ECE 1175
Embedded Systems Design

Lab 3 – Cache & Memory

ECE 1175 – Lab 3

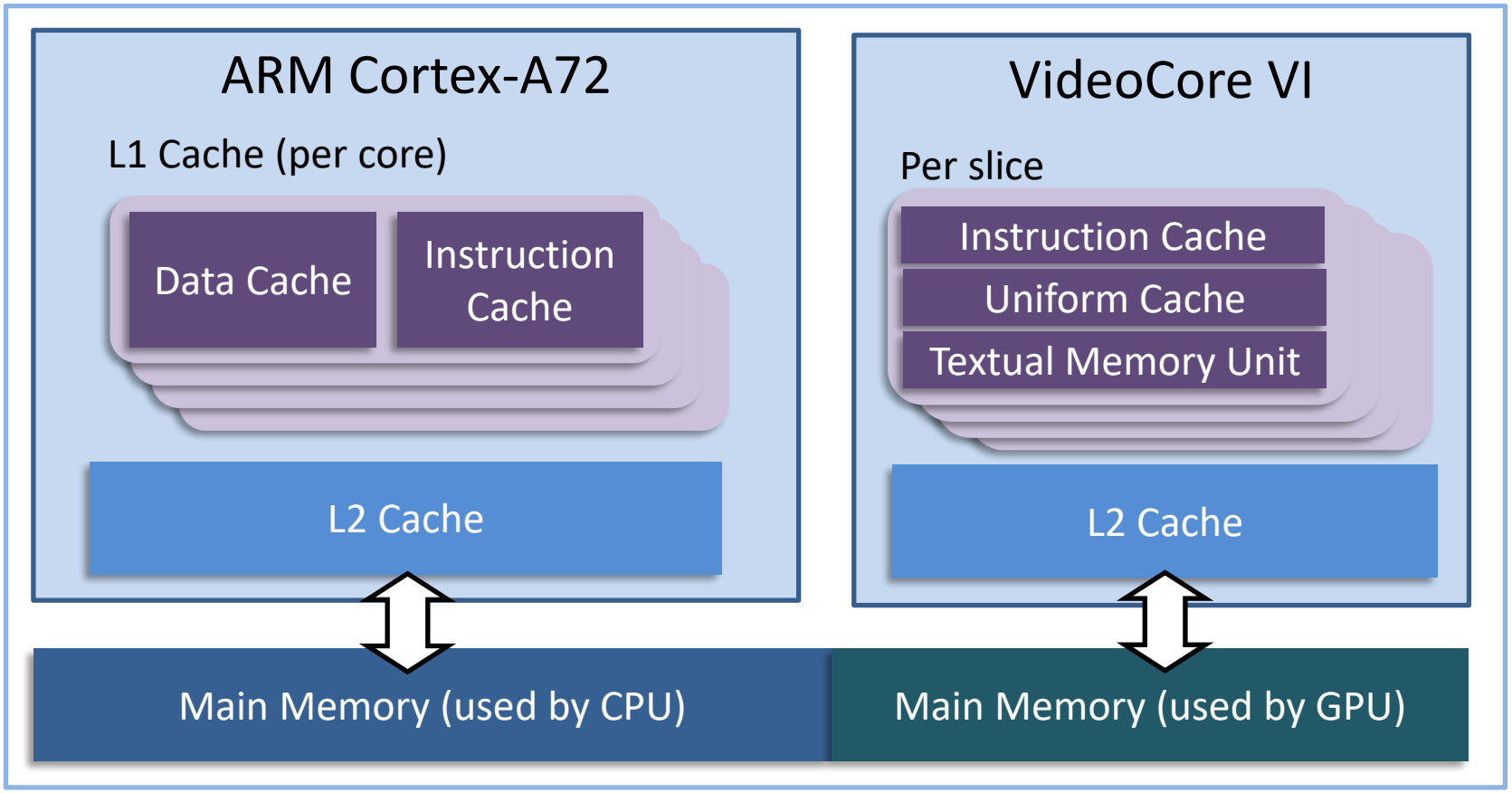
- **Monitor Cache Misses**
  - Cache basics
  - Performance analysis tool – perf
  - Lab task 1
- **Direct GPIO Access**
  - Virtual/physical memory basics
  - Raspberry Pi GPIO
  - Lab task 2

You need to use C/C++ to complete your lab work.

