CS 60038: Advances in Operating Systems Design

Topic 1: Linux Kernel Overview and Process Management

Department of Computer Science and Engineering

INDIAN INSTITUTE OF TECHNOLOGY KHARAGPUR

Arobinda Gupta
agupta@cse.iitkgp.ac.in

Sandip Chakraborty
sandipc@cse.iitkgp.ac.in

Preface

- **Course Instructors**
  - Arobinda Gupta
  - Sandip Chakraborty
- **Teaching Assistants**
  - Neha Dalmia
  - Nisarg Upadhyaya
- **Class Timing**
  - Monday: 11:00 – 12:00
  - Tuesday: 08:00 – 10:00
- **Course Credit**
  - 3-0-0-3
- **Prerequisite**
  - Operating Systems (Undergrad)
  - Programming in C
  - Data Structures and Algorithms
- **Grading and Evaluation**
  - Term Project: 15%
  - Assignments: 20%
  - Mid Sem: 25%
  - End Sem: 35%
  - Attendance: 5%

Hands-on Inside the Kernel

- **Assignment 1**: Loadable Kernel Modules (LKM) -- how can you inject your own functionalities within the Linux kernel
- **Assignment 2**: Implementing your own system calls
- **Assignment 3**: eXtended Barkeley Packet Filters (eBPF) and eXpress Data Paths (XDP) -- Runtime programmability inside the kernel

Topics We are Going to Cover

- Linux Kernel Architecture – Kernel organization, Process structure, Process management, IPC, System calls
- Process scheduling in Linux Kernel – Scheduling algorithms and their implementations
- Hybrid kernel architecture
- Embedded kernel – optimizations for embedded devices
- File system architecture
- Virtualization – Hypervisor architecture, full and para virtualization, hardware assisted virtualization, introduction to confidential computing

Understanding Kernel Versions

- Prepatch
  - Mainline kernel pre-releases that are mostly aimed at other kernel developers and Linux enthusiasts.
  - Must be compiled from source and usually contain new features that must be tested before they can be put into a stable release.
  - Prepatch kernels are maintained and released by Linus Torvalds.
- Mainline
  - Mainline tree is maintained by Linus Torvalds.
  - All new features are introduced and where all the exciting new development happens.
  - New mainline kernels are released every 9-10 weeks.

Understanding Kernel Versions

- Stable
  - After each mainline kernel is released, it is considered "stable."
  - Any bug fixes for a stable kernel are backported from the mainline tree and applied by a designated stable kernel maintainer.
  - There are usually only a few bugfix kernel releases until next mainline kernel becomes available -- unless it is designated a "longterm maintenance kernel."
  - Stable kernel updates are released on as-needed basis, usually once a week.
- Longterm
  - Usually several "longterm maintenance" kernel releases provided for the purposes of backporting bugfixes for older kernel trees.
  - Only important bugfixes are applied to such kernels and they don't usually see very frequent releases, especially for older trees.

Linux Kernel Architecture

- Check the kernel versions from here -- https://www.kernel.org/
  - We'll follow Kernel 5.10.188 (Longterm release, Release date: 13 December 2020, EOL: December, 2026)

- Cross reference to browse the kernel source: https://elixir.bootlin.com/linux/v5.10.188/source

Kernel Responsibilities

- OS based on UNIX always divides OS functionality into two components:
  - Kernel – executes in supervisor mode
  - OS components – executes in user mode

- Kernel responsibility
  - Abstract and manage hardware
  - Manage sharing of resources among executing programs
    - Support processes, files and other resources
    - Managed by system calls

- Linux differs from other systems that use UNIX in :
  - The data structures and
  - The algorithms used.

Resource Abstraction

- Creation of software to simplify operations on hardware
  - Hides the details of hardware operations needed to control it.
  - Other software can use the abstraction without knowing the details of how the hardware is managed.
  - E.g. device driver.

- In Linux, abstractions are into
  - Processes
    - A process is an abstraction of the CPU operation that executes an object program placed in the RAM.
  - Resources

- Linux is a multiprogrammed OS where:
  - CPU is time-multiplexed
  - Memory is space-multiplexed
    - Divided into blocks called memory partitions.

Resource Management

- Procedure for creating resource abstractions, and allocating and deallocating of system resources to and from processes as they execute

- Resource – any logical component that the OS manages and can be obtained only by having the process request the use of the resource from the OS.
  - E.g: memory (RAM), storage, CPU, files.

Allowing sharing of Resources

- Sometimes, processes need to share a resource
  - Eg. Process that reads a database with one that writes into it.
- Sharing violates exclusive use mechanism
- Mechanism for resource sharing adds complexity to resource managers design.

Resource Sharing

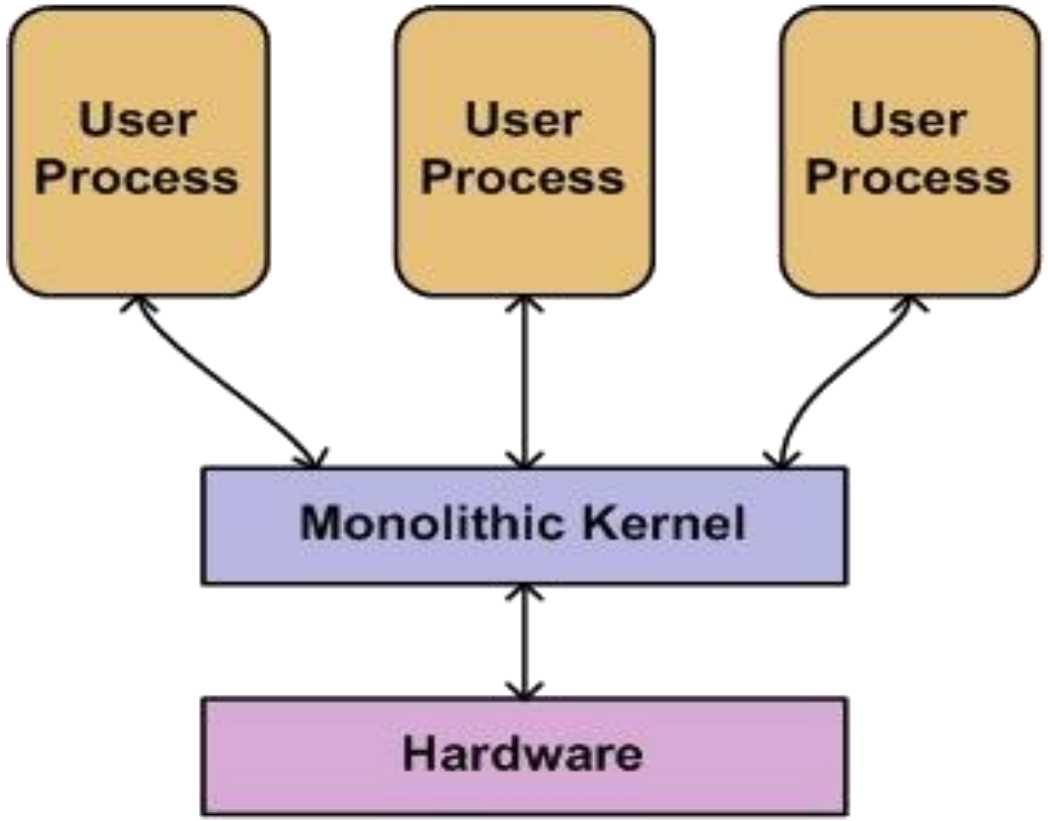
- Processes request, use and release resources
- Therefore, the OS needs to
  - manage competition for resources.
  - assure exclusive use of resources.
    - CPU – by disabling interrupt
    - RAM – by hardware memory protection mechanism
    - Devices – by preventing CPU from executing I/O instructions unless on behalf of processes allocated.
- Implementation of exclusive use.
  - Partly hardware, partly software.
  - Hardware knows when OS is getting CPU use – using mode bit (supervisor(or kernel)/user)

OS Function Partitioning

- 4 categories of functions:
  - Process and Resource management
    - gives illusion of multiple virtual machines.
  - Memory Management
    - Allocation to processes
    - Protection from unauthorized access.
    - swapping in virtual memory system.
  - Device Management
    - Provides controlling functions to many devices
  - File Management
    - Provides organization and storage abstractions

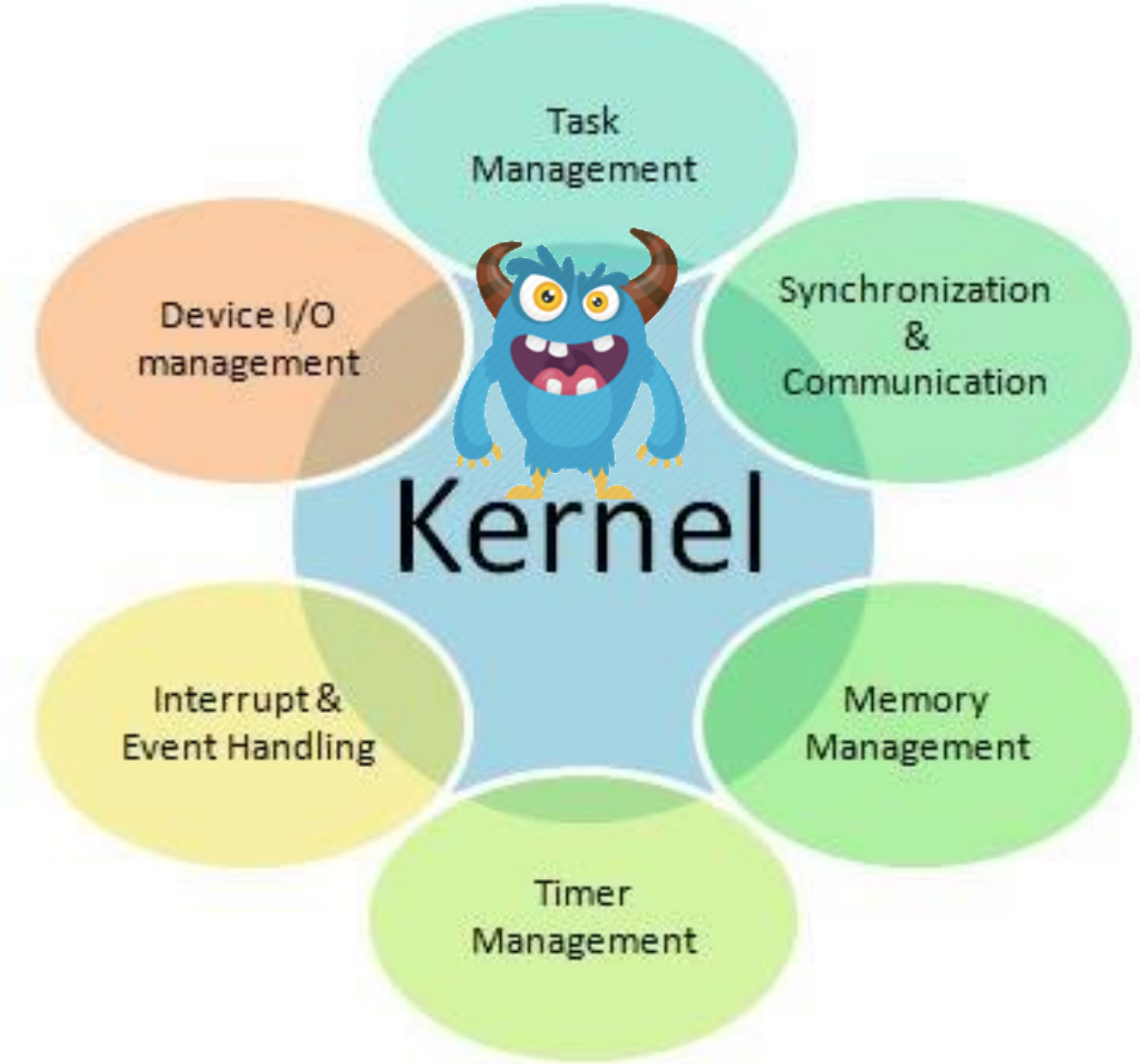
Monolithic Kernel

- Kernel takes care of almost all the system tasks
- Applications do not have control over resources
- Example:
  - Windows 9x series: Windows 95, 98
  - BSD: FreeBSD, OpenBSD
  - Linux



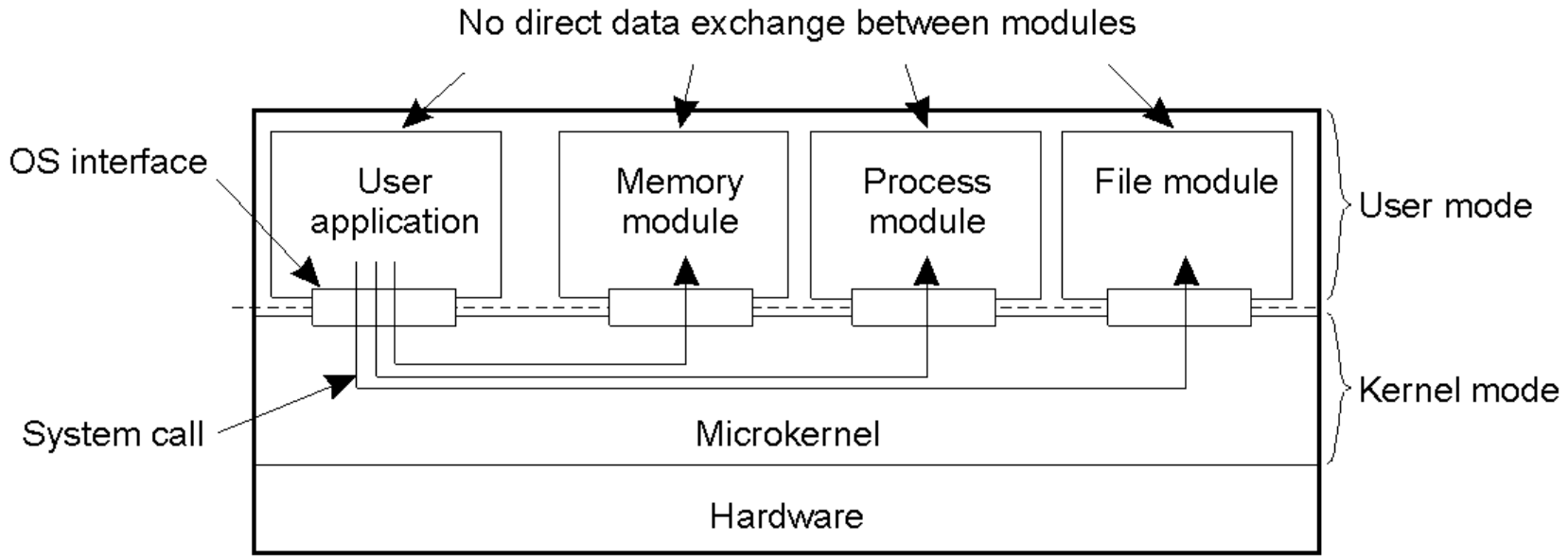
Ref: Kaashoek et al.

