Linux Device Drivers

- Advanced Char Drivers

Sunny B
(sunnyb@cdac.in)
C-DAC, Hyderabad

Agenda

- Circular Buffers
- Handling Multiple Devices
- Wait Queues
- ioctl

Circular Buffers

Structure & Macros

- <linux/circ_buf.h>

```
struct circ_buf {
    char *buf;
    int head;          /* Elements are inserted at 'head' */
    int tail;          /* Elements are deleted at 'tail'  */
};
```

- CIRC_CNT(head, tail, size) : Returns the buffer count
- CIRC_SPACE(head, tail, size) : Returns the space available
- 'size' should always be taken as a power of 2.

Operations on Circular Buffer

- Insertion :

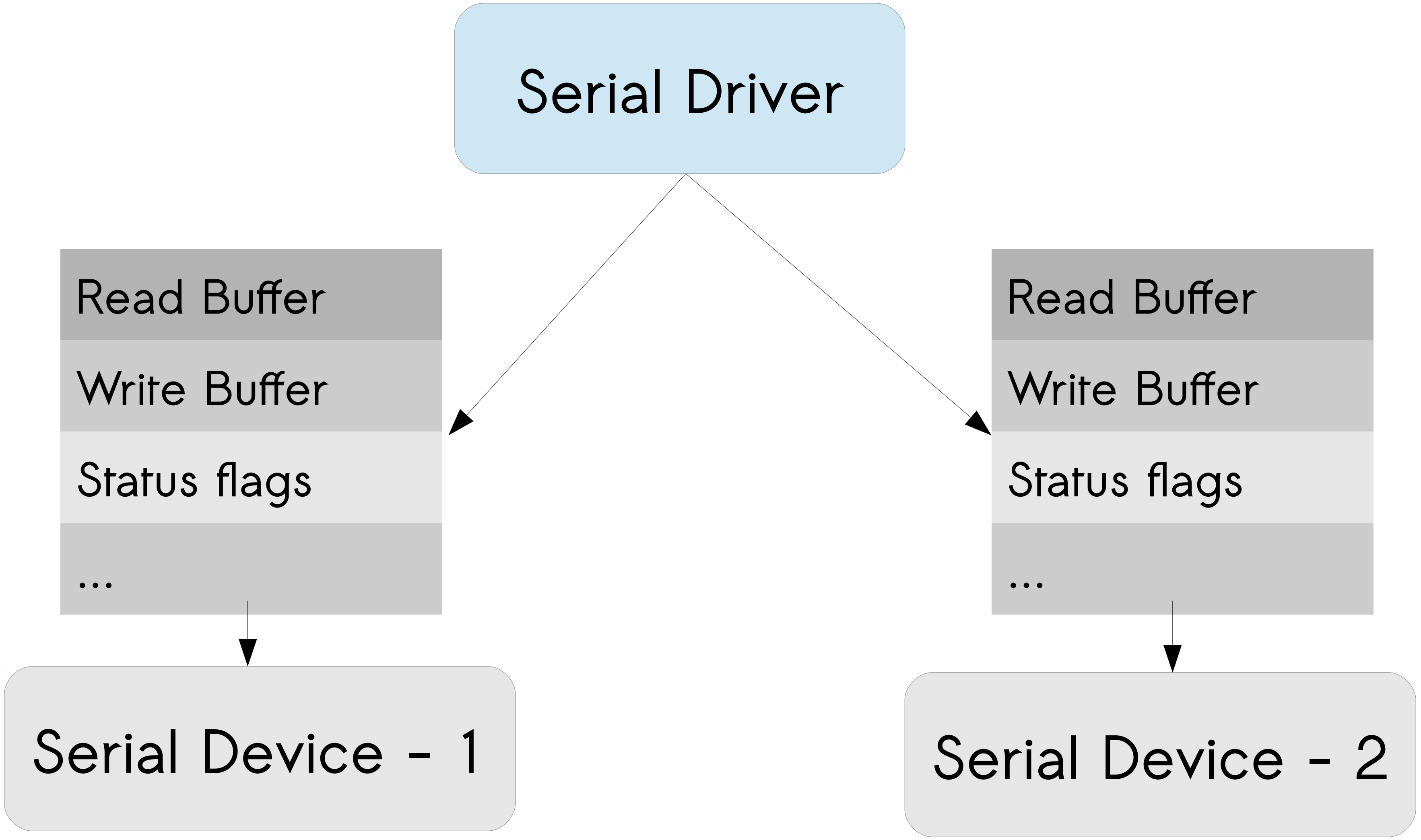
```
if (CIRC_SPACE(head, tail, size) >= 1) {
    buf[head] = ch;
    head = (head + 1) & (size - 1);
}
```

