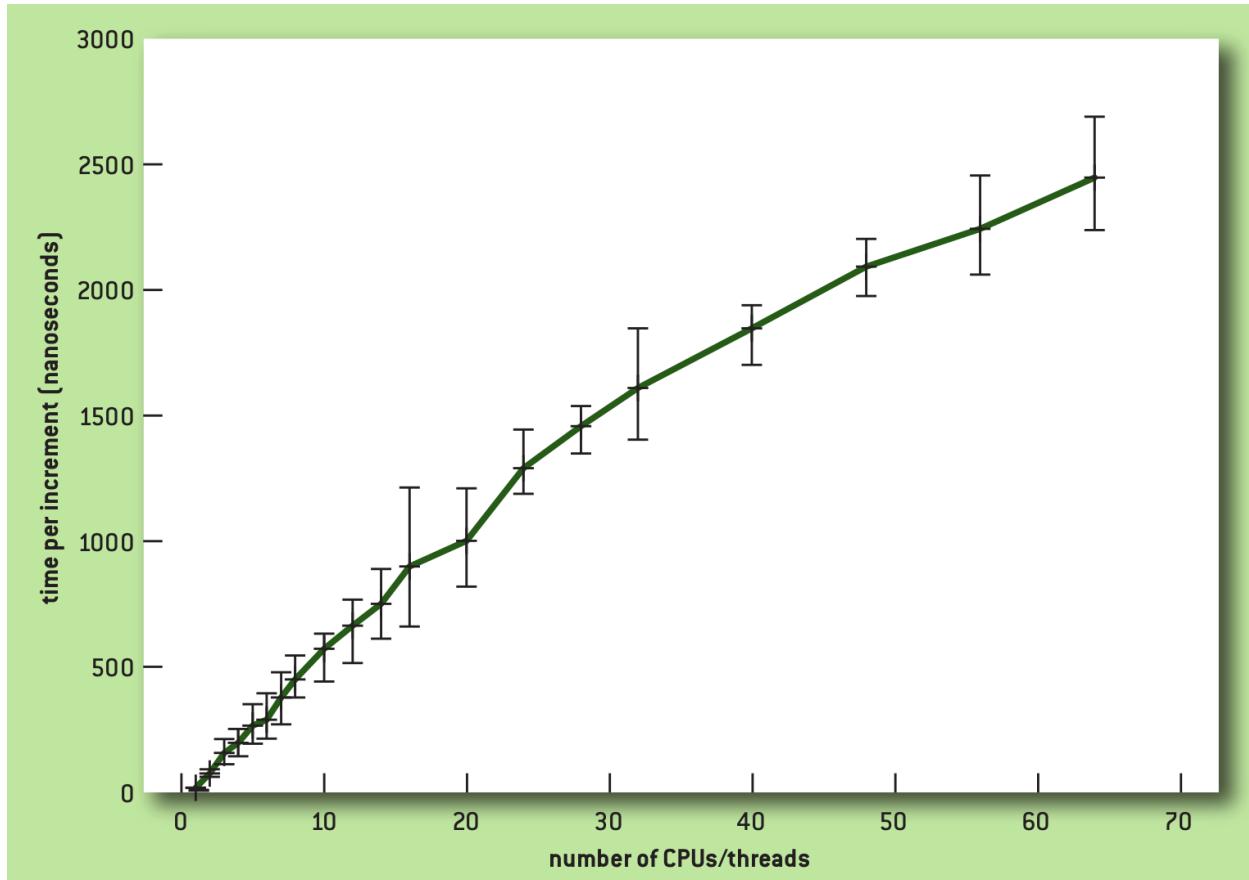
- No locks, no barriers
- `rcu_read_lock()` just sets the status of a thread “reading” RCU data.
- Update exactly one pointer, which is atomic.
- Free delayed until all readers return (e.g., by waiting for all CPU’s to schedule)

Delayed free: grace period, quiescent state



- Efficient and scalable grace period detection is a key challenge
 - Some obvious solutions, such as reference counting, never work.

Atomic increment does not scale



Toy RCU implementation

```
static inline void rcu_read_lock(void)
{
    preempt_disable();
}

static inline void rcu_read_unlock(void)
{
    preempt_enable();
}

#define rcu_assign_pointer(p, v)      ({ \
    smp_wmb(); \
    ACCESS_ONCE(p) = (v); \
})

#define rcu_dereference(p)          ({ \
    typeof(p) _value = ACCESS_ONCE(p); \
    smp_read_barrier_depends(); /* nop on most architectures */ \
    (_value); \
})
```

Toy RCU implementation

```
void call_rcu(void (*callback) (void *), void *arg)
{
    /* add callback/arg pair to a list */
}

void synchronize_rcu(void)
{
    int cpu, ncpus = 0;

    for_each_cpu(cpu)
        schedule_current_task_to(cpu);

    for each entry in the call_rcu list
        entry->callback (entry->arg);
}
```

RCU list

```
/* linux/include/linux/rculist.h */
/* Circular doubly-linked list */

/* Add a new entry to rcu-protected list
 * @new: new entry to be added
 * @head: list head to add it after */
void list_add_rcu(struct list_head *new, struct list_head *head);

/* Deletes entry from list without re-initialization
 * @entry: the element to delete from the list. */
void list_del_rcu(struct list_head *entry);

/* Replace old entry by new one
 * @old : the element to be replaced
 * @new : the new element to insert */
void list_replace_rcu(struct list_head *old, struct list_head *new);

/* Iterate over rcu list of given type
 * @pos:    the type * to use as a loop cursor.
 * @head:   the head for your list.
 * @member: the name of the list_head within the struct. */
#define list_for_each_entry_rcu(pos, head, member) ..
```

RCU hlist

```
/* linux/include/linux/rculist.h */
/* Non-circular doubly-linked list */

/* Adds the specified element to the specified hlist,
 * while permitting racing traversals.
 * @n: the element to add to the hash list.
 * @h: the list to add to. */
void hlist_add_head_rcu(struct hlist_node *n, struct hlist_head *h);

/* Replace old entry by new one
 * @old : the element to be replaced
 * @new : the new element to insert */
void hlist_replace_rcu(struct hlist_node *old, struct hlist_node *new);

/* Deletes entry from hash list without re-initialization
 * @n: the element to delete from the hash list. */
void hlist_del_rcu(struct hlist_node *n);

/* Iterate over rcu list of given type
 * @pos:    the type * to use as a loop cursor.
 * @head:   the head for your list.
 * @member: the name of the hlist_node within the struct. */
#define hlist_for_each_entry_rcu(pos, head, member) ...
```

Limitations of RCU

- Do not provide a mechanism to coordinate multiple writers
 - Most RCU-based algorithms end up using `spinlock` to prevent concurrent write operations
- All modification should be `a single-pointer-update`.
 - This is challenging!

Further readings

- [Read-log-update: a lightweight synchronization mechanism for concurrent programming, SOSP15](#)
- [Is Parallel Programming Hard, And, If So, What Can You Do About It?](#)
- [Structured Deferral: Synchronization via Procrastination](#)
- [Introduction to RCU Concepts](#)
- [LWN: What is RCU, Fundamentally?](#)
- [Notes on Read-Copy Update](#)
- [Tiny Little Things for Manycore Scalability: Scalable Locking and Lockless Data Structures](#)

Timers and Time Management

Dongyoon Lee

Summary of last lectures

- Tools: building, exploring, and debugging Linux kernel
- Core kernel infrastructure
 - syscall, module, kernel data structures
- Process management & scheduling
- Interrupt & interrupt handler
- Kernel synchronization

Today: timers and time management

- Kernel notion of time
- Tick rate and Jiffies
- hardware clocks and timers
- Timers
- Delaying execution
- Time of day

Kernel notion of time

- Having the notion of time passing in the kernel is essential in multiple cases:
 - Perform periodic tasks (e.g., CFS time accounting)
 - Delay some processing at a relative time in the future
 - Give the time of the day
- **Absolute vs relative** time

Kernel notion of time

- Central role of the system timer
 - Periodic interrupt, system timer interrupt
 - Update system uptime, time of day, balance runqueues, record statistics, etc.
 - Pre-programmed frequency, timer tick rate
 - $\text{tick} = 1/(\text{tick rate})$ seconds
- Dynamic timers to schedule event a relative time from now in the future

Tick rate and jiffies

- The tick rate (system timer frequency) is defined in the `HZ` variable
- Set to `CONFIG_HZ` in `include/asm-generic/param.h`
 - Kernel compile-time configuration option
- Default value is per-architecture:

Architecture	Frequency (HZ)	Period (ms)
x86	1000	1
ARM	100	10
PowerPC	100	10

Tick rate: the ideal **Hz** value

- High timer frequency → high precision
 - Kernel timers (finer resolution)
 - System call with timeout value (e.g., `poll`) → significant performance improvement for some applications
 - Timing measurements

Tick rate: the ideal **Hz** value

- High timer frequency → high precision
 - Process preemption occurs more accurately → low frequency allows processes to potentially get (way) more CPU time after the expiration of their timeslices
- High timer frequency → more timer interrupt → larger overhead
 - Not very significant on modern hardware

Tickless OS

- Option to compile the kernel as a tickless system
 - NO_HZ family of compilation options
- The kernel dynamically reprogram the system timer according to the current timer status
 - Situation in which there are no events for hundreds of milliseconds
- Overhead reduction, Energy savings
 - CPUs spend more time in low power idle states

jiffies

- A global variable holds the number of timer ticks since the system booted (`unsigned long`)
- Conversion between `jiffies` and seconds
 - `jiffies` = seconds * `HZ`
 - seconds = `jiffies` / `HZ`

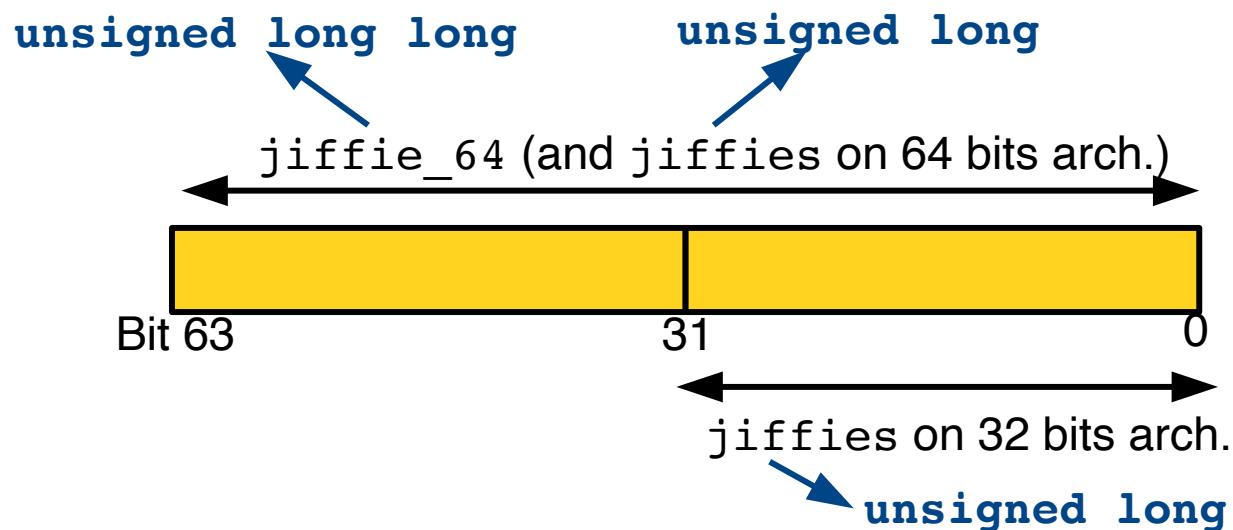
```
unsigned long time_stamp = jiffies;          /* Now */  
unsigned long next_tick = jiffies + 1;        /* One tick from now */  
unsigned long later = jiffies + 5*HZ;          /* 5 seconds from now */  
unsigned long fraction = jiffies + HZ/10;      /* 100 ms from now */
```

Internal representation of jiffies

- `sizeof(jiffies)` is 32 bits on 32-bit architectures and 64 bits for 64-bit architectures
- On a 32 bits variable with `HZ == 100`, overflows in 497 days
 - Still on 32 bits with `HZ == 1000`, overflows in 50 days
- But on a 64 bits variable, no overflow for a very long time

Internal representation of jiffies

- We want access to a 64 bits variable while still maintaining an unsigned long on both architectures → linker magic



jiffies wraparound

- An unsigned integer going over its maximum value wraps around to zero
 - On 32 bits, $0xFFFFFFFF + 0x1 == 0x0$

```
/* WARNING: THIS CODE IS BUGGY */
unsigned long timeout = jiffies + HZ/2; /* timeout in 0.5s */

/* do some work ... */

/* then see whether we took too long */
if (timeout > jiffies) { /* What happen if jiffies wrapped back to zero? */
    /* we did not time out, good ... */
} else {
    /* we timed out, error ... */
}
```

jiffies wraparound

```
/* linux/include/linux/jiffies.h */
#define time_after(a,b)
#define time_before(a,b)
#define time_after_eq(a,b)
#define time_before_eq(a,b)

/* -----
 * An example of using a time_() macro */
unsigned long timeout = jiffies + HZ/2; /* timeout in 0.5s */

/* do some work ... */

/* then see whether we took too long */
if (time_before(jiffies, timeout)) { /* Use time_() macros */
    /* we did not time out, good ... */
} else {
    /* we timed out, error ... */
}
```

Userspace and HZ

- For conversion between architecture-specific `jiffies` and user-space clock tick, Linux kernel provides APIs and macros
- `USER_HZ` : user-space clock tick (100 in x86)
- Conversion between `jiffies` and user-space clock tick
 - `clock_t jiffies_to_clock(unsigned long x);`
 - `clock_t jiffies_64_to_clock_t(u64 x);`
- [clock\(3\)](#)

Hardware clocks and timers

- Real-Time Clock (RTC)
 - Stores the wall-clock time (still incremented when the computer is powered off)
 - Backed-up by a small battery on the motherboard
 - Linux stores the wall-clock time in a data structure at boot time
`(xtime)`

Hardware clocks and timers

- System timer
 - Provide a mechanism for driving an interrupt at a periodic rate regardless of architecture
 - System timers in x86
 - Local APIC timer: primary timer today
 - Programmable interrupt timer (PIT): was a primary timer until 2.6.17

Hardware clocks and timers

- Processor's time stamp counter (TSC)
 - `rdtsc`, `rdtscp`
 - most accurate (CPU clock resolution)
 - invariant to clock frequency (x86 architecture)
 - $\text{seconds} = \text{clocks} / \text{maximum CPU clock Hz}$

Timer interrupt processing

- Constituted of two parts: (1) architecture-dependent and (2) architecture-independent
- Architecture-dependent part is registered as the handler (top-half) for the timer interrupt
 1. Acknowledge the system timer interrupt (reset if needed)
 2. Save the wall clock time to the RTC
 3. Call the architecture independent function (still executed as part of the top-half)

Timer interrupt processing

- Architecture independent part: `tick_handle_periodic()`
 1. Call `tick_periodic()`
 2. Increment `jiffies64`
 3. Update statistics for the currently running process and the entire system (load average)
 4. Run dynamic timers
 5. Run `scheduler_tick()`

Timer interrupt processing

```
/* linux/kernel/time/tick-common.c */

static void tick_periodic(int cpu)
{
    if (tick_do_timer_cpu == cpu) {
        write_seqlock(&jiffies_lock);

        /* Keep track of the next tick event */
        tick_next_period =
            ktime_add(tick_next_period, tick_period);

        do_timer(1); /* ! */
        write_sequnlock(&jiffies_lock);
        update_wall_time(); /* ! */
    }

    update_process_times(
        user_mode(get_irq_regs())); /* ! */
    profile_tick(CPU_PROFILING);
}
```

do_timer()

```
/* linux/kernel/time/timekeeping.c */  
  
void do_timer(unsigned long ticks)  
{  
    jiffies_64 += ticks;  
    calc_global_load(ticks);  
}
```

update_process_times()

- Call account_process_tick() to add one tick to the time passed:
 - In a process in user space
 - In a process in kernel space
 - In the idle task
- Call `run_local_timers()` and run expired timers
 - Raise the TIMER_SOFTIRQ softirq

update_process_times()

- Call `scheduler_tick()`
 - Call the `task_tick()` function of the currently running process's scheduler class → Update timeslices information → Set `need_resched` if needed
 - Perform CPU runqueues load balancing (raise the `SCHED_SOFTIRQ` softirq)

Timer

- Timers == dynamic timers == kernel timers
 - Used to delay the execution of some piece of code for a given amount of time

```
/* linux/include/linux/timer.h */

struct timer_list {
    struct hlist_node entry; /* linked list of timers */
    unsigned long      expires; /* expiration time in jiffies */
    void (*function)(unsigned long); /* handler */
    unsigned long      data; /* argument of the handler */
    u32                flags; /* */
        TIMER_IRQSAFE: executed with interrupts disabled
        TIMER_DEFERRABLE: does not wake up an idle CPU */
    /* ... */
}
```

Using timers

```
/* Declaring, initializing and activating a timer */
void handler_name(unsigned long data)
{
    /* executed when the timer expires */
    /* ... */
}

void another_function(void)
{
    struct timer_list my_timer;

    init_time(&my_timer);                      /* initialize internal fields */
    my_timer.expires = jiffies + 2*HZ;          /* expires in 2 secs */
    my_timer.data = 42;                         /* 42 passed as parameter to the handler */
    my_timer.function = handler_name;

    /* activate the timer: */
    add_timer(&my_timer);
}

/*****************/
/* Modify the expiration date of an already running timer */
mod_timer(&my_timer, jiffies + another_delay);
```

Using timers

- `del_timer(struct timer_list *)`
 - Deactivate a timer prior
 - Returns 0 if the timer is already inactive, and 1 if the timer was active
 - Potential race condition on SMP when the handler is currently running on another core

Using timers

- `del_timer_sync(struct timer_list *)`
 - Waits for a potential currently running handler to finishes before removing the timer
 - Can be called from interrupt context only if the timer is irqsafe (declared with TIMER_IRQSAFE)
 - Interrupt handler interrupting the timer handler and calling `del_timer_sync()` → deadlock

Timer race conditions

- Timers run in softirq context → Several potential race conditions exist
- Protect data shared by the handler and other entities
- Use `del_timer_sync()` rather than `del_timer()`
- Do not directly modify the `expire` field; use `mod_timer()`

```
/* THIS CODE IS BUGGY! DO NOT USE! */
del_timer(&my_timer);
my_timer->expires = jiffies + new_delay;
add_timer(&my_timer);
```

Timer implementation

- In the system timer interrupt handler, `update_process_times()` is called
 - Calls `run_local_timers()`
 - Raises a softirq (`TIMER_SOFTIRQ`)

Timer implementation

- Softirq handler is `run_timer_softirq()` and it calls `__run_timers()`
 - Grab expired timers through `collect_expired_timers()`
 - Executes `function` handlers with `data` parameters for expired timers with `expire_timers()`
- **Timer handlers are executed in interrupt (softirq) context**

Timer example

```
#include <linux/module.h>
#include <linux/kernel.h>
#include <linux/init.h>
#include <linux/timer.h>

#define PRINT_PREF "[TIMER_TEST] "

struct timer_list my_timer;

static void my_handler(unsigned long data)
{
    printk(PRINT_PREF "handler executed!\n");
}

static int __init my_mod_init(void)
{
    printk(PRINT_PREF "Entering module.\n");

    /* initialize the timer data structure internal values: */
    init_timer(&my_timer);
```

Timer example

```
/* fill out the interesting fields: */
my_timer.data = 0;
my_timer.function = my_handler;
my_timer.expires = jiffies + 2*HZ; /* timeout == 2secs */

/* start the timer */
add_timer(&my_timer);
printk(PRINT_PREF "Timer started\n");

return 0;
}

static void __exit my_mod_exit(void)
{
    del_timer(&my_timer);
    printk(PRINT_PREF "Exiting module.\n");
}

module_init(my_mod_init);
module_exit(my_mod_exit);
```

Delaying execution

- Sometimes the kernel needs to wait for some time without using timers (bottom-halves)
 - For example drivers communicating with the hardware
 - Needed delay can be quite short, sometimes shorter than the timer tick period
- Several solutions
 - Busy looping
 - Small delays and BogoMIPS
 - `schedule_timeout()`

Busy looping

- Spin on a loop until a given amount of ticks has elapsed
 - Can use `jiffies`, `HZ`, or `rdtsc`
 - Busy looping is good for delaying very short period time but in general it is sub-optimal as wasting CPU cycles.

Busy looping

- A better solution is to leave the CPU while waiting using `cond_resched()`
 - `cond_resched()` invokes the scheduler only if the `need_resched` flag is set
 - Cannot be used from interrupt context (not a schedulable entity)
 - Pure busy looping is probably also not a good idea from interrupt handlers as they should be fast
 - Busy looping can severely impact performance while a lock is held or while interrupts are disabled

Busy looping

```
/* Example 1: wait for 10 time ticks */
unsigned long timeout = jiffies + 10;      /* timeout in 10 ticks */
while(time_before(jiffies, timeout));        /* spin until now > timeout */

/* Example 2: wait for 2 seconds */
unsigned long timeout = jiffies + 2*HZ; /* 2 seconds */
while(time_before(jiffies, timeout));

/* Example 3: wait for 1000 CPU clock cycles */
unsigned long long timeout = rdtsc() + 1000;
while(rdtsc() > timeout);

/* Example 4: wait for 2 seconds using cond_resched()*/
unsigned long delay = jiffies + 2*HZ;
while(time_before(jiffies, delay))
    cond_resched(); /* WARNING: cannot use in interrupt context */
```

Small delays and BogoMIPS

- What if we want to delay for time shorter than one clock tick?
 - If `HZ` is 100, one tick is 10 ms
 - If `HZ` is 1000, one tick is 1 ms
- Use `mdelay()`, `udelay()`, or `ndelay()`
 - Implemented as a busy loop
 - `udelay/ndelay` should only be called for delays <1ms due to risk of overflow

Small delays and BogoMIPS

- Kernel knows how many loop iterations the kernel can be done in a given amount of time: **BogoMIPS**
 - Unit: iterations/jiffy
 - Calibrated at boot time
 - Can be seen in /proc/cpuinfo

Small delays and BogoMIPS

```
/* linux/include/linux/delay.h */

void mdelay(unsigned long msecs);
void udelay(unsigned long usecs); /* only for delay <1ms due to overflow */
void ndelay(unsigned long nsecs); /* only for delay <1ms due to overflow */
```

schedule_timeout()

- `schedule_timeout()` put the calling task to sleep for at least `n` ticks
 - Must change task status to `TASK_INTERRUPTIBLE` or `TASK_UNINTERRUPTIBLE`
 - Should be called from process context without holding any lock

```
set_current_state(TASK_INTERRUPTIBLE); /* can also use TASK_UNINTERRUPTIBLE */  
schedule_timeout(2 * HZ); /* go to sleep for at least 2 seconds */
```

Sleeping on a waitqueue with a timeout

- Tasks can be placed on wait queues to wait for a specific event
- To wait for such an event with a timeout:
 - Call `schedule_timeout()` instead of `schedule()`

schedule_timeout() implementation

```
signed long __sched schedule_timeout(signed long timeout)
{
    struct timer_list timer;
    unsigned long expire;

    switch (timeout)
    {
    case MAX_SCHEDULE_TIMEOUT:
        schedule();
        goto out;
    default:
        if (timeout < 0) {
            printk(KERN_ERR "schedule_timeout: wrong timeout "
                  "value %lx\n", timeout);
            dump_stack();
            current->state = TASK_RUNNING;
            goto out;
        }
    }
}
```

schedule_timeout() implementation

```
expire = timeout + jiffies;
setup_timer_on_stack(&timer, process_timeout, (unsigned long)current);
__mod_timer(&timer, expire, false);
schedule();
del_singleshot_timer_sync(&timer);

/* Remove the timer from the object tracker */
destroy_timer_on_stack(&timer);

timeout = expire - jiffies;

out:
    return timeout < 0 ? 0 : timeout;
}
```

- When the timer expires, `process_timeout()` calls
`wake_up_process()`

The time of day

- Linux provides plenty of function to get / set the time of the day
- Several data structures to represent a given point in time
 - `struct timespec` and `union ktime`

```
/* linux/include/uapi/linux/time.h */
struct timespec {
    __kernel_time tv_sec; /* seconds */
    long          tv_nsec; /* nanoseconds */
    /* __kernel_time_t is long on x86_64 */
}

/* linux/include/linux/time64.h */
#define timespec64 timespec

/* linux/include/linux/ktime.h */
union ktime {
    s64 tv64; /* nanoseconds */
};
typedef union ktime ktime_t;
```

The time of day: example

```
#include <linux/module.h>
#include <linux/kernel.h>
#include <linux/init.h>
#include <linux/timekeeping.h>
#include <linux/ktime.h>
#include <asm-generic/delay.h>

#define PRINT_PREF "[TIMEOFDAY]: "

extern void getboottime64(struct timespec64 *ts);

static int __init my_mod_init(void)
{
    unsigned long seconds;
    struct timespec64 ts, start, stop;
    ktime_t kt, start_kt, stop_kt;

    printk(PRINT_PREF "Entering module.\n");

    /* Number of seconds since the epoch (01/01/1970) */
    seconds = get_seconds();
    printk("get_seconds() returns %lu\n", seconds);
```

The time of day: example

```
/* Same thing with seconds + nanoseconds using struct timespec */
ts = current_kernel_time64();
printf(PRINT_PREF "current_kernel_time64() returns: %lu (sec),"
      "i %lu (nsec)\n", ts.tv_sec, ts.tv_nsec);

/* Get the boot time offset */
getboottime64(&ts);
printf(PRINT_PREF "getboottime64() returns: %lu (sec),"
      "i %lu (nsec)\n", ts.tv_sec, ts.tv_nsec);

/* The correct way to print a struct timespec as a single value: */
printf(PRINT_PREF "Boot time offset: %lu.%09lu secs\n",
      ts.tv_sec, ts.tv_nsec);
/* Otherwise, just using %lu.%lu transforms this:
 * ts.tv_sec == 10
 * ts.tv_nsec == 42
 * into: 10.42 rather than 10.000000042
 */

/* another interface using ktime_t */
kt = ktime_get();
printf(PRINT_PREF "ktime_get() returns %llu\n", kt.tv64);
```

The time of day: examples

```
/* Subtract two struct timespec */
getboottime64(&start);
stop = current_kernel_time64();
ts = timespec64_sub(stop, start);
printf(PRINT_PREF "Uptime: %lu.%09lu secs\n", ts.tv_sec, ts.tv_nsec);

/* measure the execution time of a piece of code */
start_kt = ktime_get();
udelay(100);
stop_kt = ktime_get();

kt = ktime_sub(stop_kt, start_kt);
printf(PRINT_PREF "Measured execution time: %llu usecs\n", (kt.tv64)/1000);

return 0;
}
```

The time of day: examples

```
static void __exit my_mod_exit(void)
{
    printk(PRINT_PREF "Exiting module.\n");
}

module_init(my_mod_init);
module_exit(my_mod_exit);

MODULE_LICENSE("GPL");
```

Next lecture

- Memory management

Memory Management

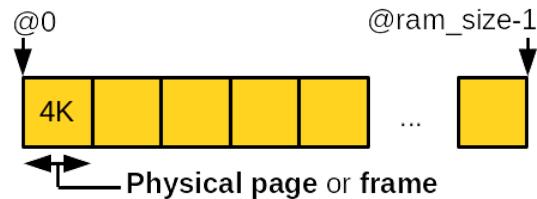
Dongyoon Lee

Today: Memory Management

- Pages and zones
- Page allocation
- kmalloc, vmalloc
- Slab allocator
- Stack, high memory, per-CPU data structures

Pages

- Memory is divided into **physical pages** or **frames**



- The **page** is the basic management unit in the kernel
- Page size is machine-dependent
 - Determined by the the memory management unit (MMU) support
 - 4 KB** in general, some are 2 MB and 1 GB: `getconf PAGESIZE`

Pages

- Each **physical page** is represented by `struct page`
- Defined in `include/linux/mm_types.h`

```
struct page {  
    unsigned long flags;      /* page status (permission, dirty, etc.) */  
    unsigned counters;        /* usage count */  
    struct address_space *mapping;  
                           /* address space mapping */  
    pgoff_t index;           /* offset within the mapping */  
    struct list_head lru;    /* LRU list buffer cache */  
    void *virtual;           /* virtual address */  
}
```

Pages

- The kernel uses `struct page` to keep track of the owner of the page
 - User-space process, kernel statically/dynamically allocated data, page cache, etc.
- There is one `struct page` object per physical memory page
 - `sizeof(struct page)` : 64 bytes
 - Assuming 8GB of RAM and 4K-sized pages: 128MB reserved for `struct page` objects (~1.5%)

Zones

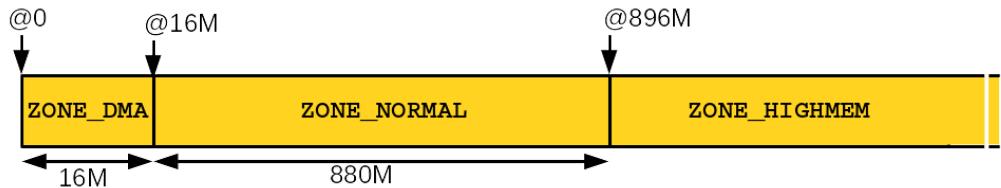
- Certain contexts require certain physical pages due to hardware limitations
 - Some devices can only access the lowest 16 MB of physical memory
 - High memory should be mapped before being accessed
- Physical memory is partitioned into **zones** having the same constraints
 - Zone layout is architecture- and machine-dependent
- Page allocator considers the constraints while allocating pages

Zones

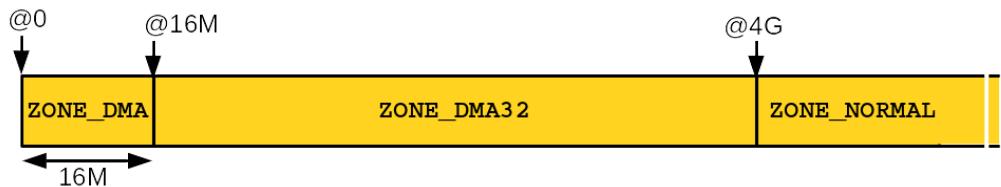
Name	Description
ZONE_DMA	Pages can be used for DMA
ZONE_DMA32	Pages for 32-bit DMA devices
ZONE_NORMAL	Pages always mapped to the address space
ZONE_HIGHMEM	Pages should be mapped prior to access

Zones

- x86_32 zones layout



- x86_64 zones layout



Zones

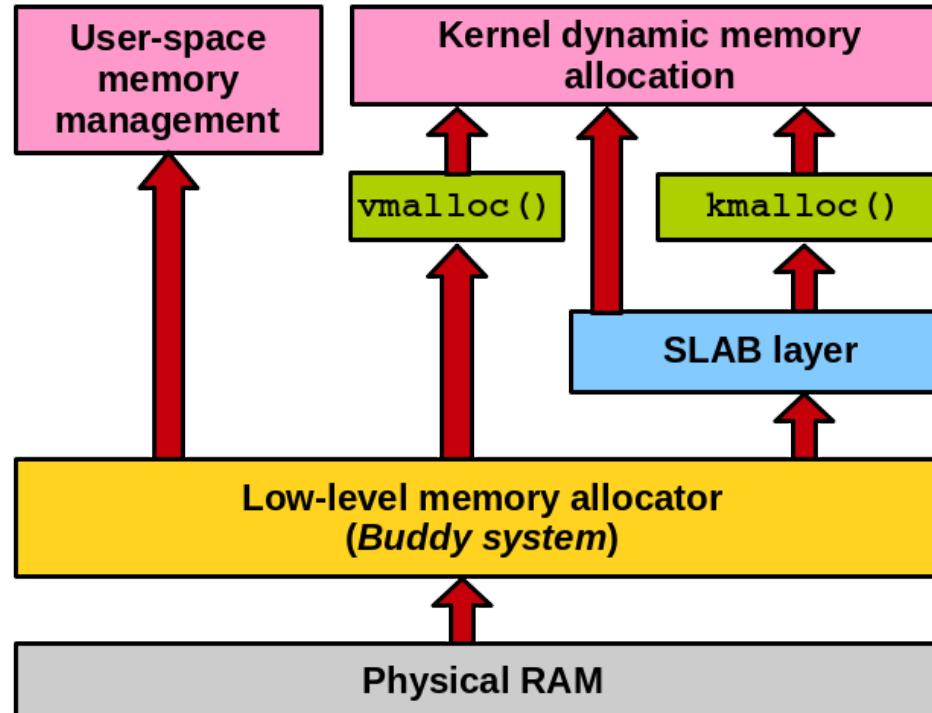
- Each zone is managed with `struct zone` data structure defined in
`include/linux/mmzone.h`

```
struct zone {  
    const char *name;                      /* Name of this zone */  
    unsigned long zone_start_pfn;          /* starting page frame number of the zone */  
    unsigned long watermark[NR_WMARK];  
        /* minimum, low, and high watermarks  
         * for per-zone memory allocation */  
    spinlock_t lock; /* protects against concurrent accesses */  
    struct free_area free_area[MAX_ORDER];  
        /* list of free pages of different sizes */  
}
```

Memory layout (x86_32)

Memory layout (x86_32)

Hierarchy of memory allocators

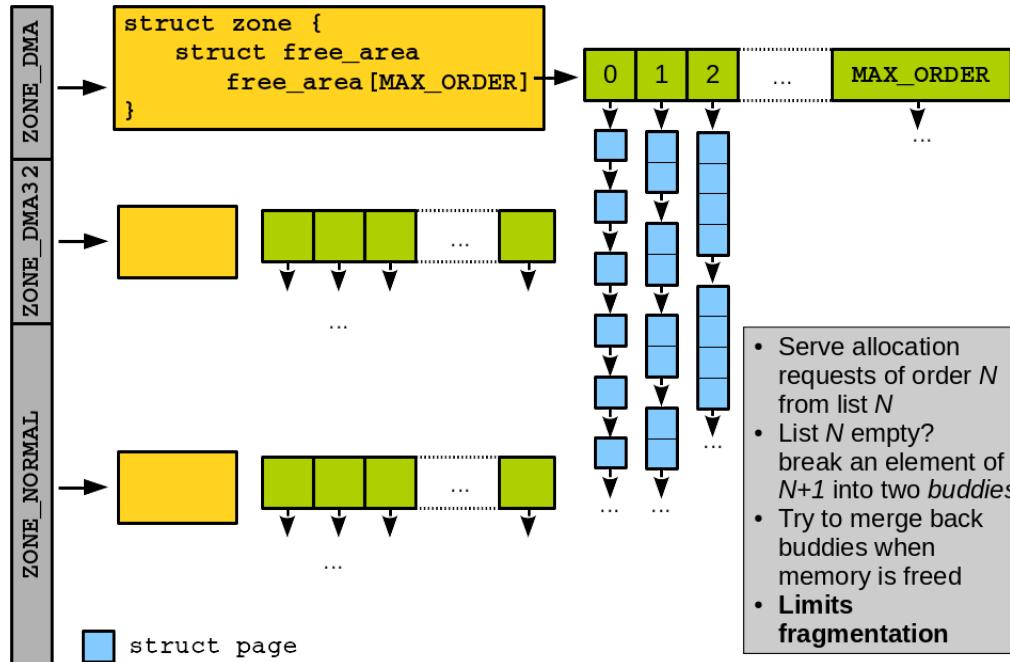


Low-level memory allocator (Buddy system)

- Low-level mechanisms to allocate memory at the *page* granularity
- Interfaces in `include/linux/gfp.h`

Buddy system

- Prevent memory from being fragmented



Status of Buddy System

```
$> cat /proc/buddyinfo
```

Node 0, zone	DMA	1	0	0	1	2	1	1	0	1	1
Node 0, zone	DMA32	9	7	8	9	7	11	8	7	8	9
Node 0, zone	Normal	18184	5454	2414	2628	1562	727	254	721	999	451

Page allocation / deallocation

```
/**  
 * Allocate 2^{order} *physically* contiguous pages  
 * Return the address of the first allocated `struct page'  
 */  
struct page *alloc_pages(gfp_t gfp_mask, unsigned int order);  
struct page *alloc_page(gfp_t gfp_mask);  
  
/**  
 * Deallocate 2^{order} *physically* contiguous pages  
 * Be careful to put the correct order otherwise corrupt the memory  
 */  
void __free_pages(struct page *page, unsigned int order);  
void __free_page(struct page *page);
```

Page access

```
/**  
 * Obtain the virtual address to the page frame  
 */  
void *page_address(struct page *page);  
  
/**  
 * Allocate and get the virtual address directly  
 */  
unsigned long __get_free_pages(gfp_t gfp_mask, unsigned int order);  
unsigned long __get_free_page(gfp_t gfp_mask);  
  
/**  
 * Free pages using their addresses  
 */  
void free_pages(unsigned long addr, unsigned int order);  
void free_page(unsigned long addr);
```

Allocate zeroed page

- By default, the page *data* is not cleared
- May leak information through the page allocation
- To prevent information leakage, allocate a zero-out page for user-space request
 - `unsigned long get_zeroed_page(gfp_t gfp_mask);`

gfp_t: get free page flags

- Specify options for memory allocation
 - Action modifier
 - How the memory should be allocated
 - Zone modifier
 - From which zone the memory should be allocated
 - Type flags
 - Combination of action and zone modifiers
 - Generally preferred compared to the direct use of action/zone
- Defined in `include/linux/gfp.h`

gfp_t : action modifiers

Flag	Description
__GFP_WAIT	Allocator may sleep
__GFP_HIGH	Allocator can access emergency pools
__GFP_IO	Allocator can start disk IO
__GFP_FS	Allocator can start filesystem IO
__GFP_NOWARN	Allocator does not print failure warnings
__GFP_REPEAT	Repeat the allocation if it fails
__GFP_NOFAIL	The allocation is guaranteed
__GFP_NORETRY	No retry on allocation failure

gfp_t: action modifiers

- Some action modifiers can be used together

```
struct page *p = alloc_page(__GFP_WAIT | __GFP_FS | __GFP_IO);
```

gfp_t: zone modifiers

Flag	Description
__GFP_DMA	Allocate only from ZONE_DMA
__GFP_DMA32	Allocate only from ZONE_DMA32
__GFP_HIGHMEM	Allocate from ZONE_HIGHMEM or ZONE_NORMAL

- If not specified, allocated from ZONE_NORMAL or ZONE_DMA (high preference to ZONE_NORMAL)

gfp_t : type flags

- GFP_ATOMIC : Allocate without sleeping
 - __GFP_HIGH
- GFP_NOWAIT : Same to GFP_ATOMIC but does not fall back to the emergency pools

gfp_t : type flags

- **GFP_NOIO** : Can block but does not initiate disk IO
 - Used in block layer code to avoid recursion
 - **__GFP_WAIT**
- **GFP_NOFS** : Can block and perform disk IO, but does not initiate filesystem operations
 - Used in filesystem code
 - **__GFP_WAIT | __GFP_IO**

gfp_t : type flags

- GFP_KERNEL : Default. Can sleep and perform IO
 - __GFP_WAIT | __GFP_IO | __GFP_FS
- GFP_USER : Normal allocation for user-space memory
- GFP_HIGHUSER : Normal allocation for user-space memory
 - GFP_USER | __GFP_HIGHMEM
- GFP_DMA : Allocate from ZONE_DMA

gfp_t: Cheat sheet

Context	Solution
Process context, can sleep	GFP_KERNEL
Process context, cannot sleep	GFP_ATOMIC
Interrupt handler	GFP_ATOMIC
Softirq, tasklet	GFP_ATOMIC
DMA-able, can sleep	GFP_DMA GFP_KERNEL
DMA-able, cannot sleep	GFP_DMA GFP_ATOMIC

Low-level memory allocation example

```
#include <linux/module.h>
#include <linux/kernel.h>
#include <linux/init.h>
#include <linux/gfp.h>

#define PRINT_PREF          "[LOWLEVEL]: "
#define PAGES_ORDER_REQUESTED 3
#define INTS_IN_PAGE        (PAGE_SIZE/sizeof(int))

unsigned long virt_addr;

static int __init my_mod_init(void)
{
    int *int_array;
    int i;

    printk(PRINT_PREF "Entering module.\n");

    virt_addr = __get_free_pages(GFP_KERNEL, PAGES_ORDER_REQUESTED);
    if(!virt_addr) {
        printk(PRINT_PREF "Error in allocation\n");
        return -1;
    }
}
```

Low-level memory allocation example

```
int_array = (int *)virt_addr;
for(i=0; i<INTS_IN_PAGE; i++)
    int_array[i] = i;

for(i=0; i<INTS_IN_PAGE; i++)
    printk(PRINT_PREF "array[%d] = %d\n", i, int_array[i]);

return 0;
}

static void __exit my_mod_exit(void)
{
    free_pages(virt_addr, PAGES_ORDER_REQUESTED);
    printk(PRINT_PREF "Exiting module.\n");
}

module_init(my_mod_init);
module_exit(my_mod_exit);

MODULE_LICENSE("GPL");
```

High memory

- On x86_32, physical memory above 896 MB is not permanently mapped within the kernel address space
 - Due to the limited size of the address space and the 1/3 GB kernel/user-space memory split
- Before use, pages from highmem should be mapped to the address space

High memory

```
/**  
 * Permanent mappings  
 * - Maps the `page` and return the address to the `page`  
 * - May sleep  
 * - Has a limited number of slots  
 */  
void *kmap(struct page *page);  
void kunmap(struct page *page);  
  
/**  
 * Temporary mappings  
 * - Use a per-CPU pre-reserved mapping slots  
 * - Disable kernel preemption  
 * - Should not sleep while holding the mapping  
 */  
void *kmap_atomic(struct page *page);  
void kunmap_atomic(void *addr);
```

High memory

- Example

```
struct page *my_page;
void *my_addr;

my_page = alloc_page(GFP_HIGHUSER);
my_addr = kmap(my_page);

memcpy(my_addr, buffer, sizeof(buffer));

kunmap(my_page);
__free_page(my_page);
```

kmalloc() / kfree()

- `void *kmalloc(size_t size, gfp_t flags)`
 - Allocates byte-sized chunks of memory
 - Similar to the user-space `malloc()`
 - Returns a pointer to the first allocated byte on success
 - Returns NULL on error
 - Allocated memory is *physically contiguous*
- `void kfree(const void *ptr)`
 - Free the memory allocated with `kmalloc()`

kmalloc() / kfree()

- Example

```
struct my_struct *p;  
  
p = kmalloc(sizeof(*p), GFP_KERNEL);  
if (!p) {  
    /* Handle error */  
} else {  
    /* Do something */  
    kfree(p);  
}
```

vmalloc()

- `void *vmalloc(unsigned long size)`
 - Allocates *virtually contiguous* chunk of memory
 - May not be physically contiguous
 - Cannot be used for I/O buffers requiring physically contiguous memory
 - Used for allocating a large virtually contiguous memory chunk
 - May sleep so cannot be called from interrupt context
- Free using `vfree()`
 - `void vfree(const void *addr)`

vmalloc()

- However, most of the kernel uses `kmalloc()` for performance reasons
 - Pages allocated with `kmalloc()` are directly mapped
 - Less overhead in virtual to physical mapping setup and translation
- `vmalloc()` is still needed to allocate large portions of memory
- Declared in `include/linux/vmalloc.h`

vmalloc() vs. kmalloc()

```
#include <linux/module.h>
#include <linux/kernel.h>
#include <linux/init.h>
#include <linux/slab.h>

#define PRINT_PREF "[KMALLOC_TEST]: "

static int __init my_mod_init(void)
{
    unsigned long i;
    void *ptr;

    printk(PRINT_PREF "Entering module.\n");

    for(i=1;;i*=2) {
        ptr = kmalloc(i, GFP_KERNEL);
        if(!ptr) {
            printk(PRINT_PREF "could not allocate %lu bytes\n", i);
            break;
        }
        kfree(ptr);
    }
}
```

vmalloc() vs. kmalloc()

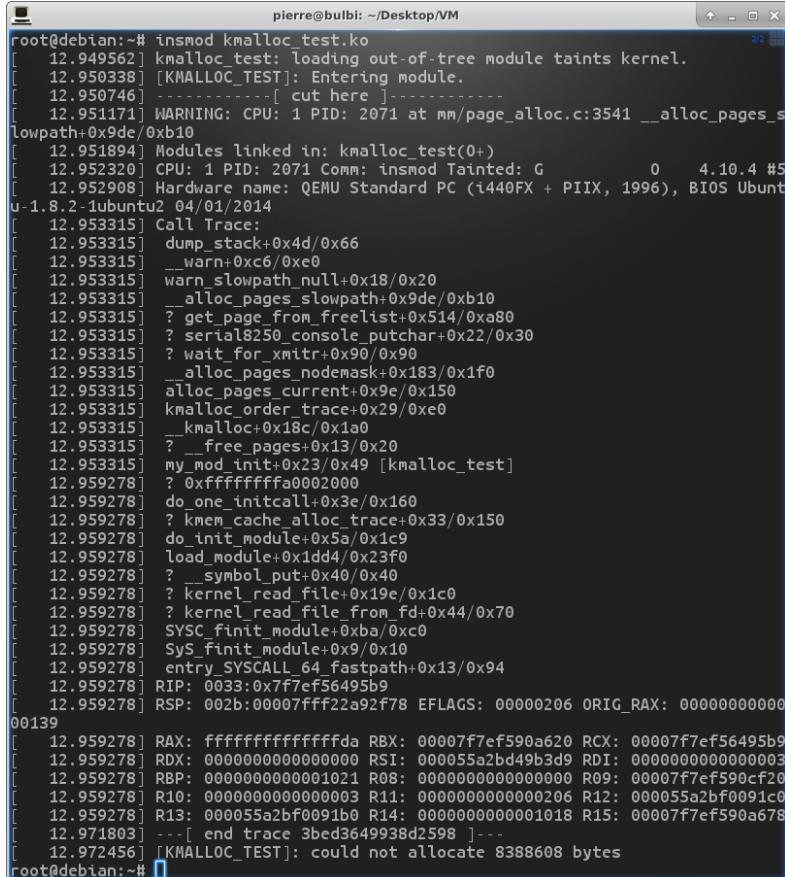
```
    return 0;
}

static void __exit my_mod_exit(void)
{
    printk(KERN_INFO "Exiting module.\n");
}

module_init(my_mod_init);
module_exit(my_mod_exit);

MODULE_LICENSE("GPL");
```

