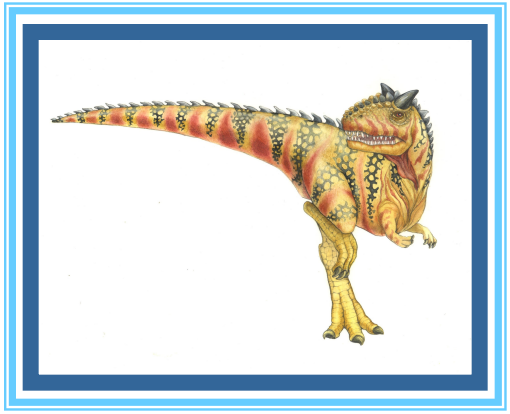
Spring 2022 COMP 3511
Review #2



Coverages

- Quick review on Chapters 2
- Process creation
- fork() examples
- Multi-threaded process

Quick review on Chapters 2

- To best utilize the fast CPU, modern OSes employ multiprogramming, which allows many jobs to be in memory at the same time, thus ensuring that the CPU always has a job to execute
- Multitasking can be considered an extension of multiprogramming wherein CPU scheduling (to be discussed in Chapter 5) can rapidly switch CPU between executing different processes, providing users with a fast response time, which is important for interactive types of jobs
- Operating systems provide mechanisms for protecting. Protection measures the control of the resource (hardware and software) access available in computer systems.
- Virtualization is a technology that allows an OS to run as an application within another OS. It involves abstracting hardware into several different execution environments – each referred as a virtual machine.
- The virtual machine or VM creates an illusion for multiple processes in that each process “thinks” that it runs on a dedicate CPU with its own memory.

Quick review on Chapters 2 (cont.)

- The three primary approaches for interacting with an operating system are (1) command interpreters (CLI or Shell), (2) graphical user interfaces, and (3) touchscreen interfaces
- **System calls** provide services made available by an operating system, where programmers use a system call's application programming interface (API) for accessing system-call services. The standard C library provides the system-call interface for UNIX and Linux
- Operating systems include a collection of **system programs** that provide utilities to users – so users can use the operating system services
- A linker combines several relocatable object modules into a single binary executable file. A loader loads the executable file into memory, where it becomes eligible to run on an available CPU
- An operating system is designed with specific goals in mind. These goals ultimately determine the operating system's **policies**. An operating system implements these policies through specific **mechanisms**

System Call and API

- System calls are programming interfaces to the services provided by the OS – not directly accessed by application programs
- Application Program Interface (API) specifies a set of functions that are available to application programmers, including the parameters passed to the function and return values it may expect
- The system call interface of a programming language serves as a link to system calls made available by the OS.
- There is a need for separating API and underlying system call:
  - Program portability by using API, in which API can remain the same across different OSes or different platforms
  - To hide the complex details in system calls from users

Operating System Structures

- A monolithic operating system has no structure; all functionality provided in a single, static binary file that runs in a single address space. Although such systems are difficult to modify, their primary benefit is efficiency
- A layered operating system is divided into a number of discrete layers, where the bottom layer is the hardware interface and the highest layer is the user interface. It provides modularity to ease coding and debugging, but it is generally not ideal for designing OS due to performance problems and difficulty in defining proper number of layers.
- The microkernel approach for designing operating systems uses a minimal kernel, which is extensible, portable, more secured and reliable; other OS services run as user-level applications, in which communication takes place via the message passing mechanism – a major overhead
- The loadable kernel module for designing OS provides services through modules that can be dynamically loaded and removed during runtime. It provides the most flexibility, and is widely used in contemporary OSes
- Many operating systems are constructed as hybrid systems using a combination of a monolithic kernel and loadable kernel modules.