Cache Basics

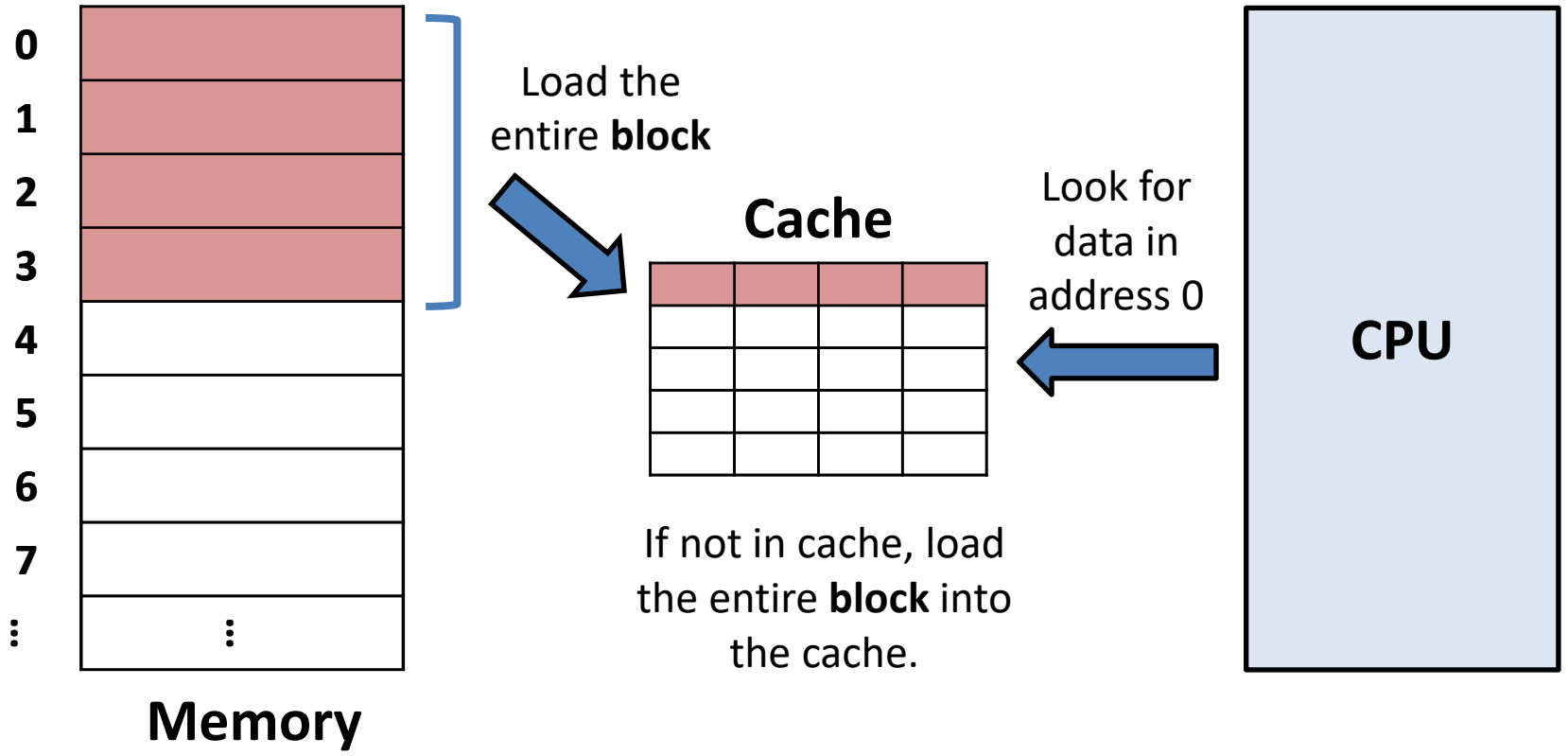
- **Cache: Fast but small memory close to the processor**

Caches on Raspberry Pi 4 Processor BCM2711



Cache Basics

- **How does cache work?**



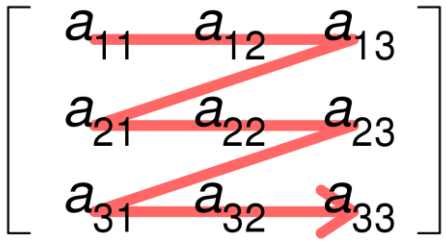
The block size depends on specific cache design.