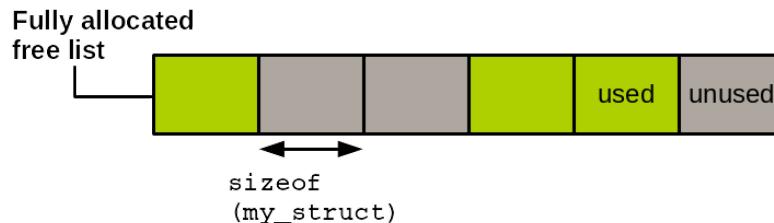
vmalloc() vs. kmalloc()



```
pierre@bulbi: ~/Desktop/VM
root@debian:~# insmod kmalloc_test.ko
12.949562] kmalloc_test: loading out-of-tree module taints kernel.
12.950338] [KMALLOC_TEST]: Entering module.
12.950746] -----[ cut here ]-----
12.951171] WARNING: CPU: 1 PID: 2071 at mm/page_alloc.c:3541 __alloc_pages_s
lowpath+0x9de/0xb10
12.951894] Modules linked in: kmalloc_test(0+)
12.952320] CPU: 1 PID: 2071 Comm: insmod Tainted: G      0   4.10.4 #5
12.952908] Hardware name: QEMU Standard PC (i440FX + PIIX, 1996), BIOS Ubuntu
1.8.2-1ubuntu2 04/01/2014
12.953315] Call Trace:
12.953315] dump_stack+0x4d/0x66
12.953315] __warn+0xc6/0xe0
12.953315] warn_slowpath_null+0x18/0x20
12.953315] __alloc_pages_slowpath+0x9de/0xb10
12.953315] ? get_page_from_freelist+0x514/0xa80
12.953315] ? serial8250_console_putchar+0x22/0x30
12.953315] ? wait_for_xmtr+0x90/0x90
12.953315] __alloc_pages_nodemask+0x183/0x1f0
12.953315] alloc_pages_current+0x9e/0x150
12.953315] kmalloc_order_trace+0x29/0xe0
12.953315] __kmalloc+0x18c/0x1a0
12.953315] ? __free_pages+0x13/0x20
12.953315] my_mod_init+0x23/0x49 [kmalloc_test]
12.959278] ? 0xfffffffffa0002000
12.959278] do_one_initcall+0x3e/0x160
12.959278] ? kmem_cache_alloc_trace+0x33/0x150
12.959278] do_init_module+0x5a/0x1c9
12.959278] load_module+0x1dd4/0x23f0
12.959278] ? __symbol_put+0x40/0x40
12.959278] ? kernel_read_file+0x19e/0x1c0
12.959278] ? kernel_read_file_from_fd+0x44/0x70
12.959278] SYS_finit_module+0xba/0xc0
12.959278] Sys_finit_module+0x9/0x10
12.959278] entry_SYSCALL_64_fastpath+0x13/0x94
12.959278] RIP: 0033:0x7f7ef56495b9
12.959278] RSP: 002b:00007fff22a92f78 EFLAGS: 000000206 ORIG_RAX: 000000000000
00139
12.959278] RAX: ffffffffffffffd8 RBX: 00007f7ef590a620 RCX: 00007f7ef56495b9
12.959278] RDX: 0000000000000000 RSI: 000055a2bd49b3d9 RDI: 0000000000000003
12.959278] RBP: 00000000000001021 R08: 0000000000000000 R09: 00007f7ef590cf20
12.959278] R10: 0000000000000003 R11: 0000000000000206 R12: 000055a2bf0091c0
12.959278] R13: 000055a2bf0091b0 R14: 00000000000001018 R15: 00007f7ef590a678
12.971803] ...[ end trace 3bed3649938d2598 ]...
12.972456] [KMALLOC_TEST]: could not allocate 8388608 bytes
root@debian:~#
```

Slab allocator

- Allocating/freeing data structures is done very often in the kernel
- **Q: how to make memory allocation faster? → caching using a free list**
- **Free lists:**
 - Block of pre-allocated memory for a given type of data structure
 - Allocate from the free list = pick an element in the free list
 - Deallocate an element = add an element to the free list



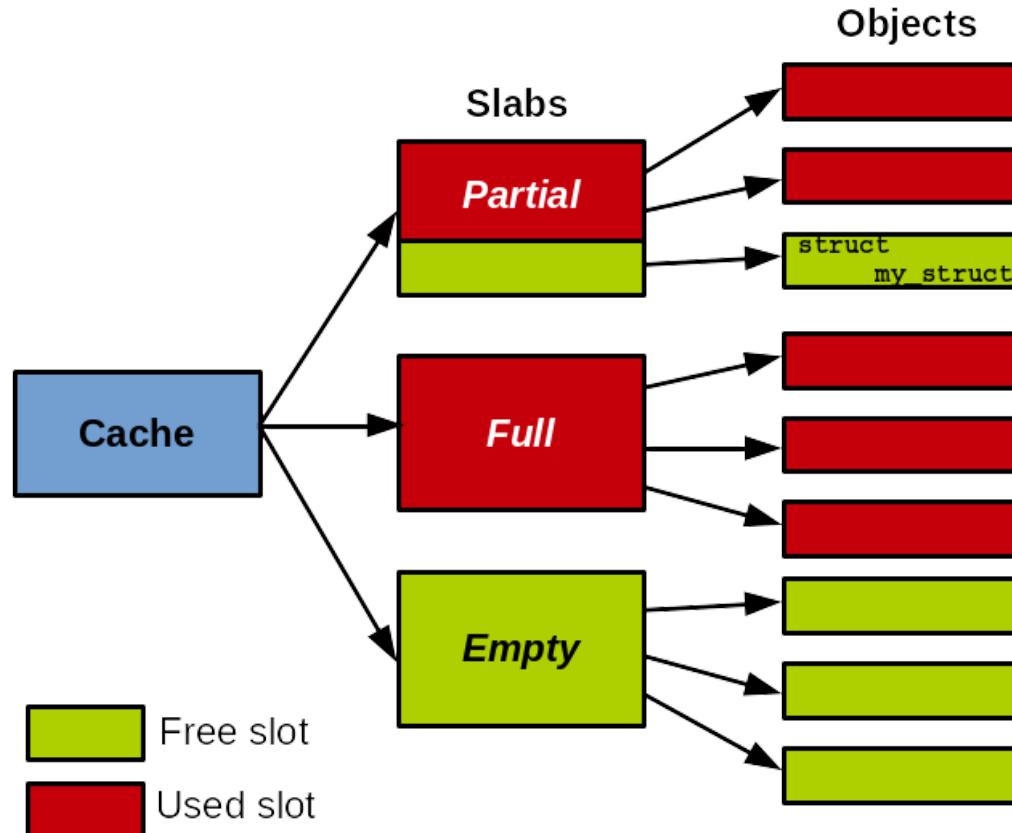
Slab allocator

- Issue with ad-hoc free lists: no global control
 - **When and how to free free lists?**
- **Slab allocator**
 - Generic allocation caching interface
 - Cache **objects** of a data structure type
 - E.g., an object cache for `struct task_struct`
 - Consider the data structure size, page size, NUMA, and cache coloring

Slab allocator

- A cache has one or more **slabs**
 - One or several physically contiguous pages
- **Slabs contain objects**
- A slab may be empty, partially full, or full
- Allocate objects from the partially full slabs to prevent memory fragmentation

Slab allocator

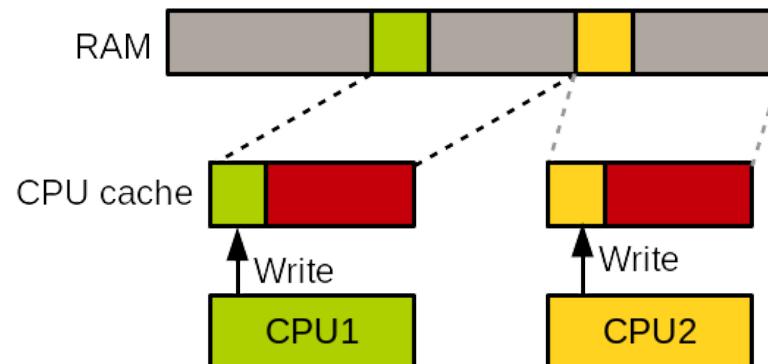


Slab allocator

```
/**  
 * Create a cache for a data structure type  
 */  
struct kmem_cache *kmem_cache_create(  
    const char *name,      /* Name of the cache */  
    size_t size,          /* Size of objects */  
    size_t align,         /* Offset of the first element  
                           within pages */  
    unsigned long flags, /* Options */  
    void (*ctor)(void *) /* Constructor */  
);  
  
/**  
 * Destroy the cache  
 * - Should be only called when all slabs in the cache are empty  
 * - Should not access the cache during the destruction  
 */  
void kmem_cache_destroy(struct kmem_cache *cachep);
```

Slab allocator

- **SLAB_HW_CACHEALIGN**
 - Align objects to the cache line to prevent false sharing
 - Increase memory footprint



Slab allocator

- SLAB_POISON
 - Initially fill slabs with a known value(0xa5a5a5a5) to detect accesses to uninitialized memory
- SLAB_RED_ZONE
 - Put extra padding around objects to detect overflows
- SLAB_PANIC
 - Panic if allocation fails
- SLAB_CACHE_DMA
 - Allocate from DMA-enabled memory

Slab allocator

```
/**  
 * Allocate an object from the cache  
 */  
void *kmem_cache_alloc(struct kmem_cache *cachep, gfp_t flags);  
  
/**  
 * Free an object allocated from a cache  
 */  
void kmem_cache_free(struct kmem_cache *cachep, void *objp);
```

Slab allocator example

```
#include <linux/module.h>
#include <linux/kernel.h>
#include <linux/init.h>
#include <linux/slab.h>

#define PRINT_PREF "[SLAB_TEST] "

struct my_struct {
    int int_param;
    long long_param;
};

static int __init my_mod_init(void)
{
    int ret = 0;
    struct my_struct *ptr1, *ptr2;
    struct kmem_cache *my_cache;

    printk(PRINT_PREF "Entering module.\n");

    my_cache = kmem_cache_create("lkp-cache", sizeof(struct my_struct),
        0, 0, NULL);
    if(!my_cache)
        return -1;
```

Slab allocator example

```
ptr1 = kmem_cache_alloc(my_cache, GFP_KERNEL);
if(!ptr1){
    ret = -ENOMEM;
    goto destroy_cache;
}

ptr2 = kmem_cache_alloc(my_cache, GFP_KERNEL);
if(!ptr2){
    ret = -ENOMEM;
    goto freeptr1;
}

ptr1->int_param = 42;
ptr1->long_param = 42;
ptr2->int_param = 43;
ptr2->long_param = 43;

printf(PRINT_PREF "ptr1 = {%-d, %ld} ; ptr2 = {%-d, %ld}\n", ptr1->int_param,
       ptr1->long_param, ptr2->int_param, ptr2->long_param);

kmem_cache_free(my_cache, ptr2);
```

Slab allocator example

```
freeptr1:  
    kmem_cache_free(my_cache, ptr1);  
  
destroy_cache:  
    kmem_cache_destroy(my_cache);  
  
    return ret;  
}  
  
static void __exit my_mod_exit(void)  
{  
    printk(KERN_INFO "Exiting module.\n");  
}  
  
module_init(my_mod_init);  
module_exit(my_mod_exit);  
  
MODULE_LICENSE("GPL");
```

Status of Slab allocator

```
$> sudo cat /proc/slabinfo
slabinfo - version: 2.1
# name          <active_objs> <num_objs> <objsize> <objperslab> <pagesperslab> : tunables <l
nf_conntrack      575      675      320      25      2 : tunables      0      0      0 : slabdata      27
rpc_inode_cache    46       46       704      46      8 : tunables      0      0      0 : slabdata      1
fat_inode_cache   133      176      744      44      8 : tunables      0      0      0 : slabdata      4
fat_cache          0       0       40      102      1 : tunables      0      0      0 : slabdata      0
squashfs_inode_cache 368      368      704      46      8 : tunables      0      0      0 : slabdata      :
kvm_async_pf        0       0      136      30      1 : tunables      0      0      0 : slabdata      0
kvm_vcpu            0       0     15104      2       8 : tunables      0      0      0 : slabdata      0
kvm_mmu_page_header 0       0      168      24      1 : tunables      0      0      0 : slabdata      0
x86_emulator         0       0      2672      12       8 : tunables      0      0      0 : slabdata      0
x86_fpu              0       0      4160       7       8 : tunables      0      0      0 : slabdata      0
ext4_groupinfo_4k   3724     3724      144      28      1 : tunables      0      0      0 : slabdata      133
i915_dependency      512      512      128      32      1 : tunables      0      0      0 : slabdata      16
execute_cb            0       0      128      32      1 : tunables      0      0      0 : slabdata      0
i915_request         1964     1988      576      28      4 : tunables      0      0      0 : slabdata      71
intel_context         630      630      384      42      4 : tunables      0      0      0 : slabdata      15
fuse_inode            156      156      832      39      8 : tunables      0      0      0 : slabdata      4
btrfs_delayed_node    0       0      312      26      2 : tunables      0      0      0 : slabdata      0
btrfs_ordered_extent   0       0      416      39      4 : tunables      0      0      0 : slabdata      0
btrfs_free_space_bitmap 0       0     12288      2       8 : tunables      0      0      0 : slabdata      0
btrfs_inode            0       0     1184      27      8 : tunables      0      0      0 : slabdata      0
fsverity_info          0       0      256      32      2 : tunables      0      0      0 : slabdata      0
```

Slab allocator variants

- **SLOB (Simple List Of Blocks)**
 - Used in early Linux version (from 1991)
 - Low memory footprint, suitable for embedded systems
- **SLAB**
 - Integrated in 1999
 - Cache-friendly
- **SLUB**
 - Integrated in 2008
 - Improved scalability over SLAB on many cores

Per-CPU data structure

- Allow each core to have their own values
 - No locking required
 - Reduce cache thrashing
- Implemented through arrays in which each index corresponds to a CPU

```
unsigned long my_percpu[NR_CPUS];
int cpu;
cpu = get_cpu();      /* get_cpu() disables kernel preemption
                      * otherwise `cpu` might become incorrect
                      * across context switches */
my_percpu[cpu]++;
put_cpu();           /* put_cpu() enables kernel preemption */
```

Per-CPU API

- Defined in `include/linux/percpu.h`

```
DEFINE_PER_CPU(type, name);
DECLARE_PER_CPU(name, type);

get_cpu_var(name); /* Start accessing the per-CPU variable */
put_cpu_var(name); /* Done accessing the per-CPU variable */

/* Access per-CPU data through pointers */
get_cpu_ptr(name);
put_cpu_ptr(name);

per_cpu(name, cpu); /* Access other CPU's data */
```

Per-CPU data structure

- Example

```
DEFINE_PER_CPU(int, my_var);

int cpu;
for (cpu = 0; cpu < NR_CPUS; cpu++)
    per_cpu(my_var, cpu) = 0;

printf("%d\n", get_cpu_var(my_var)++);
put_cpu_var(my_var);

int *my_var_ptr;
my_var_ptr = get_cpu_ptr(my_var);
put_cpu_ptr(my_var_ptr);
```

Stack

- Each process has
 - A user-space stack for execution
 - A kernel stack for in-kernel execution
- **User-space stack** is large and grows dynamically
- **Kernel-stack** is small and has a fixed-size → two pages (= 8 KB)
- **Interrupt stack** is for interrupt handlers → one page for each CPU
- Reduce kernel stack usage to a minimum
 - Local variables and function parameters

Take-away

- Need physically contiguous memory
 - `kmalloc()` or `alloc_page()` series
- Virtually contiguous chunk
 - `vmalloc()`
- Frequently creating/destroying large amount of the same data structures
 - Use the slab allocator
- Need to allocate from high memory
 - Use `alloc_page()` then `kmap()`/`kmap_atomic()`

Further readings

- [Virtual Memory: 3 What is Virtual Memory?](#)
- [Complete virtual memory map x86_64 architecture](#)

Next Lecture

- Process Address Space

Process Address Space

Dongyoon Lee

Summary of last lectures

- Tools: building, exploring, and debugging Linux kernel
- Core kernel infrastructure
 - syscall, module, kernel data structures
- Process management & scheduling
- Interrupt & interrupt handler
- Kernel synchronization
- Memory management

Today: process address space

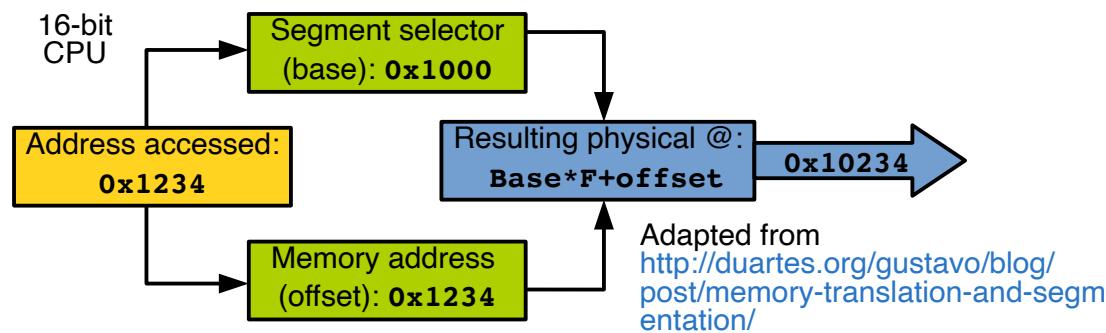
- Address space
- Memory descriptor: `mm_struct`
- Virtual Memory Area (VMA)
- VMA manipulation
- Page tables

Address space

- The memory that a process can access
 - Illusion that the process can access 100% of the system memory
 - With virtual memory, can be much larger than the actual amount of physical memory
- Defined by the process page table set up by the kernel

Address space

- A memory address is an index within the address spaces:
 - Identify a specific byte
- Each process is given a flat 32/64-bits address space
 - Not segmented



Address space

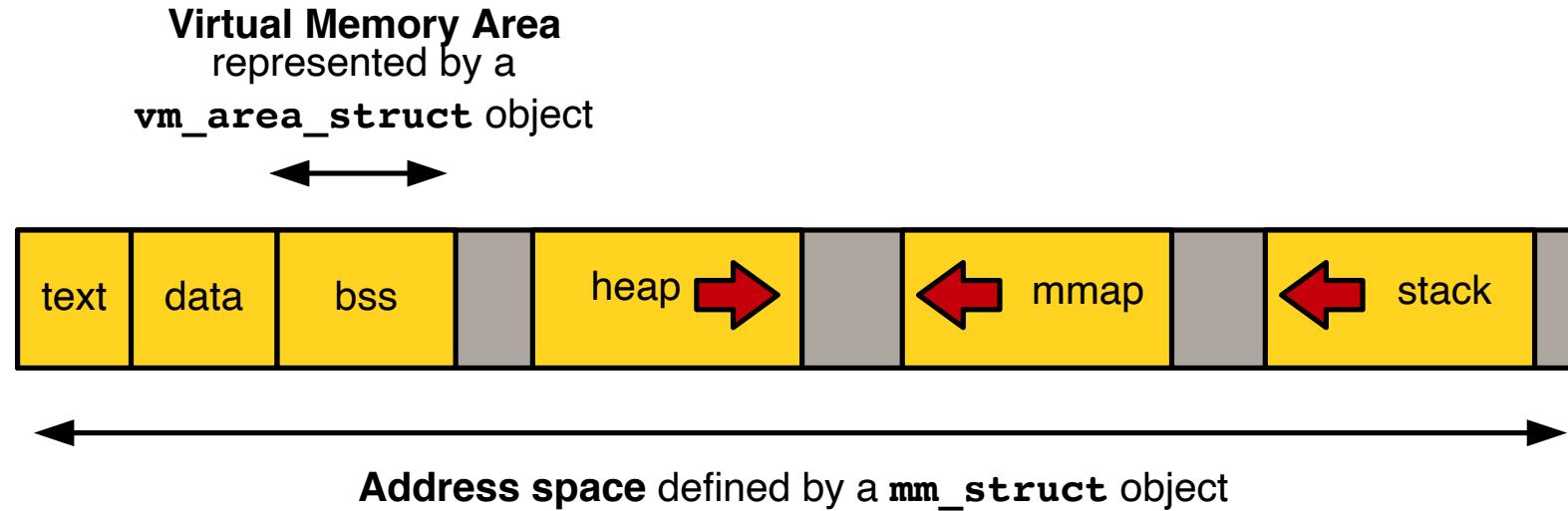
- **Virtual Memory Areas (VMA)**
 - Interval of addresses that the process has the right to access
 - Can be dynamically added or removed to the process address space
 - Associated permissions: read, write, execute
 - *Illegal access → segmentation fault*

```
$ cat /proc/1/maps      # or sudo pmap 1
55fe3bf02000-55fe3bff9000 r-xp 00000000 fd:00 1975429 /usr/lib/systemd/systemd
55fe3bffa000-55fe3c021000 r--p 000f7000 fd:00 1975429 /usr/lib/systemd/systemd
55fe3c021000-55fe3c022000 rw-p 0011e000 fd:00 1975429 /usr/lib/systemd/systemd
55fe3db4a000-55fe3ddfd000 rw-p 00000000 00:00 0 [heap]
7f7522769000-7f7522fd9000 rw-p 00000000 00:00 0
7f7523150000-7f7523265000 r-xp 00000000 fd:00 1979800 /usr/lib64/libm-2.25.so
7f7523265000-7f7523464000 ---p 00115000 fd:00 1979800 /usr/lib64/libm-2.25.so
7f7523464000-7f7523465000 r--p 00114000 fd:00 1979800 /usr/lib64/libm-2.25.so
7f7523465000-7f7523466000 rw-p 00115000 fd:00 1979800 /usr/lib64/libm-2.25.so
```

Address space

- VMAs can contain:
 - Mapping of the executable file code (*text section*)
 - Mapping of the executable file initialized variables (*data section*)
 - Mapping of the zero page for uninitialized variables (*bss section*)
 - Mapping of the zero page for the *user-space stack*
 - Text, data, bss for each *shared library* used
 - Memory-mapped files, shared memory segment, anonymous mappings (used by malloc)

Address space



Memory descriptor: `mm_struct`

- Address space in linux kernel: `struct mm_struct`

```
/* linux/include/linux/mm_types.h */

struct mm_struct {
    struct vm_area_struct *mmap;           /* list of VMAs */
    struct rb_root        mm_rb;           /* rbtree of VMAs */
    pgd_t                *pgd;             /* page global directory */
    atomic_t              mm_users;         /* address space users */
    atomic_t              mm_count;         /* primary usage counters */
    int                  map_count;        /* number of VMAs */
    struct rw_semaphore  mmap_sem;         /* VMA semaphore */
    spinlock_t            page_table_lock; /* page table lock */
    struct list_head      mmlist;           /* list of all mm_struct */
    unsigned long          start_code;       /* start address of code */
    unsigned long          end_code;         /* end address of code */
    unsigned long          start_data;      /* start address of data */
    unsigned long          end_data;         /* end address of data */
    unsigned long          start_brk;        /* start address of heap */
    unsigned long          end_brk;          /* end address of heap */
    unsigned long          start_stack;     /* start address of stack */
    /* ... */
```

Memory descriptor: `mm_struct`

```
unsigned long          arg_start;    /* start of arguments */
unsigned long          arg_end;      /* end of arguments */
unsigned long          env_start;    /* start of environment */
unsigned long          total_vm;     /* total pages mapped */
unsigned long          locked_vm;   /* number of locked pages */
unsigned long          flags;        /* architecture specific data */
spinlock_t             ioctx_lock;   /* Asynchronous I/O list lock */
/* ... */
};
```

- `mm_users` : number of processes (threads) using the address space
- `mm_count` : reference count:
 - +1 if `mm_users` > 0
 - +1 if the kernel is using the address space
 - When `mm_count` reaches 0, the `mm_struct` can be freed

Memory descriptor: `mm_struct`

- `mmap` and `mm_rb` are respectively a linked list and a tree containing all the VMAs in this address space
 - List used to iterate over all the VMAs in an ascending order
 - Tree used to find a specific VMA
- All `mm_struct` are linked together in a doubly linked list
 - Through the `mmlist` field if the `mm_struct`

Allocating a memory descriptor

- A task memory descriptor is located in the `mm` field of the corresponding

`task_struct`

```
/* linux/include/linux/sched.h */

struct task_struct {
    struct thread_info          thread_info;
    /* ... */
    const struct sched_class     *sched_class;
    struct sched_entity          se;
    struct sched_rt_entity       rt;
    /* ... */
    struct mm_struct             *mm;
    struct mm_struct             *active_mm;
    /* ... */
};
```

Allocating a memory descriptor

- Current task memory descriptor: `current->mm`
- During `fork()`, `copy_mm()` is making a copy of the parent memory descriptor for the child
 - `copy_mm()` calls `dup_mm()` which calls `allocate_mm()` → allocates a `mm` struct object from a slab cache
- Two threads sharing the same address space have the `mm` field of their `task_struct` pointing to the same `mm_struct` object
 - Threads are created using the `CLONE_VM` flag passed to `clone()` → `allocate_mm()` is not called

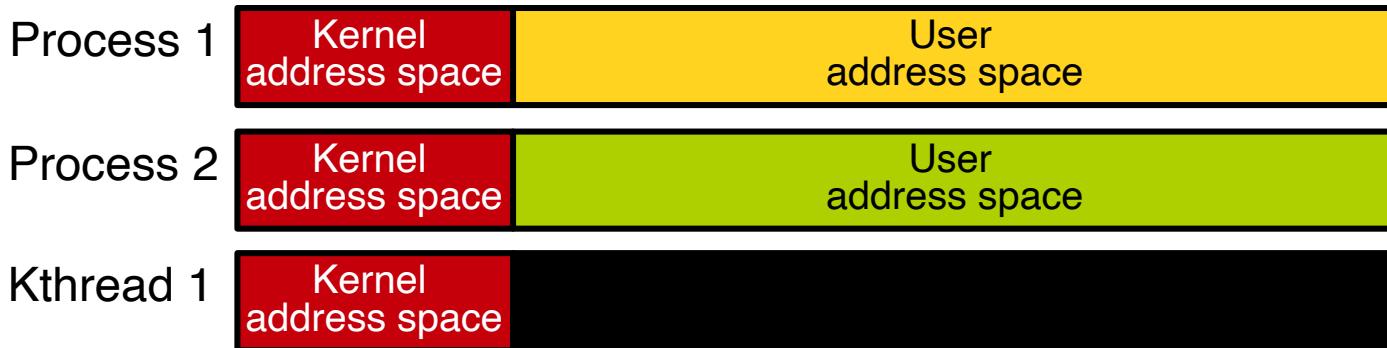
Destroying a memory descriptor

- When a process exits, `do_exit()` is called and it calls `exit_mm()`
 - Performs some housekeeping/statistics updates and calls `mmpput()`

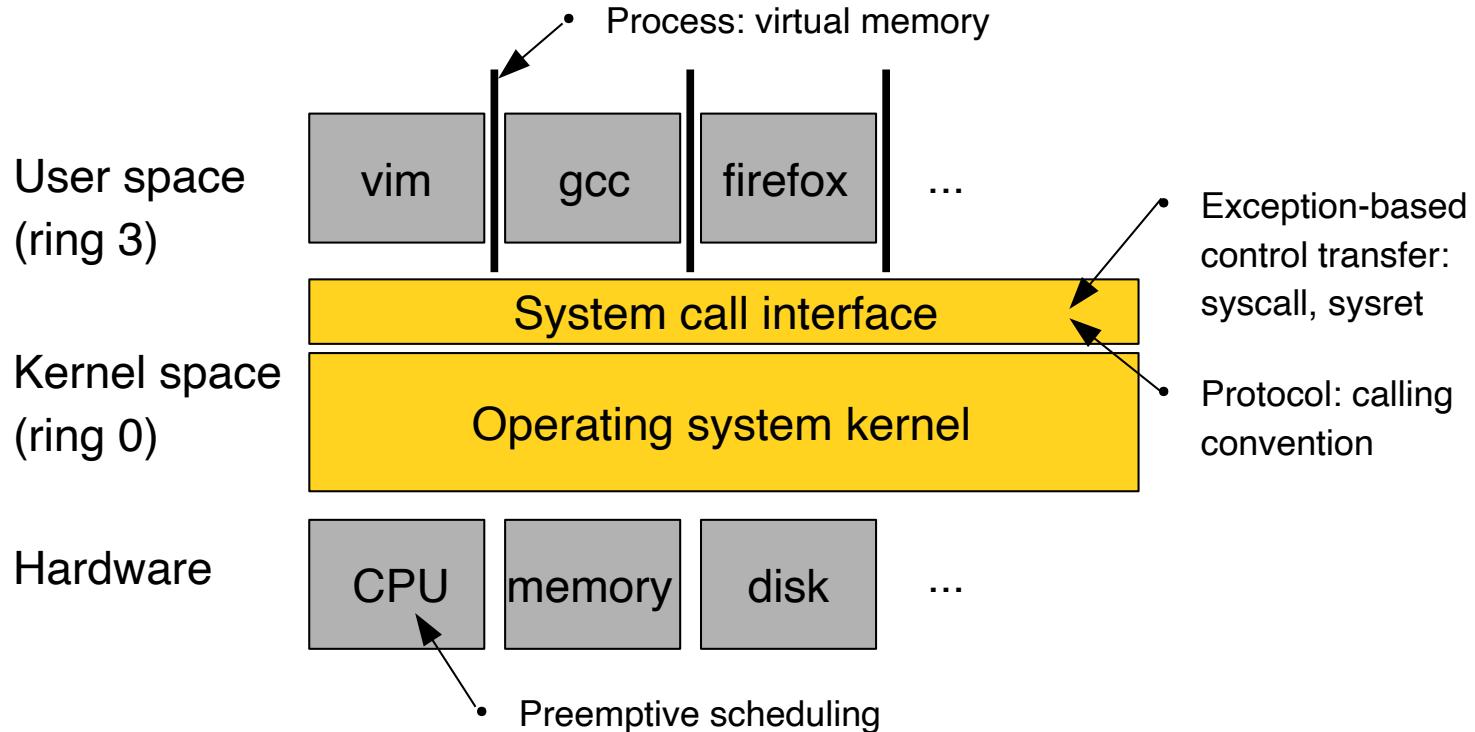
```
void mmpput(struct mm_struct *mm) {
    might_sleep();
    if (atomic_dec_and_test(&mm->mm_users))
        __mmpput(mm);
}
static inline void __mmpput(struct mm_struct *mm) {
    /* ... */
    mmdrop(mm);
}
static inline void mmdrop(struct mm_struct *mm) {
    if (unlikely(atomic_dec_and_test(&mm->mm_count)))
        __mmdrop(mm);
}
void __mmdrop(struct mm_struct *mm) {
    /* ... */
    free_mm(mm);
}
```

The `mm_struct` and kernel threads

- Kernel threads do not have a user-space address space
 - `mm` field of a kernel thread `task_struct` is `NULL`



The `mm_struct` and kernel threads



The `mm_struct` and kernel threads

- However kernel threads still need to access the kernel address space
 - When a kernel thread is scheduled, the kernel notice its `mm` is `NULL` so it keeps the previous address space loaded (page tables)
 - Kernel makes the `active_mm` field of the kernel thread to point on the borrowed `mm_struct`
 - It is okay because the kernel address space is the same in all tasks

Review: Segmentation in x86

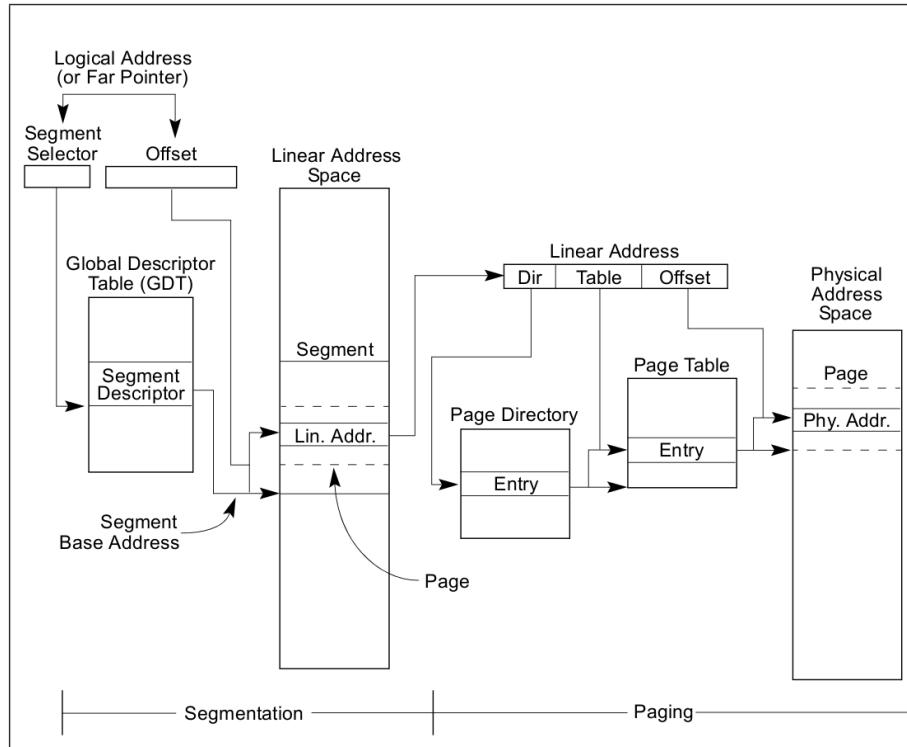


Figure 3-1. Segmentation and Paging

Review: Privilege levels of a segment

- CPL (current privilege level)
 - the privilege level of currently executing program
 - bits 0 and 1 in the `%cs` register
- RPL (requested privilege level)
 - an override privilege level that is assigned to a segment selector
 - a segment selector is a part (16-bit) of segment registers (e.g., `ds` ,
`fs`), which is an index of a segment descriptor and RPL
- DPL (descriptor privilege level)
 - the privilege level of a segment

Review: How isolation is enforced in x86?

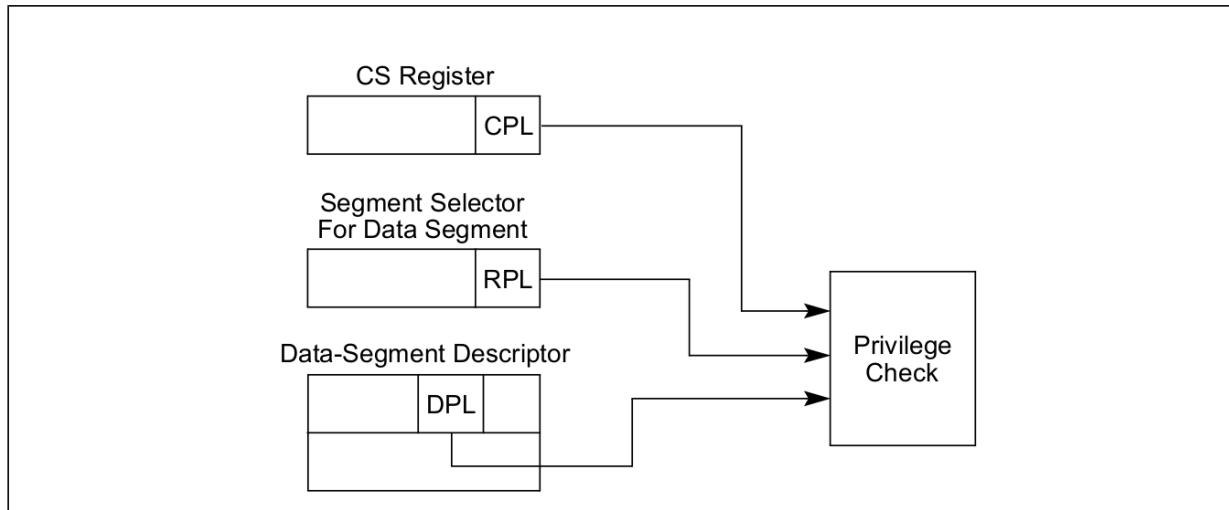


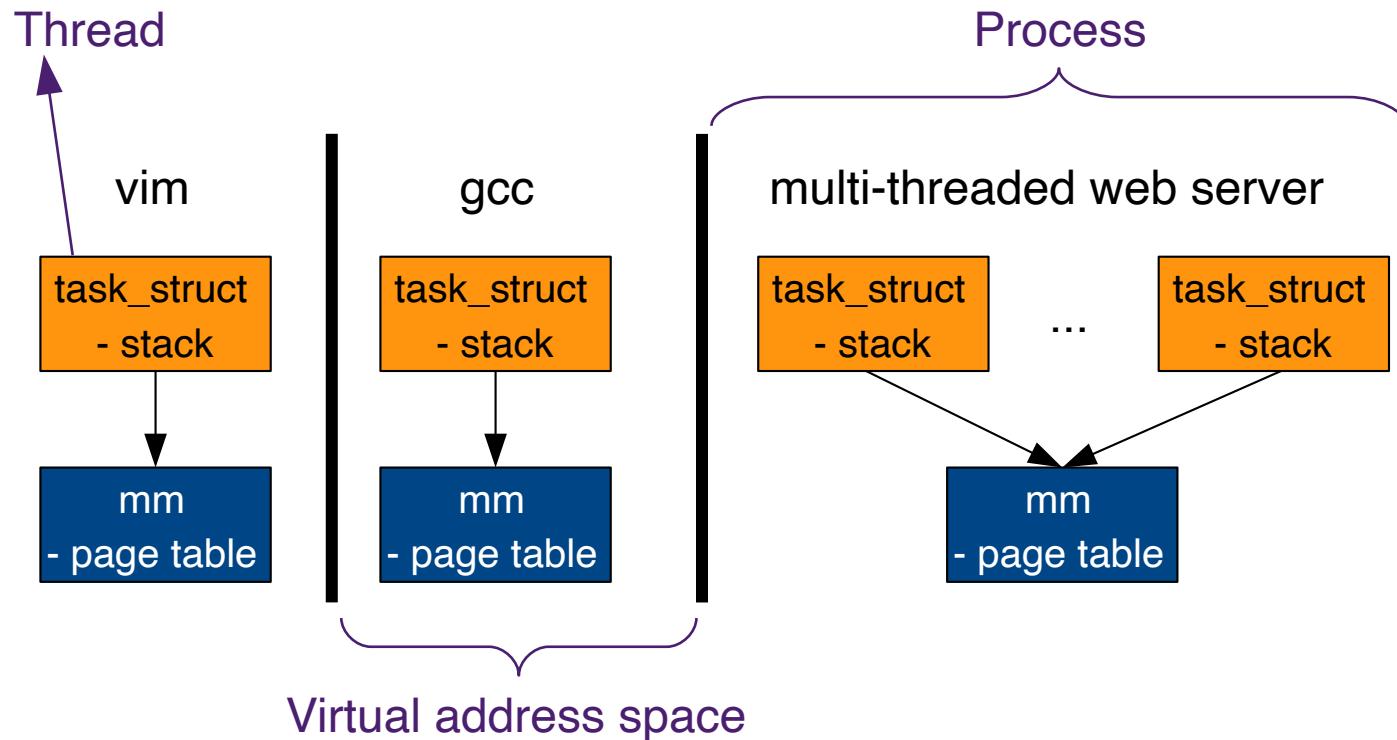
Figure 5-4. Privilege Check for Data Access

- Access is granted if $RPL \geq CPL$ and $DPL \geq CPL$

Review: How to switch b/w rings (ring 0 ↔ ring 3)?

- Controlled transfer: system call
 - `int`, `sysenter` or `syscall` instruction set CPL to 0; change to KERNEL_CS and KERNEL_DS segments
 - set CPL to 3 before going back to user space; change to USER_CS and USER_DS segments

mm_struct vs. task_struct



Virtual Memory Area (VMA)

- Each line corresponds to one VMA

```
$ cat /proc/1/maps # or sudo pmap 1
55fe3bf02000-55fe3bff9000 r-xp 00000000 fd:00 1975429 /usr/lib/systemd/systemd
55fe3bffa000-55fe3c021000 r--p 000f7000 fd:00 1975429 /usr/lib/systemd/systemd
55fe3c021000-55fe3c022000 rw-p 0011e000 fd:00 1975429 /usr/lib/systemd/systemd
55fe3db4a000-55fe3ddfd000 rw-p 00000000 00:00 0 [heap]
7f7522769000-7f7522fd9000 rw-p 00000000 00:00 0
7f7523150000-7f7523265000 r-xp 00000000 fd:00 1979800 /usr/lib64/libm-2.25.so
7f7523265000-7f7523464000 ---p 00115000 fd:00 1979800 /usr/lib64/libm-2.25.so
7f7523464000-7f7523465000 r--p 00114000 fd:00 1979800 /usr/lib64/libm-2.25.so
7f7523465000-7f7523466000 rw-p 00115000 fd:00 1979800 /usr/lib64/libm-2.25.so

# r = read
# w = write
# x = execute
# s = shared
# p = private (copy on write)
```

Virtual Memory Area (VMA)

- Each VMA is represented by an object of type `vm_area_struct`

```
/* linux/include/linux/mm_types.h */

struct vm_area_struct {
    struct mm_struct *vm_mm; /* associated address space (mm_struct) */
    unsigned long vm_start; /* VMA start, inclusive */
    unsigned long vm_end; /* VMA end, exclusive */
    struct vm_area_struct *vm_next; /* list of VMAs */
    struct vm_area_struct *vm_prev; /* list of VMAs */
    pgprot_t vm_page_prot; /* access permissions */
    unsigned long vm_flags; /* flags */
    struct rb_node vm_rb; /* VMA node in the tree */
    struct list_head anon_vma_chain; /* list of anonymous mappings */
    struct anon_vma *anon_vma; /* anonymous vma object */
    struct vm_operations_struct *vm_ops; /* operations */
    unsigned long vm_pgoff; /* offset within file */
    struct file *vm_file; /* mapped file (can be NULL) */
    void *vm_private_data; /* private data */
} /* ... */
```

Virtual Memory Area (VMA)

- The VMA exists over `[vm_start, vm_end)` in the corresponding address space → size in bytes: `vm_end - vm_start`
- Address space is pointed by the `vm_mm` field (of type `mm_struct`)
- Each VMA is unique to the associated `mm_struct`
 - Two processes mapping the same file will have two different `mm_struct` objects, and two different `vm_area_struct` objects
 - Two threads sharing a `mm_struct` object also share the `vm_area_struct` objects

VMA flags

Flag	Effect on the VMA and Its Pages
VM_READ	Pages can be read from.
VM_WRITE	Pages can be written to.
VM_EXEC	Pages can be executed.
VM_SHARED	Pages are shared.
VM_MAYREAD	The VM_READ flag can be set.
VM_MAYWRITE	The VM_WRITE flag can be set.
VM_MAYEXEC	The VM_EXEC flag can be set.
VM_MAYSHARE	The VM_SHARE flag can be set.

VMA flags

Flag	Effect on the VMA and Its Pages
VM_GROWSDOWN	The area can grow downward.
VM_GROWSUP	The area can grow upward.
VM_SHM	The area is used for shared memory.
VM_DENYWRITE	The area maps an unwritable file.
VM_EXECUTABLE	The area maps an executable file.
VM_LOCKED	The pages in this area are locked.
VM_IO	The area maps a device's I/O space.
VM_SEQ_READ	The pages seem to be accessed sequentially.

VMA flags

Flag	Effect on the VMA and Its Pages
VM_RAND_READ	The pages seem to be accessed randomly.
VM_DONTCOPY	This area must not be copied on fork().
VM_DONTEXPAND	This area cannot grow via mremap().
VM_RESERVED	This area must not be swapped out.
VM_ACCOUNT	This area is an accounted VM object.
VM_HUGETLB	This area uses hugetlb pages.
VM_NONLINEAR	This area is a nonlinear mapping.

VMA flags

- Combining `VM_READ`, `VM_WRITE` and `VM_EXEC` gives the permissions for the entire area, for example:
 - Object code is `VM_READ` and `VM_EXEC`
 - Stack is `VM_READ` and `VM_WRITE`
- `VM_SEQ_READ` and `VM_RAND_READ` are set through the `madvise()` system call
 - Instructs the file pre-fetching algorithm read-ahead to increase or decrease its pre-fetch window

VMA flags

- `VM_HUGETLB` indicates that the area uses pages larger than the regular size
 - 2M and 1G on x86
 - Larger page size → less TLB miss → faster memory access

VMA operations

- `vm_ops` in `vm_area_struct` is a struct of function pointers to operate on a specific VMA

```
/* linux/include/linux/mm.h */
struct vm_operations_struct {
    /* called when the area is added to an address space */
    void (*open)(struct vm_area_struct * area);

    /* called when the area is removed from an address space */
    void (*close)(struct vm_area_struct * area);

    /* invoked by the page fault handler when a page that is
     * not present in physical memory is accessed*/
    int (*fault)(struct vm_area_struct *vma, struct vm_fault *vmf);

    /* invoked by the page fault handler when a previously read-only
     * page is made writable */
    int (*page_mkwrite)(struct vm_area_struct *vma, struct vm_fault *vmf);
    /* ... */
}
```

VMA manipulation: `find_vma()`

```
/* linux/mm/mmap.c */

/* Look up the first VMA which satisfies addr < vm_end, NULL if none. */
struct vm_area_struct *find_vma(struct mm_struct *mm, unsigned long addr)
{
    struct rb_node *rb_node;
    struct vm_area_struct *vma;

    /* Check the cache first. */
    vma = vmacache_find(mm, addr);
    if (likely(vma))
        return vma;

    rb_node = mm->mm_rb.rb_node;

    while (rb_node) {
        struct vm_area_struct *tmp;

        tmp = rb_entry(rb_node, struct vm_area_struct, vm_rb);
```

VMA manipulation: `find_vma()`

```
if (tmp->vm_end > addr) {  
    vma = tmp;  
    if (tmp->vm_start <= addr)  
        break;  
    rb_node = rb_node->rb_left;  
} else  
    rb_node = rb_node->rb_right;  
}  
  
if (vma)  
    vmacache_update(addr, vma);  
return vma;  
}
```

VMA manipulation

```
/* linux/include/linux/mm.h */

/* Look up the first VMA which satisfies addr < vm_end, NULL if none. */
struct vm_area_struct *find_vma(struct mm_struct *mm, unsigned long addr);

/*
 * Same as find_vma, but also return a pointer to the previous VMA in *pprev.
 */
struct vm_area_struct *
find_vma_prev(struct mm_struct *mm, unsigned long addr,
             struct vm_area_struct **pprev);

/* Look up the first VMA which intersects the interval start_addr..end_addr-1,
   NULL if none. Assume start_addr < end_addr. */
struct vm_area_struct * find_vma_intersection(struct mm_struct * mm,
                                              unsigned long start_addr, unsigned long end_addr);
```

Creating an address interval

- `do_mmap()` is used to create a new linear address interval:
 - Can result in the creation of a new VMAs
 - Or a merge of the create area with an adjacent one when they have the same permissions

```
/*
 * The caller must hold down_write(&current->mm->mmap_sem).
 */
unsigned long do_mmap(struct file *file, unsigned long addr,
                      unsigned long len, unsigned long prot,
                      unsigned long flags, vm_flags_t vm_flags,
                      unsigned long pgoff, unsigned long *populate,
                      struct list_head *uf);
```

Creating an address interval

- `prot` specifies access permissions for the memory pages

Flag	Effect on the new interval
<code>PROT_READ</code>	Corresponds to <code>VM_READ</code>
<code>PROT_WRITE</code>	Corresponds to <code>VM_WRITE</code>
<code>PROT_EXEC</code>	Corresponds to <code>VM_EXEC</code>
<code>PROT_NONE</code>	Cannot access page

Creating an address interval

- `flags` specifies the rest of the VMA options

Flag	Effect on the new interval
<code>MAP_SHARED</code>	The mapping can be shared.
<code>MAP_PRIVATE</code>	The mapping cannot be shared.
<code>MAP_FIXED</code>	The new interval must start at the given address addr.
<code>MAP_ANONYMOUS</code>	The mapping is not file-backed, but is anonymous.
<code>MAP_GROWSDOWN</code>	Corresponds to <code>VM_GROWSDOWN</code> .

