Linux Process Architecture

Acknowledgement

Many of the slides are borrowed from the same course offered by Prof. Sandip Chakraborty in earlier years (with some changes done in some cases)

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

The task_struct structure

- The kernel maintains info about each process in a process descriptor, of type *task_struct*
  - https://elixir.bootlin.com/linux/v5.10.188/source/include/linux/sched.h#L644
  - Total around 780 lines ☺
    - Though quite a bit of it is comments, also a lot of ifdef's
    - Still, a large no. of fields
- *task_struct* structures are allocated from a memory cache (why?)
  - https://elixir.bootlin.com/linux/v5.10.188/source/kernel/fork.c#L168

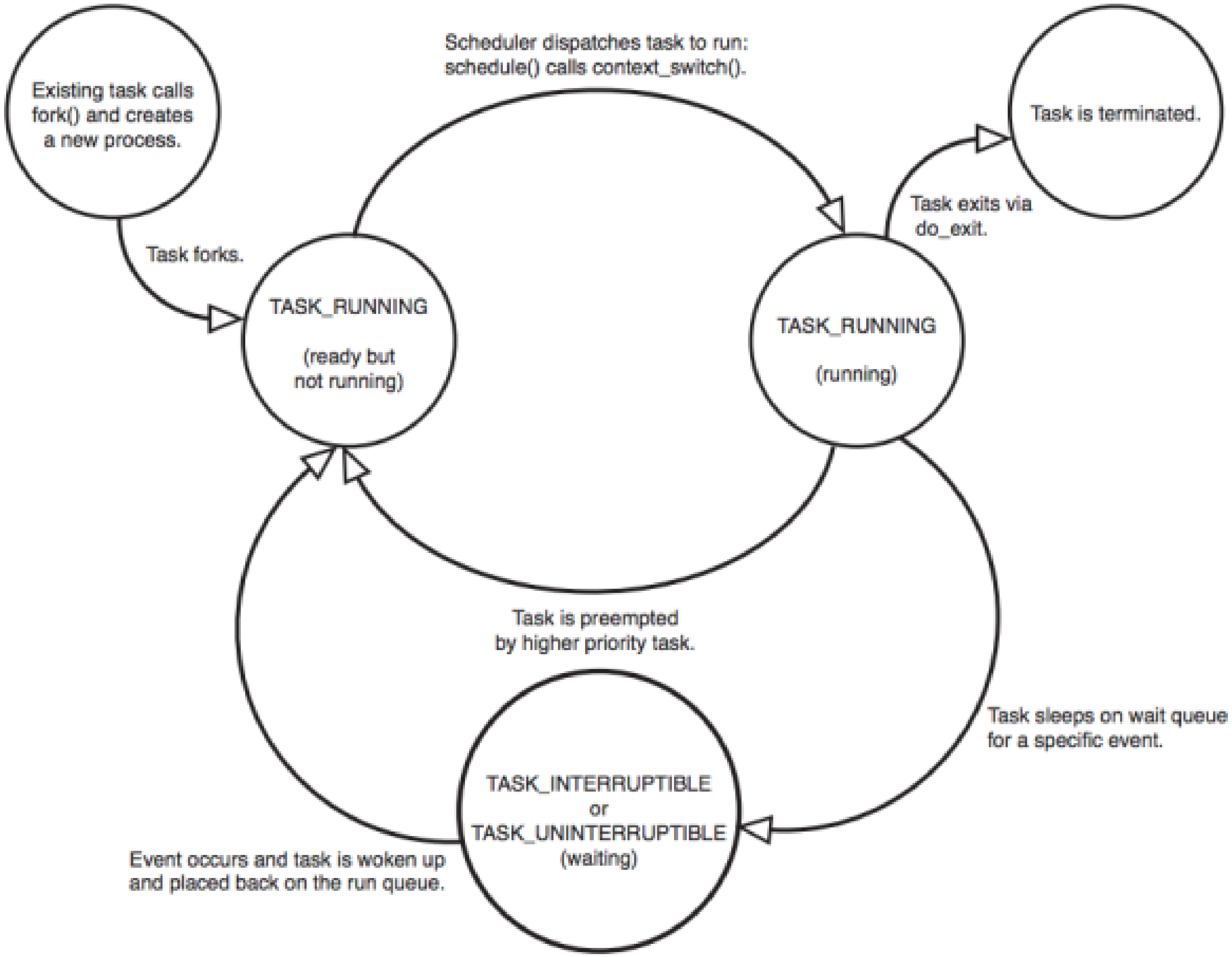
Some example fields in task_struct

```
volatile long                   state;
void                            *stack;
int                             on_cpu;
int                             on_rq;

int                             prio;
int                             static_prio;
int                             normal_prio;
unsigned int                    rt_priority;
const struct sched_class        *sched_class;
struct sched_entity             se;
struct sched_rt_entity          rt;
struct sched_dl_entity          dl;
struct sched_info               sched_info;
```

