Operating Systems

Lecture 10b

Dr Ronald Grau

School of Engineering and Informatics

Spring term 2018

Previously

The I/O System

Interaction protocols

Blocking vs. non-blocking I/O

Recap questions

1. Which general factors make the I/O system of an operating system complex?
2. How do OS designers cope with this complexity?

3. What can we do with a block device that we cannot with a character device?

4. What are the advantages/disadvantages of

   - Polling?
   - Interrupt-driven I/O?
   - DMA?

5. What is the difference between blocking and non-blocking I/O?

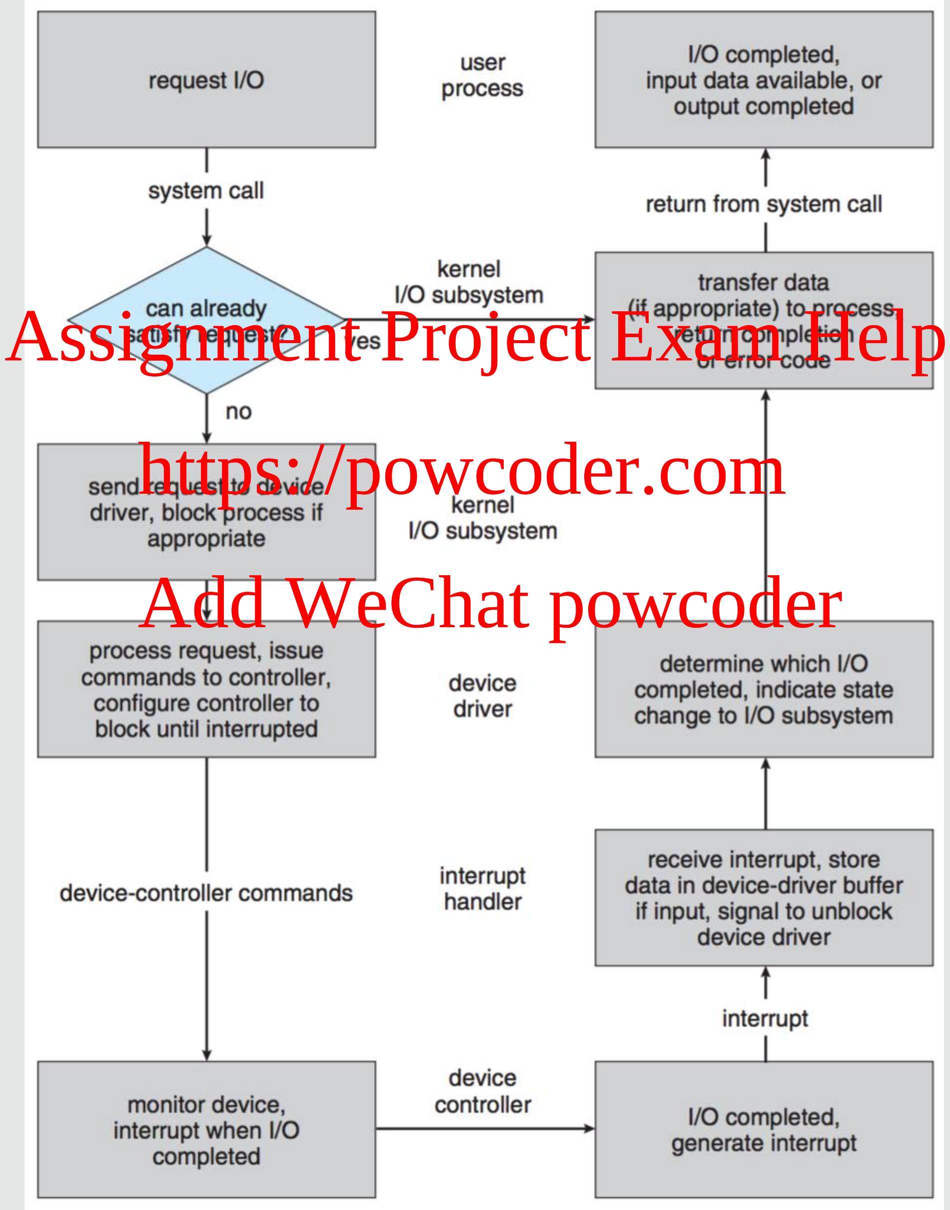
Example: Programmed I/O

```c
// This example is for an AVR microcontroller.
#include <avr/io.h> // defines constants, I/O registers, ...

#define F_CPU 8000000UL  // 8 MHz
#define USART_BAUDRATE 38400
#define BAUD_PRESCALE (((F_CPU / (UART_BAUDRATE * 16UL))) - 1)

unsigned char receiveByte() {
  while((UCSRA &(1<<RXC)) == 0); //busy wait on receive complete bit
  value = UDR;
}

void main() {
  // Set baud rate
  UBRRL = BAUD_PRESCALE;
  UBRRH = (BAUD_PRESCALE >> 8);
  // Enable receiver
  UCSRB = (1<<RXEN);
  while(1) {
    unsigned char value = receiveByte();
    // ... do something with value
  }
}
```

