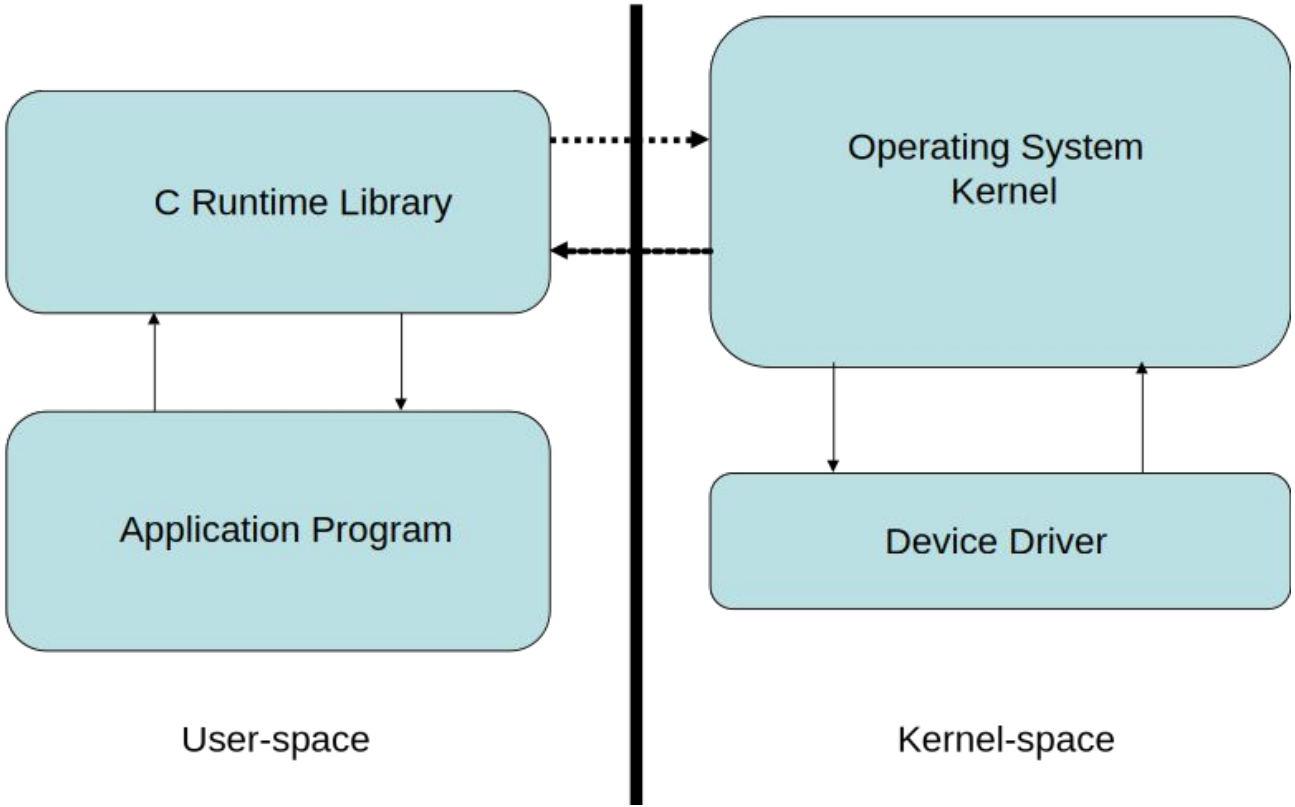