Creating an address interval

- `flags` specifies the rest of the VMA options

Flag	Effect on the new interval
<code>MAP_DENYWRITE</code>	Corresponds to <code>VM_DENYWRITE</code> .
<code>MAP_EXECUTABLE</code>	Corresponds to <code>VM_EXECUTABLE</code> .
<code>MAP_LOCKED</code>	Corresponds to <code>VM_LOCKED</code> .
<code>MAP_NORESERVE</code>	No need to reserve space for the mapping.
<code>MAP_POPULATE</code>	Populate (prefault) page tables.
<code>MAP_NONBLOCK</code>	Do not block on I/O.

Creating an address interval

- On error `do_mmap()` returns a negative value
- On success
 - The kernel tries to merge the new interval with an adjacent one having same permissions
 - Otherwise, create a new VMA
 - Returns a pointer to the start of the mapped memory area
- `do_mmap()` is exported to user-space through `mmap2()`

```
void *mmap2(void *addr, size_t length, int prot,  
            int flags, int fd, off_t pgoffset);
```

Removing an address interval

- Removing an address interval is done through do_munmap()

```
/* linux/include/linux/mm.h */  
int do_munmap(struct mm_struct *, unsigned long, size_t);
```

- Exported to user-space through munmap()

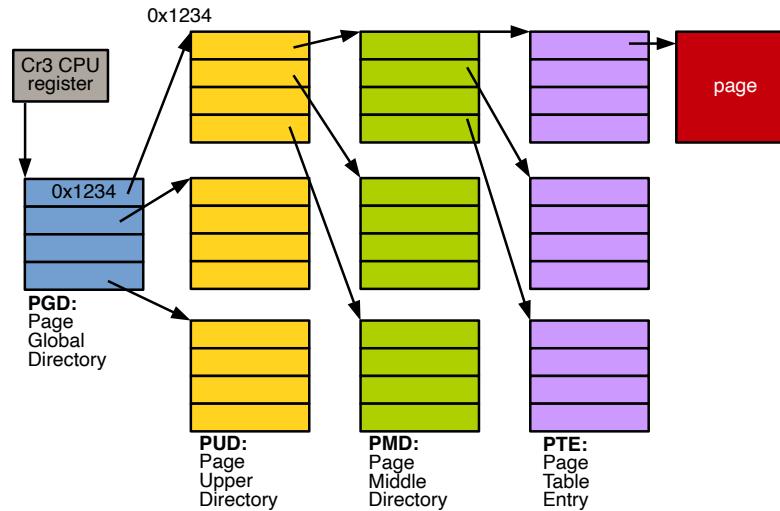
```
int munmap(void *addr, size_t len);
```

Page tables

- Linux enables paging early in the boot process
 - All memory accesses made by the CPU are virtual and translated to physical addresses through the page tables
 - Linux sets the page tables and the translation is made automatically by the hardware (MMU) according to the page tables content
- The address space is defined by VMAs and is sparsely populated
 - One address space per process → one page table per process
 - Lots of “empty” areas

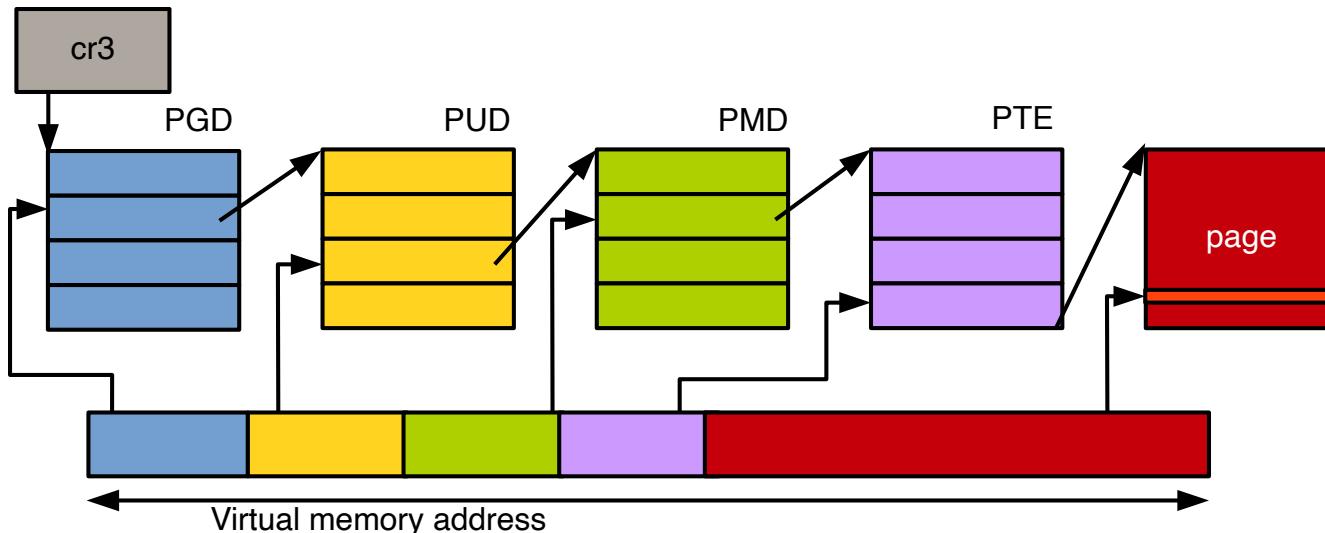
Page tables

```
/* linux/include/linux/mm types.h */  
struct mm_struct {  
    struct vm_area_struct *mmap;           /* list of VMAs */  
    struct rb_root          mm_rb;         /* rbtree of VMAs */  
    pgd_t                  *pgd;           /* page global directory */  
    /* ... */  
};
```



Page tables

- Address translation is performed by the hardware (MMU)



Virtual address map in linux

```
0000000000000000 - 00007ffffffffffff (=47 bits) user space, different per mm
hole caused by [47:63] sign extension
ffff800000000000 - ffff87ffffffffffff (=43 bits) guard hole, reserved for hypervisor
ffff880000000000 - ffffc7ffffffffffff (=64 TB) **direct mapping of all phys. memory**
ffffc80000000000 - ffffc8ffffffffffff (=40 bits) hole
ffffc90000000000 - fffffe8ffffffffffff (=45 bits) vmalloc/ioremap space
fffffe90000000000 - fffffe9ffffffffffff (=40 bits) hole
fffffea0000000000 - ffffffea0000000000 (=40 bits) virtual memory map (1TB)
... unused hole ...
fffffec0000000000 - ffffffbfffffff (=44 bits) kasan shadow memory (16TB)
... unused hole ...
                    vaddr_end for KASLR
fffffe0000000000 - ffffffe7ffffffffffff (=39 bits) cpu_entry_area mapping
fffffe8000000000 - ffffffeffffffffff (=39 bits) LDT remap for PTI
ffffff0000000000 - ffffff7ffffffffffff (=39 bits) %esp fixup stacks
... unused hole ...
fffffffef00000000 - ffffffeffffffffff (=64 GB)   EFI region mapping space
... unused hole ...
ffffffffff80000000 - ffffffff9ffffffffffff (=512 MB) kernel text mapping, from phys 0
fffffffffffa000000 - ffffffffffffeffffffff (1520 MB) module mapping space
[fixmap start] - fffffffffff5fffff                 kernel-internal fixmap range
ffffffffff600000 - fffffffffff600ffff (=4 kB)    legacy vsyscall ABI
fffffffffffe00000 - ffffffffffffefffff (=2 MB)   unused hole
```

Further readings

- [Introduction to Memory Management in Linux](#)
- [20 years of Linux virtual memory](#)
- [Linux Kernel Virtual Memory Map](#)
- [Kernel page-table isolation](#)
- [Addressing Meltdown and Spectre in the kernel](#)
- [Meltdown and Spectre](#)
- [Meltdown Attack Lab](#)

Further readings

- Supporting bigger and heterogeneous memory efficiently
 - [AutoNUMA](#), [Transparent Hugepage Support](#), [Five-level page tables](#)
 - [Heterogeneous memory management](#)
- Optimization for virtualization
 - [Kernel same-page merging \(KSM\)](#)
 - [MMU notifier](#)

Next class

- Virtual File System

Virtual File System

Dongyoon Lee

Summary of last lectures

- Tools: building, exploring, and debugging Linux kernel
- Core kernel infrastructure
 - syscall, module, kernel data structures
- Process management & scheduling
- Interrupt & interrupt handler
- Kernel synchronization
- Memory management & address space

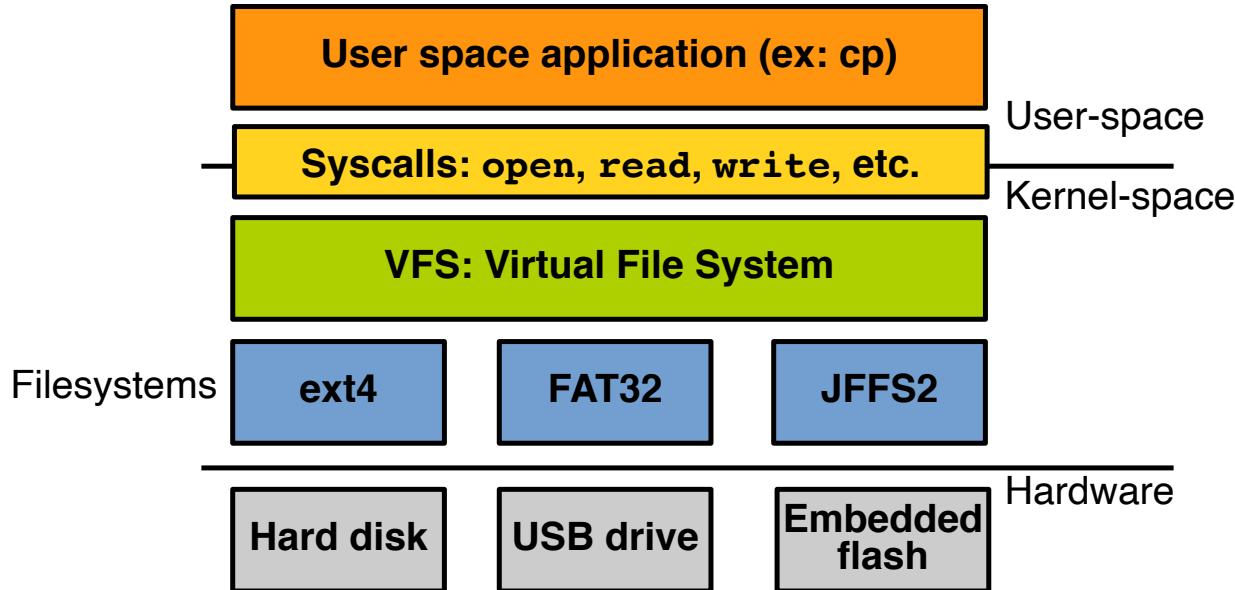
Today: virtual file system

- Introduction
- VFS data structures
- Filesystem data structures
- Process data structures

The Virtual File System (VFS)

- Abstract all the filesystem models supported by Linux
 - Similar to an abstract base class in C++
- Allow them to *coexist*
 - Example: a user can have a USB drive formatted with FAT32 mounted at the same time as a HDD rootfs with ext4
- Allow them to *cooperate*
 - Example: a user can seamlessly copy a file between the FAT32 and Ext4 partitions

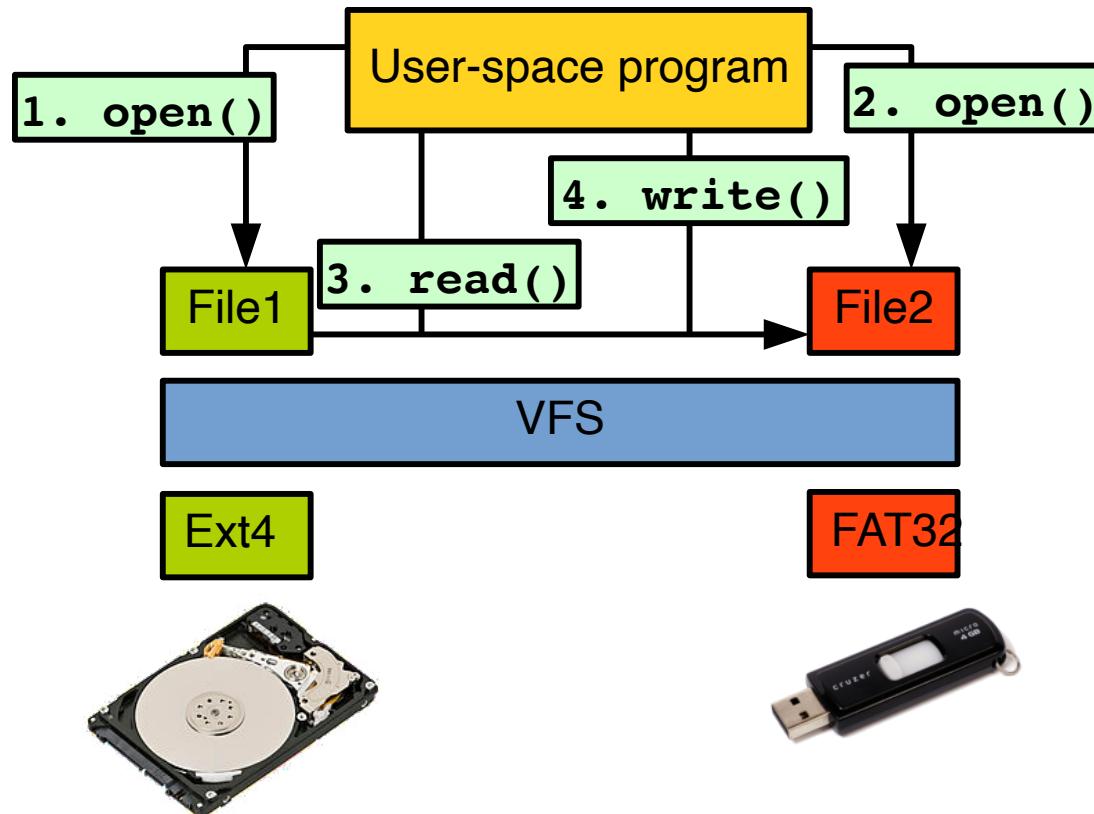
The Virtual File System (VFS)



Common filesystem interface

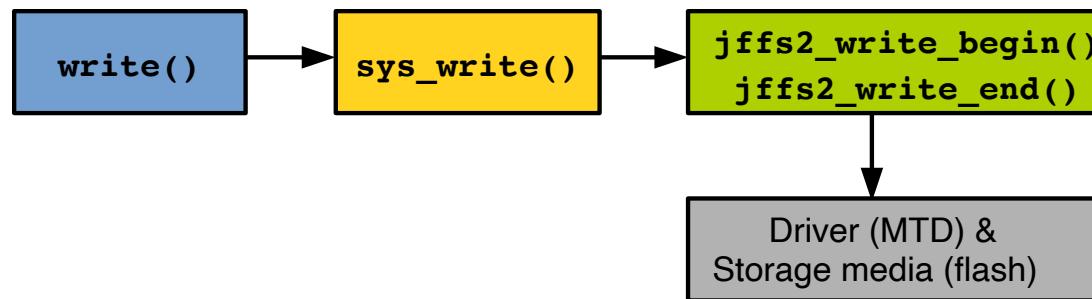
- VFS allows user-space to access files *independently* of the concrete filesystem they are stored on with a *common interface*
 - Standard system calls: `open()`, `read()`, `write()`, `lseek()`, etc.
 - “top” VFS interface (with user-space)
- Interface can work transparently between filesystems

Common filesystem interface



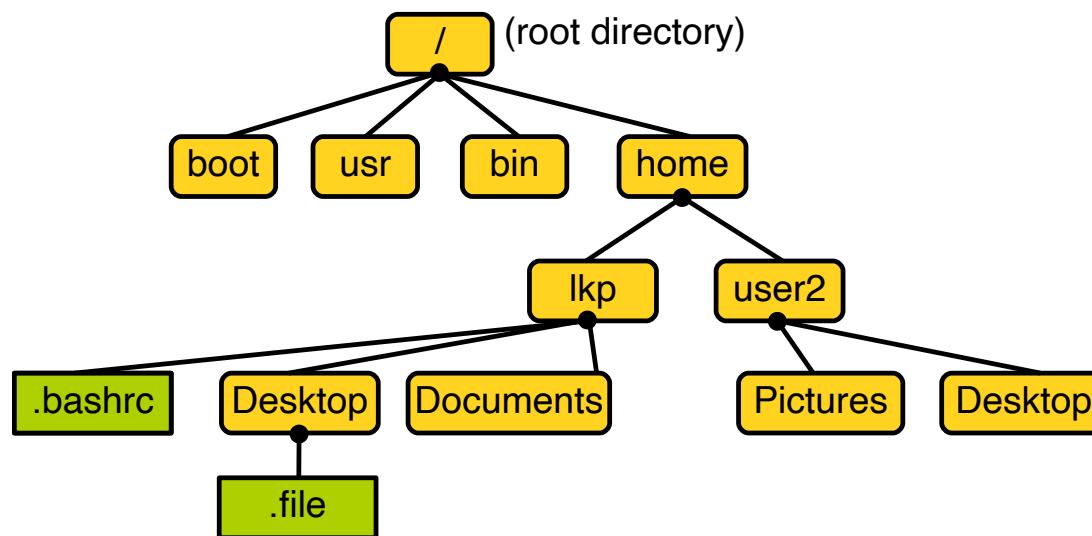
Filesystem abstraction layer

- VFS redirects user-space requests to the corresponding concrete filesystem
 - “bottom” VFS interface (with the filesystem)
 - Developing a new filesystem for Linux means *conforming* with the bottom interface



Unix filesystems

- The term *filesystem* can refer to a filesystem type or a partition
- Hierarchical tree of *files* organized into *directories*



Unix filesystems

- **File:** ordered string of bytes from file address 0 to address (file size -1)
 - Metadata: name, access permissions, modification date, etc.
 - Separated from the file data into specific objects *inodes, dentries*

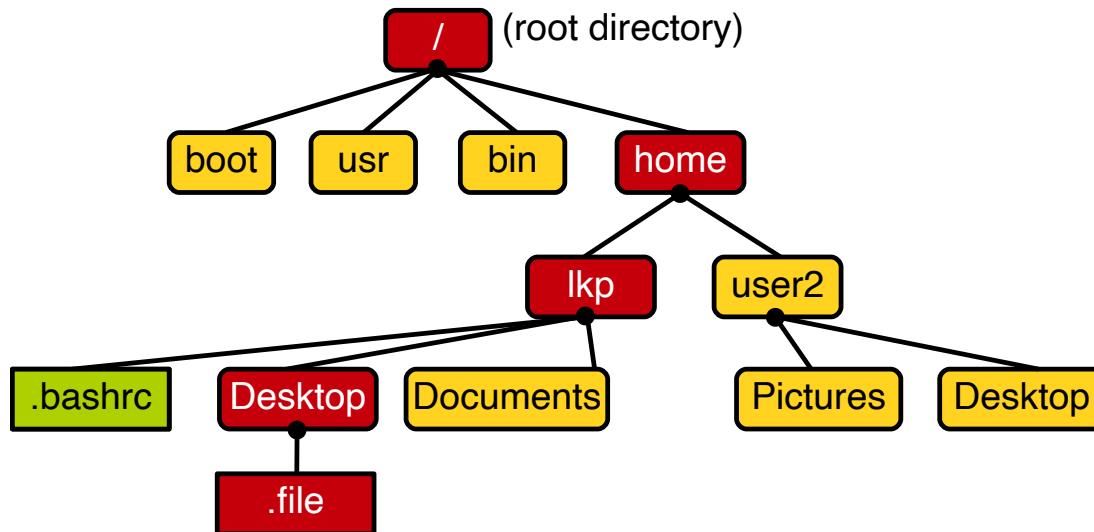


- **Directory:** folder containing files or other directories (sub-directories)
 - Sub-directories can be nested to create path:

/home/lkp/Desktop/file

Unix filesystems

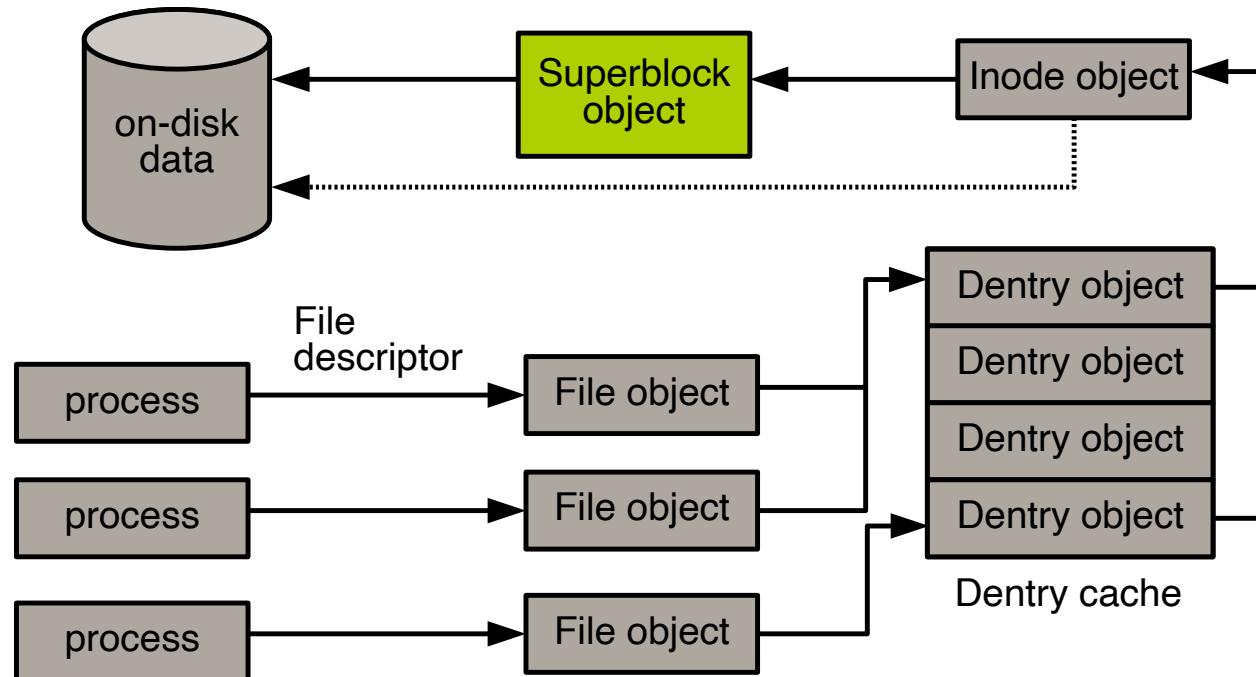
- Path example: /home/lkp/Desktop/.file



VFS data structures

- **dentry**: contains file/directory name and hierarchical links defining the filesystem directory tree
- **inode**: contains file/directory metadata
- **file**: contains information about a file opened by a process
- **superblock**: contains general information about the partition
- **file_system_type**: contains information about a file system type (ext4)
- Associated **operations** (“bottom” VFS interface):
 - `super_operations` , `inode_operations` ,
`dentry_operations` , `file_operations`

Superblock



Superblock

- Contains global information about the filesystem (partition)
- Created by the filesystem and given to VFS at mount time:
 - Disk-based filesystem store it in a special location
 - Other filesystems have a way to generate it at mount time
- `struct super_block` defined in `include/linux/fs.h`

```
/* linux/include/linux/fs.h */
struct super_block {
    struct list_head s_list;          /** list of all superblocks **/
    dev_t             s_dev;           /* identifier */
    unsigned long     s_blocksize;      /* block size (bytes) */
    unsigned long     s_blocksizes_bits; /* block size (bits) */
    loff_t            s_maxbytes;      /* max file size */
    /* ... */
```

Superblock

```
/* ... */
struct file_system_type      *s_type;           /** filesystem type */
struct super_operations       *s_op;             /** superblock operations */
struct dquot_operations       *dq_op;            /* quota methods */
struct quotactl_ops           *s_qcop;           /* quota control methods */
unsigned long                  s_flags;           /** mount flags */
unsigned long                  s_magic;           /* filesystem magic number */
struct dentry                  s_root;            /** directory mount point */
struct rw_semaphore             s_umount;          /* umount semaphore */
int                           s_count;           /* superblock reference count */
atomic_t                      s_active;          /* active reference count */
struct xattr_handler           **s_xattr;          /* extended attributes handler */
struct list_head                 s_inodes;          /** inodes list */
struct hlist_bl_head            s_anon;            /* anonymous entries */
struct list_lru                  s_dentry_lru;        /* list of unused dentries */
struct block_device              *s_bdev;            /** associated block device */
struct hlist_node                 s_instances;        /* instances of this filesystem */
struct quota_info                  s_dquot;           /* quota-specific options */
char                            s_id[32];          /* text name */
void                           *s_fs_info;         /* filesystem-specific info */
fmode_t                         s_mode;            /** mount permissions */
};

};
```

Superblock operations

- `struct super_operations`
 - Each field is a function pointer operating on a `struct super_block`
 - Usage: `sb->s_op->alloc_inode(sb)`

```
/* linux/include/linux/fs.h */
struct super_operations {
    struct inode *(*alloc_inode)(struct super_block *sb);
    void (*destroy_inode)(struct inode *);
    void (*dirty_inode) (struct inode *, int flags);
    int (*write_inode) (struct inode *, struct writeback_control *wbc);
    int (*drop_inode) (struct inode *);
    void (*evict_inode) (struct inode *);
    void (*put_super) (struct super_block *);
    int (*sync_fs)(struct super_block *sb, int wait);
    /* ... */
}
```

Superblock operations: **inode**

- `struct inode * alloc_inode(struct super_block *sb)`
 - Creates and initialize a new inode
- `void destroy_inode(struct inode *inode)`
 - Deallocate an inode
- `void dirty_inode(struct inode *inode)`
 - Marks an inode as dirty (Ext filesystems)

Superblock operations: `inode`

- `void write_inode(struct inode *inode, int wait)`
 - Writes the inode to disk, `wait` specifies if the write should be synchronous
- `void clear_inode(struct inode *inode)`
 - Releases the inode and clear any page containing related data
- `void drop_inode(struct inode *inode)`
 - Called by VFS when the last reference to the inode is dropped

Superblock operations: **superblock**

- `void put_super(struct super_block *sb)`
 - Called by VFS on unmount (holding `s_lock`)
- `void write_super(struct super_block *sb)`
 - Update the on-disk superblock, caller must hold `s_lock`

Superblock operations: **filesystem**

- `int sync_fs(struct super_block *sb, int wait)`
 - Synchronize filesystem metadata with on-disk filesystem, `wait` specifies if the operation should be synchronous
- `void write_super_lockfs(struct super_block *sb)`
 - Prevents changes to the filesystem and update the on-disk superblock (used by the Logical Volume Manager)
- `void unlockfs(struct super_block *sb)}`
 - Unlocks the filesystem locked by `write_super_lockfs()`

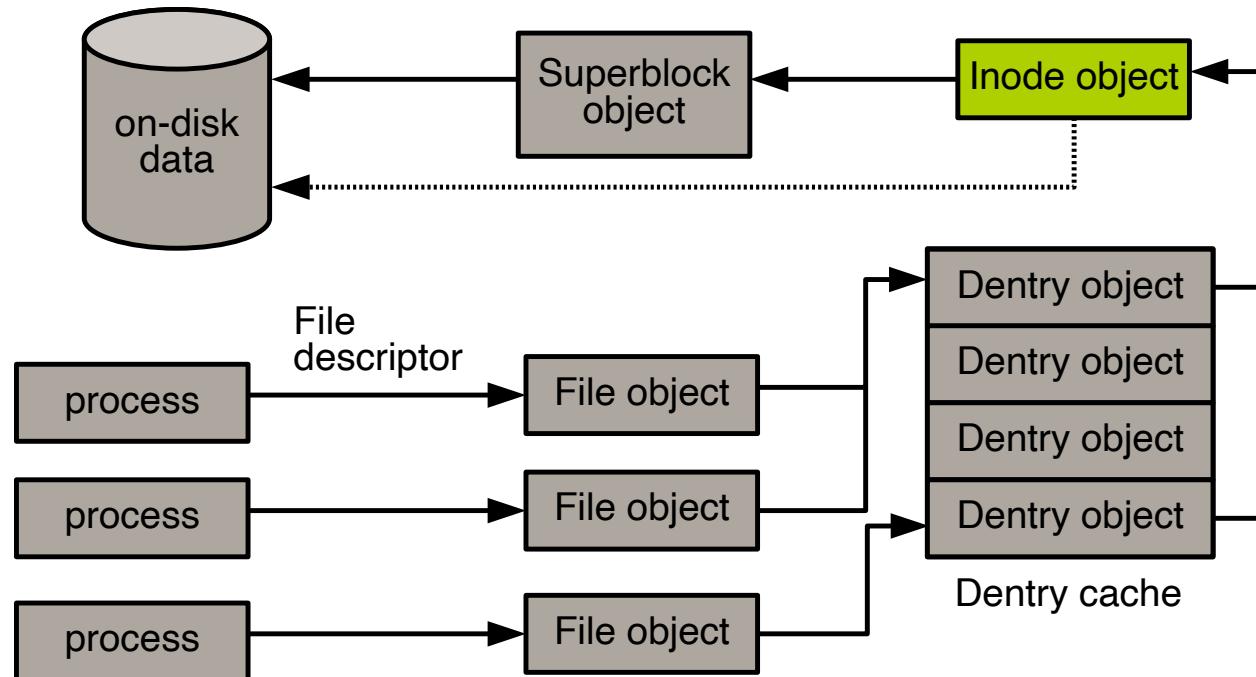
Superblock operations: **filesystem**

- `int statfs(struct super_block *sb, struct statfs *statfs)`
 - Obtain filesystem statistics
- `int remount_fs(struct super_block *sb, int *flags, char *data)`
 - Remount the filesystem with new options, caller must hold `s_lock`

Superblock operations: **filesystem**

- `void umount_begin(struct super_block *sb)`
 - Called by VFS to interrupt a mount operation (NFS)
- All of these functions are called by VFS and may block (except
`dirty_inode()`)
- **Q: where is the function to mount a file system?**
 - `mount_bdev()` in `fs/super.c`

inode



inode

- Related to a file or directory, contains metadata plus information about how to manipulate the file/directory
- Metadata: file size, owner id/group, etc
- Must be produced by the filesystem on-demand when a file/directory is accessed:
 - Read from disk in Unix-like filesystem
 - Reconstructed from on-disk information for other filesystems

inode

```
/* linux/include/linux/fs.h */
struct inode {
    struct hlist_node          i_hash;           /** hash list */
    struct list_head            i_lru;            /* inode LRU list*/
    struct list_head            i_sb_list;        /** inode list in superblock */
    struct list_head            i_dentry;         /** list of dentries */
    unsigned long                i_ino;             /** inode number */
    atomic_t                     i_count;           /** reference counter */
    unsigned int                  i_nlink;           /* number of hard links */
    uid_t                         i_uid;              /** user id of owner */
    gid_t                         i_gid;              /** group id of owner */
    kdev_t                        i_rdev;             /* real device node */
    u64                           i_version;         /* versioning number */
    loff_t                        i_size;             /* file size in bytes */
    seqcount_t                    i_size_seqcount; /* seqlock for i_size */
    i_atime;                      /* last access time */
    i_mtime;                      /* last modify time (file content) */
    i_ctime;                      /* last change time (contents or attributes) */
    unsigned int                  i_blkbits;        /* block size in bits */
    /* ... */
```

inode

```
/* ... */
const struct inode_operations *i_op;      /** inode operations */
struct super_block    *i_sb;            /** associated superblock */
struct address_space   *i_mapping;       /** associated page cache */
unsigned long          i_dnotify_mask;    /* directory notify mask */
struct dnotify_struct *i_dnotify;       /* dnotify */
struct list_head        iotify_watches;  /* inotify watches */
struct mutex             iotify_mutex;    /* protects inotify_watches */
unsigned long           i_state;         /* state flags */
unsigned long           dirtied_when;    /* first dirtying time */
unsigned int            i_flags;          /* filesystem flags */
atomic_t                i_writecount;    /* count of writers */
void *                  i_private;       /* filesystem private data */
/* ... */
};
```

inode operations

```
struct inode_operations {  
    int (*create) (struct inode *, struct dentry *, umode_t, bool);  
    int (*link) (struct dentry *, struct inode *, struct dentry *);  
    int (*unlink) (struct inode *, struct dentry *);  
    int (*symlink) (struct inode *, struct dentry *, const char *);  
    int (*mkdir) (struct inode *, struct dentry *, umode_t);  
    int (*rmdir) (struct inode *, struct dentry *);  
    int (*mknod) (struct inode *, struct dentry *, umode_t, dev_t);  
    int (*rename) (struct inode *, struct dentry *,  
                  struct inode *, struct dentry *, unsigned int);  
    /* ... */  
};
```

inode operations

- `int create(struct inode *dir, struct dentry *dentry, int mode)`
 - Create a new inode with access mode `mode`
 - Called from `creat()` and `open()` syscalls
 - **Q: how does it return a new inode?**
- `struct dentry * lookup(struct inode *dir, struct dentry *dentry)`
 - Searches a directory (`inode`) for a file/directory (`dentry`)

inode operations

- `int link(struct dentry *old_dentry, struct inode *dir, struct dentry *dentry)`
 - Creates a hard link with name `dentry` in the directory `dir`, pointing to `old_dentry`
- `int unlink(struct inode *dir, struct dentry *dentry)`
 - Remove an inode (`dentry`) from the directory `dir`

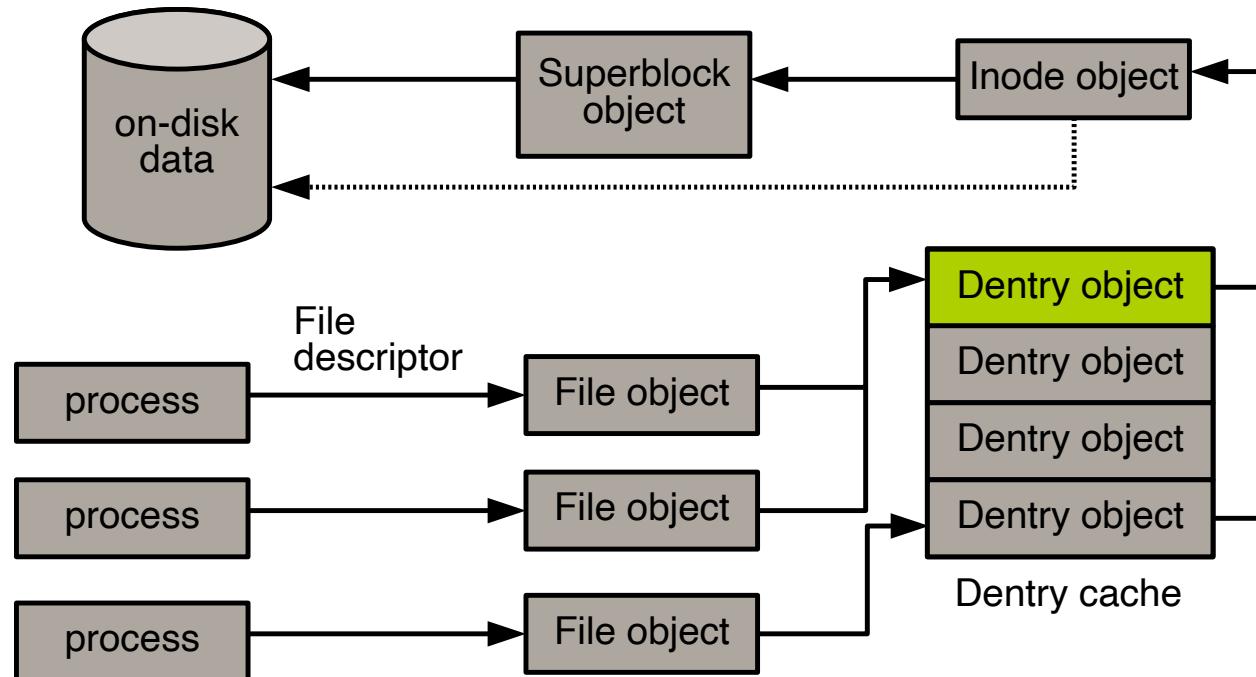
inode operations

- `int symlink(struct inode *dir, struct dentry *dentry, const char *symname)`
 - Creates a symbolic link named `symname`, to the file `dentry` in directory `dir`
- `int mkdir(struct inode *dir, struct dentry *dentry, int mode)`
 - Creates a directory inside `dir` with name
- `int rmdir(struct inode *dir, struct dentry *dentry)`
 - Removes a directory `dentry` from `dir`

inode operations

- `int mknod(struct inode *dir, struct dentry *dentry,
int mode, dev_t rev)`
 - Creates a special file (device file, pipe, socket)
- `int rename(struct struct inode *old_dir, struct
dentry *old_dentry, struct inode *new_dir, struct
dentry *new_dentry)`
 - Moves a file

dentry (or directory entry)



dentry

```
struct dentry {  
    atomic_t          d_count;    /* usage count */  
    unsigned int      d_flags;    /* dentry flags */  
    spinlock_t        d_lock;     /* per-dentry lock */  
    int               d_mounted;   /* indicate if it is a mount point */  
    struct inode      *d_inode;    /** associated inode **/  
    struct hlist_node d_hash;    /** list of hash table entries **/  
    struct dentry     *d_parent;   /** parent dentry **/  
    struct qstr        d_name;     /* dentry name */  
    struct list_head   d_lru;      /* unused list */  
    struct list_head   d_subdirs;   /** sub-directories **/  
    struct list_head   d_alias;    /** list of dentries  
                                    ** pointing to the same inode **/  
    unsigned long       d_time;     /* last time validity was checked */  
    struct dentry_operations *d_op;    /** operations **/  
    struct super_block  *d_sb;      /** superblock **/  
    void               *d_fsd data;  /* filesystem private data */  
    unsigned char       d_iname[DNAME_INLINE_LEN_MIN]; /* short name */  
/* ... */  
};
```

dentry

- Associated with a file or a directory to:
 - Store the file/directory **name**
 - Store its *location in the directory*
 - Perform directory specific operations, for example pathname lookup
- **/home/lkp/test.txt**
 - One dentry associated with each of: ‘/’, ‘home’, ‘lkp’ and ‘test.txt’
- Constructed on the fly as files and directories are accessed
 - Cache of disk representation

dentry

- A dentry can be `used`, `unused` or `negative`
- **Used** : corresponds to a valid inode (pointed by `d_inode`) with one or more users (`d_count`)
 - Cannot be discarded to free memory
- **Unused** : valid inode, but no current users
 - Kept in RAM for caching
 - Can be discarded

dentry

- **Negative** : does not point to a valid inode
 - E.g.. `open()` on a file that does not exists
 - Kept around for caching
 - Can be discarded
- Dentries are constructed on demand and **kept in RAM for quick future**

pathname lookups

- **dentry cache or dcache**
- **Q: Why does Linux cache negative dentries?**

dentry cache

- Linked list of used dentries linked by the `i_dentry` field of their inode
 - One inode can have multiple links, thus multiple dentries
- Linked list of LRU sorted unused and negative dentries
 - LRU: quick reclamation from the tail of the list
- Hash table + hash function to quickly resolve a path into the corresponding dentry present in the dcache

dentry cache

- Hash table: `dentry_hashtable` array
 - Each element is a pointer to a list of dentries hashing to the same value
- Hashing function: `d_hash()`
 - Filesystem can provide its own hashing function
- Dentry lookup in the dcache: `d_lookup()`
 - Returns dentry on success, `NULL` on failure
- Inodes are similarly cached in RAM, in the `inode cache`
 - Dentries in the dcache are pinning inodes in the inode cache

dentry operations

```
/* linux/include/linux/dcache.h */
struct dentry_operations {
    int (*d_revalidate)(struct dentry *, unsigned int);
    int (*d_weak_revalidate)(struct dentry *, unsigned int);
    int (*d_hash)(const struct dentry *, const struct qstr *);
    int (*d_compare)(const struct dentry *,
                     unsigned int, const char *, const struct qstr *);
    int (*d_delete)(const struct dentry *);
    int (*d_init)(struct dentry *);
    void (*d_release)(struct dentry *);
    void (*d_prune)(struct dentry *);
    void (*d_iput)(struct dentry *, struct inode *);
    char *(*d_dname)(struct dentry *, char *, int);
    struct vfsmount *(*d_automount)(struct path *);
    int (*d_manage)(const struct path *, bool);
    struct dentry *(*d_real)(struct dentry *, const struct inode *,
                           unsigned int);
} ____cacheline_aligned;
```

dentry operations

- `int d_revalidate(struct dentry *dentry, struct nameidata *)`
 - Determine if an entry to use from the dcache is valid
 - Generally set to `NULL`
- `int d_hash(struct dentry *dentry, struct qstr *name)`
 - Create a hash value for a dentry to insert in the dcache

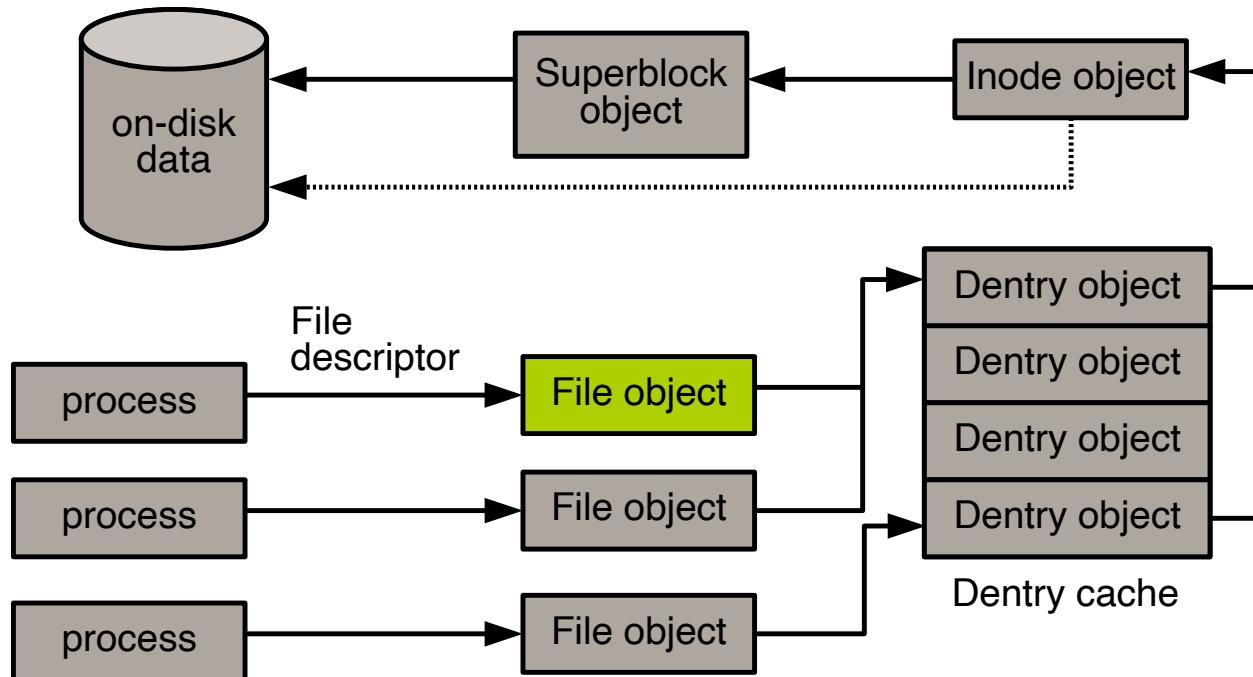
dentry operations

- `int d_compare(struct dentry *dentry, struct qstr *name1, struct qstr *name2)`
 - Compare two filenames, requires `dcache_lock`
- `int d_delete (struct dentry *dentry)`
 - Called by VFS when `d_count` reaches zero, requires `dcache_lock` and `d_lock`

dentry operations

- `void d_release(struct dentry *dentry)`
 - Called when the dentry is going to be freed
- `void d_iput(struct dentry *dentry, struct inode *inode)`
 - Called when the dentry loses its inode
 - Calls `iput()`

File object



File object

- The `file` object
 - Represents a file opened by a process
 - Created on `open()` and destroyed on `close()`
- 2 processes opening the same file:
 - Two file objects, pointing to the same unique dentry, that points itself on a unique inode
- No corresponding on-disk data structure