- Deletion :

```
if (CIRC_CNT(head, tail, size) >= 1) {
    ch = buf[tail];
    tail = (tail + 1) & (size - 1);
}
```

Handling Multiple Devices

Concept of multiple devices



Handling Multiple Devices

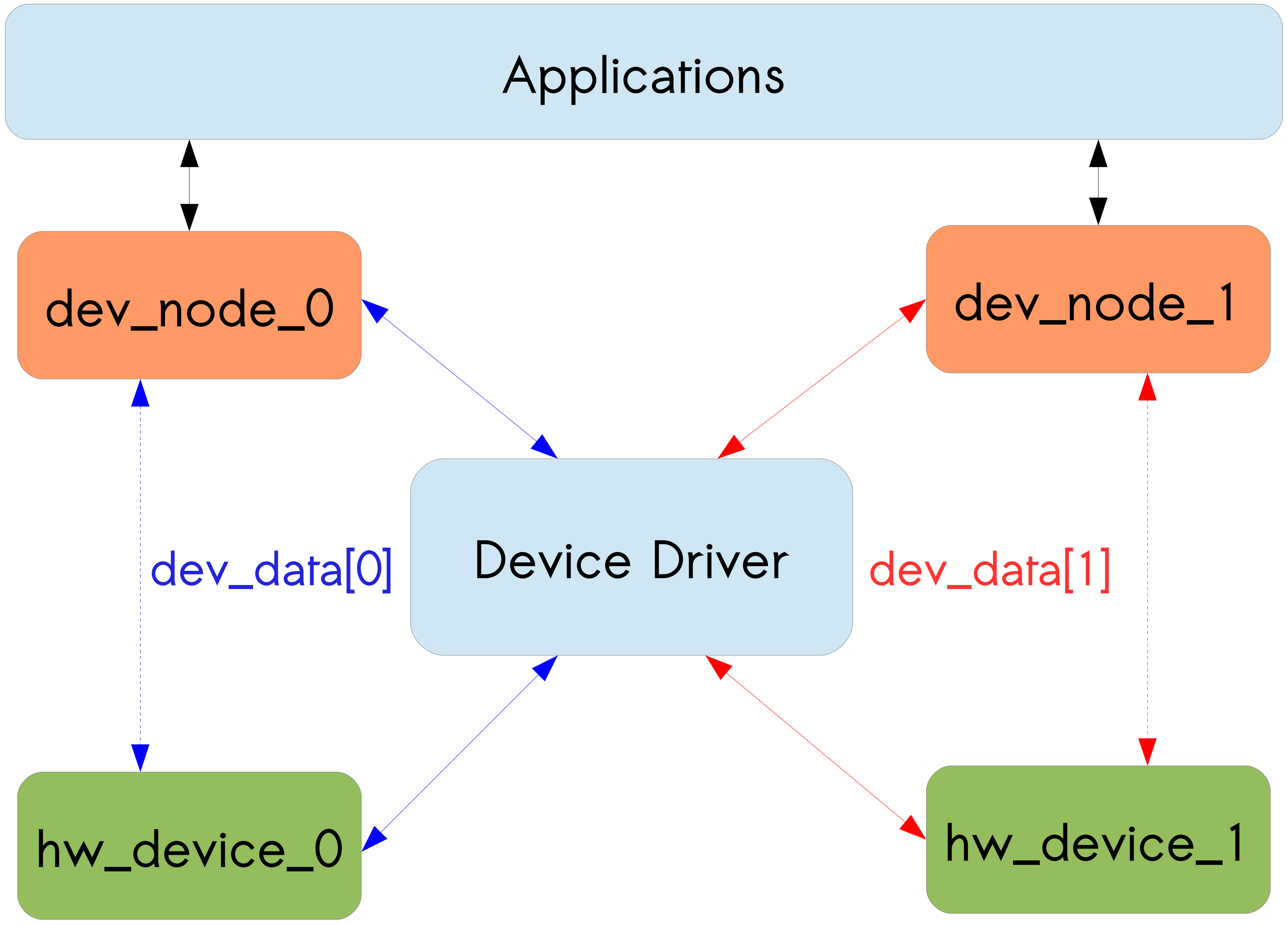
- Device Specific Structure
- Handling Devices across *file_operations*

