Operating System Principles - 1

Shyan-Ming Yuan
CS Department, NYCU
smyuan@gmail.com

Introduction - 1

- Goal: Understand the basic concepts of an OS.
  - OS history
  - the design principle of separating mechanism from policy
- Some terminologies of early Operating Systems:
  - Batch systems (1956): OSs would run a program to completion and then load and run the next program.
  - Multiprogramming (1960): OSs keep several programs in memory at once and switch between them. The primary goals are better system throughput and resource utilization.
  - Time-sharing (1961): multiprogramming with preemption, where the system may stop one process from executing and starts another. A single physical CPU is shared by multiple programs to create the illusion that each program is executed by a dedicated, slower virtual CPU.

Introduction - 2

- Some terminologies of Modern Operating Systems:
  - Portable OS: an OS that is not tied to a specific hardware platform.
    - UNIX (1970) was one of the early portable operating systems.
  - Microkernel OS: an OS that only provides the bare essential mechanisms that are necessary to interact with the hardware and manage threads and memory.
    - Higher-level OS functions, such as managing file systems, the network stack, or scheduling policies, are delegated to user-level processes that communicate with the microkernel via interprocess communication mechanisms (usually messages).
    - Mach (1985), MINIX (1987) are two early microkernel OSs.

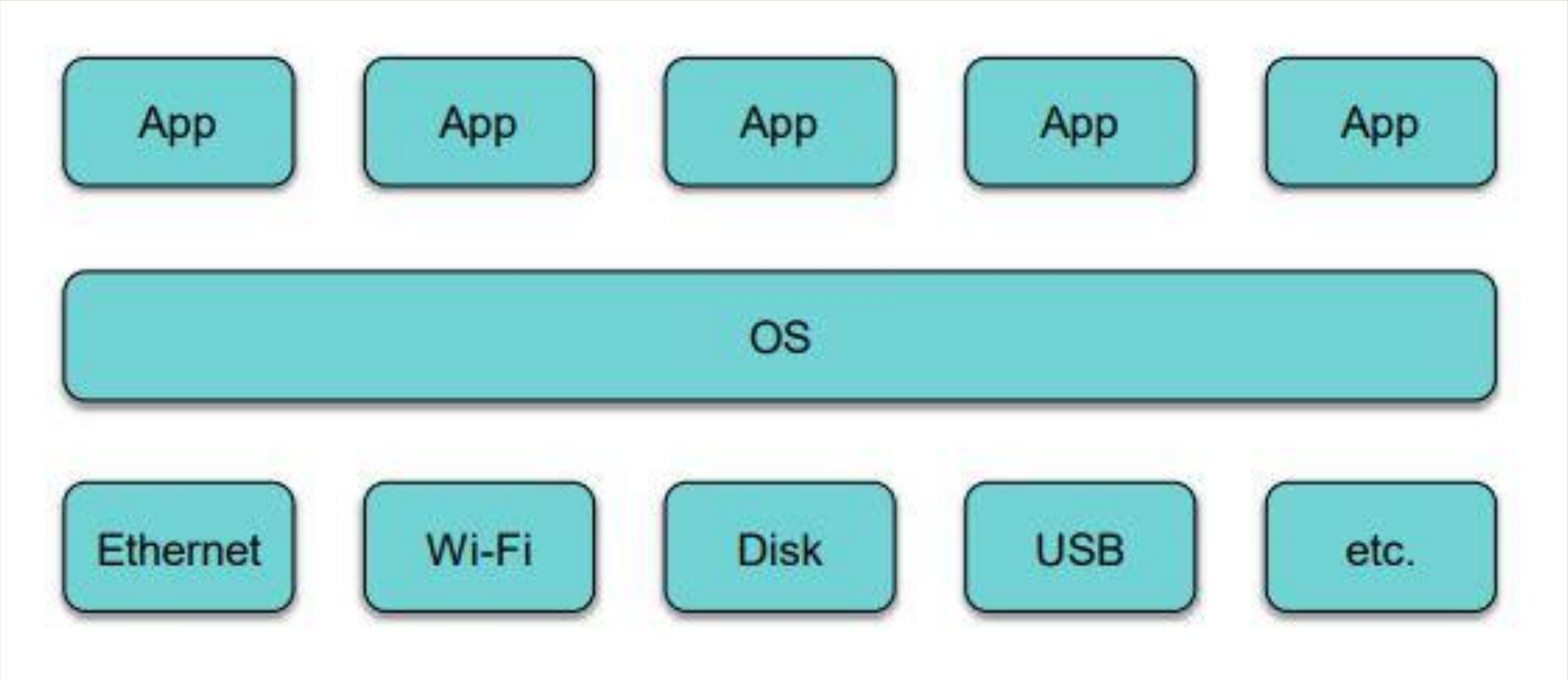
Introduction - 3

- Personal Computer Operating Systems:
  - CP/M (1971~1975): dominant OS for 8080 family of machines
  - IBM PC and Microsoft DOS (1980s)
  - Microsoft Windows series (1990~)
    - Windows 3.0, Windows NT, Windows 95, ….
  - Open Source OSs (1990~)
    - Linux, FreeBSD, NetBSD, OpenBSD
- Mobile Computer Operating Systems (2000~):
  - iOS, Android, BlackBerry OS, Windows Mobile

What is an operating system?

- The first program runs when a computer starts
  - The program that lets users to run other programs
  - The program that provides controlled access to resources:
    - CPU
    - Memory
    - Display, keyboard, mouse
    - Persistent storage
    - Network
    - …..

The Operating System



The Design Principle of OS

- A mechanism is the presentation of a software abstraction:
  - the functional interface.
  - It defines how to do something.
- A policy defines how that mechanism behaves.
  - e.g., enforce permissions, time limits, or goals.
  - Driving a car is defining a policy and the interfaces provided by a car are mechanisms.
- A Good OS design principle is to
  - keep mechanisms and policies separate.