The Kernel - The Almighty



Microkernel

- Runs most of the operating system services at the user space.
- Parts that require privilege (IPC) are in kernel mode and other critical parts (FS, Network Stack) in user mode.
- Example: L4 microkernel, Mach (Predecessor of MacOS)
- Performance issue !



Ref: Tanenbaum's distributed systems.

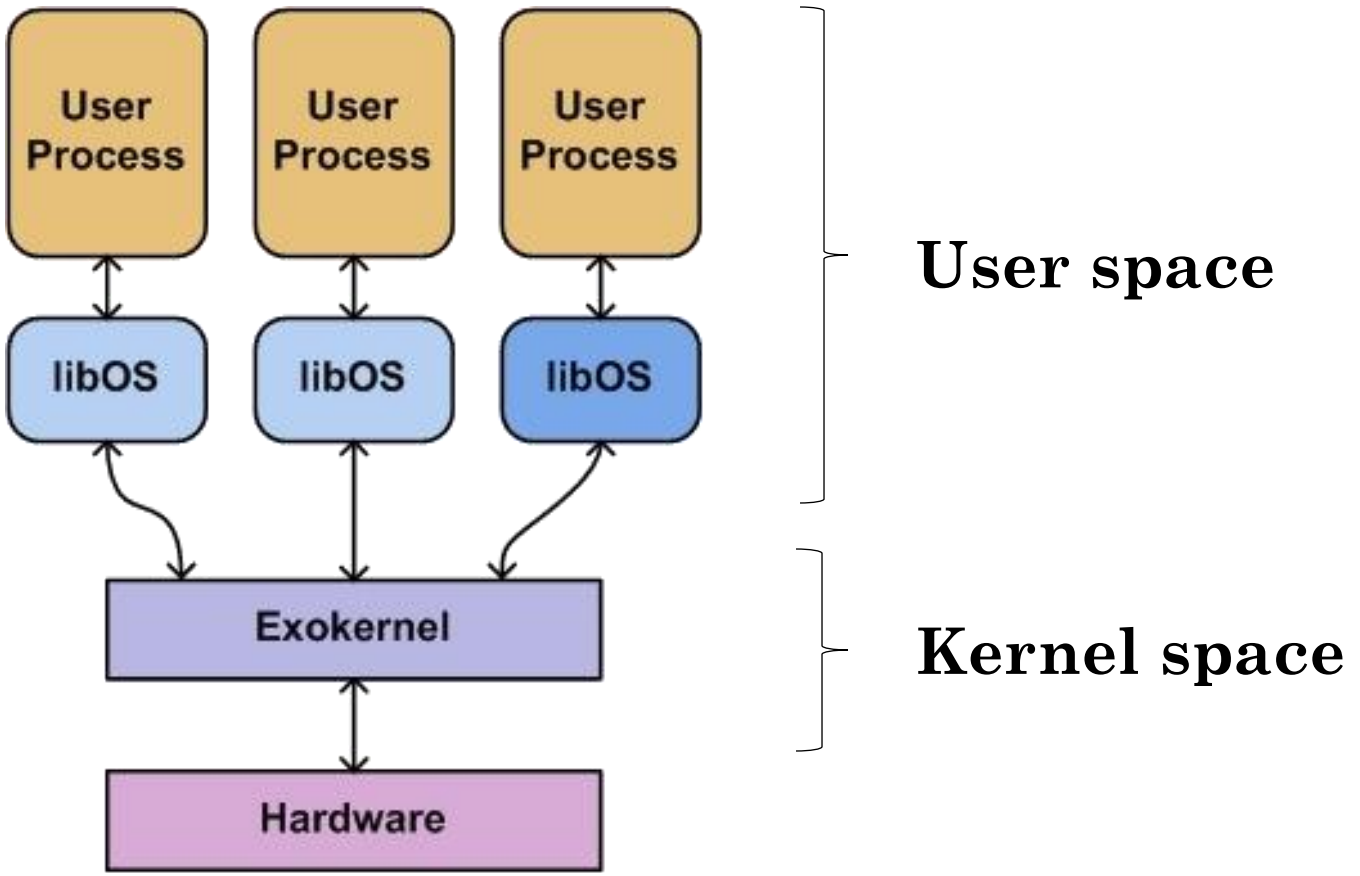
Exokernel

- Separate Management and Protection
  - **Management**: Allocate resources based on the requirement
    - Example: An application asks for a memory block
    - However, read-write strategy over that memory block will be decided by the application
  - **Protection:** Ensures no conflict in resource allocation
    - Maintain ownership of resources
    - Example: Do not allocate a memory block if it is already allocated to another application

- Design is based on the end-to-end argument
  - "Applications know better than operating systems what the goal of their resource management decisions should be and therefore, they should be given as much control as possible over those decisions"

Exokernel: Example

- Application manages its disk-block cache and kernel allows cached pages to be shared securely between applications



Kernel Organization

- Linux kernel is monolithic
  - Implemented in a single executable module
- Data structure for any aspect of the kernel is available to any other part of the kernel except normally
  - device drivers
  - Interrupt handlers

Philosophy:

- Main part of kernel should never change
- New, kernel space software can be added for device management as devices can change.

Linux Kernel

- To account for advanced devices (esp. bitmap displays and networks)
  - Linux uses **modules** as containers to implement extensions to the main part of the kernel.
- Module – an independent software unit that can be designed and implemented after the kernel and dynamically loaded into kernel space.
- Module interface is more general than a device driver interface
  - Giving more flexibility to the system programmer to extend the kernel functionality.
- Module can implement device drivers.

Interrupt

- To interrupt kernel for some service request
  - By system call from processes (Software interrupt, also called traps)
    - **`CLI, STI`** - disable/enable interrupts
  - From hardware components (Hardware interrupt)

- Hardware Interrupt is an electronic signal from device to CPU
  - From devices that finished some work
  - From timer etc.

Processing an Interrupt

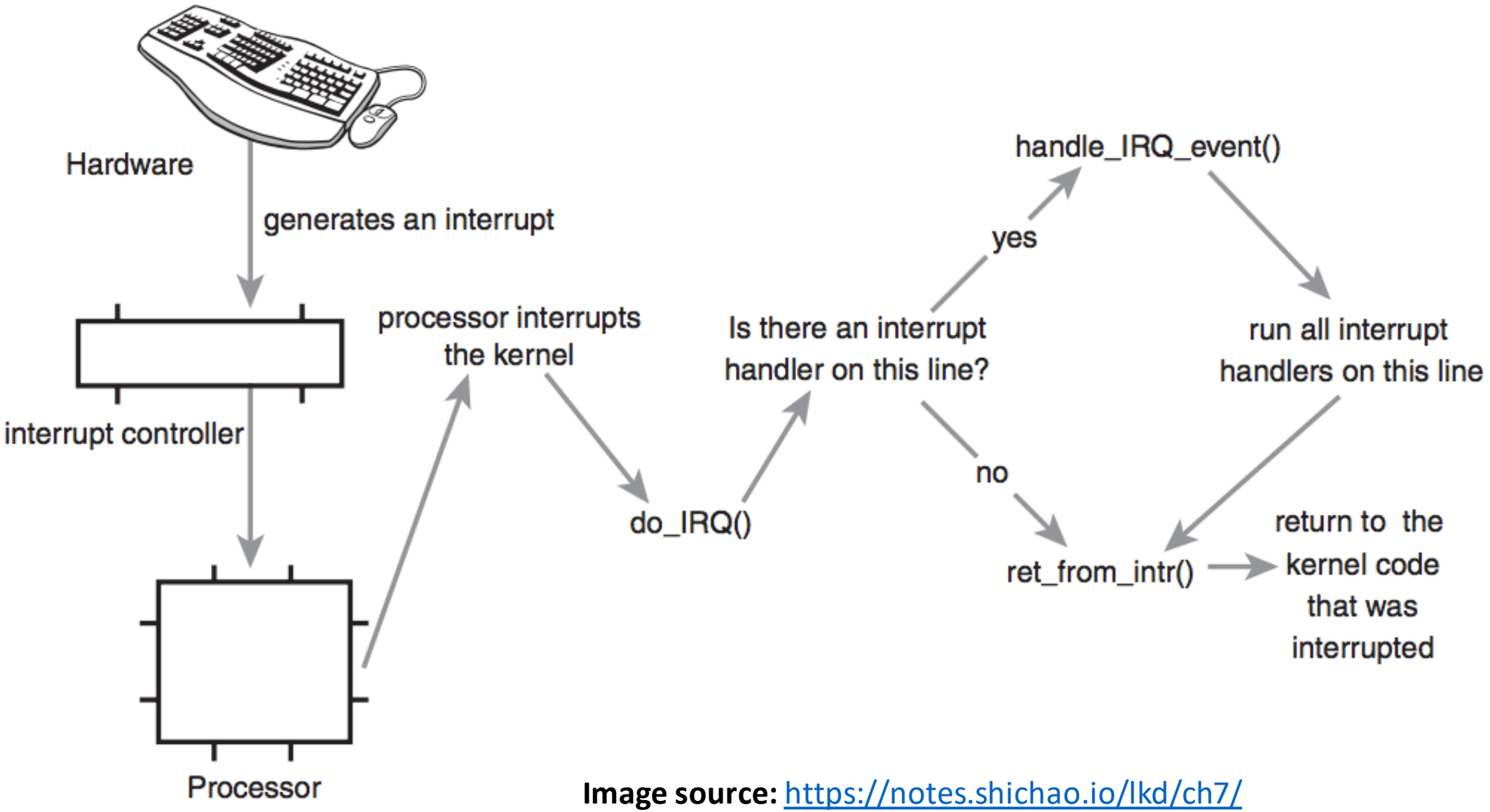


Image source: https://notes.shichao.io/lkd/ch7/

Kernel Services

- Kernel provides services by POSIX's defined system call interface specification but implementation of this specification differs (b/w different versions of Linux).

- Implementation can be anywhere:
  - In kernel
  - In module
  - or, sometimes even in user-space programs
    - E.g. the thread package is implemented in a library code.

- System call scenario
  - A process (running in user mode) needing kernel services can do system call.
  - During call, it is running in kernel mode. Switch to kernel mode normally done by **a trap** instruction through a stub procedure
  - Upon completion, switch back to user mode

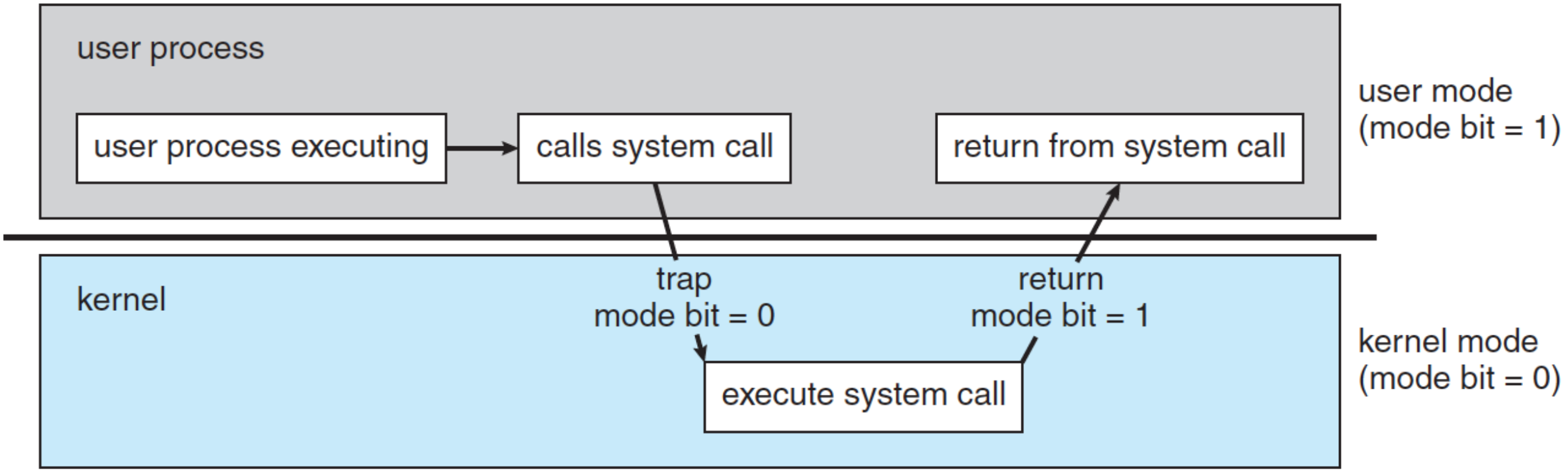
The dilemma?

- How can a user mode program switch the CPU to supervisor mode with the assurance that once the switch is done the CPU will be executing trusted kernel code and not untrusted user code?

The Path In and Out of the Kernel

- The *only* way to enter the operating kernel is to generate a processor interrupt. These interrupts come from several sources:
- *I/O devices:*
  - When a device, such as a disk or network interface, completes its current operation, it notifies the operating system by generating a processor interrupt.
- *Clocks and timers:*
  - Processors have timers that can be periodic (interrupting on a fixed interval) or count-down (set to complete at some specific time in the future). Periodic timers are often used to trigger scheduling decisions. For either of these types of timers, an interrupt is generated to get the operating system's attention.
- *Exceptions:*
  - When an instruction performs an invalid operation, such as divide-by-zero, invalid memory address, or floating point overflow, the processor can generate an interrupt.
- *Software Interrupts (Traps):*
  - Processors provide one or more instructions that will cause the processor to generate an interrupt.
  - Trap instructions are most often used to implement system calls and to be inserted into a process by a debugger to stop the process at a breakpoint.

The System Call Path



System Call Stub Functions

- The system call stub functions provide a high-level language interface to a function whose main job is to generate the software interrupt (trap) needed to get the kernel's attention.
- These functions are often called *wrappers*.
- The stub functions do the following:
  - set up the parameters,
  - trap to the kernel,
  - check the return value when the kernel returns, and
    - if no error: return immediately, else
    - if there is an error: set a global error number variable (called "errno") and return a value of -1.

Sequence of steps……

1. Switch the CPU to supervisor mode.
2. Look up a branch address in a kernel space table.
3. Branch to a trusted OS function.

Because the trap instruction is compiled into the stub procedure, a user-space program cannot easily determine the trap's destination address.

Also, it cannot branch directly to the kernel function – only through the system-provided stub.

The trap instruction

- A trap instruction is not a privileged instruction, so any program can execute a trap.

