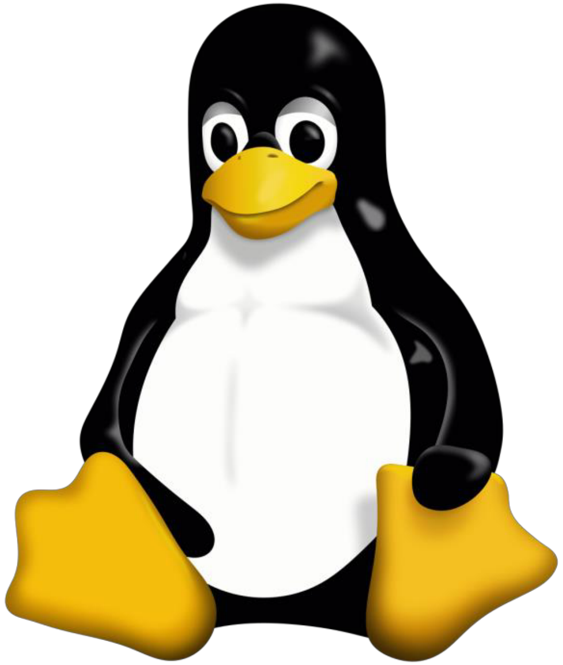


Linux Character Device Driver

Sunbeam Infotech

Register character device

- Each char device is represented by struct cdev.
- *cdev_map* is global array (hash table) to keep track of all char devices.
- cdev_map is object of struct kobj_map.
  - key = device major number
  - hash function = major % 255
  - value = struct probe
    - void *data = struct cdev
- (2) Add device into char device database *cdev_map*.
  - cdev_init(&cdev, &fops);
  - cdev_add(&cdev, devno, dev_count);

    dev no

    1
- Added device can be removed.
  - cdev_del(&cdev);

file_operations

owner
open *
release *
read *
write *
...