Booting an Operating System

- Boot loader: is a small program that is run at boot time that loads the operating system.
- Boot loaders are sometimes broken into several stages.
  - A multi-stage boot loader starts off with a simple boot loader that then loads a more sophisticated boot loader that then loads the OS.
    - This cascaded boot sequence is called chain loading.
  - Two well known chain loading mechanisms are
    - Basic Input/Output System (BIOS) based booting
    - Unified Extensible Firmware Interface (UEFI) based booting
- Upon loading the OS, the boot loader transfers control to the OS.
  - The OS will initialize itself and load various modules as needed (for example, device drivers and various file systems)

OS Essential Concepts

- An operating system contains of the following functionalities.
  - It is a program that loads and runs other programs.
  - It provides programs with a level of abstraction so they don’t have to deal with the details of accessing hardware.
  - It manages access to resource, including
    - the CPU (via the scheduler),
    - memory (via the memory management unit),
    - persistent files (via the file system),
    - a communications network (via sockets and IP drivers), and
    - devices (via device drivers).

Kernel Mode vs User Mode

- The operating system (the microkernel or kernel) runs in kernel mode (also called privileged, supervisor, or system mode).
  - In this mode, the processor can execute privileged instructions:
    - define interrupt vectors,
    - enable or disable system interrupts,
    - interact with I/O ports,
    - set timers and
    - manipulate memory mappings.
- Programs other than the kernel run in user mode and do not have privileges to execute these instructions.

Mode Switch

- A processor running in user mode can switch to kernel mode by executing a trap instruction, also known as a software interrupt.
  - Some processors also offer explicit system call instructions.
    - This is a faster mechanism since it does not need to read the branch address from an interrupt vector table (in memory) but keeps that address in a CPU register.
- Both approaches can switch the processor to kernel mode.
  - It saves the current program counter on the stack, and transfers execution to a predefined address for that trap.
  - Control is switched back to user mode and to the location right after the trap by executing a return from exception instruction.

System Calls

- User processes can interact with the OS via system calls.
- A system call uses the trap mechanism to switch control to OS code running in kernel mode.
- A system call does the following:
  - Set the system call number
  - Save parameters
  - Issue the trap (jump to kernel mode)
    - OS gets control
    - Saves registers, does the requested work
    - Return from exception (back to user mode)
  - Retrieve results and return them to the calling function
- System call interfaces are encapsulated as library functions

How Can the OS Get Control?

- When the OS decides that a process has been executing long enough, it may preempt that process to run another process.
- To allow the OS to get control at regular intervals, a programmable interval timer can be configured to generate periodic hardware interrupts, for example every 10 milliseconds.
- It is crucial for:
  - Preempting a running process to give someone else a chance to run
    - forcing a context switch or killing the running process
  - Giving the OS a chance to poll hardware status
  - Bookkeeping OS statistics

Timer Interrupt and Context Switch

- On timer interrupt, control is transferred to an interrupt handler in the kernel and the processor is switched to kernel mode.
  - When the OS is ready to return back, it issues a return from exception instruction.
- If the OS decides it is time to replace the currently executing process with another process, it will save the current process' context and restore the saved context of another process.
  - The context includes the values of a processor's registers, program counter, stack pointer, and memory mappings.
- Saving the context of one process and restoring that of another is called a context switch.

I/O Devices

- The OS is responsible for managing system devices.
- Three typical categories of devices are:
  - Character devices: mice, keyboard, audio, scanner, etc.
    - Byte streams.
  - Block devices: disk drives, flash memory
    - Persistent storage with addressable blocks (suitable for caching)
      - This cache of frequently-used blocks of data is called the buffer cache.
    - Basically, any device that can be used to hold a file system is a block device.
  - Network devices: Ethernet & wireless networks
    - Packet-based I/O.