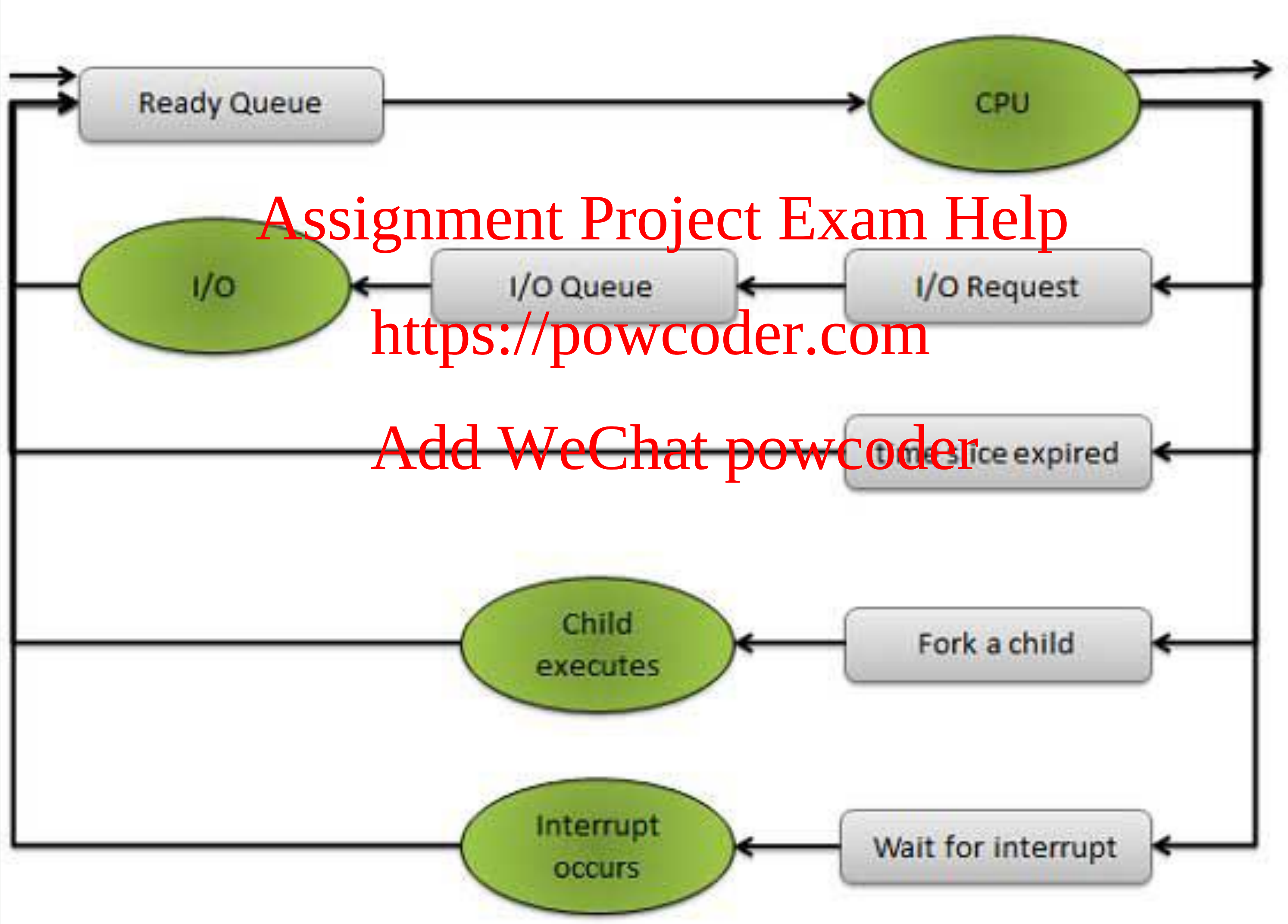
Example: Interrupt-Driven I/O


```
#include <avr/io.h>

#define F_CPU 8000000UL   // 8 MHz
#define USART_BAUDRATE 38400
#define BAUD_PRESCALE (((F_CPU / (UART_BAUDRATE * 16UL))) - 1)

volatile unsigned char value;

ISR(USART_RXC_vect) {
  value = UDR; // read UART register into value
}

void main() {
  UBRRL = BAUD_PRESCALE;
  UBRRH = (BAUD_PRESCALE >> 8);
  // Enable receiver and receive complete interrupt
  UCSRB = ((1<<RXEN) | (1<<RXCIE));
  sei(); // enable all interrupts
  while(1) {
    sleep(); // will wake up and execute after ISR
    // ... do something with value
  }
}
```

I/O Request Flow



I/O Queue


Ready Queue
CPU
I/O
I/O Queue
I/O Request
time slice expired
Child executes
Fork a child
Interrupt occurs
Wait for interrupt