File object

```
/* linux/include/linux/fs.h */
struct file {
    struct path                  f_path;           /* contains the dentry */
    struct file_operations *f_op;          /** operations ***/
    spinlock_t                  f_lock;          /* lock */
    atomic_t                     f_count;         /* usage count */
    unsigned int                 f_flags;         /* open flags */
    mode_t                       f_mode;          /* file access mode */
    loff_t                       f_pos;           /* file offset */
    struct fown_struct          f_owner;         /* owner data for signals */
    const struct cred            *f_cred;         /* file credentials */
    struct file_ra_state         f_ra;            /* read-ahead state */
    u64                           f_version;       /* version number */
    void                          *private_data;    /* private data */
    struct list_head              f_ep_link;       /* list of epoll links */
    spinlock_t                  f_ep_lock;       /* epoll lock */
    struct address_space          *f_mapping;      /** page cache
                                                ** == inode->i_mapping ***/
    /* ... */
};
```

File operations

```
/* linux/include/linux/fs.h */
struct file_operations {
    loff_t (*llseek) (struct file *, loff_t, int);
    ssize_t (*read) (struct file *, char __user *, size_t, loff_t *);
    ssize_t (*write) (struct file *, const char __user *, size_t, loff_t *);
    ssize_t (*read_iter) (struct kiocb *, struct iov_iter *);
    ssize_t (*write_iter) (struct kiocb *, struct iov_iter *);
    int (*iterate) (struct file *, struct dir_context *);
    int (*iterate_shared) (struct file *, struct dir_context *);
    unsigned int (*poll) (struct file *, struct poll_table_struct *);
    /* ... */
};
```

File operations

- `loff_t llseek(struct file *file, loff_t offset, int origin)`
 - Update file offset
- `ssize_t read(struct file *file, char *buf, size_t count, loff_t *offset)`
 - Read operation
- `ssize_t aio_read(struct kiocb *iocb, char *buf, size_t count, loff_t offset)`
 - Asynchronous read

File operations

- `ssize_t write(struct file *file, const char *buf, size_t count, loff_t *offset)`
 - Write operation
- `ssize_t aio_write(struct kiocb *iocb, const char *buf, size_t count, loff_t offset)`
 - Asynchronous write
- `int readdir(struct file *file, void *dirent, filldir_t filldir)`
 - Read the next directory in a directory listing

File operations

- `unsigned int poll(struct file *file, struct poll_table_struct *poll_table)`
 - Sleeps waiting for activity on a given file
- `int ioctl(struct inode *inode, struct file *file, unsigned int cmd, unsigned long arg)`
 - Sends a command and arguments to a device
 - Unlocked/compat versions

File operations

- `int mmap(struct file *file, struct vm_area_struct *vma)`
 - Maps a file into an address space
- `int open(struct inode *inode, struct file *file)`
 - Opens a file
- `int flush(struct file *file)`
 - Called by VFS when the reference count of an open file decreases

File operations

- `int release(struct inode *inode, struct file *file)`
 - Called by VFS when the last reference to a file is destroyed
`close()` / `exit()`
- `int fsync(struct file *file, struct dentry *dentry, int datasync)`
 - Flush cached data on disk
- `int aio_fsync(struct kiocb *iocb, int datasync)`
 - Flush aio cached data on disk

File operations

- `int lock(struct file *file, int cmd, struct file_lock *lock)`
 - Manipulate a file lock
- `ssize_t writev(struct file *file, const struct iovec *vector, unsigned long count, loff_t *offset)`
- `ssize_t readv(struct file *file, const struct iovec *vector, unsigned long count)`
 - Vector read/write operations (used by the `readv` and `writev` family functions)

File operations

- `ssize_t sendfile(struct file *file, loff_t *offset, size_t size, read_actor_t actor, void *target)`
 - Copy data from one file to another entirely in the kernel
- `ssize_t sendpage(struct file *file, struct page *page, int offset, size_t size, loff_t *pos, int more)`
 - Send data from one file to another

File operations

- `unsigned long get_unmapped_area(struct file *file,
unsigned long addr, unsigned long len, unsigned long
offset, unsigned long flags)`
 - Get a section of unused address space to map a file
- `int flock(struct file *filp, int cmd, struct
file_lock *fl)`
 - Used by the `flock()` syscall

Filesystem data structures

- `struct file_system_type` : information about a specific concrete filesystem type
- One per filesystem supported (chosen at compile time) independently of the mounted filesystem
- Defined in `include/linux/fs.h`

Filesystem data structures

```
struct file_system_type {
    const char *name;          /** name: e.g., ext4 */
    int fs_flags;              /* flags */

    /** mount a partition */
    struct dentry *(*mount) (struct file_system_type *, int,
                            const char *, void *);

    /** terminate access to the superblock */
    void (*kill_sb) (struct super_block *);

    struct module *owner;      /* module owning the fs */
    struct file_system_type * next;        /* linked list of fs types */
    struct hlist_head fs_supers;           /* linked list of superblocks */

    /* runtime lock validation */
    struct lock_class_key s_lock_key;
    struct lock_class_key s_umount_key;
    struct lock_class_key s_vfs_rename_key;
    struct lock_class_key s_writers_key[SB_FREEZE_LEVELS];

    struct lock_class_key i_lock_key;
    struct lock_class_key i_mutex_key;
    struct lock_class_key i_mutex_dir_key;
};
```

Filesystem data structures

- When a filesystem is mounted, a `vfsmount` structure is created
 - Represent a specific instance of the filesystem: a mount point

```
/* linux/include/linux/fs.h */
struct vfsmount {
    struct dentry *mnt_root;      /* root of the mounted tree */
    struct super_block *mnt_sb;   /* pointer to superblock */
    int mnt_flags;
};
```

Process data structure

- `struct files_struct` : contains per-process information about opened files and file descriptors
 - `include/linux/fdtable.h`
- `struct fs_struct` : filesystem information related to a process
 - `include/linux/fs_struct.h`
- `struct mnt_namespace` : provide processes with unique views of a mounted filesystem
 - `fs/mount.h`

Summary

- Key data structures
 - `struct file_system_type` : file system (e.g., ext4)
 - `struct super_block` : mounted file system instance (i.e., partition)
 - `struct dentry` : path name
 - `struct inode` : file metadata
 - `struct file` : open file descriptor
 - `struct address_space` : per-inode page cache

Summary

- Three key caches
 - dentry cache: `dentry_hashtable`, `dentry->d_hash`,
`dentry->d_hash`
 - inode cache: `inode_hashtable`, `inode->i_hash`
 - page cache: `inode->i_mapping`

Further readings

- SFS: Random Write Considered Harmful in Solid State Drives, FAST12
- NOVA: A Log-structured File System for Hybrid Volatile/Non-volatile Main Memories, FAST16
- Performance and protection in the ZoFS user-space NVM file system, SOSP19
- CrossFS: A Cross-layered Direct-Access File System, OSDI20

Next lecture

- Page Cache and Page Fault

Page cache and Page fault

Dongyoon Lee

Summary of last lectures

- Tools: building, exploring, and debugging Linux kernel
- Core kernel infrastructure
- Process management & scheduling
- Interrupt & interrupt handler
- Kernel synchronization
- Memory management
- Virtual file system

Today: page cache and page fault

- Introduction to cache
- Page cache in Linux
- Cache eviction
- Interaction with memory management
- Flusher daemon

Latency numbers

L1 cache reference	0.5 ns
Branch mispredict	5 ns
L2 cache reference	7 ns
Mutex lock/unlock	25 ns
Main memory reference	100 ns
Compress 1K bytes with Zippy	3,000 ns = 3 µs
Send 2K bytes over 1 Gbps network	20,000 ns = 20 µs
SSD random read	150,000 ns = 150 µs
Read 1 MB sequentially from memory	250,000 ns = 250 µs
Round trip within same datacenter	500,000 ns = 0.5 ms
Read 1 MB sequentially from SSD*	1,000,000 ns = 1 ms
Disk seek	10,000,000 ns = 10 ms
Read 1 MB sequentially from disk	20,000,000 ns = 20 ms
Send packet CA->Netherlands->CA	150,000,000 ns = 150 ms

- Source: [Latency numbers every programmer should know](#)

Humanized version (x 1,000,000,000)

L1 cache reference	0.5 s	One heart beat (0.5 s)
Branch mispredict	5 s	Yawn
L2 cache reference	7 s	Long yawn
Mutex lock/unlock	25 s	Making a coffee
Main memory reference	100 s	Brushing your teeth
Compress 1K bytes with Zippy	50 min	One episode of a TV show (including ad breaks)
Send 2K bytes over 1 Gbps network	5.5 hr	From lunch to end of work day
SSD random read	1.7 days	A normal weekend
Read 1 MB sequentially from memory	2.9 days	A long weekend
Round trip within same datacenter	5.8 days	A medium vacation
Read 1 MB sequentially from SSD	11.6 days	Waiting for almost 2 weeks for a delivery
Disk seek	16.5 weeks	A semester in university
Read 1 MB sequentially from disk	7.8 months	Almost producing a new human being
The above 2 together	1 year	
Send packet CA->Netherlands->CA	4.8 years	Average time it takes to complete a bachelor's degree

Why caching

- Disk access is several orders of magnitude slower than memory access
- Data accessed once will, with a high likelihood, find itself accessed again in the near future → **temporal locality**

Page cache (or buffer cache)

- Physical pages in RAM holding disk content (blocks)
 - Disk is called a *Backing store*
 - Works for regular files, memory mapped files, and block device files
- Dynamic size
 - Grows to consume free memory unused by kernel and processes
 - Shrinks to relieve memory pressure

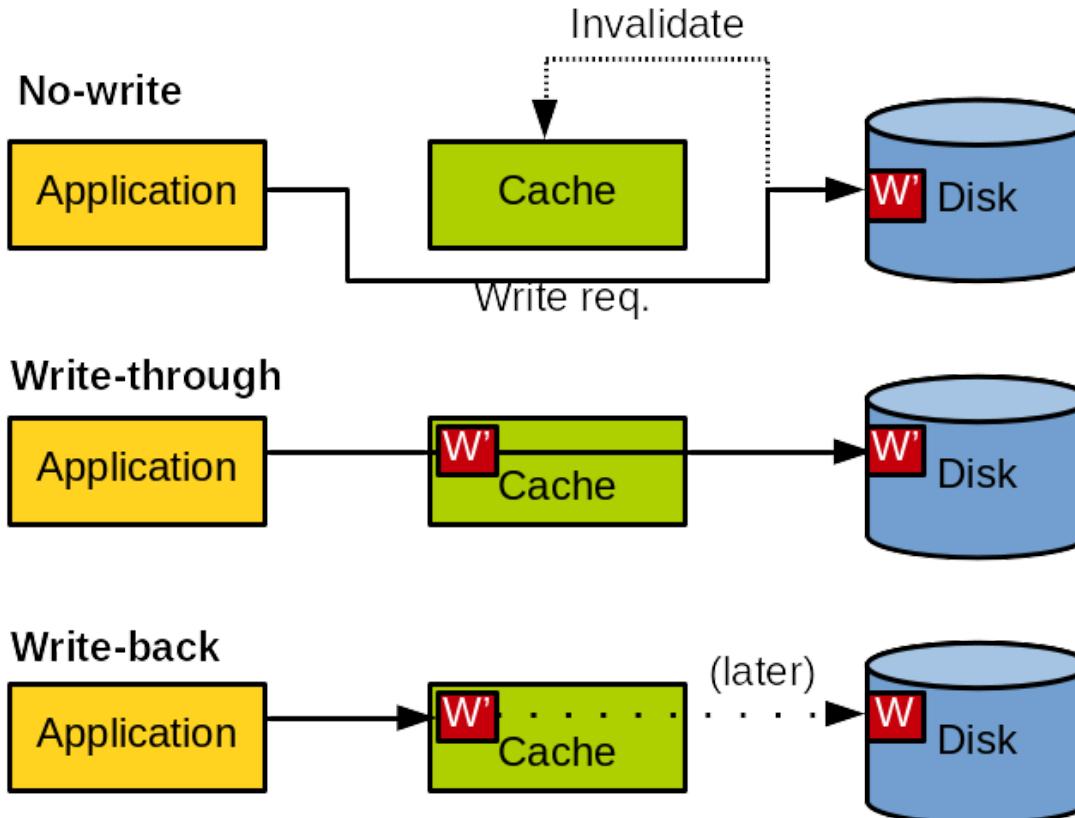
Page cache (or buffer cache)

- Buffered IO operations (without `O_DIRECT`), the page cache of a file is first checked
- **Cache hit:** if data is in the page cache, copy from/to user memory
- **Cache miss:** otherwise, VFS asks the concrete file system (e.g., ext4) to read data from disk
 - Read/write operations populate the page cache

Write caching policies

- **No-write:** does not cache write operations
- **Write-through:** write operations immediately go through to disk
 - Keeping the cache coherent
 - No need to invalidate cached data → simple
- **Write-back:** write operations update page cache but disk is not immediately updated → **Linux page cache policy**
 - Pages written are marked *dirty* using a tag in radix tree
 - Periodically, write dirty pages to disk → *writeback*
 - Page cache absorbs temporal locality to reduce disk access

Write caching policies

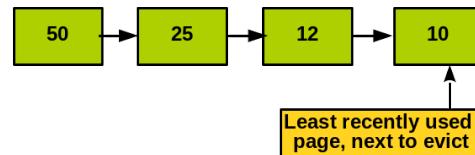


Cache eviction

- **When data should be removed from the cache?**
 - Need more free memory (memory pressure)
- **Which data should be removed from the cache?**
 - Ideally, evict cache pages that will not be accessed in the future
 - **Eviction policy:** deciding what to evict

Eviction policy: LRU

- **Least recently used (LRU)** policy
 - Keep track of when each page is accessed
 - Evict the pages with the oldest timestamp



- Failure cases of LRU policy
 - Many files are accessed once and then never again
 - LRU puts them at the top of LRU list → not optimal

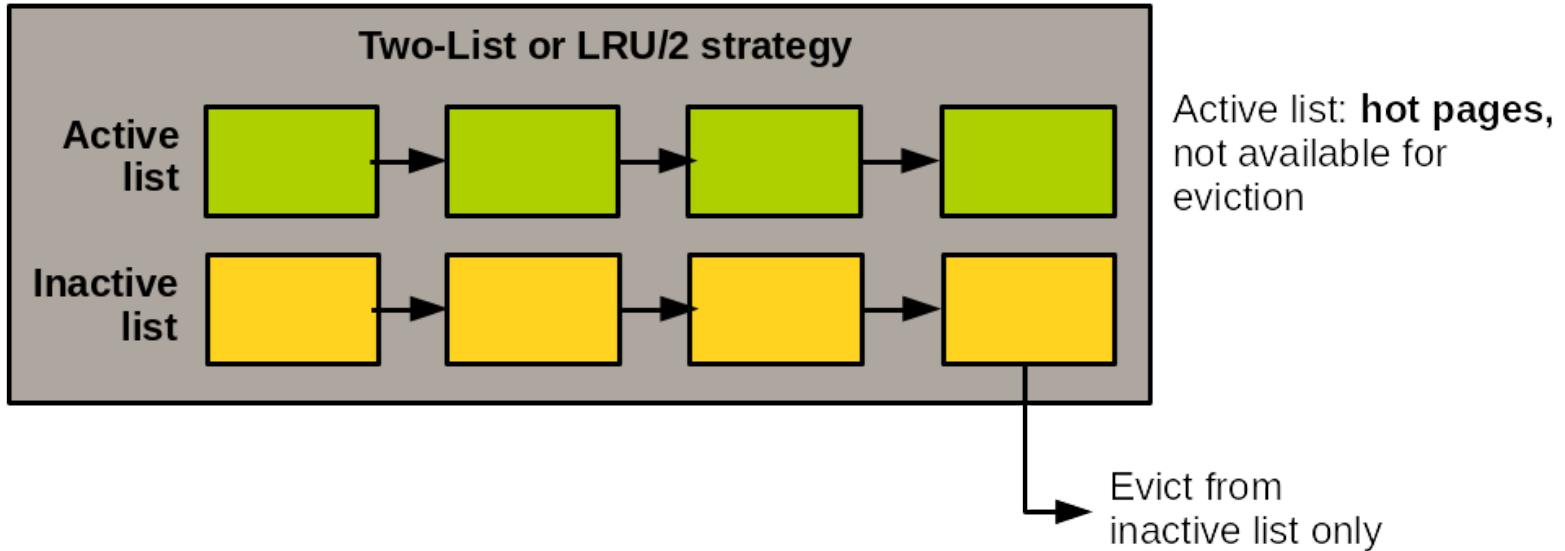
The two-list strategy

- **Active list**
 - Pages in the active list is considered *hot*
 - Not available for eviction
- **Inactive list**
 - Pages in the inactive list is considered *cold*
 - Available for eviction

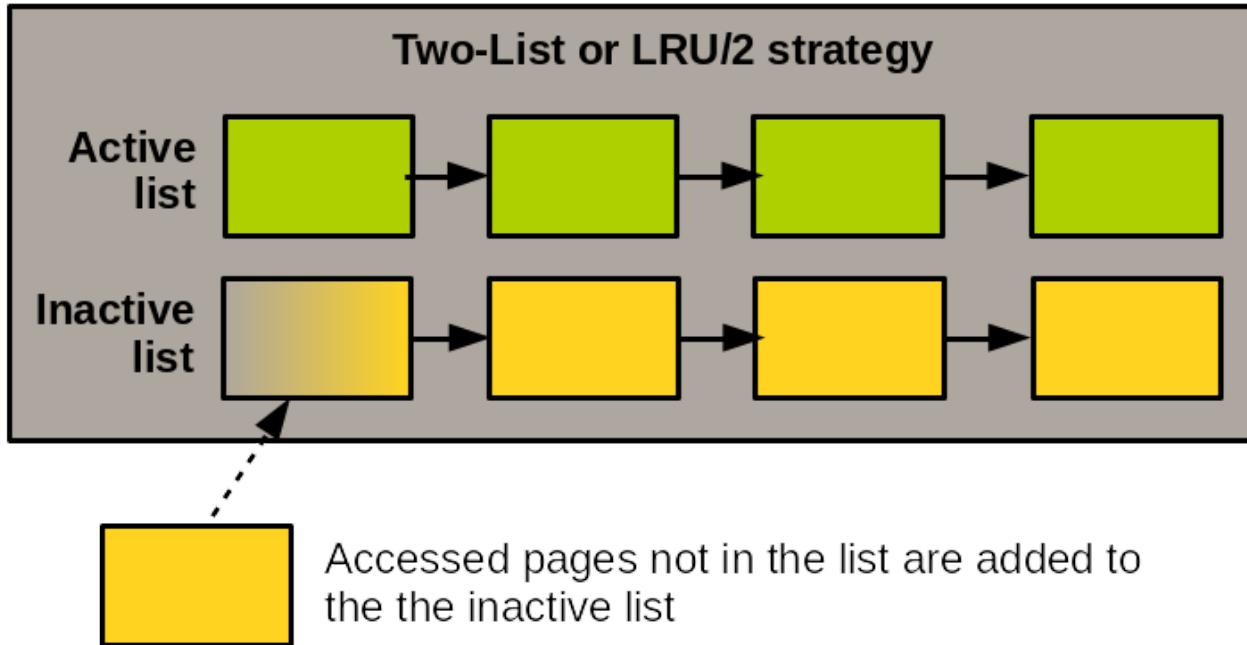
The two-list strategy

- Newly accessed pages are added to inactive list
- If a page in an inactive list is accessed again, it is promoted to an active list
 - When a page is moved to an inactive list, its access permission in a page table is removed.
- If an active list becomes much larger than an inactive list, items from the active list's head are moved back to the inactive list.
- *When a page is added to inactive list, its access permission in the page table is disabled to track its access.*

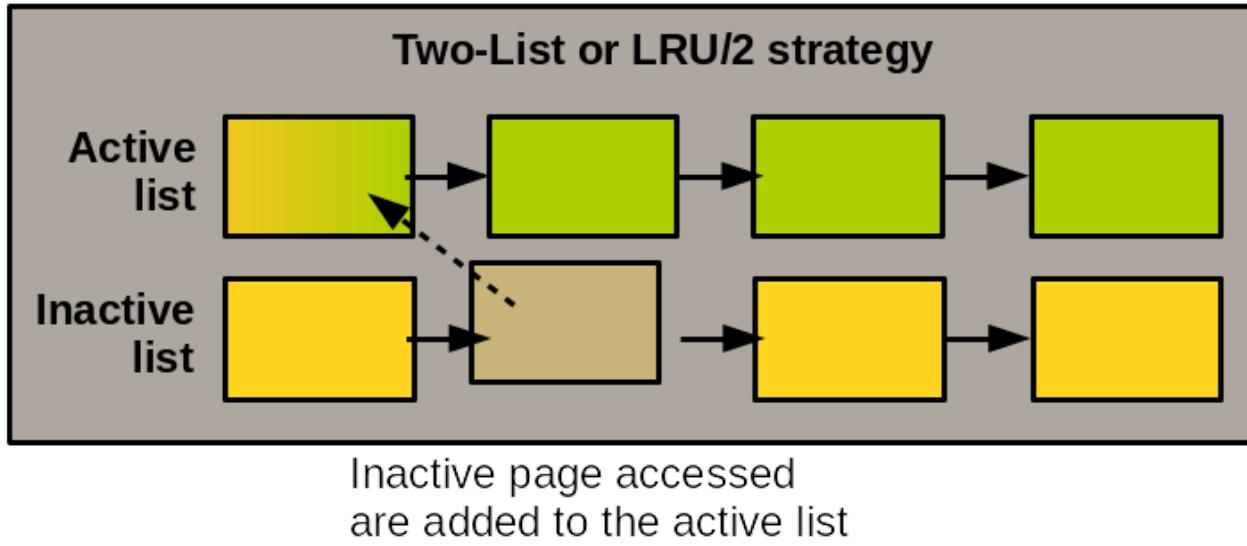
The two-list strategy



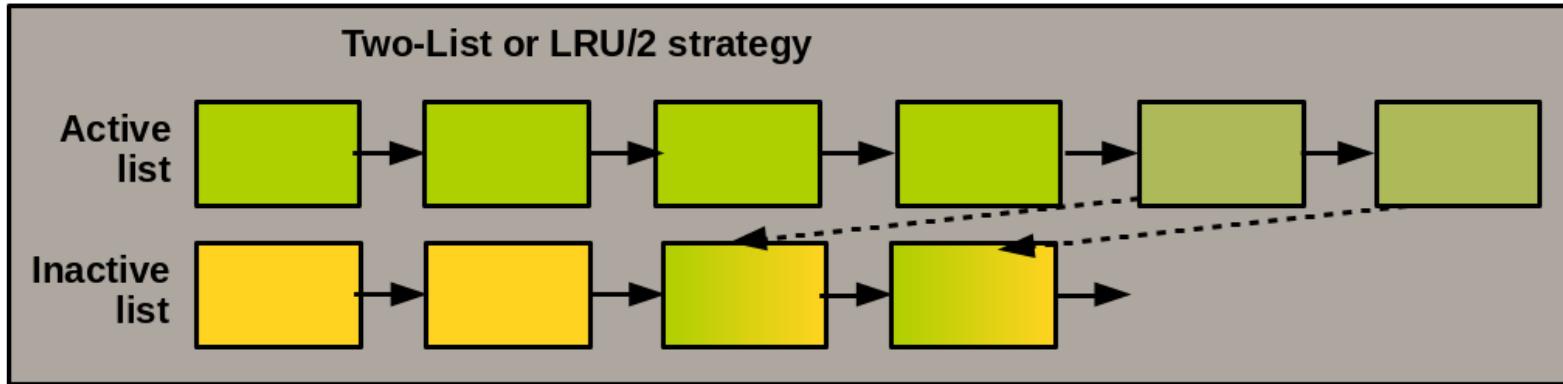
The two-list strategy



The two-list strategy



The two-list strategy



Lists are balanced and active pages are evicted in the inactive list

The Linux page cache (or buffer cache)

```
/* linux/include/linux/fs.h */
struct inode {
    const struct inode_operations *i_op;
    struct super_block *i_sb;
    struct address_space *i_mapping;
    unsigned long i_ino;
};

struct address_space {
    struct inode *host;          /* owner: inode, block_device */
    struct radix_tree_root page_tree; /* radix tree of all pages */
    spinlock_t tree_lock; /* and lock protecting it */
};

/* Insert an item into the radix tree at position @index. */
int radix_tree_insert(struct radix_tree_root *root,
                      unsigned long index, void *item);

/* linux/mm/shmem.c */
static int shmem_add_to_page_cache(struct page *page,
                                   struct address_space *mapping, pgoff_t index, void *expected)
{
    error = radix_tree_insert(&mapping->page_tree, index, page);
}
```

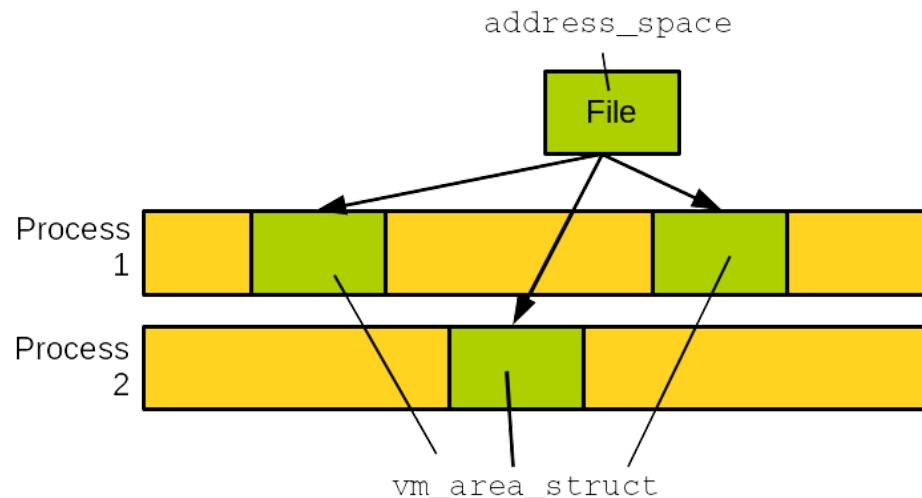
The Linux page cache (or buffer cache)

```
$> sudo cat /proc/1/maps
7fe87b1f1000-7fe87b21d000 r-xp 00000000 fd:00 1975147 /usr/lib64/libseccomp.so
7fe87b21d000-7fe87b41c000 ---p 0002c000 fd:00 1975147 /usr/lib64/libseccomp.so
7fe87b41c000-7fe87b431000 r--p 0002b000 fd:00 1975147 /usr/lib64/libseccomp.so
7fe87b431000-7fe87b432000 rw-p 00040000 fd:00 1975147 /usr/lib64/libseccomp.so
7fe87b432000-7fe87b439000 r-xp 00000000 fd:00 1975989 /usr/lib64/librt-2.26.so
7fe87b439000-7fe87b638000 ---p 00007000 fd:00 1975989 /usr/lib64/librt-2.26.so
7fe87b638000-7fe87b639000 r--p 00006000 fd:00 1975989 /usr/lib64/librt-2.26.so
7fe87b639000-7fe87b63a000 rw-p 00007000 fd:00 1975989 /usr/lib64/librt-2.26.so
```

- Q: the number of `vm_area_struct`
- Q: the number of `inode`
- Q: the number of `address_space`

address_space

- An entity present in the page cache
 - an `address_space` = a file = accessing a page cache of a file
 - an `address_space` = one or more `vm_area_struct`



address_space

```
/* linux/include/linux/fs.h */
struct address_space {
    struct inode                  *host;           /* owning inode */
    struct radix_tree_root        page_tree;       /* radix tree of all pages */
    spinlock_t                   tree_lock;       /* page tree lock */
    unsigned int                 i_mmap_writable; /* VM_SHARED (writable)
                                                * mapping count */
    struct rb_root                i_mmap;          /* list of all mappings */
    unsigned long                 nrpages;         /* total number of pages */
    pgoff_t                      writeback_index; /* writeback start offset */
    struct address_space_operations a_ops;        /* operations table */
    unsigned long                 flags;           /* error flags */
    gfp_t                         gfp_mask;        /* gfp mask for allocation */
    struct backing_dev_info       backing_dev_info; /* read-ahead info */
    spinlock_t                   private_lock;    /* private lock */
    struct list_head               private_list;   /* private list */
    struct address_space          assoc_mapping; /* associated buffers */
    /* ... */
}
```

address_space

- `i_mmap` : all shared and private mappings concerning this address space
- `nrpages` : total number of pages in the address space
- `host` : points to the inode of the corresponding file
- `a_ops` : address space operations

address_space_operations

```
/* linux/include/linux/fs.h */
struct address_space_operations {
    int (*writepage)(struct page *page, struct writeback_control *wbc);
    int (*readpage)(struct file *, struct page *);
    int (*writepages)(struct address_space *, struct writeback_control *);
    int (*set_page_dirty)(struct page *page);
    int (*readpages)(struct file *filp, struct address_space *mapping,
                    struct list_head *pages, unsigned nr_pages);
    int (*write_begin)(struct file *, struct address_space *mapping,
                      loff_t pos, unsigned len, unsigned flags,
                      struct page **pagep, void **fsdata);
    int (*write_end)(struct file *, struct address_space *mapping,
                    loff_t pos, unsigned len, unsigned copied,
                    struct page *page, void *fsdata);
    /* ... */
};
```

Page read operation

- `read()` function from the `file_operations`
 - `generic_file_buffered_read()`
- Search the data in the page cache
 - `page = find_get_page(mapping, index)`
- Adding the page to the page cache
 - `page = __page_cache_alloc(gfp_mask);`
- Then, read data from disk
 - `mapping->a_ops->readpage(filp, page)`

Page write operation

- When a page is modified in the page cache, mark it as dirty
 - `SetPageDirty(page)`
- Default write path: in `mm/filemap.c`

```
/* search the page cache for the desired page. If the page is not present,
an entry is allocated and added: */
page = __grab_cache_page(mapping, index, &cached_page, &lru_pvec);
/* Set up the write request: */
status = a_ops->write_begin(file, mapping, pos, bytes, flags, &page, &fsdata);
/* Copy data from user-space into a kernel buffer: */
copied = iov_iter_copy_from_user_atomic(page, i, offset, bytes);
/* write data to disk: */
status = a_ops->write_end(file, mapping, pos, bytes, copied, page, fsdata);
```

Interaction with memory management

- `file`, `file_operations`
 - How to access the contents of a file
- `address_space`, `address_space_operations`
 - How to access the page cache of a file
- `vm_area_struct`, `vm_operations_struct`
 - How to handle page fault of a virtual memory region
- Page table in x86 processor

file

```
/* linux/include/linux/fs.h */
struct file {
    struct path                  f_path;           /* contains the dentry */
    struct file_operations *f_op;          /* operations */
    spinlock_t                  f_lock;           /* lock */
    atomic_t                    f_count;          /* usage count */
    unsigned int                f_flags;          /* open flags */
    mode_t                      f_mode;           /* file access mode */
    loff_t                      f_pos;            /* file offset */
    struct fown_struct         f_owner;          /* owner data for signals */
    const struct cred           *f_cred;          /* file credentials */
    struct file_ra_state        f_ra;             /* read-ahead state */
    u64                         f_version;        /* version number */
    void                        *private_data;     /* private data */
    struct list_head             f_ep_link;        /* list of epoll links */
    spinlock_t                  f_ep_lock;        /* epoll lock */
    struct address_space        *f_mapping;       /* page cache mapping */
    /* ... */
};
```

file

```
/* linux/include/linux/fs.h */
struct file_operations {
    struct module *owner;
    loff_t (*llseek) (struct file *, loff_t, int);
    ssize_t (*read) (struct file *, char __user *, size_t, loff_t *);
    ssize_t (*write) (struct file *, const char __user *, size_t, loff_t *);
    ssize_t (*read_iter) (struct kiocb *, struct iov_iter *);
    ssize_t (*write_iter) (struct kiocb *, struct iov_iter *);
    int (*iterate) (struct file *, struct dir_context *);
    int (*iterate_shared) (struct file *, struct dir_context *);
    unsigned int (*poll) (struct file *, struct poll_table_struct *);
    /* ... */
};
```

address_space

```
/* linux/include/linux/fs.h */
struct address_space {
    struct inode                  *host;           /* owning inode */
    struct radix_tree_root        page_tree;       /* radix tree of all pages */
    spinlock_t                   tree_lock;       /* page tree lock */
    unsigned int                 i_mmap_writable; /* VM_SHARED (writable)
                                                * mapping count */
    struct rb_root                i_mmap;          /* list of all mappings */
    unsigned long                 nrpages;         /* total number of pages */
    pgoff_t                      writeback_index; /* writeback start offset */
    struct address_space_operations a_ops;        /* operations table */
    unsigned long                 flags;           /* error flags */
    gfp_t                         gfp_mask;        /* gfp mask for allocation */
    struct backing_dev_info       backing_dev_info; /* read-ahead info */
    spinlock_t                   private_lock;    /* private lock */
    struct list_head               private_list;   /* private list */
    struct address_space          assoc_mapping; /* associated buffers */
    /* ... */
}
```

address_space

```
/* linux/include/linux/fs.h */
struct address_space_operations {
    int (*writepage)(struct page *page, struct writeback_control *wbc);
    int (*readpage)(struct file *, struct page *);
    int (*writepages)(struct address_space *, struct writeback_control *);
    int (*set_page_dirty)(struct page *page);
    int (*readpages)(struct file *filp, struct address_space *mapping,
                    struct list_head *pages, unsigned nr_pages);
    int (*write_begin)(struct file *, struct address_space *mapping,
                      loff_t pos, unsigned len, unsigned flags,
                      struct page **pagep, void **fsdata);
    int (*write_end)(struct file *, struct address_space *mapping,
                    loff_t pos, unsigned len, unsigned copied,
                    struct page *page, void *fsdata);
    /* ... */
};
```

- *Q: what is the difference between `file->read()` and `asop->readpage()`? See `linux/fs/ext4/file.c`*

vm_area_struct

```
struct vm_area_struct {  
    struct mm_struct *vm_mm; /* associated address space */  
    unsigned long vm_start; /* VMA start, inclusive */  
    unsigned long vm_end; /* VMA end, exclusive */  
    struct vm_area_struct *vm_next; /* list of VMAs */  
    struct vm_area_struct *vm_prev; /* list of VMAs */  
    pgprot_t vm_page_prot; /* access permissions */  
    unsigned long vm_flags; /* flags */  
    struct rb_node vm_rb; /* VMA node in the tree */  
    struct list_head anon_vma_chain; /* list of anonymous mappings */  
    struct anon_vma *anon_vma; /* anonymous vma object */  
    struct vm_operations_struct *vm_ops; /* operations */  
    unsigned long vm_pgoff; /* offset within file */  
    struct file *vm_file; /* mapped file (can be NULL) */  
    void *vm_private_data; /* private data */  
/* ... */  
}
```

vm_area_struct

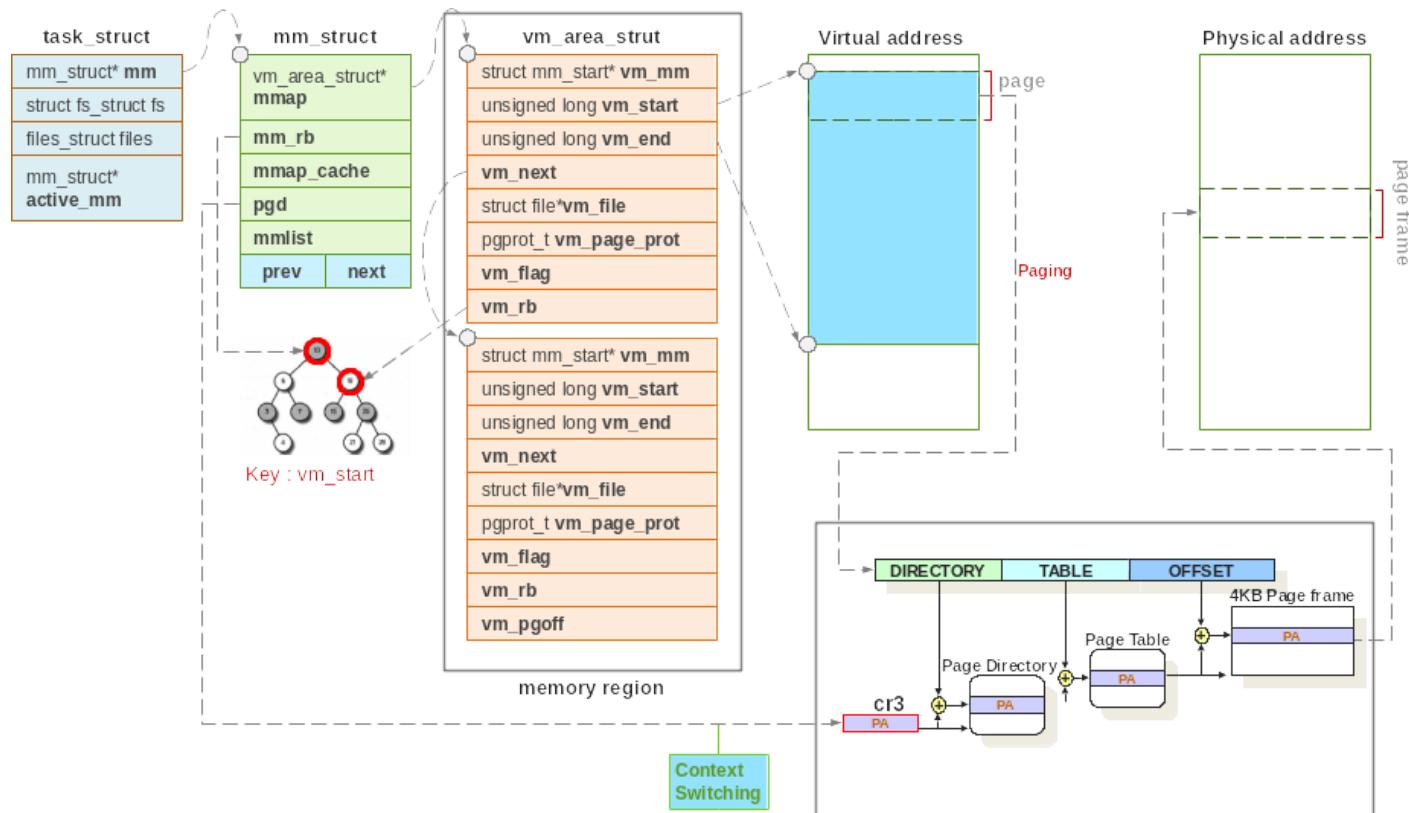
```
/* linux/include/linux/mm.h */
struct vm_operations_struct {
    /* called when the area is added to an address space */
    void (*open)(struct vm_area_struct * area);

    /* called when the area is removed from an address space */
    void (*close)(struct vm_area_struct * area);

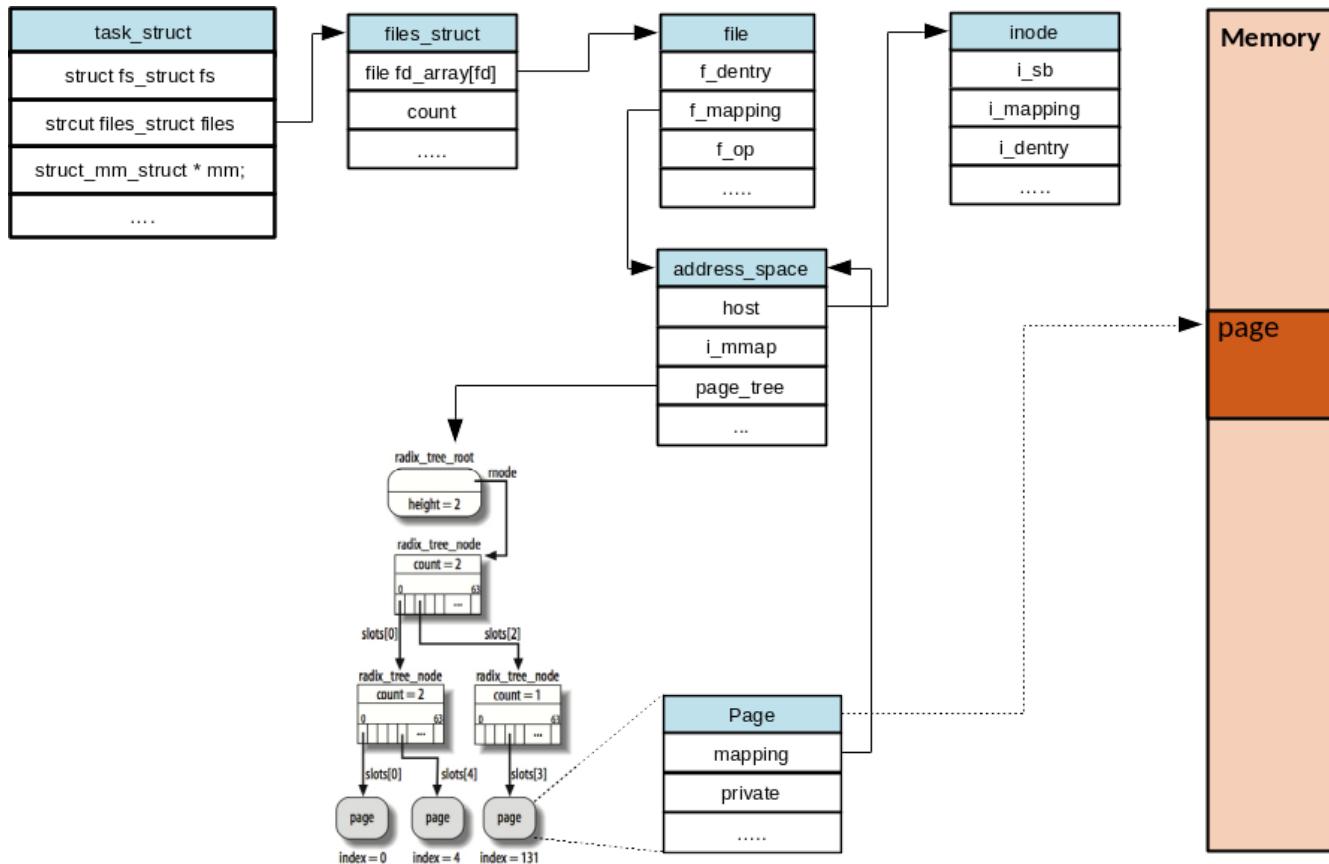
    /* invoked by the page fault handler when a page that is
     * not present in physical memory is accessed*/
    int (*fault)(struct vm_area_struct *vma, struct vm_fault *vmf);

    /* invoked by the page fault handler when a previously read-only
     * page is made writable */
    int (*page_mkwrite)(struct vm_area_struct *vma, struct vm_fault *vmf);
    /* ... */
}
```

vm_area_struct - page table



Page cache - physical page



Page fault handling

- Entry point: `handle_pte_fault` (mm/memory.c)
- Identify which VMA faulting address falls in
- Identify if VMA has a registered fault handler
- Default fault handlers
 - `do_anonymous_page` : no page and no file
 - `filemap_fault` : page backed by file
 - `do_wp_page` : write protected page (CoW)
 - `do_swap_page` : page backed by swap

File-mapped page fault: `filemap_fault`

- PTE entry does not exist (---)
- BUT VMA is marked as accessible (e.g., `rwx`) and has an associated file (`vm_file`)
- Page fault handler notices differences
 - In `filemap_fault`
 - Look up a page cache of the file
 - If cache hit, map the page in the cache
 - Otherwise, `mapping->a_ops->readpage(file, page)`

Copy on Write: `do_wp_page`

- PTE entry is marked as un-writable (e.g., `r--`)
- But VMA is marked as writable (e.g., `rw-`)
- Page fault handler notices differences
 - In `do_wp_page`
 - Must mean CoW
 - Make a duplicate of physical page
 - Update PTEs and flush TLB entry

Flusher daemon

- Write operations are deferred, data is marked *dirty*
 - RAM data is out-of-sync with the storage media
- Dirty page writeback occurs
 - Free memory is low and the page cache needs to shrink
 - Dirty data grows older than a specific threshold
 - User process calls `sync()` or `fsync()`
- Multiple **flusher threads** are in charge of syncing dirty pages from the page cache to disk

Flusher daemon

- When the free memory goes below a given threshold, the kernel
 - `wakeup_flusher_threads()`
 - Wakes up one or several flusher threads performing writeback through `bdi_writeback_all`
- Thread write data to disk until
 - `num_pages_to_write` have been written
 - and the amount of memory drops below the threshold
- percentage of total memory to trigger flusher daemon
 - `/proc/sys/vm/dirty_background_ratio`

Flusher daemon

- At boot time a timer is initialized to wake up a flusher thread calling
 - `wb_writeback()`
- Writes back all data older than a given value
 - `/proc/sys/vm/dirty_expire_interval`
- Timer reinitialized to expire at a given time in the future: now + period
 - `/proc/sys/vm/dirty_writeback_interval`
- Multiple other parameters related to the writeback and the control of the page cache in general are present in `/proc/sys/vm`
 - More info: Documentation/sysctl/vm.txt

Further readings

- [Latency numbers every programmer should know](#)
- [LWN: Better active/inactive list balancing](#)
- [LWN: Flushing out pdflush](#)
- [LWN: User-space page fault handling](#)
- [W4118 @ Columbia University](#)

Ext4 file system and crash consistency

Dongyoon Lee

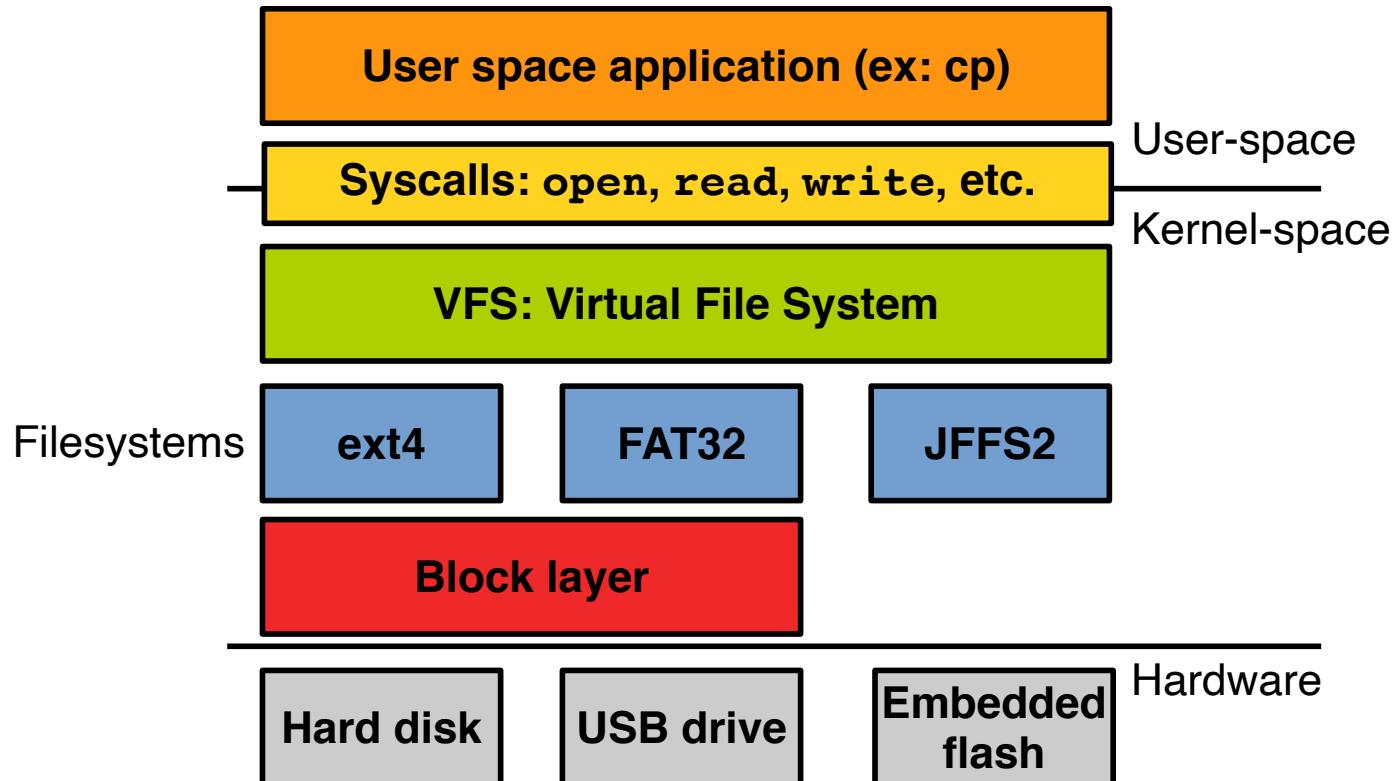
Summary of last lectures

- Tools: building, exploring, and debugging Linux kernel
- Core kernel infrastructure
- Process management & scheduling
- Interrupt & interrupt handler
- Kernel synchronization
- Memory management
- Virtual file system
- Page cache and page fault

Today: ext4 file system and crash consistency

- File system in Linux kernel
- Design considerations of a file system
- History of file system
- On-disk structure of Ext4
- File operations
- Crash consistency

File system in Linux kernel



What is a file system fundamentally?

```
int main(int argc, char *argv[])
{
    int fd;
    char buffer[4096];
    struct stat_buf;
    DIR *dir;
    struct dirent *entry;

    /* 1. Path name -> inode mapping */
    fd = open("/home/lkp/hello.c" , O_RDONLY);

    /* 2. File offset -> disk block address mapping */
    pread(fd, buffer, sizeof(buffer), 0);

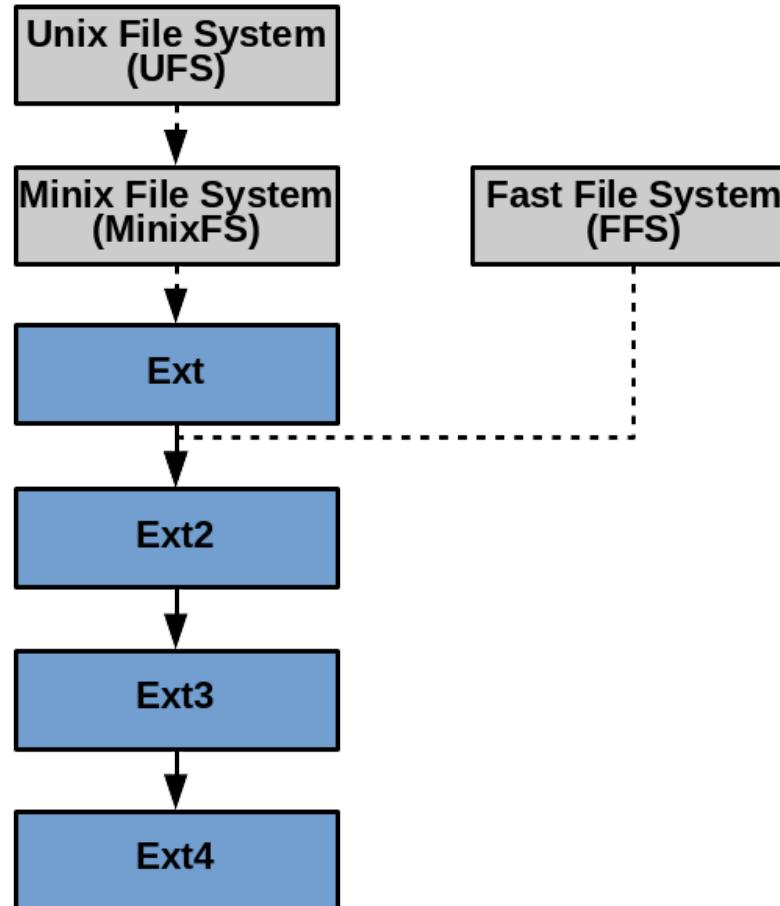
    /* 3. File meta data operation */
    fstat(fd, &stat_buf);
    printf("file size = %d\n", stat_buf.st_size);

    /* 4. Directory operation */
    dir = opendir("/home");
    entry = readdir(dir);
    printf("dir = %s\n", entry->d_name);
    return 0;
}
```

Why do we care EXT4 file system?