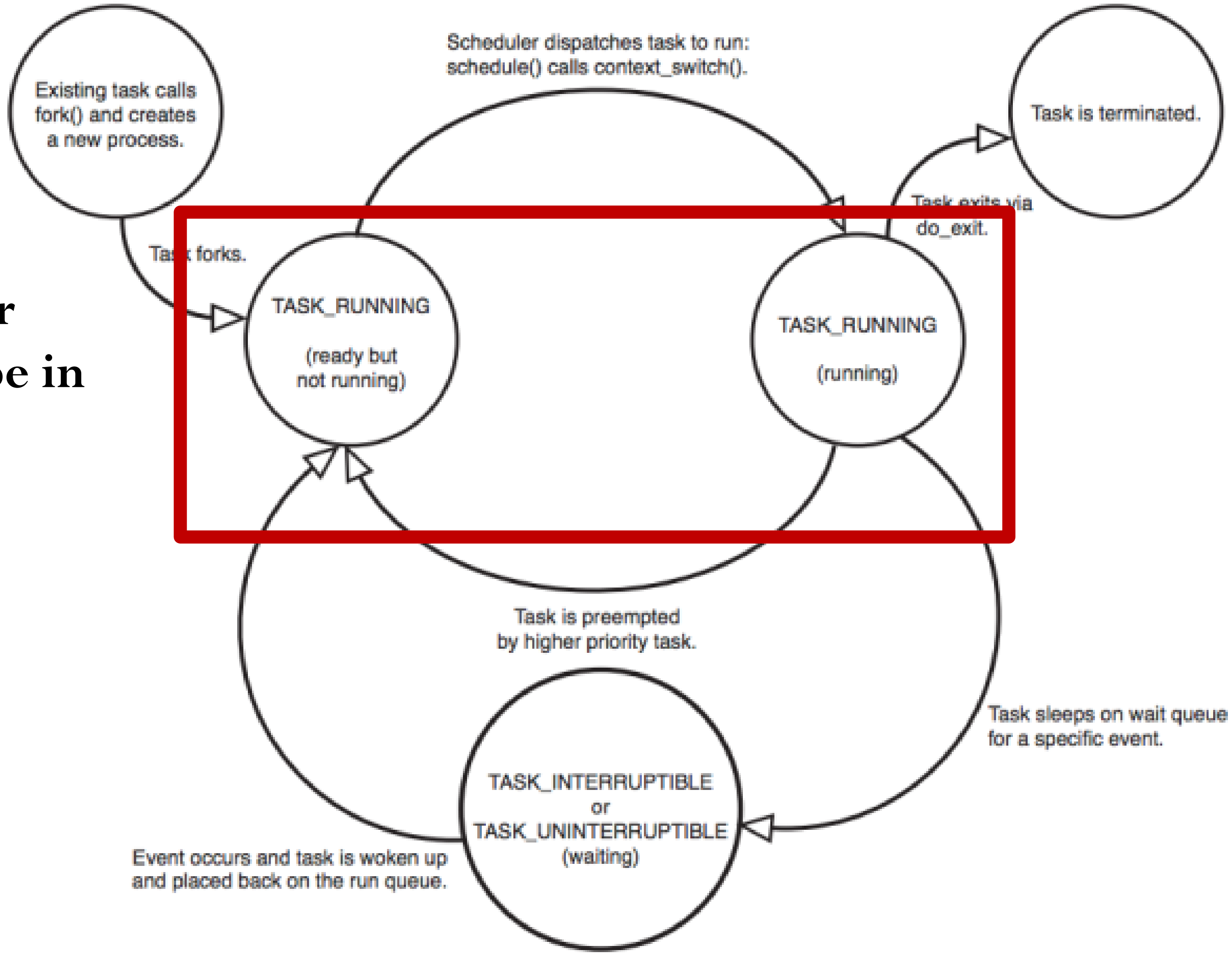
```
struct list_head                tasks;

struct mm_struct                *mm;

int                             exit_code;

pid_t                           pid;
struct task_struct __rcu        *real_parent;
struct task_struct __rcu        *parent;
struct list_head                children;
struct list_head                sibling;
struct task_struct              *group_leader;
struct pid                      *thread_pid;

struct files_struct             *files;

struct signal_struct            *signal;
struct sighand_struct __rcu     *sighand;
sigset_t                        blocked;
```