Cache Basics

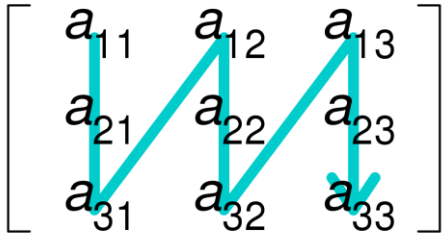
- **Impact on C programming**

  In C, multidimensional arrays are stored in row-major order in the memory. The way you access entries affects cache misses.

Row-major order



Column-major order



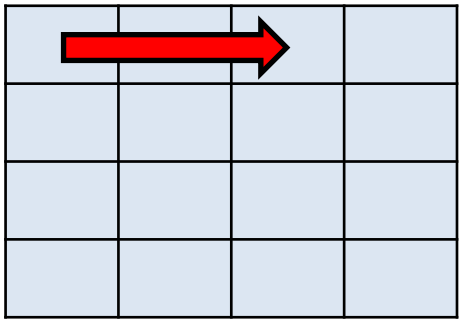
Address	Row-major	Column-major
0	a11	a11
1	a12	a21
2	a13	a31
3	a21	a12
4	a22	a22
5	a23	a32
6	a31	a13
⋮	⋮	⋮

Cache Basics

- **Impact on C programming**

Traverse a 2D array in row major order

Let's assume N is very large

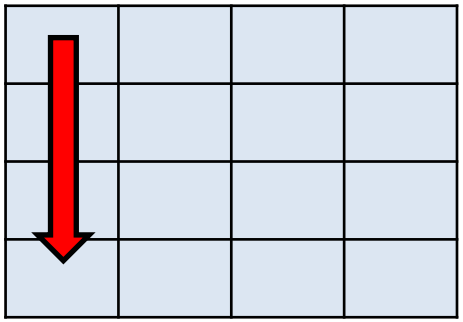


stride = 1

```
for (i = 0; i < N; i++) {
    for (j = 0; j < N; j++) {
        // Access a[i][j]
    }
}
```

Sequential access, a few compulsory cache misses

Traverse a 2D array in column major order



stride = N

```
for (i = 0; i < N; i++) {
    for (j = 0; j < N; j++) {
        // Access a[j][i]
    }
}
```

Not sequential access, 100% cache misses!

Performance Analysis Tool – perf

- **perf**
  - Provide access to performance counters
    - Hardware: CPU cycles, bus cycles, cache misses, etc.
    - Software: task clock, page faults, alignment faults, etc.
    - Use perf list to see available events
  - Offer a rich set of commands
    - Support multiple events
    - Repeated measurement
    - Processor-wide mode
    - Use perf --help to check info on a specific command

Performance Analysis Tool – perf

- **Use perf on Raspberry Pi OS**
  - To start:

1. Install perf

```
pi@raspberrypi:~ $ sudo apt-get install linux-perf
```

2. Open /usr/bin/perf (use vim, nano, etc.)

```
pi@raspberrypi:~/Desktop $ sudo vim /usr/bin/perf
```

3. Change exec “perf_$version” “$@” to exec “perf_4.9” “$@”

Your installed version

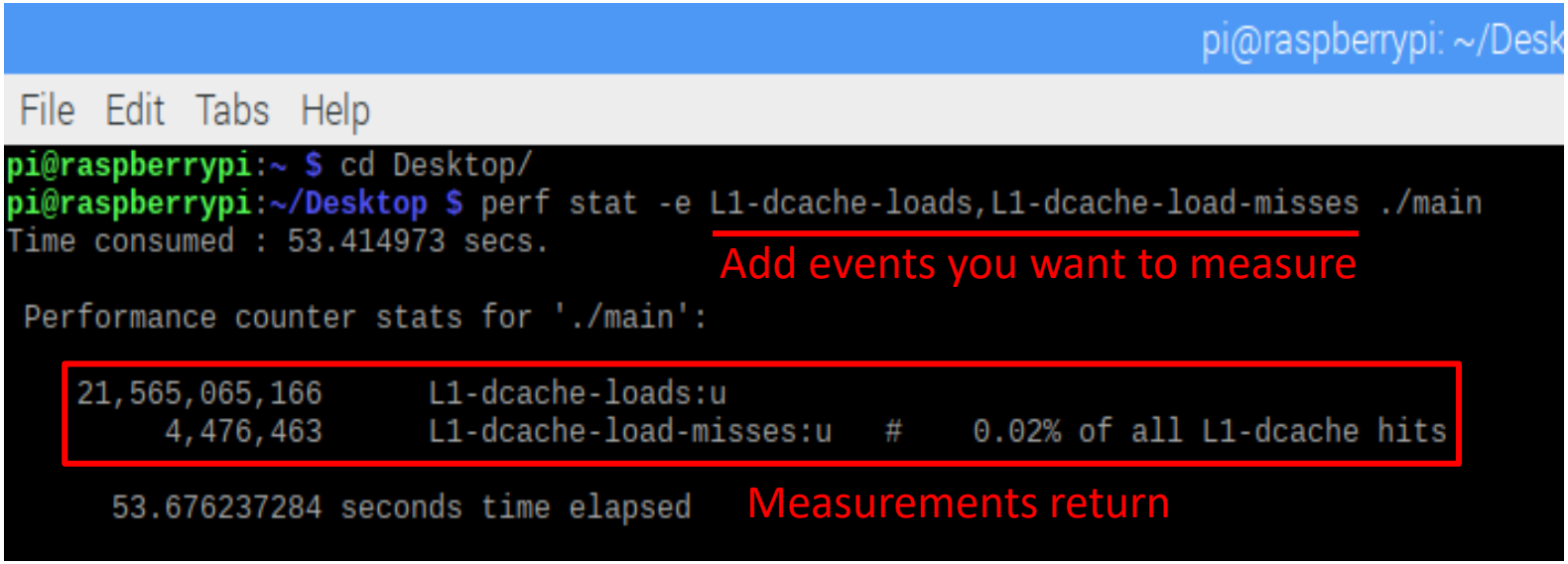
4. Check your installed perf version

```
pi@raspberrypi:~/Desktop $ perf --version
perf version 4.9.82
```

Performance Analysis Tool – perf

- **An example of analyzing your program via perf**

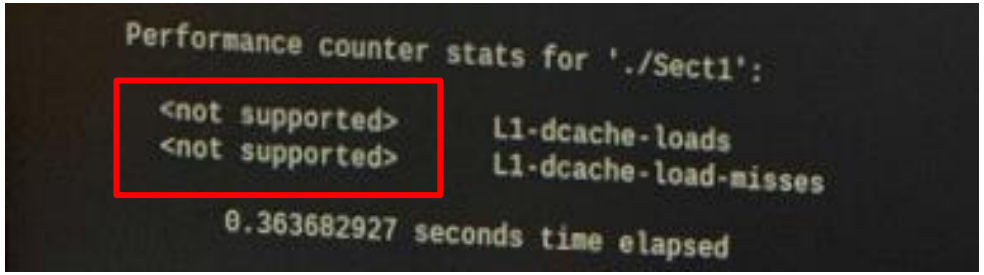
perf stat -e event1,event2,event3 [...] ./your_program