Device Specific Structure

- #define NUM_DEVICES 2
- struct device_data {

  ```
      struct circ_buf cbuff;
      int buff_size;
      struct cdev c_cdev;
      ...
  };
  ```
- struct device_data dev_data[NUM_DEVICES];

Device Specific Structure



Handling Devices across *file_operations*

- open(struct inode inod, struct file filp)

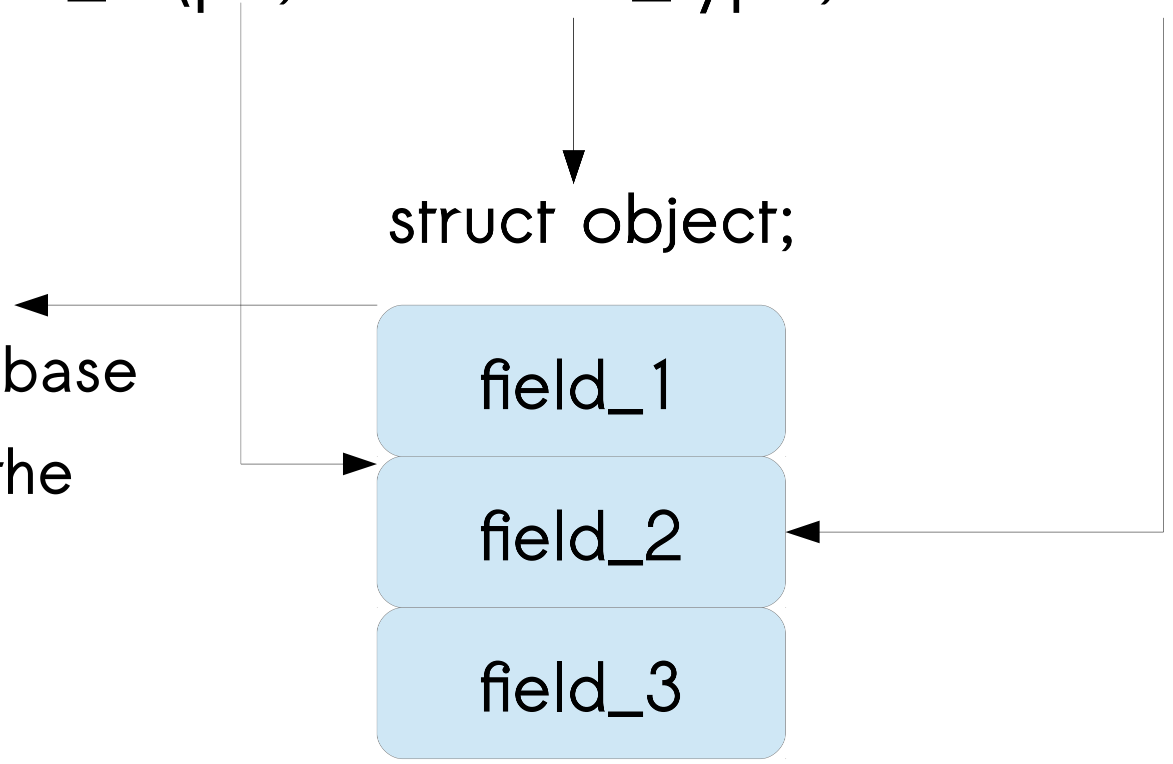
```
{
    struct devData *data = container_of(inod->i_cdev,
                                        struct dev_data,
                                        c_cdev);

    /* Setup the device specific data */

    filp->private_data = data;
}
```

container_of()

- container_of(ptr, container_type, container_field);