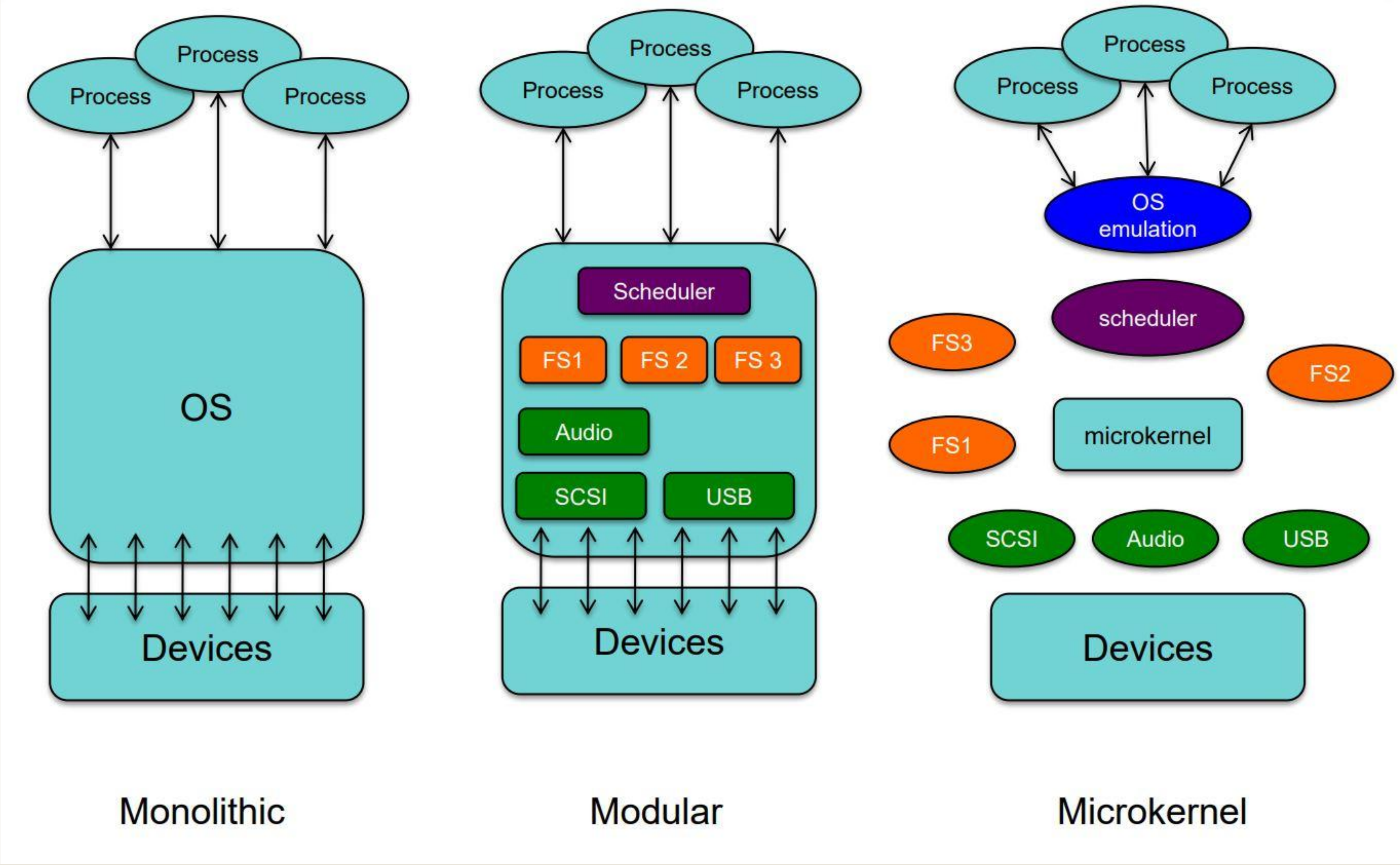
Interacting with Devices

- Devices have command (device) registers for the following OPs:
  - Transmit, receive, data ready, read, write, seek, status
- Memory mapped I/O
  - Map device registers into memory
  - Standard memory load/store instructions can be used to interact with the device
- When is the device ready?
  - Polling
    - Wait for device to be ready
    - To avoid busy loop, check status in each clock interrupt
  - Interrupts from the device
    - Interrupt when device has data or when the device is done transmitting
    - No checking needed – but extra context switch may be costly

Moving Data to/from Devices

- Programmed I/O (PIO):
  - Data can be transferred between the device and system via software that reads/writes the device's memory.
    - Use memory-mapped device registers
    - The processor is responsible for transferring data to/from the device by writing/reading these registers
- Direct Memory Access (DMA):
  - Allow the device to access system memory directly.
    - The device may have access to the system's memory bus and can use direct memory access (DMA) to transfer data to/from system memory without using the processor.

OS Structures



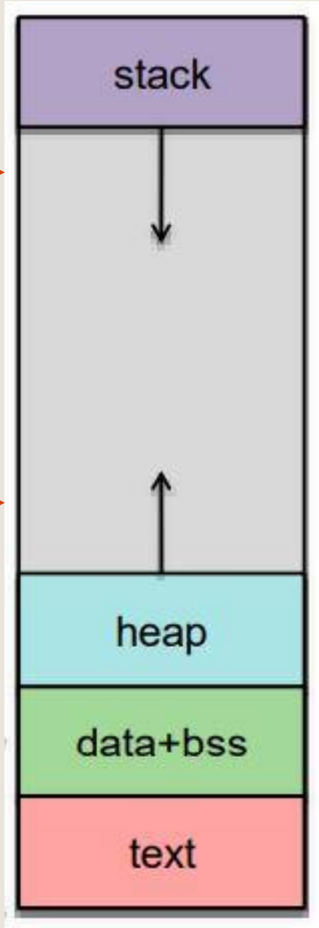
Process

- A program is a file that contains of code and static data.
- A process is a program in execution.
  - Each process has its own address space. It comprises the state of the processor's registers and the memory map for the program.
  - The memory map contains several regions. These regions are:
    - Text: compiled program (the machine instructions)
    - Data: initialized static and global data
    - Bss: uninitialized static data that was defined in the program
      - global uninitialized strings, numbers, structures, etc.
    - Heap: dynamically allocated memory (obtained in run time)
    - Stack: the subroutine call stack
      - return addresses, local variables, temporary data, saved registers, etc.

Memory Map

Stack expanded automatically

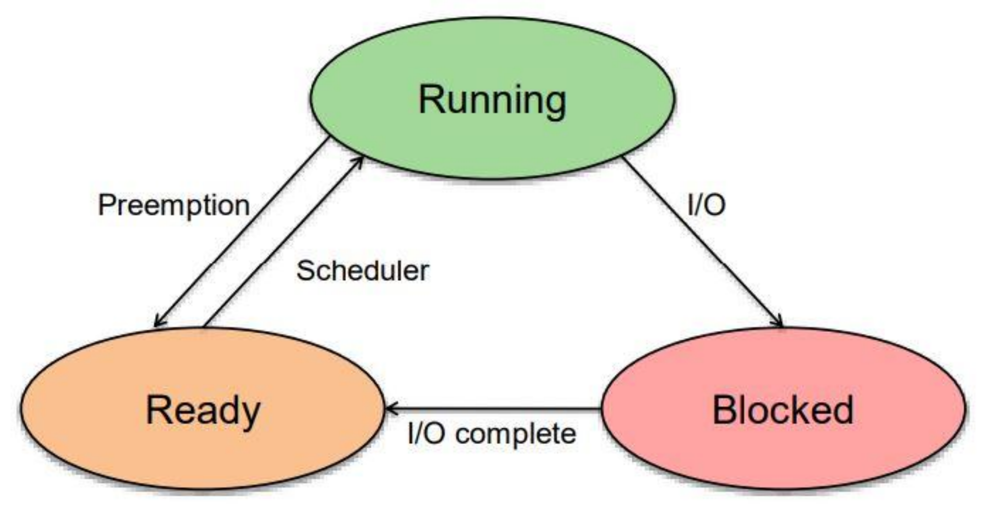
Heap can grow via a system call that requests more memory



Text (code) & initialized data come from the stored program

Process States

- A process may be in one of three states:
  - running: the process is currently executing code on the CPU
  - ready: the process is not currently executing code but is ready to do so if the OS would give it the chance
  - blocked (or waiting): the process is waiting for some event to occur (such as a requested I/O operation to complete) and will not ready to execute instructions until this event is ready