I/O Performance Optimisation

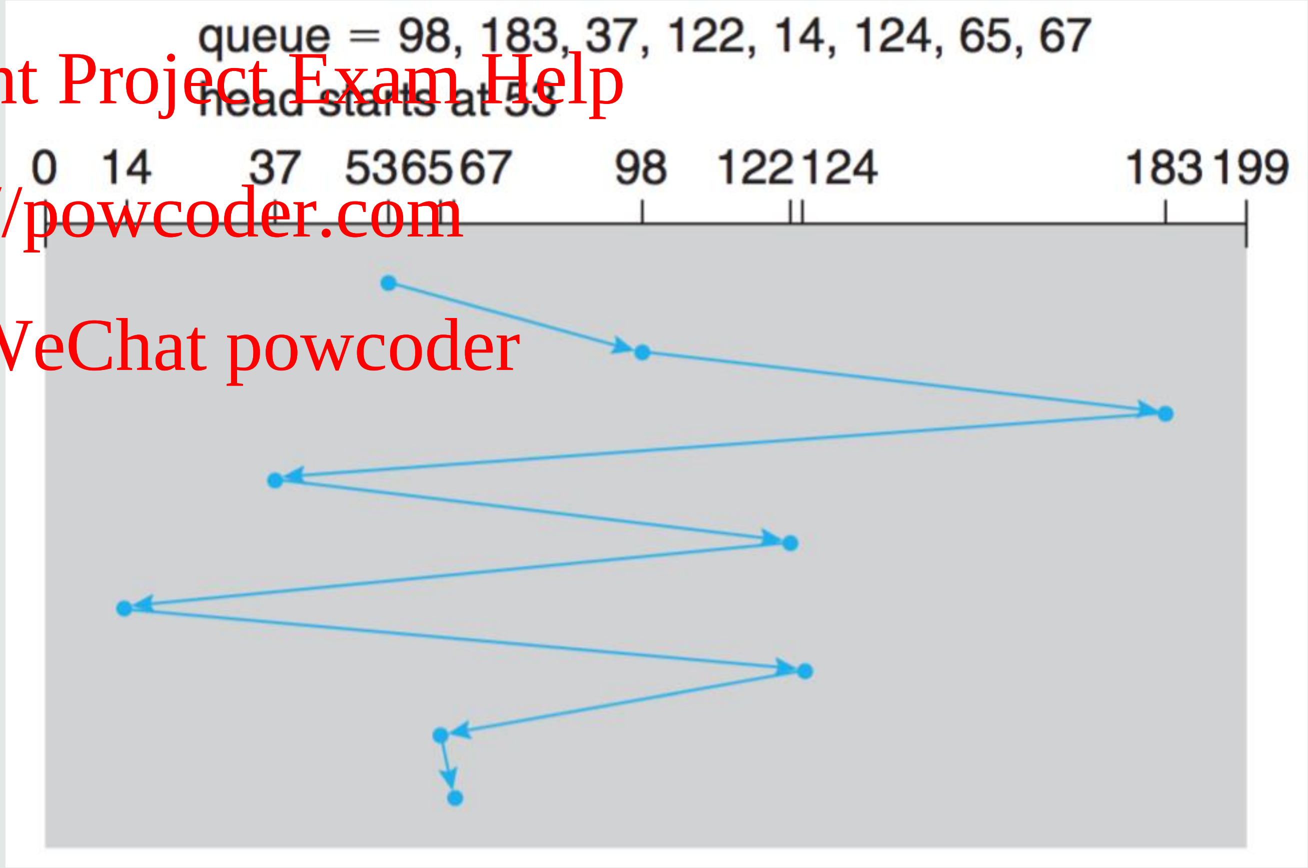
Disk operations in disk I/O queue

- Reorder requests to optimise access times?
  → scheduling

Example: Disk scheduling

- First Come First Served (FCFS)

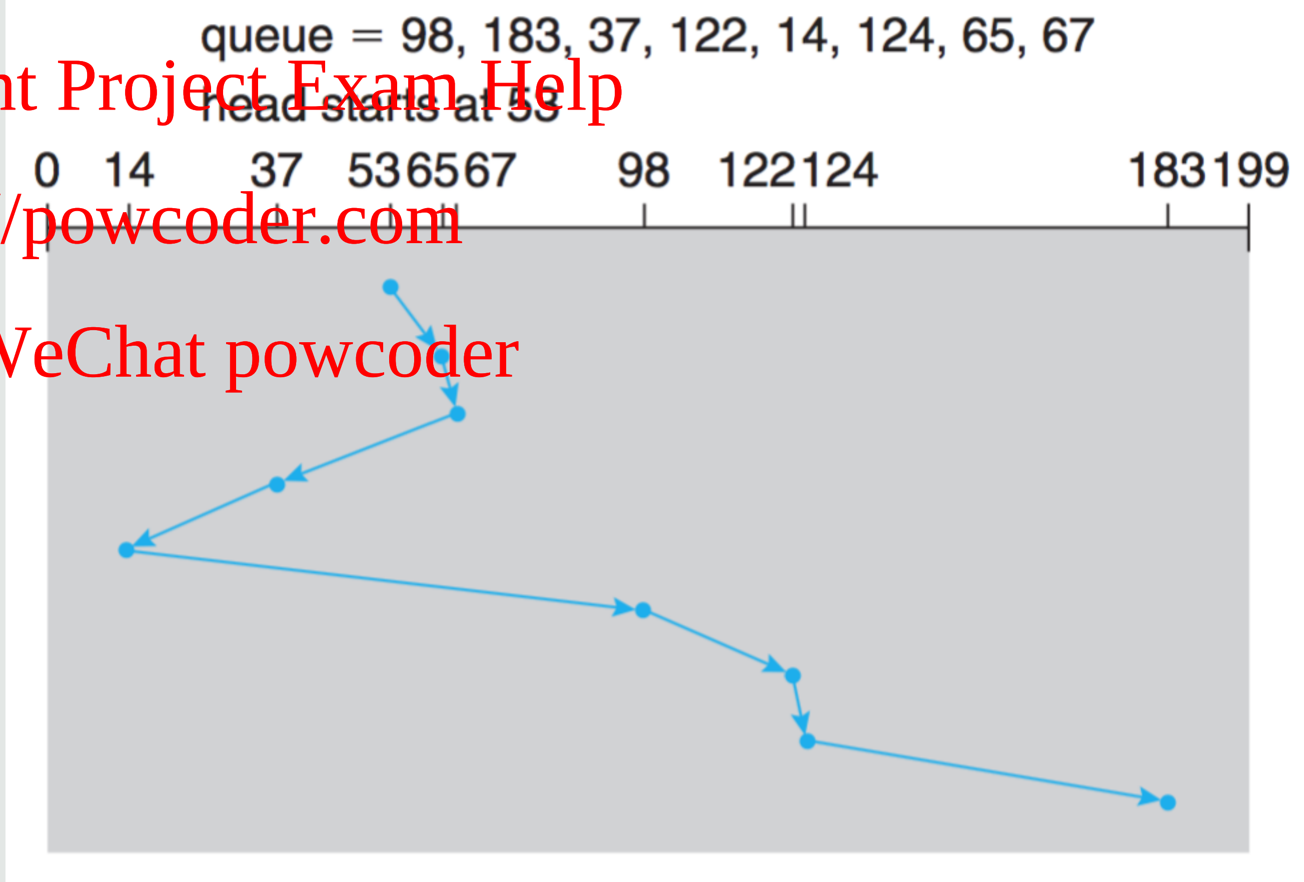
  Head moves 640 cylinders
  → minimise head movement



I/O Performance Optimisation

Example: Disk scheduling

- Shortest Seek Time First (SSTF)



Head moves 236 cylinders

I/O Performance Optimisation

SCAN (elevator) algorithm:

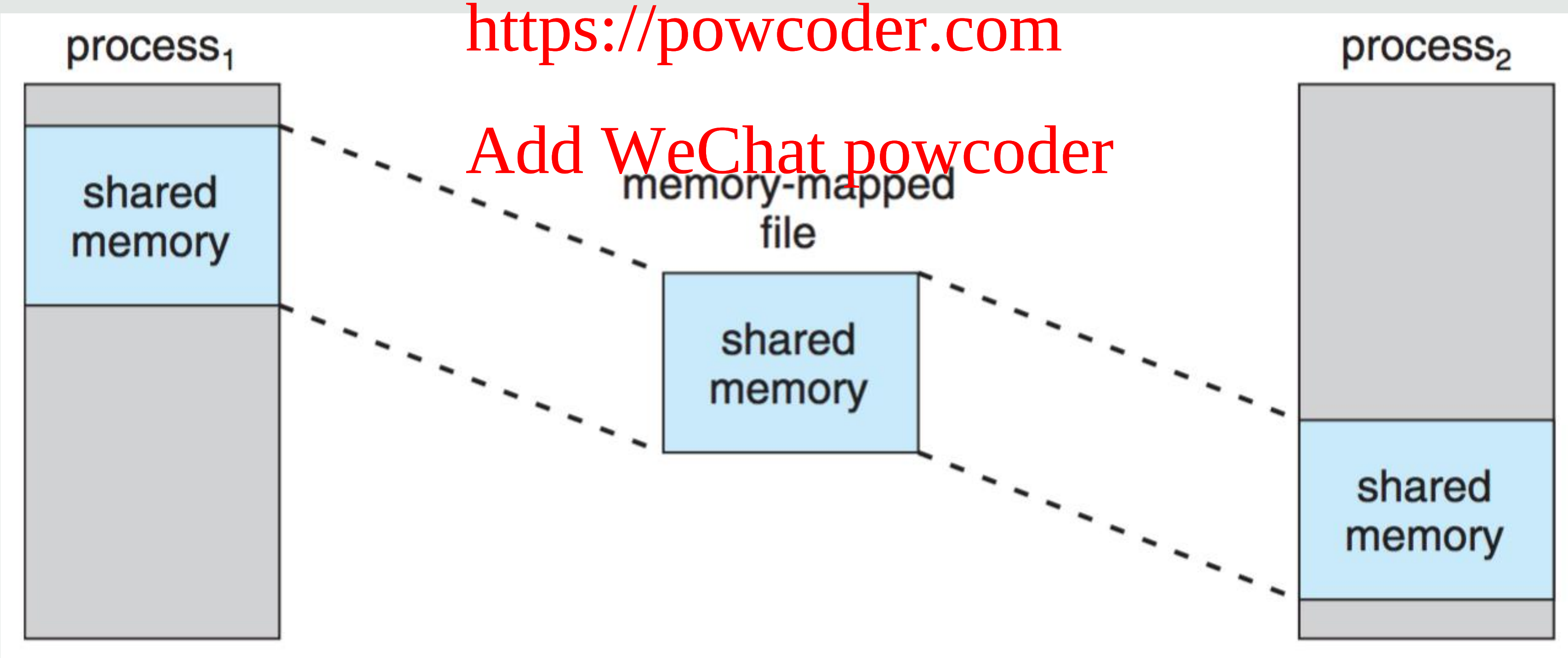
- Move head back and forth in one direction
  → moves head only 208 cylinders
- Variants: C-SCAN, LOOK, . . .

NOOP

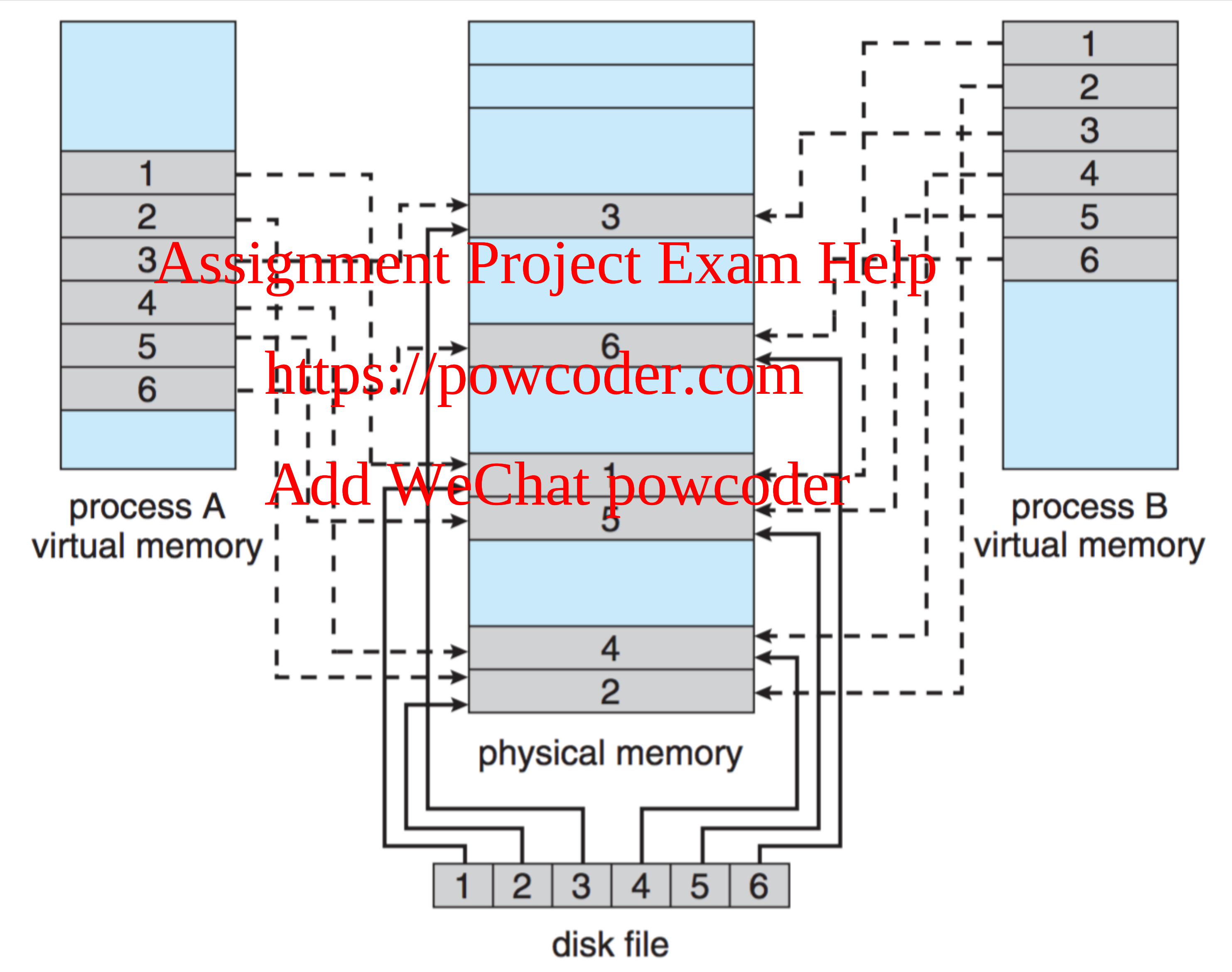
- FCFS, but looks for adjacent requests and merges them
- Low overhead
- Works well for SSD
- Used in Linux

