# ОСНОВЫ COMPUTER SCIENCE

Operation systems

Balun.Courses

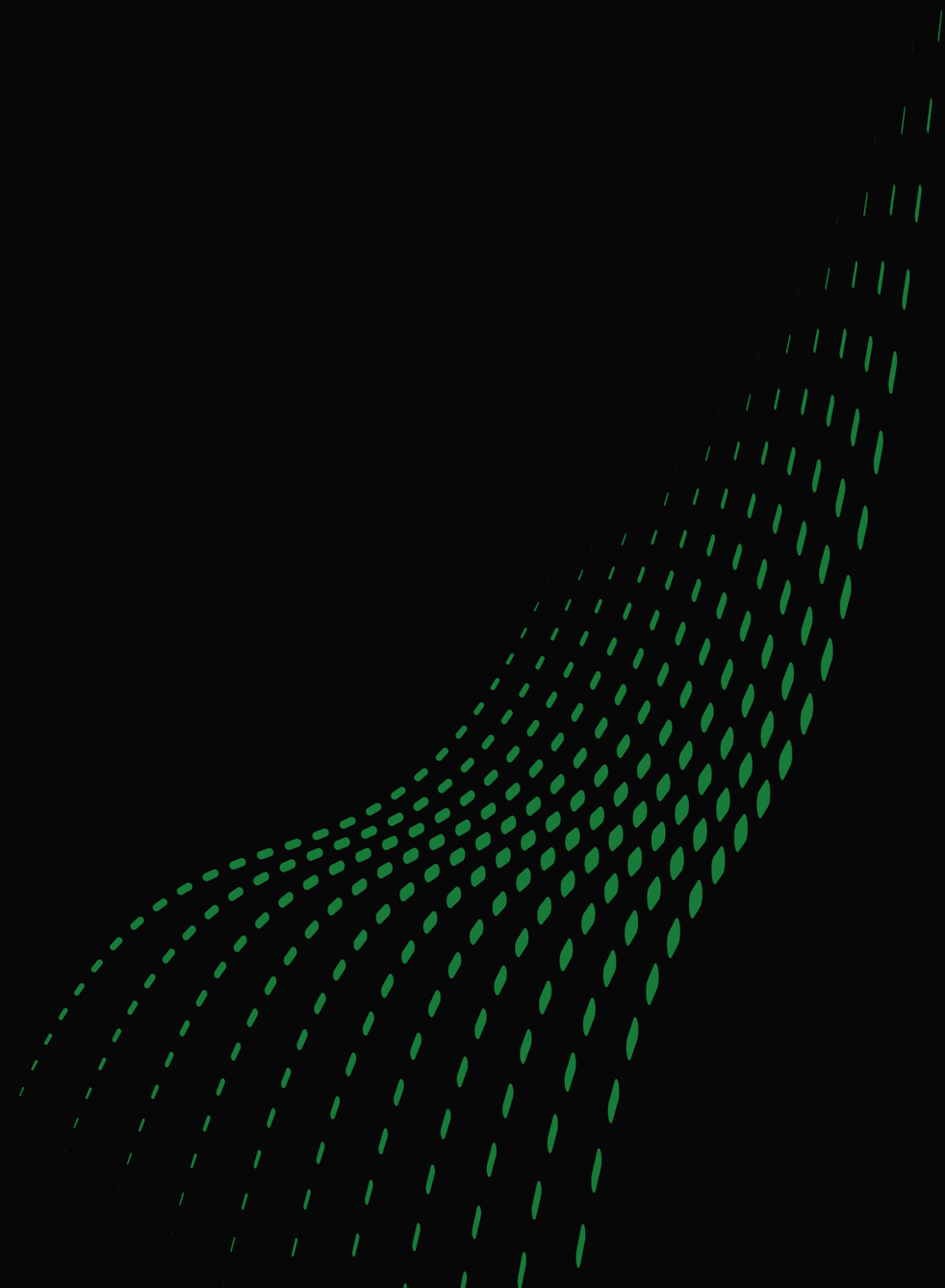
Balun.Courses

# ЗАПИСЬ

Balun.Courses

# ПРАВИЛА ЗАНЯТИЯ

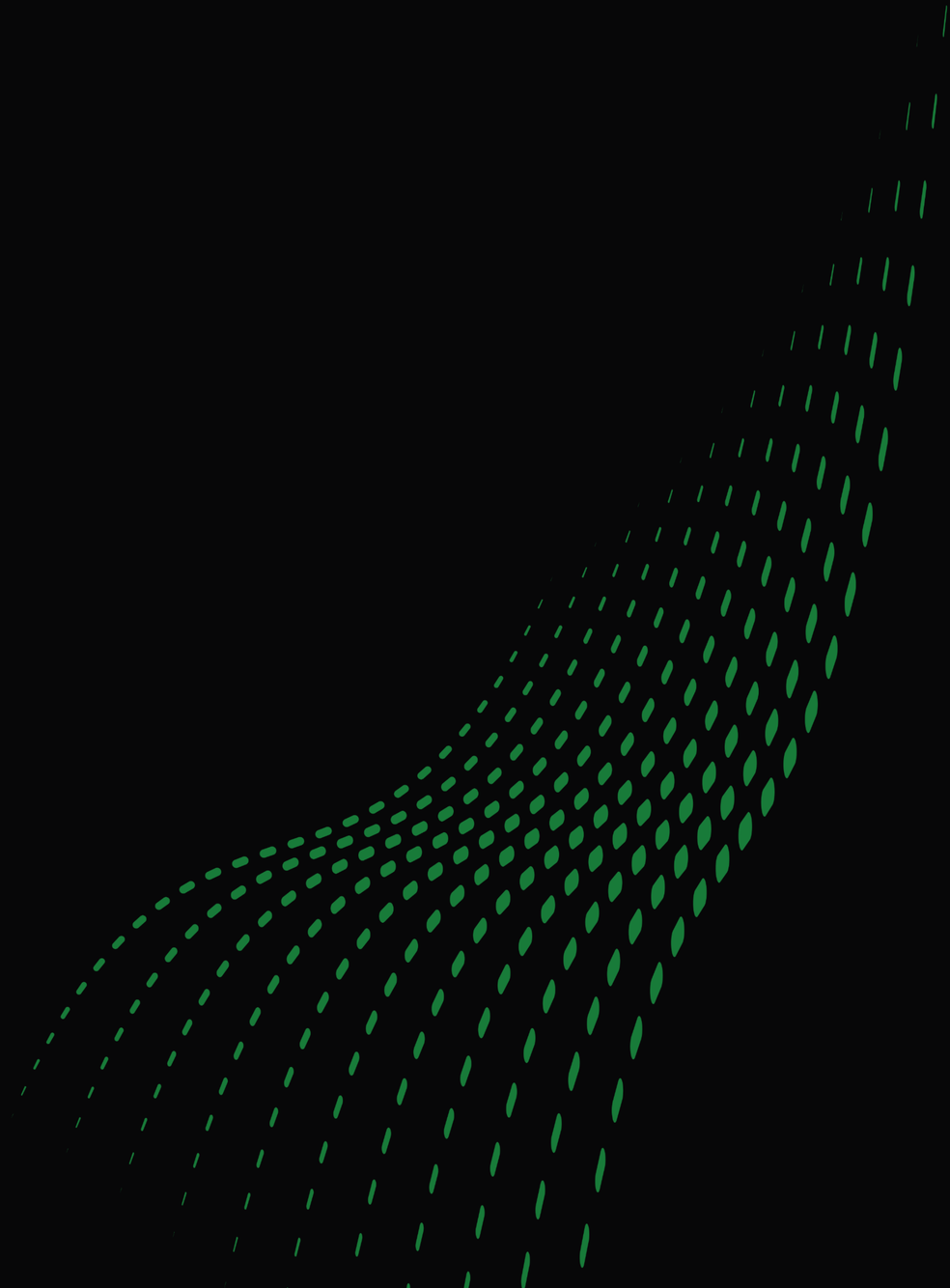
1. Вопросы можно и нужно задавать в любое время, если что-то непонятно
2. Лучше всего задавать вопросы голосом



Balun.Courses

# ПЛАН ЛЕКЦИИ

1. Устройство файловых систем
2. Современные возможности взаимодействия с OS, eBPF
3. OS Virtual Memory layout
4. Механизмы управления памятью
5. Lazy memory allocation
6. Memory allocation, buddy allocator, пример в xv6
7. Copy-on-write optimization
8. Demand paging, swapping
9. Memory mapped files, shared memory
10. Системные вызовы clone, fork, exec
11. Pipe
12. os/exec, обзор домашнего задания



Balun.Courses

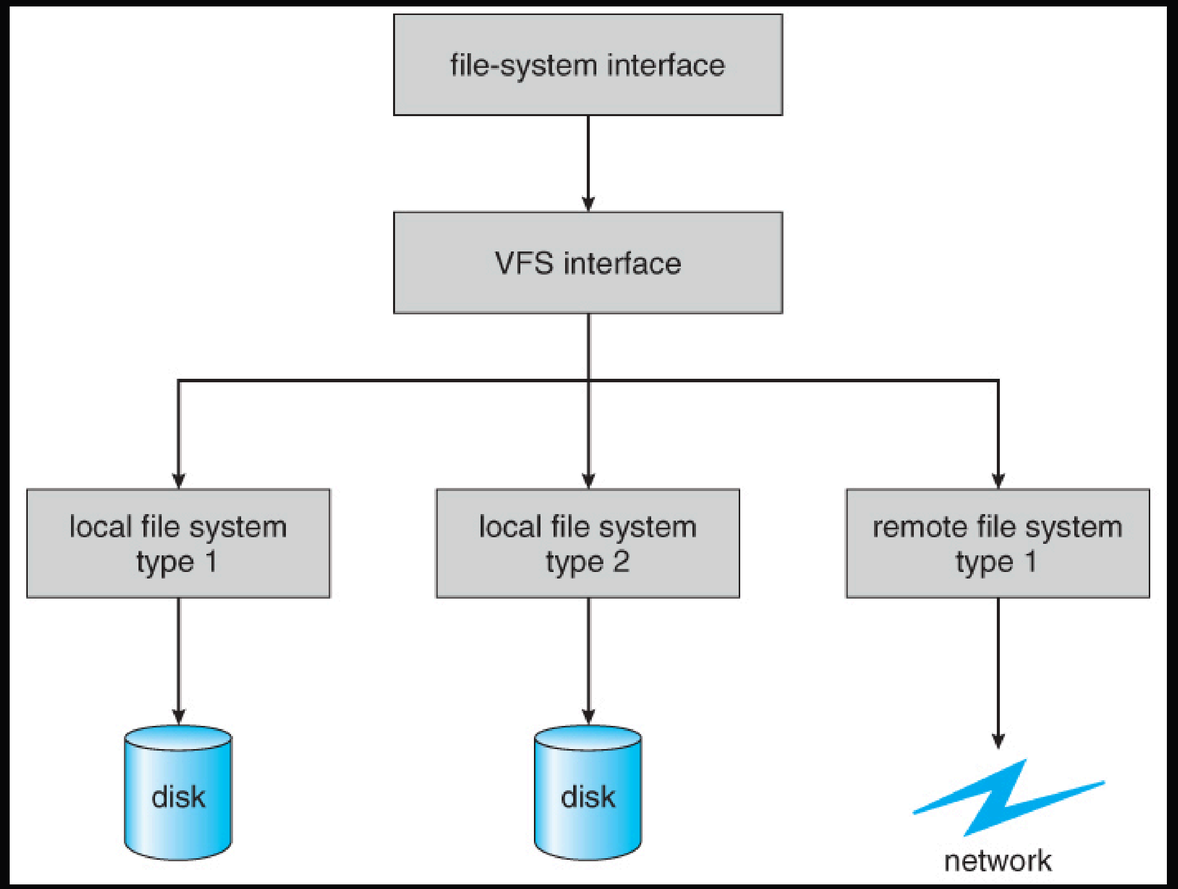
# УСТРОЙСТВО ФАЙЛОВЫХ СИСТЕМ

```go
func main() {
    wd, err := os.Getwd()

    if err != nil {
        panic(err)
    }

    f, err := os.Create(filepath.Join(wd, "file.txt"))
```