Fundamental OS concepts

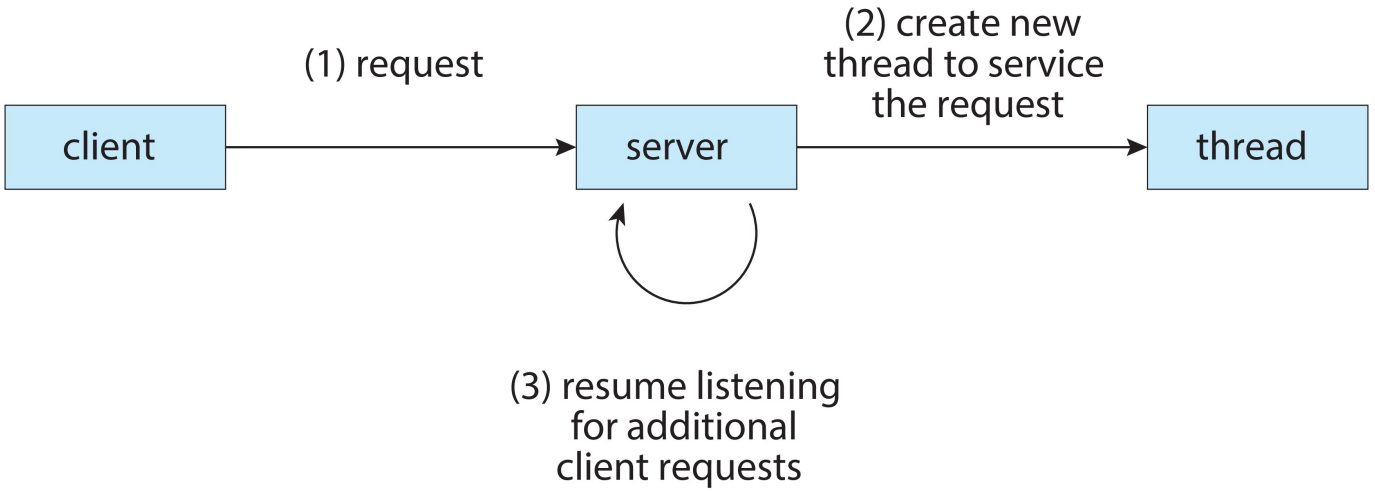
- Process
  - An instance of an executing program is a process consisting of an address space and one or more threads of control
- Thread
  - A single unique execution context: fully describes program state
  - It is presented by program counter, registers, execution flags, stack
- Address Space (with address translation)
  - Programs execute in its own address space that is distinct from the memory space of the physical machine
- Dual mode operation / Protection
  - Only the “system” has the ability to access certain resources
  - The OS and the hardware are protected from user programs
  - User programs are isolated from one another by proper access control

Process

- Process – An OS abstraction of a running program. A process can be described by its state: the contents of memory in its address space, the contents of CPU registers (including the program counter and stack pointer, among others), and information about I/O (such as open files to be read or written).
- The OS provides system calls for operations on processes, typically, including creation, termination, and other useful calls
- Processes exist in one of many different process states, including ready, running, and waiting waiting (or called blocked). Different events, e.g., scheduled or de-scheduled, or waiting for an I/O to complete transition a process from one of these states to another
- A process list – a kernel data structure OS manages to contain information about all processes currently in the system. Each entry is called a process control block (PCB) - an OS kernel data structure that contains all information about a specific process

Multithreaded Process

- Many modern applications are multithreaded - an application may be required to perform several similar tasks – for instance, a web server accepts client requests for retrieving web pages, which may have many (perhaps thousands of) clients concurrently accessing it
- Creating a separate process to service each request is time-consuming and resource-intensive, and each process essentially performs the same task - It is generally much more efficient to create one process containing multiple threads, and each thread listens and serves requests.