Memory-Mapped Files

- Shared file access through shared memory
- Synchronisation: copy-on-write
- Use paging system



Memory-Mapped Files



Memory-Mapped I/O

Problem: Page replacement

- Cannot evict page during an I/O request → locking / pinning

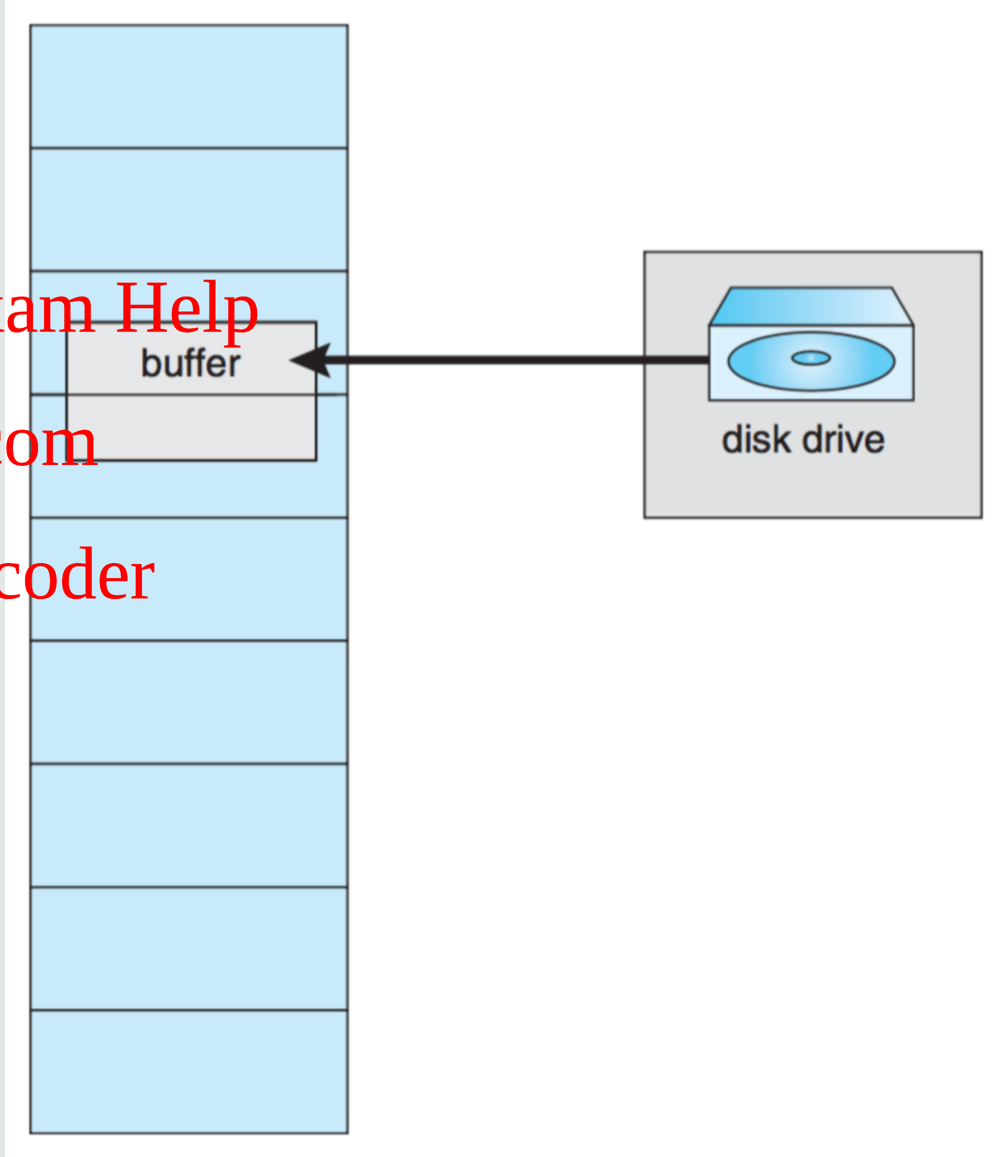
Page caching:

- Physical memory written by DMA or I/O controller:

