Iowa State University

Department of Electrical and Computer Engineering

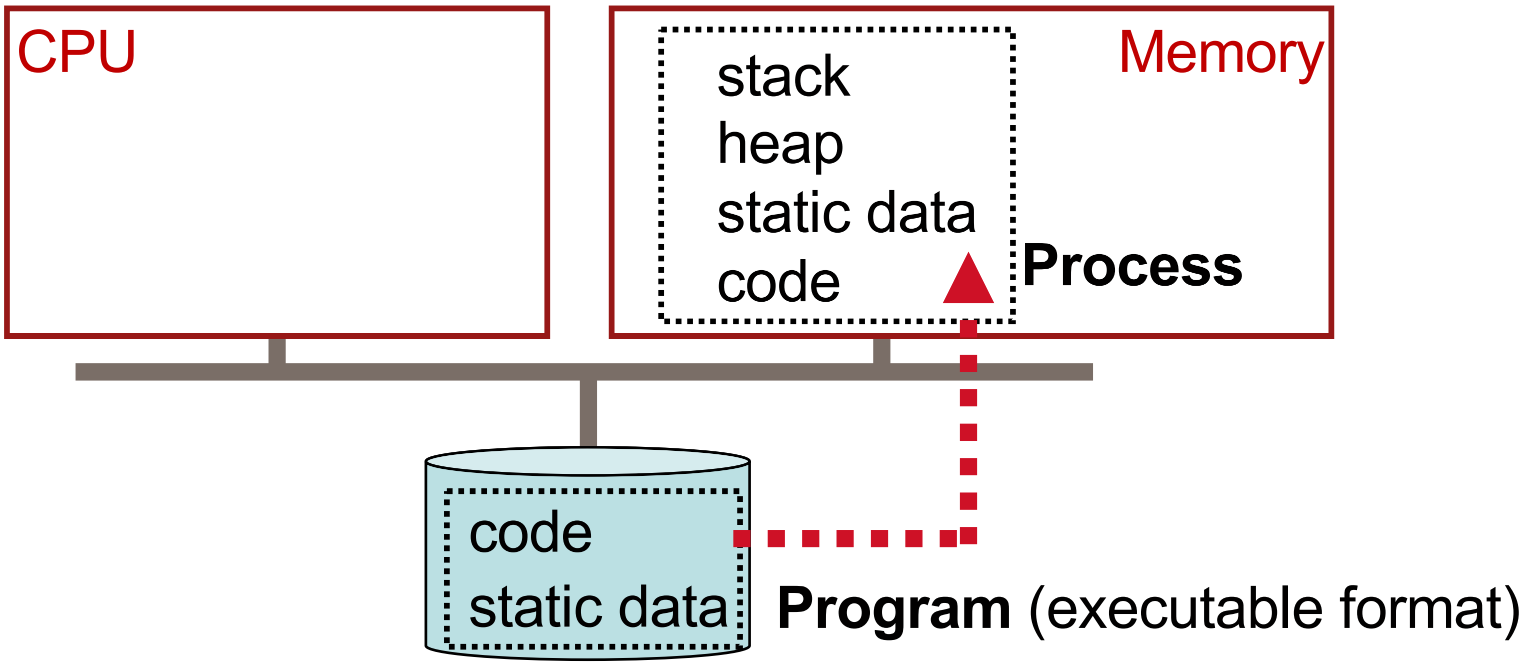
Lecture 06:
Processes II

Agenda

- Recap
- Processes II
  - Process State
  - Process Context
  - Process API: fork()