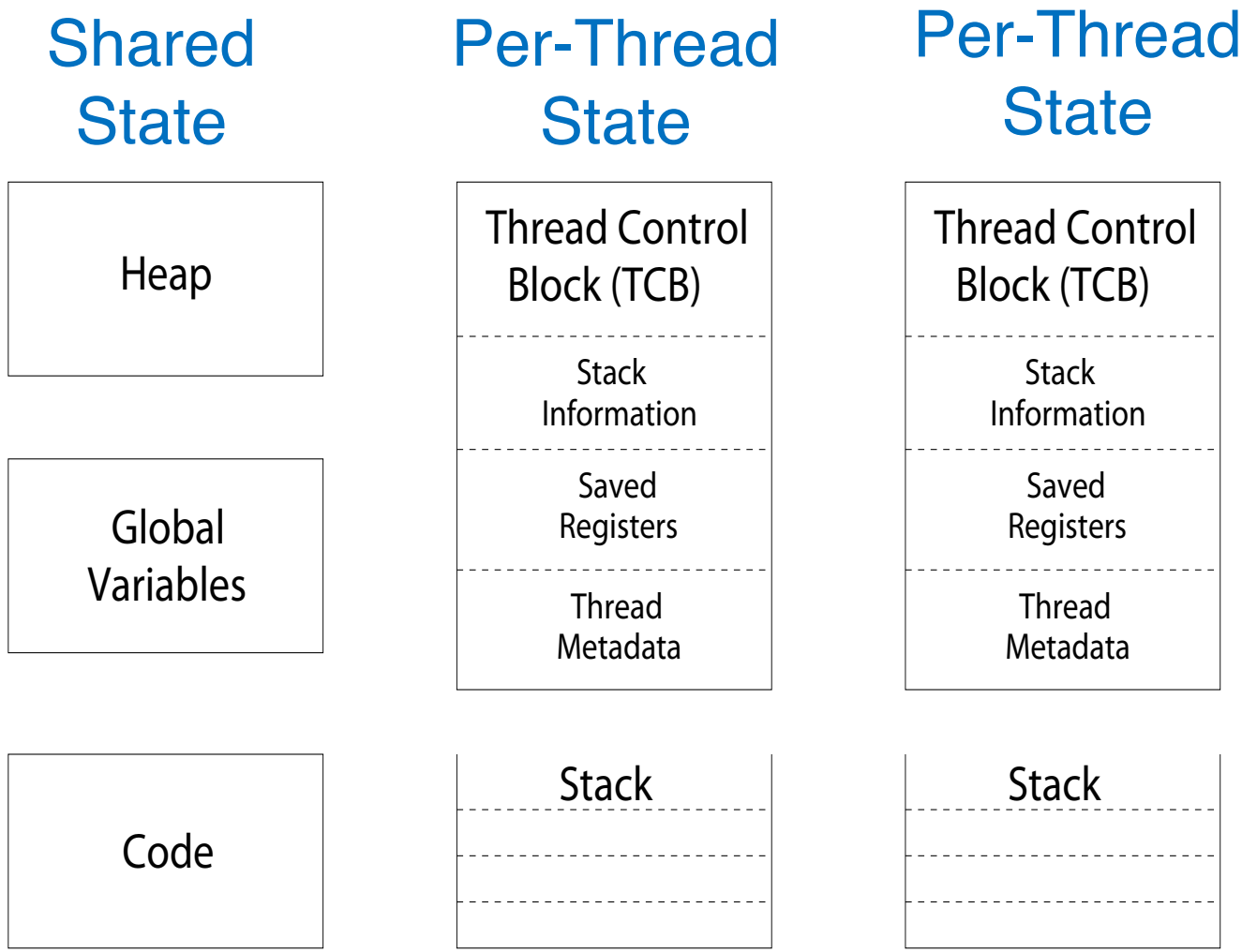
Why Multithreaded

- A single-threaded program or process runs sequentially, performing one operation followed by another
- There are two major reasons for multi-threaded process
- **Parallelism**: to take advantage of multicores or multi-processors - speed up process execution considerably by using each of the processors to perform a portion of the work
- **Overlap** I/O with other activities within a single program
- It is feasible to create multiple processes instead of a multi-threaded process. However, threads within a process share the address space and thus make it easy to share data naturally.
- Processes are natural choices for logically separate tasks where little sharing of data structures in memory is needed.