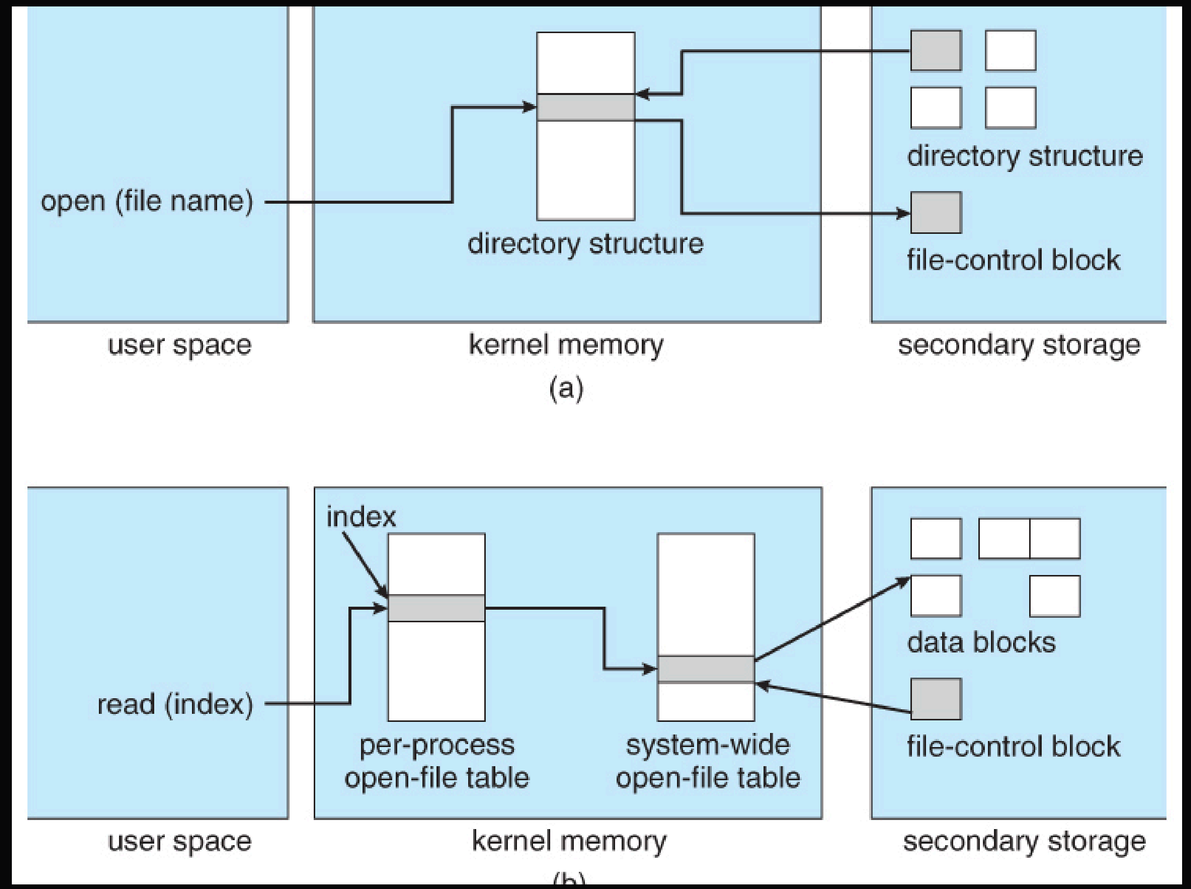
Что происходит в этом случае?