- However, the destination of the branch instruction is predetermined by a set of addresses that are kept in supervisory space and that are configured to point to kernel code

System Call Interface in Linux Kernel

- Use-space program calls a syscall stub, which contains a trap instruction
- For x86 code, the trap instruction is `syscall`, which acts analogous to a function call but with a difference
  - Instead of jumping to a function within the same program, `syscall` triggers a mode switch and jumps to a routine in the kernel portion of the memory
- The kernel validates the system call parameters and check the process's access permissions
  - Example: If the system call is a request to write a file, the kernel determines whether the user running the program is allowed to perform this action
- Once the kernel completes the system call, it uses `sysret` instruction
  - Unlike `ret`, `sysret` also changes the priviledge level

System Call Interface in the Linux Kernel

- Example
  - System call 0 is the read() system call
  - When a user-mode program executes the `read()` system call, the system will trigger a mode switch and jump to the `sys_read()` function within the Linux kernel

- x86 system call mechanics only use the system call number.
  - The name that associated with each number is just to give meaning to the programmer, just as we use function names instead of relying on memorization of hard-coded addresses.

- The names of the entry point functions in Linux are the names of the system calls with `sys_` prepended; for instance, the `open()` system call will call the `sys_open()` function in the kernel, and `mmap()` will call `sys_mmap()`.

System Call Interface in the Linux Kernel

- Each syscall is identified by a unique number; Linux maintains a table to map the syscall number to the name commonly used, and the entry point routine within the kernel itself
  - Check https://elixir.bootlin.com/linux/v5.10.188/source/arch/x86/entry/syscalls/syscall_64.tbl

```
#
# 64-bit system call numbers and entry vectors
#
# The format is:
# <number> <abi> <name> <entry point>
#
# The __x64_sys_*() stubs are created on-the-fly for sys_*() system calls
#
# 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
3       common  close                   sys_close
4       common  stat                    sys_newstat
5       common  fstat                   sys_newfstat
6       common  lstat                   sys_newlstat
7       common  poll                    sys_poll
8       common  lseek                   sys_lseek
9       common  mmap                    sys_mmap
10      common  mprotect                sys_mprotect
```

System Call Interface in the Linux Kernel

- The names of the system calls correspond to many common C standard library functions.
  - `open()` and `close()` are the system calls that are used to establish connections to files,
  - `socket()` is the system call to create a socket for network communication,
  - `exit()` can be used to terminate the current process.
  - That is, many C functions are simply wrappers for system calls.

System Call Interface in the Linux Kernel

- In contrast, many C functions are implemented to provide additional functionality on top of system calls.
  - `printf()`: the code will eventually trigger the `write()` system call.
  - Primary difference: `write()` requires low-level details of how the system is being used that `printf()` abstracts away.
  - In addition, calling `write()` requires exact knowledge of the length of the message to be printed, whereas `printf()` does not.
  - In summary, many C standard library functions provide a thin wrapper for invoking system calls, while other functions do not.

Calling System Calls in Assembly

- Arguments are passed to the system call using the general-purpose registers and the stack as needed
  - System call number is stored into the `%rax` register
  - Example: Print "Hello World" in `stdout`

```
# An assembly-language “hello world” program with system
calls

  .global _start

  .text
_start:
  # write(1, message, 13)
  mov $1, %rax           # system call 1 is write
  mov $1, %rdi           # file handle 1 is stdout
  mov $message, %rsi     # address of string to output
  mov $13, %rdx          # number of bytes
  syscall                # invoke OS to write to stdout

  # exit(0)
  mov $60, %rax          # system call 60 is exit
  xor %rdi, %rdi         # we want return code 0
  syscall                # invoke OS to exit

  .data
message:
  .ascii "Hello, world\n"
```

System Call Wrappers in Glibc

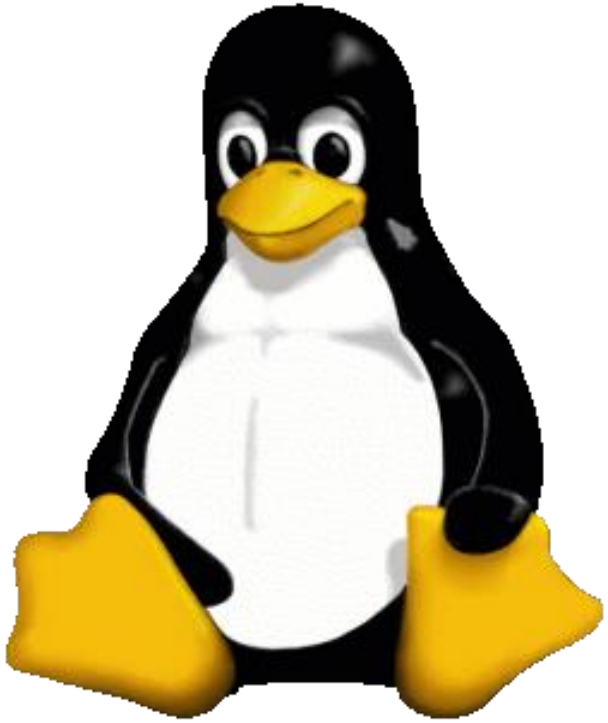
- Three types of wrappers used in glibc
  - **Assembly syscalls:** Translated from a list of names to the assembly wrappers (example shown in the previous slide)
    - Example: `socket()` syscall
  - **Macro syscalls:** Define macros to perform special operations before/after the syscalls; for example, to shuffle the return value to make it suitable for the user-space structure
    - Example: `write()` syscall
    - Check the corresponding example macro here -- https://codebrowser.dev/glibc/glibc/sysdeps/unix/sysv/linux/write.c.html
    - `SYSCALL_CANCEL` implementa cancellable syscalls (syscalls that can be cancelled after they are raised, say, through interrupts)

Syscall Implementation (Glibc + Kernel)

- Glibc uses an assembly code to trap and execute the syscall functions
  - Check https://codebrowser.dev/glibc/glibc/sysdeps/unix/sysv/linux/x86_64/syscall.S.html

- Kernel implementation of the `write()` syscall
  - https://elixir.bootlin.com/linux/v5.10.188/source/fs/read_write.c#L646

The Linux Kernel: Process Management



Daemons

- There is no special “kernel process” that executes the kernel code.
  - Kernel code is actually induced by normal user processes.
- Daemons are processes that do not belong to a user process.
- Started when machine is started
- Used for functioning in areas such as for
  - Processing network packets
  - Logging of system and error messages etc.
  - Normally, filename ends with d, such as
    - inetd, syslogd, crond, lpd etc

Process Descriptors

- The kernel maintains info about each process in a process descriptor, of type **`task_struct`**.
  - See **`include/linux/sched.h`**
  - https://elixir.bootlin.com/linux/v5.10/source/include/linux/sched.h#L640
  - Each process descriptor contains info such as run-state of process, address space, list of open files, process priority etc…

```
struct task_struct {
  volatile long state; /* -1 unrunnable, 0 runnable, >0 stopped */
  unsigned long flags; /* per process flags */
  mm_segment_t addr_limit; /* thread address space:
                      0-0xBFFFFFFF for user-thead
                      0-0xFFFFFFFF for kernel-thread */
  struct exec_domain *exec_domain;
  long need_resched;
  long counter;
  long priority;
  /* SMP and runqueue state */
    struct task_struct *next_task, *prev_task;
    struct task_struct *next_run,  *prev_run;
    ...
  /* task state */
  /* limits */
  /* file system info */
  /* ipc stuff */
  /* tss for this task */
  /* filesystem information */
  /* open file information */
  /* memory management info */
  /* signal handlers */
     ...
};
```

Contents of process descriptor

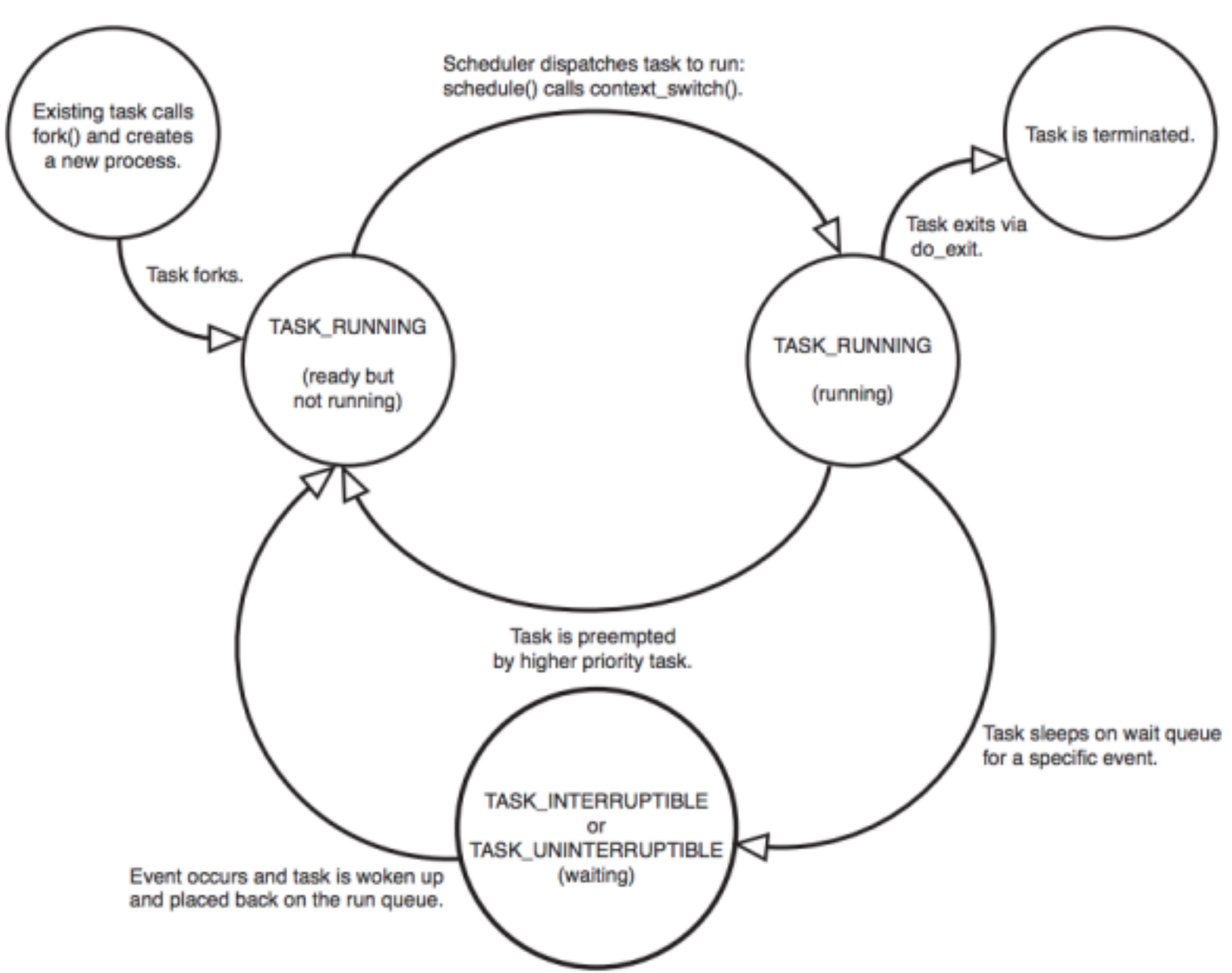
Process State

- Consists of an array of mutually exclusive flags
  - implies exactly one **`state`** flag is set at any time.
  - https://elixir.bootlin.com/linux/v5.10/source/include/linux/sched.h#L80

Process State

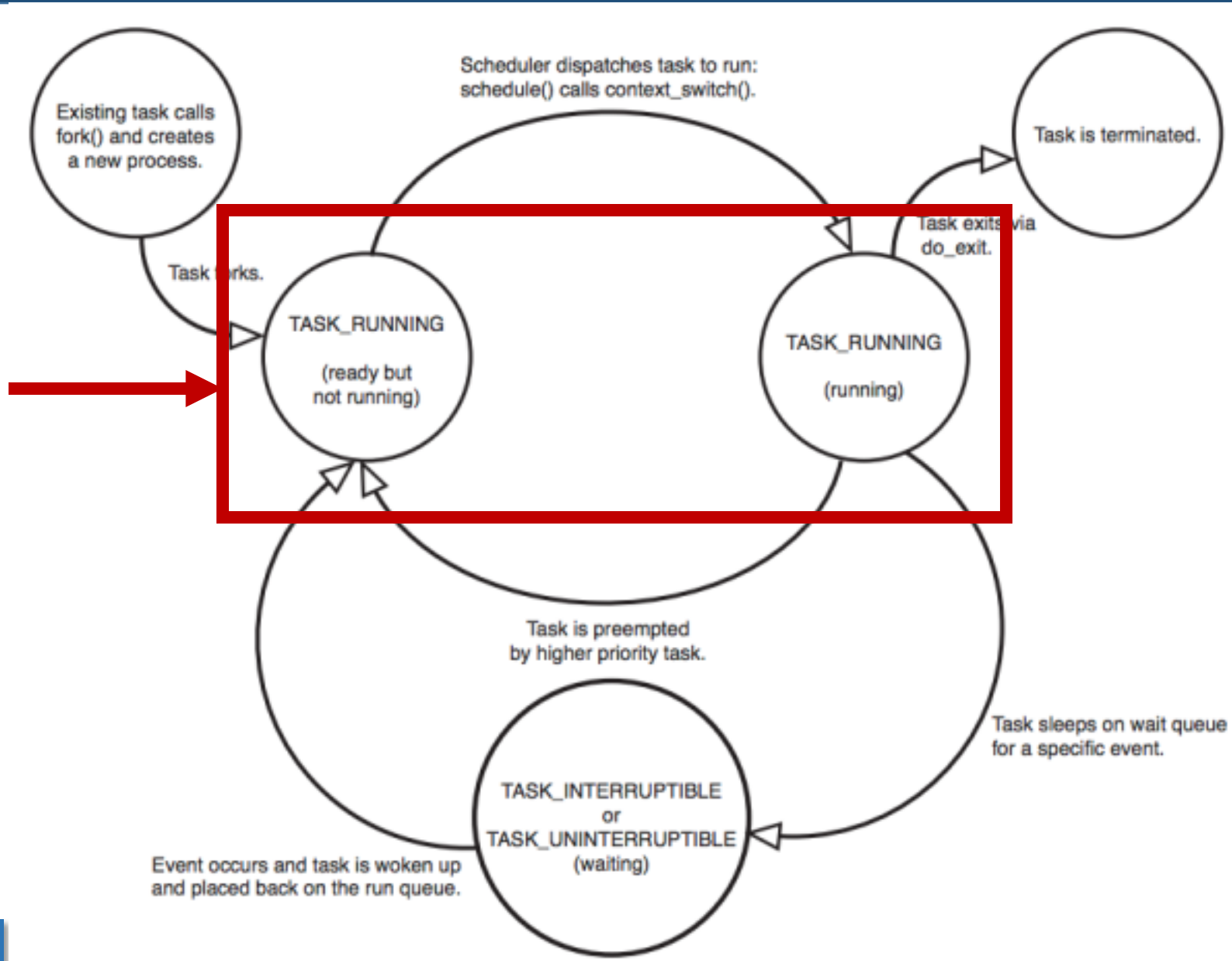
- **`state`** values:
  - **`TASK_NEW`** (new task)
  - **`TASK_RUNNING`** (executing on CPU or runnable).
  - **`TASK_INTERRUPTIBLE`** (waiting on a condition: interrupts, signals and releasing resources may "wake" process).
  - **`TASK_UNINTERRUPTIBLE`** (Sleeping process cannot be woken by a signal).
  - **`TASK_NOLOAD`** (uninterruptible tasks that doesn't contribute to load average, say idle kernel processes)
  - **`TASK_STOPPED`** (task execution has stopped).
  - **`TASK_TRACED`** (task is being traced by a debugger)
  - **`EXIT_ZOMBIE`** (process has completed execution but still in the process table, reaped out by the parent later on).