Threads within a Process

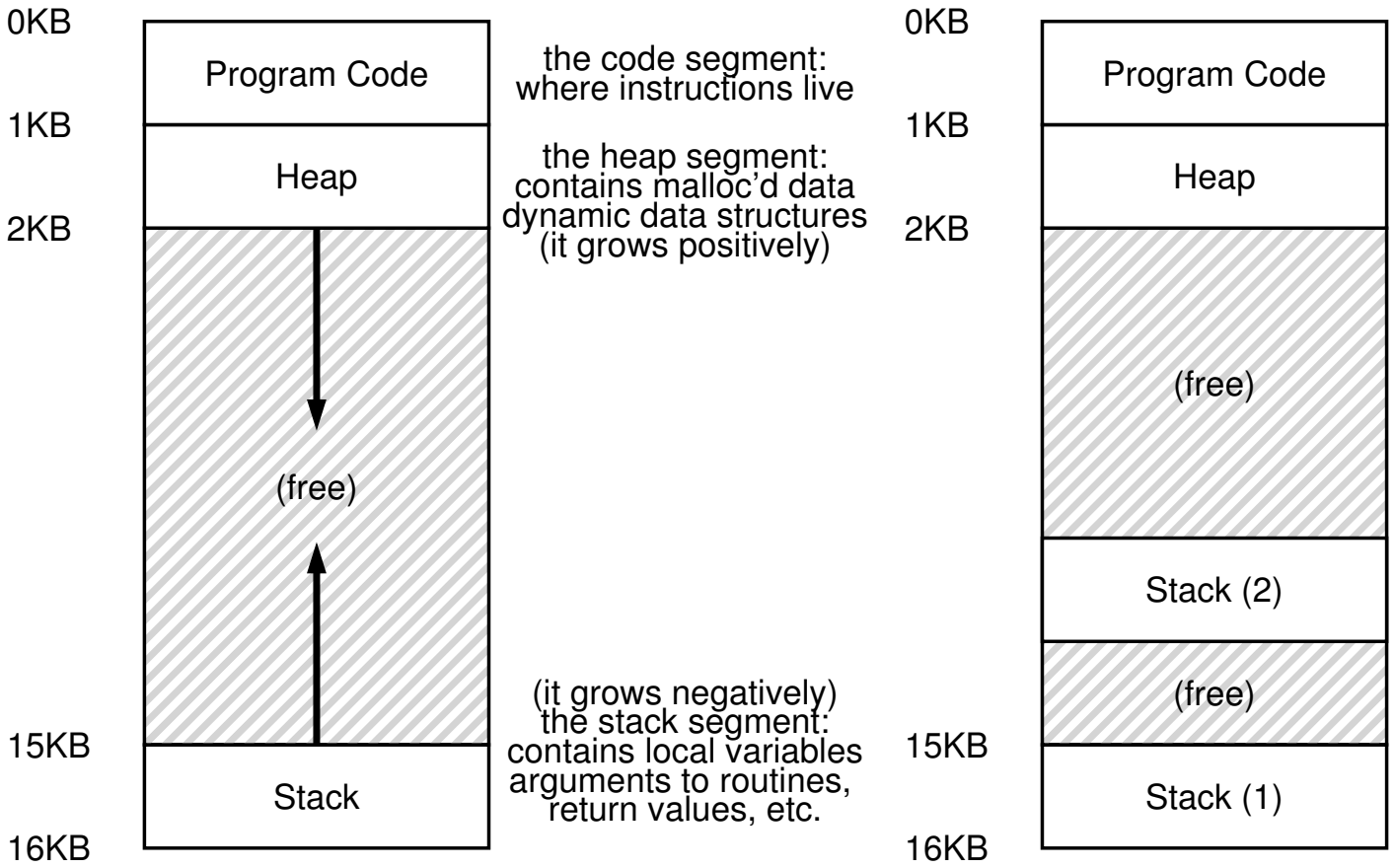
- If a process has a single or multiple thread(s) of execution – the execution context fully describes the state, i.e., the current activity of the thread
- A thread is represented by
  - Program counter, registers, execution flags, and stack
  - A thread is executing on a processor (CPU) when it is resident in the processor registers. PC register holds the address of executing instruction in the thread
- Threads within a process share code, data, I/O and files
  - On multicore systems, multiple threads of a process can run in parallel on different CPU cores
- Each Thread has a Thread Control Block (TCB) containing execution state (CPU registers, program counter, pointer to stack), scheduling info, accounting info, various pointers including the pointer to the process PCB that the thread belongs to