For more details: https://perf.wiki.kernel.org/index.php/Tutorial

Performance Analysis Tool – perf

- **If cannot get perf work**



Rollback the kernel version using: sudo rpi-update 8382ece

Then make sure you do not use sudo apt-get upgrade again

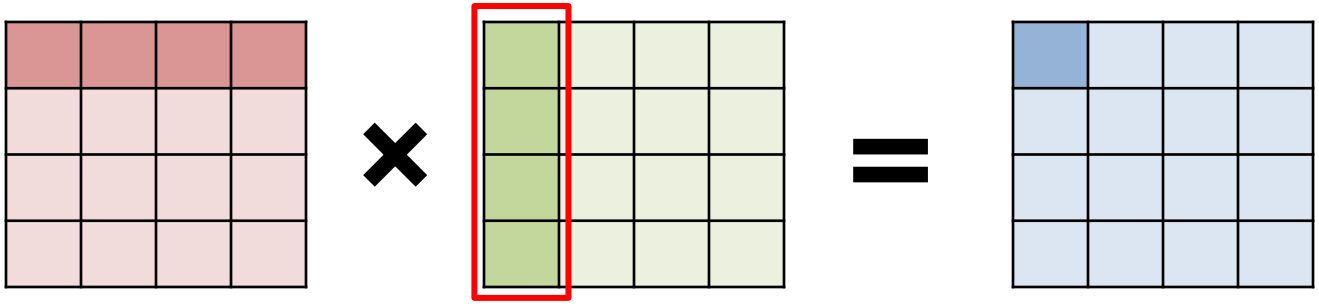
If it still does not work, enter NOOBS and reinstall the Raspbian system

Lab Task 1

- **Analyze your matrix multiplication program via perf**

$\mathbf{C} = \mathbf{AB}$ is defined by $c_{ij} = \sum_{k=0}^{N-1} a_{ik} b_{kj}$.

Not sequentially accessed in memory



Very high cache miss rate!

To reduce cache misses, you can try interchanging your loops. Use perf to measure L1 data cache misses.

Lab Task 1

- **Define a 2-D array (matrix):**

```
const int N = 256;
static float A[N][N] = {0};
```

N cannot be changed during runtime

or

```
int N = 256;
float **A = (float**)malloc(N * sizeof(float*));
for(int i = 0; i < N; i++) {
    A[i] = (float*)malloc(N * sizeof(float));
}
```

N can be changed during runtime

- **Assign random values:**

```
srand((unsigned)time(NULL));
for(int i=0; i<N; i++) {
    for(int j=0; j<N; j++) {
        A[i][j] = rand()/(float)RAND_MAX;
    }
}
```

- **Calculate the time consumed:**

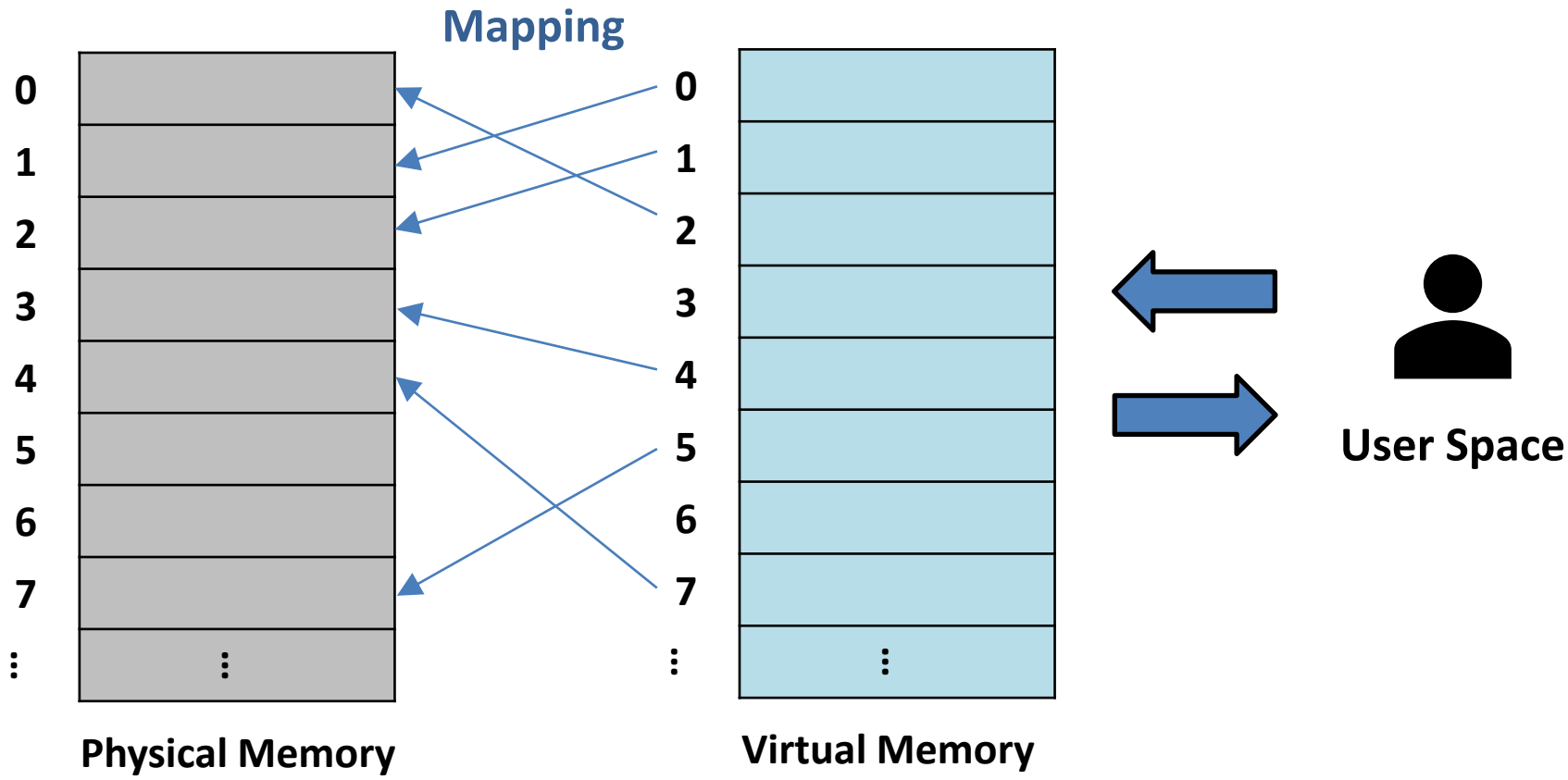
```
startTime = clock();
// Your code here
stopTime = clock();
cout << "Time consumed: " << (stopTime-startTime)/CLOCKS_PER_SEC << "secs" <<endl;
```