Balun.Courses

# УСТРОЙСТВО ФАЙЛОВЫХ СИСТЕМ



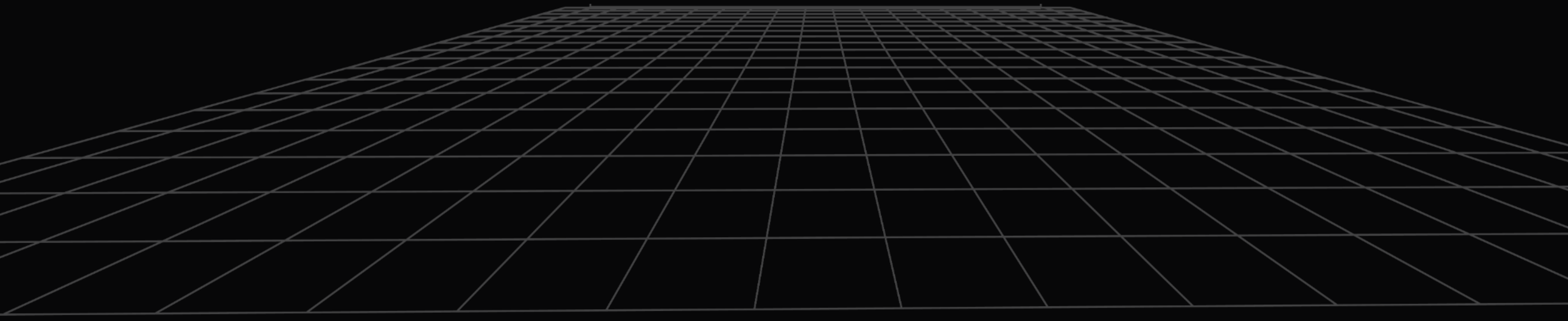

Virtual file system




(a) - открытие файла, (b) - чтение файла

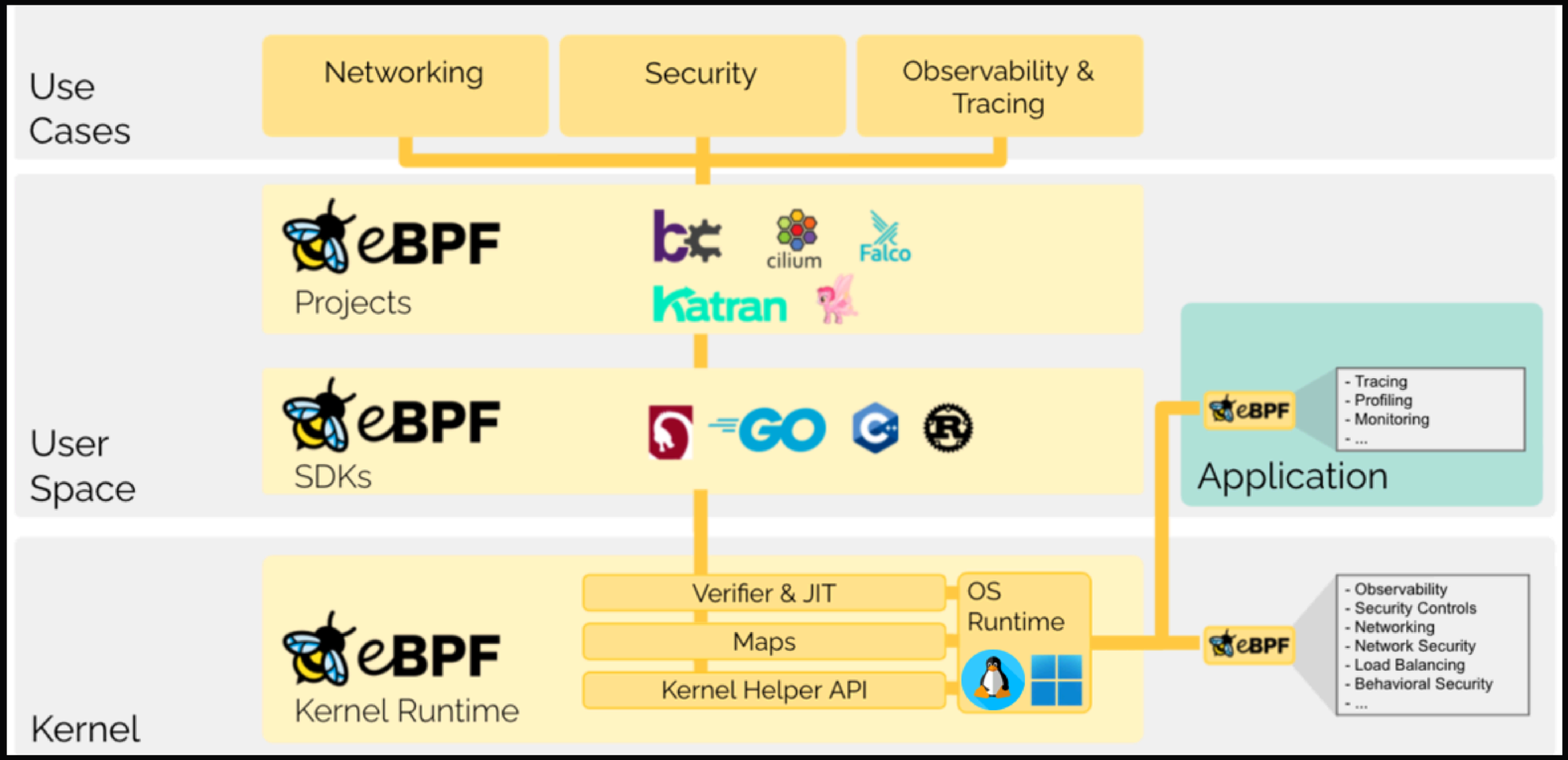
# DEMO

remote file system on linux



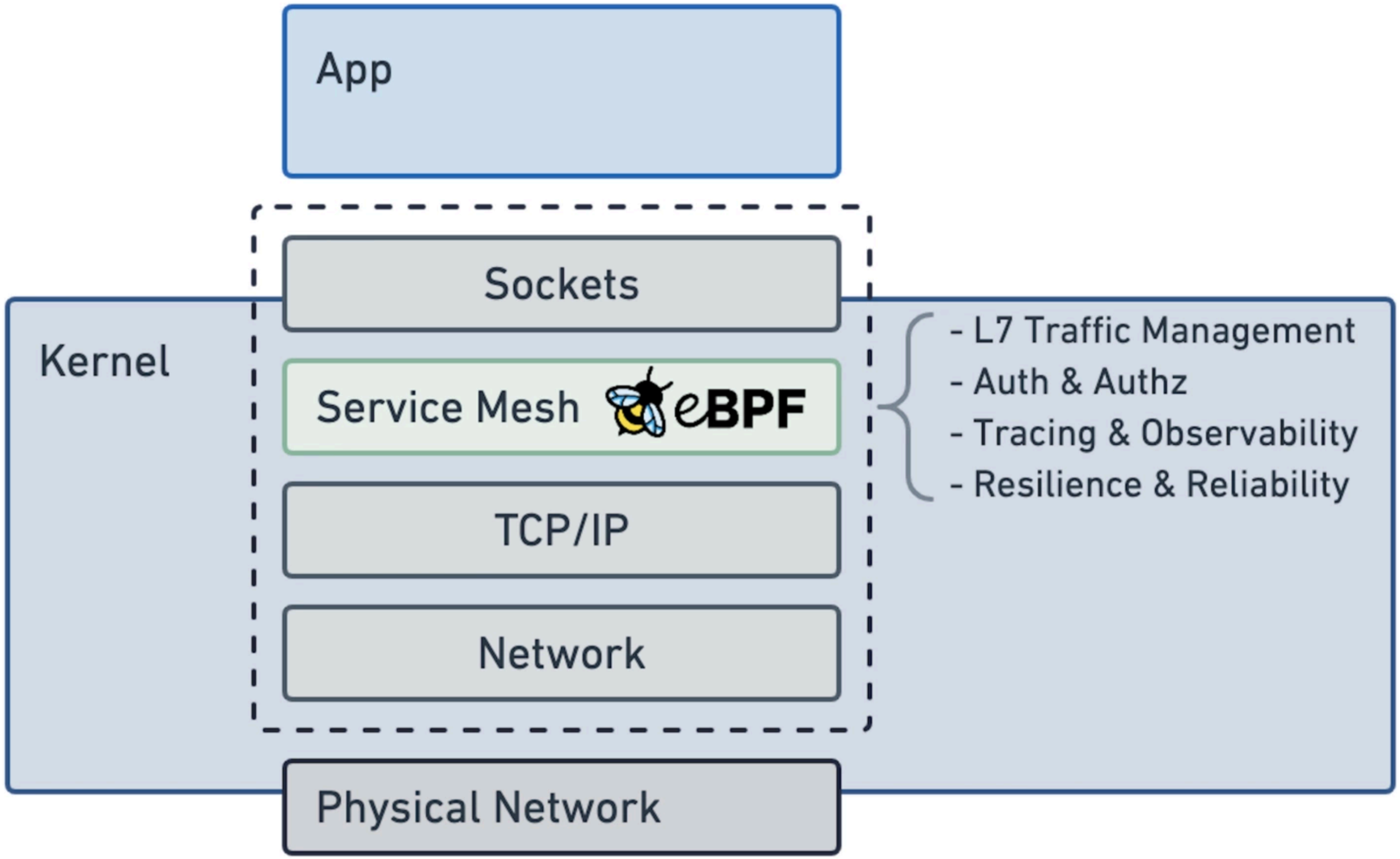
Balun.Courses

# EBPF



Balun.Courses

# EBPF



Balun.Courses

# VM LAYOUT AARCH64 LINUX

AArch64 Linux memory layout with 4KB pages + 4 levels (48-bit):

| Start | End | Size | Use |
|---|---|---|---|
| 0000000000000000 | 0000ffffffffffff | 256TB | user |
| ffff000000000000 | ffff7fffffffffff | 128TB | kernel logical memory map |
| [ffff600000000000 | ffff7fffffffffff] | 32TB | [kasan shadow region] |
| ffff800000000000 | ffff80007fffffff | 2GB | modules |
| ffff800080000000 | fffffbffefffffff | 124TB | vmalloc |
| fffffbfff0000000 | fffffbfffdffffff | 224MB | fixed mappings (top down) |
| fffffbfffe000000 | fffffbfffe7fffff | 8MB | [guard region] |
| fffffbfffe800000 | fffffbffff7fffff | 16MB | PCI I/O space |
| fffffbffff800000 | fffffbffffffffff | 8MB | [guard region] |
| fffffc0000000000 | fffffdffffffffff | 2TB | vmemmap |
| fffffe0000000000 | ffffffffffffffff | 2TB | [guard region] |

Balun.Courses

# МЕХАНИЗМЫ УПРАВЛЕНИЯ ПАМЯТЬЮ

```go
func main() {
    arr := make([]int64, 268_435_456_000) // 1000 GB

    for i := 0; i < 1000; i++ {
        fmt.Println(arr[i])
    }
}
```

Почему программа завершается успешно?

Balun.Courses

# МЕХАНИЗМЫ УПРАВЛЕНИЯ ПАМЯТЬЮ

```go
func main() {
    slices := make([][]int64, 100_000)

    for i := range 100_000 {
        fmt.Println(i)
        // 1000 GB
        slices = append(slices, make([]int64, 268_435_456_000))
    }

    fmt.Println(len(slices))
}
```

```
64
runtime: out of memory: cannot allocate 2147483648000-byte block (137439020023808 in use)
fatal error: out of memory
```

128_000 GB in use

| | |
|---|---|
| Memory | 36 GB |
| Startup disk | Macintosh HD |

Balun.Courses

# LAZY MEMORY ALLOCATION

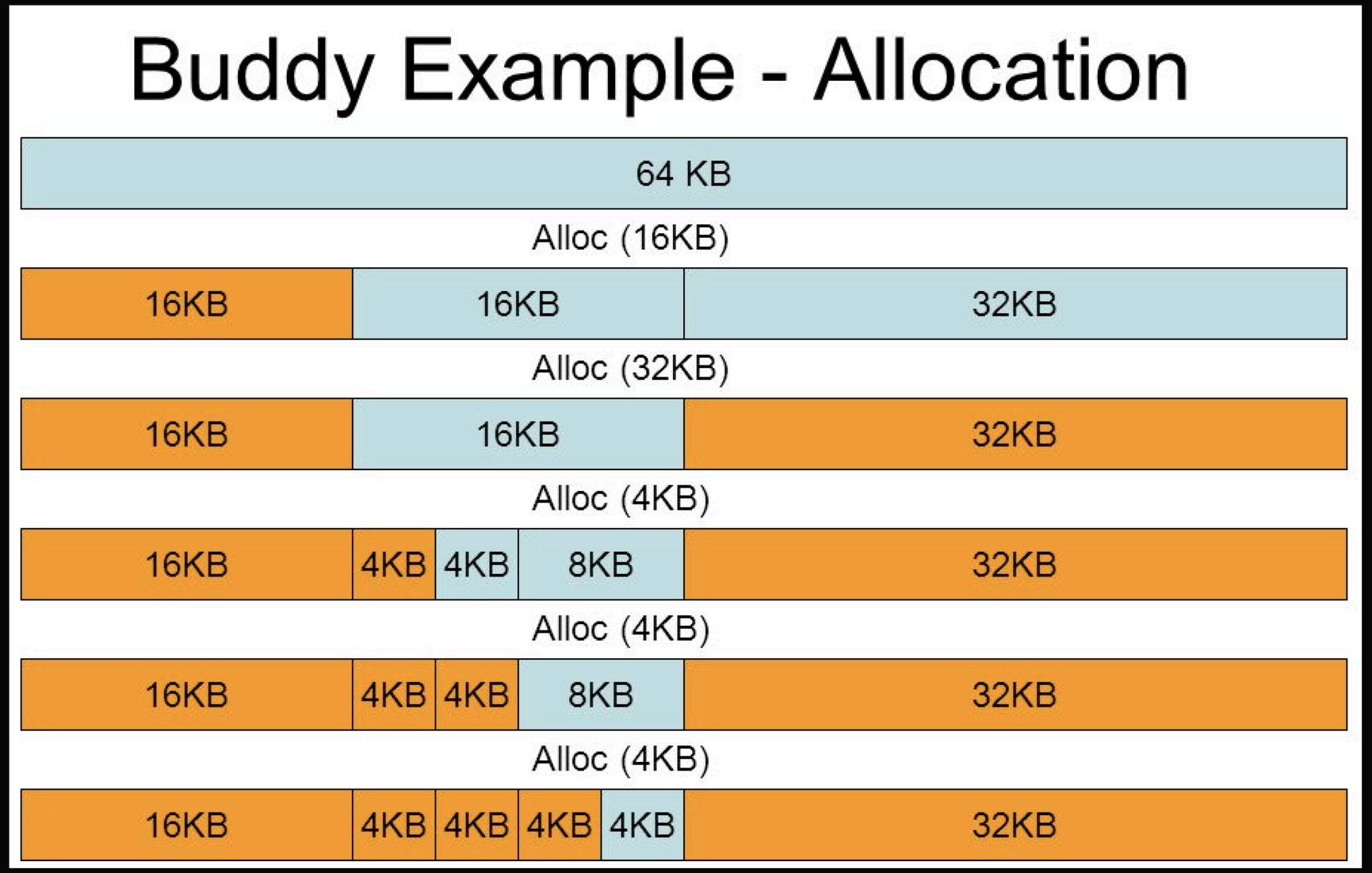
```go
func main() {
    slices := make([][]int64, 100_000)

    for i := range 100_000 {
        fmt.Println(i)
        // 1000 GB
        slices = append(slices, make([]int64, 268_435_456_000))
    }

    fmt.Println(len(slices))
}
```

Операционная система может лениво выделять память. При запросе памяти в соответствующие page tables может быть установлен флаг ленивой аллокации. При обращении к этой памяти произойдет прерывание, после чего операционная система выделит память

В чём плюсы и минусы ленивой аллокации?

Balun.Courses

# MEMORY ALLOCATION

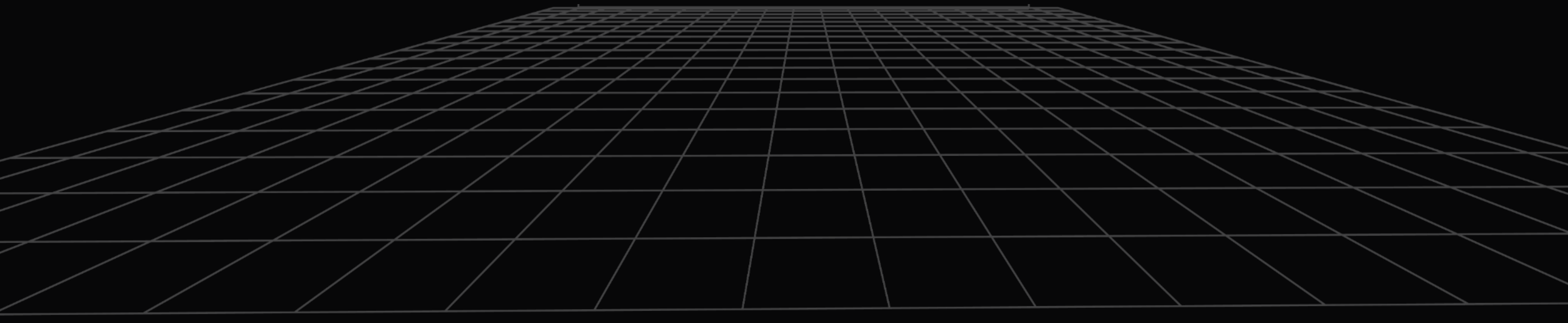



Что такое фрагментация?