  stale page → disable caching of page

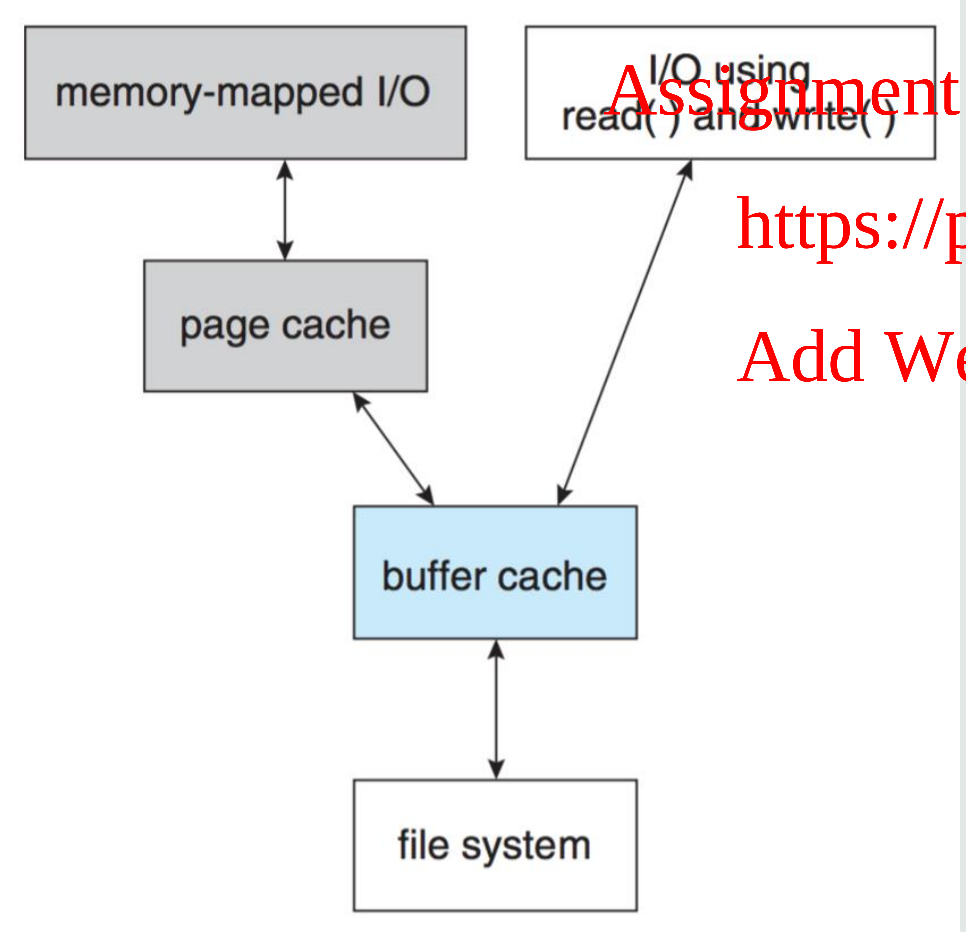
HW support

- e.g. by page attributes

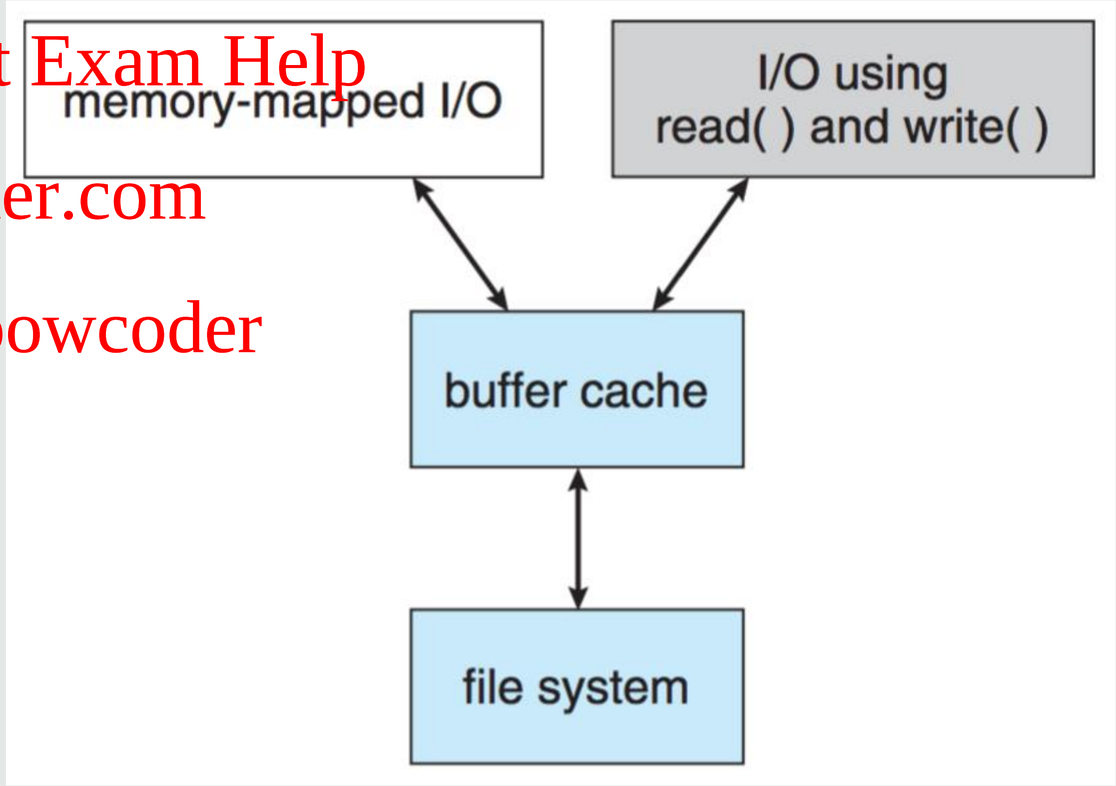


Direct I/O and Memory-Mapped I/O

Double buffering



Unified buffer cache



I/O System Design

Efficiency

- I/O is often bottleneck of the system, i.e. CPU has to wait, processes are blocked, . . .
- But: I/O devices types have different properties

  → require specific solutions, i.e. disk scheduling for hard disks vs SSD

Flexibility

- Many different devices with different properties, new devices must be easily supported, . . .
- But: uniform programming interface required, e.g. files

