Linux Process Architecture

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

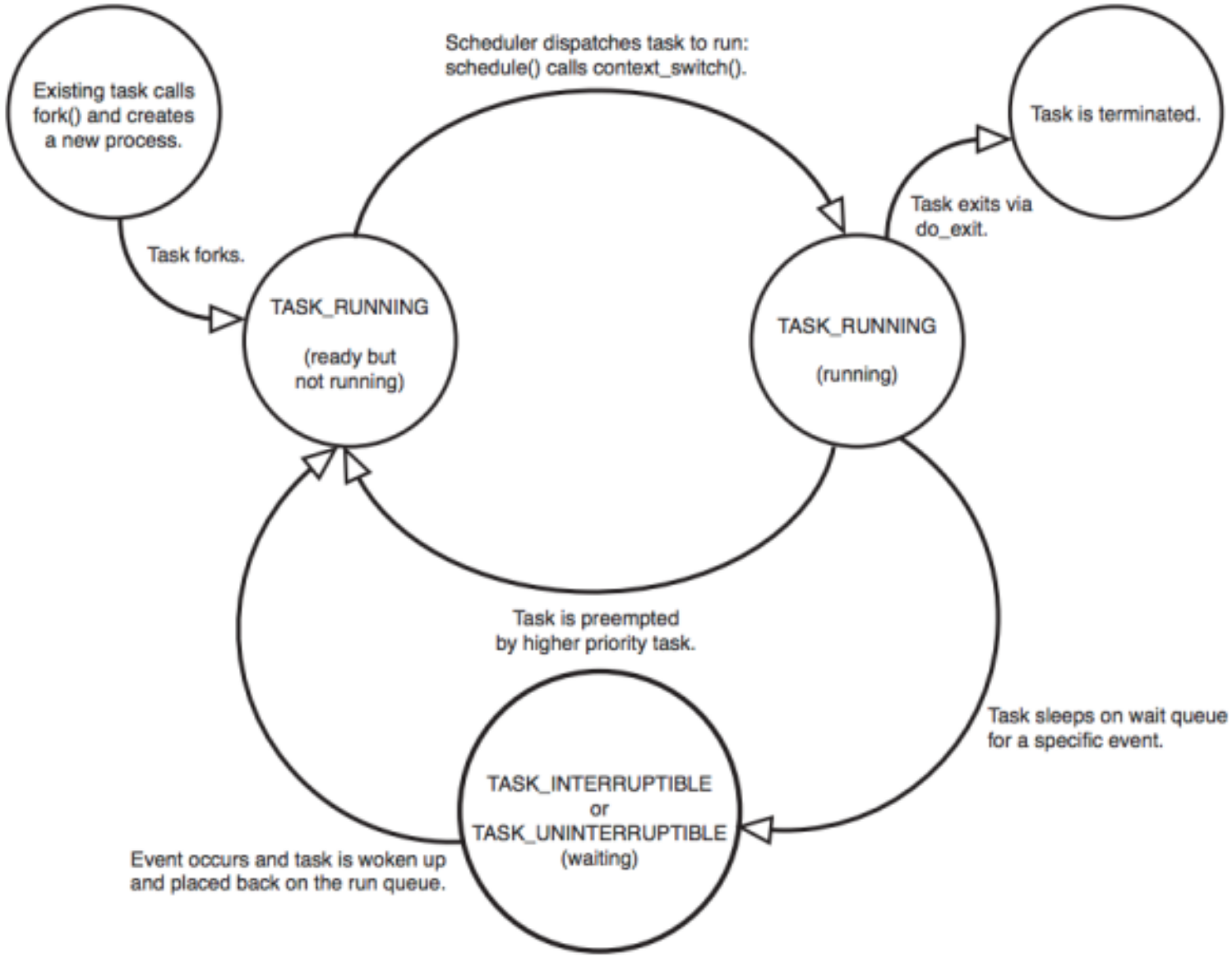
Some example fields in task_struct

```
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;
```

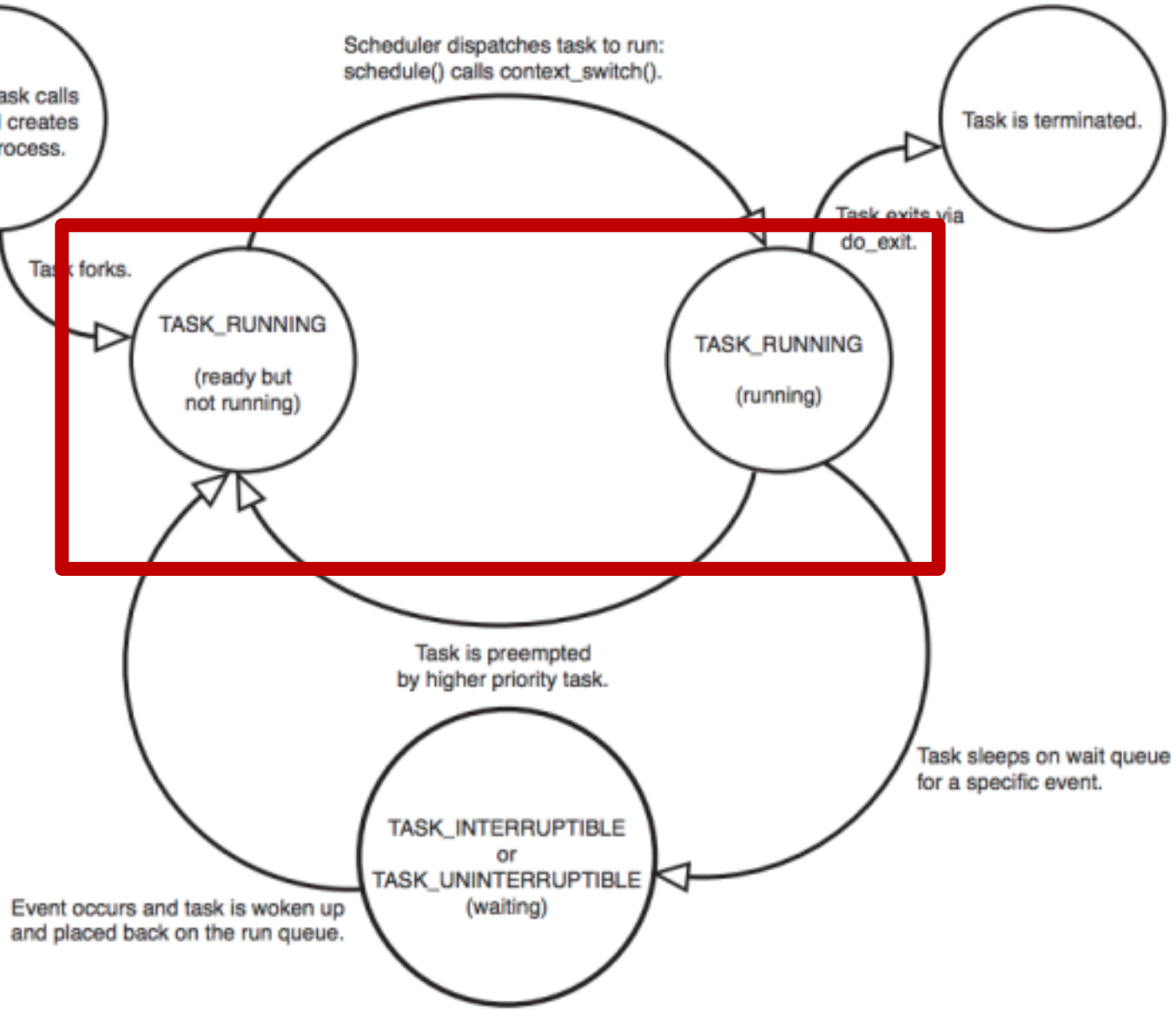