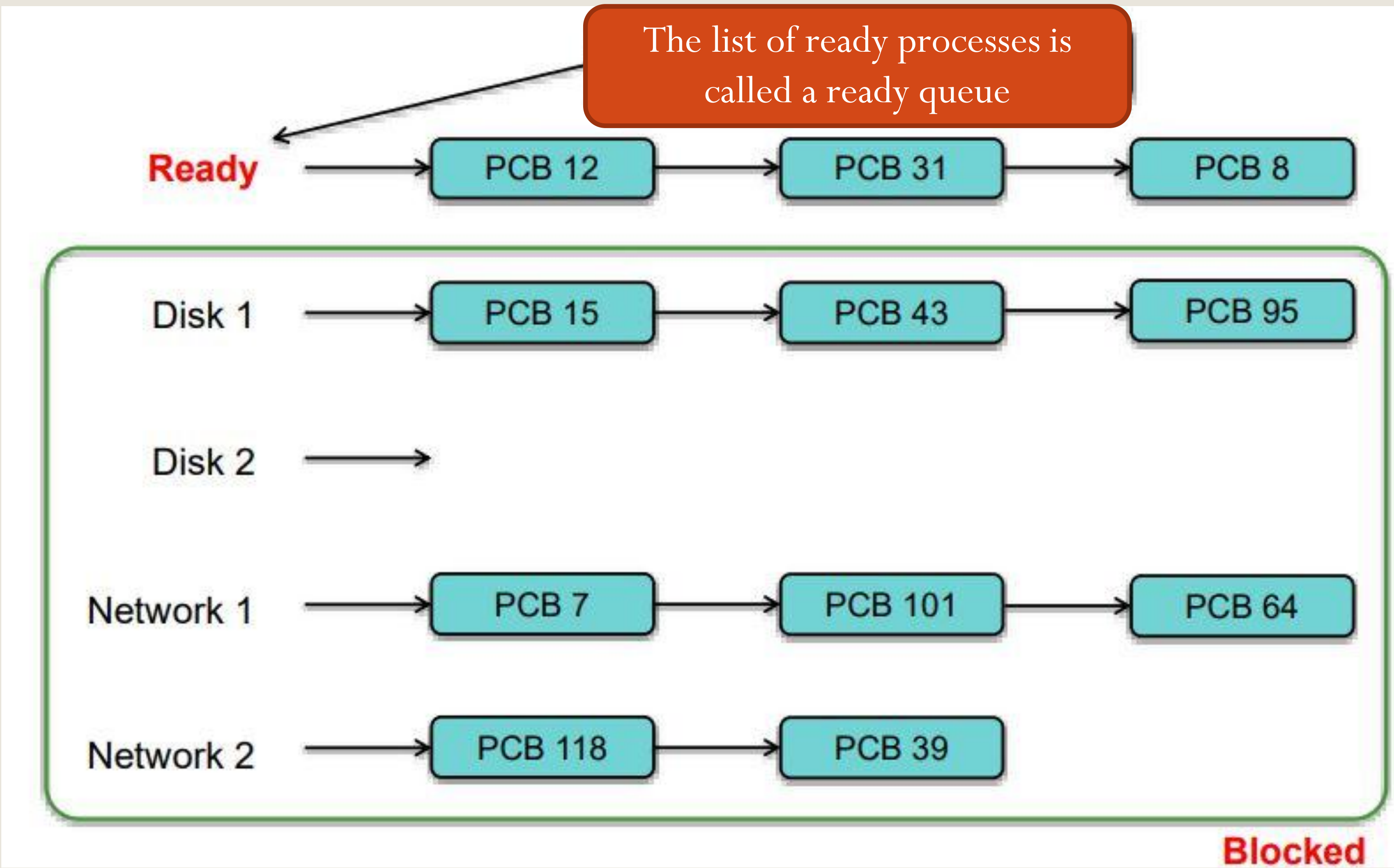
Scheduler and PCB

- The process scheduler of an OS is responsible for moving processes between the ready and running states.
  - An OS can save the context of a running process and restore the context of another ready process to run is called a preemptive multitasking system.
    - Most current operating systems are preemptive multitasking.
  - Systems that allow program to run until it terminates or blocks on I/O are called nonpreemptive.
- An OS keeps track of all processes via a list.
  - Each element in the list points to a data structure, called a process control block (PCB), that contains information about one process.
  - The PCB stores all relevant information about the process.