pchar_open()
{...}

pchar_close()
{...}

pchar_read()
{...}

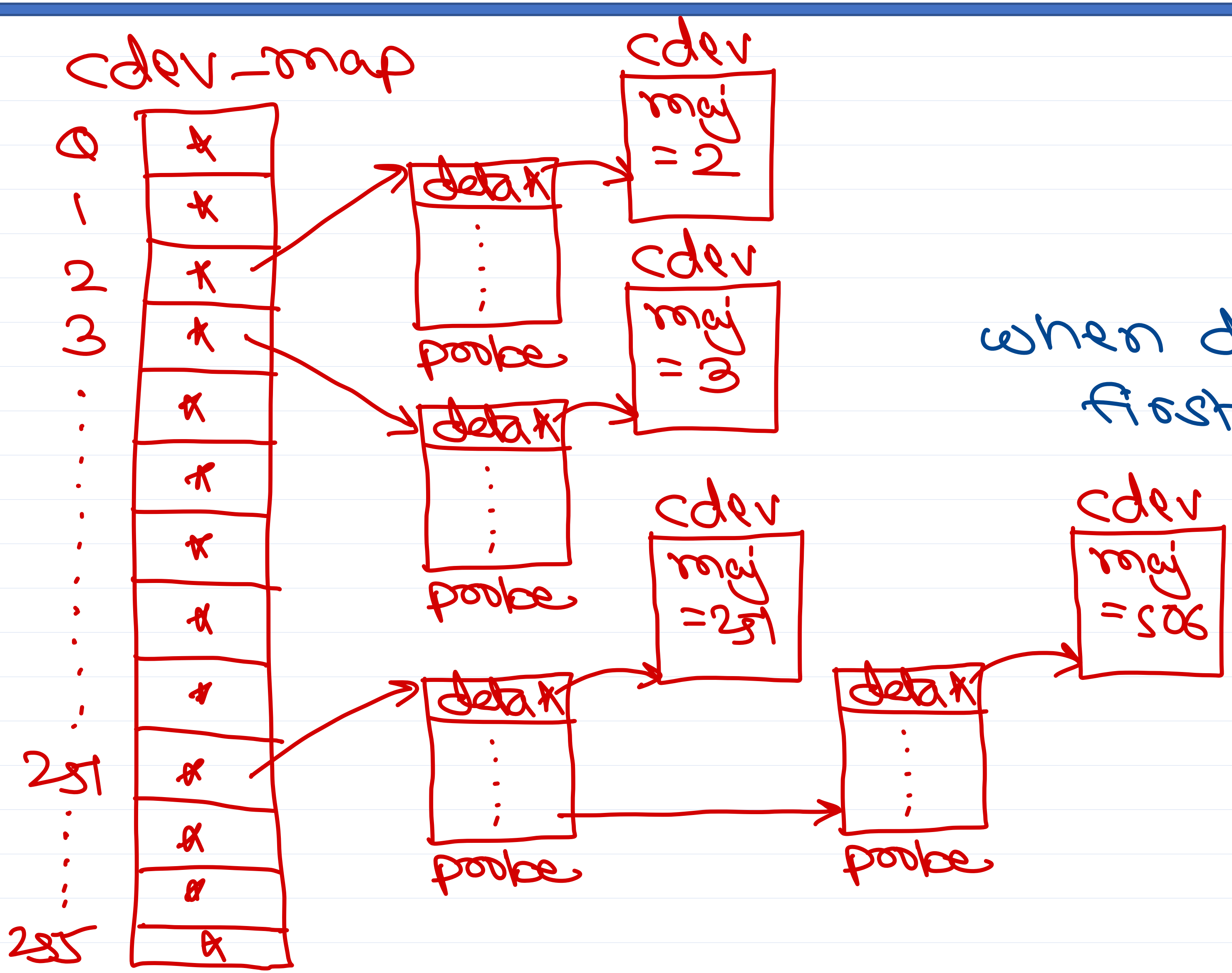
pchar_write()
{...}

```
struct kobj_map {
  struct probe {
    dev_t dev;
    unsigned long range;
    ...
    void *data;
  } *probes[255];
  ...
};
```

struct kobj_map cdev_map;

```
struct cdev {
    struct kobject kobj;
    struct module *owner;
    struct file_operations *ops;
    struct list_head list;
    dev_t dev;
    unsigned count;
};
```