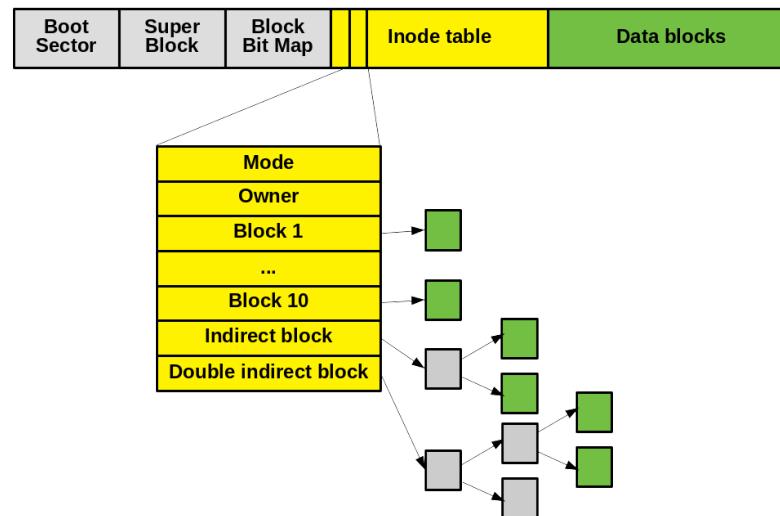
- **Most widely-deployed file system**
 - Default file system of major Linux distributions
 - File system used in Google data center
 - Default file system of Android kernel
- **Follows the traditional file system design**

History of file system design



UFS (Unix File System)

- The original UNIX file system
 - Design by Dennis Ritche and Ken Thompson (1974)
- The first Linux file system (ext) and Minix FS has a similar layout



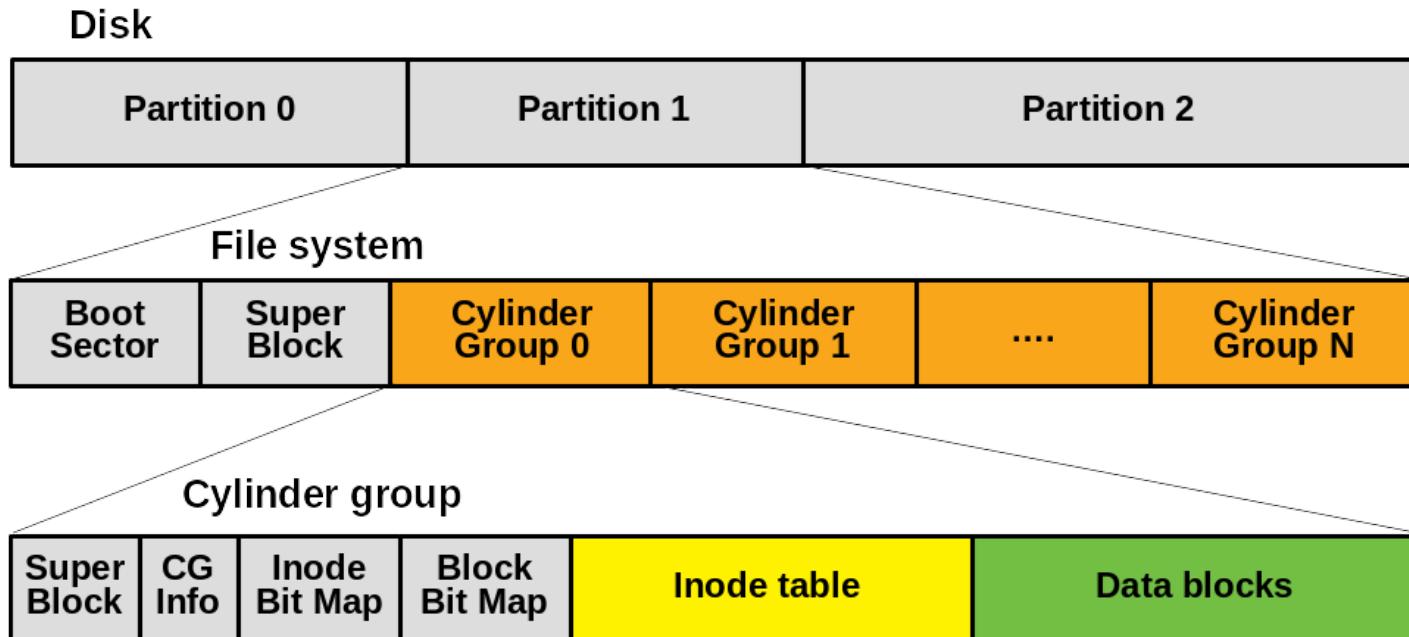
UFS (Unix File System)

- Performance problem of UFS (and the first Linux file system)
 - Especially, long seek time between an inode and data block

FFS (Fast File System)

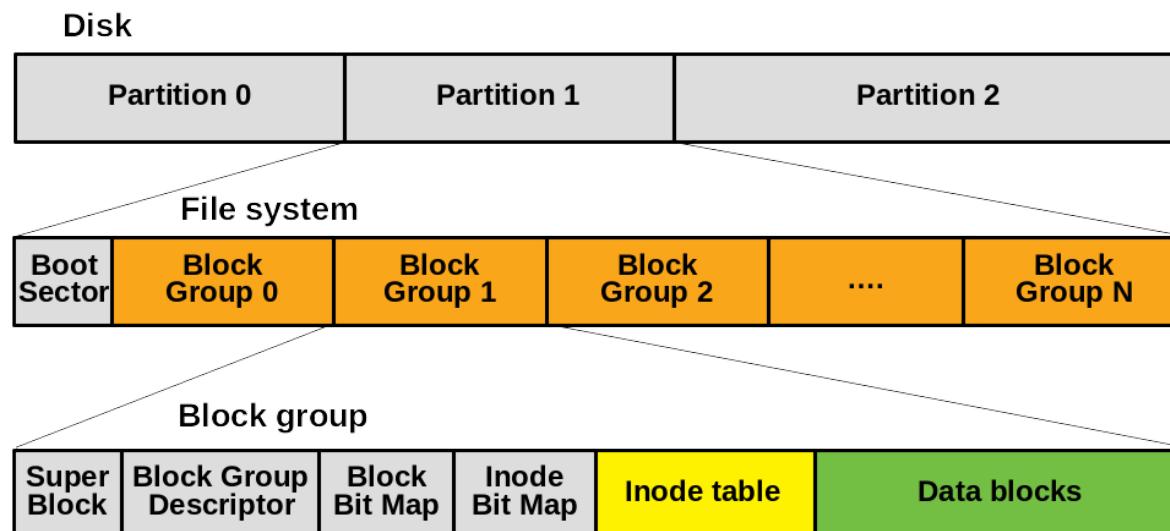
- The file system of BSD UNIX
 - Designed by Marshall Kirk McKusick, et al. (1984)
- Cylinder group: one or more consecutive cylinders
 - Disk block allocation heuristics to reduce seek time
 - Try to locate inode and associate data in the same cylinder group
- Many in-place-update file systems follows the design of FFS

FFS (Fast File System)

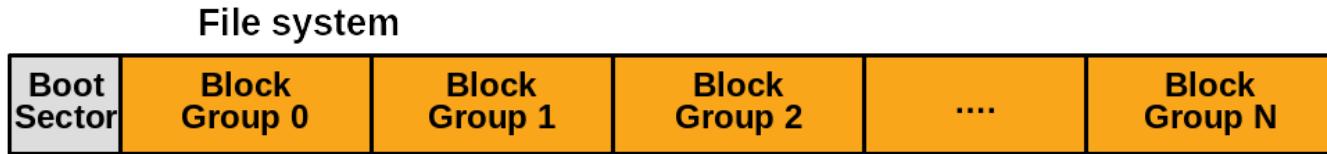


Ext2 file system

- The second extended version (ext2) of the first Linux file system (ext)
- Design is influenced by FFS in BSD UNIX
 - block group (similar to cylinder group) to reduce disk seek time

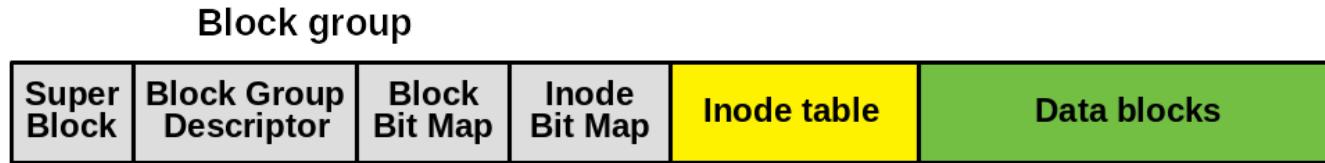


Ext2 file system



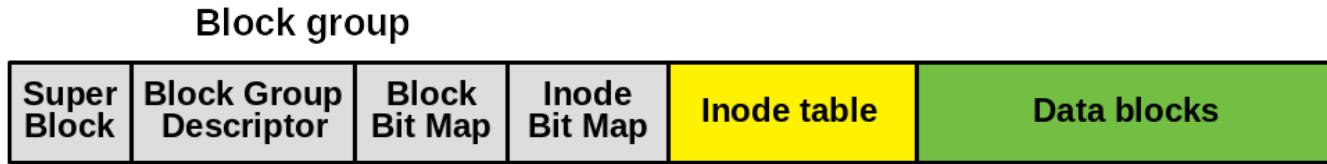
- The first block is reserved for the boot sector (not managed by file system)
- The rest of a Ext2 partition is split into block groups
- All the block groups have same size and are stored sequentially

Ext2 file system



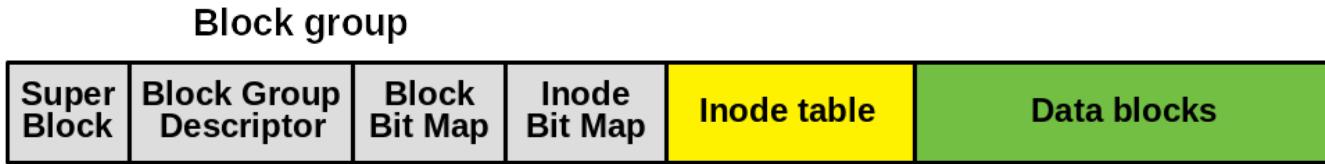
- **Superblock**: stores information describing filesystem
 - E.g., block size, number of inodes and blocks, number of free blocks in a file system, etc.
 - Each block group has a copy of superblock for recovery
- **Group Descriptor**: describe each block group information
 - E.g., number of free blocks in a block group, free inodes, and used directories in the group

Ext2 file system



- **Inode bitmap**: summarized inode usage information
 - 0 → free, 1 → in use
 - 100th bit in inode bitmap = usage of 100th inode in inode table
- **Block bitmap**: summarized data block usage information
 - 0 → free, 1 → in use
 - 100th bit in block bitmap = usage of 100th data block in a block group

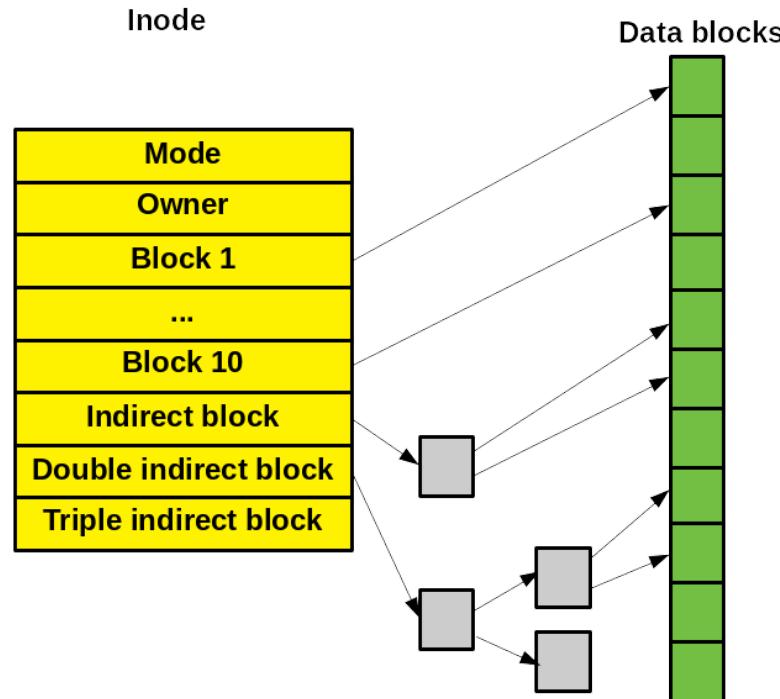
Ext2 file system



- **Inode table**: fixed size array of inodes
 - inode number = starting inode number of a block group + inode location in inode table
- **Data blocks**: actual data contents

Ext2 file system

- **Inode indirect block map:** file offset → disk block address mapping



Ext2 file system

- Directory: a linked list
 - Directory access performance with many files is terrible

```
$> ls
.  home  sbin
..  usr
```

	inode	rec_len	file_type	name_len	name
0	21	12	1	2	.
12	22	12	2	2	.
24	53	16	5	2	h o m e
40	67	28	3	2	u s x
52	0	16	7	1	o l d f i l e
68	34	12	4	2	s b i n

Deleted file

Ext2 file system

- **Q: How to open /home/lkp/hello.c**
 - / → home/ → lkp/ → hello.c
- **Q: Where to start?**

```
/* linux/fs/ext2/ext2.h */  
  
/*  
 * Special inode numbers  
 */  
#define EXT2_BAD_INO          1 /* Bad blocks inode */  
#define EXT2_ROOT_INO          2 /* Root inode */  
#define EXT2_BOOT_LOADER_INO   5 /* Boot loader inode */  
#define EXT2_UNDEL_DIR_INO     6 /* Undelete directory inode */
```

Ext3 file system

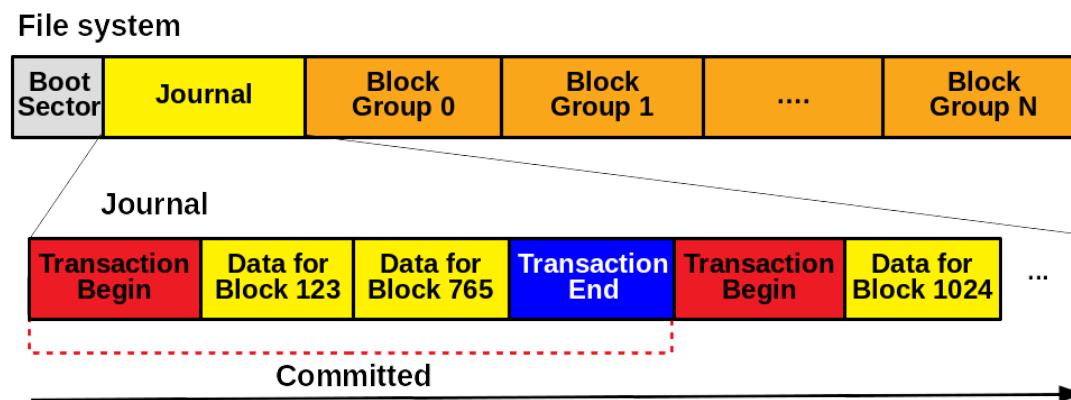
- Integrate **journaling** into ext2 file system to guarantee file system consistency after sudden system crash/power-off
- Journaling = WAL (Write-Ahead Logging) in database systems
- When updating disks, before overwriting the structure in place, first write down what you are about to do in a well-known location.

File system



Ext3 file system: journaling

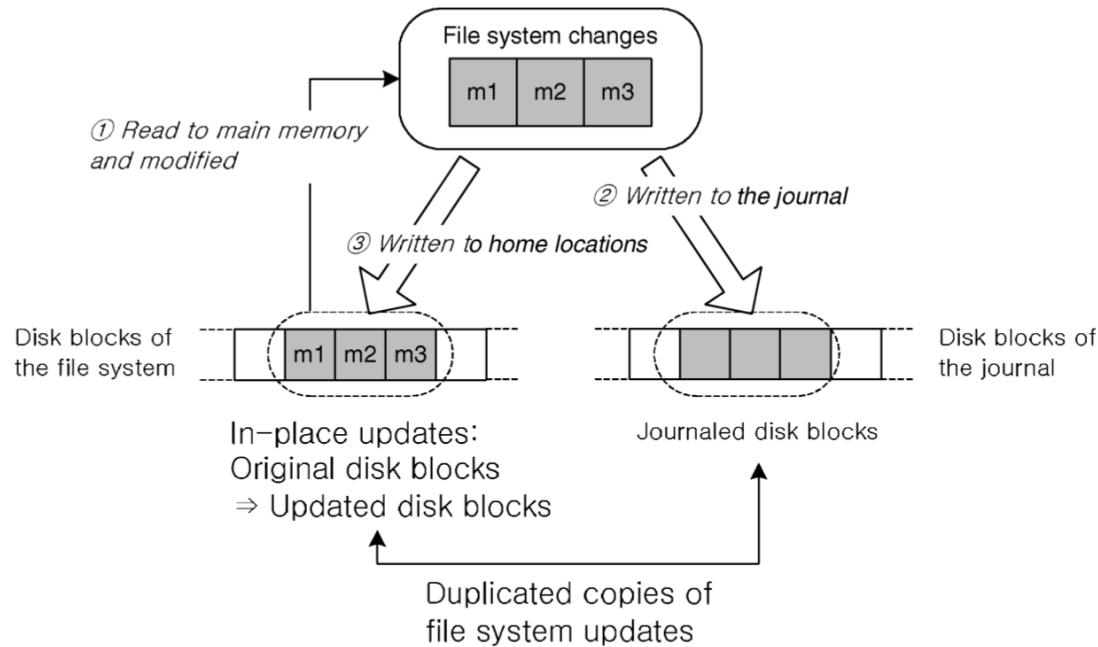
- Journal is logically a fixed-size, circular array
 - Implemented as a special file with a hard-coded inode number
 - Each journal transaction is composed of a begin marker, log, and end marker



Ext3 file system: journaling

- **Commit**
 1. Write a transaction begin marker
 2. Write logs
 3. **Write barrier**
 4. Write a transaction end marker
- **Checkpointing**
 - After ensuring this journal transaction is written to disk (i.e., commit), we are ready to update the original data

Ext3 file system: journaled write



- Source: [JFTL: A Flash Translation Layer Based on a Journal Remapping for Flash Memory](#)

Ext3 file system: journal recovery

- **Case 1: failure between journal commit and checkpointing**
 - There is a transaction end marker at the tail of the journal → logs between begin and end markers are consistent
 - Replay logs in the last journal transaction
- **Case 2: failure before journal commit**
 - There is no transaction end marker at the tail of the journal → there is no guarantee that logs between begin and end markers are consistent
 - Ignore the last journal transaction

Ext3 file system: journaling mode

- **Journal**
 - Metadata and content are saved in the journal.
- **Ordered**
 - Only metadata is saved in the journal. Metadata are journaled only after writing the content to disk. This is the default.
- **Writeback**
 - Only metadata is saved in the journal. Metadata might be journaled either before or after the content is written to the disk.

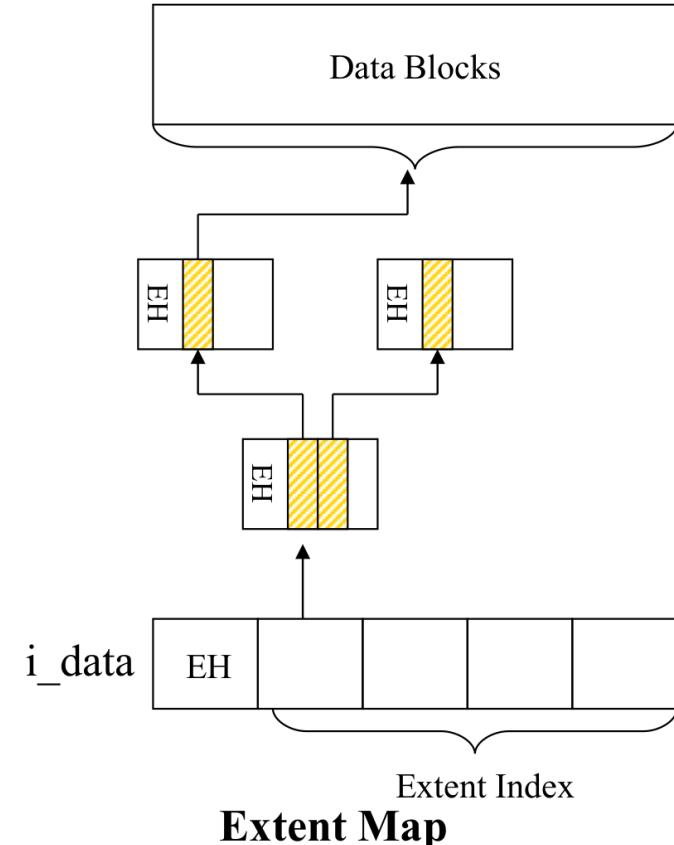
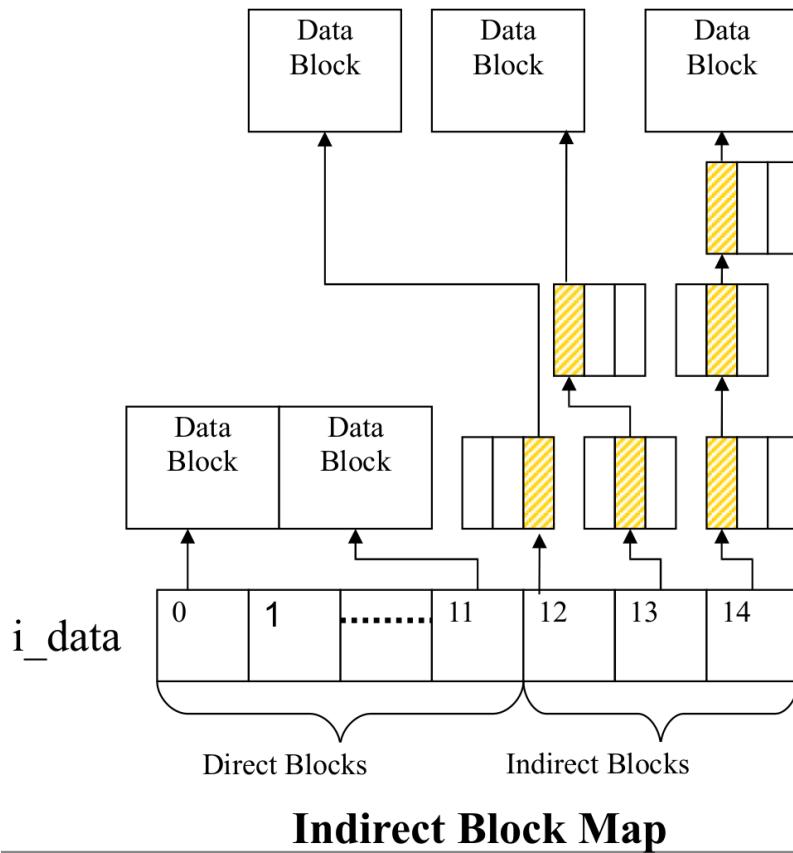
Ext4 file system

- Larger file system capacity with 64-bit
- Supports huge file size (from 16GB to 16TB) and file system (from 2^{32} to 2^{64} blocks)
 - **Indirect block map → extent tree**
- Directory can contain up to 64,000 subdirs (32,000 in ext3)
 - **List → HTree**

Indirect Block Map vs Extent map

- **Indirect block map**
 - Incredibly inefficient
 - One extra block read (and seek) every 1024 blocks
 - Really obvious when deleting large movie files
- **Extents**
 - An efficient way to represent large file
 - A single descriptor for a range of contiguous blocks
 - E.g., {file block offset, length, disk block address} = {100, 1000, 200}

Indirect Block Map vs Extent map



Extent related works

- Multiple block allocation
 - Allocate contiguous blocks together
 - Reduce fragmentation, reduce extent metadata
- Delayed allocation
 - Defer block allocation to writeback time
 - Improve chances allocating contiguous blocks, reducing fragmentation

HTree (hashed b-tree)

- Directory structure of ext3/4
- B-tree using hashed value of a file name in a directory as a key

Crash consistency in file system

- Crash consistency
- Journaling file system
- Log-structured file system
- Copy-on-Write file system

The problem: crash consistency

- Single file system operation updates **multiple** disk blocks
 - Classic storage device interface is block-based
 - While we can submit many writes, only one block is processed at a time
- System might crash in the **middle** of operation
 - Some blocks updated, some blocks not updated
 - Inconsistent file system
- After crash, file system need to be repaired
 - In order to restore **consistency** among blocks

Example: File Append

```
$ echo "hello" >> log.txt
```

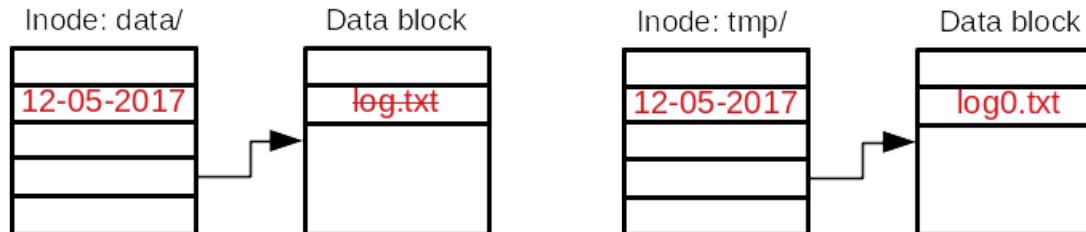
1. Allocates new data block (block bitmap)
2. Fills data block with user data (data block)
3. Records location of data block (inode)
4. Updates the last modified time (inode)



Example: Rename

```
$ mv /data/log.txt /tmp/log0.txt
```

1. Remove `log.txt` in `/data/` (directory)
2. Updates the last modified time of `/data` (inode)
3. Add `log0.txt` to `/tmp/` (directory)
4. Updates the last modified time of `/tmp` (inode)



The root cause of the problem

- **Multiple block update is not atomic in a traditional storage device**
 - Single file system operation updates **multiple** disk blocks
 - System might crash in the **middle** of operation

Solution #1: Lazy, optimistic approach

- **Fix inconsistency upon reboot** → Directly updates the data
- Advantage: simple, high performance
- Disadvantage: expensive recovery
- Example: ext2 with `fsck`

[Questions](#)

To fsck or not fsck after 180 days



By default after 180 days or some number of mounts, most Linux filesystems force a file system check (fsck). Of course this can be turned off using, for example, tune2fs -c 0 -i 0 on ext2 or ext3.

18



On small filesystems, this check is merely an inconvenience. However, given larger filesystems, this check can take hours upon hours to complete. When your users depend on this filesystem for their productivity, say it is serving their home directories via NFS, would you disable the scheduled file system check?

2

I ask this question because it is currently 2:15am and I'm awaiting a very long fsck to complete (ext3)!

Solution #2: Eager, pessimistic approach

- **Maintains the copy of data**
- Advantage: quick recovery
- Disadvantage: perpetual performance penalty
- Examples
 - Logging: ext3/4, XFS
 - Copy-on-write: btrfs, ZFS, WAFL
 - Log-structured writing: F2FS

Logging: UNDO vs. REDO

	UNDO logging	REDO logging
AKA	Rollback journaling	Write-ahead logging (WAL), journaling
Logging	How to undo a change (old copy)	How to reproduce a change (new copy)
Read	Read from original location	Read from log if there is a new copy

Journaling file system

- Use REDO logging to achieve atomic update despite crash
 - REDO logging = write-ahead logging (WAL) = journaling
- Record information about pending update (logging or journaling)
- If crash occurs during update, just replay what is in log to repair (recovery)
- **Turns multiple writes into atomic action**

Basic protocol

- **Logging:** write to journal
 - Record what is about to be written
- **Checkpointing:** write to main structures
 - Record information to final locations
- If crash occurs, **recover** from log
 - Replay log entries and finish update

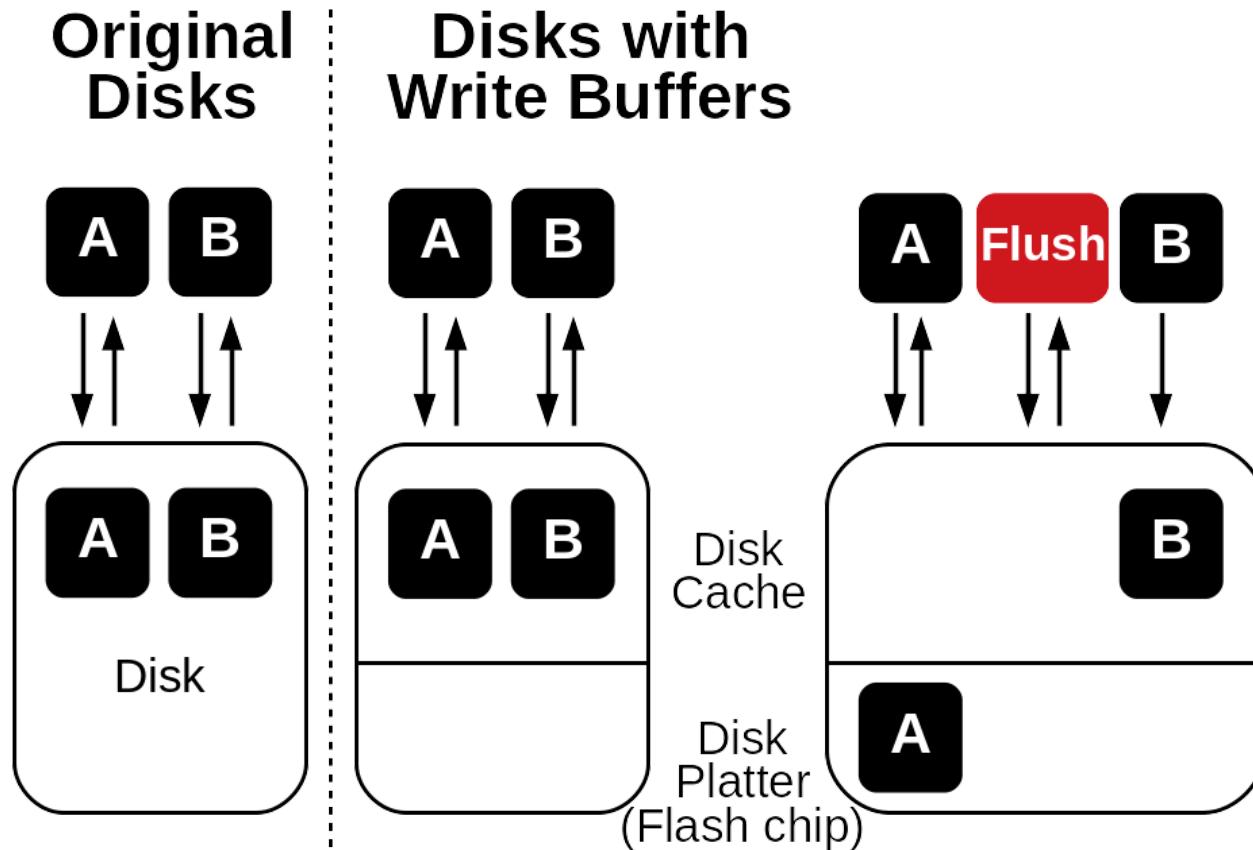
Basic (Buggy) Journaling protocol

- **Write protocol**
 - Transaction begin + contents
 - Transaction **commit**
 - **Checkpoint** contents
- **Read protocol**
 - **Design 1)** If a block is in the log, read the log. Otherwise, read contents from the original location.
 - **Design 2)** Prevent eviction of journaled pages until checkpointing.
Read contents from page cache or the original location.

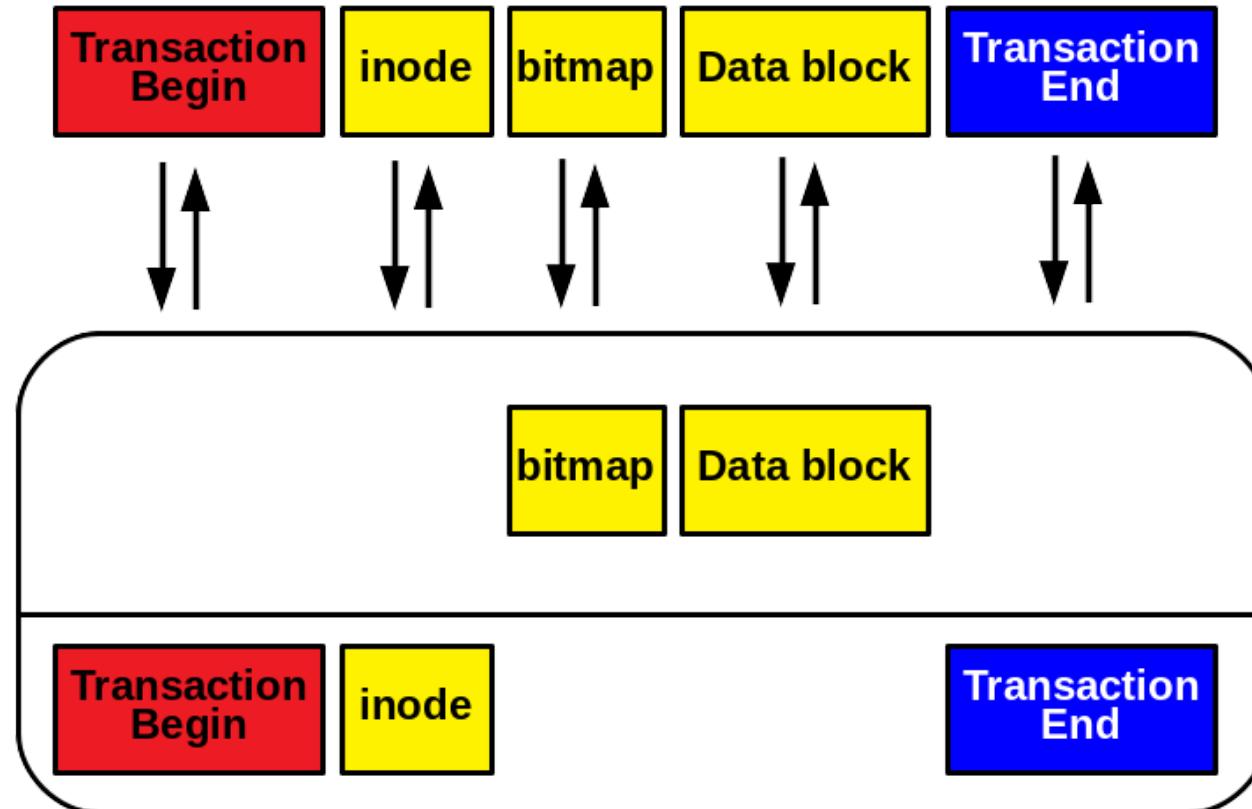
Modern disk caches

- **Disk caches:** critical for performance
 - cache tracks on reads
 - Buffer writes before committing to physical storage medium
 - HDD, SSD, SMR drive, etc.
- Example: why buffering matters
 - Performance: **factor of 2x** or greater (for some workloads)

How writes are ordered



(Buggy) Journaling protocol with disk cache

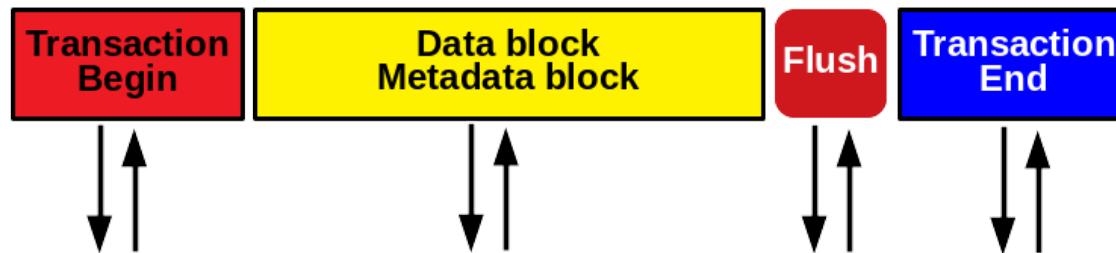


Correct journaling protocol

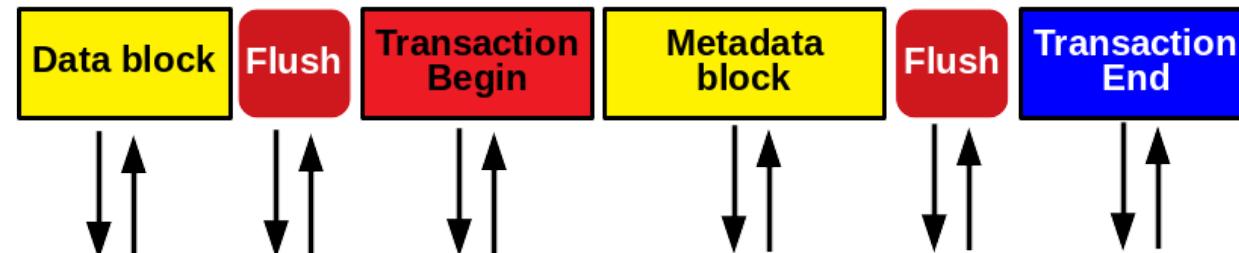
- Write protocol
 - (Optional) Disk cache flush (write barrier)
 - Transaction begin + contents
 - Disk cache flush (write barrier)
 - Transaction **commit**
 - **Checkpoint** contents

Correct journaling protocol

Data journaling



Ordered journaling



Journaling file system in Linux

- Almost all major file systems in Linux
- ext3, ext4, XFS

Log-structured file system (LFS)

- “The Design and Implementation of a Log-Structured File System”
 - Mendel Rosenblum and John K. Ousterhout
 - ACM TOCS 1992
- Key idea: **treats entire disk space as a log**

Motivation

- Most systems now have large enough memory to cache disk blocks to hold recently-accessed blocks
- Most reads are thus satisfied from the page cache
- From the point of view of the disk, most traffic is write traffic
- So to speed up disk I/O, we need to make writes go faster
- But disk performance is limited ultimately by disk head movement (seek time)
- With current file systems, adding a block takes several writes (to the file and to the metadata), requiring several disk seeks

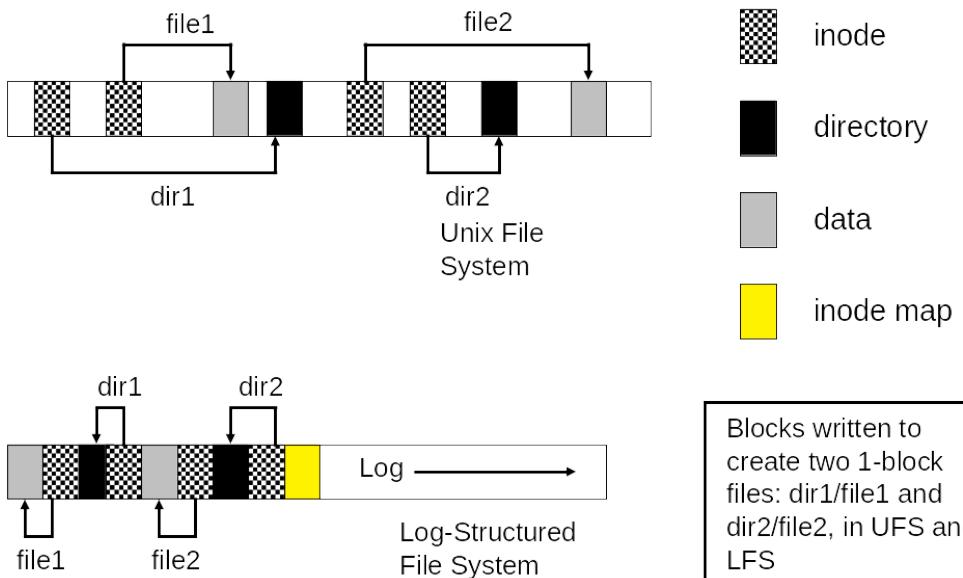
LFS: key idea

- An alternative is to use the disk as a **log**
- A log is a data structure that is written only at the head
- If the disk were managed as a log, there would be effectively no head seeks for write
- The **file** is always added to **sequentially**
- New data and metadata (inodes, directories) are accumulated in the buffer cache, then written all at once in large blocks (e.g., **segments** of .5M or 1M)
- This would greatly increase disk throughput

LFS data structures

- **inode**: as in UNIX, an inode maintain file offset to disk block mapping
- **inode map**: a table indicating where each inode is on the disk
 - inode map blocks are written as part of the segment
- **segment**: fixed size large chunk of block (e.g., .5M or 1M)
- **segment summary**: information on every block in a segment
- **segment usage table**: information on the amount of live data in a block
- **checkpoint region**: locates blocks of inode map and segment usage table, identifies last checkpoing in logl ; *located in a fixed location*
- **superblock**: static configuration of LFS; *located in a fixed location*