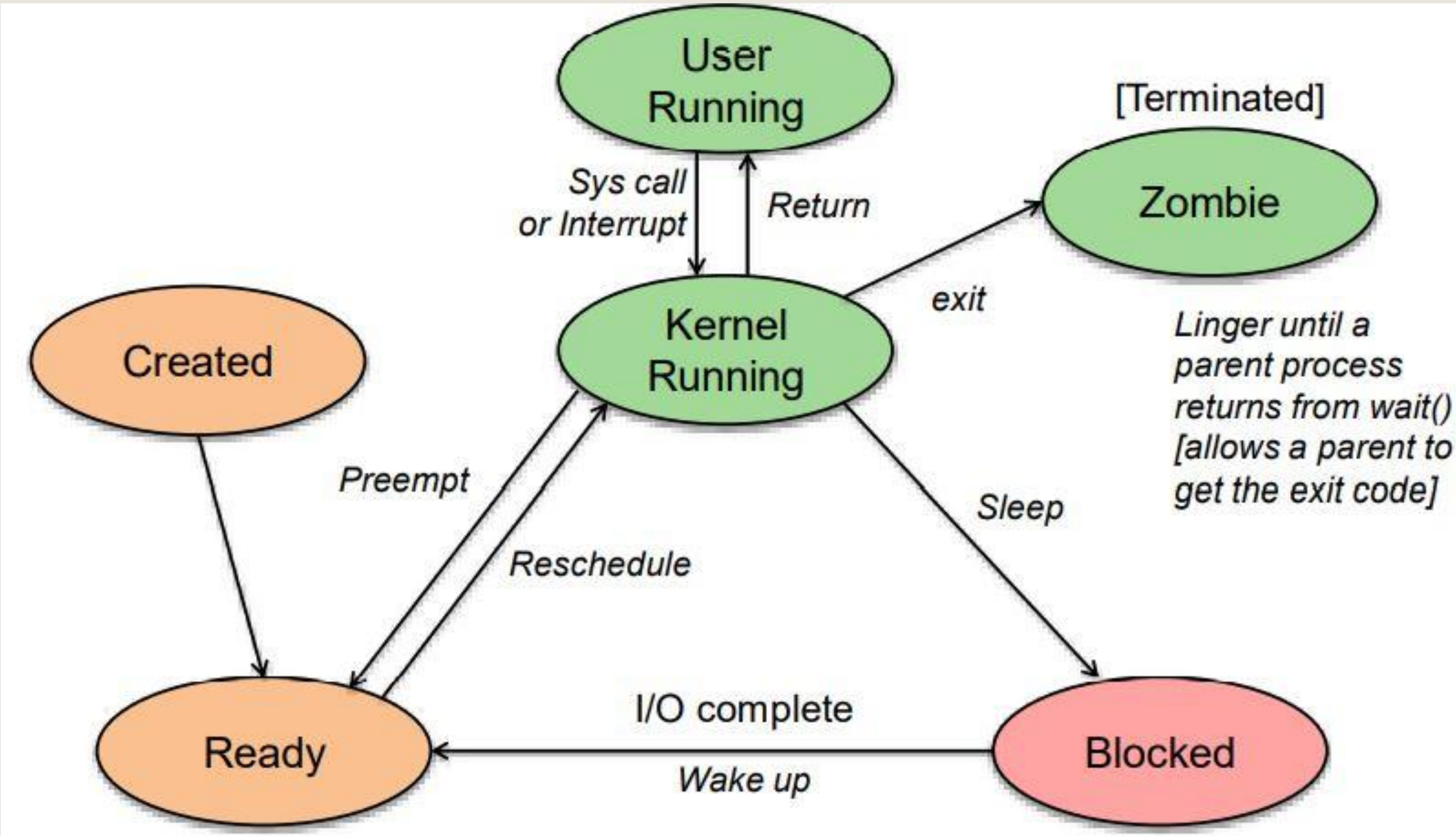
The Process Control Block

- A Process Control Block (PCB) may contain:
  - Process ID (and process group ID)
  - Machine state (registers, program counter, stack pointer)
  - Parent & list of children
  - Process state (ready, running, blocked)
  - Memory map
  - Open file descriptors
  - Owner (user ID) – determine access & signaling privileges
  - Event descriptor if the process is blocked
  - Signals that have not yet been handled
  - Policy items: Scheduling parameters, memory limits, etc.
  - Timers for accounting (time & resource utilization)

Processes on Ready & Blocked Queues



Process States: a more detail



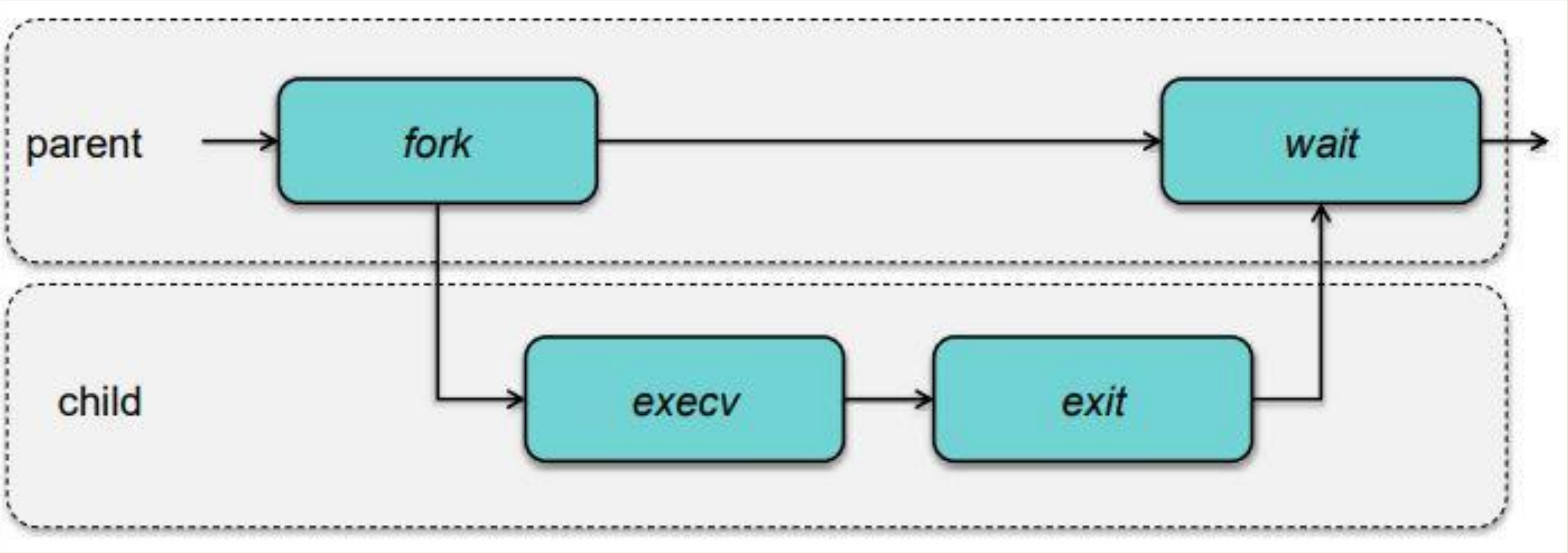
The POSIX

- POSIX stands for portable operating system interface of UNIX.
  - It was first released by IEEE in 1988. Latest version was in 2017.
  - It defines the application programming interface (API), along with command line shells and utility interfaces, for software compatibility with variants of Unix and other operating systems.
- Most modern OSs compliant to POSIX or at least partly support.
  - POSIX-certified OSs include:
    - AIX, HP-UX, macOS, IRIX, Solaris, etc..
  - Mostly POSIX-compliant include:
    - Android, FreeBSD, Linux, MINIX, NetBSD, VMware ESXi, etc.
  - There are all kinds of POSIX compliant toolkits for Microsoft Windows
    - Windows Subsystem for Linux (WSL) by Ubuntu
    - Windows C Runtime Library and Windows Sockets API, by Microsoft

Process Management in POSIX

- Fork() creates a new process.
- Execve() loads a new program to the current process.
- Exit() tells the operating system to terminate the current process.
- Wait() allows a parent process to wait for and detect the termination of child processes.
- Signal() allows a process to detect signals.
- Kill() sends a signal to a specified process(es).