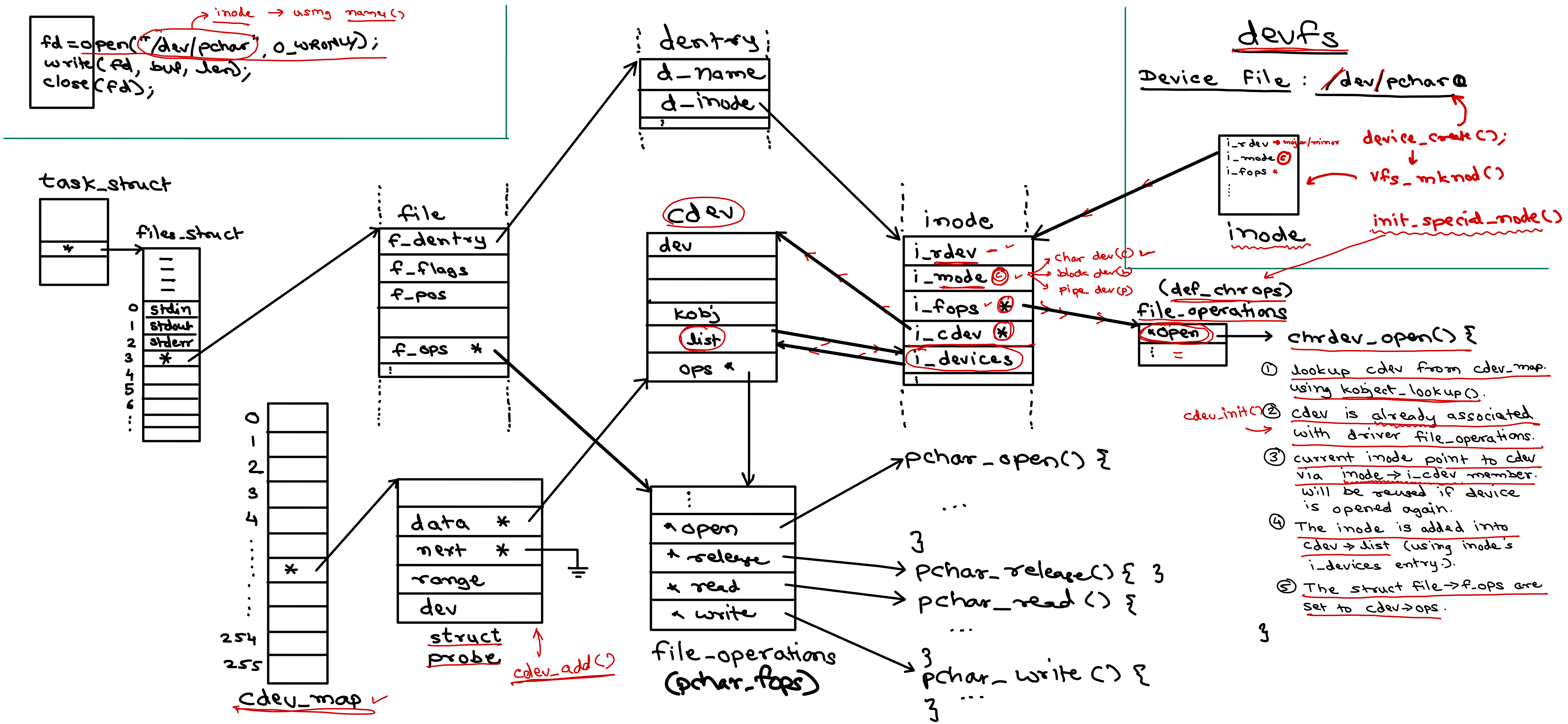
cdev_add()



key = major e.g. 506
hash fn = key % 255
e.g. 506 % 255 = 251

when device file is opened for first time, its cdev will be searched from this cdev_map using dev no (i_rdev).

Execution Flow of Pseudo Char Device Driver

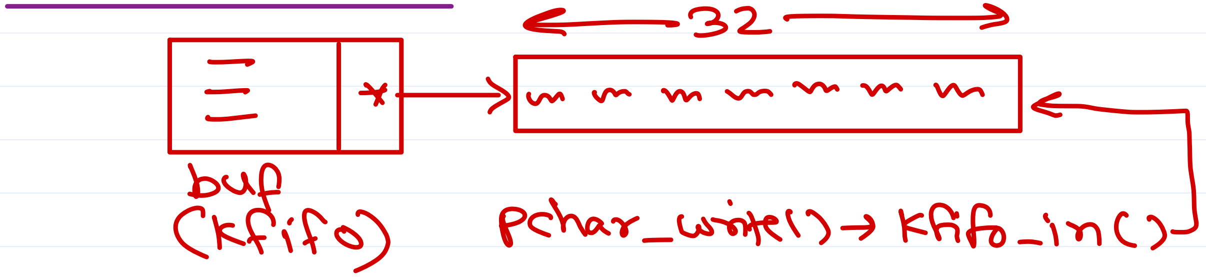


```
fd = open("/dev/pchar", O_WRONLY);
write(fd, buf, len);
close(fd);
```

chrdev_open() {
1. lookup cdev from cdev_map. using kobject_lookup().
2. cdev is already associated with driver file_operations.
3. current inode point to cdev via inode->i_cdev member. will be reused if device is opened again.
4. The inode is added into cdev->list (using inode's i_devices entry).
5. The struct file->f_ops are set to cdev->ops.
}