```
struct list_head                tasks;

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

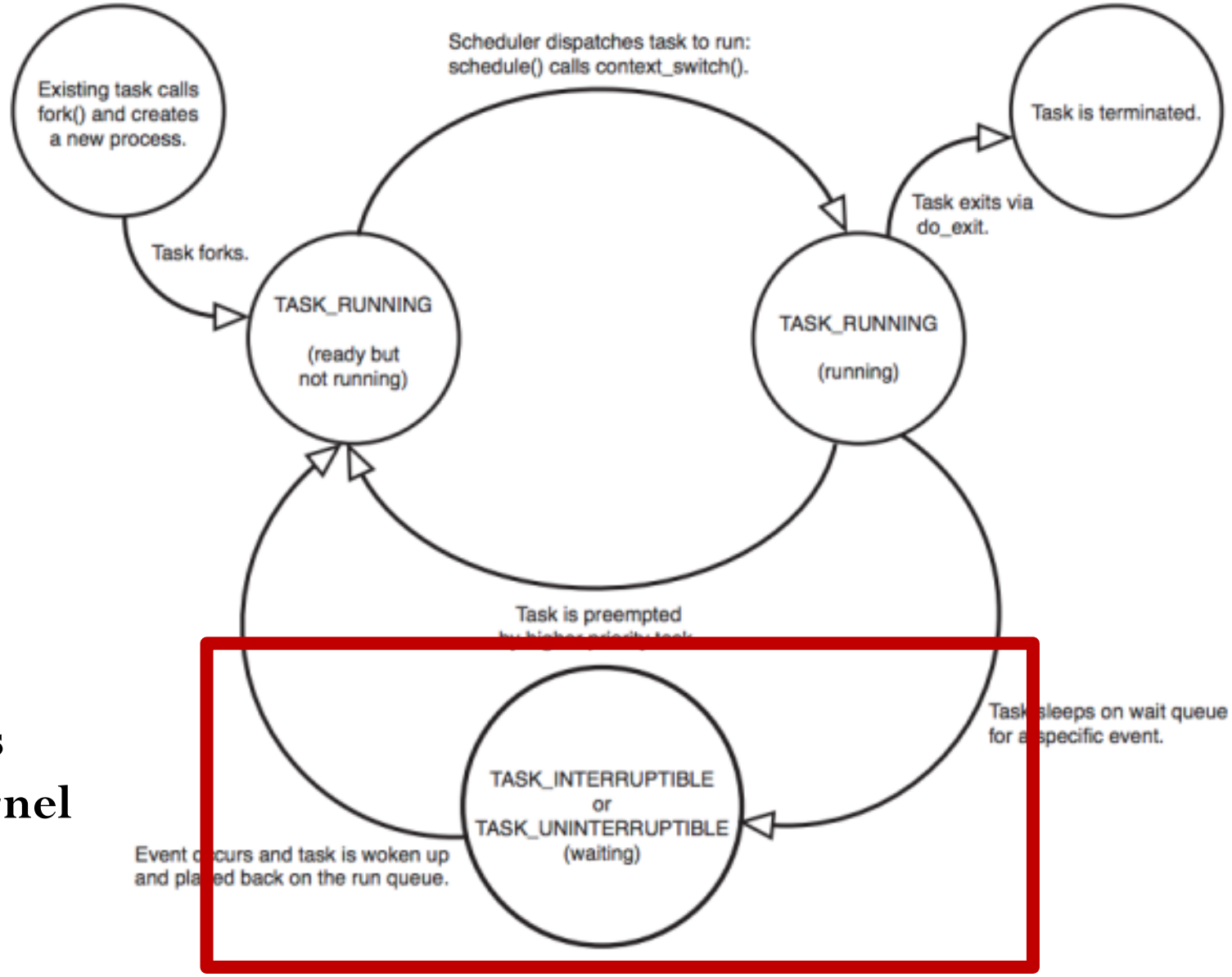
- Consists of an array of mutually exclusive flags
  - https://elixir.bootlin.com/linux/v5.10.188/source/include/linux/sched.h#L80
- Example values:
  - *TASK_NEW* (new task)
  - *TASK_RUNNING* (executing on CPU or runnable)
  - *TASK_INTERRUPTIBLE* (waiting on a condition: interrupts, signals and releasing 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).