Parent & child processes

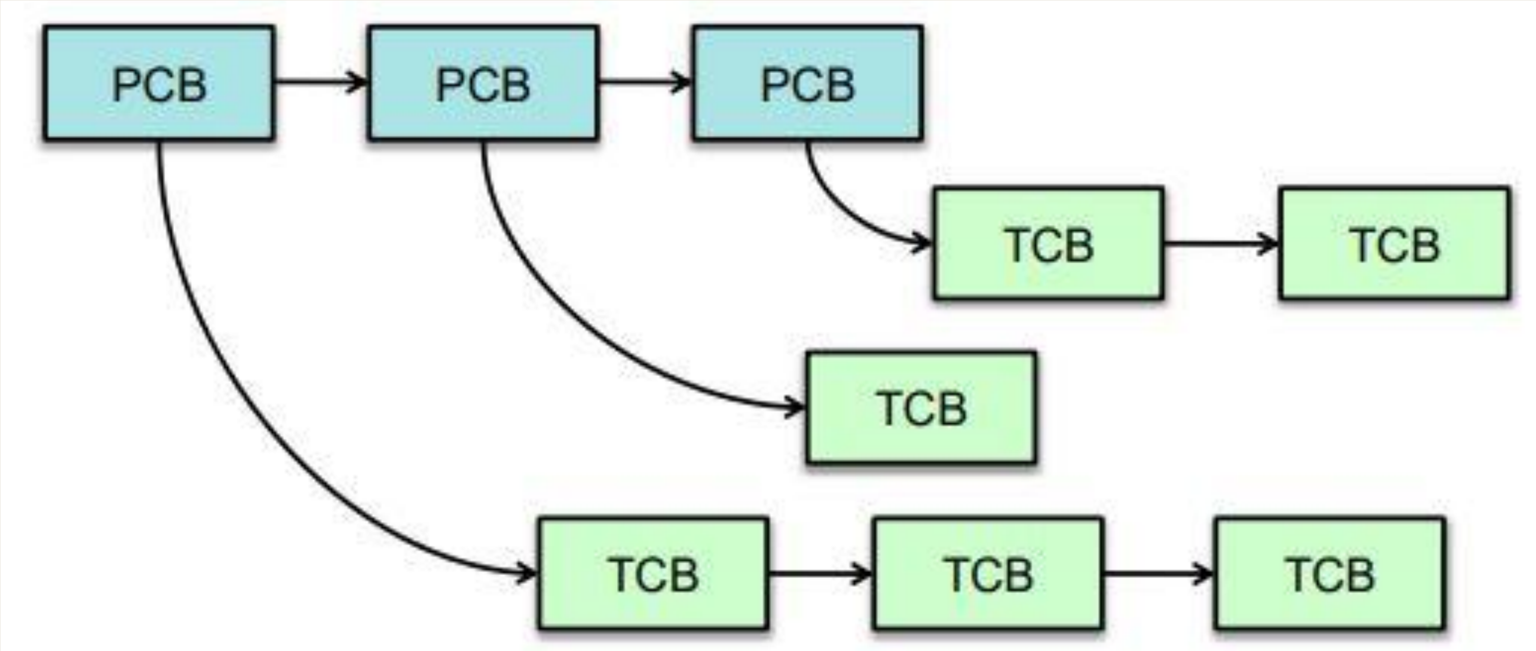


Thread

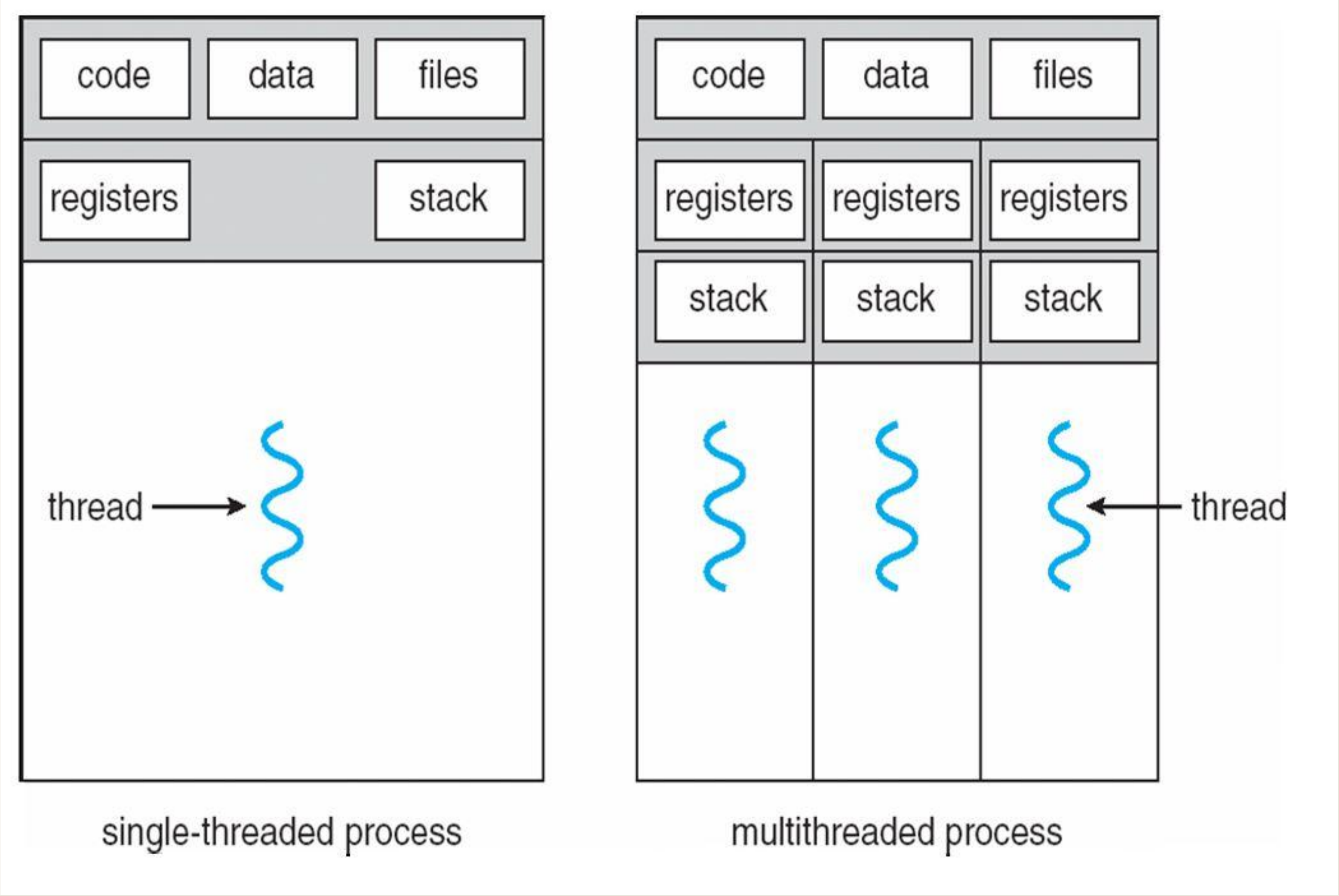
- A thread can be thought of as the part of a process that is related to the execution flow.
- A process may have one or more threads of execution.
  - A process with more than one thread is said to be multithreaded.
- A thread-aware OS keeps track of processes via a list of PCBs.
  - Each PCB keeps a list of thread control blocks (TCBs) for that process.
  - A TCB keeps only thread-specific information
    - registers, program counter, stack pointer, priority, etc..
  - All other information that is shared among threads in a process is stored within the PCB.

