Introduction to Operating Systems

Prof. Yongtae Kim

Computer Science and Engineering
Kyungpook National University

What Happens when a Program Runs?

- **A running program does one very simple thing**
  - Executing instructions by von Neumann model of computing
  - The processor fetches an instruction from memory, decode and executes
  - Keep doing this until the program finally completes

- **There is a body of software that is responsible for making it easy to run programs, such as**
  - Allowing programs to share memory
  - Enabling programs to interact with devices

- **That body of software is called the operating system (OS)**
  - It is in charge of making sure the system operates correctly and efficiently in an easy-to-use manner

Virtualization → 가상화.

추상화, 가상화.

- **The primary way the OS does this is through virtualization**
  - The OS takes a physical resource (i.e. processor, memory) and transforms it into a more general, powerful, and easy-to-use virtual form
  - Thus, we sometimes refer to the OS as a virtual machine

- **A typical OS exports a few hundred system calls that are available to applications**

  운영체제 → fopen

  - To allow users to tell the OS what to do and thus make use of the features of the virtual machine (i.e. running a program, allocating memory, accessing file)
  - We sometime can say that the OS provides a standard library to applications

  운영체제가 관리

- **Virtualization allows many programs**

  디스크들

  - 1) to run (sharing the CPU), 2) to access devices (sharing the disks, etc), 3) to concurrently access their own instructions and data (sharing the memory)
  - The OS is sometime known as a resource manager in which each of the CPU, memory, and disk is the resource of the system

  가상화 해서 제공.

Virtualizing the CPU (1)

- **Let's consider a simple example of code that loops and prints**

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/time.h>
#include <assert.h>
#include "common.h"

int
main(int argc, char *argv[])
{
    if (argc != 2) {
        fprintf(stderr, "usage: cpu <string>\n");
        exit(1);
    }
    char *str = argv[1];
    while (1) {
        Spin(1);
        printf("%s\n", str);
    }
    return 0;
}
```

output of the program

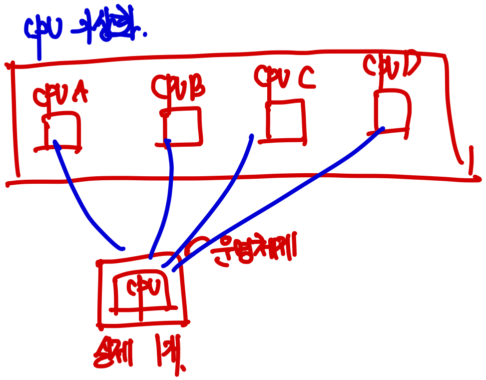
```
prompt> gcc -o cpu cpu.c -Wall
prompt> ./cpu "A"
A
A
A
A
^C
prompt>
```

Spin() is a function that repeatedly checks the time and returns once it has run for a second

- **When running it on a system with a single processor**
  - Not too interesting of a run – repeatedly checks time until a sec. has elapsed
  - Run forever; by pressing "Control-C" to halt the program

Virtualizing the CPU (2)

- **When running many programs at once on the same system,**
  - Things are getting a little more interesting – somehow all four of these programs seems to be running at the same time




```
prompt> ./cpu A & ./cpu B & ./cpu C & ./cpu D &
[1] 7353
[2] 7354
[3] 7355
[4] 7356
A
B
D
C
A
B
D
C
A
...
```

4개를 동시에

OS가 가상화된 CPU를 여러개 만들어줌.

- **The OS is in charge of this illusion that the system has a very large number of virtual CPUs**
  - Turning a single CPU into a seemingly infinite number of CPUs and thus allowing many programs to seemingly run at once
  - We call it virtualizing the CPU

Virtualizing Memory (1)

- **The physical memory model of modern machines is very simple**
  - Memory is just an array of bytes and is accessed using address and data
  - Memory is accessed all the time when a program is running (e.g. IF, LW, SW)
- **Let's look at a program that allocates some memory by malloc()**

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

int
main(int argc, char *argv[])
{
    int *p = malloc(sizeof(int));                          // a1
    assert(p != NULL);
    printf("(%d) address pointed to by p: %p\n",
           getpid(), p);                                   // a2
    *p = 0;                                                // a3
    while (1) {
        Spin(1);
        *p = *p + 1;
        printf("(%d) p: %d\n", getpid(), *p);              // a4
    }
    return 0;
}
```

```
prompt> ./mem
(2134) address pointed to by p: 0x200000
(2134) p: 1
(2134) p: 2
(2134) p: 3
(2134) p: 4
(2134) p: 5
^C
```