Waiting Queue

pchar device



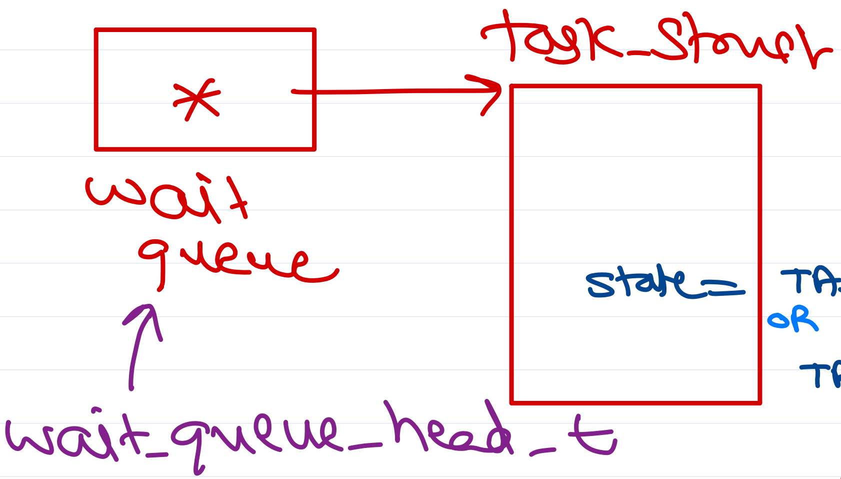
if kfifo is full, then block the writing process.

↳ wait_event_interruptible()

OR

wait_event()

→ kfifo_is_full()



task_struct ← of writer process (if kfifo is full)

state= TASK_INTERRUPTIBLE → blocked until data is popped from fifo. OR Signal is received.

OR

TASK_UNINTERRUPTIBLE → blocked until data is popped from fifo.

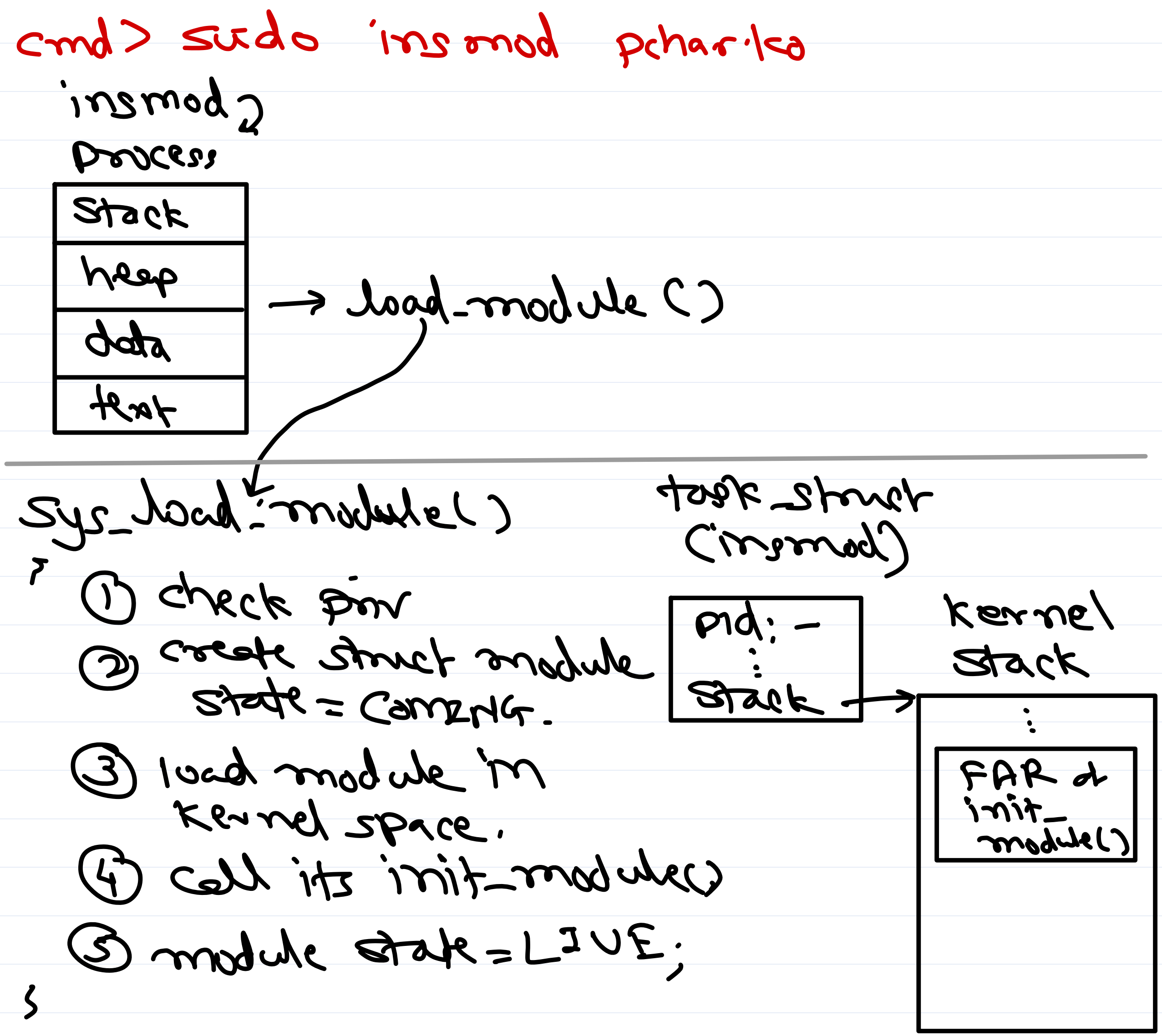
wake up blocked process, when some data is removed from fifo → kfifo_out() - read op.