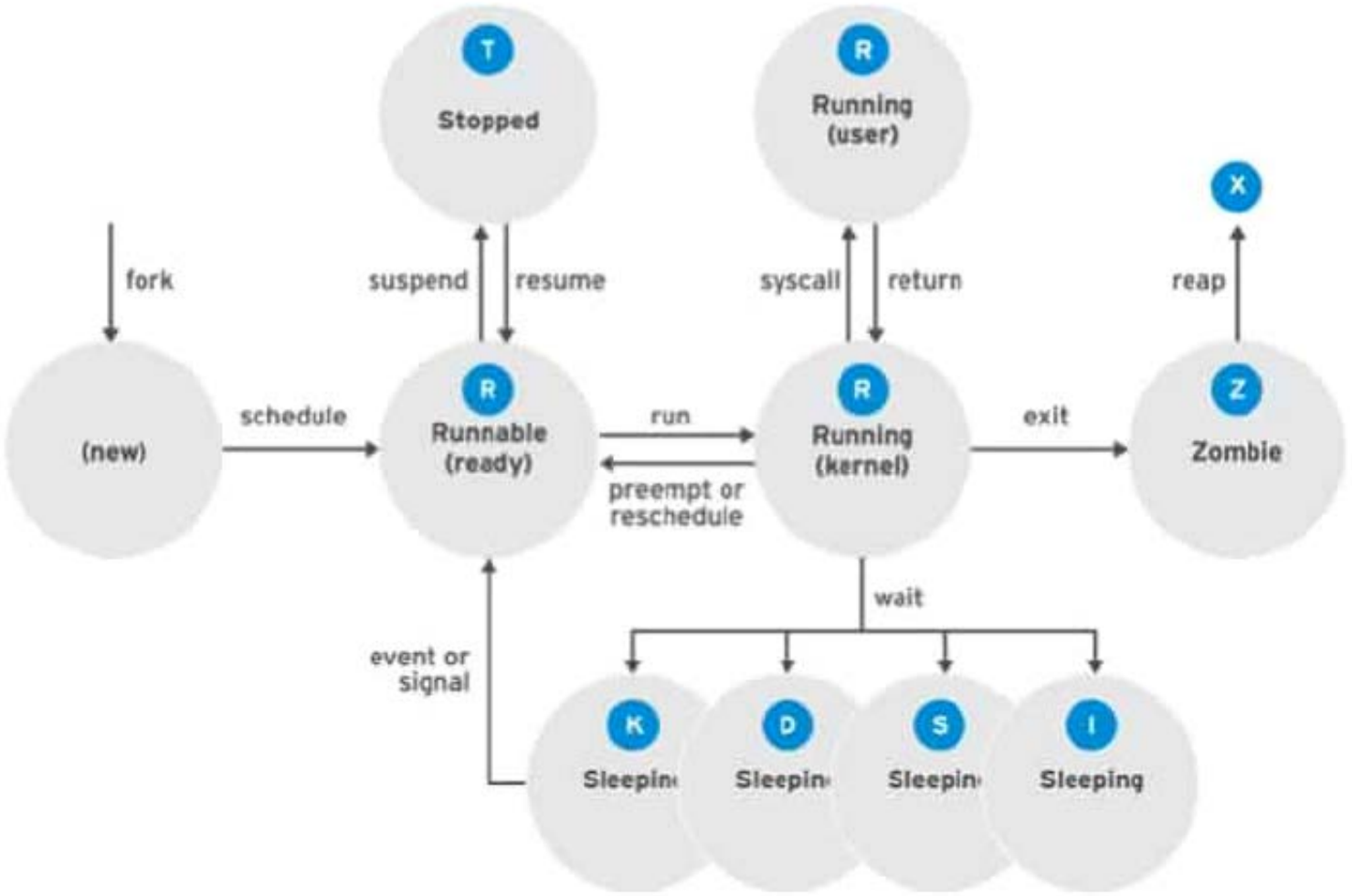

Existing task calls fork() and creates a new process.
Scheduler dispatches task to run: schedule() calls context_switch().
Task is terminated.
Task exits via do_exit.
Task forks.
TASK_RUNNING (ready but not running)
TASK_RUNNING (running)
Task is preempted by higher priority task.
Task sleeps on wait 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      239494  0.0  0.0      0     0 ?        S    11:30   0:00 [kworker/40:2]
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*?