  - a1) allocate some memory, a2) print its address, a3) put zero into its first slot, a4) increase p by 1 and print process id (PID), which is unique per process
  - In fact, this result is not too interesting

Virtualizing Memory (2)

- **When running multiple instances of the same program,**
  - The memory at the same address (**0x200000**) is allocated for two programs
  - Each program seems to be updating the value at **0x200000** independently!
  - It is as if each has its own private memory, instead of sharing memory

```
prompt> ./mem &; ./mem &
[1] 24113
[2] 24114
(24113) address pointed to by p: 0x200000
(24114) address pointed to by p: 0x200000
(24113) p: 1
(24114) p: 1
(24114) p: 2
(24113) p: 2
(24113) p: 3
(24114) p: 3
(24113) p: 4
(24114) p: 4
...
```

→ 백그라운드

→ 가상화된 메모리 주소.

→ 논리주소

실제 물리 주소.

소프트웨어가 따로 돌리는 공간이 있음

- **This is what is happening here as the OS is virtualizing memory**
  - Each process accesses its own private virtual address space, which the OS somehow maps onto the physical memory of the machine
  - A memory within a running program does not affect the other's address space
  - The reality is that physical memory is a shared resource, managed by the OS

Concurrency (1) →동시성

- **The OS is juggling many things at once by first running one process, then another, and so forth**
  - Modern multi-threaded programs exhibit the problems of concurrency
- **Let's demonstrate with an example of a multi-threaded program**
  - The main program creates two threads using **Phread_create()**
  - The thread is a function running within the same memory space
  - Each thread calls **worker()** that increases a counter in a loop
  - Output with **loops** = 1000

```
prompt> gcc -o thread thread.c -Wall -pthread
prompt> ./thread 1000
Initial value : 0
Final value   : 2000
```

  - When **loops** = N, the output is expected to be 2N since we have two threads

```c
#include <stdio.h>
#include <stdlib.h>
#include "common.h"
#include "common_threads.h"

volatile int counter = 0;
int loops;

void *worker(void *arg) {
    int i;
    for (i = 0; i < loops; i++) {
        counter++;
    }
    return NULL;
}

int main(int argc, char *argv[]) {
    if (argc != 2) {
        fprintf(stderr, "usage: threads <value>\n");
        exit(1);
    }
    loops = atoi(argv[1]);
    pthread_t p1, p2;
    printf("Initial value : %d\n", counter);

    Pthread_create(&p1, NULL, worker, NULL);
    Pthread_create(&p2, NULL, worker, NULL);
    Pthread_join(p1, NULL);
    Pthread_join(p2, NULL);
    printf("Final value   : %d\n", counter);
    return 0;
}
```

Concurrency (2)

- **Let's run the same, but with higher values for `loops`**
  - When we gave an input of 100000, we first get 143012, instead of 200000
  - We still get a wrong value of 137298 for the second try

```
prompt> ./thread 100000
Initial value : 0
Final value   : 143012     // huh??
prompt> ./thread 100000
Initial value : 0
Final value   : 137298     // what the??
```

- **The reason for these odd and unusual outcomes related to how instructions are executed, which is once at a time**
  - Incrementing the shared counter takes three instructions: 1) load the counter value from memory into a register, 2) increment it, 3) store it back into memory
  - These three instructions do not execute atomically, which causes the problem of concurrency

동시성 문제

Persistence (1)

disk.

영속성

- **Hardware and software are needed to store data persistently**
  - In system memory, data can be easily lost as devices, such as DRAM, store values in a volatile manner
  - The hardware comes in I/O devices, such as hard drive or solid-state drives
  - The software (OS) that manages the disk is called the file system, which is responsible for storing any files the user creates efficiently and reliably

- **Unlike the CPU and memory virtualization, the OS does not create a private, virtualized disk for each application**
  - It is assumed that users will want to share information that is in files
  - e.g.) google docs

데이터를 share 할 수 있어야 한다.