LFS vs. UFS



LFS read

- Reads are not different than in Unix File System, once we find the inode for a file
 - checkpoint region in a fixed location → inode map → inode
 - inode: file offset → disk block mapping

LFS write

- Every write causes new blocks to be added to the current segment buffer in memory
 - When that segment is full, it is written to the disk
- Overwrite makes the overwritten, previous block obsolete
 - Live block \leftarrow newly written block
 - Dead block \leftarrow obsolete, overwritten block
- Over time, segments in the log become fragmented as we replace old blocks of files with new block
- In steady state, we need to have contiguous free space in which to write

Segment cleaning (or garbage collection)

- The major problem for a LFS is *cleaning*, i.e., producing contiguous free space on disk
- A **cleaner process** cleans old segments, i.e., takes several non-full segments and compacts them, creating one full segment, plus free space

How to choose victim segments to clean

- The cleaner choose segments on disk based on:
- **Utilization of a segment**
 - Higher utilization (more live blocks) → larger copy overhead
- **Age**
 - Recently written segment → more likely to change soon

Crash consistency in LFS

- Until disk block is cleaned by the cleaner process, the update history of a disk block is remained
- No special consistency guarantee mechanism is needed

Log-structured file system in Linux

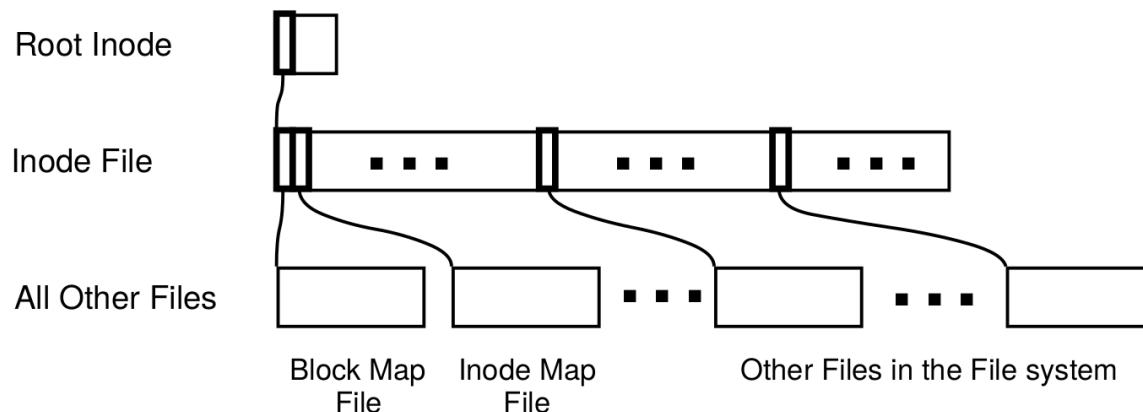
- **F2FS (Flash-Friendly File System)**
 - Optimized log-structured file system for NAND flash SSD
 - Used for mobile devices
- **NILFS**
 - Continuous snapshot file system
- **LFS design is widely adopted other than file systems**
 - Flash Translation Layer (FTL), which is a firmware for NAND flash SSD hiding complexity of NAND flash medium, is a kind of LFS managing a single file

Copy-on-Write (CoW) file system

- “File System Design for an NFS File Server Appliance”
 - Dave Hitz, James Lau, and Michael Malcolm, USENIX Winter 1994
 - **Write-Anywhere File Layout (WAFL)**: the core design of NetApp storage appliance
- **Inspired by LFS**
 - Never overwrite a block like LFS
 - No segment cleaning unlike LFS
- **Key idea**
 - Represent a filesystem as a single tree; never overwrite blocks (CoW)

WAFL layout: a tree of blocks

- **A root inode:** root of everything
- **An inode file:** contains all inodes
- **A block map file:** indicates free blocks
- **An inode map file:** indicates free inodes



Why keeping metadata in files

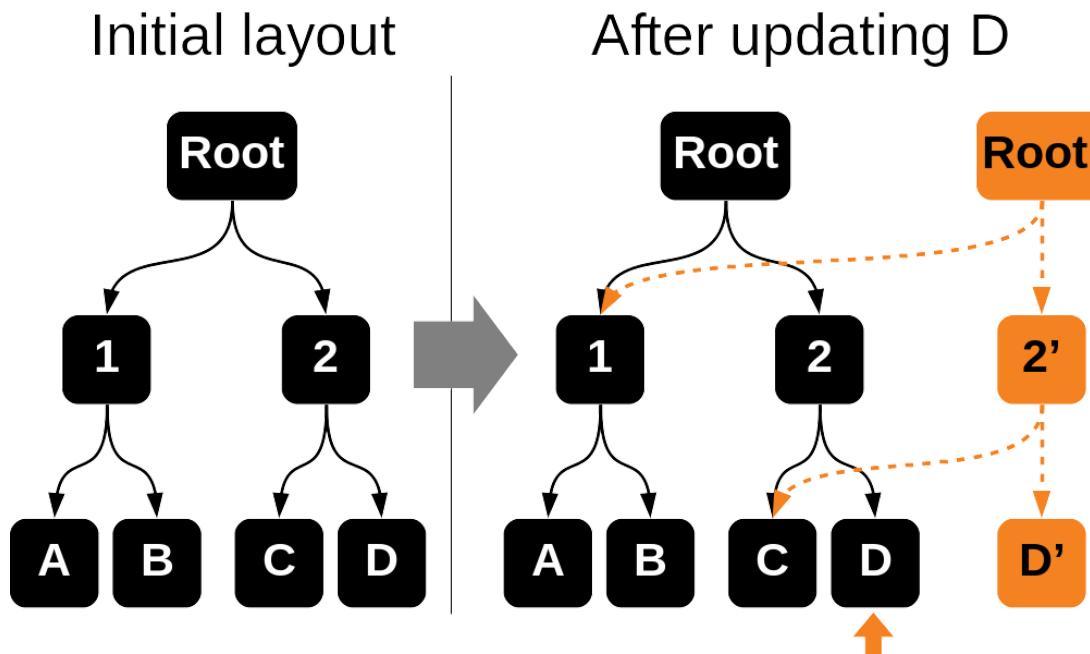
- **Allow meta-data blocks to be written anywhere on disk**
 - This is the origin of “Write Anywhere File Layout”
- **Easy to increase the size of the file system dynamically**
 - Add a disk can lead to adding i-nodes
- **Enable copy-on-write to create snapshots**
 - Copy-on-write new data and metadata on new disk locations
 - Fixed metadata locations are cumbersome

WAFL read

- Reads are not different than in Unix File System, once we find the inode for a file
 - root inode → inode file → inode
 - inode: file offset → disk block mapping

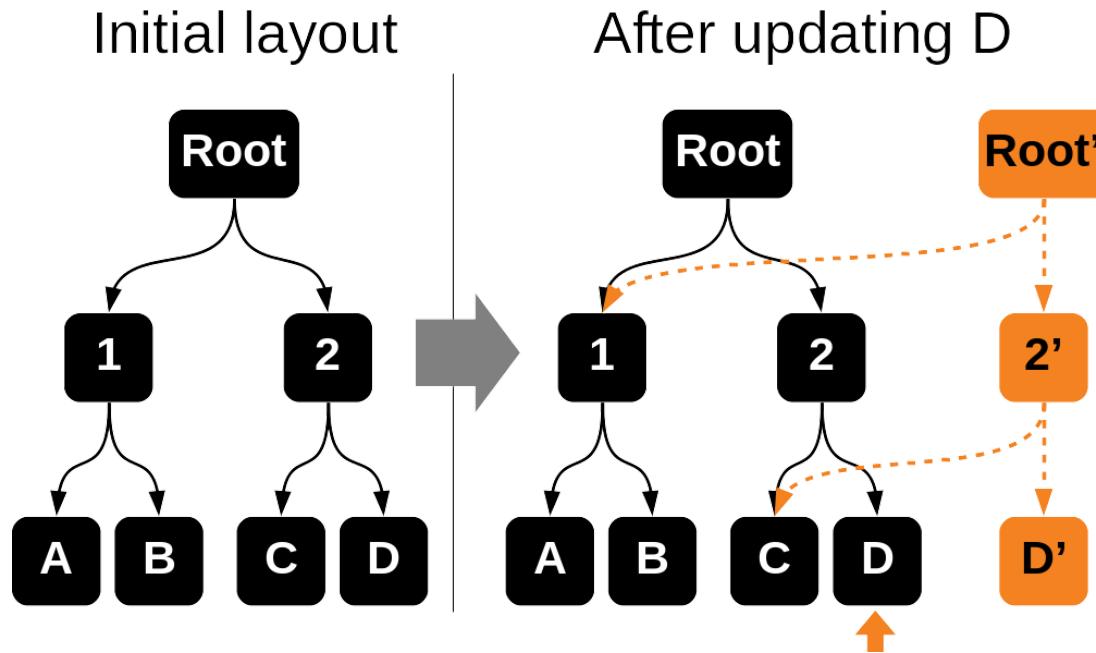
WAFL write

- WAFL filesystem is a tree of blocks
- Never overwrite blocks → Copy-on-Write



Crash consistency in LFS

- Each root inode represents a consistent snapshot of a file system



Copy-on-Write file system in Linux

- **btrfs (b-tree file system)**
 - A file system is a tree of four CoW-optimized B-trees
- **ZFS**
 - Default file system of Solaris

Further readings I

- [Ext4: The Next Generation of Ext2/3](#)
- [Outline of Ext4 File System & Ext4 Online Defragmentation Foresight](#)
- [Speeding up file system checks in ext4](#)
- [Ext4 Disk Layout](#)
- [Chapter 9. The Extended Filesystem Family in Professional Linux Kernel Architecture](#)
- [Chapter 18. The Ext2 and Ext3 Filesystems in Understanding Linux Kernel](#)
- [An introduction to Linux's EXT4 filesystem](#)

Further readings II

- What Used To Be Right Is Now Wrong
- Optimistic crash consistency
- Consistency Without Ordering
- Lightweight Application-Level Crash Consistency on Transactional Flash Storage
- SFS: Random Write Considered Harmful in Solid State Drives

Further readings III

- [The design and implementation of a log-structured file system](#)
- [File System Design for an NFS File Server Appliance](#)
- [How Do SSDs Work?](#)
- [Wikipedia: disk buffer](#)

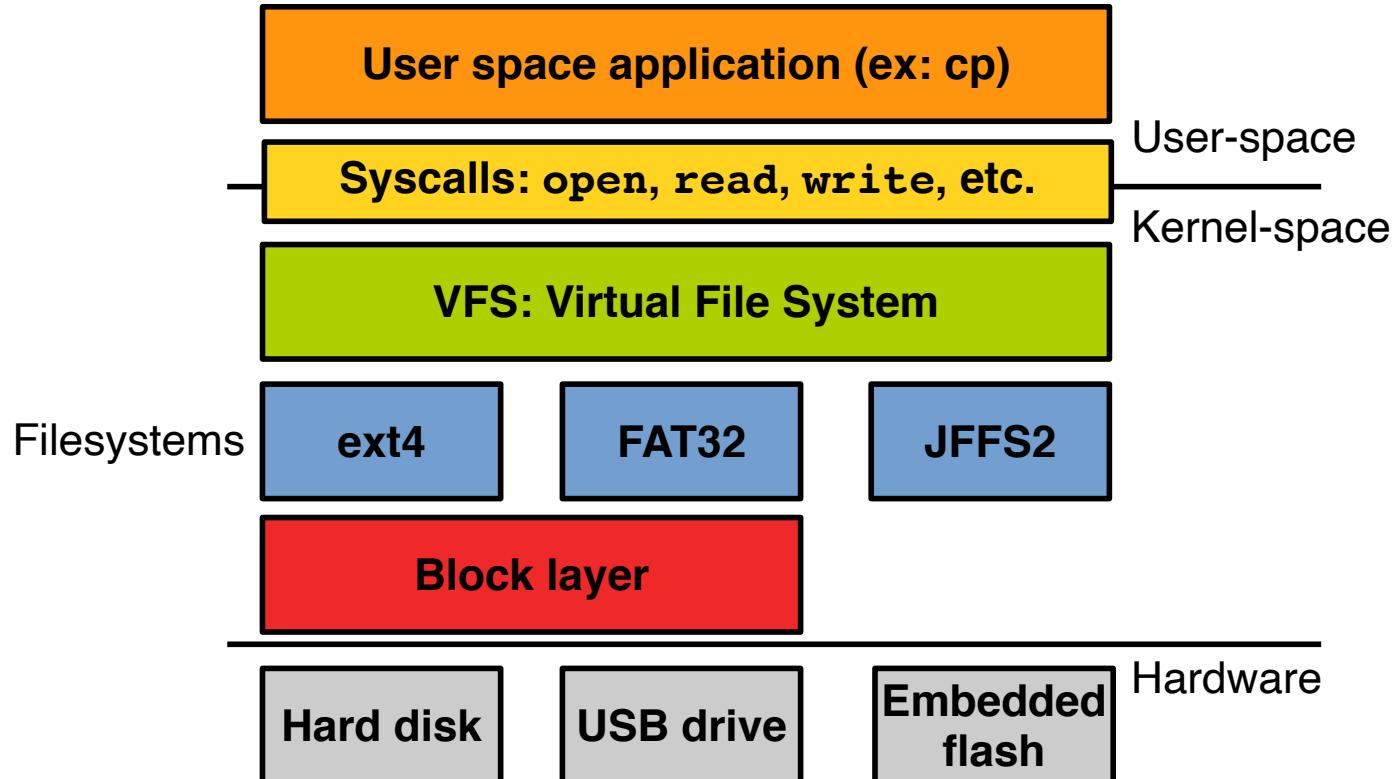
The Block IO Layer

Dongyoon Lee

Summary of last lectures

- Tools: building, exploring, and debugging Linux kernel
- Core kernel infrastructure
- Process management & scheduling
- Interrupt & interrupt handler
- Kernel synchronization
- Memory management
- Virtual file system
- Page cache and page fault
- Ext4 file system and crash consistency

Today: block IO layer



Today: block IO layer

- Block devices and the block layer
- Buffers and buffer heads
- The `bio` structure and request queues
- IO schedulers

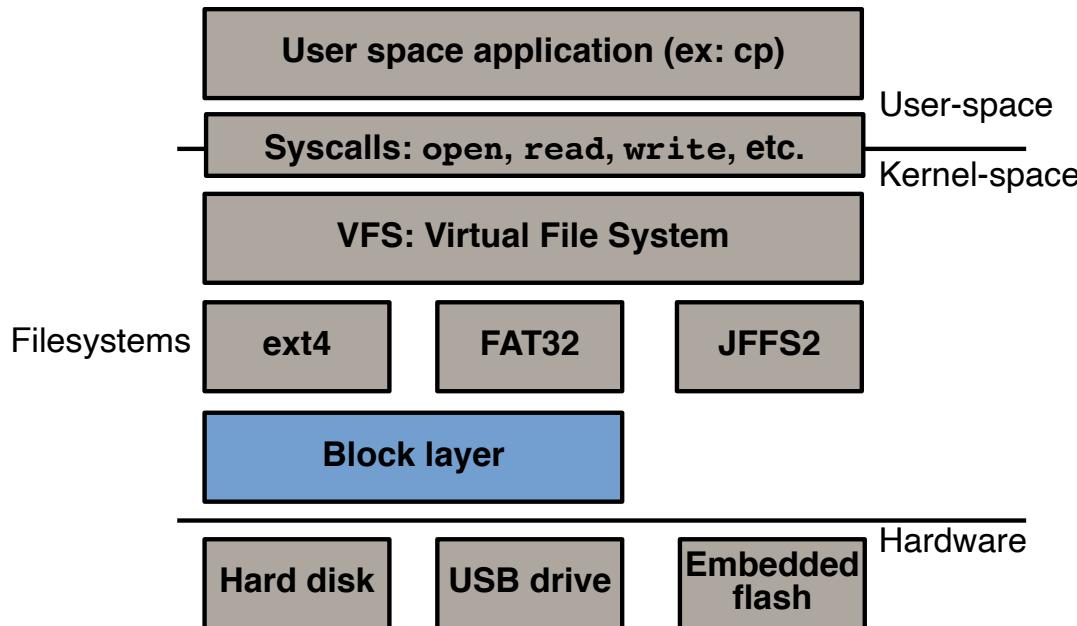
Types of device in Linux kernel



- **Character device:** serial port, mouse, keyboard, etc.
 - Accessed sequentially as a stream of bytes
 - Stream access: typing `test` on a keyboard → 't','e','s','t'
 - Relatively simple
- **Block device:** HDD, SSD, CD/DVD, floppy disks, etc.
 - Random access: device can `seek` to a specific position
 - More complex and performance critical

Block devices and block layer

- More complex and performance critical → block layer



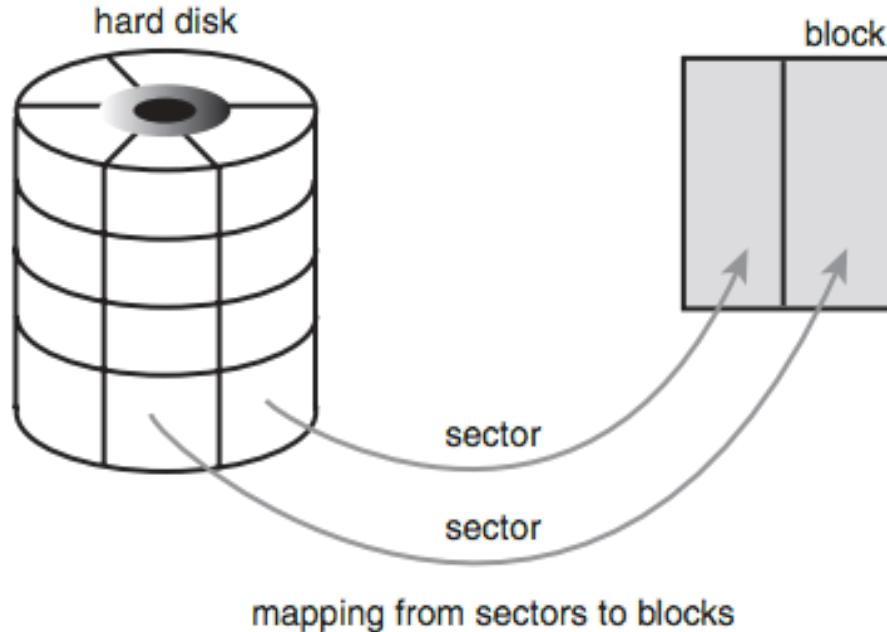
Linux block layer

- BIO layer
- Request layer
- IO scheduler

Anatomy of a block device

- **Sector**
 - Minimum addressable unit in a block device
 - Physical property of the device → hard sector, device block
 - Generally 512 bytes (2K for CD-ROM)
- **Block**
 - Unit of filesystem access → filesystem block, IO block
 - Multiple of a sector (device limitation) and multiple of a page (kernel limitation)
 - Mostly 4 KB or 8 KB

Anatomy of a block device



Buffers and buffer heads

- **Buffer:** blocks are stored in memory
- **Buffer head:** metadata of a buffer

```
/* linux/include/linux/buffer_head.h */
struct buffer_head {
    unsigned long          b_state;           /* buffer state flags */
    struct buffer_head*   *b_this_page;        /* list of page's buffers */
    struct page*           *b_page;            /* associated page */
    sector_t                b_blocknr;          /* starting block number */
    size_t                  b_size;             /* size of mapping */
    char*                  *b_data;             /* pointer to data within the page */
    struct block_device*   *b_bdev;             /* associated block device */
    bh_end_io_t*           *b_end_io;           /* I/O completion */
    void*                  *b_private;          /* reserved for b_end_io */
    struct list_head        b_assoc_buffers;    /* associated mappings */
    struct address_space*  *b_assoc_map;        /* associated address space */
    atomic_t                 b_count;            /* use count: get_bh(), put_bh() */
}
```

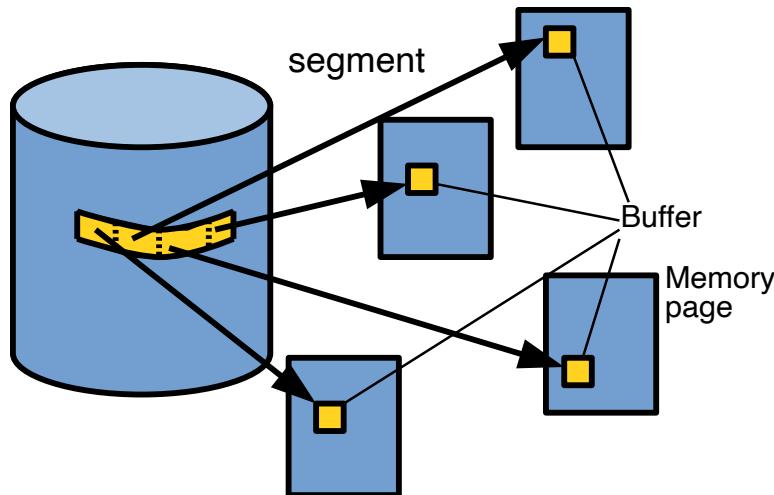
Buffer state: **b_state**

```
/* linux/include/linux/buffer_head.h*/
enum bh_state_bits {
    BH_Uptodate, /* Contains valid data */
    BH_Dirty,    /* Is dirty */
    BH_Lock,     /* Is locked */
    BH_Req,      /* Has been submitted for I/O */
    BH_Uptodate_Lock, /* Used by the first bh in a page, to serialise
                       * IO completion of other buffers in the page */
    BH_Mapped,   /* Has a disk mapping */
    BH_New,      /* Disk mapping was newly created by get_block */
    BH_Async_Read, /* Is under end_buffer_async_read I/O */
    BH_Async_Write, /* Is under end_buffer_async_write I/O */
    BH_Delay,    /* Buffer is not yet allocated on disk */
    BH_Boundary, /* Block is followed by a discontiguity */
    BH_Write_EIO, /* I/O error on write */
    BH_Unwritten, /* Buffer is allocated on disk but not written */
    BH_Quiet,    /* Buffer Error Prinks to be quiet */
    BH_Meta,     /* Buffer contains metadata */
    BH_Prio,     /* Buffer should be submitted with REQ_PRIO */
    BH_Defer_Completion, /* Defer AIO completion to workqueue */

    BH_PrivateStart, /* not a state bit, but the first bit available
                      * for private allocation by other entities */
};
```

The **bio** structure

- Basic container for an active *block I/O operation*
- An individual buffer being divided into segments, it needs not to be contiguous in memory



The **bio** structure

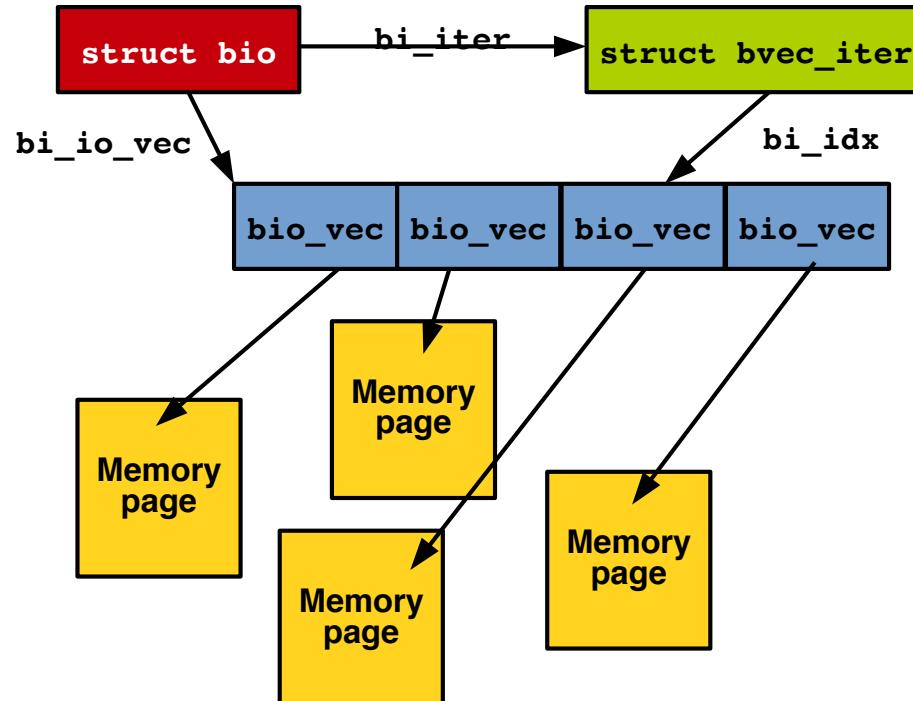
```
/* linux/include/linux/blk_types.h */
struct bio {
    struct bio          *bi_next;           /* list of requests */
    struct block_device *bi_bdev;           /* associated block device */
    unsigned short       bi_flags;           /* status and command flags */
    unsigned int         bi_phys_segments; /* number of segments */
    struct bvec_iter    bi_iter;            /* vector iterator */
    unsigned int         bi_seg_front_size; /* size of front segment */
    unsigned int         bi_seg_back_size; /* size of last segment */
    bio_end_io_t         *bi_end_io;          /* I/O completion method */
    void                *bi_private;         /* owner private data */
    unsigned short       bi_vcnt;            /* number of bio_vecs */
    unsigned short       bi_max_vecs;        /* maximum bio_vecs possible */
    atomic_t             __bi_cnt;           /* usage counter */
    struct bio_vec       *bi_io_vec;          /* bio_vec list */
    struct bio_vec       bi_inline_vecs[0]; /* inline bio vectors */
    /* ... */
};
```

The **bio** structure

```
/* linux/include/linux/bvec.h */
struct bvec_iter {
    sector_t      bi_sector; /* target address on the device in sectors */
    unsigned int   bi_size;   /* I/O count */
    unsigned int   bi_idx;   /* current index into bi_io_vec */
    /* ... */
};

/* linux/include/linux/bio.h */
struct bio_vec {
    /* pointer to the target physical page: */
    struct page *bv_page;
    /* length in bytes of the buffer: */
    unsigned int   bv_len;
    /* offset inside the page where the buffer resides: */
    unsigned int   bv_offset;
};
```

The **bio** structure



Request queues

- Block devices maintain **request queues** to store pending I/O requests
- Request queues are represented by the `request_queue` structure defined in `include/linux/blkdev.h`
- Requests are added to the queue by high-level code, such as a file system
- Requests are pulled from the queue by the block device driver and submitted to the device

```
struct request_queue {  
    /* Together with queue_head for cacheline sharing */  
    struct list_head    queue_head;  
    struct request     *last_merge;  
    struct elevator_queue *elevator;  
    /* ... */  
};
```

Request queues

- A single request:
 - Represented by `struct request`
 - Can operate on multiple consecutive disk blocks, so it is composed of *one or more* `bio` objects

```
struct request {
    struct list_head queuelist;
    union {
        struct call_single_data csd;
        u64 fifo_time;
    };
    struct request_queue *q;
    struct blk_mq_ctx *mq_ctx;
    /* ... */
};
```

IO schedulers

- Directly sending requests to the disk as they arrive is sub-optimal:
 - Increase random accesses
 - The kernel tries to reduce **disk seek** as much as possible
- The kernel combines and re-order I/O requests in the request queue:
 - **merging, sorting**
- Rules for merging and sorting are defined by the **I/O scheduler**
 - Multiple I/O scheduler models implemented in Linux
- The I/O scheduler virtualizes the disk as the process scheduler virtualizes the CPU

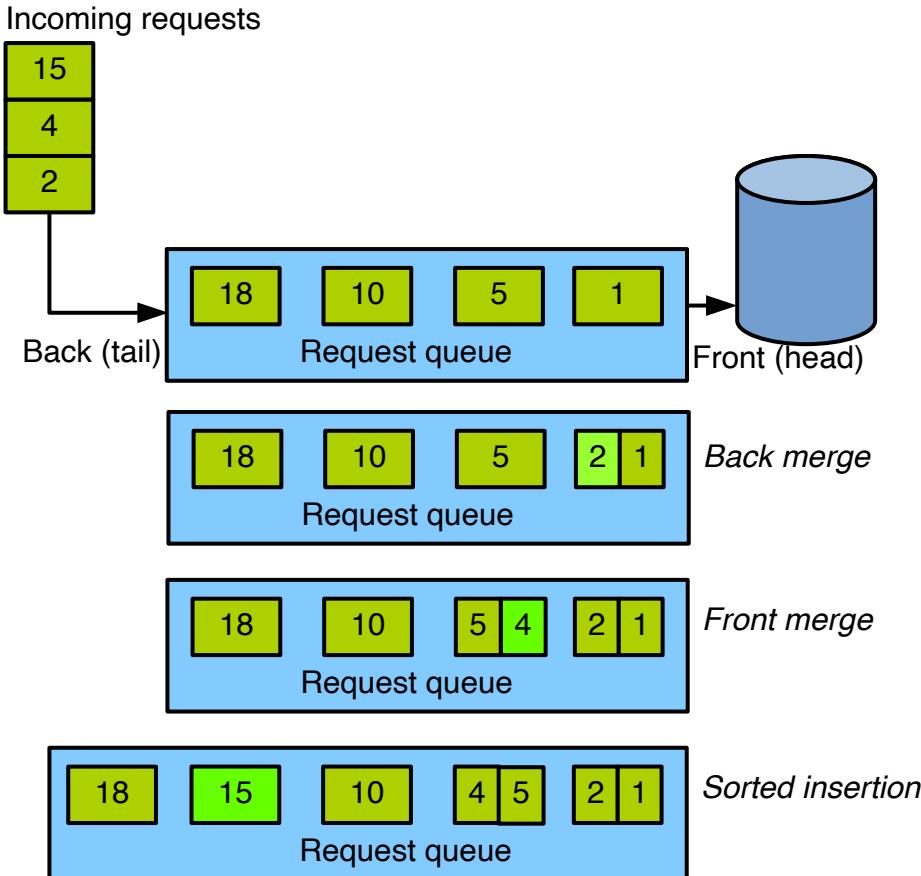
Linus elevator

- Default IO scheduler until Kernel 2.4
- Define where an upcoming request should be added into the queue:
 - front merge, back merge
 - sorted insertion
- Goal: minimize disk seek, best global throughput

Linus elevator

1. If a request to an adjacent on-disk sector is in the queue, the existing request and the new request merge into a single request.
2. If a request in the queue is sufficiently old, the new request is inserted at the tail of the queue to prevent starvation of the other, older, requests.
3. If a suitable location sector-wise is in the queue, the new request is inserted there. This keeps the queue sorted by physical location on disk.
4. Finally, if no such suitable insertion point exists, the request is inserted at the tail of the queue.

Linus elevator

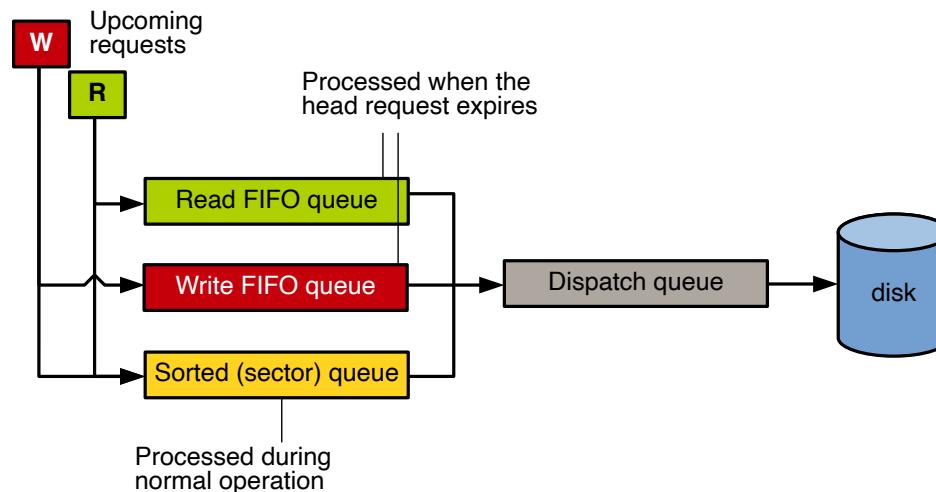


Problems with Linus Elevator

- Goal: minimize disk seek, best global throughput
 - Can cause starvation
- **Write starving reads**
 - Buffer IO operation with buffer page cache
 - Write operations are buffered to page cache → asynchronous
 - Read operations upon page cache miss should be immediately handled → synchronous
 - Read latency is important for the system → *read starvation must be minimized*

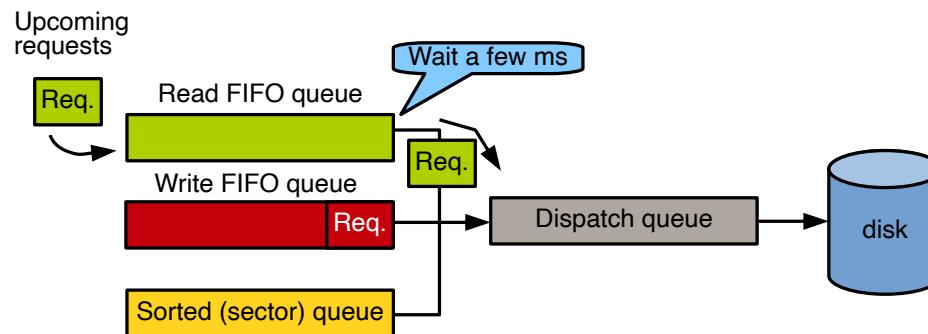
The deadline IO scheduler

- Tries to provide fairness while maximizing the global throughput
- Each request is given an expiration time, the deadline:
 - Reads = now + .5s, Writes = now + 5s



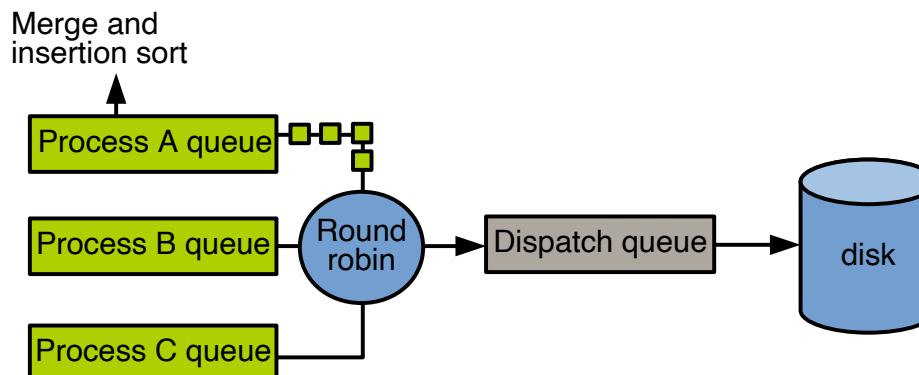
The anticipatory IO scheduler

- Tries to improve the throughput of the deadline scheduler
- Anticipation heuristic
 - Instead of immediately seeking back, it waits for a few milliseccons hoping an application sends other IO requests.



The complete fair queuing (CFO) IO scheduler

- Default IO scheduler
- Per-process request queues
- Serves the queues round robin



The noop IO scheduler

- Does not perform anything in particular apart from merging sequential request
- Used for truly random devices such as flash cards

Configuring IO scheduler

- I/O scheduler can be selected at boot time as a kernel parameter:-
`elevator=<value>` - `value` could be either of `cfq` , `deadline` , or `noop`
- Or you can choose an IO scheduler per device

```
$> cat /sys/block/<block device name>/queue/scheduler  
$> echo noop /sys/block/<block device name>/queue/scheduler
```

Adding a new IO scheduler

```
/* linux/include/linux/elevator.h */
struct elevator_type
{
    /* managed by elevator core */
    struct kmem_cache *icq_cache;

    /* fields provided by elevator implementation */
    union {
        struct elevator_ops sq;
        struct elevator_mq_ops mq;
    } ops;
    size_t icq_size;      /* see iocontext.h */
    size_t icq_align;    /* ditto */
    struct elv_fs_entry *elevator_attrs;
    char elevator_name[ELV_NAME_MAX];
    struct module *elevator_owner;
    bool uses_mq;

    /* managed by elevator core */
    char icq_cache_name[ELV_NAME_MAX + 6]; /* elvname + "_io_cq" */
    struct list_head list;
};
```

Adding a new IO scheduler

```
/* linux/include/linux/elevator.h */
struct elevator_ops
{
    elevator_merge_fn *elevator_merge_fn;
    elevator_merged_fn *elevator_merged_fn;
    elevator_merge_req_fn *elevator_merge_req_fn;
    elevator_allow_bio_merge_fn *elevator_allow_bio_merge_fn;
    elevator_allow_rq_merge_fn *elevator_allow_rq_merge_fn;
    elevator_bio_merged_fn *elevator_bio_merged_fn;

    elevator_dispatch_fn *elevator_dispatch_fn;
    /* ... */
};
```

Further readings

- [LWN: A block layer introduction part 1: the bio layer](#)
- [LWN: A block layer introduction part 2: the request layer](#)
- [Linux Block IO: Introducing Multi-queue SSD Access on Multi-core Systems, SYSTOR13](#)
- [LWN: The multiqueue block layer](#)
- [LWN: Two new block I/O schedulers for 4.12](#)
- [LWN: The future of DAX](#)
- [Kernel Recipes 2017 - What's new in the world of storage for linux - Jens Axboe](#)

Memory Consistency Model

Ack: Prof. Sarita Adve, UIUC

Memory Consistency Model: Definition

Memory consistency model

Order in which memory operations will appear to execute
⇒ What value can a read return?

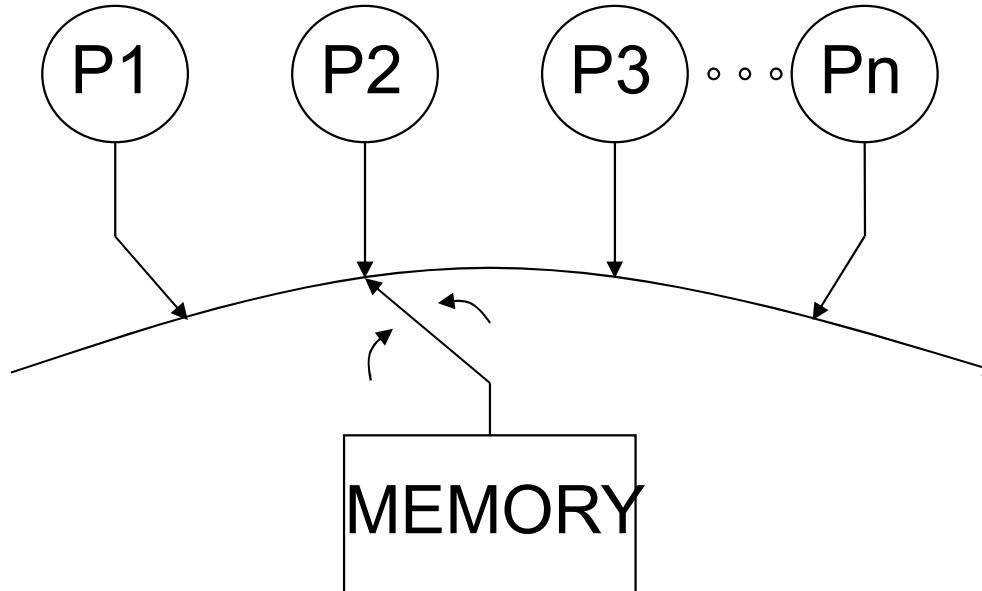
Affects ease-of-programming and performance

Implicit Memory Model

Sequential consistency (SC) [Lamport]

Result of an execution appears as if

- All operations executed in some **sequential order**
- Memory operations of each process in **program order**



No caches, no write buffers

Implicit Memory Model

Sequential consistency (SC) [Lamport]

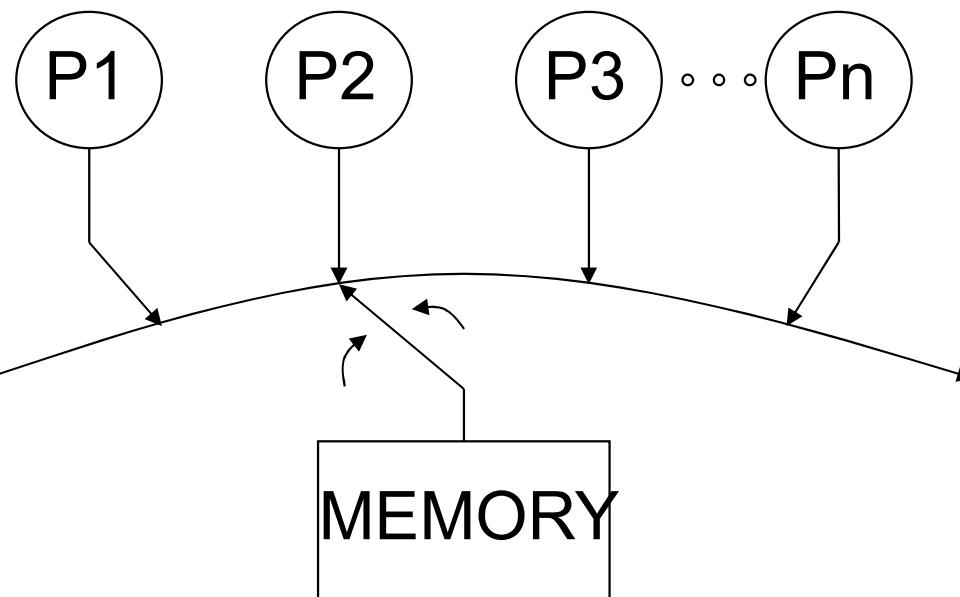
Result of an execution appears as if

- All operations executed in some sequential order
- Memory operations of each process in program order

Two aspects:

Program order

Atomicity



No caches, no write buffers

Understanding Program Order – Example 0

Initially X = 2

P1

.....

r0=Read(X)

r0=r0+1

Write(r0,X)

.....

P2

.....

r1=Read(x)

r1=r1+1

Write(r1,X)

.....

Possible execution sequences:

P1:r0=Read(X)

P2:r1=Read(X)

P1:r0=r0+1

P1:Write(r0,X)

P2:r1=r1+1

P2:Write(r1,X)

x=3

P2:r1=Read(X)

P2:r1=r1+1

P2:Write(r1,X)

P1:r0=Read(X)

P1:r0=r0+1

P1:Write(r0,X)

x=4

Atomic Operations

- sequential consistency has nothing to do with **atomicity** as shown by example on previous slide
- atomicity: use atomic operations such as exchange
 - **exchange(r,M)**: swap contents of register r and location M

r0 = 1;

do exchange(r0,S) while (r0 != 0); //S is memory location
//enter critical section

.....