PCB & TCB

- Each PCB contains one or more TCBs:
  - Thread ID
  - Saved CPU registers
  - Other per-thread info (signal mask, scheduling parameters)



Single and Multithreaded Processes



Thread Sharing

- Threads share:
  - Text segment (instructions)
  - Data segment (static and global data)
  - BSS segment (uninitialized data)
  - Open file descriptors
  - Signals
  - Current working directory
  - User and group IDs

- Threads do not share:
  - Thread ID
  - Saved CPU registers,
  - Stack pointer
  - Program counter
  - Stack (local variables, temporary variables, return addresses)
  - Signal mask
  - Priority (scheduling information)

Advantage of Threads

- Threads are more efficient
  - Much less overhead to create: no need to create new copy of memory space, file descriptors, etc.
- Sharing memory is easy (automatic)
  - No need to figure out inter-process communication mechanisms
- Take advantage of multiple CPUs – just like processes
  - Program scales with increasing # of CPUs
  - Take advantage of multiple cores

Thread Implementations

- Kernel-level threads
  - Threads supported by operating system
  - OS handles scheduling, creation, synchronization
- User-level threads
  - Library with code for creation, termination, scheduling of threads
  - Kernel sees one execution context: one process
  - May or may not be preemptive
- Hybrid threading models: use user-level thread library on top of kernel threads.
  - 1:1 – kernel threads only (1 user thread = 1 kernel thread)
  - N:1 – user threads only (N user threads on 1 kernel thread)
  - N:M – hybrid threading (N user threads on M kernel threads, N>M)

User-Level Threads

- Advantages:
  - Low-cost: user level operations without switching to the kernel mode
  - Scheduling algorithms can be replaced easily & customized
    - The thread library can have its own thread scheduling algorithm that is optimized to a specific job
  - Greater portability
- Disadvantages:
  - If a thread is blocked, all threads for the process are blocked
    - Unless asynchronous I/O calls are supported
  - Cannot take advantage of multiprocessors

POSIX Threads: pthreads

- POSIX.1c, Threads extensions (IEEE Std 1003.1c-1995)
  - Defines API for managing threads
    - Linux, Solaris, Mac OS X, NetBSD, FreeBSD all support pthreads
    - Microsoft Windows: pthread API library on top of Win32 API
- Thread life cycle management APIs include:
  - pthread_attr_init(): initializes thread attributes
  - pthread_create(): creates a new thread
  - pthread_join(): waits for termination of other thread in the same process
  - pthread_exit(): terminates the calling thread itself
  - pthread_cancel(): terminates a target thread
- All pthread functions must be compiled and linked with

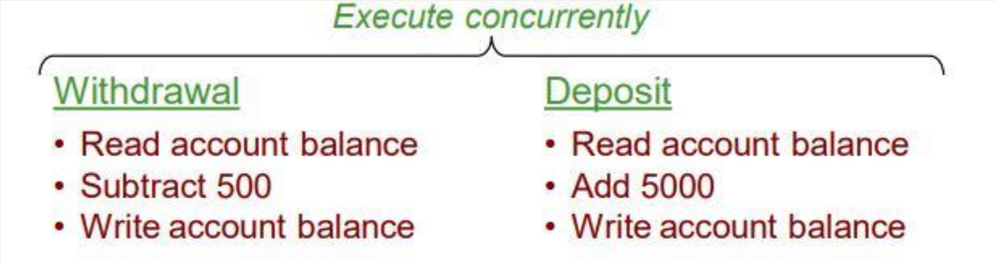
  -pthread option.

Concurrent Threads / Processes

- Threads (or processes) are concurrent if they exist at the same time.
  - Concurrent threads (or processes) can be either independent or cooperating.
  - Independent threads (or processes) have no dependency on each other.
    - It does not affect a thread whether another thread (or process) exists or not.
  - Cooperating threads (or processes) can affect and be affected by the executions of other cooperating threads (processes.)
    - Cooperating threads (or processes) need to interact with each other through some synchronization mechanisms so that their relative order of execution can be guaranteed.

Race Condition

- Cooperating threads may access shared data simultaneously.
  - If the consistent result depends on some specific data access orders, the race condition may occur.
  - Data consistency is maintained by synchronization mechanisms to ensure orderly execution of cooperating threads (processes.)
- A race condition example
  - Assume that your current bank balance is $1,000.
  - When withdraw $500 from an ATM machine while a $5,000 direct deposit is coming in at the same time.