∴ shared-disk.

disk들.

Persistence (2)

- **Let's look at some code for better understanding**

```
#include <stdio.h>
#include <unistd.h>
#include <assert.h>
#include <fcntl.h>
#include <sys/types.h>

int main(int argc, char *argv[]) {
    int fd = open("/tmp/file", O_WRONLY|O_CREAT|O_TRUNC,
                  S_IRWXU);
    assert(fd > -1);
    int rc = write(fd, "hello world\n", 13);
    assert(rc == 13);
    close(fd);
    return 0;
}
```

file. descriptor.

2

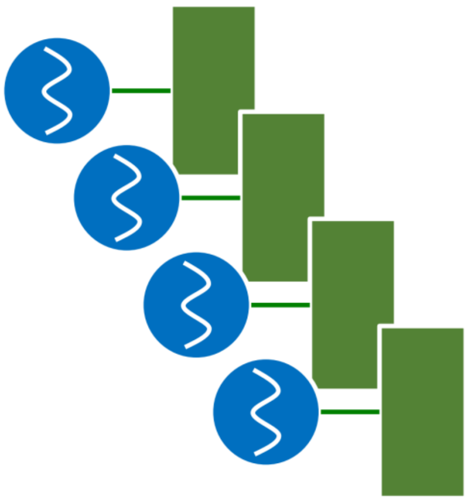
  - The program makes three calls into the OS: 1) **`open()`** to open the file, 2) **`write()`** to write some data to the file, 3) **`close()`** to close the file
  - These system calls are routed to the part of the OS called the file system, which handles the requests

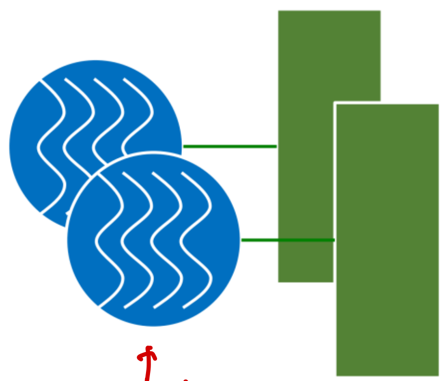
crash에 대한 것도.

- **What the OS does in order to write to disk?**
  - Figure out where on disk this new data will reside
  - Issue I/O request to the underlying storage device
- **The file system can handle system crashes during writes**

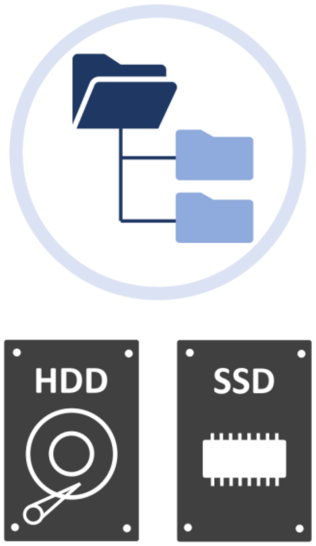
Summary

- **Virtualization**
  - Processes
  - Scheduling
  - Virtual memory
- **Concurrency**
  - Threads
  - Locks
  - Semaphores
- **Persistence**
  - I/O devices
  - File systems