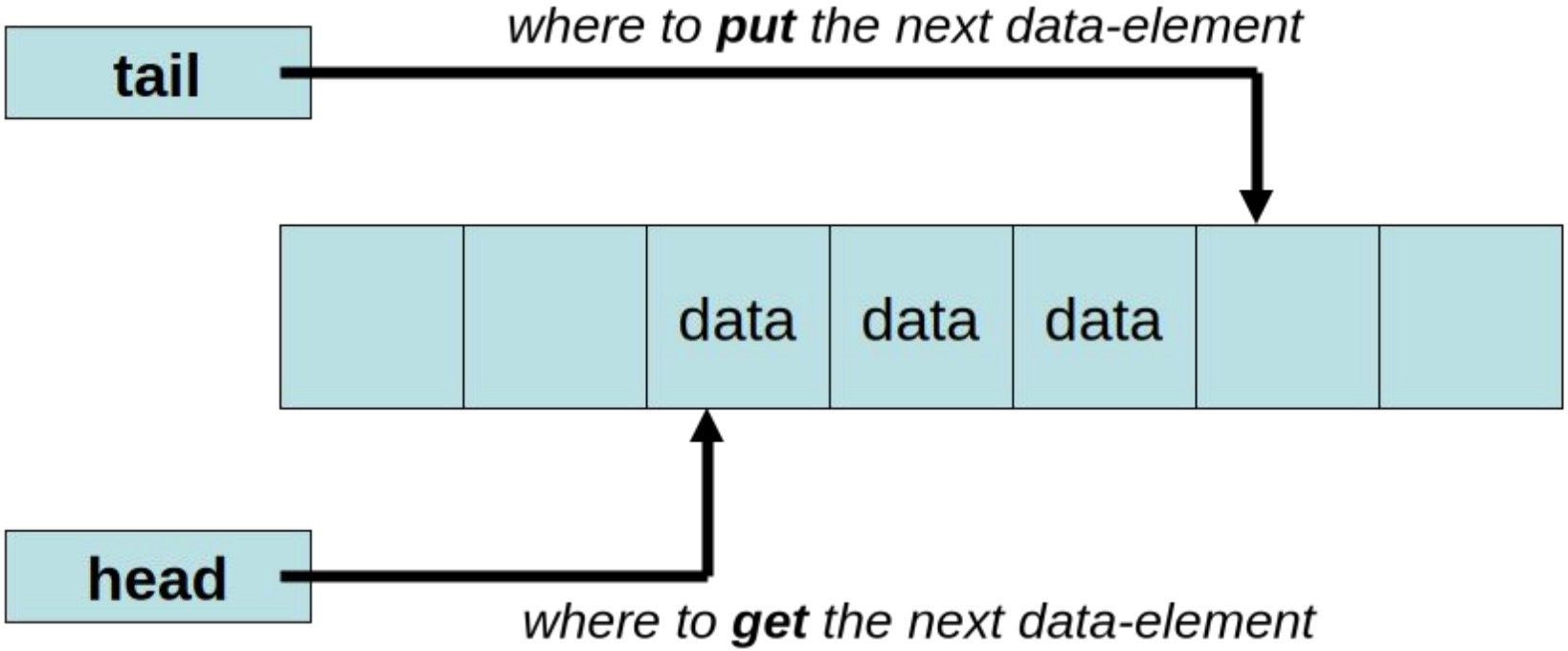
Character Device Driver