wake_up()

OR

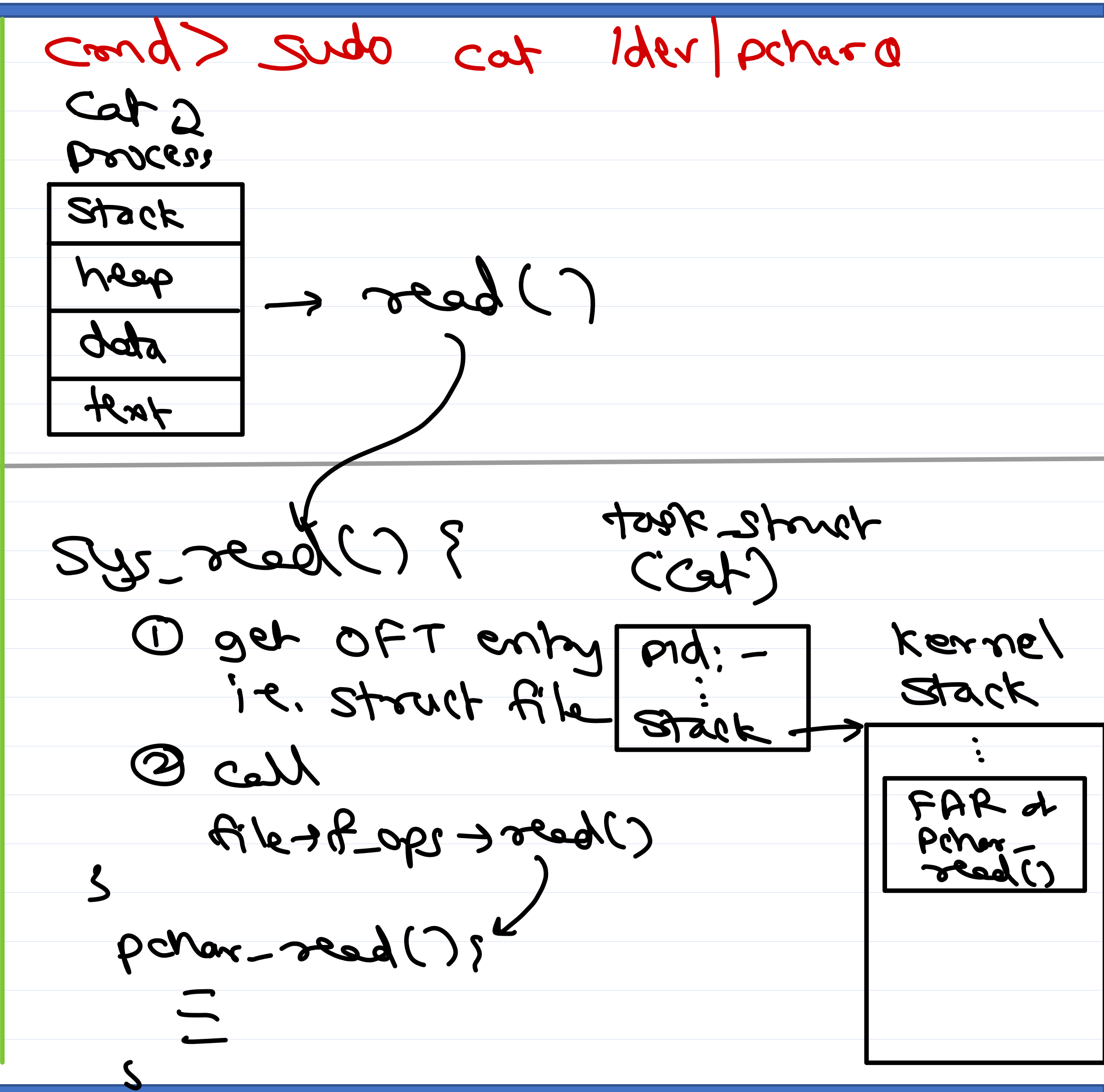
wake_up_interruptible()