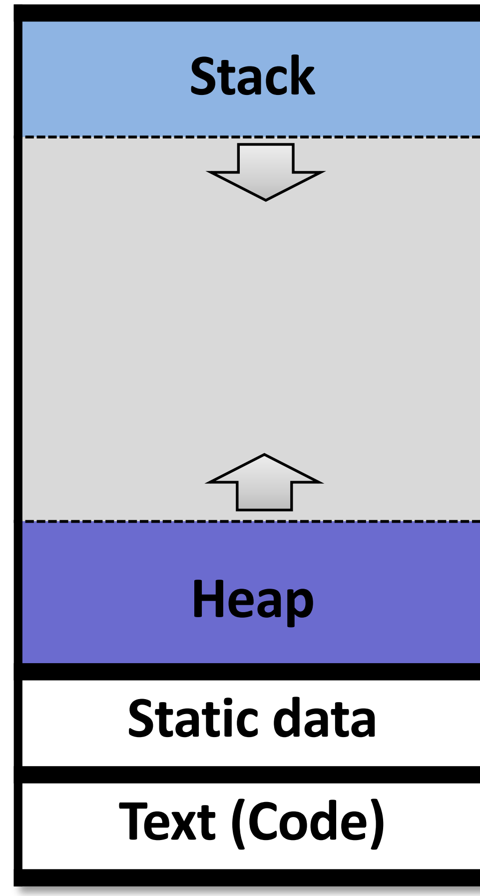
Recap

- Process
  - a program in execution
  - an instance of a running program
  - an execution stream in the context of a process state



Recap

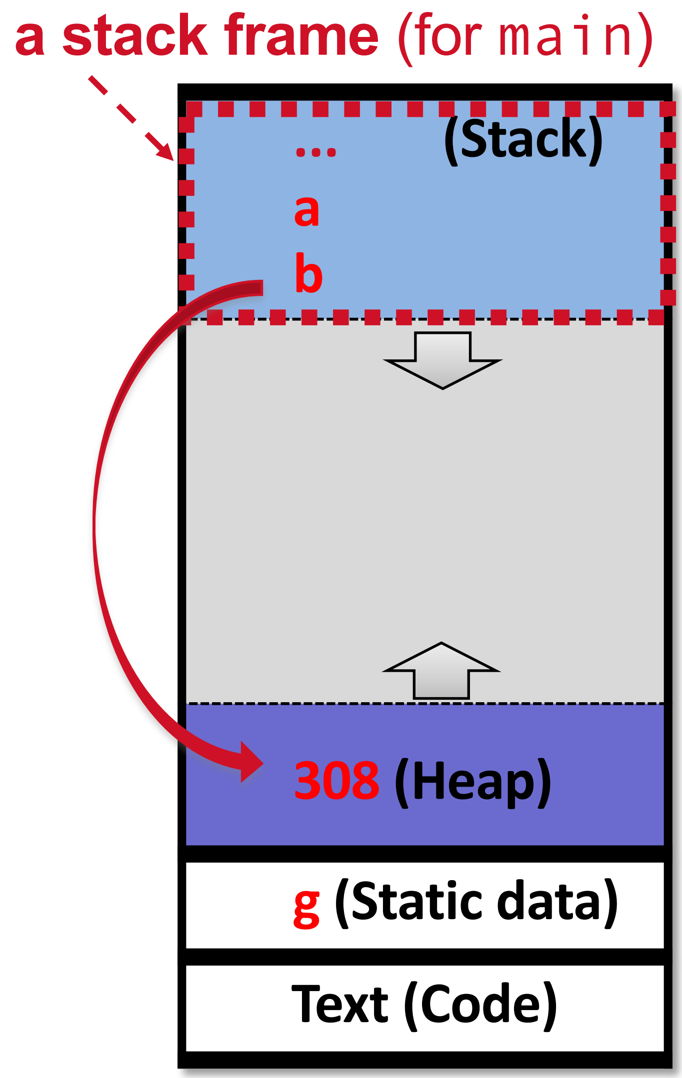
- Process Address Space
  - stack
    - local variables, function parameters, return address
  - heap
    - dynamically allocated data
      - e.g., `malloc()`
  - static data
    - global/static variables
  - code (text)
    - instructions of the program



Recap

- Process Address Space
  - example
    - memory leak problem

```
int g;
int main() {
  int a;
  int*b = (int*)malloc(sizeof(int));
  *b = 308;
  return 0;
}
```