Shared vs. Per-Thread State



Thread Address Space

- Single-threaded and multi-threaded address spaces



Here, we assume the virtual memory address space is 16KB
In practice, the address space is much larger than 16KB

Thread Libraries

- Thread library provides programmers with API for creating and managing threads. Three main thread libraries are in use today :
  - POSIX Pthreads - either a user-level or a kernel-level library
  - Windows - kernel-level library available on Windows systems
  - Java - Since JVM is running on top of a host operating system, the Java thread API is generally implemented using a thread library available on the host system, i.e., Pthreads or Windows threads
- There are two primary ways of implementing a thread library
  - Library entirely in user space - invoking a function in the library results in a local function call in user space and not a system call
  - Kernel-level library supported by the OS - invoking a function in the API for the library typically results in a system call to the kernel
- For POSIX and Windows thread, data declared globally (outside of any function) - are shared among all threads belonging to the same process
- Java has no equivalent notion of global data, access to shared data must be explicitly arranged between threads

Process Creation

- **`fork()`** creates a new process, perhaps the strangest routine you have seen so far - the newly created process (child process) is an (almost) ***exact copy*** of the calling process (parent process). That means that it looks like there are two copies of the same program running, and both are about to return from the **`fork()`** system call
- The child isn't an exact copy. Specifically, although it has its own copy of the address space (i.e., its own private memory), its own registers, its own PC, and so forth, the value it returns to the caller of **`fork()`** is different - the parent receives the PID of the newly-created child, the child receives a return code of zero. This is essential to distinguish which process (parent or child) is executing the code after **`fork()`**
- When the child process is created, there are now two active processes in the system along with other processes – either the child or the parent might run before the other. The execution sequences are now **non-deterministic**, which is determined by CPU scheduling (to be discussed in Chapter 5). In another word, any process can not determine when it will be executed by CPU and for how long.

Process Creation (cont.)

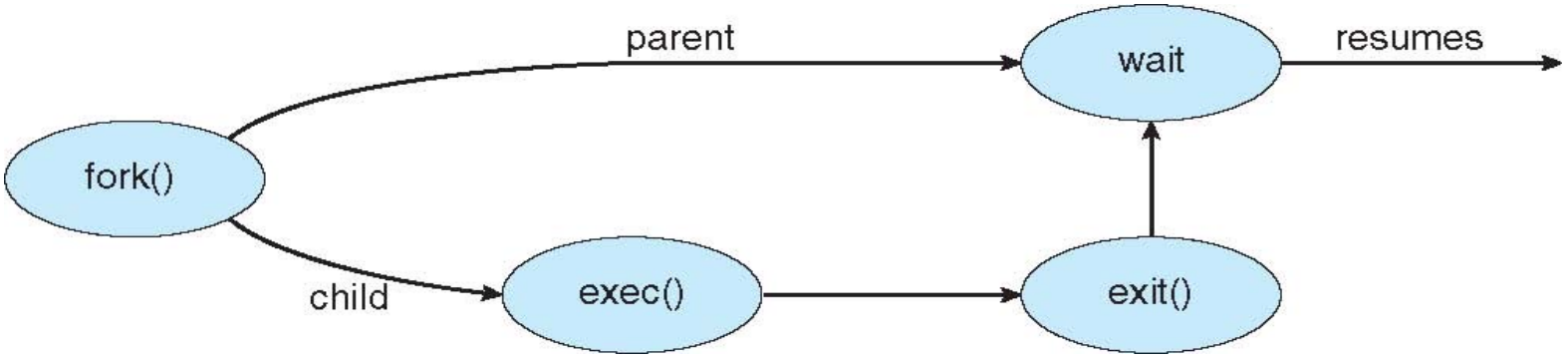
- http://pages.cs.wisc.edu/~remzi/OSTEP/cpu-api.pdf
- Figure 5.1 (p1.c) the simple fork example (with error handling)