Process States in Linux Kernel

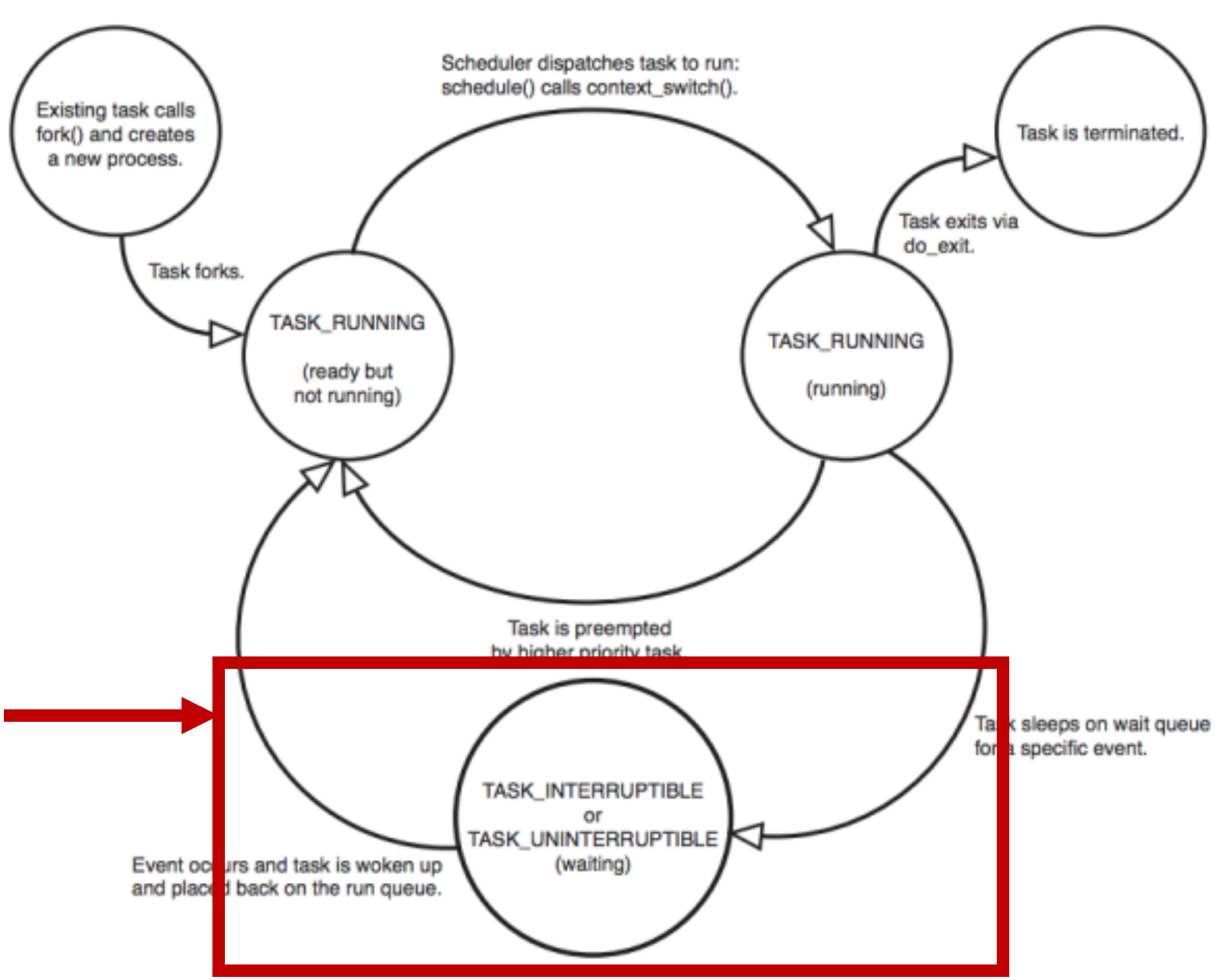


Process States in Linux Kernel

A process in user mode can only be in these states

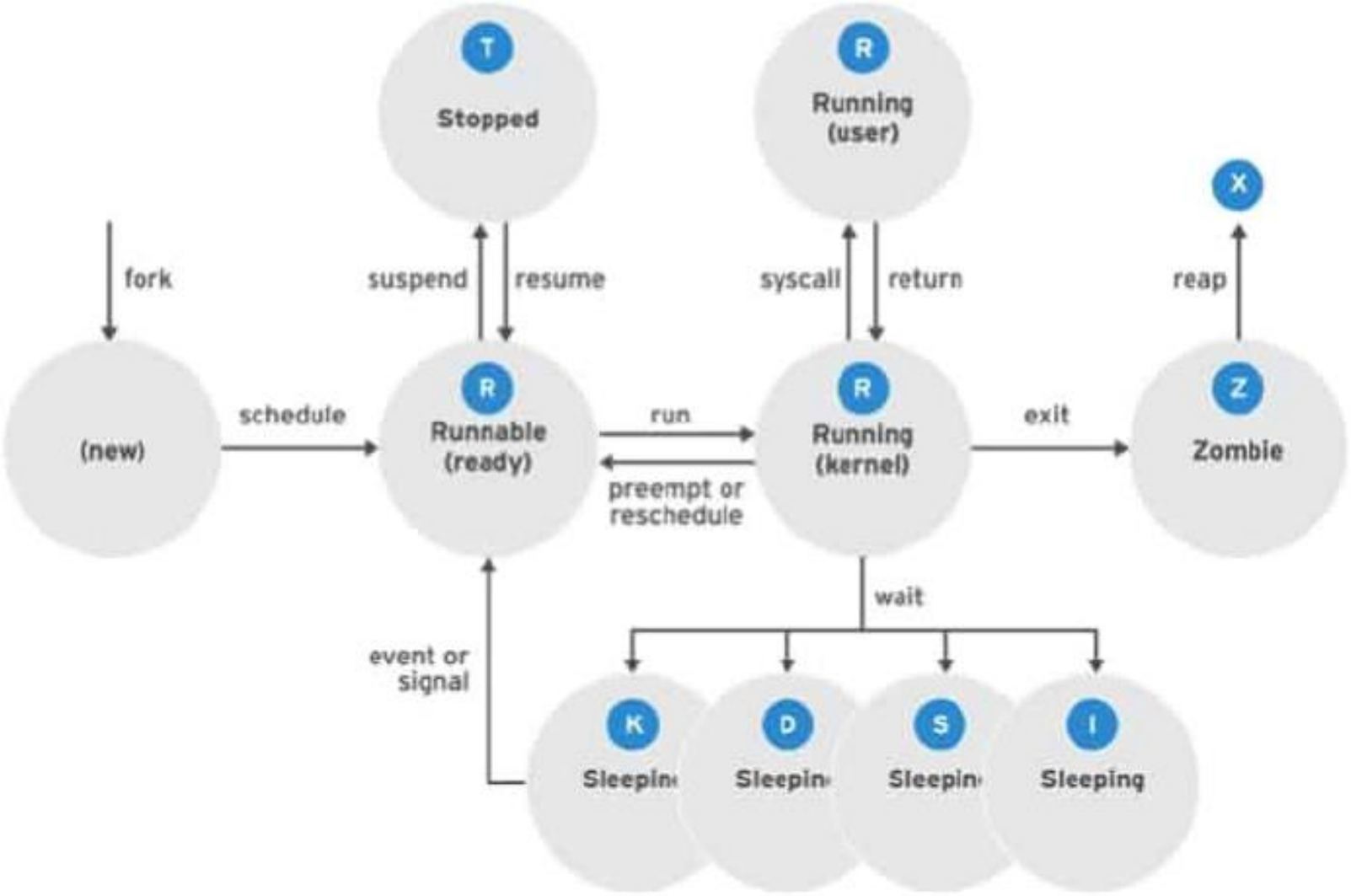


Process States in Linux Kernel



The user process moves to the kernel mode

A Detailed View of the Process States (User to Kernel Mode)



R	TASK_RUNNING
S	TASK_INTERRUPTABLE
D	TASK_UNINTERRUPTABLE
K	TASK_WAKEKILL
I	TASK_IDLE
T	TASK_STOPPED or TASK_TRACED
Z	EXIT_ZOMBIE
X	EXIT_DEAD

https://www.thegeeksearch.com/beginners-guide-to-managing-processes-running-on-a-linux-system/

Check Process States (top)

PID	USER	PR	NI	VIRT	RES	SHR	S	%CPU	%MEM	TIME+	COMMAND
3025	mysql	20	0	23.6g	211352	13120	S	8.4	0.1	366:19.98	mysqld
4626	gnocchi	20	0	691368	82824	3992	S	6.5	0.0	43:36.10	gnocchi-m+
4630	gnocchi	20	0	691368	82820	3992	S	6.5	0.0	43:42.22	gnocchi-m+
4634	gnocchi	20	0	691368	82824	3992	S	6.5	0.0	43:44.67	gnocchi-m+
5477	user	20	0	11.2g	3.8g	3.3g	S	5.8	1.5	405:36.10	VBoxHeadl+
2364	**gnocchi**	**20**	**0**	**412132**	**64004**	**8960**	**R**	**2.3**	**0.0**	**229:47.32**	**gnocchi-m+**
241678	**user**	**20**	**0**	**163772**	**3960**	**1576**	**R**	**2.3**	**0.0**	**0:01.36**	**top**
2295	nova	20	0	486120	103248	7288	S	1.9	0.0	181:06.93	nova-cond+
2305	nova	20	0	546388	131784	8808	S	1.9	0.0	188:31.61	nova-api
2373	nova	20	0	492184	109544	7300	S	1.9	0.0	171:08.88	nova-sche+
2351	glance	20	0	452368	101992	7284	S	1.6	0.0	171:15.94	glance-re+
2369	ceilome+	20	0	538372	73136	11244	S	1.6	0.0	138:50.21	ceilomete+
2430	glance	20	0	530088	115752	9540	S	1.6	0.0	172:41.15	glance-api
2569	aodh	20	0	387540	66024	7084	S	1.6	0.0	138:27.88	aodh-eval+
2570	aodh	20	0	387540	66024	7084	S	1.6	0.0	138:30.44	aodh-list+
2297	aodh	20	0	387536	66024	7084	S	1.3	0.0	138:29.81	aodh-noti+
9	root	20	0	0	0	0	S	0.6	0.0	34:08.31	rcu_sched

Check Process States (ps –aux)

```
root      239908  0.0  0.0      0     0 ?        S    11:34   0:00 [kworker/14:0]
root      240412  0.0  0.0      0     0 ?        S    11:37   0:00 [kworker/46:1]
root      240672  0.0  0.0      0     0 ?        S    11:39   0:00 [kworker/14:2]
root      240678  0.0  0.0      0     0 ?        S    11:40   0:00 [kworker/u626:
root      241606  0.2  0.0 171236  5872 ?        Ss   11:46   0:00 sshd: user [pr
user      241611  0.0  0.0 171236  2532 ?        R    11:47   0:00 sshd: user@pts
root      241614  0.0  0.0      0     0 ?        S    11:47   0:00 [kworker/u625:
user      241615  0.1  0.0 117172  3684 pts/0    Ss   11:47   0:00 -bash
root      241621  0.0  0.0      0     0 ?        S    11:47   0:00 [kworker/u626:
root      241644  0.0  0.0 350536  6704 ?        Sl   11:47   0:00 /usr/sbin/abrt
root      241818  0.0  0.0 108056   356 ?        S    11:48   0:00 sleep 60
user      241823  0.0  0.0 165720  1876 pts/0    R+   11:48   0:00 ps -aux
root      244189  0.0  0.0      0     0 ?        S    Aug18   0:00 [kworker/45:1]
root      253078  0.0  0.0      0     0 ?        S<   Aug15   0:00 [kworker/49:1H
root      254134  0.0  0.0      0     0 ?        S<   Aug15   0:00 [kworker/82:1H
root      262642  0.0  0.0      0     0 ?        S    Aug18   0:00 [kworker/50:2]
root      263913  0.0  0.0      0     0 ?        S<   Aug15   0:00 [kworker/86:1H
root      273219  0.0  0.0      0     0 ?        S    Aug19   0:00 [kworker/33:0]
root      275849  0.0  0.0      0     0 ?        S    Aug19   0:00 [kworker/37:0]
```

Process State Codes in ps Command

```
PROCESS STATE CODES
       Here are the different values that the s, stat and state output
       specifiers (header "STAT" or "S") will display to describe the state of
       a process:

               D    uninterruptible sleep (usually IO)
               R    running or runnable (on run queue)
               S    interruptible sleep (waiting for an event to complete)
               T    stopped by job control signal
               t    stopped by debugger during the tracing
               W    paging (not valid since the 2.6.xx kernel)
               X    dead (should never be seen)
               Z    defunct ("zombie") process, terminated but not reaped by
                    its parent

       For BSD formats and when the stat keyword is used, additional
       characters may be displayed:

               <    high-priority (not nice to other users)
               N    low-priority (nice to other users)
               L    has pages locked into memory (for real-time and custom IO)
               s    is a session leader
               l    is multi-threaded (using CLONE_THREAD, like NPTL pthreads
                    do)
               +    is in the foreground process group
```

Process Identification

- Each process, or independently scheduled execution context, has its own process descriptor.
  - Process descriptor addresses are used to identify processes.
  - Processes are *dynamic*, so descriptors are kept in dynamic memory.
  - An ~10KB memory area is allocated for each process, to hold process descriptor *and* kernel mode process stack.
- Process ids (or **PID**s) are 32-bit numbers, also used to identify processes.
  - For compatibility with traditional UNIX systems, LINUX uses PIDs in range 0..32767 (default maximum for 32-bit systems). You can check the MAX PID for a system: **`cat /proc/sys/kernel/pid_max`**
  - **`fork()`** will return a –1 and error code **`EAGAIN`** if it cannot allocate a process ID

PID Namespace

- PID namespaces isolate the process ID number space, meaning that processes in different PID namespaces can have the same PID
  - Example: Used in Linux Containers – the processes within a container share the same process ID (PID of the container) outside the container; internally, they use a PID namespace
  - The kernel makes room for up to 32 nested PID namespaces.