Kernel stack - to create FARs of kernel functions



cmd> sudo insmod pchar.ko

```
sys_load_module()
{
    ① check perm
    ② create struct module
        state = COMING.
    ③ load module in
        kernel space.
    ④ call its init_module()
    ⑤ module state = LIVE;
}
```



cmd> sudo cat /dev/pchar0

```
sys_read() {
    ① get OFT entry
        i.e. struct file
    ② call
        file→f_ops→read()
}
pchar_read();
    =
}
```

Get current process

* Each process (task) have a kernel stack.

- on x86-32 arch, kernel stack size = 8KB.
- the kernel stack is 8KB aligned i.e. its addr is multiple of 8KB (in kernel space).
- task_struct has pointer to kernel stack.

* thread_info obj placed at the bottom of kernel stack (on x86).

- it contains info about current thread & pointer to current task_struct.

* current_thread_info() returns addr of current task's thread_info object.

→ sp & -8192

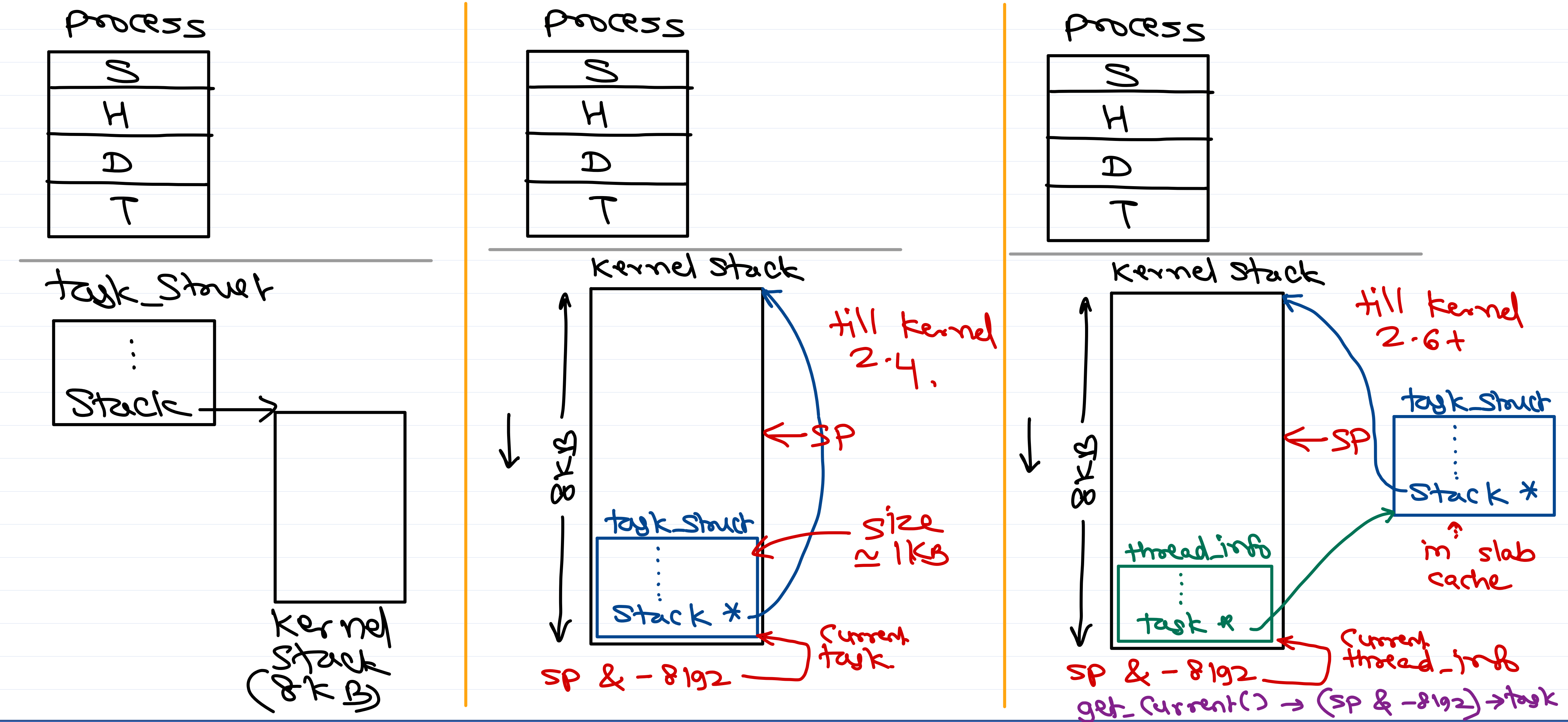
* current macro returns current task_struct.

```
#define current get_current()
```

current_thread_info() → task

returns current process's task_struct address.

task_struct and thread_info and kernel stack



Synchronization in Kernel space