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

int main(int argc, char *argv[]) {
        printf("hello world (pid: %d)\n", (int) getpid());
        int rc = fork();
        if ( rc < 0 ) {
                // fork failed
                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 (the original process)
                printf("hello, I am parent of %d (pid: %d)\n", rc, (int) getpid());
        }
        return 0;
}
```

Concurrent Execution

- **`fork()`** creates a new process, which duplicates entire address space of the parent, but with its own process ID and state
- Both processes continue execution at the next instruction after **`fork()`**
- **`exec()`** system call can be used after a **`fork()`** to replace the process's memory space with a new program (load a new program to execute)
- Parent process can call **`wait()`** system call to wait for a child to finish



`fork()` Example

- http://pages.cs.wisc.edu/~remzi/OSTEP/cpu-api.pdf
- Figure 5.3 (p3.c), example with `fork(), wait() and exec*()`…
  - In the child process, this example demonstrates the usage of execvp to run a command to count number of words in a text file (i.e. p3.c)
  - Note: exec*() are special functions. They will never return!

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <sys/wait.h>

int main(int argc, char *argv[]) {
        printf("hello world (pid:%d)\n", (int) getpid());
        int rc = fork();
        if ( rc < 0 ) {         // fork failed
                fprintf(stderr, "fork failed\n");
                exit(1);
        } else if ( rc == 0 ) { // child (new process)
                printf("hello, I am child (pid: %d)\n", (int) getpid());
                char *myargs[3];
                myargs[0] = strdup("wc");    // program: "wc" (word count)
                myargs[1] = strdup("p3.c");  // argument: file to count
                myargs[2] = NULL;            // NULL is used to mark end of array
                execvp(myargs[0], myargs);   // runs "wc" using execvp
                printf("this should't print out"); //  note: exec* will never return!
        } else { // parent
                int rc_wait = wait(NULL);
                printf("hello, I am parent of %d (rc_wait: %d) (pid: %d)\n",
                        rc, rc_wait, (int) getpid());
        }
        return 0;
}
```

fork() Example 1

- Assume that necessary header files are included
- Consider the following code segments, what is the total number of processes? Please elaborate

```
int main() {
    int i=0;
    for (i=0;i<3;i++)
        fork();
    return 0;
}
```

equivalent

```
int main() {
    fork();
    fork();
    fork();
    return 0;
}
```

Answer:

8 processes

There are 2 processes after each fork() and we have 3 consecutive fork(), so the answer is $2^3 = 8$

fork() Example 2

- Assume that necessary header files are included
- Consider the following code segments, what is the total number of processes? Please elaborate

```
int main()  {
    int i,j;
    for (i=0; i<2; i++) {
        fork();
        for (j=0; j<3; j++)
            if (!fork()) {
                // do nothing
            }
    }
    return 0;
}
```

256 processes

Each fork() will create 2 processes, and we have 8 consecutive fork() function calls, thus the total number of processes is $2^8 = 256$

equivalent

```
int main()
{
    fork();
    if (!fork()); // do nothing
    if (!fork()); // do nothing
    if (!fork()); // do nothing
    fork();
    if (!fork()); // do nothing
    if (!fork()); // do nothing
    if (!fork()); // do nothing
    return 0;
}
```

fork() Example 3

- Assume that necessary header files are included
- Consider the following code segments, what is the total number of processes? Please elaborate (Hint: You can write your answer in terms of x^y processes)

```
int main() {
    int i=0;
    for(i=0; i<10; i++) {
        if(fork()) {
            fork();
            fork();
        }
        else
            fork();
    }
    return 0;
}
```

6^{10} processes

In each loop, there are 6 processes from the initial one. Since there are 10 loops, the program will have 6^{10} processes