Agenda

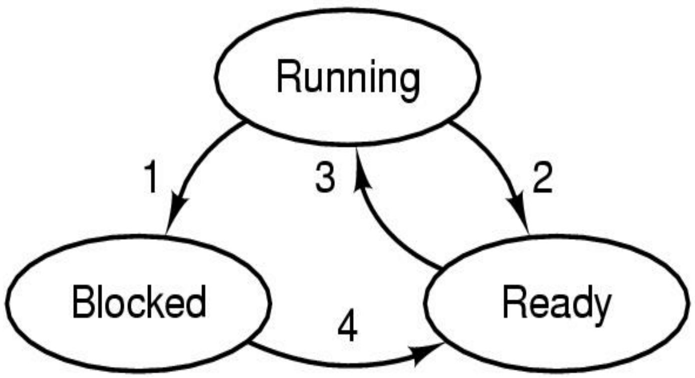
- ~~Recap~~
- Processes II
  - Process State
  - Process Context
  - Process API: fork()

Process State

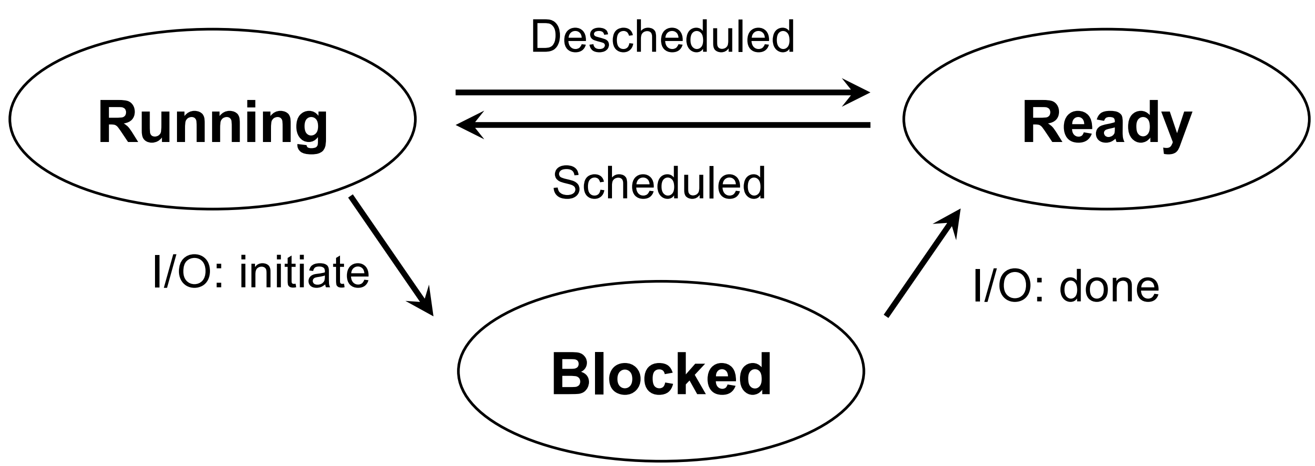
- A Process can be in one of three basic states
  - **Running**
    - A process is running on a processor.
  - **Ready**
    - A process is ready to run; but for some reason the OS has chosen not to run it at this moment.
  - **Blocked**
    - A process has performed some I/O operation (e.g., a read from the disk); it becomes blocked so that other process can use the processor
  - Note: practical OSes usually have more than 3 process states

Process State

- State transition



1. Process blocks for input
2. Scheduler picks another process
3. Scheduler picks this process
4. Input becomes available



Agenda

- ~~Recap~~
- Processes II
    - ~~Process State~~
    - Process Context
    - Process API: fork()

Process Context

- OS maintains some key data structures to track various information of processes
  - Process lists/queues
    - Ready processes
    - Blocked processes
    - Current running process
  - Process table
    - one entry per process
    - each entry is a "Process Control Block (PCB)"
      - containing all information about a process, i.e. "**process context**"