Your code of matrix multiplication should be here

Do not forget #include <time.h>

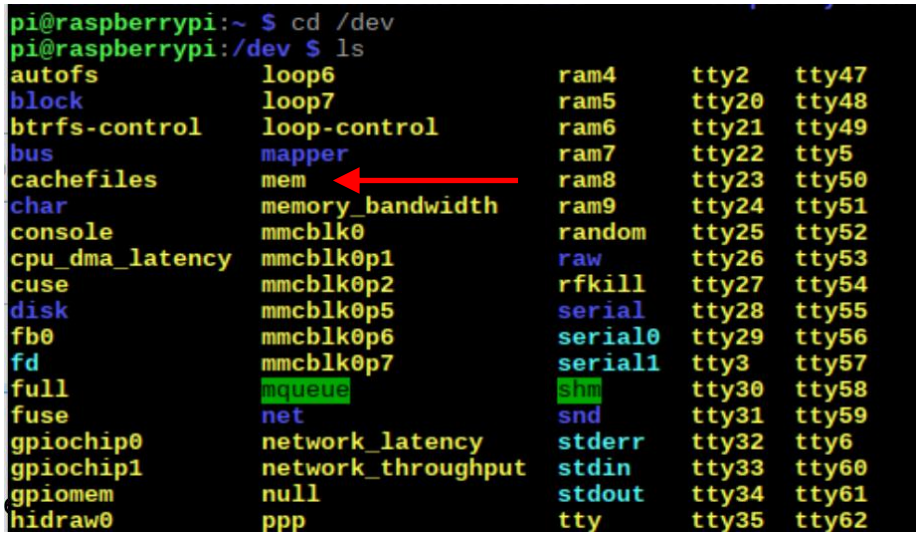
Virtual/Physical Memory Basics

- **In modern operating systems, physical memory cannot be directly accessed by users.**