Returns the base address of the structure

Handling Devices across *file_operations*

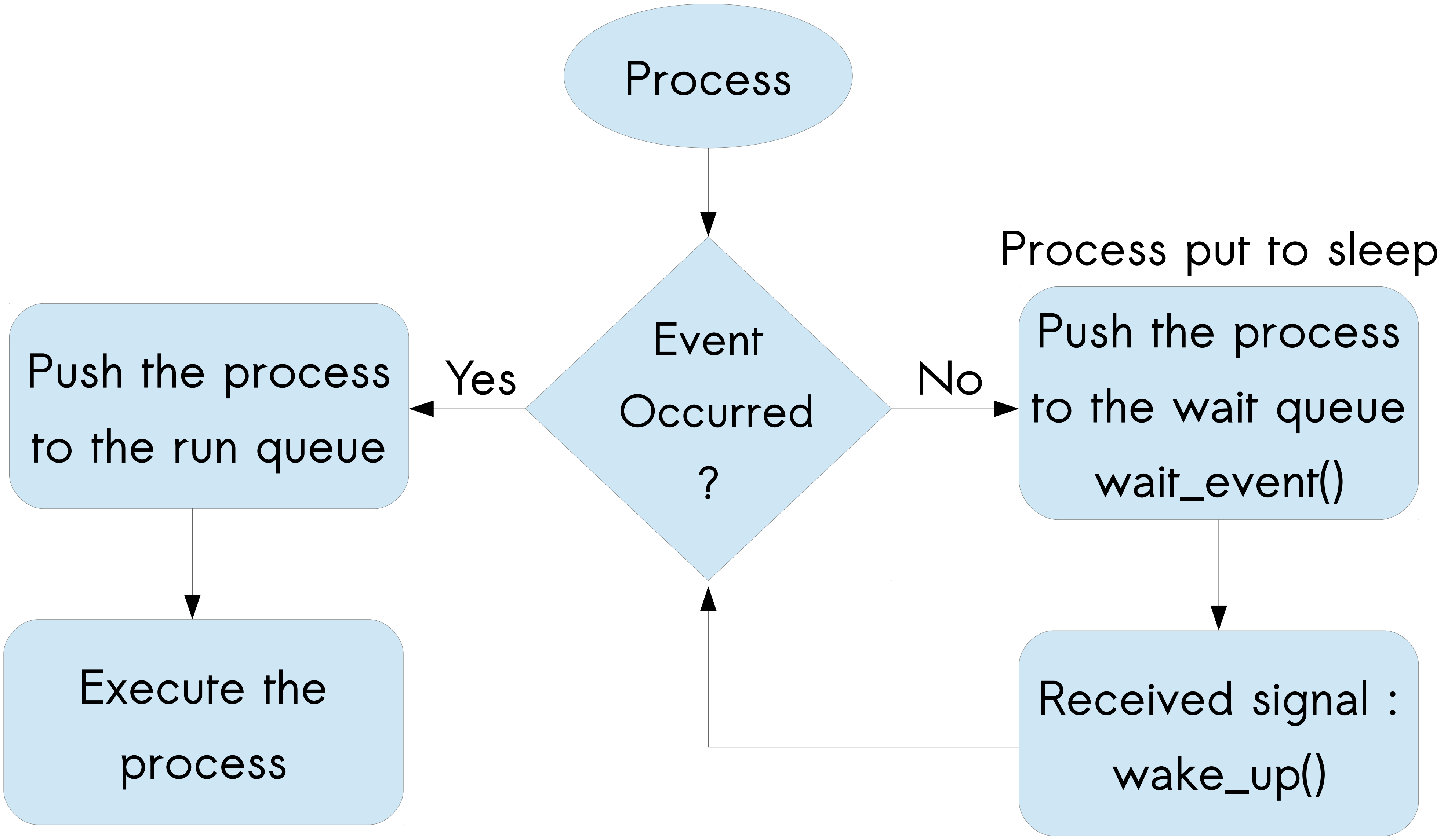
- read (struct file *filp, ...) {
```
    struct devData *data = filp->private_data;
    ...
}
```
- write (struct file *filp, ...) {
```
    struct devData *data = filp->private_data;
    ...
}
```
- release (struct inode *inod, struct file *filp) {
```
    struct devData *data = filp->private_data;
    ...
}
```

Wait Queues

Introduction

- We cannot assume that data is always available to us. For example,
  - On read(), data may not be immediately available.
  - During this period, the process is put to sleep until the data is available.
- Process states in Linux :
  - Runnable/Running : TASK_RUNNING
  - Sleeping/ Waiting/Blocked :
    - TASK_INTERRUPTIBLE : Responds to signals
    - TASK_UNINTERRUPTIBLE : Does not respond to signals

Waitqueues in the kernel


Process
Event Occurred ?
Yes
Push the process to the run queue
Execute the process
No
Process put to sleep
Push the process to the wait queue wait_event()
Received signal : wake_up()

Kernel Contexts

- In the kernel space, processes executes in one of the two contexts :
  - Process Context : Runs on behalf on user process or as a kernel thread
  - Interrupt Context : Runs as a part of the timer interrupt service routine

Precautions to be taken

- Never sleep when a process is executing in an interrupt context, as :
  - They are not associated with a 'task_struct'.
  - No backing process to wake up
- Never put a process to sleep unless it is assured that some event will wake it up

Accessing the kernel Wait Queues

- Waitqueues are manged by the structure *wait_queue_head_t.*
- It can be defined and initialised as :
  - <linux/wait.h>
  - Statically : `DECLARE_WAIT_QUEUE_HEAD(name);`
  - Dynamically :

    ```
    wait_queue_head_t my_que;
    init_waitqueue_head(&my_que);
    ```

Go to sleep..