  → uniform interface and device-specific behaviour

I/O System Design

I/O system call interface

- Provides uniform API (e.g. POSIX system calls)
- Hardware-independent

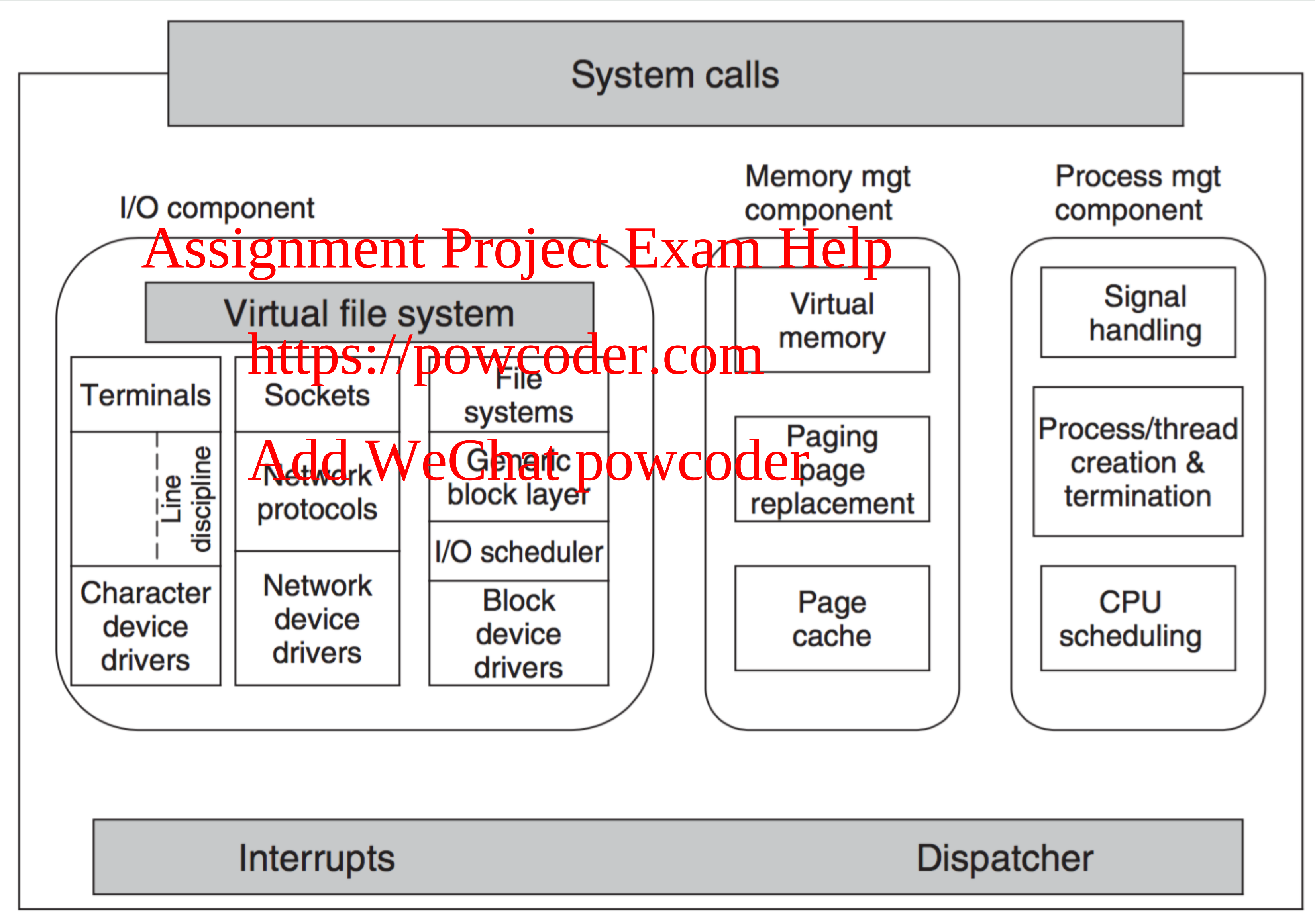
Scheduling and control

- I/O queues, buffering, caching, . . .
- (Mostly) hardware-independent

Device drivers

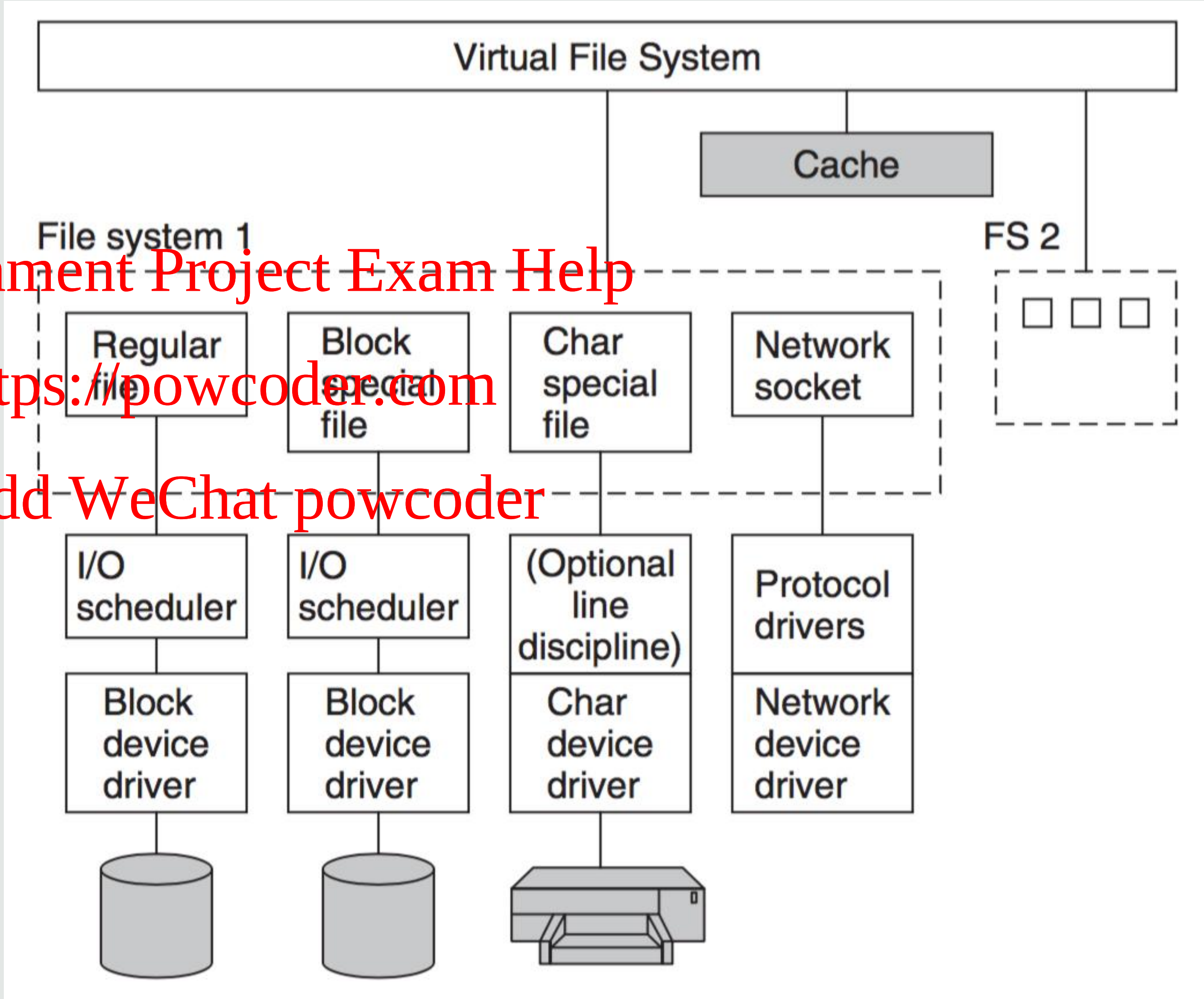
- Translate operations into device-specific commands
- (Mostly) hardware-dependent