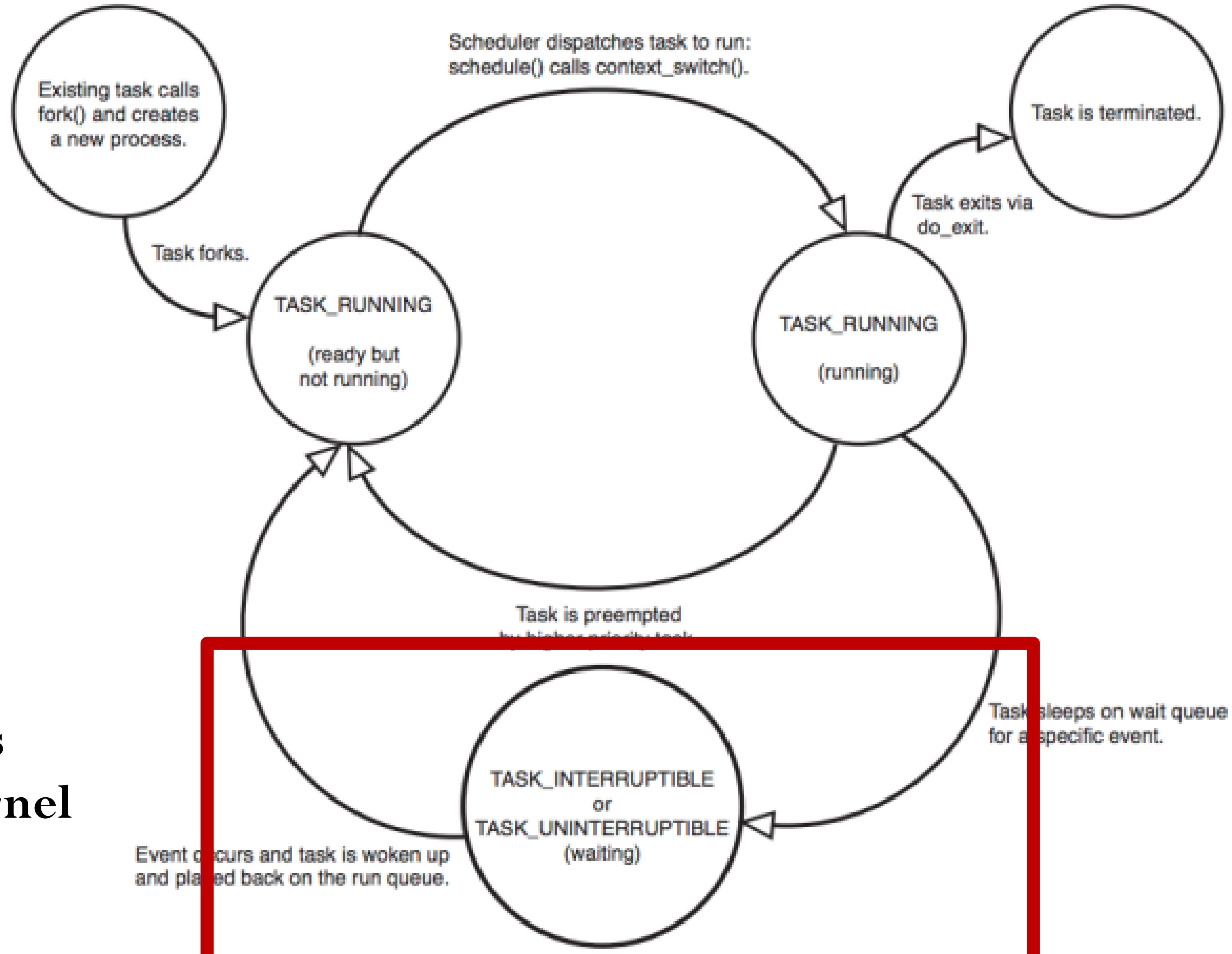
Process States

- Consists of an array of mutually exclusive flags
  - https://elixir.bootlin.com/linux/v5.10.188/source/include/linux/sched.h#L80
- Example 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 up process)
  - *TASK_UNINTERRUPTIBLE* (Sleeping process cannot be woken by a signal)
  - *TASK_NOLOAD* (uninterruptible tasks that do not contribute to load average)
  - *TASK_STOPPED* (task execution has stopped).
  - *EXIT_ZOMBIE* (process has completed execution but still in the process table, reaped out by the parent later on).