- The macros puts the process to sleep :
  - wait_event(queue, condition)
  - wait_event_interruptible(queue, condition)
  - wait_event_timeout(queue, condition, timeout)
  - wait_event_interruptible_timeout(queue, condition, timeout)
- *queue* is the wait queue head
- *condition* is an arbitrary boolean expression that is evaluated before and after sleeping
- Until the *condition* gets true, process continues to sleep

Go to sleep...

- *wait_event()* puts the process to uninterruptible sleep
- The preferred alternative is *wait_event_interruptible(),* as it can be interrupted by signals.
- Return values for the above two is non-zero, if it receives the signal other than what we are waiting for, else, 0.
- Hence it is always mandatory to check the return value :

```
if(wait_event_interruptible(...))
    return -ERESTARTSYS;
```

- The two variants with timeout are similar with the above two except they wait till time expires mentioned as third argument (expressed in *jiffies*)

Time to wake up...

- Waking up is done by some other process or an interrupt handler.
- Wakeup functions :
  - `void wake_up(wait_queue_head_t *);`
  - `void wake_up_interruptible(wait_queue_head_t *);`
- Once the process has made an attempt to wake up the process, the sleeping process evaluates the condition again. If the condition is satisfied, it wakes up completely, else it is put back to sleep.

Blocking & Non-Blocking processes

- Not every process can be put to sleep.
- Before putting a process to sleep, check whether the process capable of sleeping or not.
- This can be achieved through O_NONBLOCK flag, which can be set when calling open() in user space :

```
if (filp->f_flags & O_NONBLOCK)
    return -EAGAIN;
wait_event( ... );
```

ioctl

ioctl

- Provides the ability to perform various types of hardware control via the device driver
- Examples :
  - Eject the CD-ROM tray.
  - Increase the speed of the motor.
  - Report error information.
  - Read the status register
  - Set the baud rate to default value, etc

Prototype of ioctl

- User space :

  int ioctl(int fd, int cmd, ...);

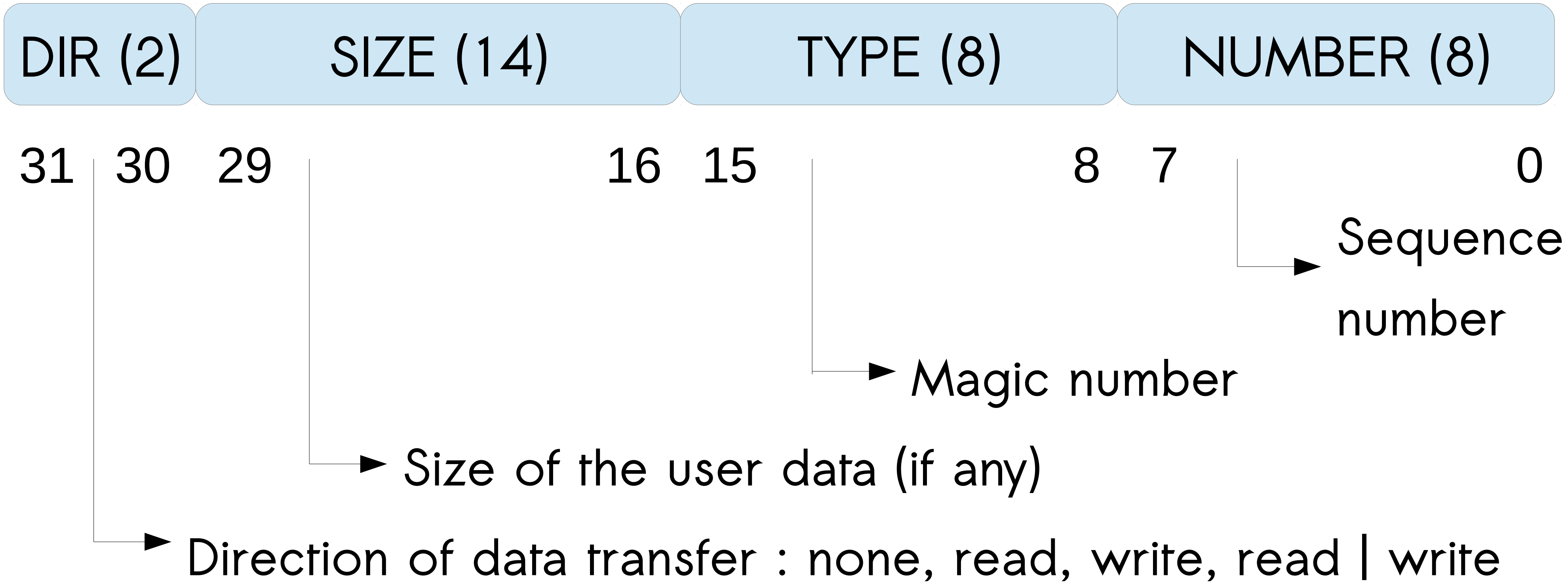
  - fd : File descriptor pertaining to the device node
  - cmd : Command to be executed
  - Third argument is optionally used to send or receive data corresponding to the command.
- Kernel Space :