Какие могут быть проблемы при таком подходе?

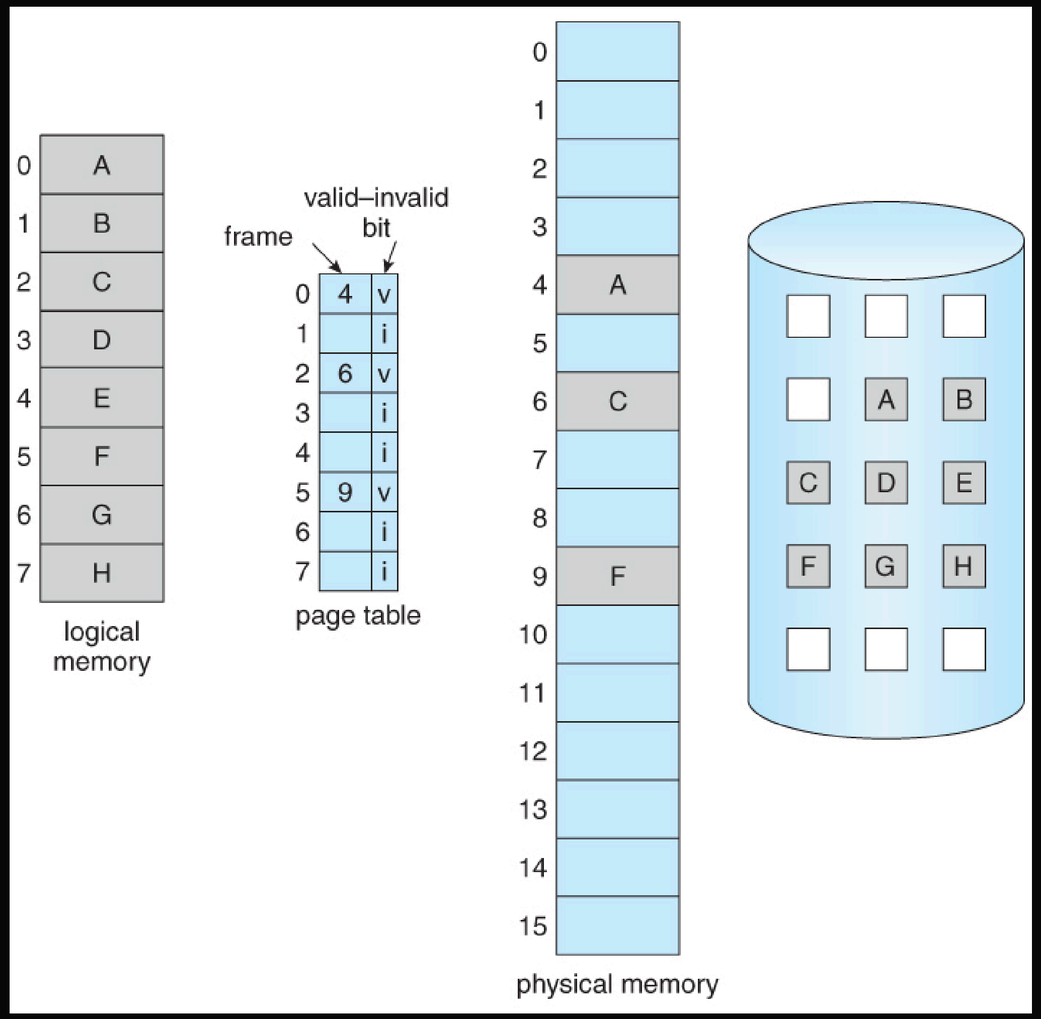
# DEMO

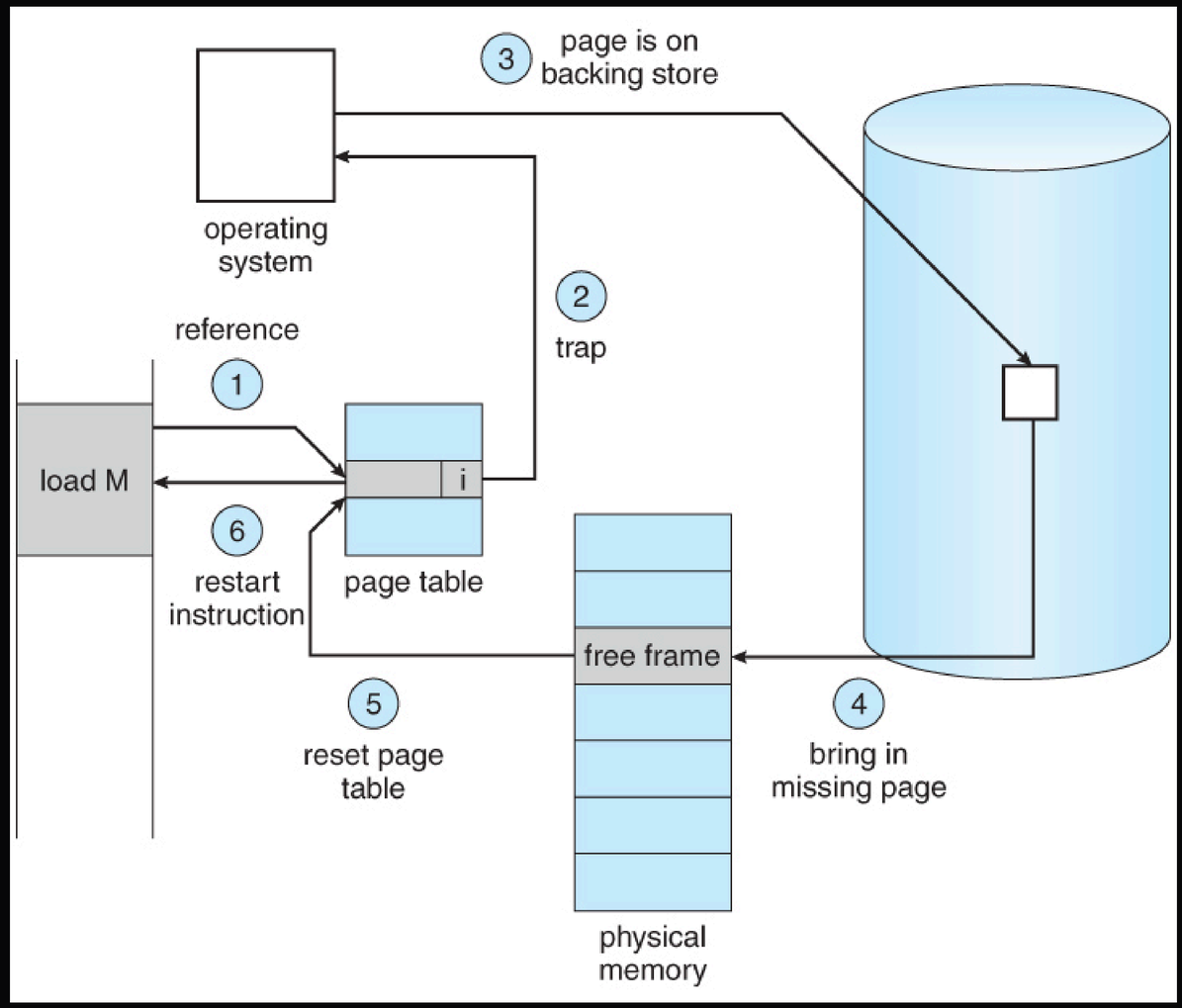
xv6 buddy allocator



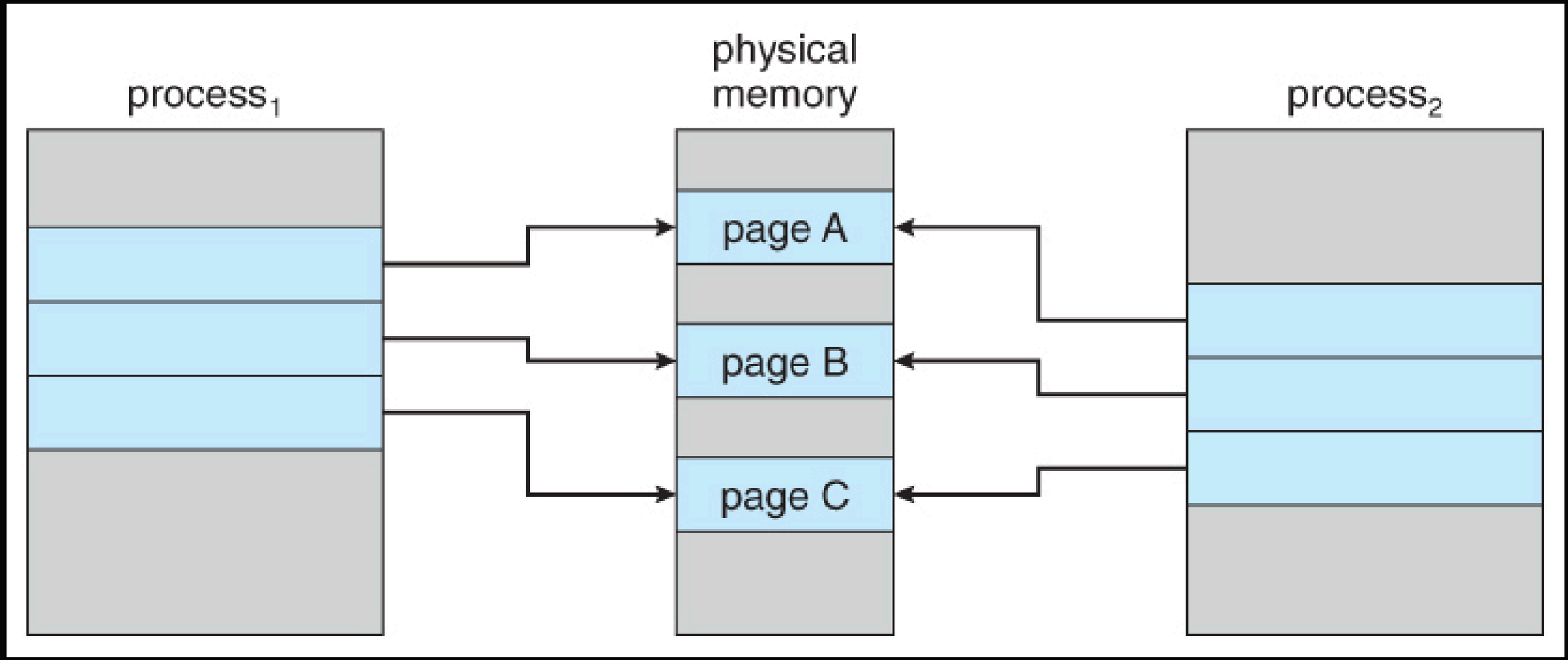
Balun.Courses

# DEMAND PAGING, SWAPPING

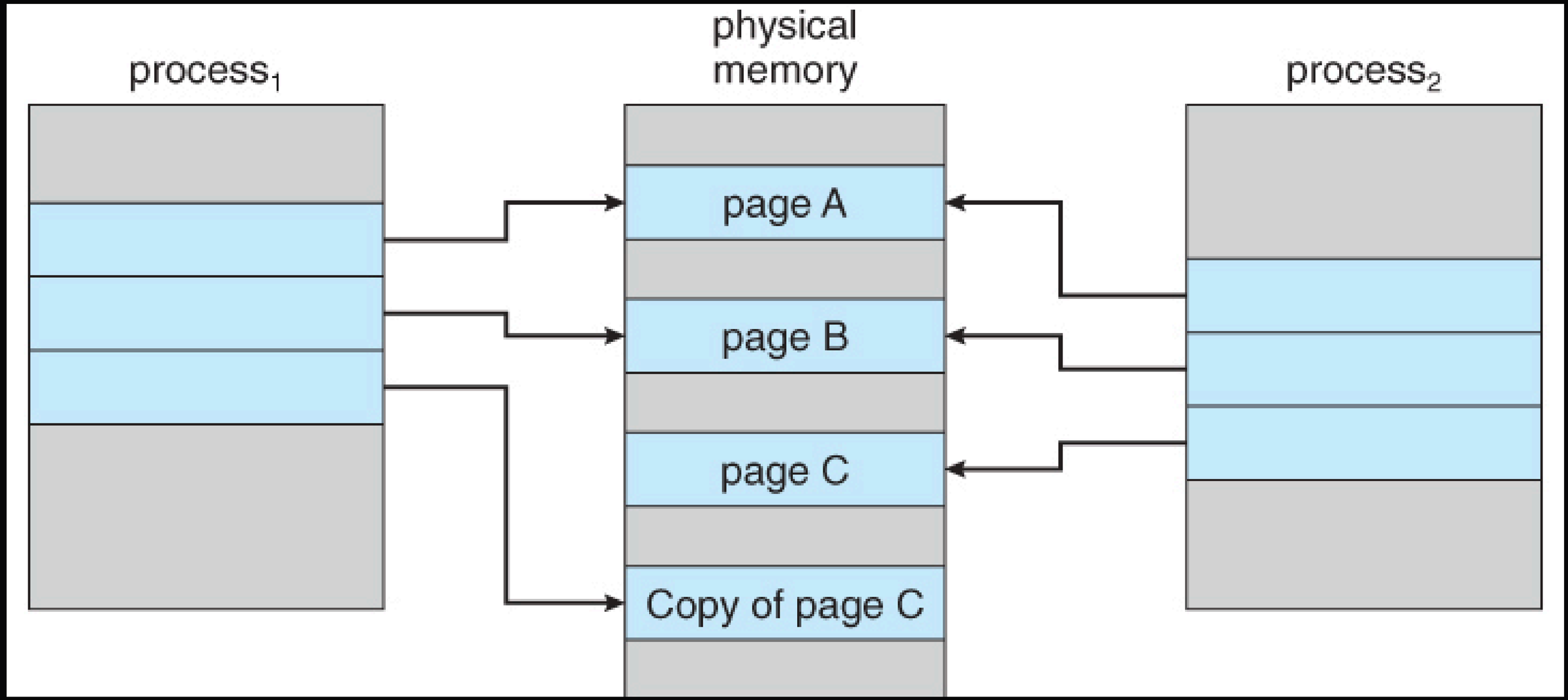


# COPY-ON-WRITE OPTIMIZATION

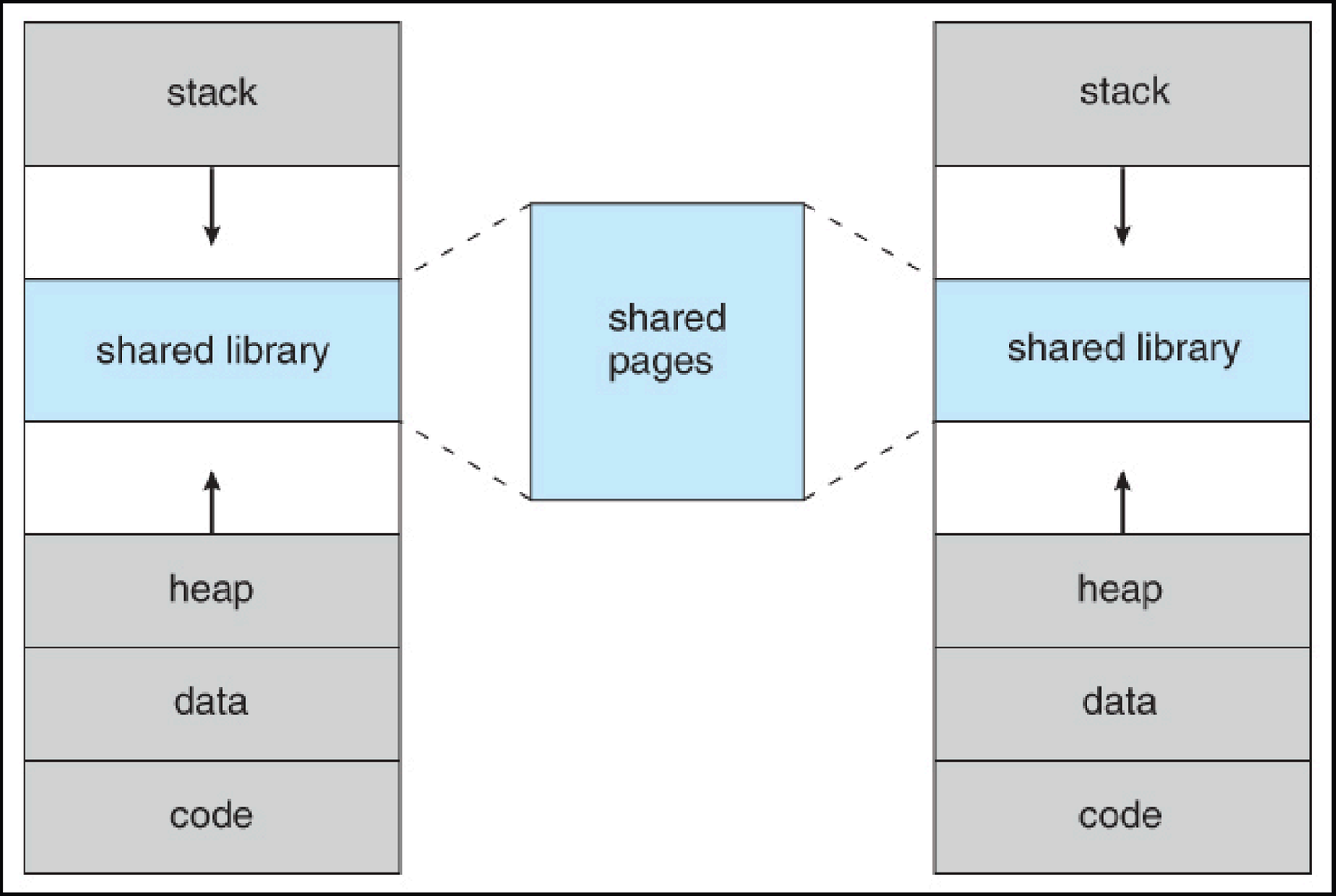


# COPY-ON-WRITE OPTIMIZATION



Balun.Courses

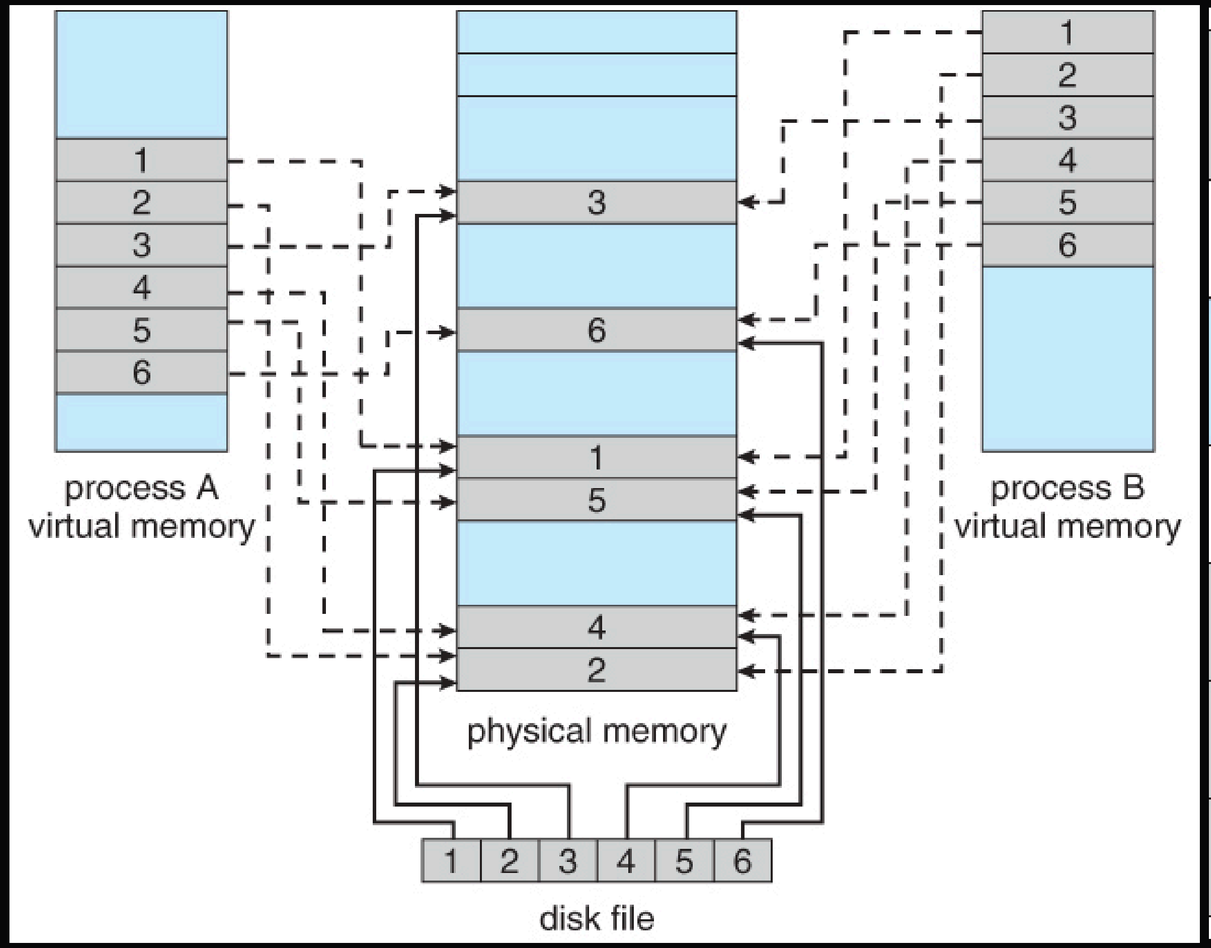
# SHARED MEMORY

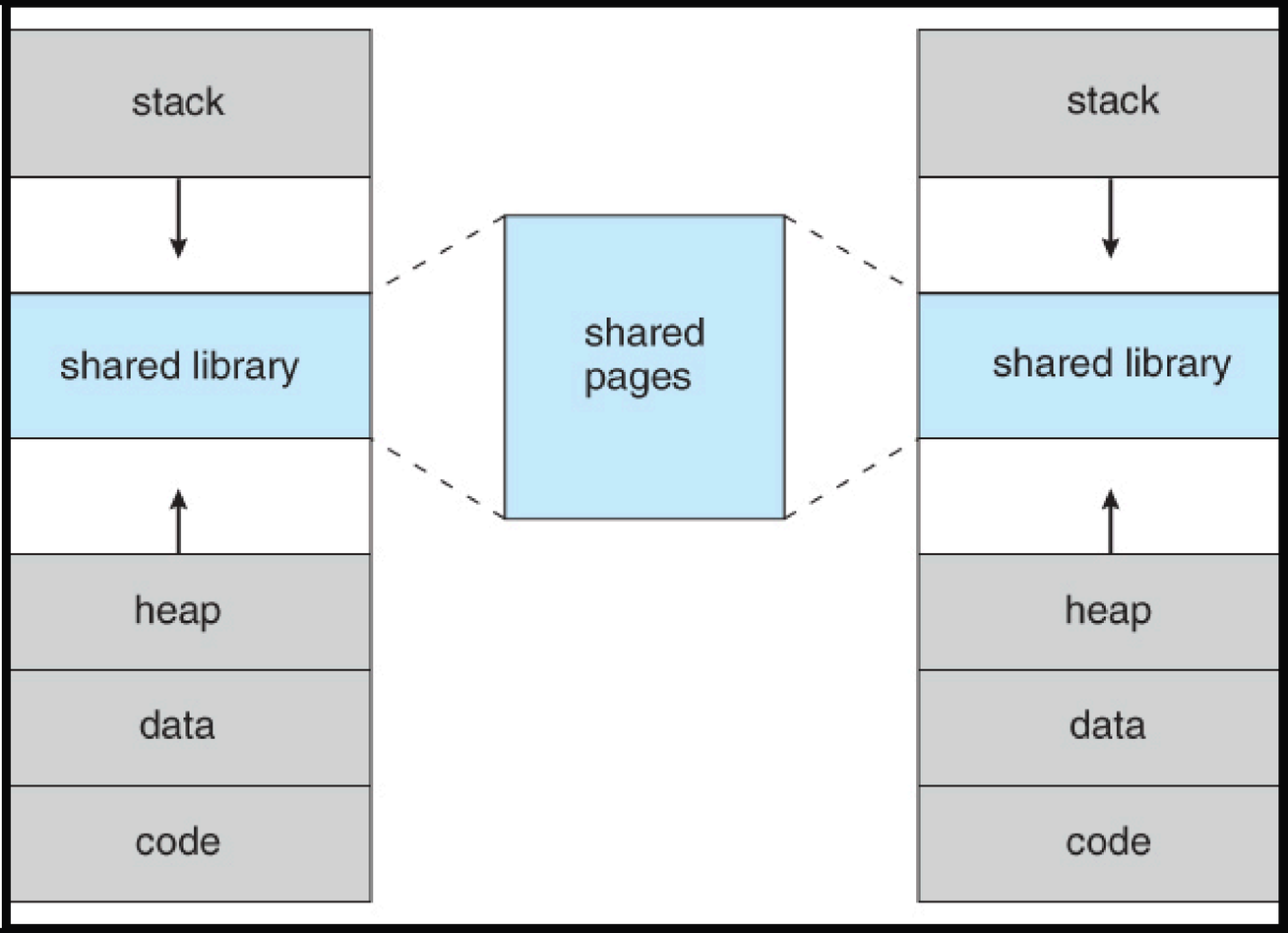



Используя механизмы виртуальной памяти появляется возможность избежать избыточного копирования

Balun.Courses

# MEMORY MAPPED FILES




IPC, загрузка динамических библиотек в память, ускорение IO операций с файлом

Balun.Courses

# MEMORY MAPPED FILES

```go
pageSize := os.Getpagesize()