I/O System in the Linux Kernel

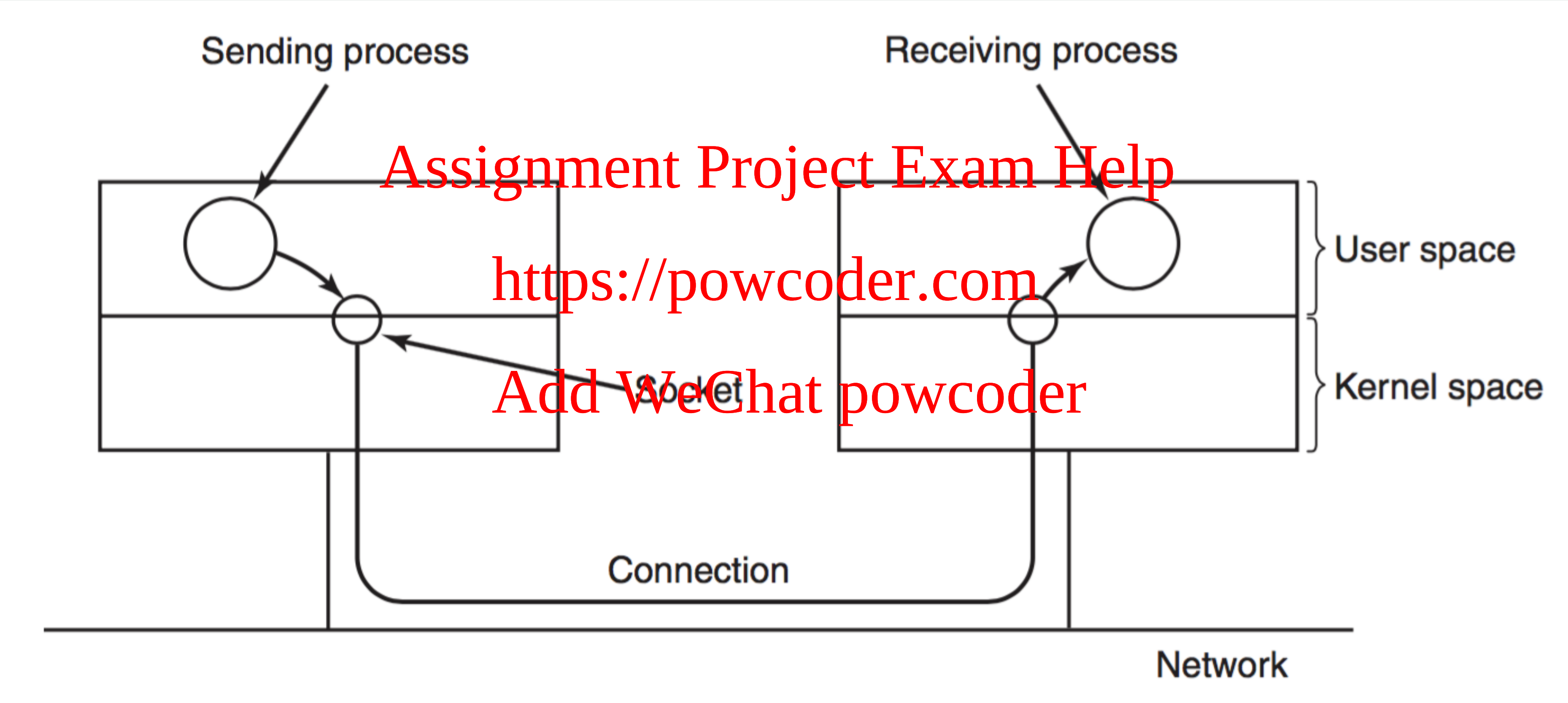


I/O System in the Linux Kernel

Uses loadable modules to install device drivers



Files vs Sockets



Summary

System call interface

- Uniform abstraction of I/O operations
- Blocking vs non-blocking calls

I/O System

- Layered structure:

  Hardware Abstraction Layer separating HW-dependent /-independent
- Device driver
- I/O controller

Interaction protocols

- Polling, interrupts, DMA

I/O Scheduling and Optimisations

- Reduce blocking times by reordering requests, e.g. disk scheduling
- Buffering, caching, memory-mapped I/O

Read

- Tanenbaum & Bos., Modern Operating Systems
  - Chapter 5
- Silberschatz et al., Operating System Concepts
  - Chapter 10 & 13

Next Lecture

- Introduction
- Operating System Architectures
- Processes
- Threads - Programming
- Process Scheduling - Evaluation
- Process Synchronisation
- Deadlocks
- Memory Management
- File Systems
- Input / Output
- **Security** and Virtualisation