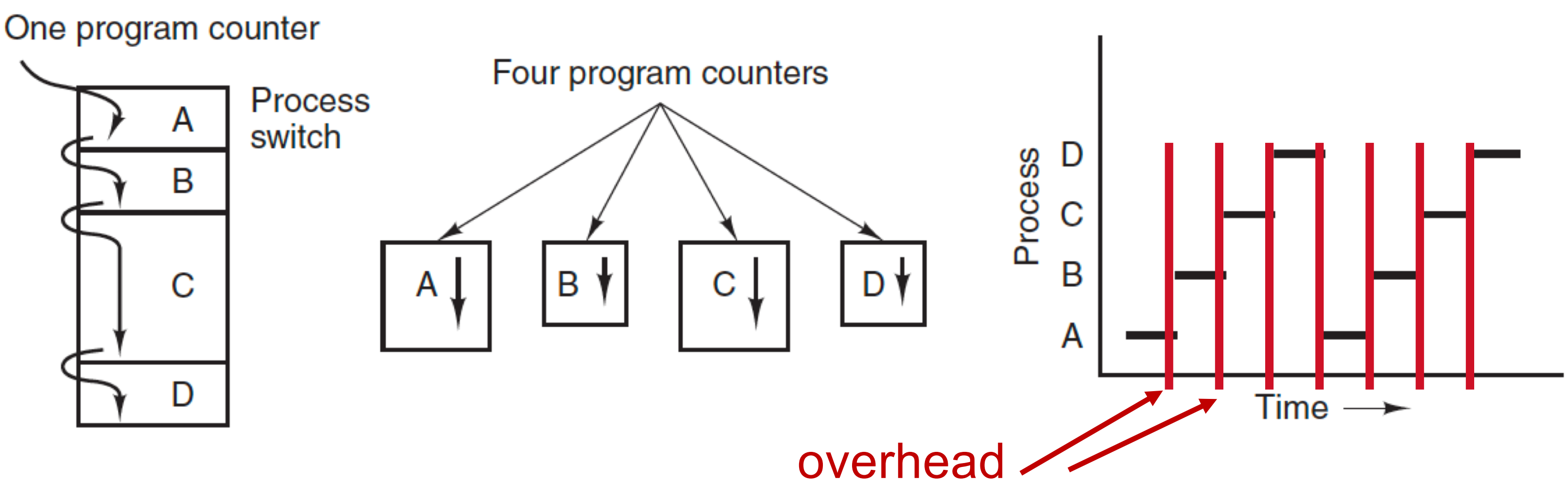
Process Context

- Typical information stored in a PCB

Process management	Memory management	File management
Registers	Pointer to text segment info	Root directory
Program counter	Pointer to data segment info	Working directory
Program status word	Pointer to stack segment info	File descriptors
Stack pointer		User ID
Process state		Group ID
Priority		
Scheduling parameters		
Process ID		
Parent process		
Process group		
Signals		
Time when process started		
CPU time used		
Children's CPU time		
Time of next alarm		

Process Context

- Context switch
  - switching the CPU to another process by
    - saving the context of an old process
    - loading the context of a new process
  - it's overhead
    - CPU not executing user instructions during the switch



Getting real: code for process

- xv6 (MIT's teaching OS)
  - https://pdos.csail.mit.edu/6.828/2019/xv6.html

Getting real: code for process

- xv6 (MIT’s teaching OS)
  - https://pdos.csail.mit.edu/6.828/2019/xv6.html

```
// xv6 saves/restores the registers (register context)
// to stop/restart a process
struct context {
    int eip;    // Index pointer register
    int esp;    // Stack pointer register
    int ebx;    // Called the base register
    int ecx;    // Called the counter register
    int edx;    // Called the data register
    int esi;    // Source index register
    int edi;    // Destination index register
    int ebp;    // Stack base pointer register
};

// a process can be in the following states
enum proc_state { UNUSED, EMBRYO, SLEEPING,
                  RUNNABLE, RUNNING, ZOMBIE};
```