Threads



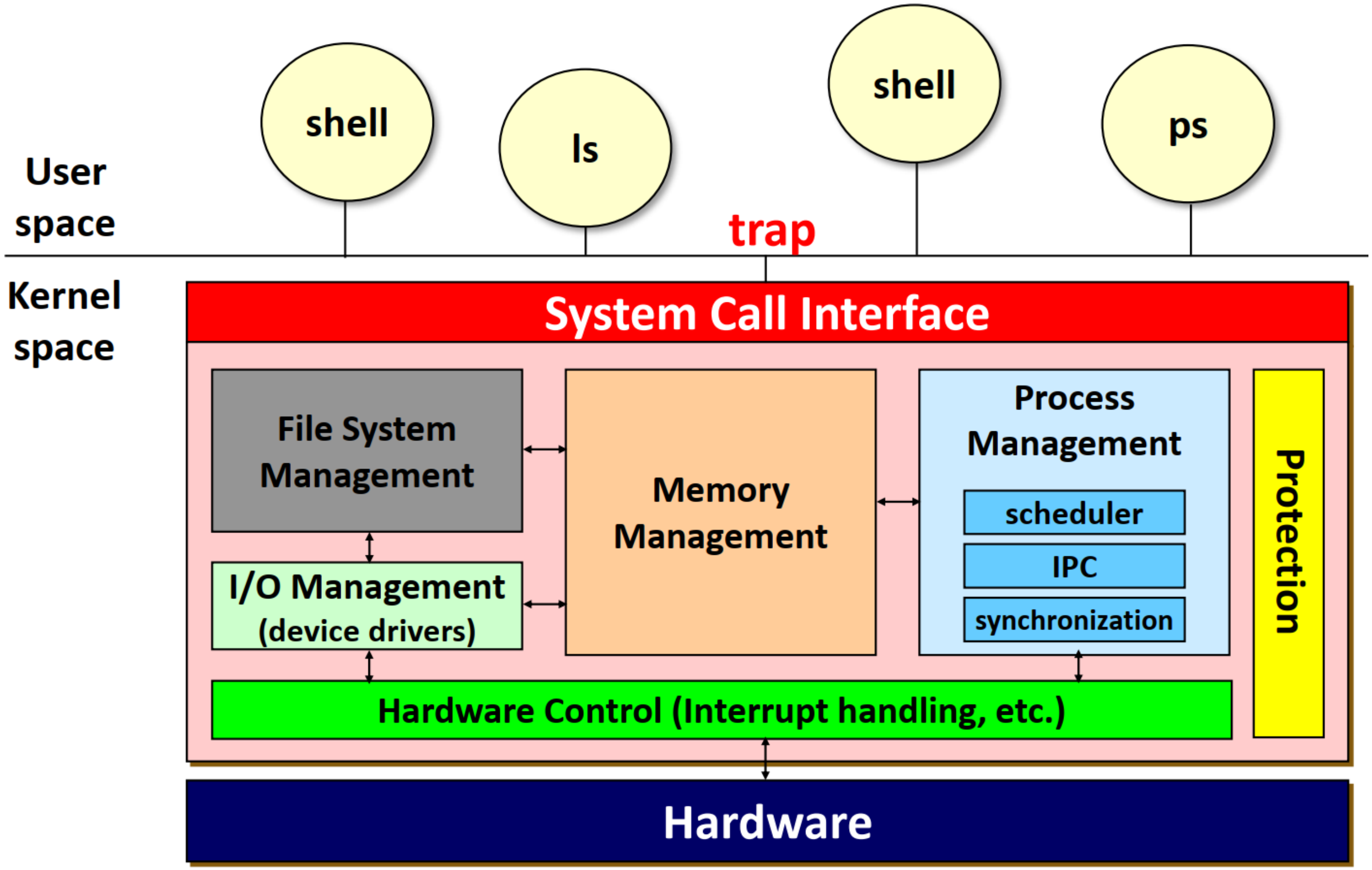
Courtesy of Prof. Jin-Soo Kim @ SNU

The Abstraction: The Process

Prof. Yongtae Kim

Computer Science and Engineering
Kyungpook National University

OS Internals



Courtesy of Prof. Jin-Soo Kim @ SNU

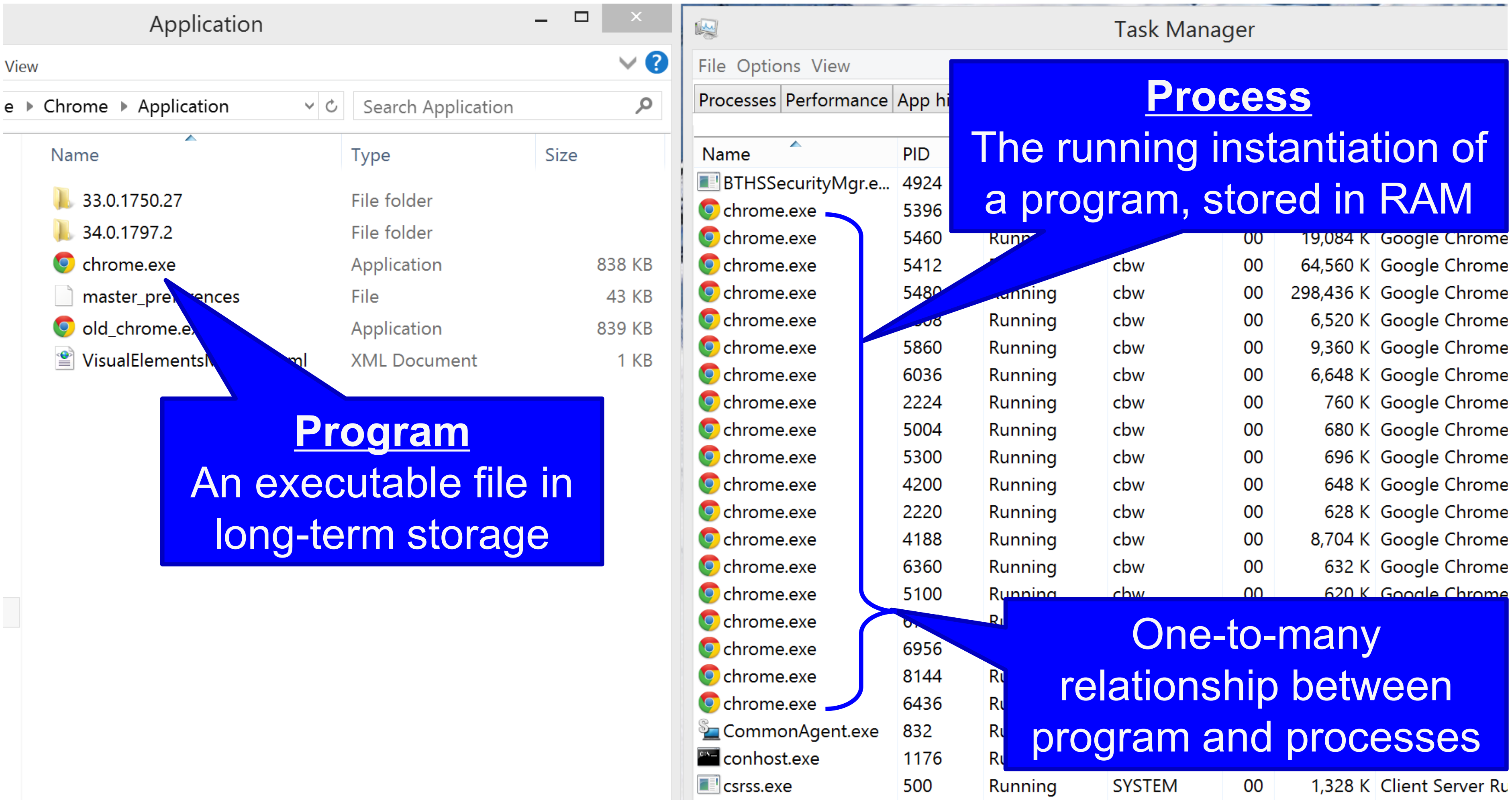
How to Provide the Illusion of Many CPUs?

- **The definition of a process, informally, is a running program**
  - The program itself is a lifeless thing; just sitting there on the disk, a bunch of instructions with some data
  - The OS gets them running, transforming it into something useful
  - We often want to run more than one program at once

- **The OS creates the illusion by virtualizing the CPU**
  - The OS can promote the illusion that many virtual CPUs exist when in fact there is only one physical CPU by time sharing of the CPU
  - Time sharing is implemented by running one process, then stopping it and running another, and so forth, and its potential cost is performance

- **To implement the virtualization, the OS will need both**
  - some low-level machinery: mechanisms; how to do something? (context switching)
  - some high-level intelligence: policies; what should be done? (scheduling)