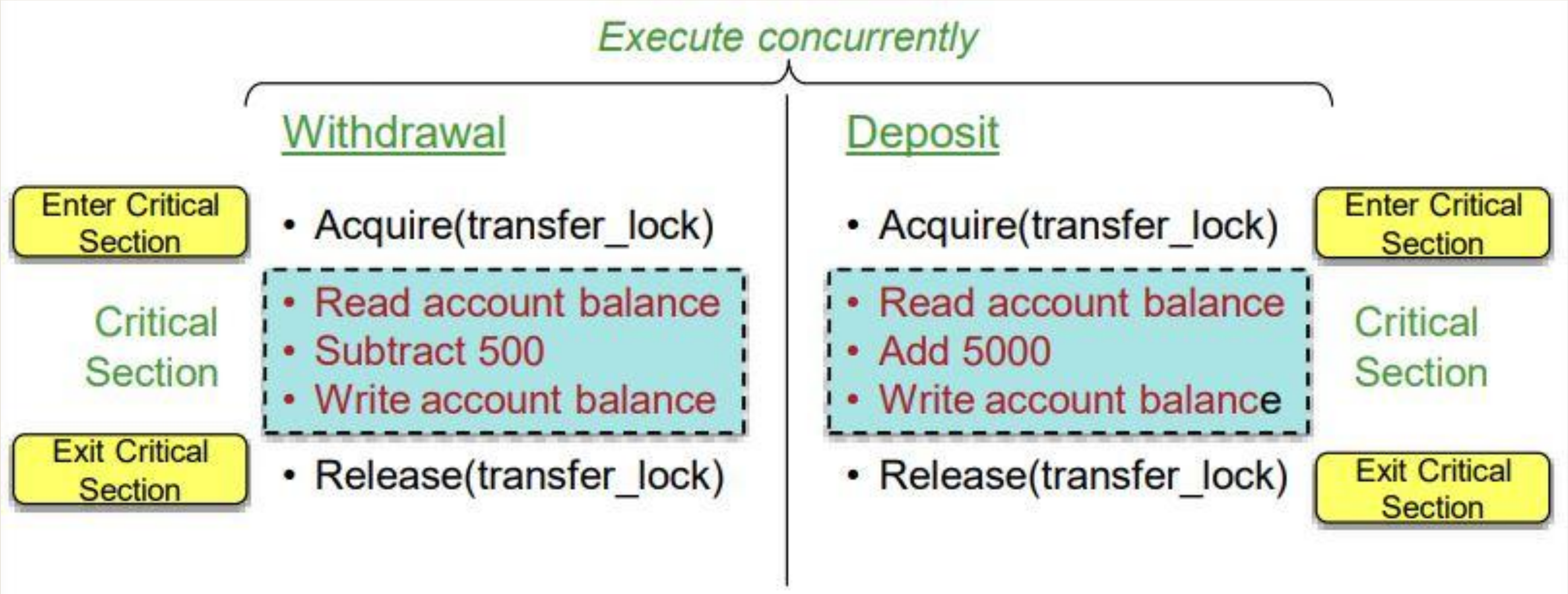


  - The possible new balances are $5500 $500 $6000

Critical Sections

- A race condition typically occurs when two or more cooperating threads try to read and/or write the shared data concurrently.
  - The regions of a program that try to access shared resources and cause race conditions are called critical sections (CS.)
- It is possible to avoid race conditions by allowing only one thread at a time to execute within a CS.
- Race conditions can be avoided by using locks with CSs.
  - A thread must acquire the associated lock before entering a CS.
    - On exit a CS, a thread must release the associated lock.
  - If the associated lock is not ready, the acquiring thread must be blocked.

Critical Section with Lock



The consistent new balances are $5500

Features of CS Problem Solutions

- Mutual exclusion:
  - No more than one thread may be inside the same CSs simultaneously.
- Progress:
  - No thread running outside its critical section may block others.
  - If no thread in CS, then the next thread that will enter the CS must be decided in finite time.
- Bounded waiting:
  - No thread should wait forever to enter a critical section.
- A good solution will make no assumptions on:
  - the number of processors
  - the number of threads/processes
  - the relative speed of threads/processes

Peterson's Algorithm

```
Shared Data {
        int turn;
        Boolean flag[2] ;
}

Thread(i), where (i=0 and j=1) or (i=1 and j=0)
do {
        flag[i] = TRUE;
        turn = j;
        while (flag[j] && turn == j) { };
        // empty while for busy waiting
        critical section
        flag[i] = FALSE;
        remainder section
} while (TRUE);
```

Hardware Solutions

- Many systems provide hardware support for critical section code
- Uniprocessor system – disable interrupts
  - Currently running code would execute without preemption
  - In general, it is too inefficient on multiprocessor systems
    - Operating systems using this are not broadly scalable
- Modern machines provide special atomic hardware instructions
  - Atomic means non-interruptable
    - test-and-set
    - swap

Solution to CS Problem Using Locks

```
do {
        acquire lock
        critical section
        release lock
        remainder section
} while (TRUE);
```

```
boolean TestAndSet (boolean *target)
{
   boolean rv = *target; // get old value
   *target = TRUE; // set to TRUE
   return rv; // return old value
}
```

```
void Swap (boolean *a, boolean *b)
{
   boolean temp = *a;
   *a = *b;
   *b = temp:
}
```

Solution using TestAndSet

```
shared boolean lock = FALSE; // shared by all threads

do {
        while (TestAndSet (&lock)) { };   // do nothing

        //    critical section

        lock = FALSE;

        //      remainder section

} while (TRUE);
```

Solution using Swap

```
shared boolean lock = FALSE; // shared by all threads

do {    key = TRUE; // local to each thread

        while ( key == TRUE)
                Swap (&lock, &key );

         //    critical section

        lock = FALSE;

         //      remainder section

} while (TRUE);
```

Spin Lock & Priority Inversion

- The atomic-instruction-based locks require looping in software to wait until the lock is released.
  - This is called busy waiting or a spinlock.
- If a high priority thread is spinlocking, it may prevent a low priority thread to release the waiting lock.
  - This situation is known as priority inversion.
- Priority Inheritance may be used to avoid priority inversion.
  - Increase the priority of a thread that is in a CS to the maximum priority of all waiting threads for the associated lock.
  - When the lock is released, the priority goes back to its normal level.

Semaphores

- Semaphore is a synchronization tool provided by OS so that threads can use for mutual exclusion and wait for a critical section without application level busy waiting.
- Semaphore S is a special integer variable.
  - Semaphores can only be accessed via two atomic operations.
  - wait() and signal() (called P() and V() in the past.)

```
wait (S) {
        while (S <= 0){ };    // busy waiting inside wait()
        // no-op
        S--;
}

signal (S) {
        S++;
}
```

Semaphore as Synchronization Tool

- Binary semaphore (Also known as mutex lock)
  - integer value can range only between 0 and 1
- Counting semaphore
  - integer value can range over an unrestricted domain
  - can be implemented by a binary semaphore
- CS problem can be solved by using semaphore

```
Semaphore mutex = 1;    //  initialized to 1

do {
        wait (mutex);    // acquire lock
        // Critical Section
        signal (mutex);  // release lock
        // remainder section
} while (TRUE);
```

No Busy Waiting Semaphore Implementation

- Associate each semaphore with a waiting queue.
  - Each entry in the waiting queue has two data items:
    - value (of type integer) and pointer to next record in the list
  - Two special operations for the waiting queue:
    - block() places the invoking thread on the corresponding waiting queue.
    - wakeup() removes one of threads from the waiting queue and places it in the ready queue.

```
wait(semaphore *S) {
    S->value --;
    if (S->value < 0) {
        add the calling thread T to S->list;
        block(T);
    }
}
```

```
signal(semaphore *S) {
    S->value++;
    if (S->value <= 0) {
        remove a thread T from S->list;
        wakeup(T);
    }
}
```