How system-calls work



How a ring buffer works



Driver Identification

- Character/Block drivers:
- Use ‘major-number’ to identify the driver
- Use ‘minor-numbers’ to distinguish among several devices the same driver controls
- Kernel also needs a driver-name
- Users need a device-node as ‘interface’

Creating our device node

- The ‘mknod’ command creates the node:
  $ mknod /dev/stash c 40 0

- The ‘chmod’ command changes the node access-permissions (if that’s needed):
  $ chmod a+rw /dev/stash

- Both commands normally are ‘privileged’

The ‘open’ function

- #include <fcntl.h>
- int open(const char *pathname, int flags);
- Converts a pathname to a file-descriptor
- File-descriptor is a nonnegative integer
- Used as a file-ID in subsequent functions
- ‘flags’ is a symbolic constant:

  O_RDONLY, O_WRONLY, O_RDWR

The ‘close’ function

- #include <unistd.h>
- int close(int fd);
- Breaks link between file and file-descriptor
- Returns 0 on success, or -1 if an error

The ‘read’ function

- #include <unistd.h>
- int read(int fd, void *buf, size_t count);
- Attempts to read up to ‘count’ bytes
- Bytes are placed in ‘buf’ memory-buffer
- Returns the number of bytes read
- Or returns -1 if some error occurred
- Return-value 0 means ‘end-of-file’

The ‘write’ function

- #include <unistd.h>
- int write(int fd, void *buf, size_t count);
- Attempts to write up to ‘count’ bytes
- Bytes are taken from ‘buf’ memory-buffer
- Returns the number of bytes written
- Or returns -1 if some error occurred
- Return-value 0 means no data was written

The ‘lseek’ function

- #include <unistd.h>
- loff_t lseek(int fd, loff_t off, int whence);
- This function moves the file’s pointer
- Three ways to do the move:
  - SEEK_SET: move from beginning position
  - SEEK_CUR: move from current position
  - SEEK_END: move from ending position
- (Could be used to determine a file’s size)

Default is ‘Blocking’ Mode

- The ‘read()’ function normally does not return 0 (unless ‘end-of-file’ is reached)
- The ‘write()’ function normally does not return 0 (unless there’s no more space)
- Instead, these functions ‘wait’ for data
- But ‘busy-waiting’ would waste CPU time, so the kernel will put the task to ‘sleep’
- This means it won’t get scheduled again (until the kernel ‘wakes up’ this task)

How multitasking works

- Can be ‘cooperative’ or ‘preemptive’
- ‘interrupted’ doesn’t mean ‘preempted’
- ‘preempted’ implies a task was switched

Tasks have various ‘states’

- A task may be ‘running’
- A task may be ‘ready-to-run’
- A task may be ‘blocked’

Kernel manages tasks

- Kernel uses ‘queues’ to manage tasks
- A queue of tasks that are ‘ready-to-run’
- Other queues for tasks that are ‘blocked’

Special ‘wait’ queues

- Needed to avoid wasteful ‘busy waiting’
- So Device-Drivers can put tasks to sleep
- And Drivers can ‘wake up’ sleeping tasks

How to use Linux wait-queues

- #include <linux/sched.h>
- wait_queue_head_t my_queue;
- init_waitqueue_head(&my_queue);
- sleep_on(&my_queue);
- wake_up(&my_queue);

- But can’t unload driver if task stays asleep!

‘interruptible’ wait-queues

- Device-driver modules should use:

  interruptible_sleep_on(&my_queue);

  wake_up_interruptible(&my_queue);

- Then tasks can be awakened by ‘signals’

How ‘sleep’ works

- Our driver defines an instance of a kernel data-structure called a ‘wait queue head’
- It will be the ‘anchor’ for a linked list of ‘task_struct’ objects
- It will initially be an empty-list
- If our driver wants to put a task to sleep, then its ‘task_struct’ will be taken off the runqueue and put onto our wait queue