pattern := strings.Repeat( s: "1", pageSize-1)
b.Run( name: "buffered read with temp buffer", func(b *testing.B) {
    scannerFd, err := os.Open(fpath)
    require.NoError(b, err)

    reader := bufio.NewReaderSize(scannerFd, os.Getpagesize())

    for i := 0; i < b.N; i++ {
        data, err := reader.ReadSlice( delim: ' ')
        require.NoError(b, err)

        require.Equal(b, unsafe.String(unsafe.SliceData(data[:len(data)-1]), len(data)-1), pattern)
    }
})

b.Run( name: "memory mapped file", func(b *testing.B) {
    mamoryMappedFd, err := os.Open(fpath)
    require.NoError(b, err)
    offset := 0

    for i := 0; i < b.N; i++ {
        data, err := unix.Mmap(int(mamoryMappedFd.Fd()), int64(offset), pageSize, unix.PROT_READ, unix.MAP_SHARED)
        require.NoError(b, err)

        offset += pageSize

        require.Equal(b, unsafe.String(unsafe.SliceData(data[:len(data)-1]), len(data)-1), pattern)
    }
})
```

MAP_SHARED

MAP_PRIVATE (CoW)

Когда что используется?

Balun.Courses

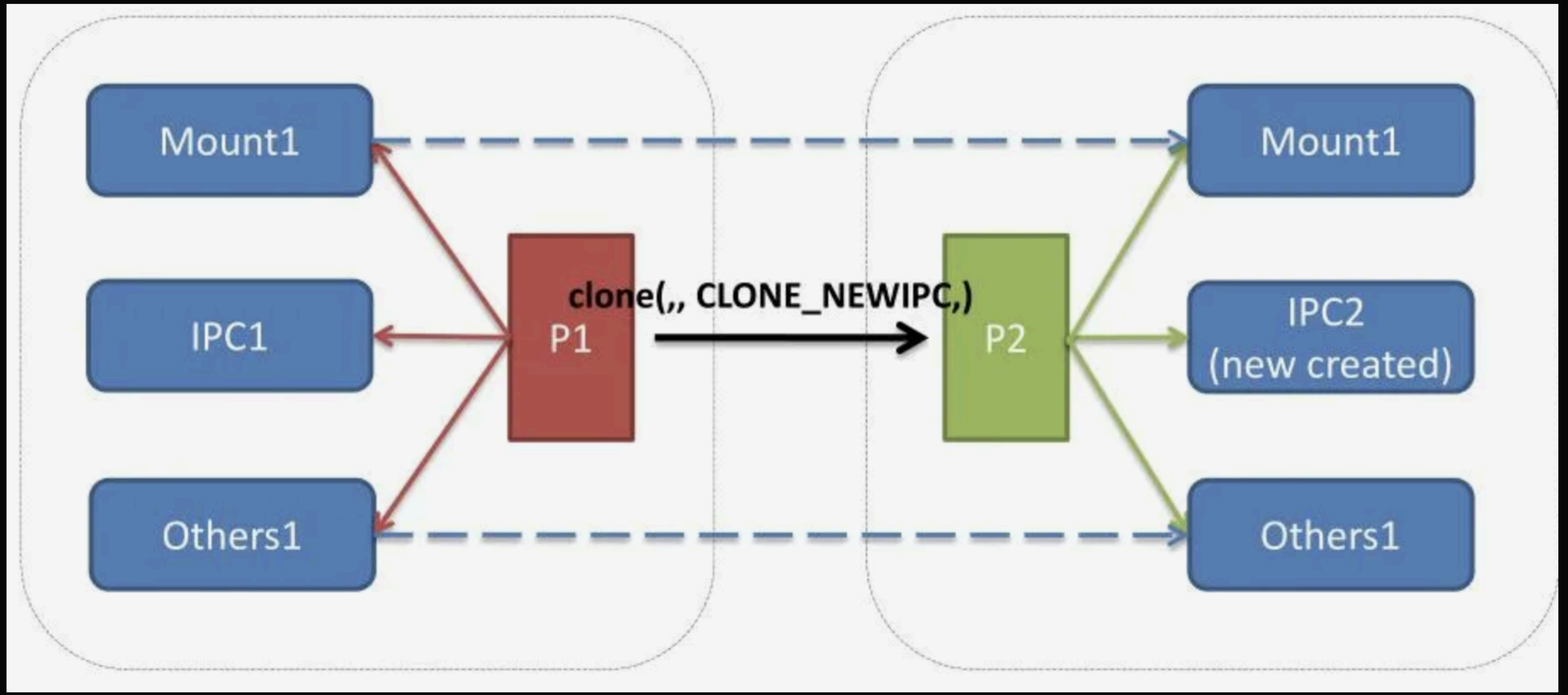
# MEMORY MAPPED FILES

```
go test -bench=. -benchmem -cpuprofile=cpu.prof -memprofile=mem.prof
goos: darwin
goarch: arm64
pkg: fork
BenchmarkMmap/buffered_read_with_temp_buffer-12                433006              2367 ns/op              32 B/op          2 allocs/op
BenchmarkMmap/memory_mapped_file-12                            426070              3874 ns/op             219 B/op          2 allocs/op
PASS
ok      fork    3.521s
```

Какие плюсы и минусы использования MMF? В каких случаях лучше использовать?