Program vs Process



The Abstraction: A Process and API

- **The abstraction provided by OS of a running program is process**

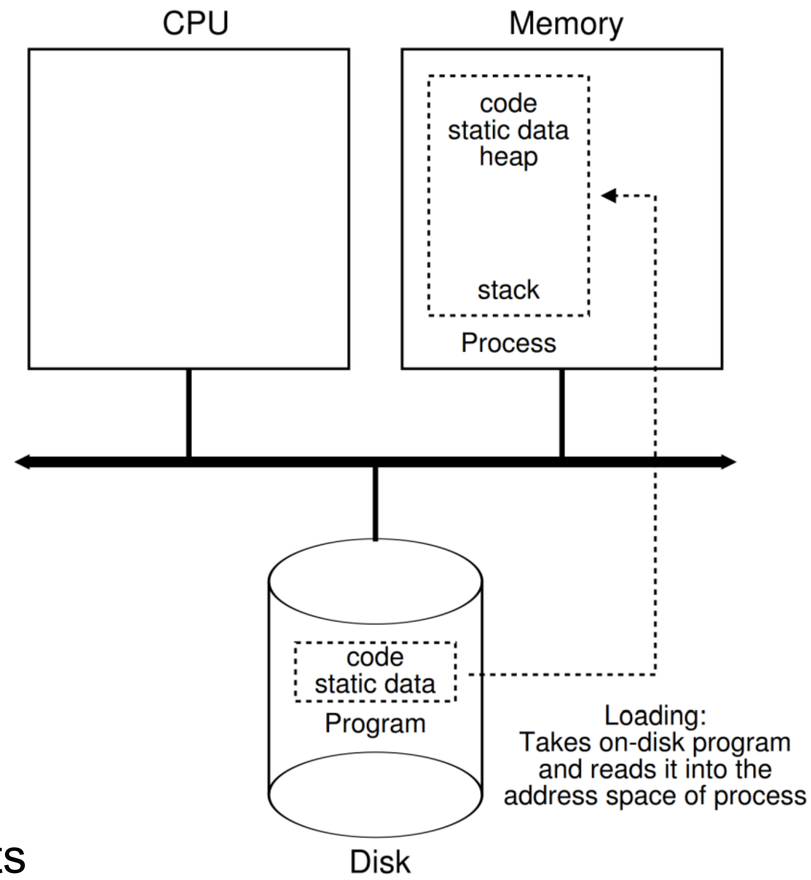
- **To understand what constitutes a process, we have to know its machine state**
  - The machine state that comprises a process are:
    1) Memory: instructions and data lie in memory (address space)
    2) Registers: many instructions explicitly read or update registers, such as program counter, stack pointer, and frame pointer
    3) Storage: I/O information that the process is currently accessing (e.g. file)

- **Modern OS has an interface by providing APIs:**
  - Create: create a new process to run a program
  - Destroy: halt a runaway process (interface to destroy processes forcefully)
  - Wait: wait for a process to stop running
  - Miscellaneous control: other controls (e.g. suspend a process and resume it)
  - Status: get some status information about a process

Process Creation: A Little More Detail (1)

1) The first step is to **load** its code and any static data into memory, into the address space of the process

- Programs initially reside on disk in an executable format
- Early OSes load eagerly all at once before running the program
- Modern OSes perform the load process lazily, by loading pieces of code or data only as they are needed during program execution (swapping, paging)



2) Some memory is allocated for the program's **run-time stack**

- The stack for local variable, function parameters, and return addresses
- The OS initializes the stack with arguments (i.e. `argc` and `argv` of `main()`)