- Note the difference between **`pid_t`** and the **`struct pid`** within **`task_struct`**
  - https://elixir.bootlin.com/linux/v5.10/source/include/linux/sched.h#L835
  - https://elixir.bootlin.com/linux/v5.10/source/include/linux/sched.h#L871
  - **`pid_t`** is a unique integer to identify a process, whereas the **`pid`** structure holds a mapping from the process ID to the process descriptor

PID Namespace

- A **`struct pid`** is the kernel's internal notion of a process identifier.
  - It refers to individual tasks, process groups, and sessions.
- While there are processes attached to it, the **`struct pid`** lives in a hash table, so it and then the processes that it refers to can be found quickly from the numeric **`pid`** value.

PID Namespace

- A **`struct pid`** is the kernel's internal notion of a process identifier.
  - It refers to individual tasks, process groups, and sessions.
- While there are processes attached to it, the **`struct pid`** lives in a hash table, so it and then the processes that it refers to can be found quickly from the numeric **`pid`** value.

- To access a process, we need a reference to the process descriptor (or the **`task_struct`**). Why do we keep a separate PID to access the processes?

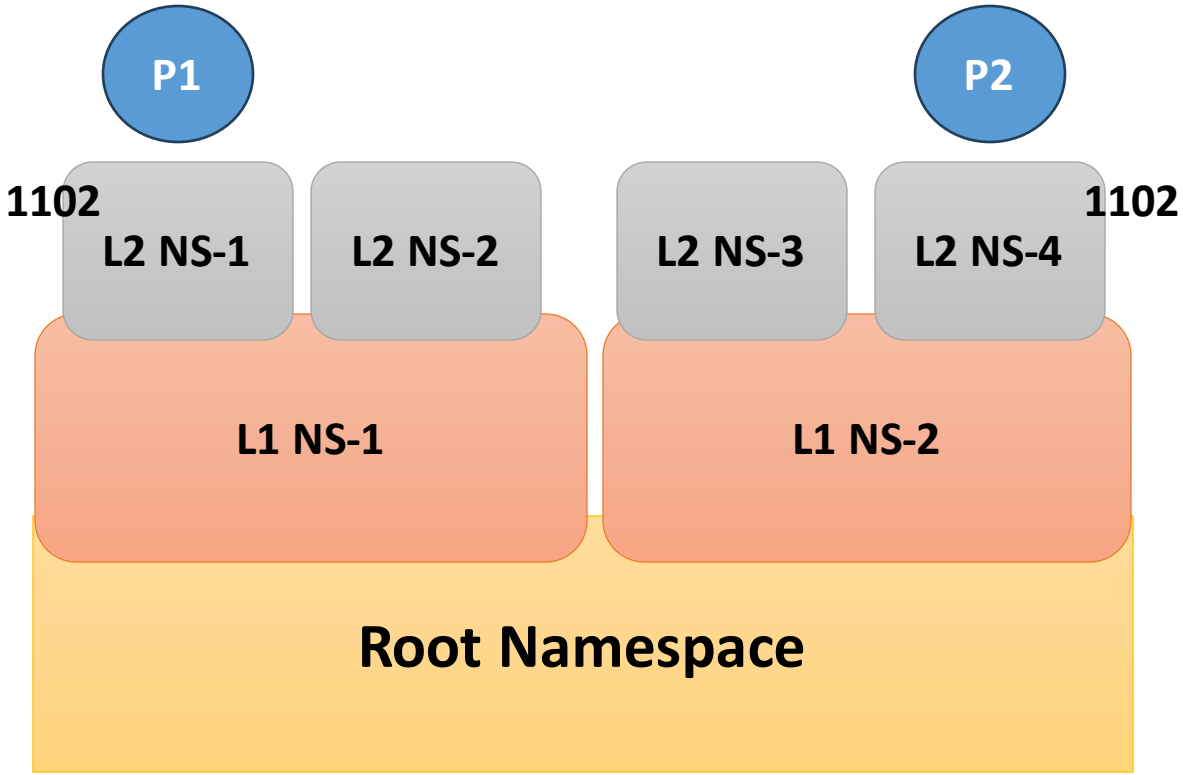
PID Namespace

- Storing **`pid_t`** values in the kernel and referring to them later:
  - The process originally with that **`pid`** may have exited and the pid allocator wrapped, and another process could have come along and been assigned that **`pid`**.

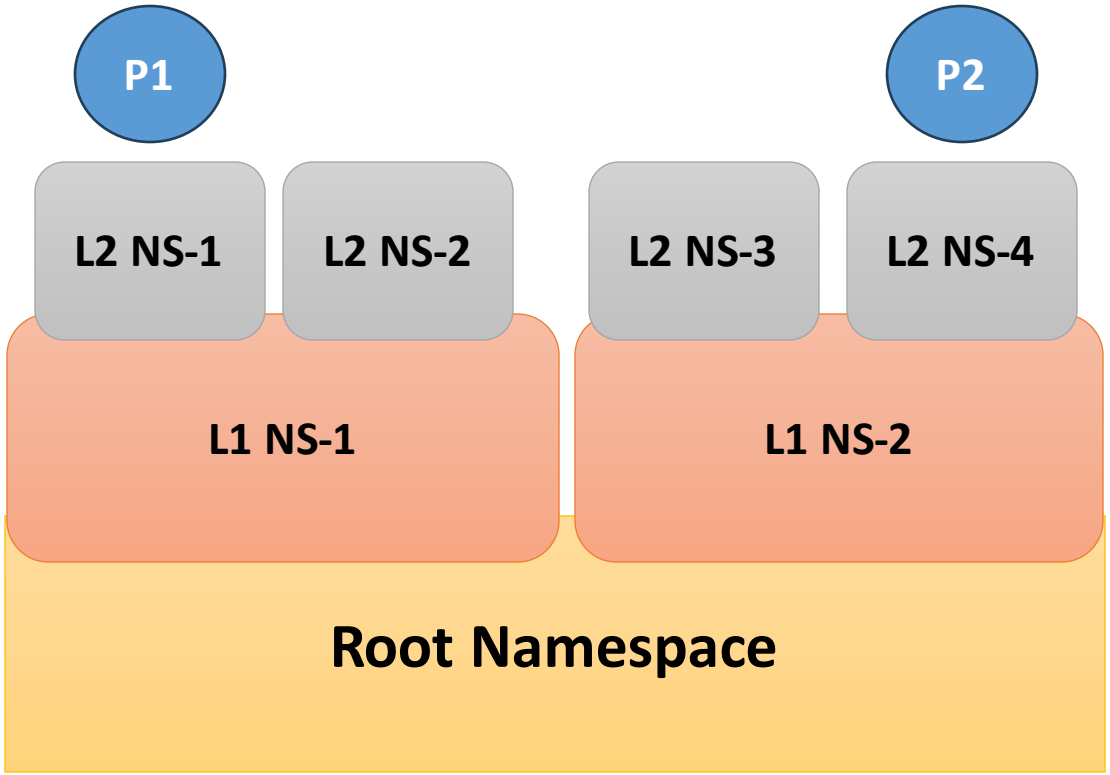
- Referring to user-space processes by holding a reference to **`struct task_struct`**
  - When the user space process exits, the **`task_struct`** is still maintained for some times, which consumes around 10K of low kernel memory. By comparison a **`struct pid`** is about 64 bytes.

- Solves both the problems
  - It is small so holding a reference does not consume a lot of resources, and since a new **`struct pid`** is allocated when the numeric **`pid`** value is reused (when **`pids`** wrap around) we don't mistakenly refer to new processes.