  long (*unlocked_ioctl) (struct file *filp, unsigned int cmd, unsigned long arg);

  - *inode* and *filp* are the same as in with open and release functions.
  - cmd : The command passed from the user space
  - arp : Optional third argument from the user space.

ioctl commands

- Should be unique among all the drivers in the system.



DIR (2)
SIZE (14)
TYPE (8)
NUMBER (8)
31
30
29
16
15
8
7
0
Sequence number
Magic number
Size of the user data (if any)
Direction of data transfer : none, read, write, read | write

ioctl command macros

- Macros to create ioctl commands :
  - _IO(type, nr) : Command with no argument
  - _IOR(type, nr, datatype) : Reading data from the driver
  - _IOW(type, nr, datatype) : Writing data to the driver
  - _IOWR(type, nr, datatype) : Bidirectional transfer
- Macros to decode ioctl commands :
  - _IOC_DIR (cmd) : _IOC_NONE, _IOC_READ, _IOC_WRITE
  - _IOC_TYPE(cmd)
  - _IOC_NR(cmd)
  - _IOC_SIZE(cmd)

Short data transfers

- The following macros are the optimized form to transfer data to/from user-space :
  - put_user(data, ptr)
    - Writes short amount of data to user space
    - Alternate form of copy_to_user();
  - get_user(data, ptr)
    - Reads small amounts of data from userspace
    - Alternate to copy_from_user()
- The size of the transfer may be as small as one, two, four or eight bytes

ioctl example : header (my_ioctl.h)

- /* Define the type/magic number for the driver */

  ```
  #define MY_TYPE  'x'
  ```

- /* Define the commands */

  ```
  #define CMD_1 _IO(MY_TYPE, 0)
  #define CMD_2 _IOR(MYTYPE, 1, int)

  ...
  ```

ioctl example : Driver code

- /* Include the commands header */

```
#include “my_ioctl.h”
```

- /* Declare the file_operations */

```
struct file_operations = {... , .unlocked_ioctl = my_ioctl };
```

- /* Implement the ioctl function */

```
long my_ioctl(struct file *filp ,unsigned int cmd, unsigned long arg) {
   switch (cmd) {
         case CMD_1 : ...; break;
         case CMD_2 : ...; put_user(0x3f, (int __user *) arg);
                                               break;
         default : return -EINVAL;
   } return 0; }
```

ioctl example : User code

- /* Include the headers */

  ```
  #include "my_ioctl.h"
  #include <sys/ioctl.h>
  ```

- /* Call ioctl on the device */

  ```
  int main() {
      ...
      ioctl(fd, CMD_1);
      ioctl(fd, CMD_2, &i);
  }
  ```

Capabilities

- Some ioctl operations must be performed by only a user who is capable, like the 'root' user.
- In order to bring out such privileged tasks into the kernel, we can use :

  int capable(int capability);

  - Returns true if capable, else false

Capabilities : example

-
  ```
  long ioctl( ... ) {
      switch(cmd) {
          ...
          /* Only root user can access this command */
          case CMD_2 : if (!capable(CAP_SYS_ADMIN))
                                return -EPERM;
                           put_user( ... );
                                break;
      ...
      }
      return 0;
  }
  ```

References

- Jonathan Corbet, Alessandro Rubini and Greg Kroah-Hartman,"Linux Device Drivers",3rd Edition, O'Reilly Publications
- Robert Love, "Linux Kernel Development", 3rd Edition, Developer's Library
- Circular Buffers : http://lwn.net/Articles/378262/

Thank You :)