How ‘wake up’ works

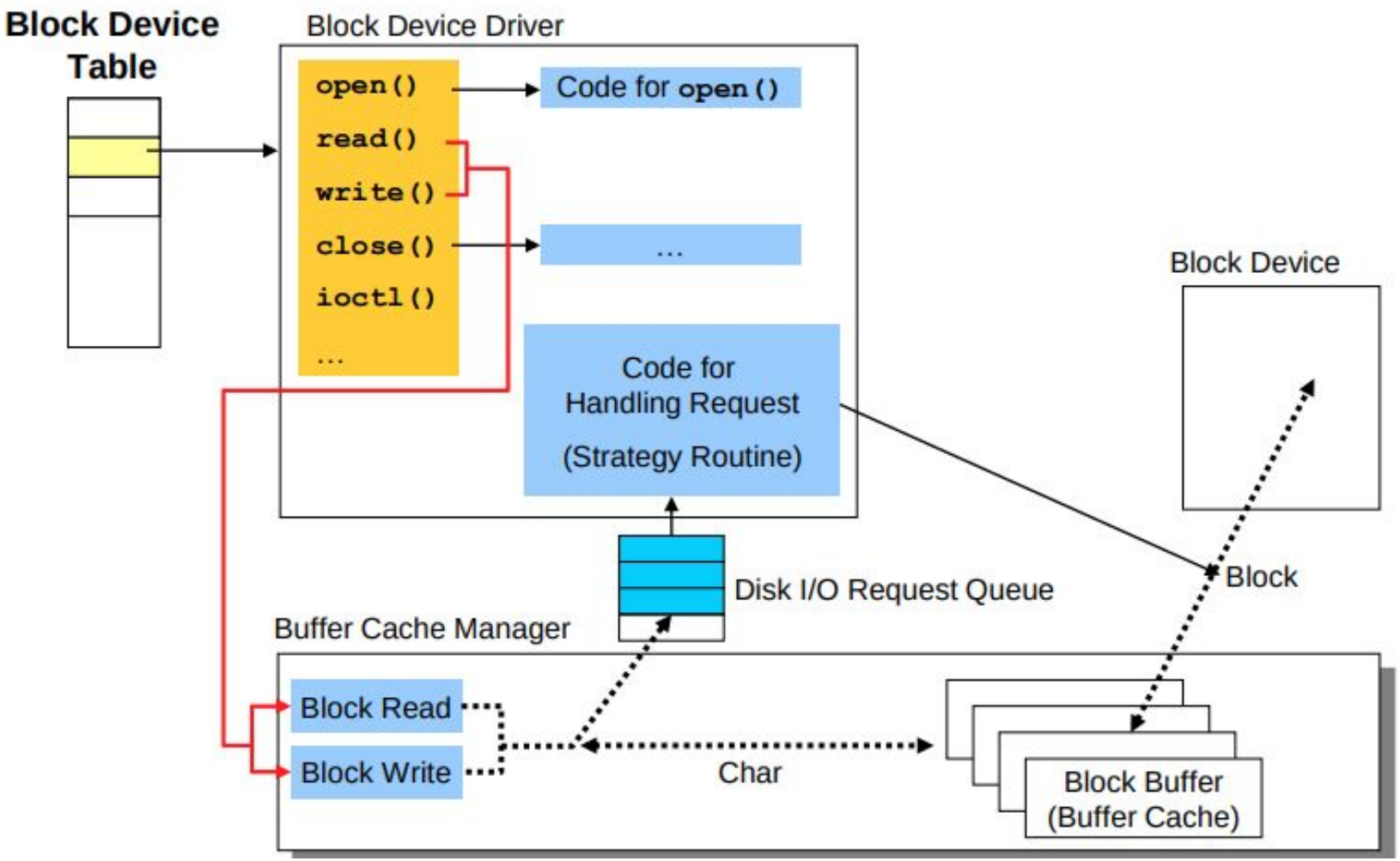
- If our driver detects that a task it had put to sleep (because no data-transfer could be done immediately) would now be allowed to proceed, it can execute a ‘wake up’ on its wait queue object
- All the task_struct objects that have been put onto that wait queue will be removed, and will be added to the CPU’s runqueue

Application to a ringbuffer

- A first-in first-out data-structure (FIFO)
- Uses a storage-array of finite length
- Uses two array-indices: ‘head’ and ‘tail’
- Data is added at the current ‘tail’ position
- Data is removed from the ‘head’ position

Ringbuffer (continued)

- One array-position is always left unused
- Condition head == tail means “empty”
- Condition tail == head-1 means “full”
- Both ‘head’ and ‘tail’ will “wraparound”
- Calculation: next = (next+1)%RINGSIZE;


Block Device Table
Block Device Driver
open()
read()
write()
close()
ioctl()
...
Code for open()
...
Code for Handling Request (Strategy Routine)
Block Device
Disk I/O Request Queue
Block
Buffer Cache Manager
Block Read
Block Write
Char
Block Buffer (Buffer Cache)