Device file – dev/mem

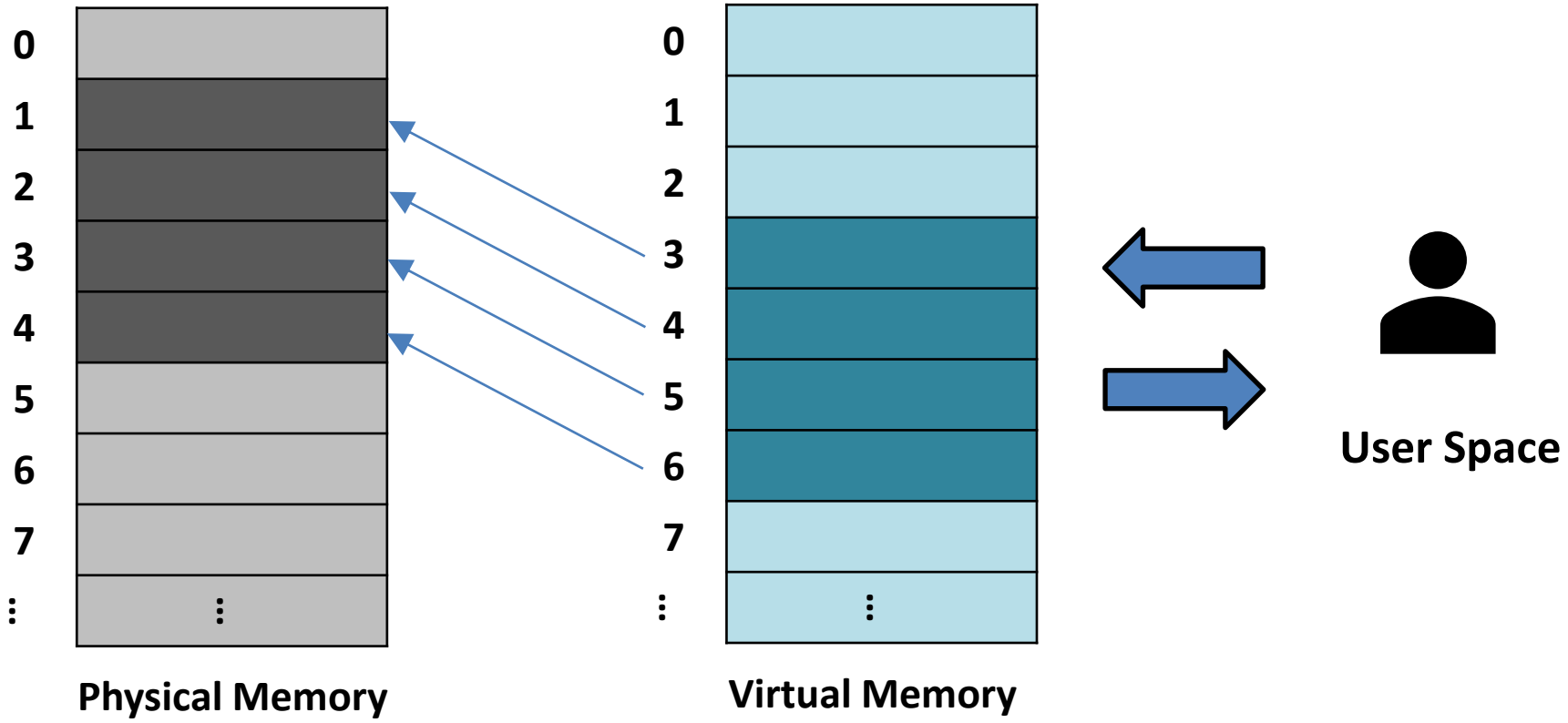
- **dev/mem**
  - An image of main memory of computer
  - Byte addresses in /dev/mem are interpreted as physical memory (actual RAM address, registers).



For more details: https://man7.org/linux/man-pages/man4/mem.4.html

Create mapping – mmap()