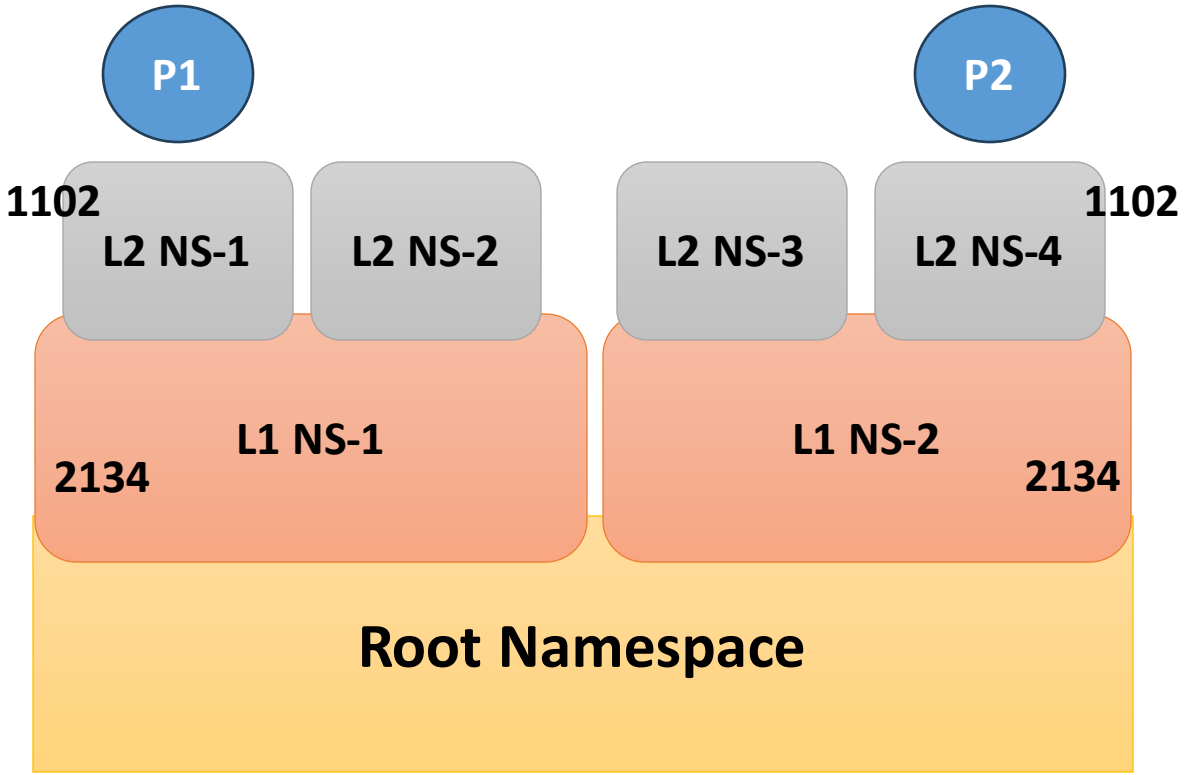
PID Namespace



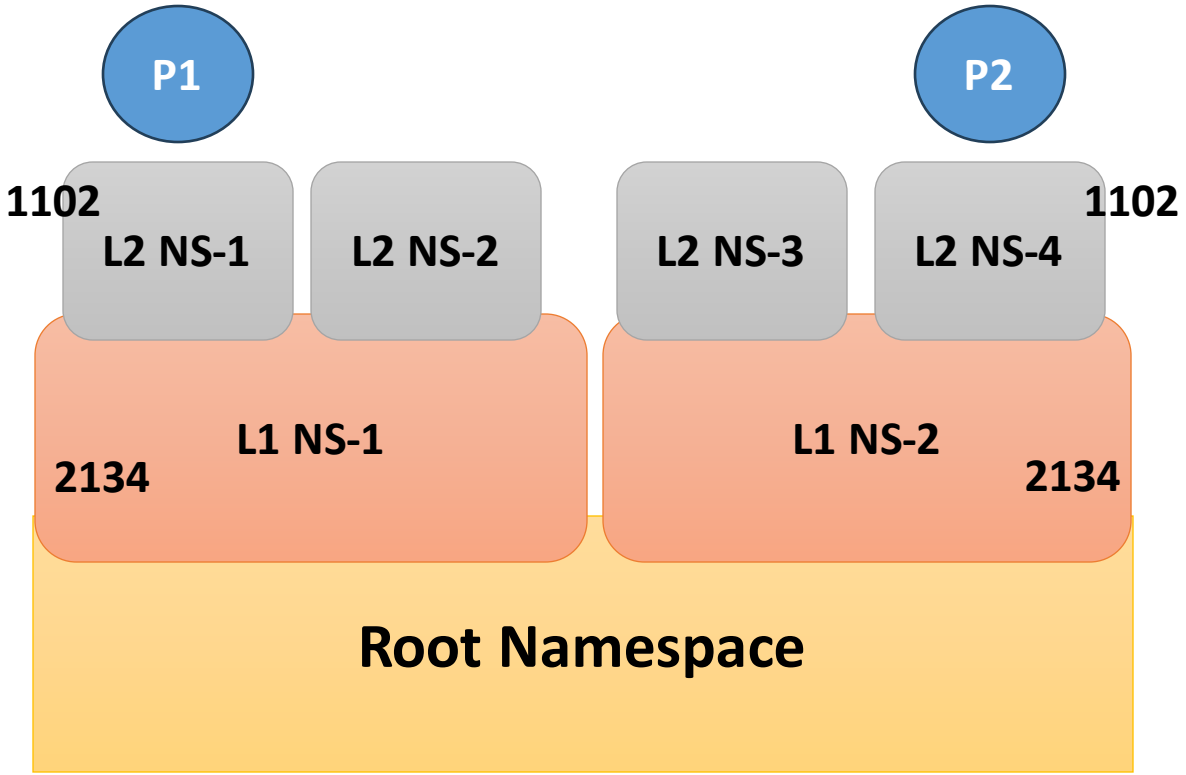
PID Namespace



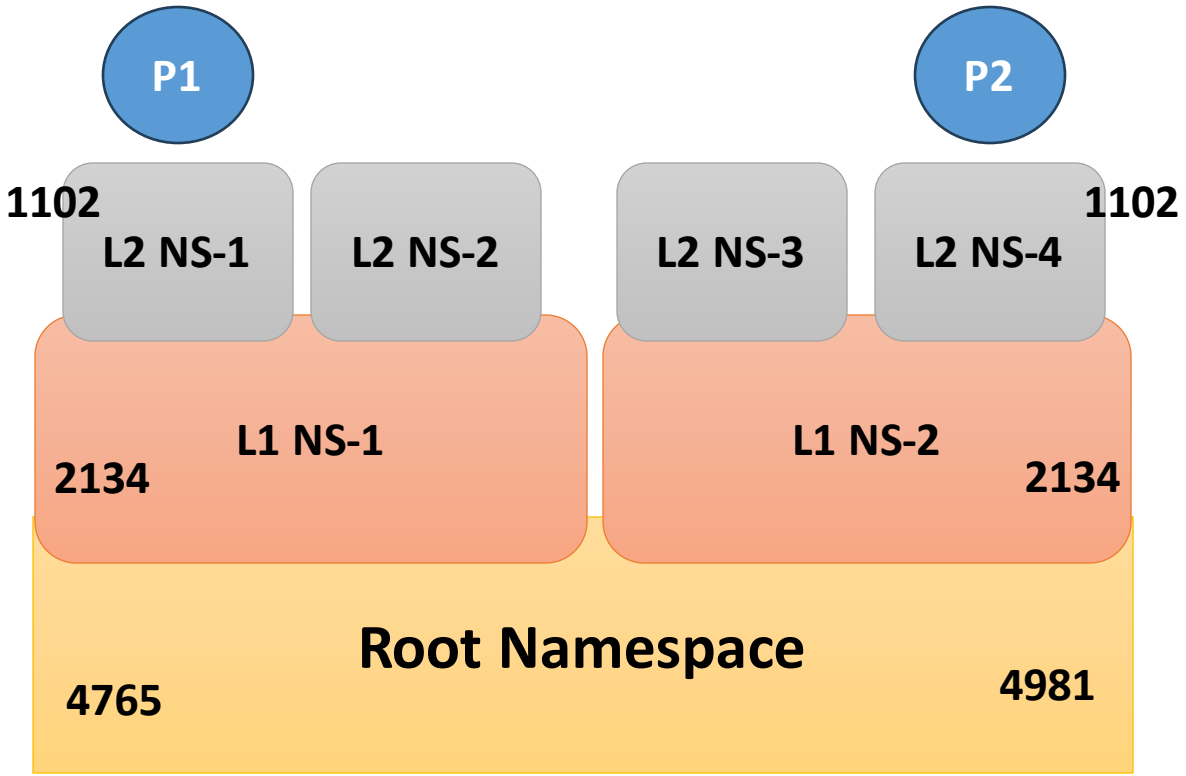
PID Namespace



PID Namespace



PID Namespace



- When a process is forked on a PID namespace, we need to allocate a PID at every nested namespace in the hierarchy, up to the root namespace.
- **`struct pid`** will reference all the tasks that use the PID at different namespaces.

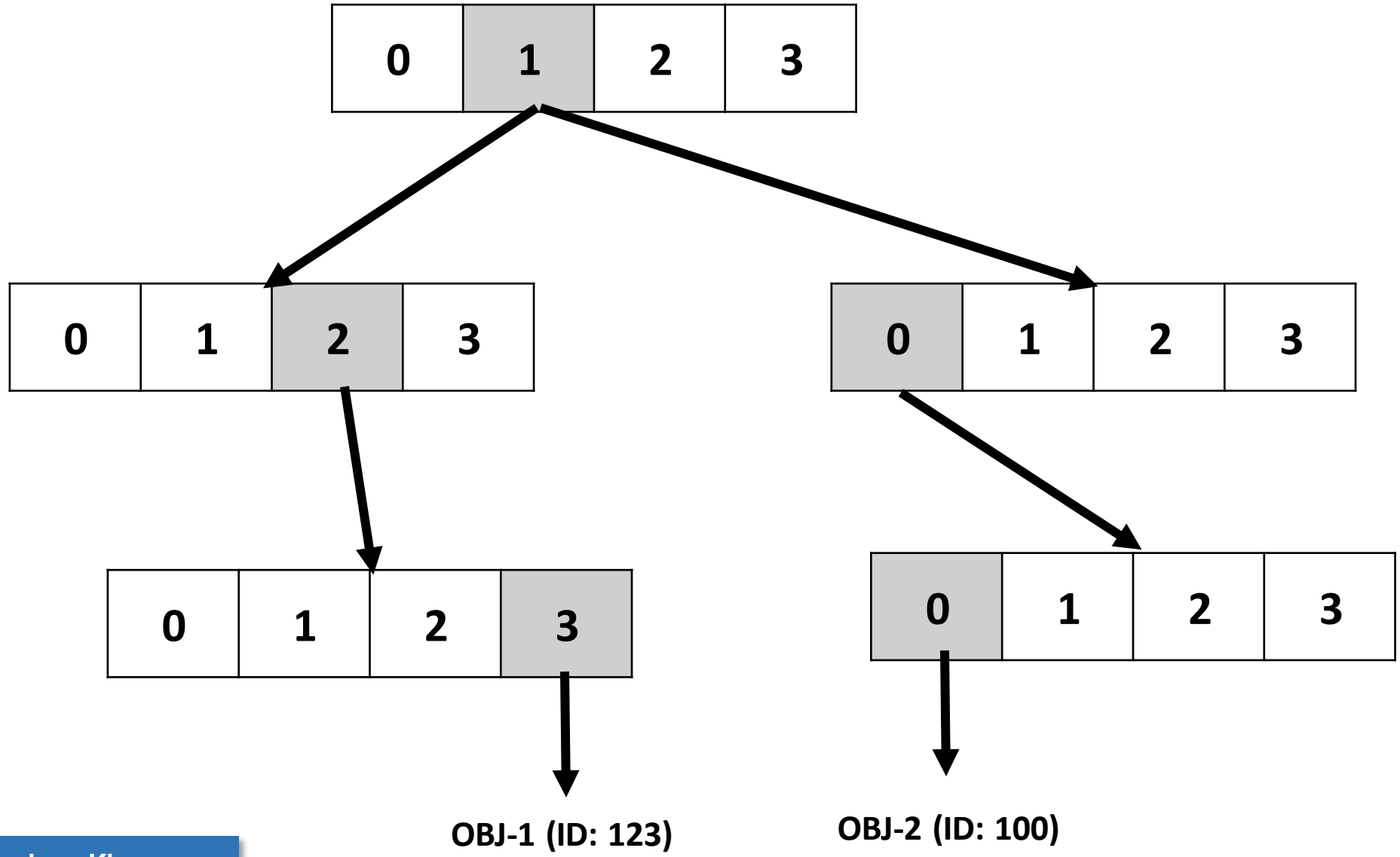
Allocating the PID

- The fork() syscall allocate a PID through **`alloc_pid()`**
  - https://elixir.bootlin.com/linux/v5.10.191/source/kernel/fork.c#L2134
  - The **`alloc_pid`** calls **`kmem_cache_alloc`** – which allocates all the structures from a cache (fast allocation as PIDs are frequently allocated, avoid kernel memory allocation)
  - It then iterate through all the namespaces starting from the current PID namespace to the root PID namespace and populate the numeric PID values for different namespaces.
    - Increment (+1) the last PID being used
    - If reaches the MAX, wrap around and search for an available PID value
    - Return –1 with error code as EAGAIN if no PID is available for a namespace
    - Use the IDR API in the kernel that allocates a free ID from a range; check https://elixir.bootlin.com/linux/v5.10.191/source/kernel/pid.c#L212

The IDR API

- **Goal:** Allocate an integer ID with a pointer
- **Use:** Several places in the Linux kernel, say associate an integer ID with device names (device ID to device name), file system blocks (inode to file system blocks), and *process IDs* (`struct pid` to `pid_t`)
  - Historically, a chained hased table was used to map `pid_t` to struct `pid`; later it got replaced by the IDR API for better efficiency and storage performance.
- Check https://elixir.bootlin.com/linux/v5.10.191/source/include/linux/pid.h (and pid.c)
- Internally, the IDR API uses a Radix tree, **why not array or list?**
  - The non-leaf nodes contain the integer ID, the leaf points to the object

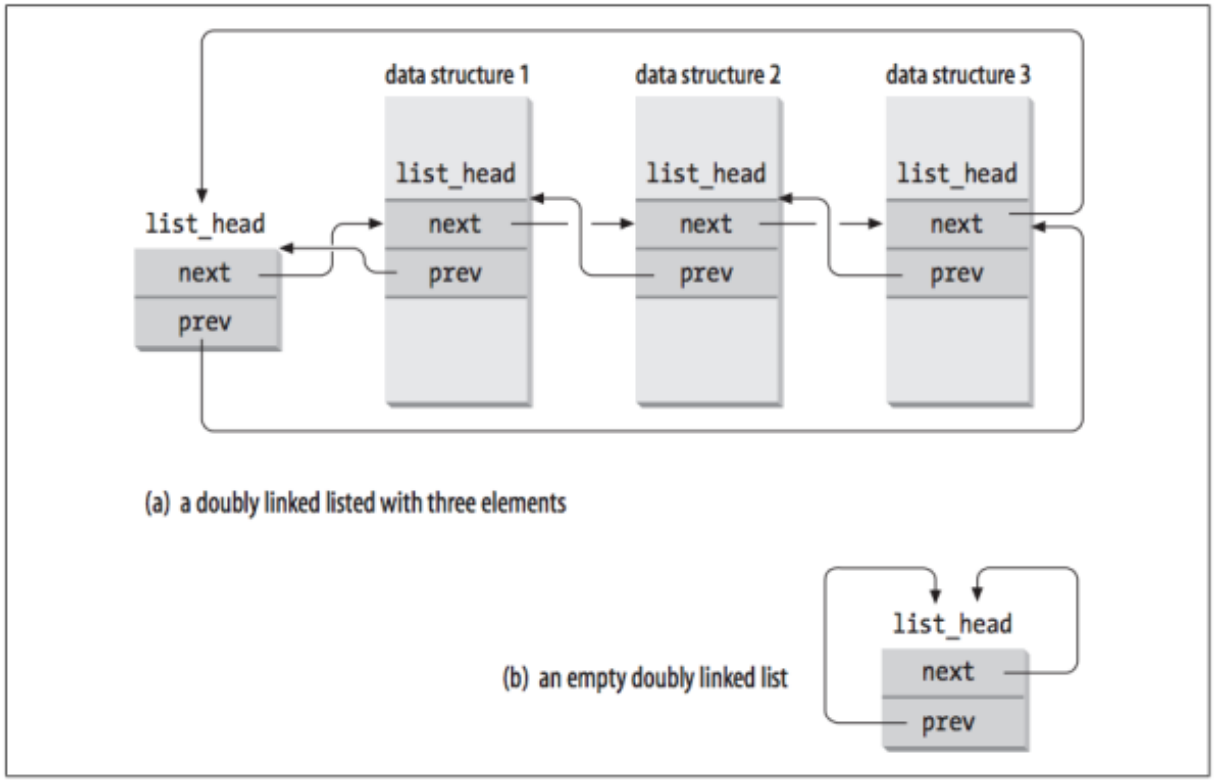
The IDR API (Example Radix tree, code 0-3)



The Process List

- The *process list* (of all processes in system) is a doubly-linked list.
  - **`prev`** & **`next`** fields under the **`list_head`** structure are used to build list.
  - **`list_for_each()`** macro scans whole list.
  - Check https://elixir.bootlin.com/linux/v5.10/source/include/linux/plist.h#L145

```
struct task_struct *task;
struct list_head *list;
list_for_each(list, &current->children)
{
task = list_entry(list, struct
task_struct, sibling);
/* task now points to one of current's
children */
}
```



The Run Queue

- Processes are scheduled for execution from a doubly-linked list of **`TASK_RUNNING`** processes, called the *runqueue* (**`struct rq`**).
  - Check: https://elixir.bootlin.com/linux/v5.10/source/kernel/sched/sched.h#L895
- Maintain scheduling-specific structures of the processes
  - CFS runqueue uses the **`curr`** and the **`next`** pointers to maintain the list of runnable processes;
    check https://elixir.bootlin.com/linux/v5.10/source/kernel/sched/sched.h#L519
  - RT runqueue maintains a priority queue;
    check https://elixir.bootlin.com/linux/v5.10/source/kernel/sched/sched.h#L619
- **`struct sched_class`** maintains a set of function pointers to perform the operations on the runqueue
  - Check https://elixir.bootlin.com/linux/v5.10/source/kernel/sched/sched.h#L1779

Wait Queues

- **`TASK_(UN)INTERRUPTIBLE`** processes are grouped into classes that correspond to specific events.
  - e.g., timer expiration, resource now available.
  - There is a separate wait queue for each class / event.
  - Processes are “woken up” when the specific event occurs.

Wait Queues

- Simple Wait Queues: Allows deterministic behavior (bounded IRQ and lock hold times)
  - Check https://elixir.bootlin.com/linux/v5.10.191/source/include/linux/swait.h#L43

- Generic Wait Queue implementation:
  - https://elixir.bootlin.com/linux/v5.10.191/source/include/linux/wait.h

Process Switching

- Part of a process's execution context is its *hardware context* i.e., register contents.
  - The task state segment (**`tss`**) and kernel mode stack save hardware context.
    - **`tss`** holds hardware context not automatically saved by hardware (i.e., CPU).
- *Process switching* involves saving hardware context of **`prev`** process (descriptor) and replacing it with hardware context of **`next`** process (descriptor).
  - Needs to be fast!
  - Recent Linux versions override hardware context switching using software (sequence of **`mov`** instructions), to be able to validate saved data and for potential future optimizations.

The `switch_to` Macro

- **`switch_to()`** performs a process switch from the **`prev`** process (descriptor) to the **`next`** process (descriptor).
- **`switch_to`** is invoked by **`schedule()`** & is one of the most hardware-dependent kernel routines.
  - See https://elixir.bootlin.com/linux/v5.10.191/source/include/asm-generic/switch_to.h
  - Architecture-specific implementations:
    - https://elixir.bootlin.com/linux/v5.10.191/source/arch/arc/include/asm/switch_to.h#L15
    - https://elixir.bootlin.com/linux/v5.10.191/source/arch/ia64/include/asm/switch_to.h#L37

Creating Processes

- Traditionally, resources owned by a parent process are duplicated when a child process is created.
  - *It is slow* to copy whole address space of parent.
    - *It is unnecessary*, if child (typically) immediately calls **`execve()`**, thereby replacing contents of duplicate address space.
- Cost savers:
  - *Copy on write* – parent and child share pages that are read; when either writes to a page, a new copy is made for the writing process.
  - *Lightweight processes* – parent & child share page tables (user-level address spaces), and open file descriptors.