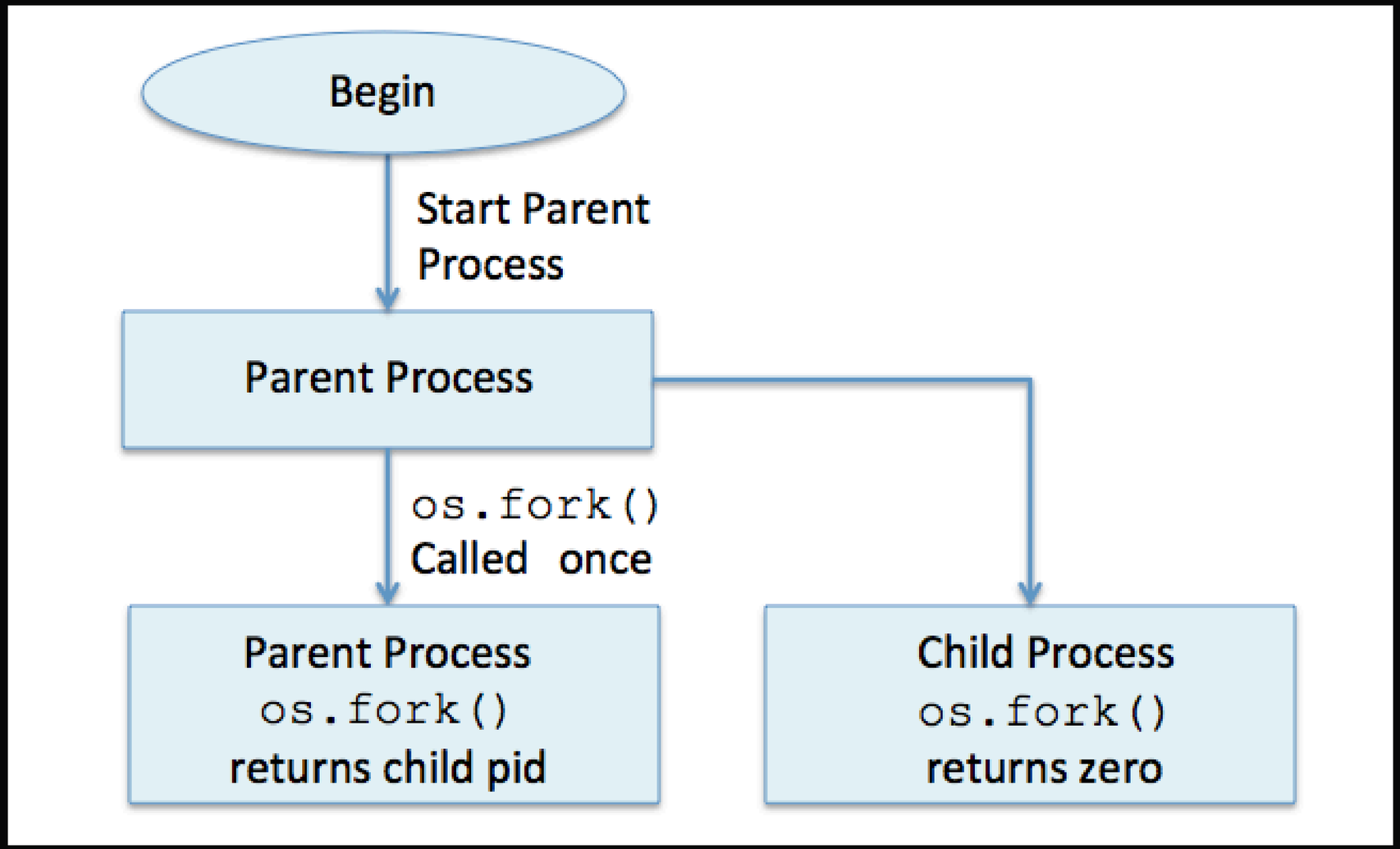
Balun.Courses

# СИСТЕМНЫЙ ВЫЗОВ CLONE



Balun.Courses

# СИСТЕМНЫЙ ВЫЗОВ FORK

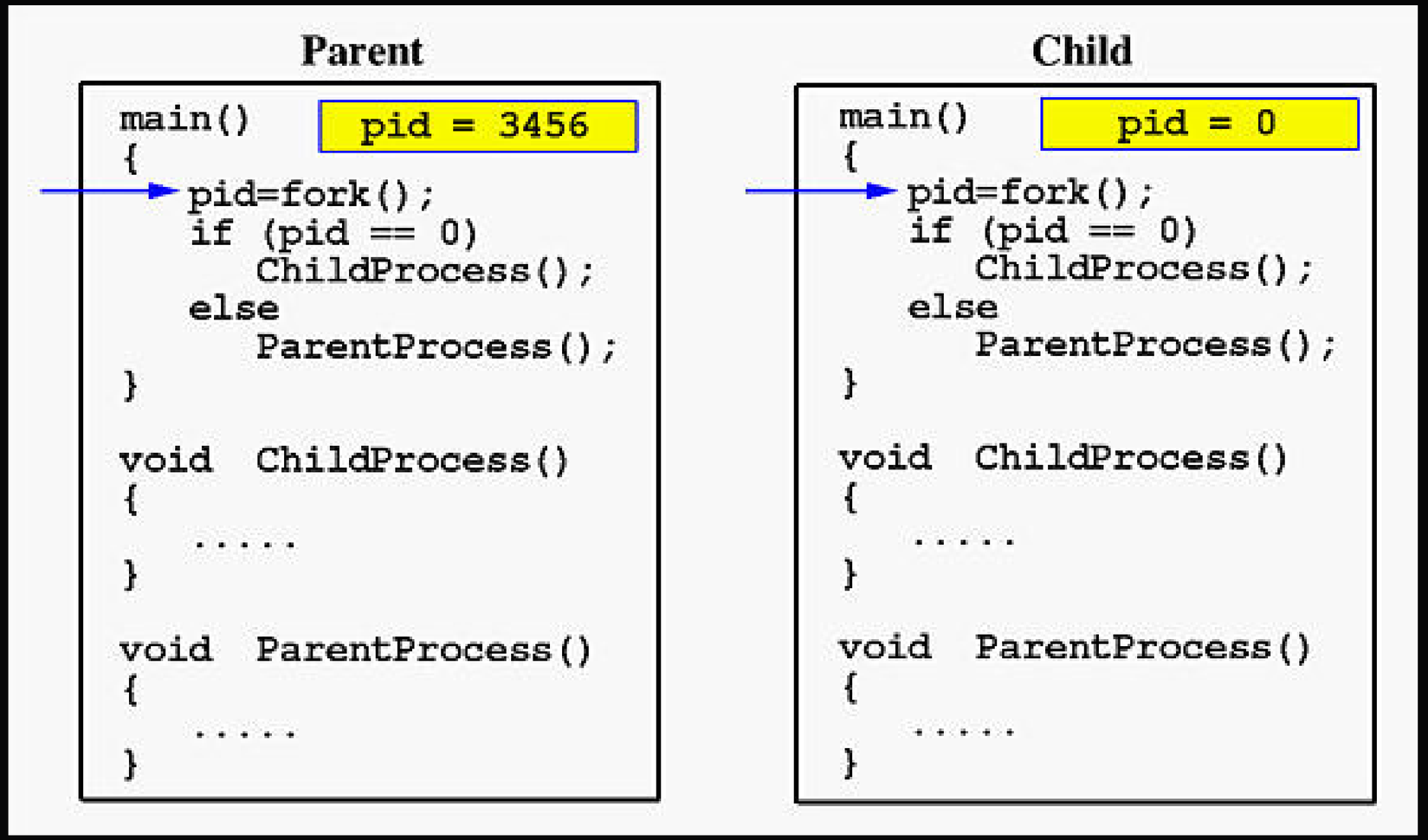



Чем отличается от clone?

Почему не будет корректно работать в случае Go?

Balun.Courses

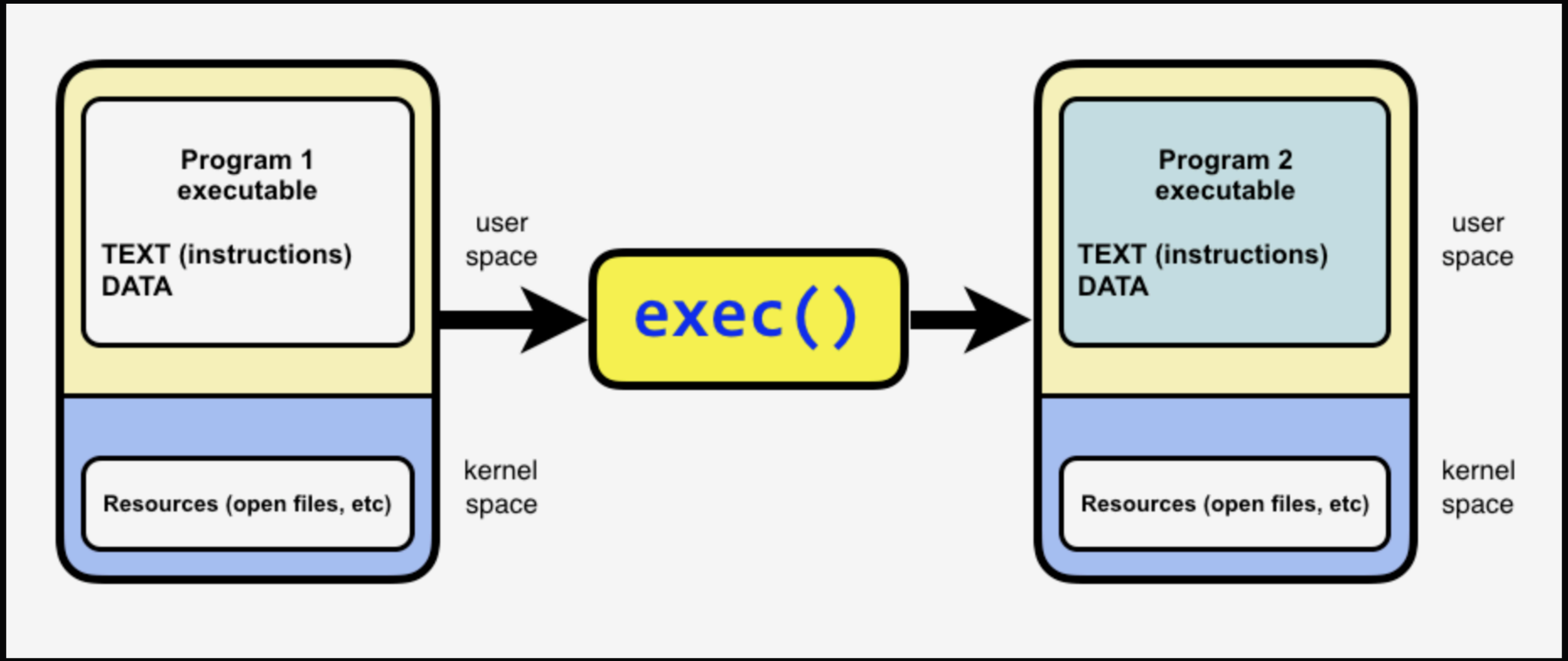
# СИСТЕМНЫЙ ВЫЗОВ FORK




Будет ли корректно работать для рантайма Go?