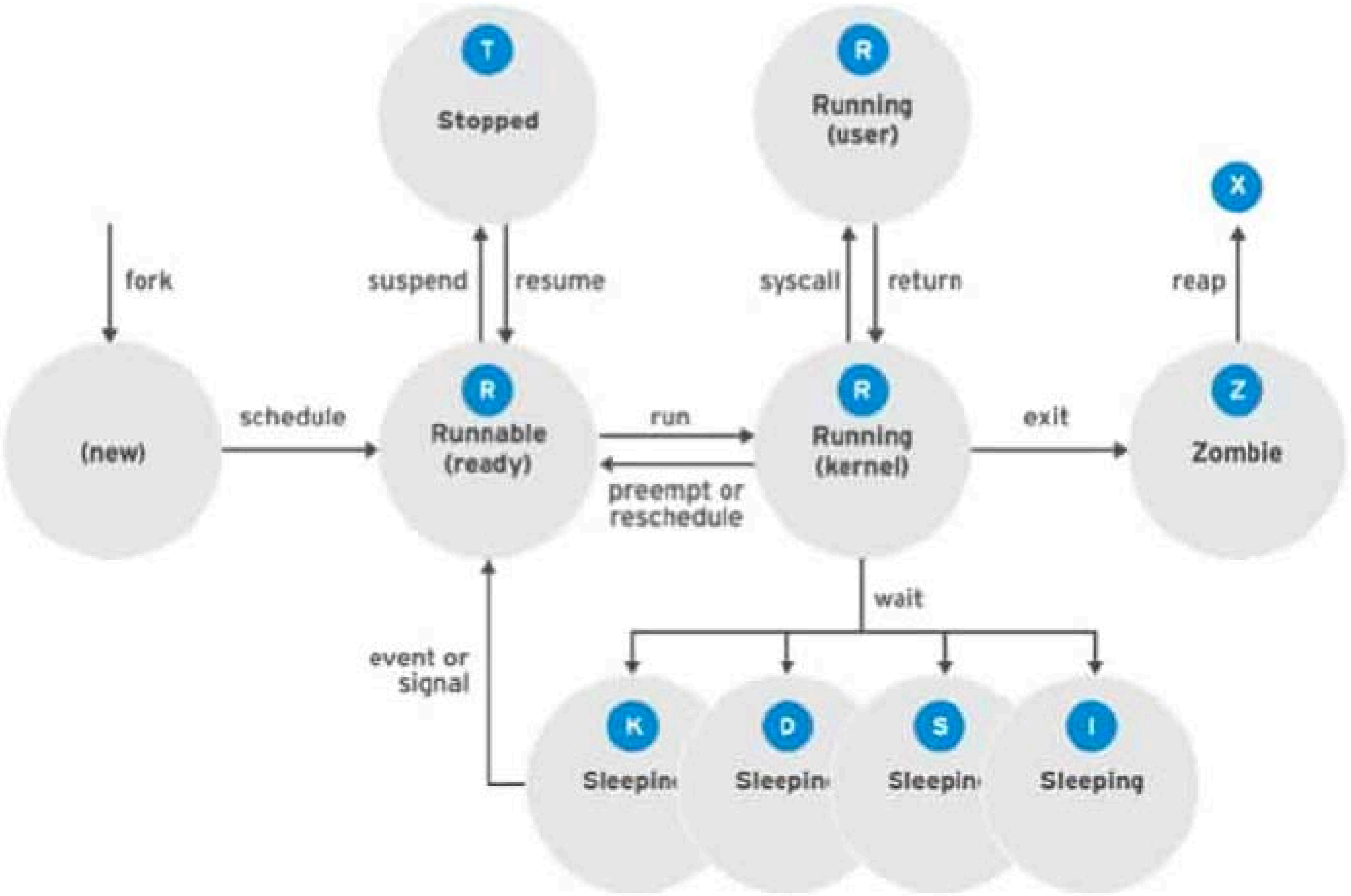

Existing task calls fork() and creates a new process.
Task forks.
Scheduler dispatches task to run: schedule() calls context_switch().
Task is terminated.
Task exits via do_exit.
TASK_RUNNING (ready but not running)
TASK_RUNNING (running)
Task is preempted by higher priority task.
Task sleeps on wait queue for a specific event.
TASK_INTERRUPTIBLE or TASK_UNINTERRUPTIBLE (waiting)
Event occurs and task is woken up and placed back on the run queue.

A process in user mode can only be in these states





The user process moves to the 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------

Check process state (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 state (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 Identification

- Each process is identified with
  - Process ids (or PIDs)
    - Default 0..32767 for compatibility with traditional UNIX systems
    - Can be set to higher value through */proc/sys/kernel/pid_max*
  - Process descriptor
    - Stored in a variable of type *struct task_struct*
    - Processes are dynamic, so descriptors are kept in dynamic memory
    - A ~10KB memory area is allocated for each process, to hold process descriptor and kernel mode process stack

The struct pid structure

- Each *task_struct* structure has a reference to a *struct pid*
  - https://elixir.bootlin.com/linux/v5.10.188/source/include/linux/pid.h#L59
- Has a list of all tasks with the same pid value, provides a mapping from pid to process descriptor

```
struct pid
{
        refcount_t count;
        unsigned int level;
        spinlock_t lock;
        …
        /* lists of tasks that use this pid */
        struct hlist_head tasks[PIDTYPE_MAX];