Clone Flags

- A set of flags that define the clone properties
  - **`CLONE_VM`** – memory is shared among the parent and child processes
  - **`CLONE_FS`** – file system information is shared among the processes
  - **`CLONE_FILES`** – open files are shared among the processes
  - **`CLONE_SIGHAND`** – signal handlers are shared among the processes
  - **`CLONE_VFORK`** – parent is blocked until child releases the memory
  - **`CLONE_NEWPID`** – New PID namespace will be used for the child
  - Check all the flags here: https://elixir.bootlin.com/linux/v5.10.191/source/include/uapi/linux/sched.h#L26

Creating Processes

- **`kernel_clone()`** is the main fork routine to create a child process
  - Check https://elixir.bootlin.com/linux/v5.10.191/source/kernel/fork.c#L2457
  - It takes a set of arguments defined through the structure **`kernel_clone_args`**
  - Check https://elixir.bootlin.com/linux/v5.10.191/source/include/linux/sched/task.h#L21
  - The above routine check the flags and call **`copy_process()`** routine to copy the content of the parent memory to the child memory
  - Assign a **`pid`** by calling **`alloc_pid()`** -- check https://elixir.bootlin.com/linux/v5.10.191/source/kernel/pid.c#L159
  - Put the task in the runqueue by calling **`wake_up_new_task()`**; check https://elixir.bootlin.com/linux/v5.10.191/source/kernel/sched/core.c#L3355

`fork()` and `vfork()`

- **`fork()`** is implemented as a **`kernel_clone()`** syscall with **`SIGCHLD`** sighandler set, all clone flags are cleared (no sharing) and **`child_stack`** is 0 (let kernel create stack for child on copy-on-write).
- **`vfork()`** is like **`fork()`** with **`CLONE_VM`** & **`CLONE_VFORK`** flags set.
  - With **`vfork()`** child & parent share address space; parent is blocked until child exits or executes a new program.
  - Parent call a subroutine **`wait_for_vfork_done()`** to check the child status and wait till child exits;
    check https://elixir.bootlin.com/linux/v5.10.191/source/kernel/fork.c#L1267

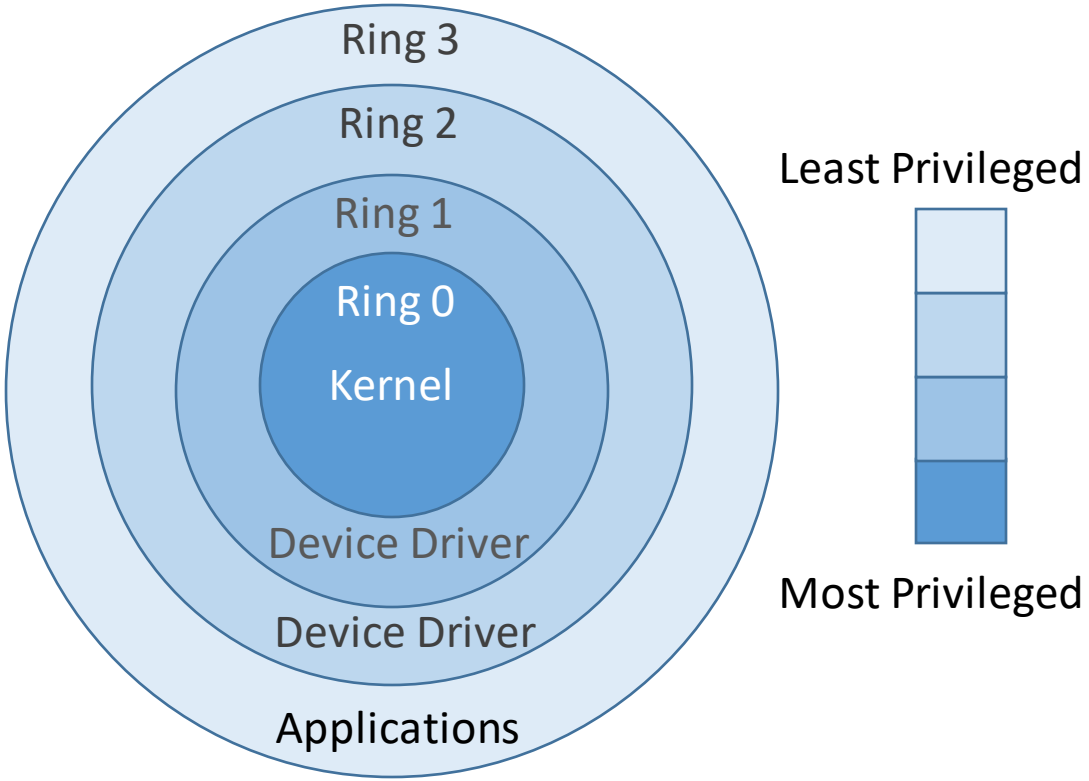
Kernel Threads

- Some (background) system processes run only in kernel mode.
  - e.g., flushing disk caches, swapping out unused page frames.
  - Can use *kernel threads* for these tasks.
- Kernel threads only execute kernel functions – normal processes execute these functions via syscalls.
- Kernel threads only execute in kernel mode as opposed to normal processes that switch between kernel and user modes.
- Check https://elixir.bootlin.com/linux/v5.10.191/source/kernel/fork.c#L2540

OS Development over Time

OS
(based on architecture)

- **Monolithic kernel OS**
  Unix, MS-DOS
- **Micro kernel OS**
  MINIX, Windows NT
- **Hybrid/Modular kernel OS**
  Linux



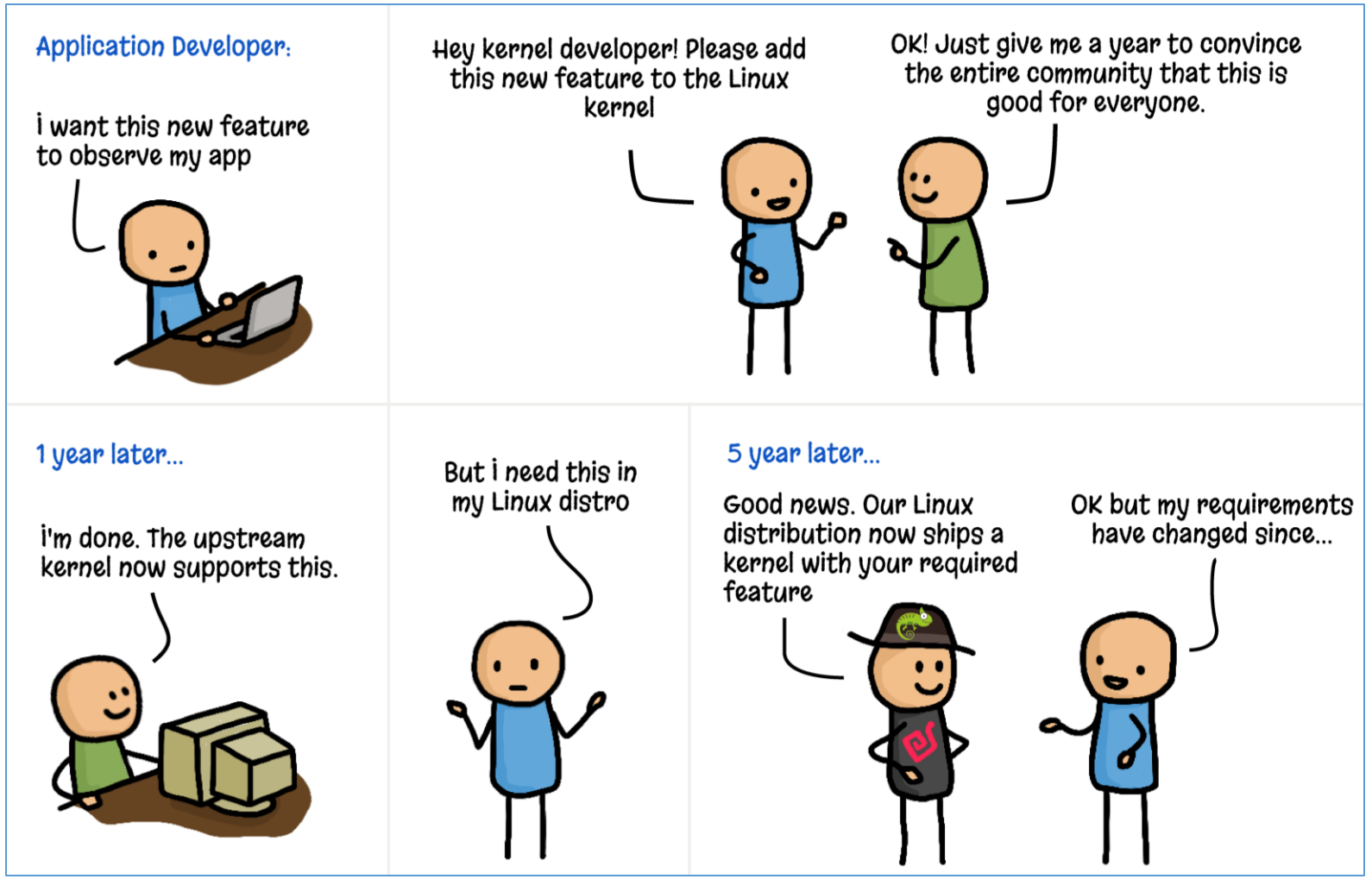
Base Kernel Module vs Loadable Kernel Module

- You often have a choice between putting a module into the kernel **by loading it as a Loadable Kernel Module (LKM)** or **binding it into the base kernel**.
- You can add code to the Linux kernel while it is running
  - Called a loadable kernel module.
  - With LKM, you don't have to rebuild your kernel.
- Sometimes it is important to build modules into the **base kernel** instead of making it an LKM.
  - Anything that is necessary to get the system up must obviously be built into the base kernel.
  - For example, the driver for the disk drive that contains the root filesystem must be built into the base kernel.

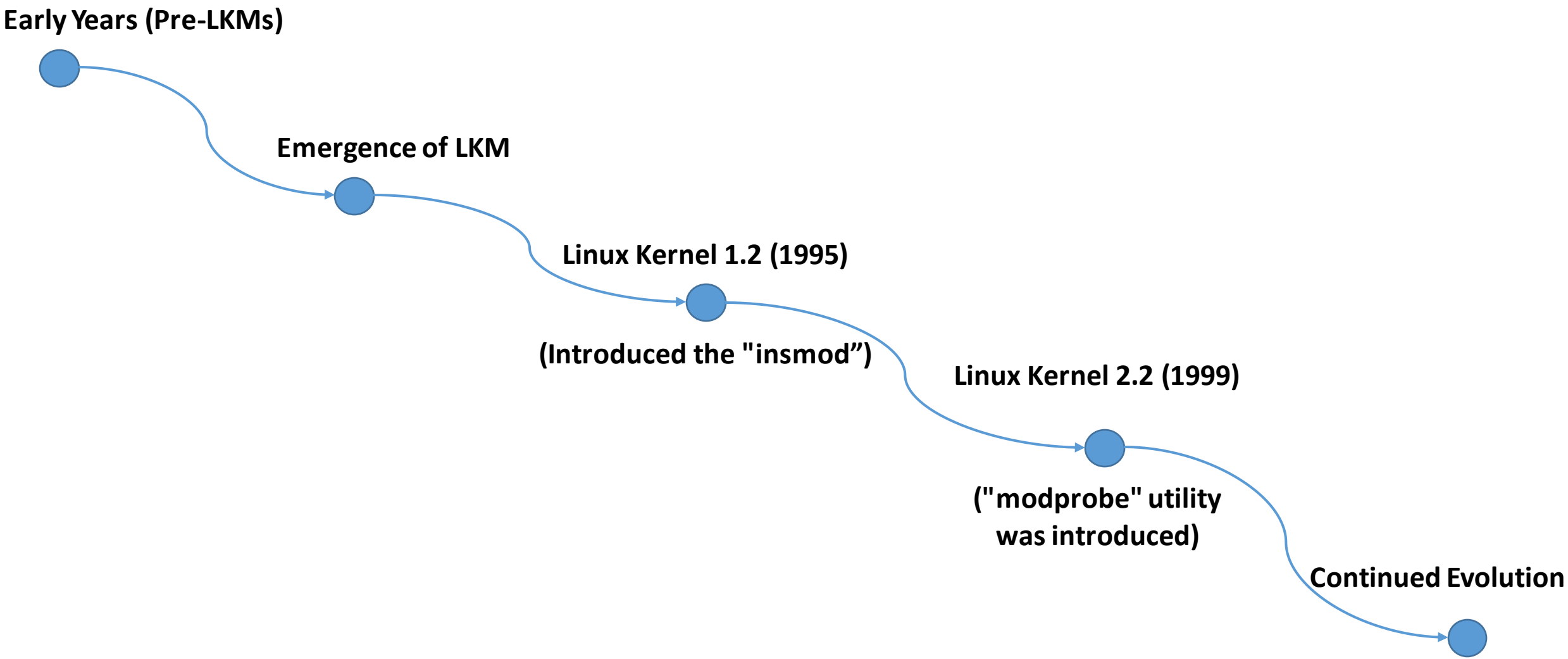
What is LKM?

- A loadable kernel module (LKM) is an object file that contains code to extend the running kernel, or so-called base kernel, of an operating system.
- LKM's are critical to the Linux administrator as they provide them the capability to add functionality to the kernel without having to recompile the kernel.
- Examples: Video and other device drivers can be added to the Linux kernel without shutting down the system, recompiling, and rebooting.

Why LKM?


Application Developer:
i want this new feature to observe my app
Hey kernel developer! Please add this new feature to the Linux kernel
OK! Just give me a year to convince the entire community that this is good for everyone.
1 year later...
i'm done. The upstream kernel now supports this.
But i need this in my Linux distro
5 year later...
Good news. Our Linux distribution now ships a kernel with your required feature
OK but my requirements have changed since...

Evolution of LKM Development?



Use Cases

- These modules can help in different ways --
- **Device drivers** - The kernel uses it to communicate with that piece of hardware without having to know any details of how the hardware works.
- **Filesystem drivers** - A filesystem driver interprets the contents of a filesystem as files and directories and such.
- **System calls** - Most system calls are built into the base kernel. But you can invent a system call of your own and install it as an LKM. Or you can override an existing system call with an LKM of your own.
- **Network drivers** - A network driver interprets a network protocol.

Linux Device Drivers – as a LKM

- Most of the Linux Device drivers are available as LKM. But Why as LKM?
- Device drivers are set of API sub-routines interface to hardware. It mainly abstracts the implementation of hardware-specific details from a user program.
- Another important aspect is that every Device is a special file in all Unix like systems i.e. Typically a user/user-program can access the device via file name in **`/dev`** , e.g. **`/dev/dv1`**.
- We have more than 70% of Linux kernel code specific to device drivers for thousands of devices.
  - Only very few of these devices are needed at any point of time in running instance.
  - As memory is costly and having all of them at once and having drivers of devices, which are useless is just a waste of memory.
- Hence, implementation of device drivers as LKM makes sense.