Getting real: code for process

- xv6 (MIT's teaching OS)

```
// xv6 maintains a proc struct for each process  (i.e., PCB)
// including its register context and state
struct proc {
    char *mem;          // Start of process memory
    uint sz;                  // Size of process memory
    char *kstack;             // Bottom of kernel stack
                        // for this process
    enum proc_state state;    // Process state
    int pid;                  // Process ID
    struct proc *parent;      // Parent process
    void *chan;               // If non-zero, sleeping on chan
    int killed;               // If non-zero, have been killed
    struct file *ofile[NOFILE]; // Open files
    struct inode *cwd;  // Current directory
    struct context context;   // Switch here to run process
    struct trapframe *tf;     // Trap frame for the
                              // current interrupt
};
```

Getting real: code for process

- Linux
  - Linux maintains a `task_struct` for each process
    - https://github.com/torvalds/linux/blob/master/include/linux/sched.h

```
struct task_struct {
    ...
    pid_t pid;
    ...
};
```

Agenda

- ~~Recap~~
- Processes II
    - ~~Process State~~
    - ~~Process Context~~
    - Process API: fork()

Process API

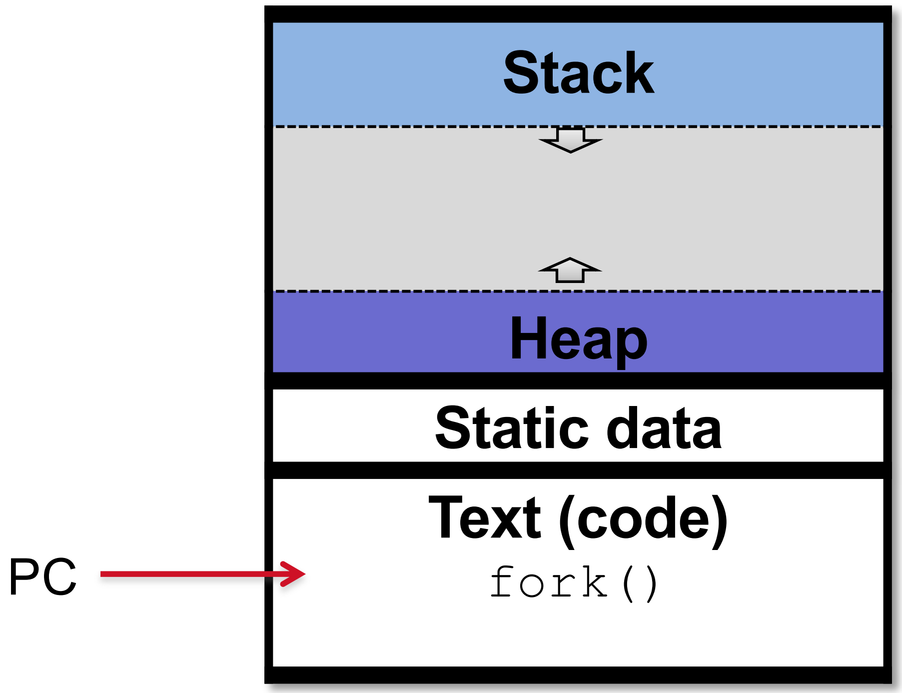
- Common APIs available on any modern OS
  - **Create**
    - Create a new process to run a program
  - **Destroy**
    - Halt a runaway process
  - **Wait**
    - Wait for a process to stop running
  - **Status**
    - Get some status info about a process
  - **Miscellaneous Control**
    - e.g., suspend a process and then resume it

Process API

- Example: fork()
  - creates a new process by duplicating the calling process
    - The new process is referred to as the **child** process
    - The calling process is referred to as the **parent** process
    - The child process is a copy of the parent process
      - Same core image
      - Same context (except process id): registers, open files, ...
  - On success
    - the **process ID** of the child is returned in the parent
    - **0** is returned in the child
  - On failure
    - -1 is returned in the parent, no child process is created