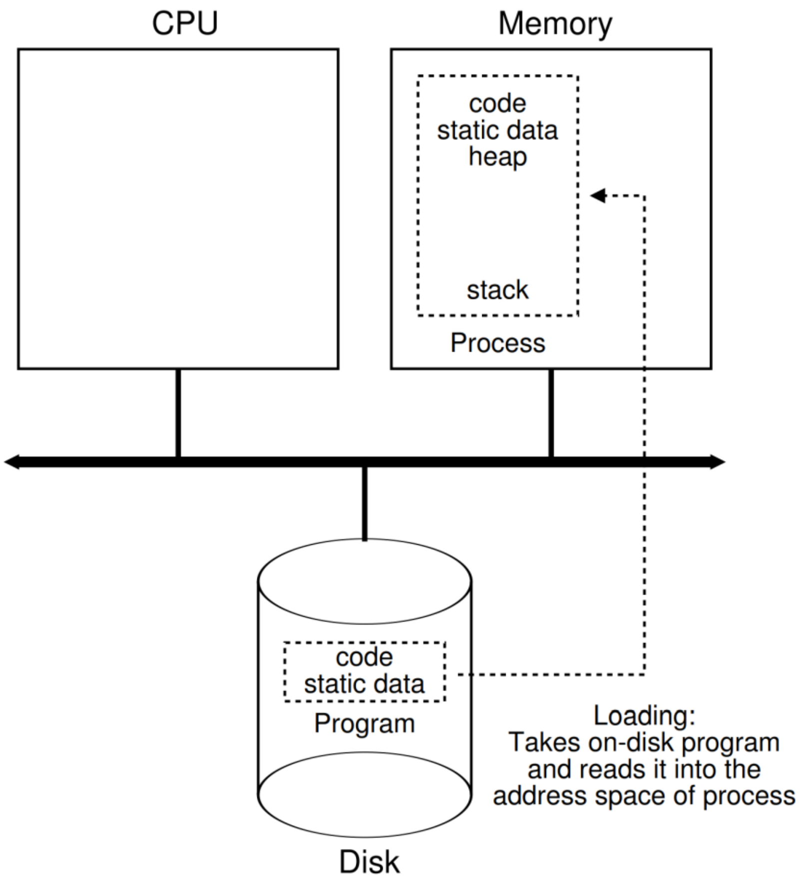
Process Creation: A Little More Detail (2)

3) The OS allocates some memory for the program's heap

- The heap for explicitly requested dynamically-allocated data (e.g. `malloc()`)

4) The OS does some other initialization tasks, related to I/O

- e.g.) each process in UNIX systems has three open file descriptors (standard input, output, error)