        …
        struct upid numbers[1];
};
```

Finding the task_struct for a pid

- *struct pid* structures are stored in a hash table
  - One per *pid*
  - Exists as long as at least one process is attached to it
    - Tracked by the *count* field
- *pid* value is used to find the *struct pid* structure first from the hash table
- The *task_struct* of the process is then found from the *struct pid* found
  - The first entry in the *task* array
- See */kernel/pid.c* for functions that manipulate and convert between *pid*, *struct task_struct*, and *struct pid*
- But why a separate *struct pid*? Why not directly map from *pid* to *struct task?*

- A *struct pid* is the kernel's internal notion of a process identifier
  - It refers to individual tasks, process groups, and sessions
- Solves two problems
  - What if a process holding a pid exits, pid value wraps around, and allocated to a new process?
  - Referring to user-space processes by holding a reference to *struct task_struct* is costly if the process exits
    - Around 10K of kernel memory. By comparison a *struct pid* is about 64 bytes

Allocating pid

- The fork()/vfork() syscall allocates a PID through *alloc_pid()*
  - Calls *kernel_clone()* which calls *copy_process()*, which then calls *alloc_pid()*
  - https://elixir.bootlin.com/linux/v5.10.188/source/kernel/fork.c#L2134
  - The *alloc_pid* calls *kmem_cache_alloc*
    - Allocates all the structures from a cache (fast allocation as PIDs are frequently allocated, avoid kernel memory allocation when one process exits and another starts)
  - Then iterates 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
    - https://elixir.bootlin.com/linux/v5.10.188/source/kernel/pid.c#L211