Process API

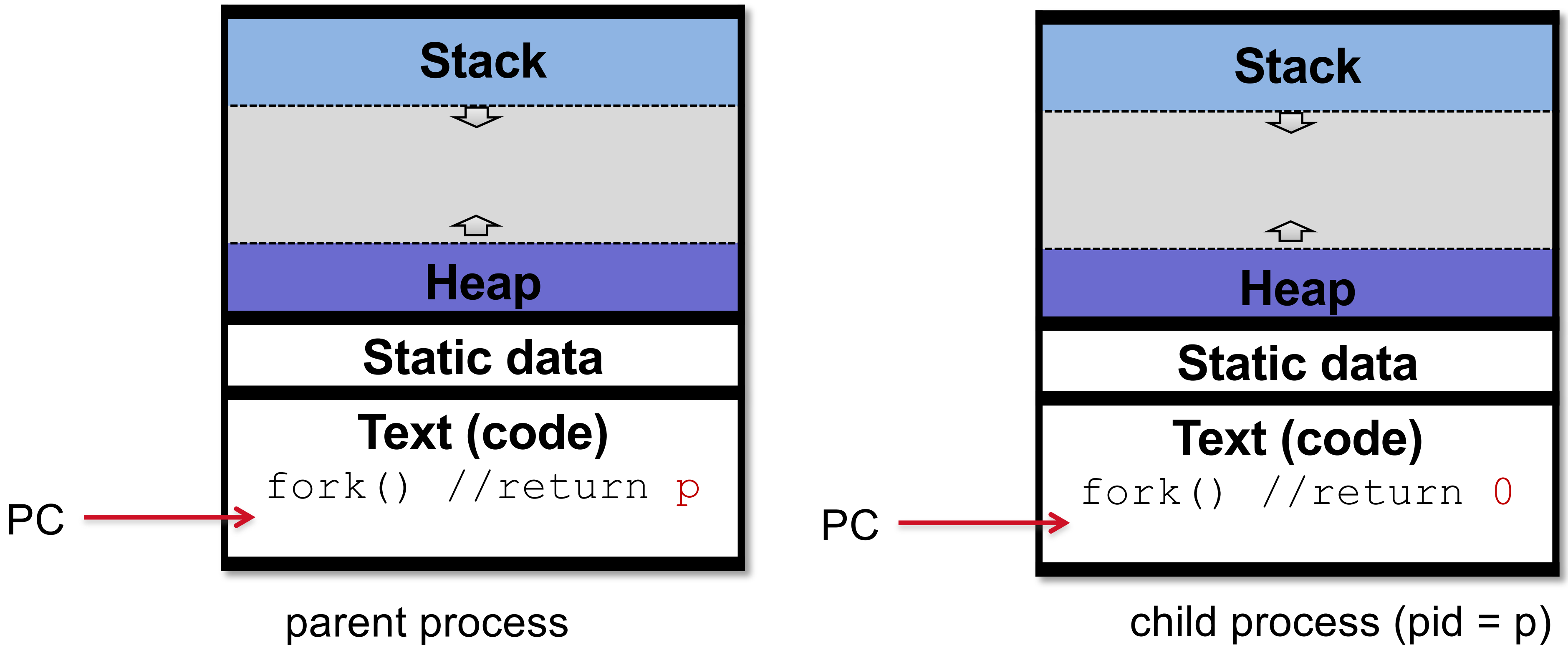
- Example: fork()
  - before fork():



parent process

Process API

- Example: fork()
  - after fork():



Process API

- Example: fork()

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

int main(int argc, char *argv[]){
    printf("hello world (pid:%d)\n", (int) getpid()); //get process ID
    int rc = fork();        // create a child process
    if (rc < 0) {           // fork failed; exit
        fprintf(stderr, "fork failed\n");
        exit(1);
    } else if (rc == 0) { // child (new process)
        printf("hello, I am child (pid:%d)\n", (int) getpid());
    } else {                // parent goes down this path (main)
        printf("hello, I am parent of %d (pid:%d)\n",
        rc, (int) getpid());
    }
    return 0;
}c
```

Agenda

- ~~Recap~~
- ~~Processes II~~
    - ~~Process State~~
    - ~~Process Context~~
    - ~~Process API: fork()~~

Questions?

*acknowledgement: slides include content from “Modern Operating Systems” by A. Tanenbaum, “Operating Systems Concepts” by A. Silberschatz etc., “Operating Systems: Three Easy Pieces” by R. Arpaci-Dusseau etc., and anonymous pictures from internet.