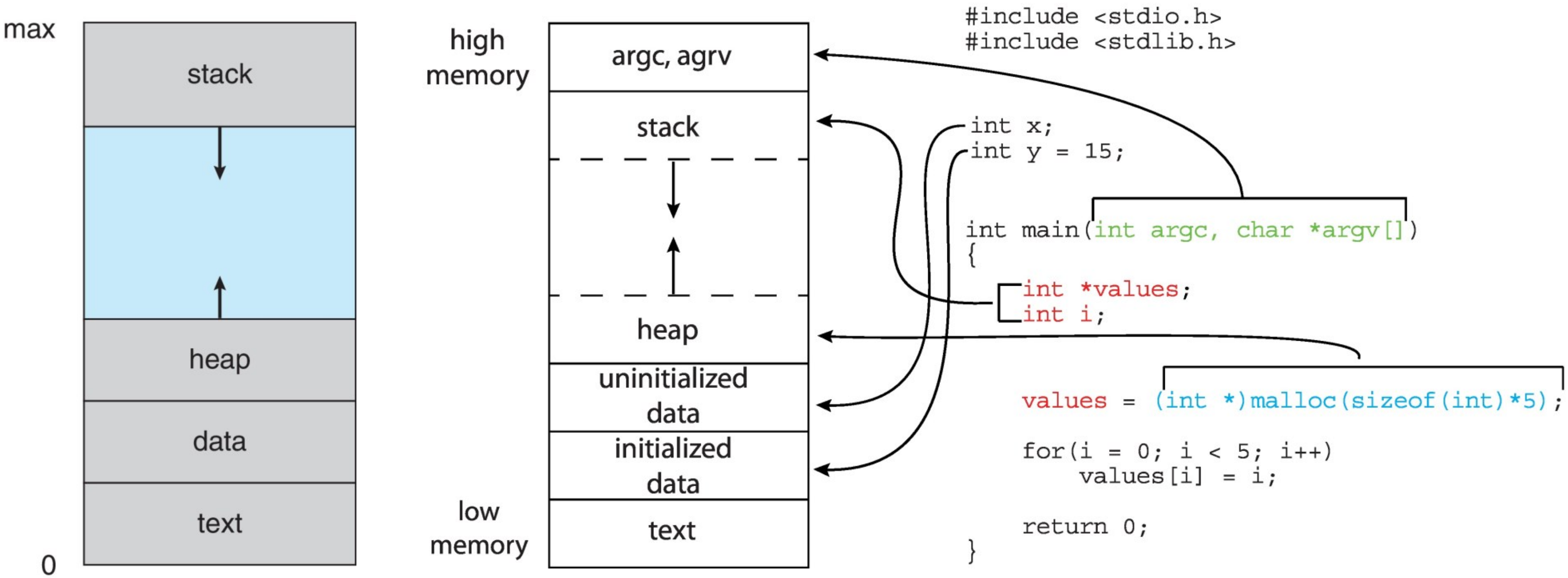


5) The OS starts the program running at the entry point, `main()`

- Then, the OS transfers control of the CPU to the newly-created process
- Thus, the program begins its execution

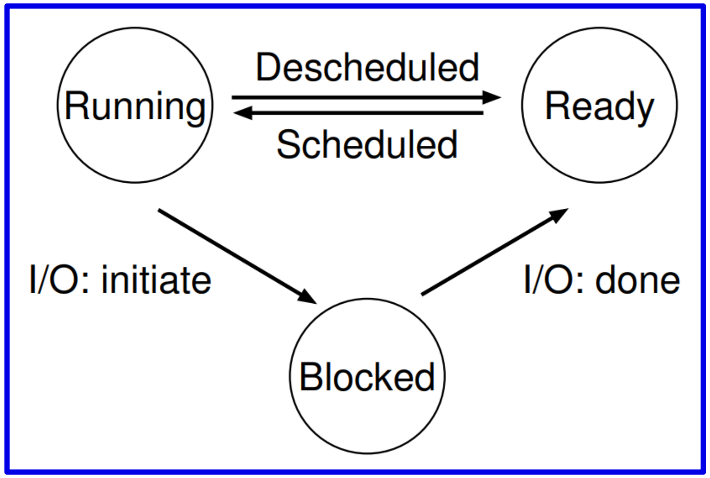
Memory Layout of a C Program

- **The memory layout of a process is typically divided into:**
  - 1) text (executable code), 2) data (global variable),
    3) heap (dynamically allocated memory), 4) stack (temporary data storage)
  - The data section is divided into (a) initialized data and (b) uninitialized data
  - A separate section is provided for the **`argc`** and **`argv`** parameters
- **Remember that each process has its own private address space**



Process States

- **In a simplified view, a process can be in one of three states:**
  1) **Running**: a process is running on a CPU (executing instructions)
  2) **Ready**: a process is ready to run, but the OS chosen not to run at this moment
  3) **Blocked**: a process has performed some kind of operation that makes it not ready to run until some other event takes place (e.g. when a process requests an I/O)



- **Examples**

Time	$Process_0$	$Process_1$	Notes
1	Running	Ready	
2	Running	Ready	
3	Running	Ready	
4	Running	Ready	$Process_0$ now done
5	–	Running	
6	–	Running	
7	–	Running	
8	–	Running	$Process_1$ now done

Tracing process states (CPU only)

Time	$Process_0$	$Process_1$	Notes
1	Running	Ready	
2	Running	Ready	
3	Running	Ready	$Process_0$ initiates I/O
4	Blocked	Running	$Process_0$ is blocked,
5	Blocked	Running	so $Process_1$ runs
6	Blocked	Running	
7	Ready	Running	I/O done
8	Ready	Running	$Process_1$ now done
9	Running	–	
10	Running	–	$Process_0$ now done

Tracing process states (CPU and I/O)

Data Structures

- **The OS has some key data structures that track various relevant pieces of information:**
  - Process list that includes all of the ready, blocked, and running processes
  - Register context to hold the register contents for a stopped process, which will be used for context switching (save them when stop → restore them when resume)
- **Each process is represented by a process control block (PCB)**
  - C-structure that contains all the information about a process

```
// the registers xv6 will save and restore
// to stop and subsequently restart a process
struct context {
  int eip;
  int esp;
  int ebx;
  int ecx;
  int edx;
  int esi;
  int edi;
  int ebp;
};

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

xv6's register context and process states

```
// the information xv6 tracks about each process
// 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 !zero, sleeping on chan
  int killed;                 // If !zero, has 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
};
```

xv6's PCB (process descriptor)