* multiple user processes may access kernel code (in drivers / syscalls / ...) simultaneously and may cause race condition.

* kernel provided a few sync objects:
    1. semaphore.
    2. mutex (since 3.18+)
    3. spinlock

semaphore in kernel space

<linux/semaphore.h>

semaphore

lock
count
wait_list

struct semaphore s;

init sem with given cnt

sema_init(&s, init cnt);

down(&s); → decr cnt & if -ve block process in uninterruptible sleep

down_interruptible(&s);

→ decr cnt & if -ve block process in interruptible sleep

up(&s); → incr count & wake up blocked process.

mutex in kernel space

3.18 + <linux/mutex.h>

mutex

owner
wait_lock
wait_list

struct mutex m;

mutex_init(&m);

mutex_lock(&m);

mutex_lock_interruptible(&m);

mutex_unlock(&m);

mutex_destroy(&m);

Semaphore Internals

Solution 1: disable intrs

process 1

```
sem_p() {
    disable intrs;
    count--;
    ==
    enable intrs;
}
```

process 2

```
sem_v() {
    disable intrs;
    count++;
    ==
    enable intrs;
}
```

```
intr handler()
{
    ==
    scheduler();
    dispatcher();
}
```

Limitation: In multiprocessor env disabling intrs will disable them on current CPU only. Other process on other CPU can still cause the race condition.