//exit critical section

S = 0;

Understanding Program Order – Example 1

Initially Flag1 = Flag2 = 0

P1

Flag1 = 1

if (Flag2 == 0)

critical section

P2

Flag2 = 1

if (Flag1 == 0)

critical section

Execution:

P1

(Operation, Location, Value)

Write, Flag1, 1

P2

(Operation, Location, Value)

Write, Flag2, 1

Read, Flag2, 0

Read, Flag1, _____

Understanding Program Order – Example 1

Initially Flag1 = Flag2 = 0

P1

Flag1 = 1

if (Flag2 == 0)

critical section

P2

Flag2 = 1

if (Flag1 == 0)

critical section

Execution:

P1

(Operation, Location, Value)

Write, Flag1, 1



Read, Flag2, 0

P2

(Operation, Location, Value)

Write, Flag2, 1



Read, Flag1, _____

Understanding Program Order – Example 1

Initially Flag1 = Flag2 = 0

P1

Flag1 = 1

if (Flag2 == 0)

critical section

P2

Flag2 = 1

if (Flag1 == 0)

critical section

Execution:

P1

(Operation, Location, Value)

Write, Flag1, 1



Read, Flag2, 0

P2

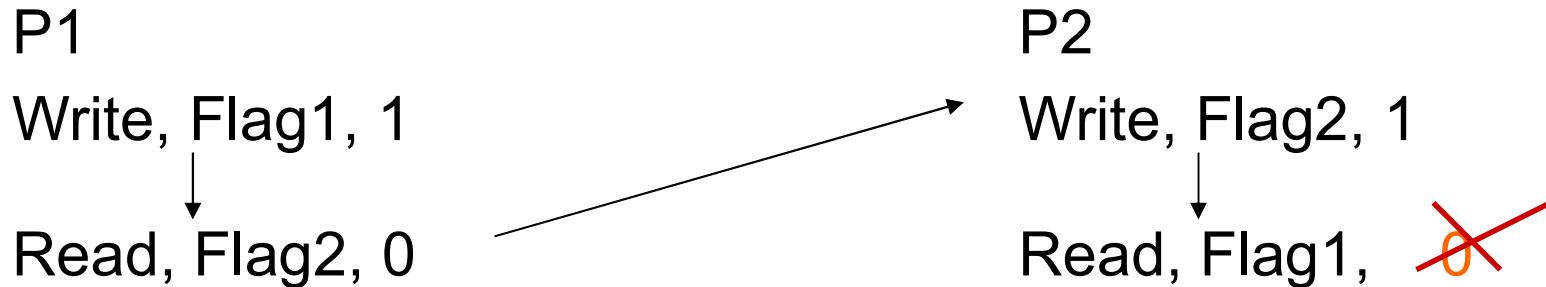
(Operation, Location, Value)

Write, Flag2, 1



Read, Flag1, ~~0~~

Understanding Program Order – Example 1



Can happen if

- Write buffers with read bypassing
- Overlap, reorder write followed by read in h/w or compiler
- Allocate Flag1 or Flag2 in registers

On AlphaServer, NUMA-Q, T3D/T3E, Ultra Enterprise Server

Understanding Program Order - Example 2

Initially A = Flag = 0

P1

A = 23;

Flag = 1;

P2

while (Flag != 1) {}

... = A;

P1

Write, A, 23

Write, Flag, 1

P2

Read, Flag, 0

Read, Flag, 1

Read, A, _____

Understanding Program Order - Example 2

Initially A = Flag = 0

P1

A = 23;

Flag = 1;

P2

while (Flag != 1) {}

... = A;

P1

Write, A, 23

Write, Flag, 1

P2

Read, Flag, 0

Read, Flag, 1

Read, A, ~~0~~

Understanding Program Order - Example 2

Initially A = Flag = 0

P1

A = 23;

Flag = 1;

P2

while (Flag != 1) {}

... = A;

P1

Write, A, 23

Write, Flag, 1

P2

Read, Flag, 0

Read, Flag, 1

Read, A, ~~0~~

Can happen if

Overlap or reorder writes or reads in hardware or compiler

On AlphaServer, T3D/T3E

Understanding Program Order: Summary

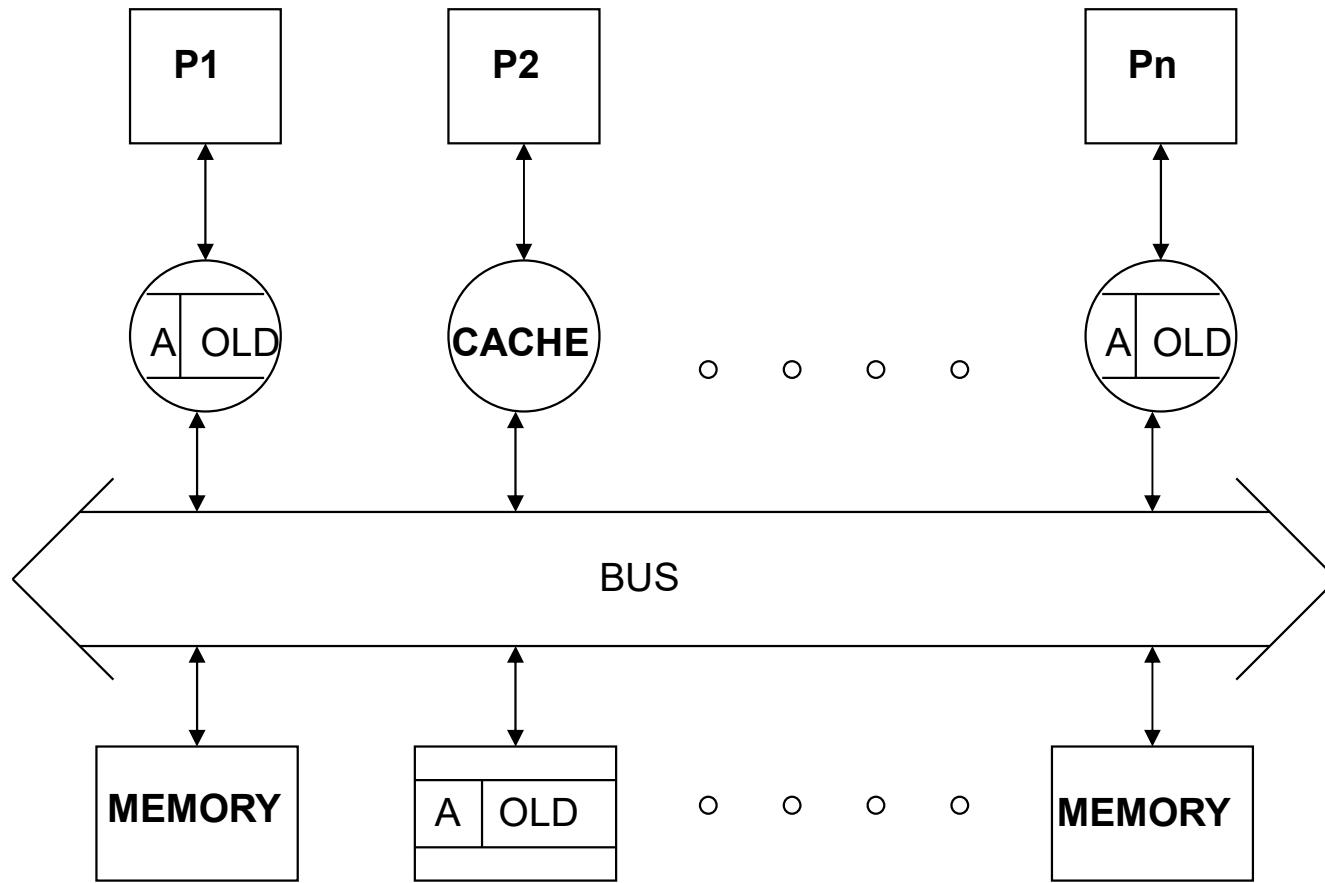
SC limits program order relaxation:

Write → Read

Write → Write

Read → Read, Write

Understanding Atomicity – Caches 101



A mechanism needed to propagate a write to other copies
⇒ Cache coherence protocol

Cache Coherence Protocols

How to propagate write?

Invalidate -- Remove old copies from other caches

Update -- Update old copies in other caches to new values

Understanding Atomicity - Example 1

Initially A = B = C = 0

P1

A = 1;

B = 1;

P2

A = 2;

C = 1;

P3

while (B != 1) {;}

while (C != 1) {;}

tmp1 = A;

P4

while (B != 1) {;}

while (C != 1) {;}

tmp2 = A;

Understanding Atomicity - Example 1

Initially A = B = C = 0

P1	P2	P3	P4
A = 1;	A = 2;	while (B != 1) {};	while (B != 1) {};
B = 1;	C = 1;	while (C != 1) {};	while (C != 1) {};
		tmp1 = A; 1	tmp2 = A; 2

Can happen if updates of A reach P3 and P4 in different order

Coherence protocol must serialize writes to same location
(Writes to same location should be seen in same order by all)

Understanding Atomicity - Example 2

Initially A = B = 0

P1

A = 1

P2

while (A != 1) ;
B = 1;

P3

while (B != 1) ;
tmp = A

P1

Write, A, 1

P2

Read, A, 1
Write, B, 1

P3

Read, B, 1
Read, A, ~~1~~

Can happen if read returns new value before all copies see it
Read-others'-write early optimization unsafe

Program Order and Write Atomicity Example

Initially all locations = 0

P1

Flag1 = 1;

P2

Flag2 = 1;

... = Flag2; 0

... = Flag1; ~~0~~

Can happen if read early from write buffer

Program Order and Write Atomicity Example

Initially all locations = 0

P1

Flag1 = 1;

A = 1;

... = A;

... = Flag2; 0

P2

Flag2 = 1;

A = 2;

... = A;

... = Flag1; 0

Program Order and Write Atomicity Example

Initially all locations = 0

P1

Flag1 = 1;

A = 1;

... = A; 1

... = Flag2; 0

P2

Flag2 = 1;

A = 2;

... = A; 2

... = Flag1; ~~0~~

Can happen if read early from write buffer

“Read-own-write early” optimization can be unsafe

SC Summary

SC limits

Program order relaxation:

Write → Read

Write → Write

Read → Read, Write

Read others' write early

Read own write early

Unserialized writes to the same location

Alternative

Give up sequential consistency

Use relaxed models

Note: Aggressive Implementations of SC

Can actually do optimizations with SC with some care

- Hardware has been fairly successful

- Limited success with compiler

But not an issue here

- Many current architectures do not give SC

- Compiler optimizations on SC still limited

Classification for Relaxed Models

Typically described as system optimizations - **system-centric**

Optimizations

Program order relaxation:

Write → Read

Write → Write

Read → Read, Write

Read others' write early

Read own write early

All models provide safety net

All models maintain uniprocessor data and control dependences,
write serialization

Some Current System-Centric Models

Relaxation:	W → R Order	W → W Order	R → RW Order	Read Others' Write Early	Read Own Write Early	Safety Net
IBM 370	✓					serialization instructions
TSO	✓				✓	RMW
PC	✓			✓	✓	RMW
PSO	✓	✓			✓	RMW, STBAR
WO	✓	✓	✓		✓	synchronization
RCsc	✓	✓	✓		✓	release, acquire, nsync, RMW
RCpc	✓	✓	✓	✓	✓	release, acquire, nsync, RMW
Alpha	✓	✓	✓		✓	MB, WMB
RMO	✓	✓	✓		✓	various MEMBARs
PowerPC	✓	✓	✓	✓	✓	SYNC

Relaxing Program Orders

- Weak ordering:
 - Divide memory operations into **data operations** and **synchronization operations**
 - Synchronization operations act like a **fence**:
 - All data operations before synch in program order must complete before synch is executed
 - All data operations after synch in program order must wait for synch to complete
 - Synchs are performed in program order
 - Implementation of fence: processor has counter that is incremented when data op is issued, and decremented when data op is completed
 - Example: PowerPC has **SYNC** instruction (caveat: semantics somewhat more complex than what we have described...)

Another model: Release consistency

- Further relaxation of weak consistency
- Synchronization accesses are divided into
 - **Acquires**: operations like lock
 - **Release**: operations like unlock
- Semantics of acquire:
 - **Acquire must complete before all following memory accesses**
- Semantics of release:
 - **all memory operations before release are complete**
 - but accesses after release in program order do not have to wait for release
 - operations which follow release and which need to wait must be protected by an acquire

Virtualization Technology

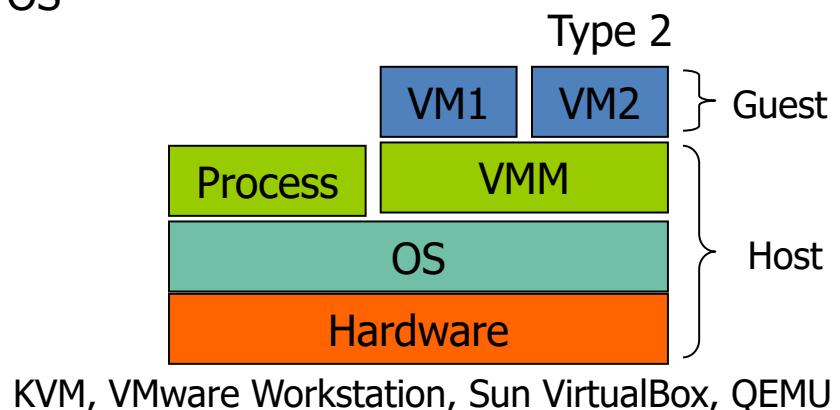
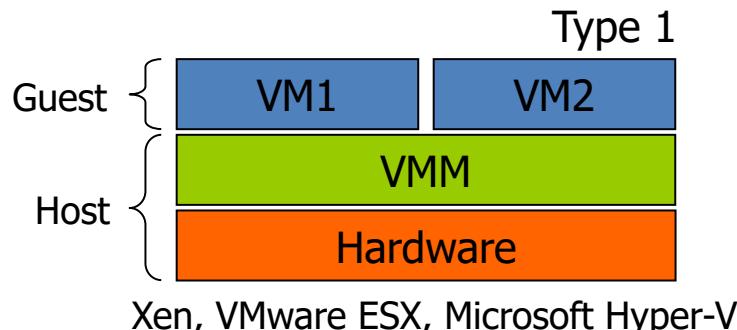
: Overview, Memory Management and I/O Virtualization

Outline

- Virtualization Overview
- Memory Management
- I/O Virtualization
- Warp-up

Introduction

- **What is virtualization?**
 - Virtualization is a way to run multiple operating systems and user applications on the same hardware
 - Virtual Machine Monitor (VMM, Hypervisor) is a software layer that allows several virtual machines to run on a physical machine
- **Types of VMMs**
 - Type-1: hypervisor runs directly on hardware
 - Type-2: hypervisor runs on a host OS



- Virtual Machine Monitor (VMM) = Hypervisor = Host OS
- Virtual Machine (VM) = Guest OS

Introduction (cont'd)

- **Advantages**
 - Isolation
 - Limits security exposures
 - Reduces spread of risks
 - Roll-Back
 - Quickly recovers from security breaches
 - Abstraction
 - Limits direct access to hardware
 - Portability
 - Disaster recovery
 - Switches to “standby” VMs
 - Deployment
 - Distributes workloads
 - Customizes guest OS security settings

Introduction (cont'd)

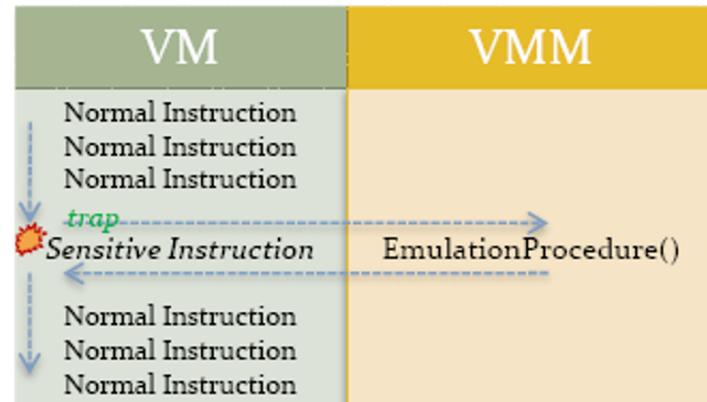
- **Applications**

- Server virtualization
 - Green IT
 - Xen, VMware ESX Server
- Desktop virtualization
 - VMware, VirtualBox, Citrix's Xen HDX
- Mobile virtualization
 - Secure execution
 - Xen on ARM
- Cloud computing
 - Storage/platform cloud services
 - Amazon EC2, MS Azure, Google AppEngine
- Emulation
 - iPhone/Android emulator
 - QEMU, Bochs



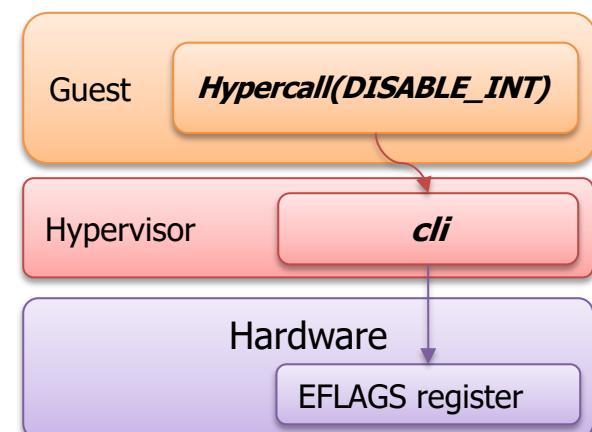
Processor Virtualization

- Classic virtualization
 - Trap & Emulate
 - For an architecture to be virtualizable, all sensitive instructions must be handled by VMM
 - Sensitive instructions include
 - Instruction that changes processor mode
 - Instruction that accesses hardware directly
 - Instruction whose behavior is different in user/kernel mode



Processor Virtualization (cont'd)

- Para-virtualization
 - Requires modifications to the guest OS
 - Guest is aware that it is running on a VM
 - Example
 - Instead of doing “cli” (turn off interrupts), guest OS should do `hypercall(DISABLE_INT)`
 - Pros
 - Near-native performance
 - No hardware support required
 - Cons
 - Requires specifically modified guest
 - Solutions : Xen



Processor Virtualization (cont'd)

- Full-virtualization
 - Emulation
 - Process of implementing the interface and functionality of one system on a different system
 - Do whatever the CPU does, but in software
 - CPU emulation
 - Fetches and decodes the next instruction
 - Executes using the emulated registers and memory
 - Pros
 - No hardware support required
 - Simple
 - Cons
 - Very slow
 - Solutions : Bochs

```
addl %ebx, %eax
```

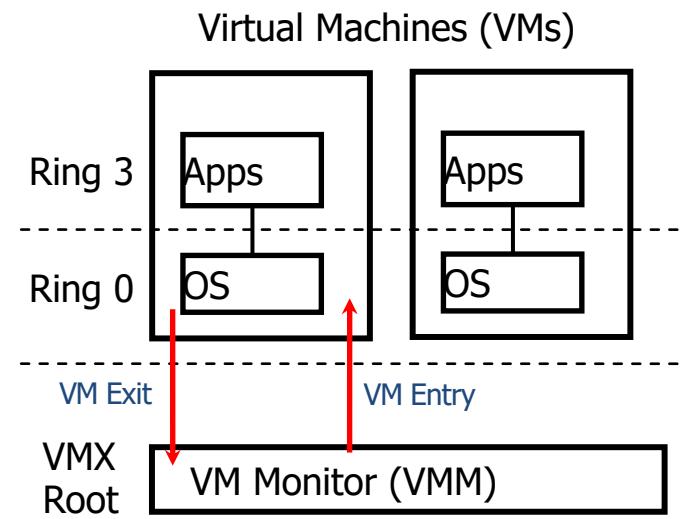
```
unsigned long regs[8];  
regs[EAX] +=  
regs[EBX];
```

Processor Virtualization (cont'd)

- Full-virtualization
 - Binary translation
 - Translates code block to safe code block (like JIT) directly
 - Dynamically translates privileged instructions to normal instructions which can be executed in user mode
 - Pros
 - No hardware support required
 - Fast
 - Cons
 - Hard to implement
 - » VMM needs x86-to-x86 binary compiler
 - Solutions : VMware, QEMU

Processor Virtualization (cont'd)

- Full-virtualization
 - Hardware-assisted virtualization
 - Runs the VM directly on the CPU
 - No emulation
 - Integrates new execution mode into the CPU by extending the instruction set and control structure
 - Pros
 - Fast
 - Cons
 - Need hardware support
 - » AMD SVM
 - » Intel VT
 - Solutions : KVM, Xen



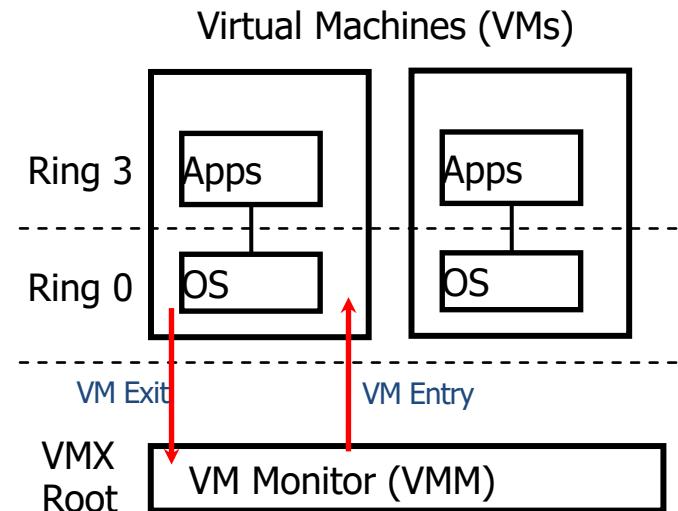
AMD SVM : Secure Virtual Machine

Intel VT : Virtualization Technology



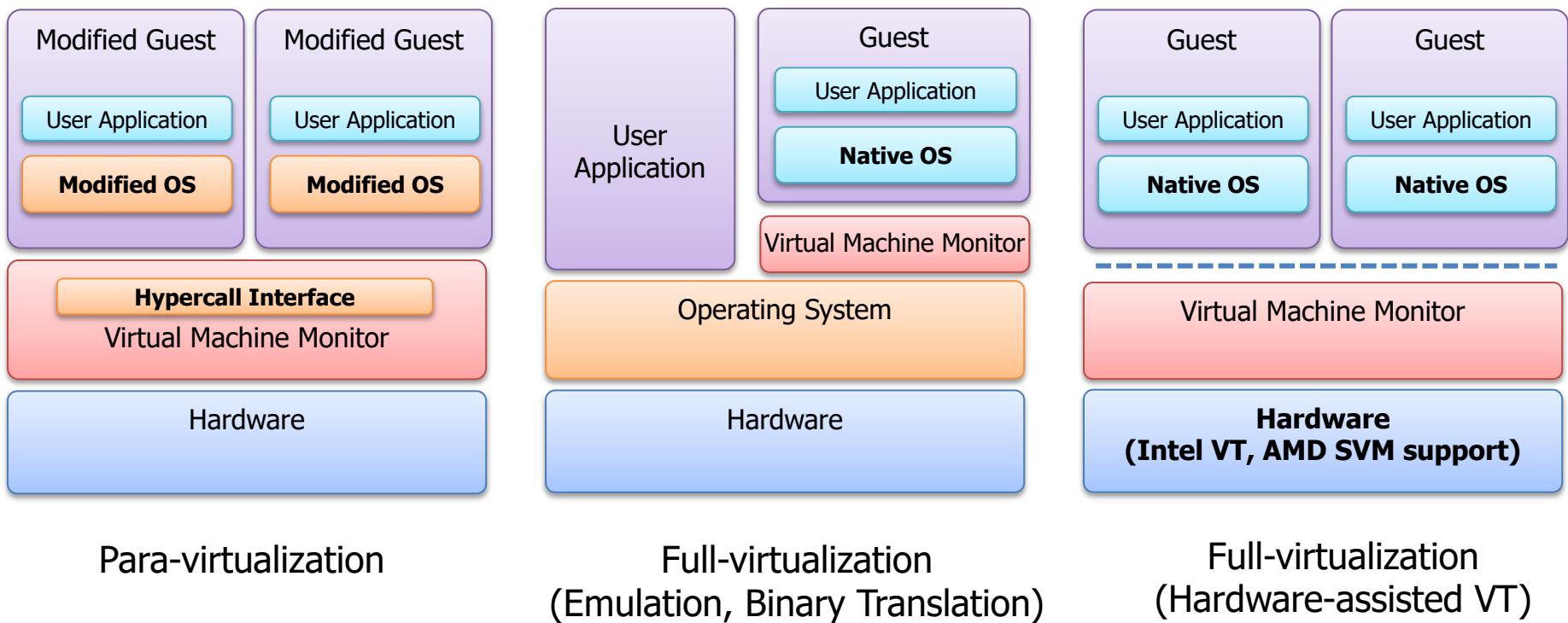
Intel VMX

- VMX(Virtual Machine Extension) supports virtualization of processor hardware.
- Two new VT-x operating modes
 - Less-privileged mode (VMX non-root) for guest OSes
 - More-privileged mode (VMX root) for VMM
- Two new transitions
 - VM entry to non-root operation
 - VM exit to root operation
- Execution controls determine when exits occur
 - Access to privilege state, occurrence of exceptions, etc.
 - Flexibility provided to minimize unwanted exits
- VM Control Structure (VMCS) controls VMX operation
 - Also holds guest and host state



Processor Virtualization (cont'd)

- Comparison

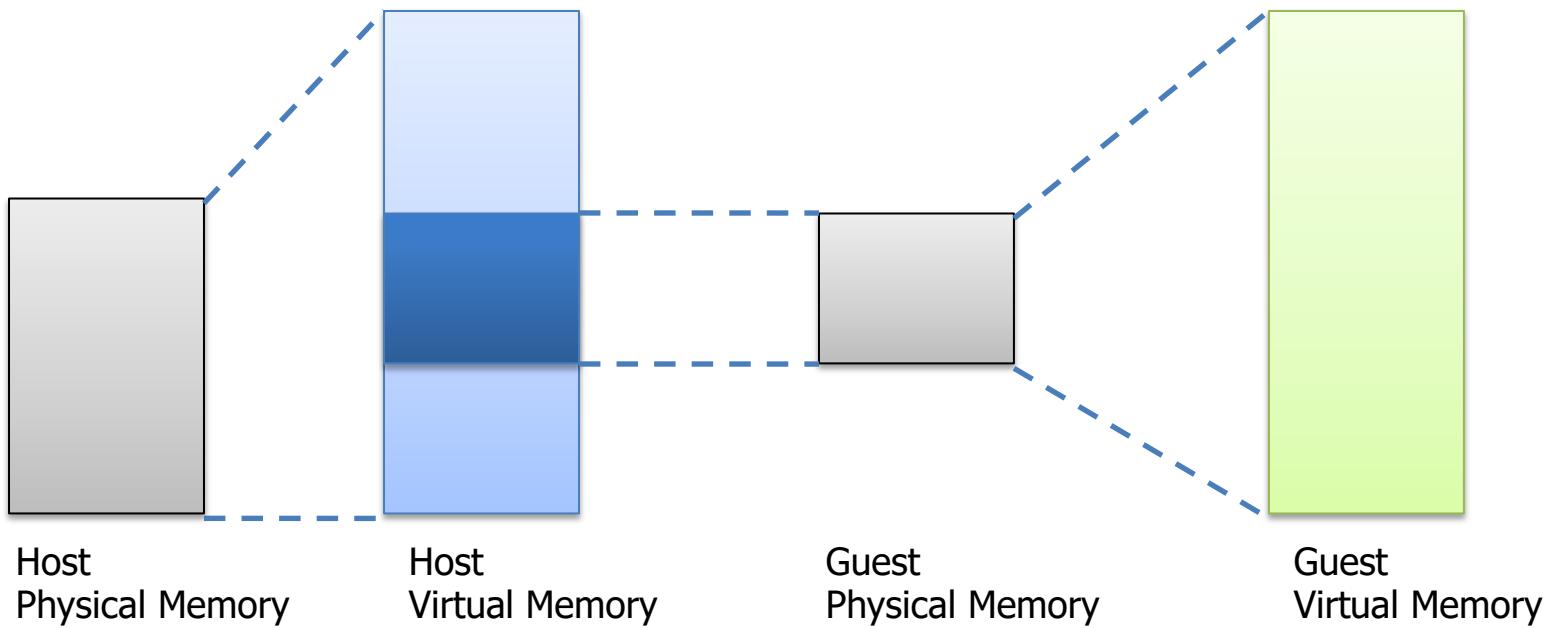


Processor Virtualization (cont'd)

- Comparison

	Para-virtualization	Full-virtualization (Emulation)	Full-virtualization (Binary translation)	Full-virtualization (Hardware-assisted VT)
Speed	Very Fast (Almost Native)	Very Slow	Fast	Fast
Guest Kernel Modification	Yes	No	No	No
Support Other Arch	No	Yes	No	No
Solutions	Xen, VMWare ESX	Bochs	VMWare, QEMU	KVM, Xen
Purposes	Server virtualization	Emulator	Desktop virtualization	Desktop virtualization

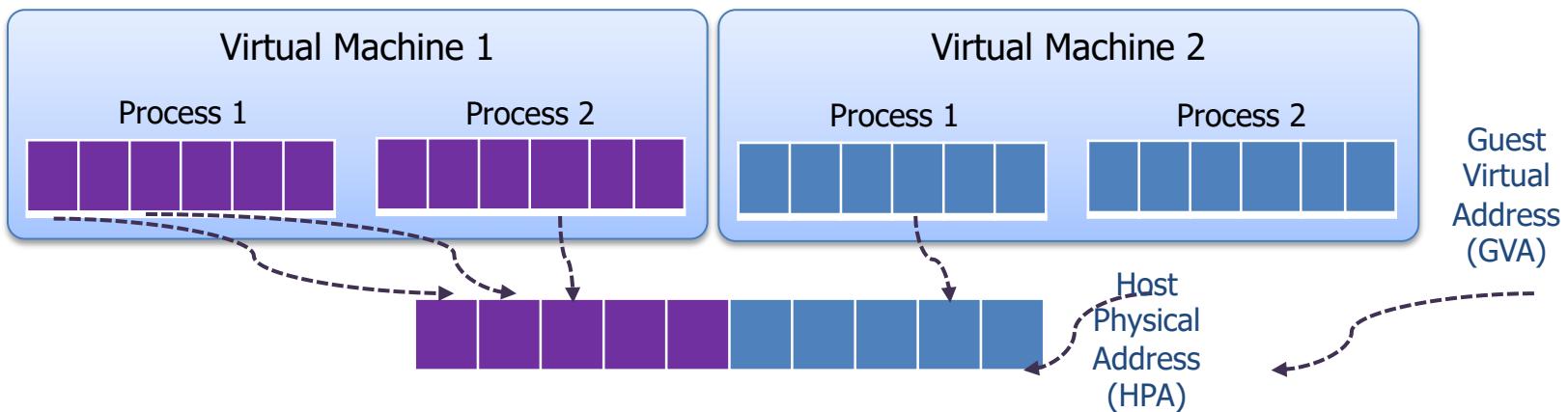
Virtual Machine Memory Map



Memory Virtualization

- **Direct Paging**

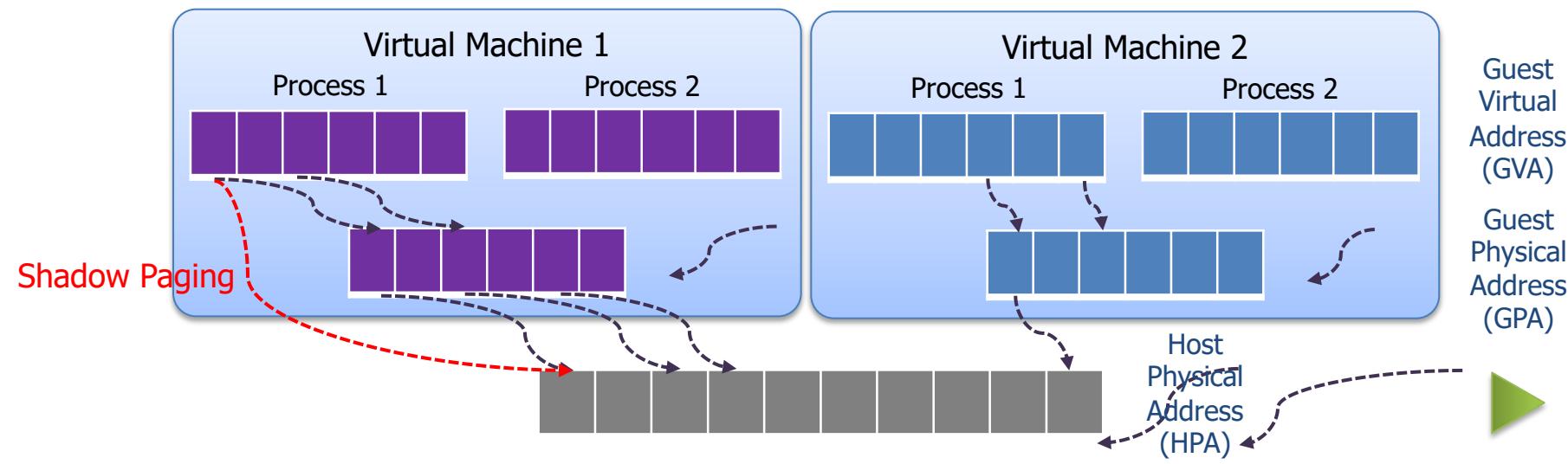
- Guest operating system directly maintains a mapping of Guest Virtual Address to Host Physical Address (GVA → HPA).
- When a logical address is accessed, the hardware walks these page tables to determine the corresponding physical address.
- Dedicated physical memory region is allocated at the initialization of guest OS.
- Pros
 - Simple to implement
 - High performance (no virtualization overhead)
- Cons
 - Need to modify guest kernel (not applicable to closed-source OS)
 - Inflexible memory management



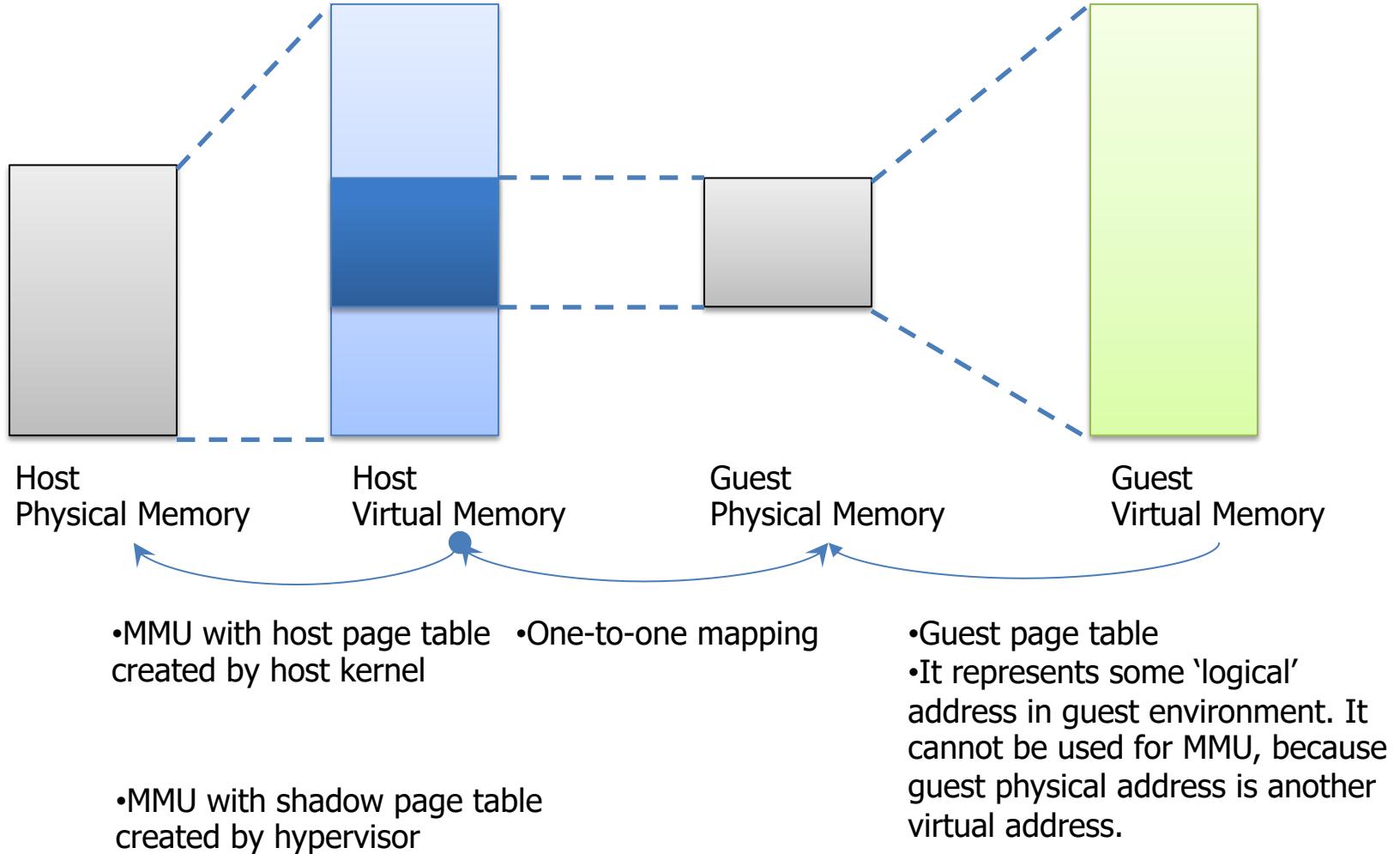
Memory Virtualization (cont'd)

- **Shadow Paging**

- VMM maintains GVA→GPA mappings in its internal data structures and stores GVA→HPA mappings in shadow page tables that are exposed to the hardware.
- The VMM keeps these shadow page tables synchronized to the guest page tables.
- This synchronization introduces virtualization overhead when the guest updates its page tables.
- Pros
 - Support unmodified guest OS
- Cons
 - Hard to implement and maintain
 - Large virtualization overhead



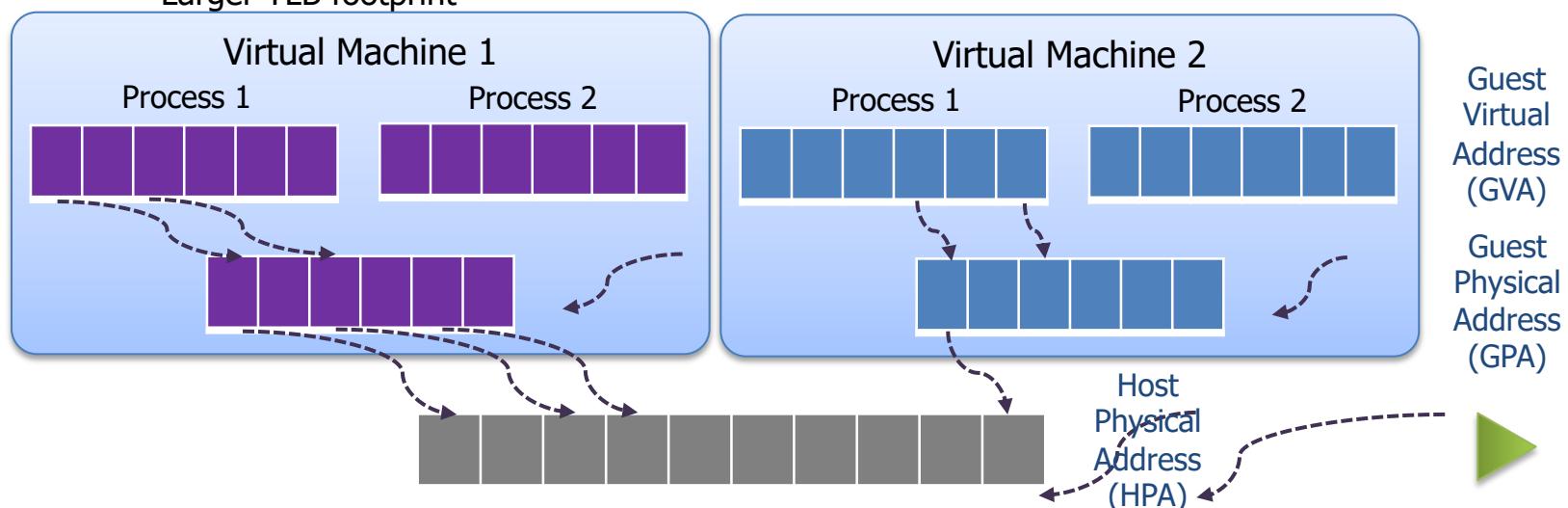
Shadow Paging



Memory Virtualization (cont'd)

- **Nested Paging**

- Guest OS continues to maintain GVA→GPA mappings in the guest page tables.
- But the VMM maintains GPA→HPA mappings in an additional level of page tables, called nested page tables.
- Both the guest page tables and the nested page tables are exposed to the hardware.
- When a logical address is accessed, the hardware walks the guest page tables as in the case of native execution, but for every GPA accessed during the guest page table walk, the hardware also walks the nested page tables to determine the corresponding HPA.
- Pros
 - Simple to implement
 - Support unmodified guest OS
- Cons
 - H/W supports is needed.
 - Larger TLB footprint

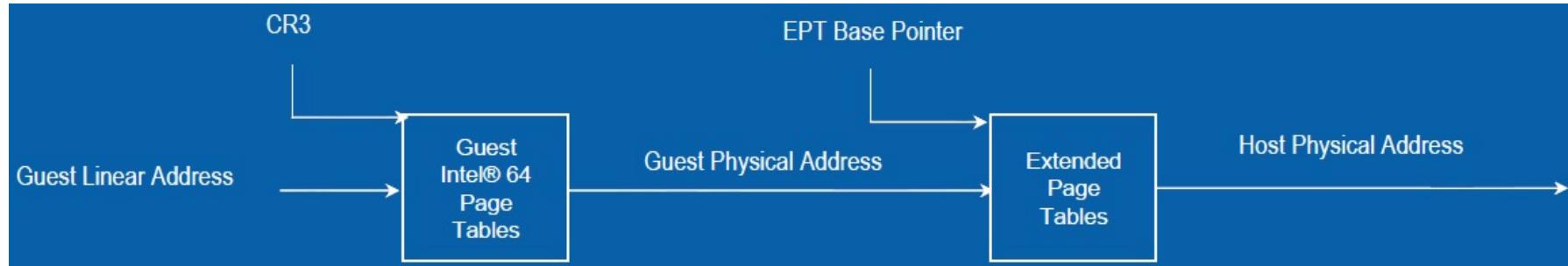


Memory Virtualization (cont'd)

- Comparison

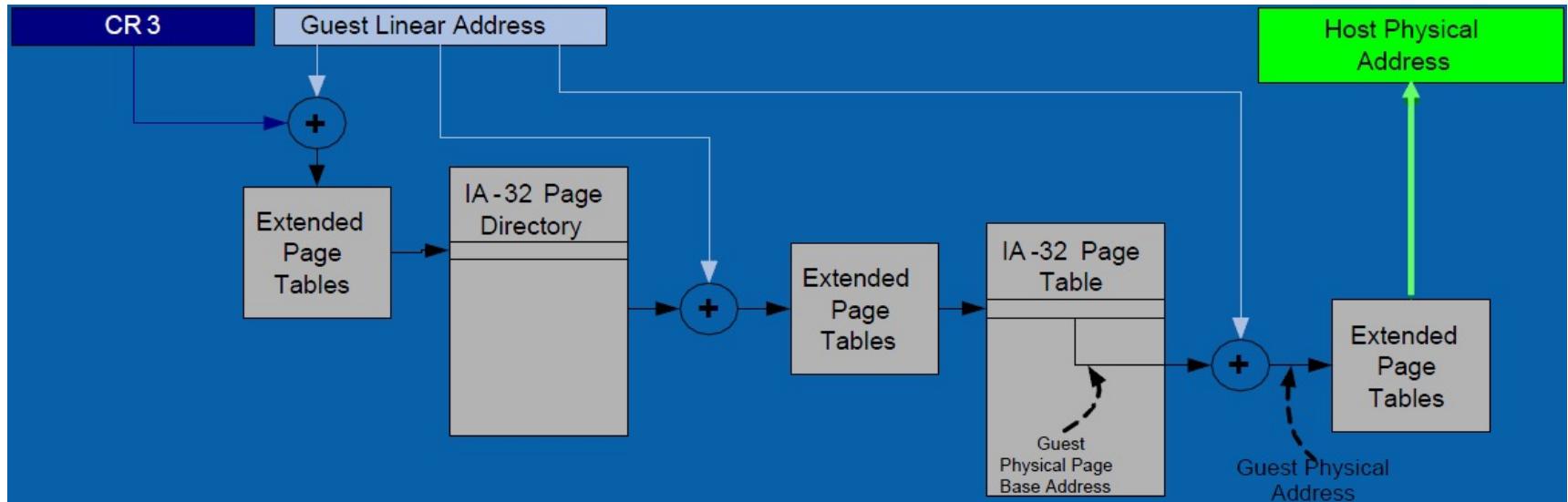
	Direct Paging	Shadow Paging	Nested Paging
Speed	Very Fast (Almost Native)	Very Slow	Fast
Guest Kernel Modification	Yes	No	No
Need H/W Support	No	No	Yes
Complexity	Simple	Complex	Very Simple

EPT



- The extended page-table mechanism (EPT) is a feature that can be used to support the virtualization of physical memory.
- Guest-physical addresses are translated by traversing a set of EPT paging structures to produce physical addresses that are used to access memory.
- Guest can have full control over page tables/events
 - CR3, CR0, CR4 paging bits, INVLPG, page fault
- VMM controls Extended Page Tables
- CPU uses both tables, guest paging structure and EPT paging structure
- EPT activated on VM entry
 - When EPT active, EPT base pointer (loaded on VM entry from VMCS) points to extended page tables
- EPT deactivated on VM exit

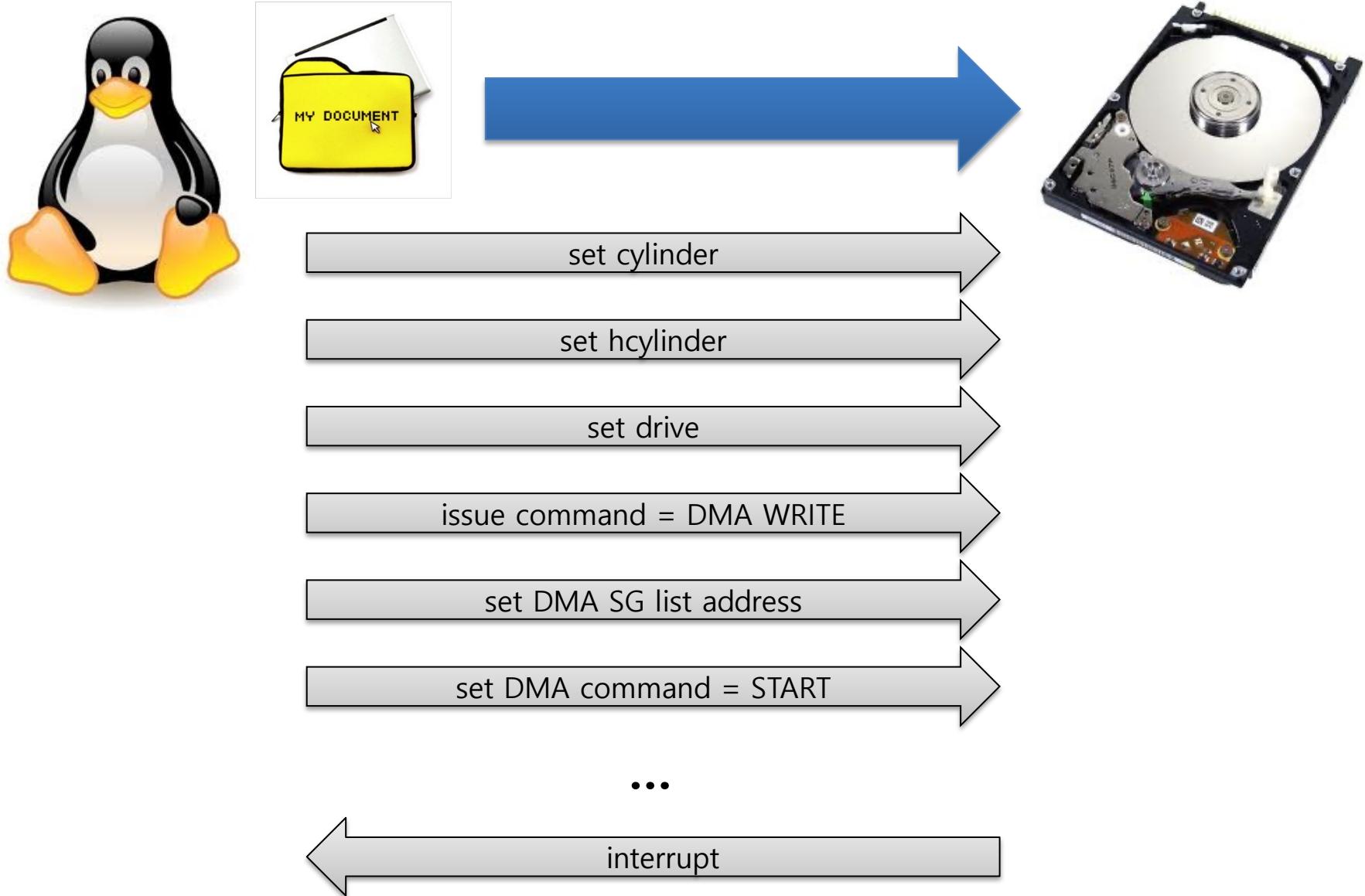
EPT Translation



- All guest-physical addresses go through extended page tables
 - Includes address in CR3, address in PDE, address in PTE, etc.
- In addition to translating a guest-physical address to a physical address, EPT specifies the privileges that software is allowed when accessing the address. Attempts at disallowed accesses are called EPT violations and cause VM exits.