LKM Utilities

- The following is the list of basic LKM commands (utilities)

Utility	Description
insmod	Insert a LKM into the kernel
rmmod	Remove a LKM from the kernel
depmod	Lists dependencies between LKMs
ksyms	Lists symbols that are exported by the kernel for use by given LKMs
lsmod	Lists currently loaded LKMs
modinfo	Display contents of .modinfo section in an LKM object file
modprobe	Loads given module after loading/unloads modules required for the given module. For example, if you must load A before loading B, 'modprobe' will automatically load A when you tell it to load B

How to Write a LKM?

- Special programs and have structures entirely different from user program.
- They don't have main functions but must have at-least 2 functions **`init_XXXXXX()`** and **`cleanup_XXXXXX()`** — here **`XXXXXX`** represent the module name.

Loadable Kernel Modules

These functions will match the following LKM Utilities, while they are plugged into kernel:

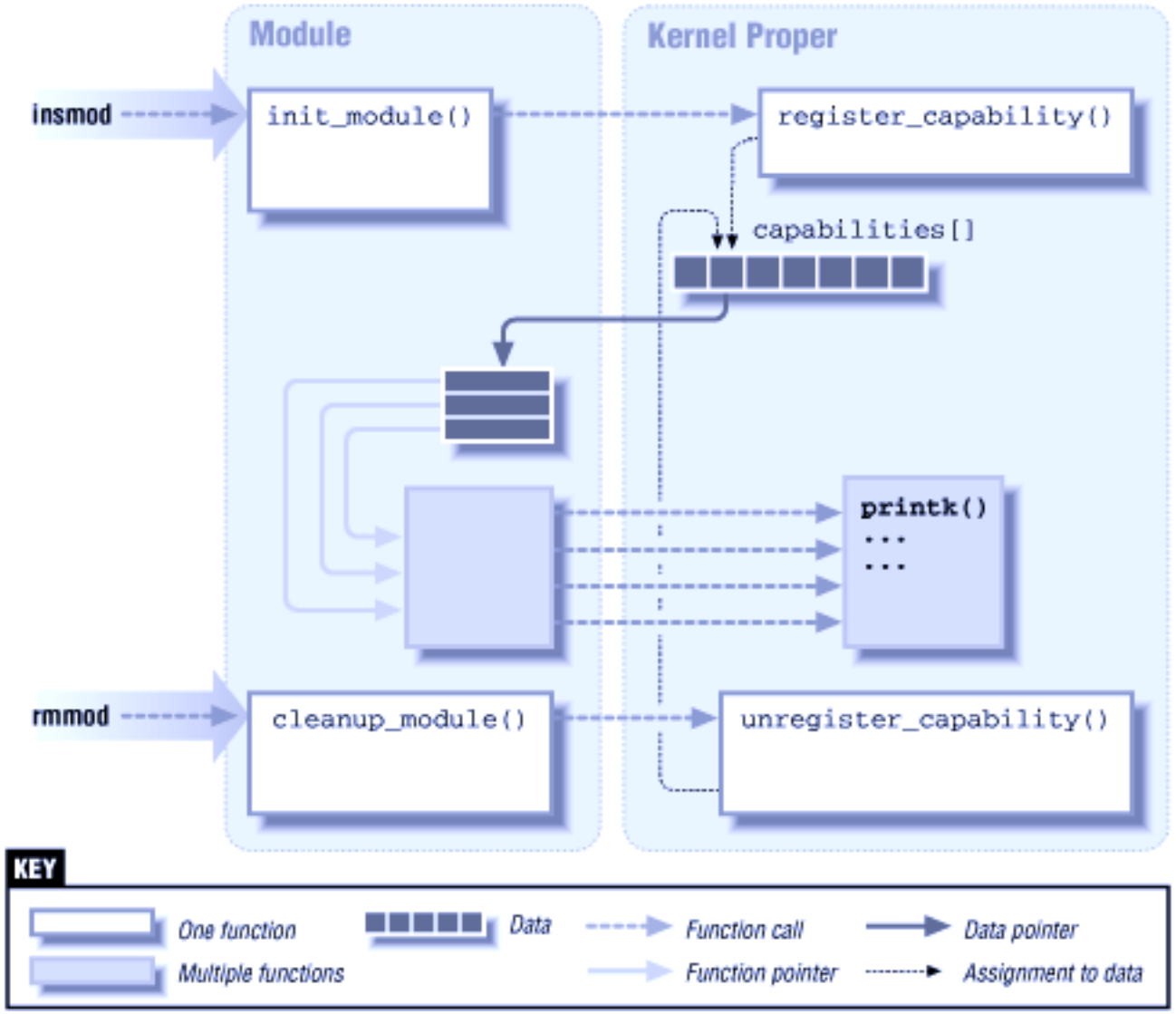
- **insmod**: Insert an LKM into the kernel.
- **rmmod**: Remove an LKM from the kernel.

```
int init_modulename (void)
{
/*initialition code*/
}

void cleanup_modulename (void)
{
/*cleanup code*/
}
```

LKM Architecture

- **init_module():**
  - Executed during loading.
  - It supposed to initialize the module and register capabilities.
  - It informs kernel that it is there and it can do this task.
- **cleanup_module():**
  - Need to free up allocated memory and resources.
  - It informs kernel that it is no longer there.
- **module_init() & module_exit():**
  - Allow any name for initialization and cleanup function.



Cautions

- **No libc** modules. Invoke a function only if it is available in the kernel.
- Use static function to avoid namespace pollution.
- Be careful about kernel space and user space.
- Concurrency in the kernel!!

Building and Running Modules

- **Note:** Vendor kernels can be heavily patched and divergent from the mainline; at times, vendor patches can change the kernel API as seen by device drivers.
  - If you are writing a driver that must work on a particular distribution, you will certainly want to build and test against the relevant kernels

- **Warning:** Faults in kernel code can bring about the demise of a user process or, occasionally, the entire system.
  - They do not normally create more serious problems, such as disk corruption.
  - Nonetheless, it is advisable to do your kernel experimentation on a system that does not contain data that you cannot afford to lose, and that does not perform essential services.

The Hello World Module – hello.c

A special macro (MODULE_LICENSE) is used to tell the kernel that this module bears a free license; without such a declaration, the kernel complains when the module is loaded.

Invoked when the module is loaded into the kernel **(hello_init)**

Invoked when the module is removed **(hello_exit)**

module_init & **module_exit** lines use special kernel macros to indicate the role of these two functions.

```
#include <linux/init.h>
#include <linux/module.h>

MODULE_LICENSE("Dual BSD/GPL");

static int hello_init(void) {
    printk(KERN_ALERT "Hello, world\n");
    return 0;
}

static void hello_exit(void) {
    printk(KERN_ALERT "Goodbye, cruel
world\n");
}

module_init(hello_init);
module_exit(hello_exit);
```

The Hello World Module - hello.c

The **printk** function is defined in the Linux kernel and made available to modules; it behaves similarly to the standard C library function **printf**.

The kernel needs its own printing function because it runs by itself, without the help of the C library.
The module can call **printk** because, after **insmod** has loaded it, the module is linked to the kernel and can access the kernel's public symbols (functions and variables).

```
#include <linux/init.h>
#include <linux/module.h>

MODULE_LICENSE("Dual BSD/GPL");

static int hello_init(void) {
    printk(KERN_ALERT "Hello, world\n");
    return 0;
}

static void hello_exit(void) {
    printk(KERN_ALERT "Goodbye, cruel
world\n");
}

module_init(hello_init);
module_exit(hello_exit);
```

The Hello World Module – hello.c

The string KERN_ALERT is the priority of the message

```
#include <linux/init.h>
#include <linux/module.h>

MODULE_LICENSE("Dual BSD/GPL");

static int hello_init(void) {
    printk(KERN_ALERT "Hello, world\n");
    return 0;
}

static void hello_exit(void) {
    printk(KERN_ALERT "Goodbye, cruel
world\n");
}

module_init(hello_init);
module_exit(hello_exit);
```

Compiling Modules - Makefile

```
obj-m+=hello.o
all:
    make -C /lib/modules/$(shell uname -r)/build/ M=$(shell pwd) modules
clean:
    make -C /lib/modules/$(shell uname -r)/build/ M=$(shell pwd) clean
```

The assignment above (which takes advantage of the extended syntax provided by GNU *make*) states that there is one module to be built from the object file *hello.o*.
The resulting module is named *hello.ko* after being built from the object file.

-C : Changing to the kernel source directory
M: Location of external module sources informs kernel an external module is being built
`uname -r`: version of currently running kernel

Testing the Module – Loading and Unloading

```
% make
make[1]: Entering directory `/usr/src/linux-x.y.z'
  CC [M] /home/ldd3/src/misc-modules/hello.o
  Building modules, stage 2.
  MODPOST
  CC /home/ldd3/src/misc-modules/hello.mod.o
  LD [M] /home/ldd3/src/misc-modules/hello.ko
make[1]: Leaving directory `/usr/src/linux-x.y.z'

% su

root# insmod ./hello.ko
root# dmesg
Hello, world

root# rmmod hello
root# dmesg
Goodbye cruel world
```

Note: Only the superuser can load and unload a module.

Configuration – for Debian UNIX

1. Update current packages of the system to the latest version.

```
$ sudo apt update && sudo apt upgrade -y
```

2. Download and install the essential packages to compile kernels

```
$ sudo apt install build-essential libncurses-dev libssl-dev libelf-dev
bison flex -y

$ sudo apt install linux-headers-$(uname -r)
```

3. Compile LKM

```
$ make
```

Sending Data to LKM

```
// Include the header
    #include <linux/proc_fs.h>

// Declare global file operation variable
    static struct proc_ops file_ops;

// Create a file in /proc file during initialization (don't forget to check for error)
    struct proc_dir_entry *entry = proc_create("hello", 0, NULL, &file_ops);
    if(!entry)
        return -ENOENT;

// Set write function pointer
   file_ops.proc_write = write;

// Define the write function
    static ssize_t write(struct file *file, const char *buf, size_t count, loff_t *pos) {
        printk("%.*s", count, buf); return count;
    }
// Remove the file in /proc during exit
        remove_proc_entry("hello", NULL);
```

Getting Data to LKM

```
// Add required header file
     #include <linux/uaccess.h>

// Add global variable to store incoming data
     static char buffer[256] = {0};
     static int buffer_len = 0;

// Update write function to store incoming data
          if(!buf || !count)
                    return -EINVAL;
          if(copy_from_user(buffer, buf, count < 256 ? count:256))
                    return -ENOBUFS;
          buffer_len = count < 256 ? count:256;
          printk(KERN_INFO "%.*s", (int)count, buf);     return buffer_len;

// Add read function to send data to user program
     static ssize_t read(struct file *file, char *buf, size_t count, loff_t *pos) {
          int ret = buffer_len;
          if(!buffer_len)
                    return 0;
          if(!buf || !count)
                    return -EINVAL;
          if(copy_to_user(buf, buffer, buffer_len))
                    return -ENOBUFS;
          printk(KERN_INFO "%.*s", (int)buffer_len, buffer);
          buffer_len = 0;
                    return ret;
     }

// Add read function pointer to file_ops
     file_ops.proc_read = read;
```

Further Reading

1. https://www.iitg.ac.in/asahu/cs421/books/LKM2.6.pdf
2. https://www.xml.com/ldd/chapter/book/
3. https://www.oreilly.com/openbook/linuxdrive3/book/
4. https://www.tldp.org/HOWTO/Module-HOWTO/index.html
5. https://lasr.cs.ucla.edu/classes/111_fall16/readings/dynamic_modules.html
6. http://derekmolloy.ie/writing-a-linux-kernel-module-part-1-introduction/

Command line argument passing to a module

```
#include <linux/init.h>
#include <linux/kernel.h> /* for ARRAY_SIZE() */
#include <linux/module.h>
#include <linux/moduleparam.h>
#include <linux/printk.h>
#include <linux/stat.h>

MODULE_LICENSE("GPL");
static short int myshort = 1;
static int myint = 420;
static long int mylong = 9999;
static char *mystring = ”hello";
static int myintarray[2] = { 420, 420 };
static int arr_argc = 0;
```

```
odule_param(myshort, short, S_IRUSR | S_IWUSR | S_IRGRP | S_IWGRP);
MODULE_PARM_DESC(myshort, "A short integer");
module_param(myint, int, S_IRUSR | S_IWUSR | S_IRGRP | S_IROTH);
MODULE_PARM_DESC(myint, "An integer");
module_param(mylong, long, S_IRUSR);
MODULE_PARM_DESC(mylong, "A long integer");
module_param(mystring, charp, 0000);
MODULE_PARM_DESC(mystring, "A character string");
module_param_array(myintarray, int, &arr_argc, 0000);
MODULE_PARM_DESC(myintarray, "An array of integers");
```

Command line argument passing to a module

```
static int __init senddata_init(void) {
    int i;
    pr_info("Sending Data\n============\n");
    pr_info("myshort is a short integer: %hd\n", myshort);
    pr_info("myint is an integer: %d\n", myint);
    pr_info("mylong is a long integer: %ld\n", mylong);
    pr_info("mystring is a string: %s\n", mystring);
    for (i = 0; i < ARRAY_SIZE(myintarray); i++)
        pr_info("myintarray[%d] = %d\n", i, myintarray[i]);
    pr_info("got %d arguments for myintarray.\n", arr_argc);
    return 0;
}
static void __exit senddata_exit(void) {
    pr_info("Removed senddata module!\n");
}

module_init(senddata_init);
module_exit(senddata_exit);
```

```
# insmod senddata.ko
```

```
[503279.562297] Sending Data
                ============
[503279.562307] myshort is a short integer: 1
[503279.562311] myint is an integer: 420
[503279.562316] mylong is a long integer: 9999
[503279.562319] mystring is a string: hello
[503279.562322] myintarray[0] = 420
[503279.562326] myintarray[1] = 420
[503279.562329] got 0 arguments for myintarray.
```

```
# insmod senddata.ko
mystring="world" myintarray=-1
```

```
[503643.772842] Sending Data
                ============
[503643.772850] myshort is a short integer: 1
[503643.772854] myint is an integer: 420
[503643.772858] mylong is a long integer: 9999
[503643.772861] mystring is a string: world
[503643.772865] myintarray[0] = -1
[503643.772868] myintarray[1] = 420
[503643.772871] got 1 arguments for myintarray.
```

```
# rmmod ./senddata.ko
```

```
[503721.112612] Removed senddata module!
```

Happy Learning