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

Process List

- The *process list* (of all processes in system) is a doubly-linked list
  - Through the *struct list_head tasks* field in the process descriptor
  - *prev* & *next* fields under the *list_head* structure are used to build list
  - *list_for_each()* macro scans whole list
    - See */include/linux/types.h*, and */include/linux/list.h* for list related functions
  - But this gives us pointers to one particular field in the middle of the *task_struct* of another process. Ow do we get the actual *task_struct*?
    - Look at the *container_of* macro in */include/linux/kernel.h*

Creating a Process

- 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/v5.10.188/source/include/uapi/linux/sched.h#L11

- *kernel_clone()* is the main fork routine to create a child process
  - https://elixir.bootlin.com/linux/v5.10.188/source/kernel/fork.c#L2462
  - It takes a set of arguments defined through the structure *kernel_clone_args*
    - https://elixir.bootlin.com/linux/v5.10.188/source/include/linux/sched/task.h#L21
  - Checks the flags and call *copy_process()*
    - Copies the *task_struct* by calling *dup_task_struct()*
    - Copies the content of the parent memory to the child memory
    - Assign a *pid* by calling *alloc_pid()*
      - Allocates both *struct pid* and the numeric pid
    - Sets task state to TASK_NEW
    - Calls *get_pid()* to increment refcount
    - Calls *attach_pid()* to attach task to its struct pid structure
  - Put the task in the runqueue by calling *wake_up_new_task()*
    - https://elixir.bootlin.com/linux/v5.10.188/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 calls a subroutine *wait_for_vfork_done()* to check the child status and wait till child exits
    - https://elixir.bootlin.com/linux/v5.10.191/source/kernel/fork.c#L1267

Exiting a Process

- *do_exit()* is the main entry point
  - https://elixir.bootlin.com/linux/v5.10.188/source/kernel/exit.c#L760
  - Calls *exit_signal()* to set task flag PF_EXITING so that signals are not delivered to it anymore
  - Sets task's exit code
  - Cleans up shared resources, memory, file information,….
    - Bunch of *exit_** functions
  - Calls *exit_notify()*
    - Calls *release_task()* which (directly or through other functions) unhashes the process from the struct pid hash, detaches the process from its struct pid, calls *put_pid()* to decrement refcount (and remove struct pid if refcount is 0), releases numeric pid etc.
  - Calls *do_task_dead()* to take it out of scheduling process and set state to TASK_DEAD.

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.
- https://elixir.bootlin.com/linux/v5.10.188/source/kernel/fork.c#L2545

PID Namespace

- Linux uses namespaces to isolate processes in their own environment
- Allows multiple processes to run on a single machine with no chance of interfering with each other
  - Without using VMs
- Essential for implementing containers
- Uses Linux *namespaces*
- Containers are extremely helpful for running multiple non-interfering services on a server, testing potentially malicious code etc.

- Typically, you see a single process tree
  - Parent-child relationships define the hierarchy
- This implies a process can trace another process or even kill it subject to having permissions
  - Sufficient privileges are needed but possible for a process to interfere with another
- Linux namespaces
  - Possible to have multiple nested process trees
  - Each process tree is an isolated set of processes
    - Has no knowledge of processes in other sibling or parent process trees

- Processes in parent namespace have a full view of the processes in the child namespaces, but not vice-versa
- Possible to nest namespaces
- The root of each new namespace has process id 1 in that namespace
    - It is the init process of the namespace
- The pid of a process is now a list, one for each namespace from its current namespace to the root
- Kept track of by a *struct upid* field in the *struct pid* field (remember it?)