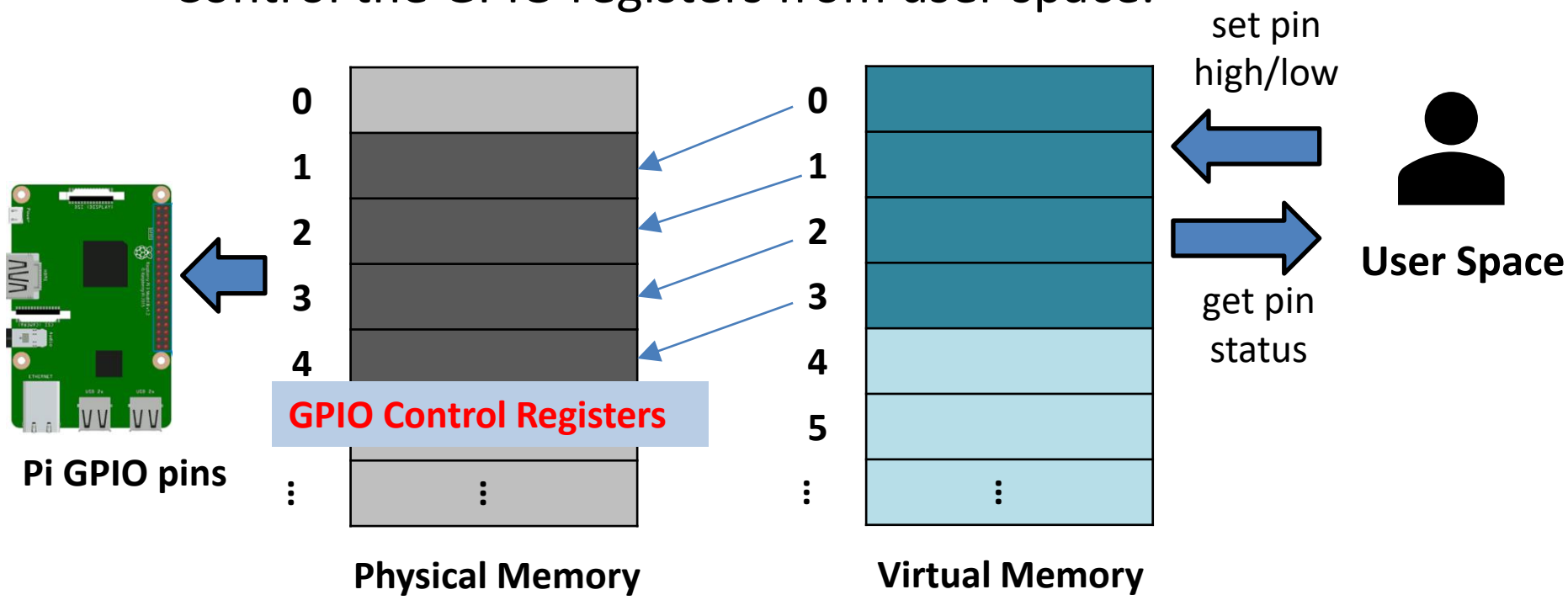
- **mmap() with dev/mem**
  - You can create a mapping from virtual to physical memory.



For more details: https://man7.org/linux/man-pages/man2/mmap.2.html

Lab Task 2

- **Direct GPIO manipulation on Raspberry Pi OS**
  - Find the physical address of GPIO registers in manual.
  - Use mmap() and dev/mem to create a mapping.
  - Control the GPIO registers from user space.

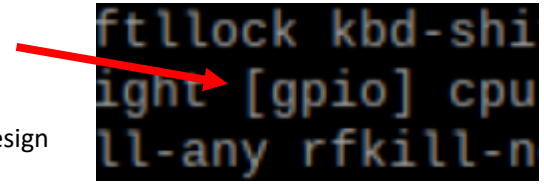


Lab Task 2

You can refer to the example here https://elinux.org/RPi_GPIO_Code_Samples.

- **For check-off**
  - Use GPIO 42 (internally connected to the onboard green LED) to generate a blinking pattern
  - The green LED is used for indicating SD card activity by default. To used it for GPIO output:
    - Switch to root user:
```
pi@raspberrypi:~ $ sudo su -
```
    - Modify the LED settings:
```
root@raspberrypi:~# echo gpio > /sys/class/leds/led0/trigger
```
    - Have a check:
```
root@raspberrypi:~# cat /sys/class/leds/led0/trigger
```

GPIO mode is selected



Lab Task 2

You can refer to the example here https://elinux.org/RPi_GPIO_Code_Samples.

- **Tips**
  - Refer to Chapter 5 in the manual:
    - https://datasheets.raspberrypi.org/bcm2711/bcm2711-peripherals.pdf
    - The registers you will be using are: GPFSEL4, GPSET1, and GPCLR1
    - If you want to use GPIO port other than 42, find corresponding registers
  - If you are using the C sample code:
    - Change GPIO_BASE to 0xFE200000
    - Change GPIO_SET to *(gpio+8)
    - Change GPIO_CLR to *(gpio+11)

Thank you!