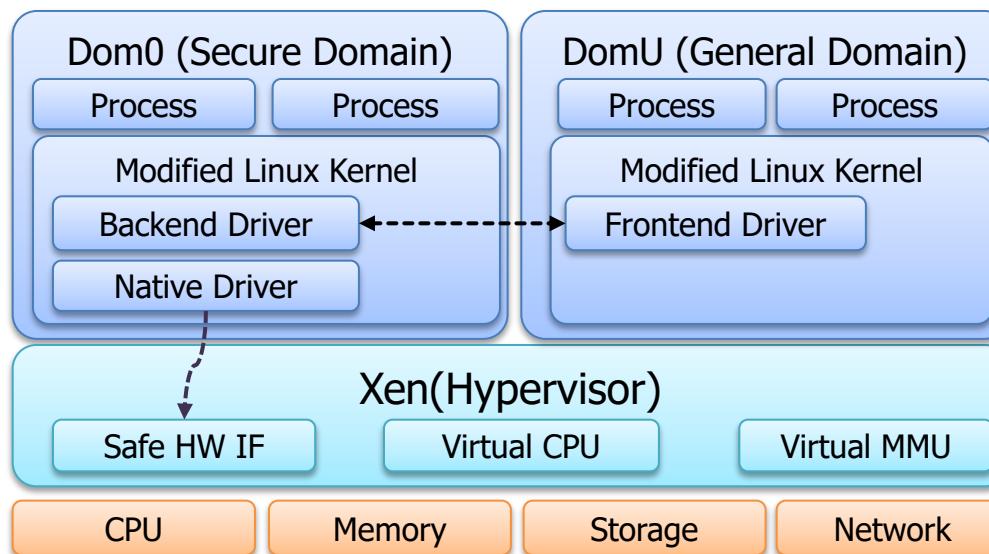


How OS and a device interacts

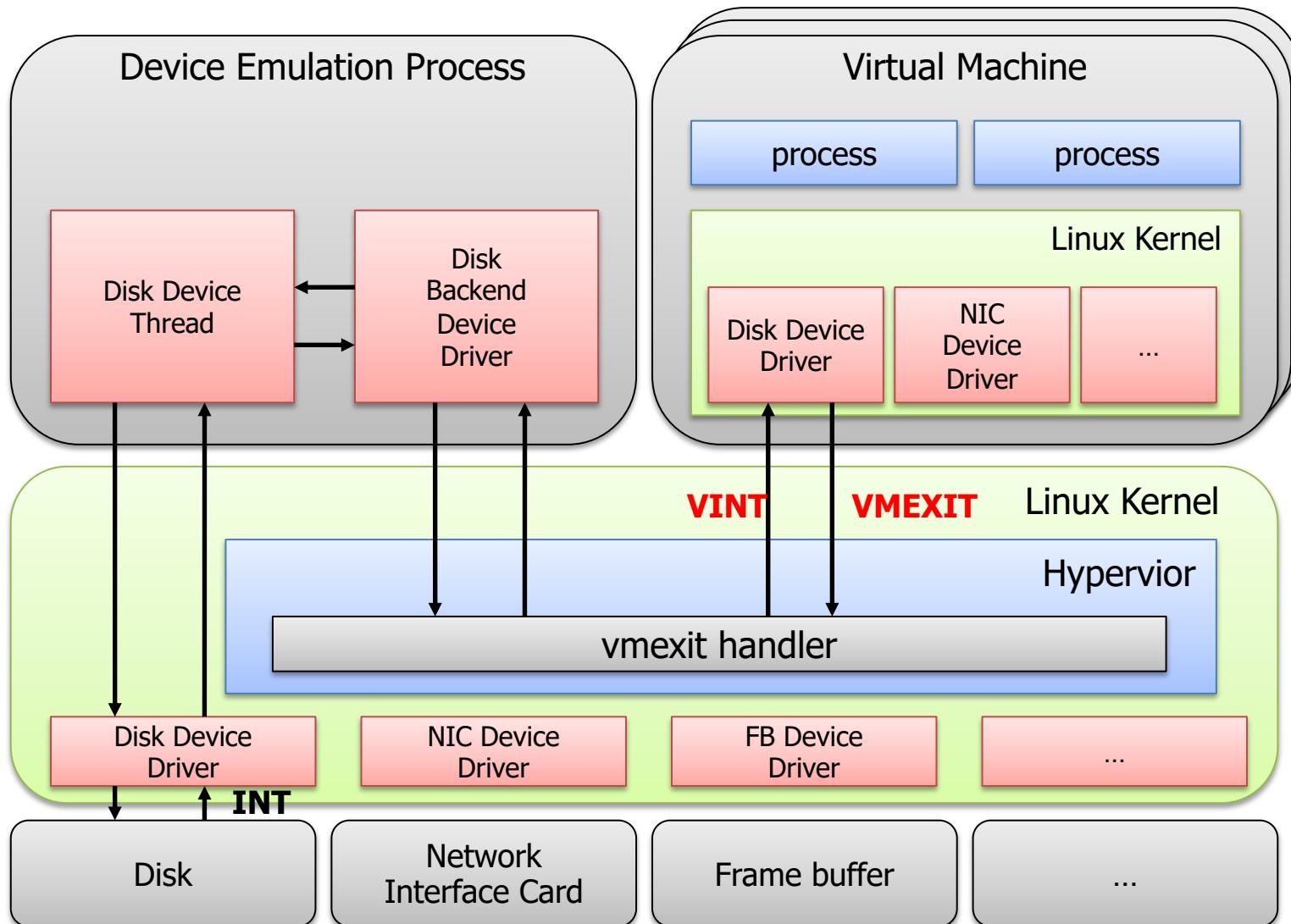


IO Virtualization

- Front-end/Back-end Driver Model
 - Guest OS uses para-virtualized front-end driver to send requests to backend driver.
 - Back-end driver on secure domain receives the requests, performs actual IO using the native driver.

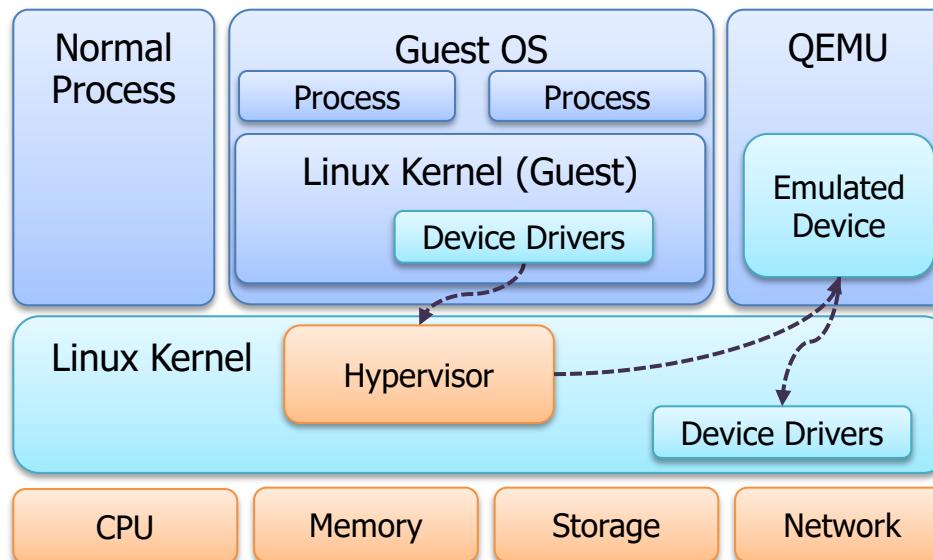


IO Virtualization Model Revisited : Front-end/Back-end Model



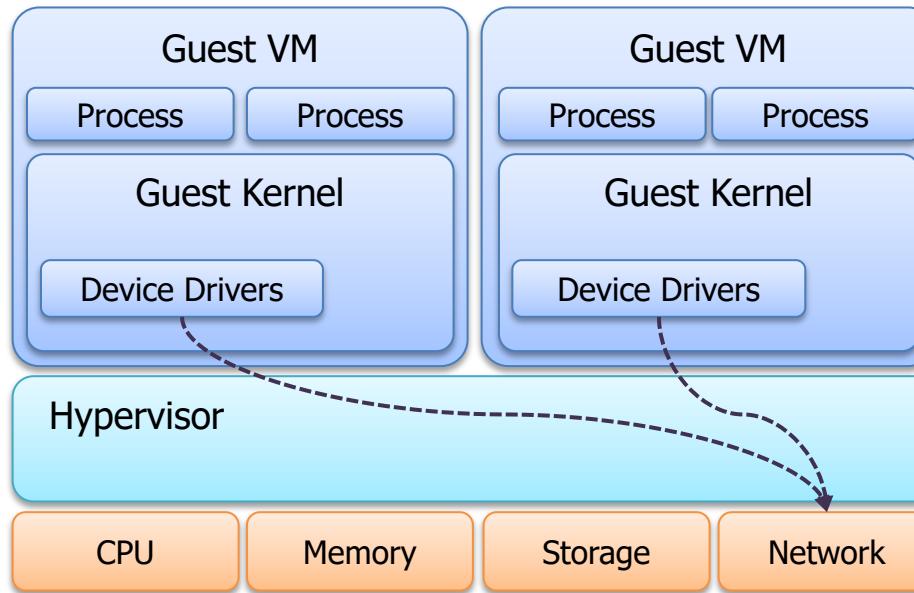
IO Virtualization (cont'd)

- Emulation
 - Behavior of a particular device is emulated as a software module.
 - Guest OS uses the native device driver for the particular device.
 - VMM intercepts all the access from guest OS to the device.
 - The intercepted accesses are sent to the emulated device.
 - The Emulated device do the actual IO operations.



IO Virtualization (cont'd)

- H/W Assisted IO Virtualization
 - A specially designed H/W supports concurrent accesses from multiple guest OS.
 - Guest OS uses the unmodified device driver.
 - An input/output memory management unit (IOMMU) allows a guest OS to directly use peripheral devices, such as Ethernet, accelerated graphics cards, and hard-drive controllers, through DMA and interrupt remapping.



IO Virtualization (cont'd)

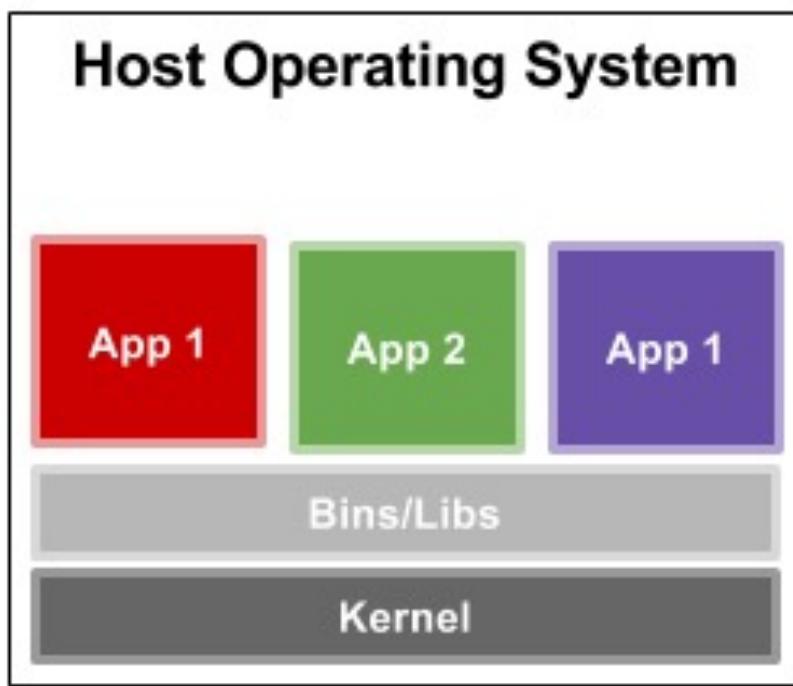
- Comparison

	Front-end/Back-end Driver Model	Emulation	H/W Assisted IO Virtualization
Speed	Very Slow	Very Slow	Fast
Device Driver Modification	Yes	No	No
Need H/W Support	No	No	Yes
Complexity	Simple	Complex	Simple

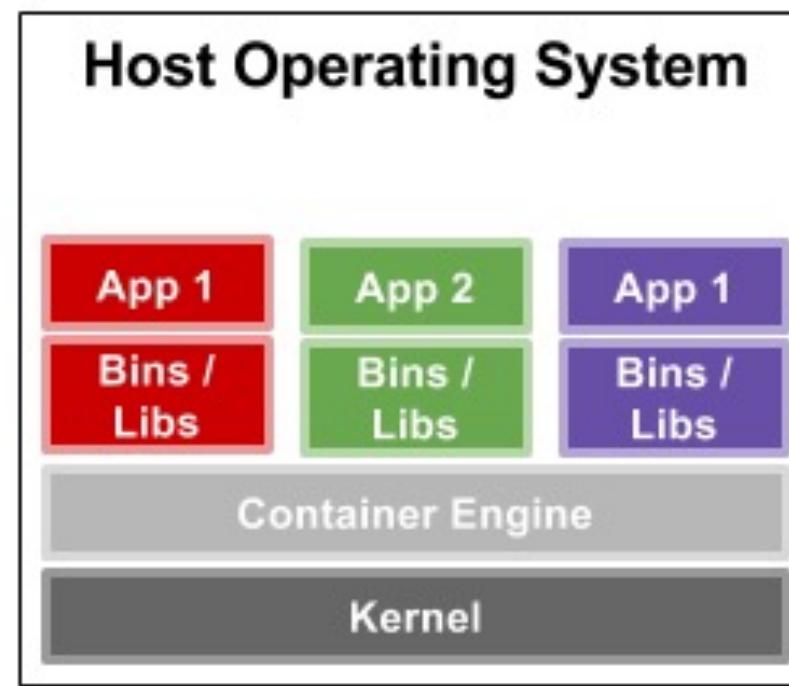
Containers

Containers

- Rather than spinning up an entire virtual machine, containerization packages together everything needed to run a single application or microservice (along with runtime libraries they need to run).
- A lighter-weight, more agile way of handling virtualization



Native Running Applications

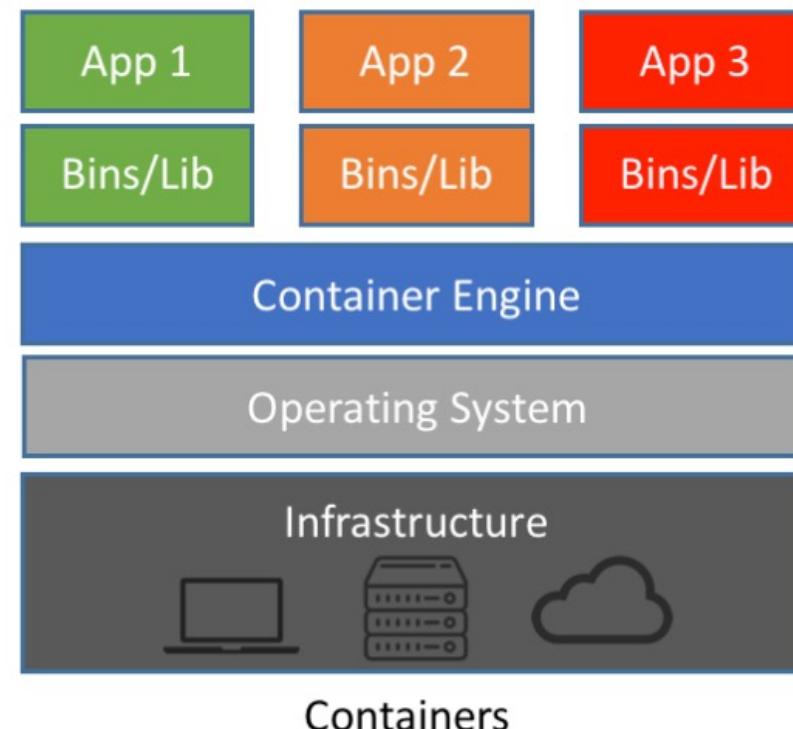
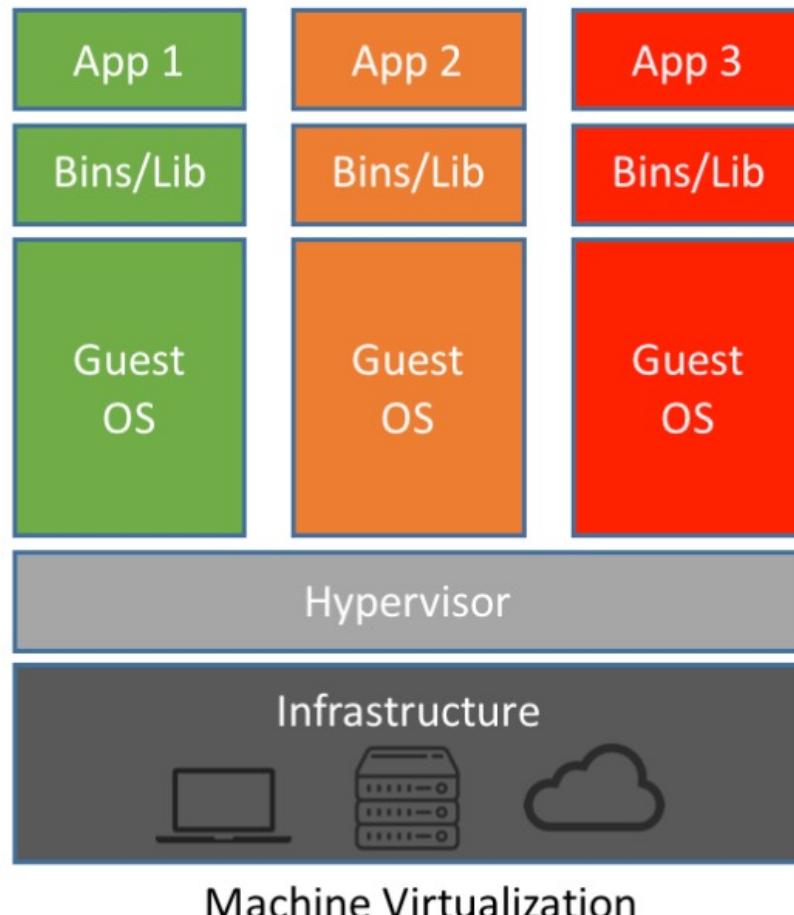


Containerized Applications

VMs vs. Containers

VMs virtualize a “hardware” to run multiple OS instances

Containers virtualize an “operating system” to run multiple workloads (on a single OS instance)

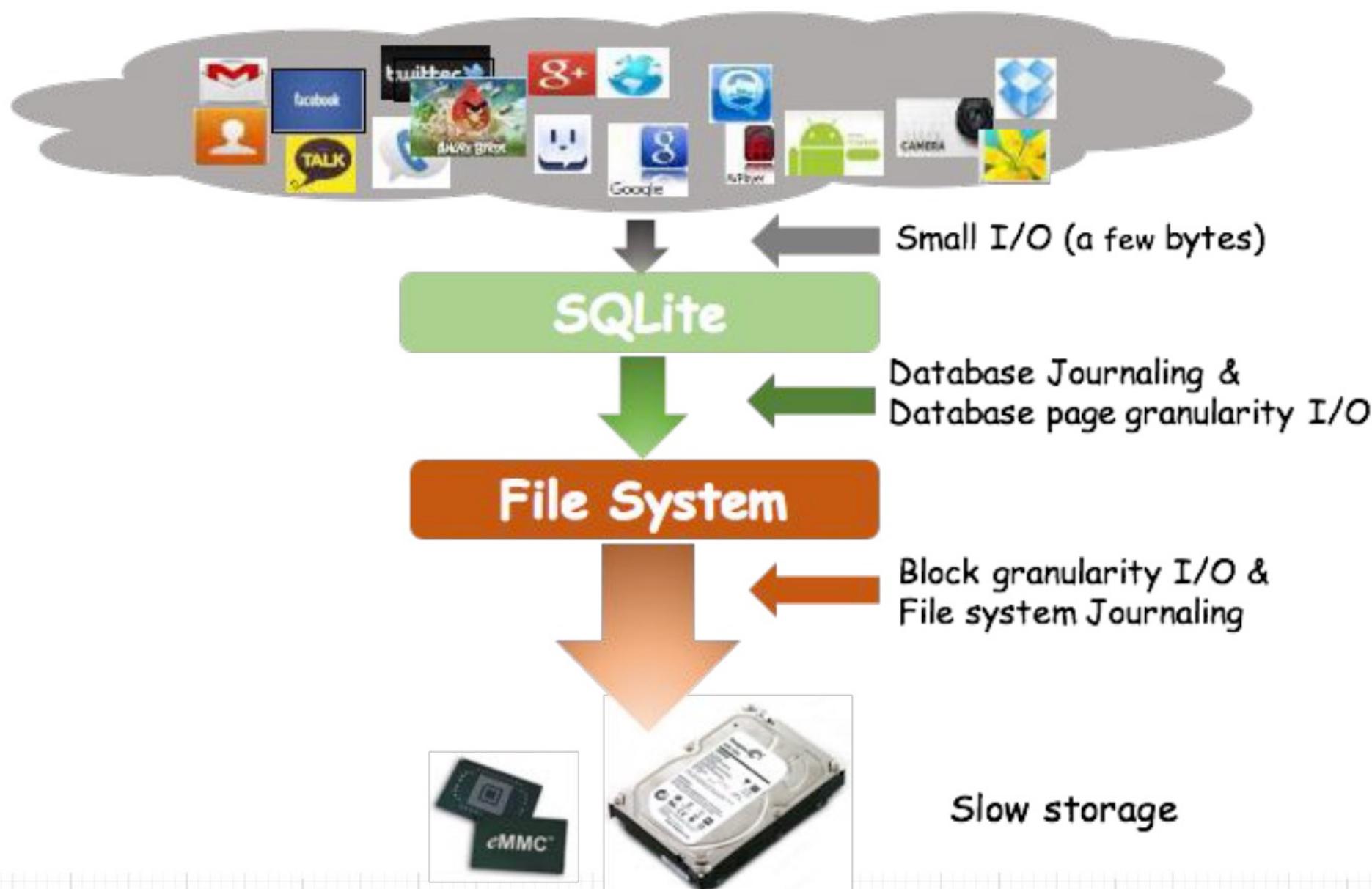


Containers

- OS-level virtualization
- Control Groups (cgroups), 2006
 - linux kernel v2.6.24
 - a collection of processes
 - limiting, accounting and isolating resource usage (CPU, memory, disk I/O, network) of a collection of processes
- LXC (LinuX Containers), 2008
 - Linux cgroups + Linux namespaces
 - Namespace allows complete isolation of an application's view of the operating environment, including process trees, networking, user IDs and mounted file systems
- Docker, 2013
 - Initially based on LXC. Evolved to use its own library libcontainers
 - Containers exploded in popularity

Persistent Memory and Its Implications to Systems Software

Problem: I/O Bottleneck



Persistent Memory Can be a Solution

What is Persistent Memory?

- Non-volatile (thus also called non-volatile memory)
- Byte-addressable
- DRAM-like performance (a bit slower)
- Large Capacity (up to 3TB per socket)

Intel Optane DC Persistent Memory is available

Persistent Memory

NEWS

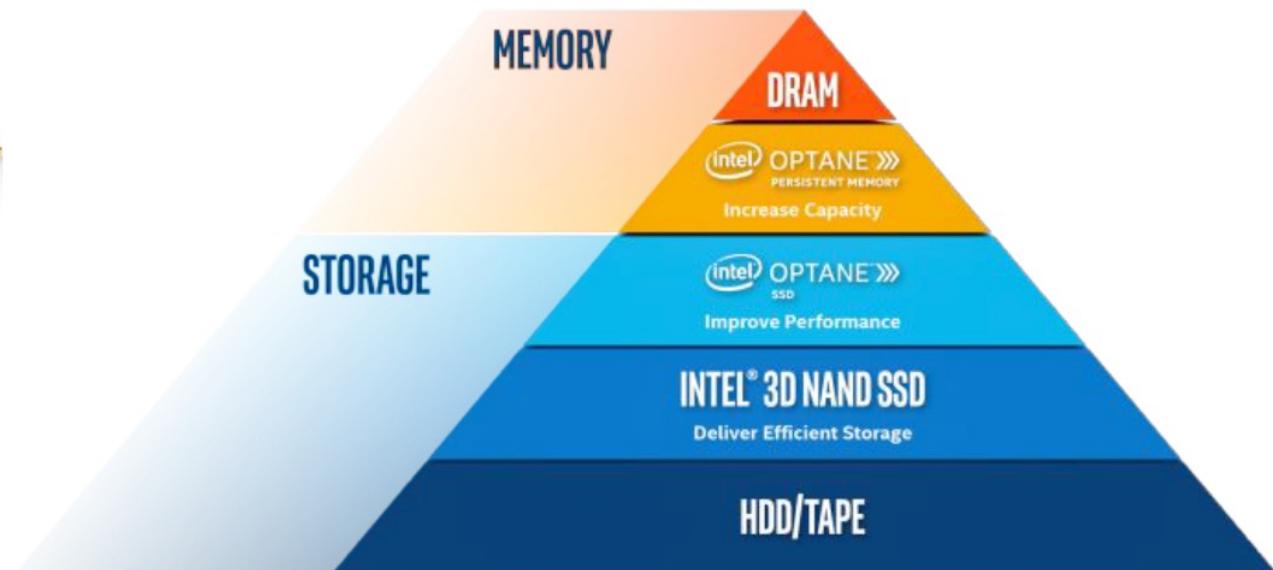
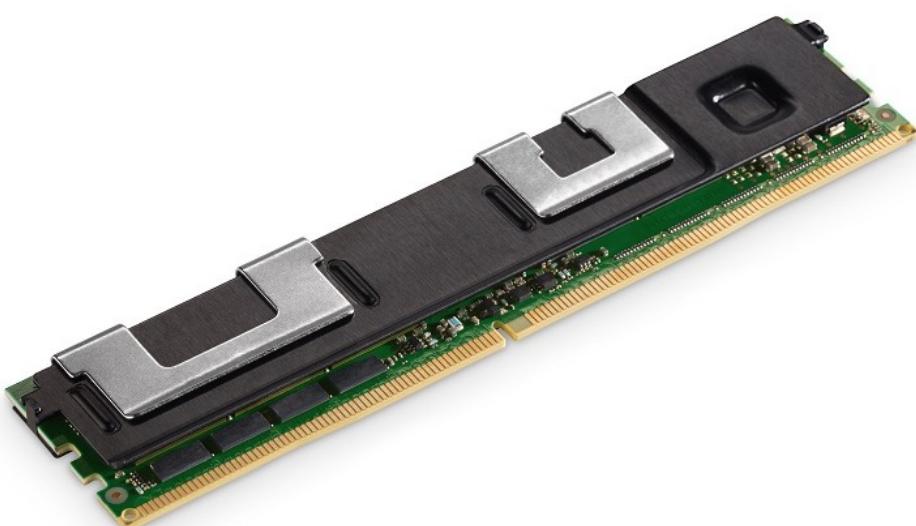
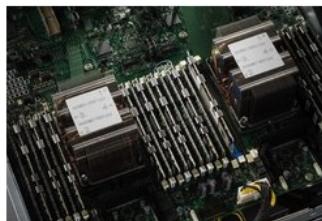
Intel Announces Optane DIMMs, Aims to Disrupt Memory/Storage Hierarchy

May 31, 2018

By: Michael Feldman

Intel has unveiled its much-anticipated non-volatile memory (NVM) modules based on its 3D XPoint technology.

The product line, which goes by the name Optane DC persistent memory, is currently sampling and will be shipped to select customers later this year, with general availability slated for 2019.



Optane DC persistent memory module. Source: Intel

Persistent Memory

Intel Optane Persistent Memory aims to fill the gap DRAM can't

We look at Optane and its PMem product that Intel believes will fill a big gap between memory and storage that DRAM technology just cannot deal with



By [Antony Adshead](#), Storage Editor

It may not have been noticed by many IT professionals, but recently Intel sold its solid-state drive (SSD) manufacturing assets to South Korean firm SK Hynix. An interesting bit is what it didn't sell, however.

Intel is holding tight to its [3D Xpoint](#)-based Optane product and sees a huge opening up for the type of performance characteristics it offers, which are so different from RAM and storage and very well-suited to multiple emerging use cases. These include simulation and analytics, virtual infrastructure and artificial intelligence (AI).

But what is Optane and what use cases is it best suited to?

In terms of performance, Optane sits between the very expensive [DRAM](#) memory and tape, which is costly but much slower to access storage media such as SSD, disk and [tape](#).

In a recent webcast, Intel senior director Kristie Mann put Optane latency between 10ns (nanoseconds) and 10μs (microsecond) in its [Optane Persistent Memory](#) (PMem) product and 10μs in its DRAM product. That compares with 100ns (nanoseconds) for DRAM while standard SSD has a latency of around 100μs. Hard disk drives and tape are obviously way slower than that.

U.S. Department of Energy and Intel to deliver first exascale supercomputer

Targeted for 2021 delivery, the Argonne National Laboratory Supercomputer will enable high-performance computing and artificial intelligence at exascale

Contributor — [Intel](#)

Mar 20th, 2019



LATEST IN COMPUTERS/CLOUD COMPUTING

Applications

[The Growing Potential for AI in the Cloud and IoT](#)

Alix Paultre, Editor

May 5th, 2021



Applications

[Circuit Card Diagnostic and Test System Offers Component-Level Diagnostics](#)

May 3rd, 2021



J.Yang,et.al., An Empirical Guide to the Behavior and Use of Scalable Persistent Memory, FAST'19

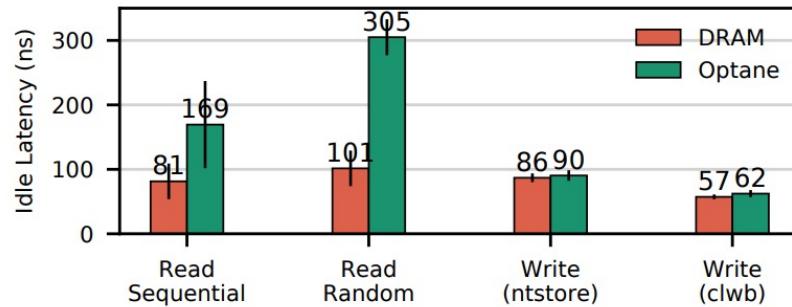


Figure 2: **Best-case latency** An experiment showing random and sequential read latency, as well as write latency using cached write with `clwb` and `ntstore` instructions. Error bars show one standard deviation.

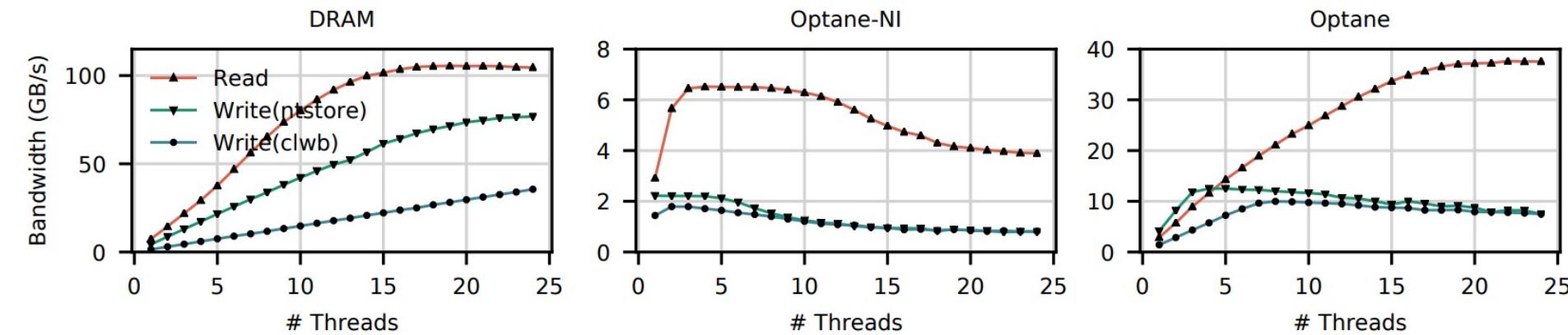
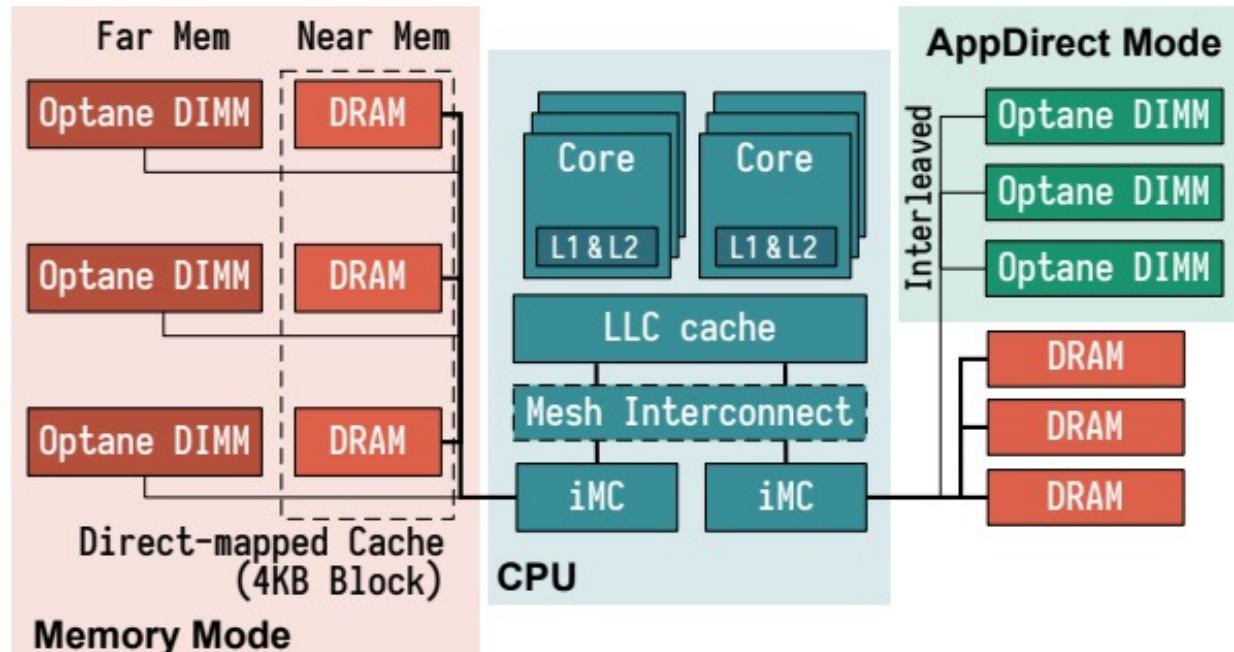


Figure 4: **Bandwidth vs. thread count** An experiment showing the maximal bandwidth as thread count increases (from left to right) on local DRAM, non-interleaved and interleaved Optane memory. All threads use a 256 B access size. (Note the difference in vertical scales).

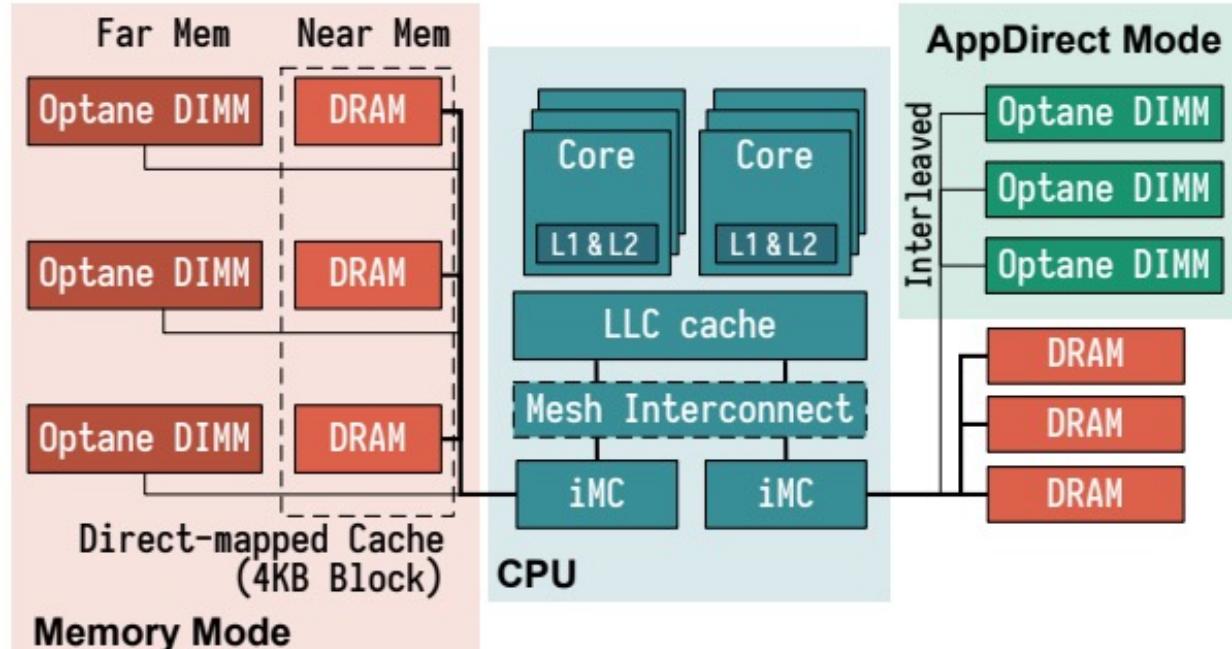
(1) Memory Mode



(a) Optane Platform Modes (Memory and AppDirect)

- DRAM acts like a last level cache
 - DRAM as near memory and PM as far memory
- Optane PM acts like large memory
 - Transparent
 - Cannot guarantee durability
 - Cannot direct access to the Optane PM

(2) AppDirect Mode



(a) Optane Platform Modes (Memory and AppDirect)

- Access both DRAM and Optane PM directly
 - Can guarantee durability
- How to use this mode?
 - File systems
 - Direct access from user applications

Use PM as File Systems

Traditional file systems are:

- Optimized for HDD/SDD
- Not adequate for Optane PM
- Optane PM has different I/O characteristics and atomicity granularity

Remove page caches

- Optane PM has DRAM-like performance, page caches add more latency
- ext4-DAX
- A. Xu,et.al. NOVA: A Log-structured File System for Hybrid Volatile/Non-volatile Main Memories, FAST'16
 - Log-structured FS provides cheaper failure-atomicity than journaling FS
 - NVM can support fast and high performance random access

Use PM as File Systems (cont.)

Moving from kernels to user space

- Context switching overhead is expensive
- Y. Kwon, et.al. Strata: A Cross Media File System, SOSP'17
 - Library OS
 - Provide user-level private operation log

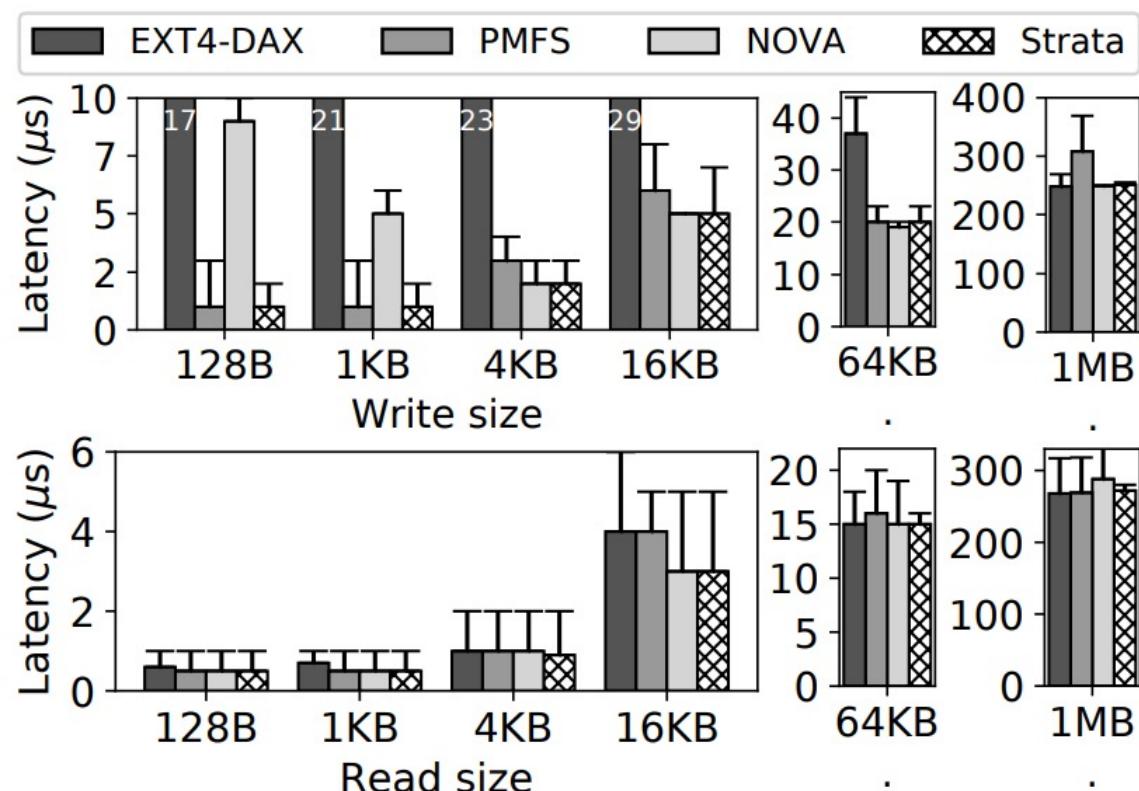


Figure 3: Average IO latency to NVM. Error bars show 99th percentile.

Use PM as “Persistent Memory” (direct access)

Design persistent applications without paying traditional storage stack overhead

- e.g., persistent key-value store backed by PM

Challenges in the presence of volatile cache

- Cacheline may be flushed in an arbitrary order
 - How to ensure that one PM location becomes persisted earlier than the other location?
- Current CPUs guarantee 8byte atomic update
 - How to update multiple PM locations atomically?

PM Programming 1010

Persistence instruction

- Cache line flush instruction
- Non-temporal store instruction
- Memory fence instruction

Persistent memory allocation

- Can recover from system crash

Persistent Memory Development Kit (PMDK)

Persistence instruction

Cache line flush instruction

- clflush: flush and invalidate the cache line. Implicitly ordered.
- clflushopt: flush and invalidate the cache line. Without serialization.
- clwb: flush the cache line but keeps the cache line valid (remains as clean)

Non-temporal store instruction

- Movntq: bypass CPU cache. No need to flush the cache line.

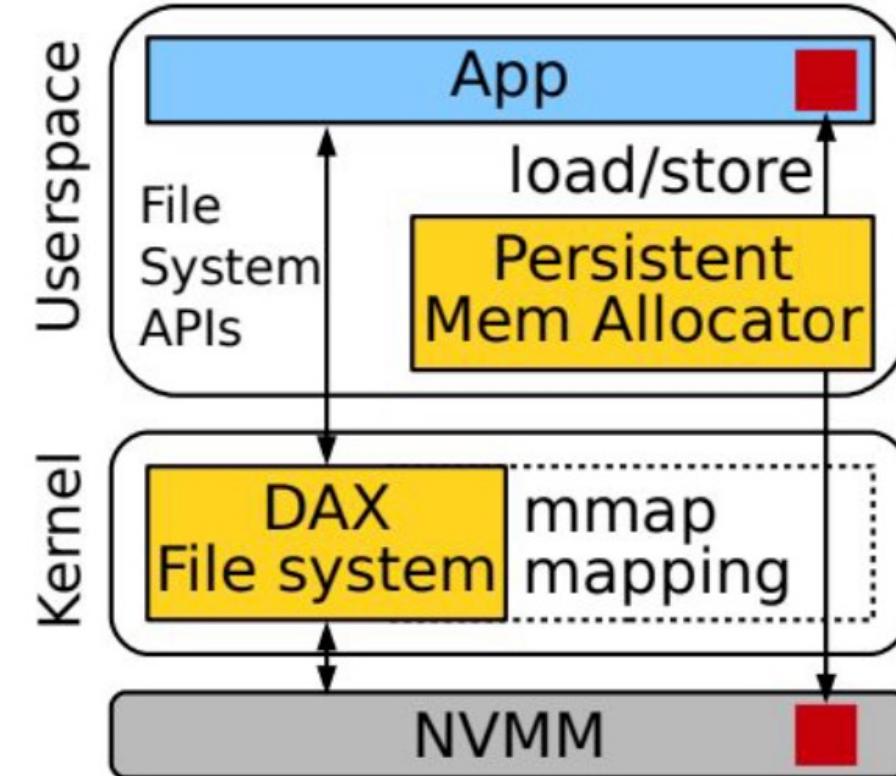
Memory fence instruction:

- clflushopt, clwb and movntq need memory fence to guarantee the order
- mfence: guarantee the completion of previous memory instructions
- sfence: guarantee the completion of previous store instructions

Persistent Memory Allocation

PMDK-libpmemobj: standard persistent memory allocator

- Based on DAX + mmap
- Inherited from traditional volatile memory allocator
 - Buddy allocator
 - AVL-tree based memory metadata management
 - In-line metadata management
 - Per-thread arena management



PM Program Example

```
PMEMobjpool *pop;
char *path = "/mnt/pmem0/example";

pop = pmemobj_create(path,"example", PMEMOBJ_MIN_POOL, 0666);

PMEMoid root_pptr = pmemobj_root(pop,sizeof(root_obj)); // create / load root object
root_obj *root= pmemobj_direct(root_pptr); // Get vaddr

char val[VAL_SIZE];
memcpy(val,"persistent",VAL_SIZE);
val[VAL_SIZE-1] = '\u00c4'0';

data_args arg;
arg.key = 1;
arg.val = val;
int ret = pmemobj_alloc(pop, &(root->addr), sizeof(data), 0, constructor, &arg);
```

PM Program Example

```
static int constructor(PMEMObjpool *pop, void *ptr, void *arg){  
  
    data_args *args = (data_args*)arg;  
  
    data *d = (data *)ptr;  
    d->key = args->key;  
    memcpy(d->val, args->val,VAL_SIZE);  
  
    pmemobj_flush(pop,d,sizeof(data));           // cache line flush instruction  
    pmemobj_drain(pop);                         // memory fence  
  
    d->valid = 1;                                // for crash consistency, it will be updated atomically  
  
    pmemobj_flush(pop,(void *)&(d->valid),sizeof(unsigned long)); // cache line flush  
    pmemobj_drain(pop);                          // memory fence  
  
}
```