- https://elixir.bootlin.com/linux/v5.10.188/source/include/linux/pid.h#L54

```
struct upid {
    int nr;
    struct pid_namespace *ns;
};
```

- Recall that the id of a process in a namespace is now a list
- The *struct upid numbers[1]* field in *task_struct* keeps track of the list
  - Each *upid* in the array keeps the namespace identifier and the actual pid in that namespace for the process

- How to create a new namespace?
  - Call clone() with the CLONE_NEWPID flag set
  - Must be done at creation time only
- New namespaces are not restricted to PIDs only
  - Network namespace
    - Allows each namespace to see a different set of network interfaces, including loopback
  - Mount namespace
    - Allows each namespace to change mount points without affecting othee namespaces
  - User namespace
    - Ex. Can allow a process to have root privileges within a namespace without giving it any access outside thenamespace
  - IPC namespace
  - UTS namespace

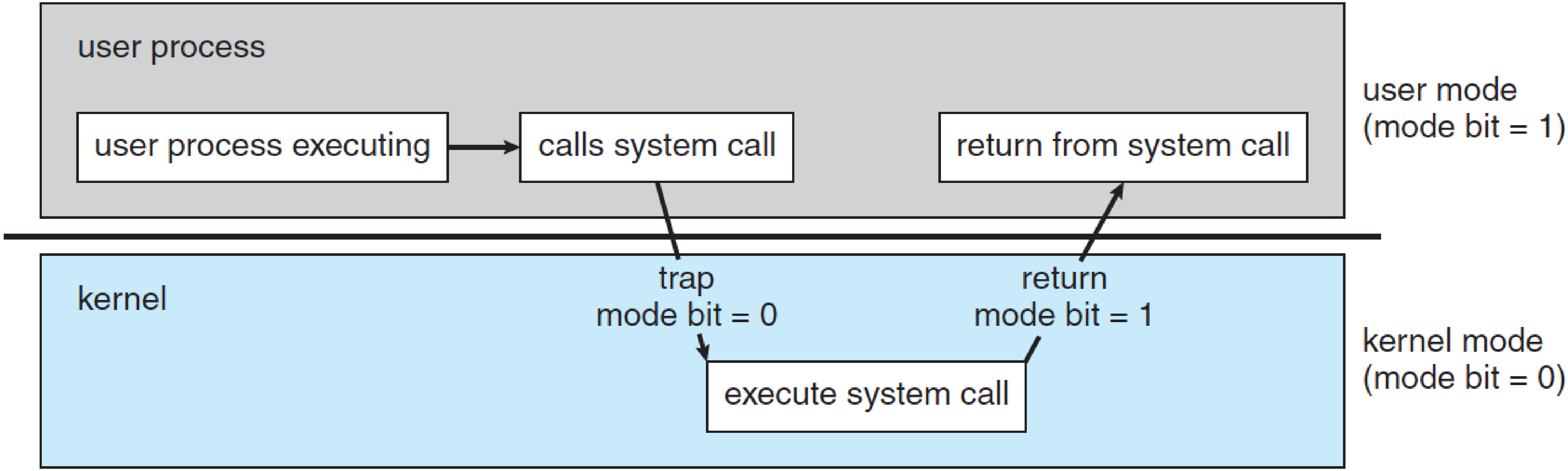
- https://elixir.bootlin.com/linux/v5.10.188/source/include/linux/nsproxy.h#L31

```
struct nsproxy {
        atomic_t count;
        struct uts_namespace        *uts_ns;
        struct ipc_namespace        *ipc_ns;
        struct mnt_namespace        *mnt_ns;
        struct pid_namespace        *pid_ns_for_children;
        struct net                  *net_ns;
        …
};
```

Linux System Calls

- The *only* way to enter the operating kernel is to generate a processor interrupt. These interrupts come from several sources:
  - I/O devices
  - Clocks and timers
  - Exceptions
  - 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.

System Call Path



System Call Stub Functions

- The system call stub functions (*wrappers*) 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
- 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

- Switch the CPU to supervisor mode
- Look up a branch address in a kernel space table
- Branch to a trusted OS function

A user space program 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

- User-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 privilege level

- 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()*.

- 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
```

- 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.

- 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"
```

- 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
    - Example: *write()* syscall
      - Check the corresponding example macro here -- https://codebrowser.dev/glibc/glibc/sysdeps/unix/sysv/linux/write.c.html
      - *SYSCALL_CANCEL* implements 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