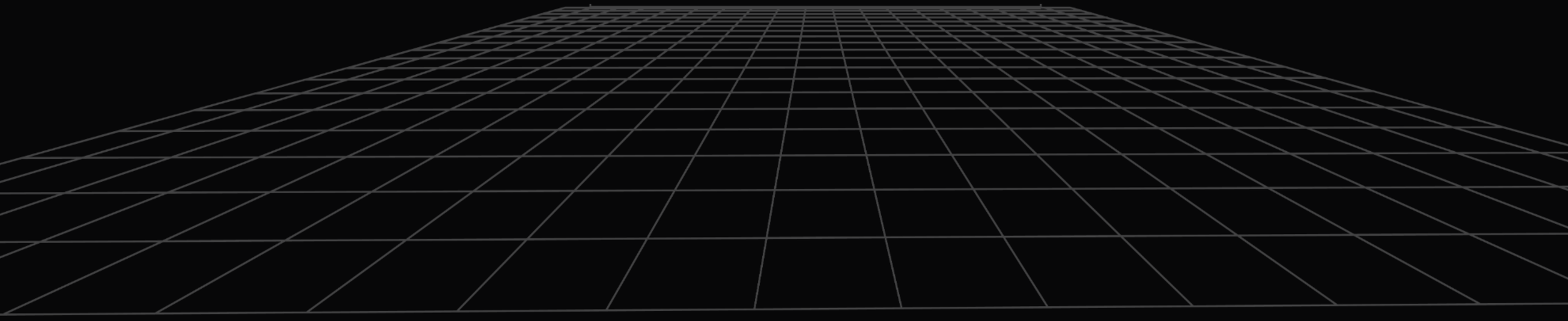
Balun.Courses

# СИСТЕМНЫЙ ВЫЗОВ EXEC



# DEMO

fork() CoW optimization in xv6



Balun.Courses

# FORKEXEC B GO

```go
func main() {
    environ := os.Environ()
    pid := os.Getpid()

    for i := range environ {
        pair := strings.Split(environ[i], sep: "=")
        k, v := pair[0], pair[1]

        if k == "CHILD_ENV" && v == "OK" {
            fmt.Printf( format: "I am child %d", pid)
            return
        }
    }

    wd, err := os.Getwd()

    if err != nil {
        panic(err)
    }
```

```go
args := append(os.Args, fmt.Sprintf( format: "child of %d", pid))
env := append(environ, fmt.Sprintf( format: "CHILD_ENV=OK"))

child, err := syscall.ForkExec(os.Args[0], args, &syscall.ProcAttr{
    Dir:   wd,
    Env:   env,
    Files: []uintptr{os.Stdin.Fd(), os.Stdout.Fd(), os.Stderr.Fd()},
})

if err != nil {
    panic(err)
}

fmt.Printf( format: "I am parent: %d, my child: %d\n", pid, child)
time.Sleep(time.Second * 1)
```

```
/Users/igorwalther/Library/Caches/JetBrains/GoLand2024.1/tmp/GoLand/___go_build_fork
I am parent: 92239, my child: 92241
I am child 92241
Process finished with the exit code 0
```

Balun.Courses

# OS/EXEC

```
cmd := exec.CommandContext(ctx, *sp)
cmd.Stdin = serverReader
cmd.Stdout = watchdogWriter
cmd.Stderr = os.Stderr
cmd.Env = os.Environ()
cmd.Args = append(cmd.Args, strings.Split(*serverArguments, sep: " ")...)

err = cmd.Start()

//for {
//  pid1, e = syscall.Wait4(p.Pid, &status, 0, &rusage)
//  if e != syscall.EINTR {
//      break
//  }
//}
err = cmd.Wait()
```

Помним, что пакет os платформонезависимый. Но для всех ли платформ одинаковые возможности?

Balun.Courses

# OS/EXEC

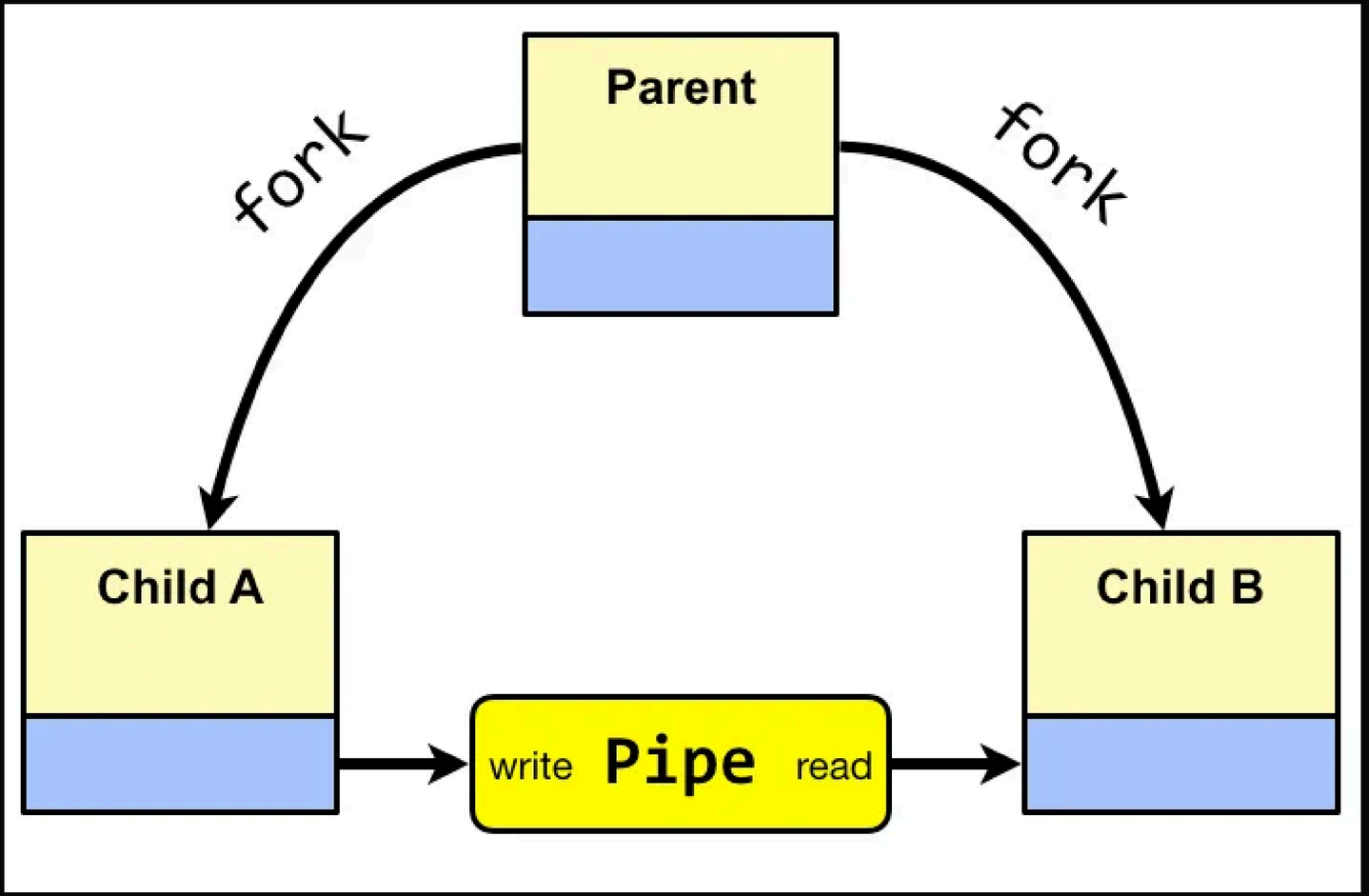
```
// Stdin specifies the process's standard input.
//
// If Stdin is nil, the process reads from the null device (os.DevNull).
//
// If Stdin is an *os.File, the process's standard input is connected
// directly to that file.
//
// Otherwise, during the execution of the command a separate
// goroutine reads from Stdin and delivers that data to the command
// over a pipe. In this case, Wait does not complete until the goroutine
// stops copying, either because it has reached the end of Stdin
// (EOF or a read error), or because writing to the pipe returned an error,
// or because a nonzero WaitDelay was set and expired.
Stdin io.Reader

// ExtraFiles specifies additional open files to be inherited by the
// new process. It does not include standard input, standard output, or
// standard error. If non-nil, entry i becomes file descriptor 3+i.
//
// ExtraFiles is not supported on Windows.
ExtraFiles []*os.File
```

Как организовать взаимодействие между процессами? (IPC)

Как наследовать дескрипторы на Windows в Go?

Balun.Courses

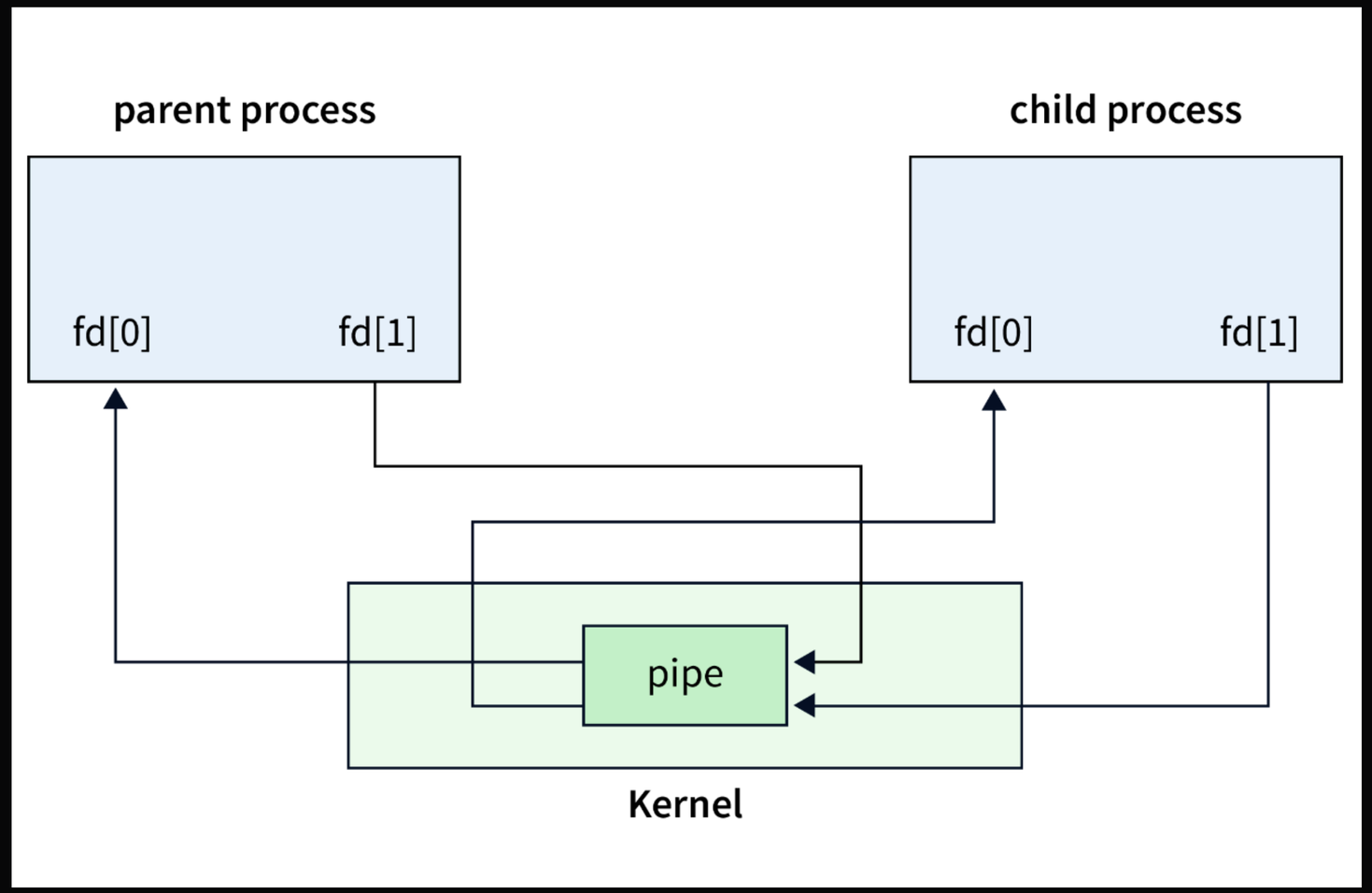
# PIPE




Как поддержать межпроцессное взаимодействие?

Balun.Courses

# PIPE




Как поддержать межпроцессное взаимодействие?

# DEMO

Homework