Solution 2: spin locks

lock

~~1~~ ~~0~~ 0

process 1

```
sem_p() {
    spin_lock(&l);
    count--;
    ==
    spin_unlock(&l);
}
```

process 2

```
sem_v() {
    spin_lock(&l);
    count++;
    ==
    spin_unlock();
}
```

#include <linux/spinlock.h>

declare spinlock obj.

```
spinlock_t l;
```

init the spinlock

```
spin_lock_init(&l);
```

lock the spinlock:

```
spin_lock(&l);
```

unlock the spinlock:

```
spin_unlock(&l);
```

Spinlock

- hw based sync mechanism.
- it uses special instructions of exclusive access/bus holding.
  1. ARM7 → SWP instruction.
     - Reference: Sloss book.
  2. ARM Cortex A/M →
     - LDREX, STREX, CLREX
     - Reference: Yiu book.
- these special instruction - test_and_set() kind.
  - testing (read) var value and setting var value is done in exclusive access to bus.
  - only one CPU can access bus at a time.

spinlock pseudo code:

```
lock

① spinlock_init():
    lock = 0;
② spin_lock():
    while(lock == 1)
        ;
    lock = 1;
③ spin_unlock():
    lock = 0;
```

ARM instructions:

LDREX : Load var value in CPU regr and mark that addr for exclusive access.

STREX : Store CPU regr value in given var. STREX will be successful only if it is done after LDREX for that addr.

CLREX : clear exclusive access for given addr (marked by LDREX).

addr of lock var in RAM

1

```
spinlock(lock) {
    do {
        while(__LDREX(lock) != 0)
            ;
        Status = __STREX(1, lock);
    } while (status != 0);
    __DMB();
}
```

Spinlock → in Linux Kernel.

```
#include <linux/spinlock.h>
 spinlock_t lock;

 init:
     spin_lock_init(&lock);
         ↳ lock=0;
 lock:
     spin_lock(&lock);
         ↳ while (lock != 0)
                ;
             lock = 1;
 unlock:
     spin_unlock(&lock);
         ↳ lock = 0,
```

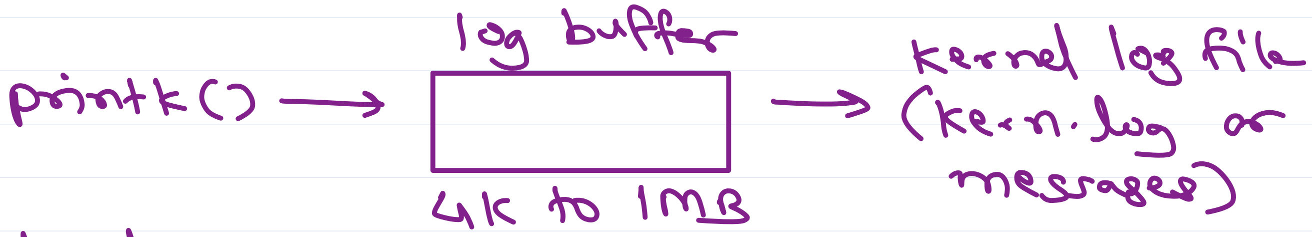
Kernel debugging techniques

① debugging by printing

printk() → log levels

0 7

EMERG...DEBUG



printk() → log buffer (4k to 1MB) → kernel log file (kern.log or messages)

kernel log daemon → syslogd or klogd.

② debugging by querying

① ioctl() operation ② procfs entry.

③ debugging by watching

strace command → shows which sys calls are called

④ System faults/hangs

① Kernel OOPs message → register values + stack trace.

② Kernel panic - crash - Scheduler stopped

⑤ Kernel debuggers

① gdb ② kdb

Thank you!

Nilesh Ghule <nilesh@sunbeaminfo.com>