Distributed Systems

CSCI-B 534/ENGR E-510

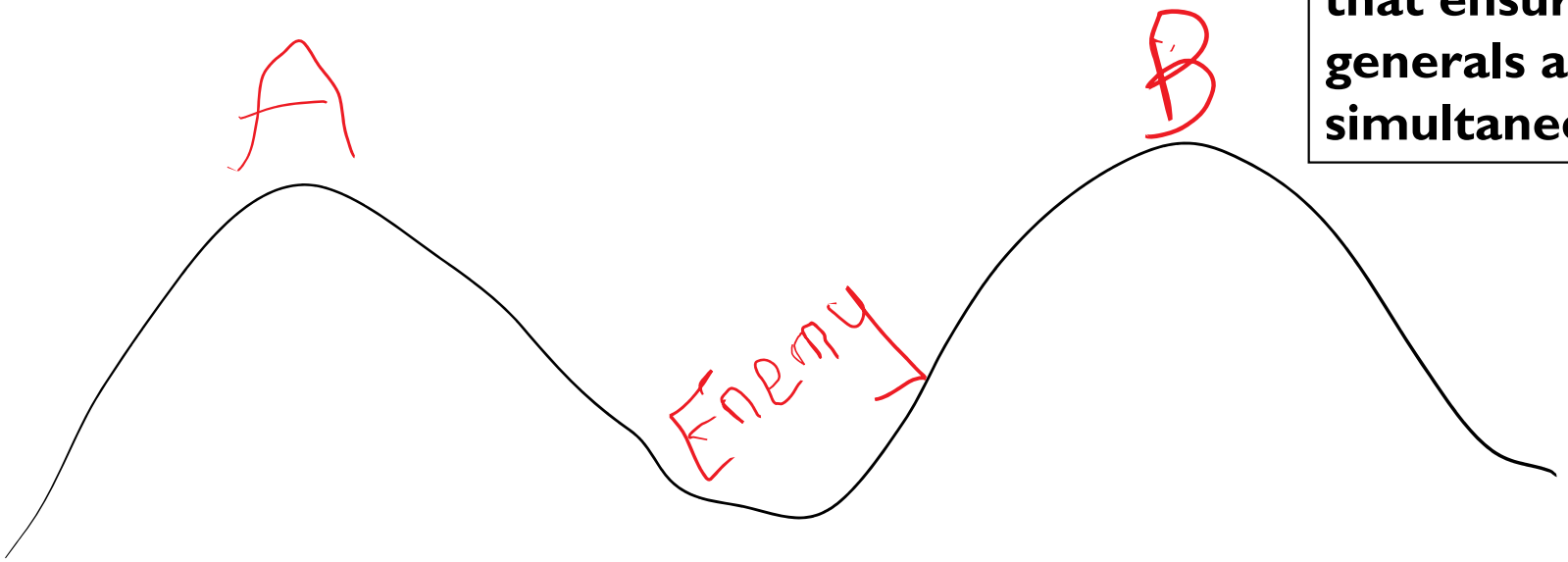
Spring 2019

Instructor: Prateek Sharma

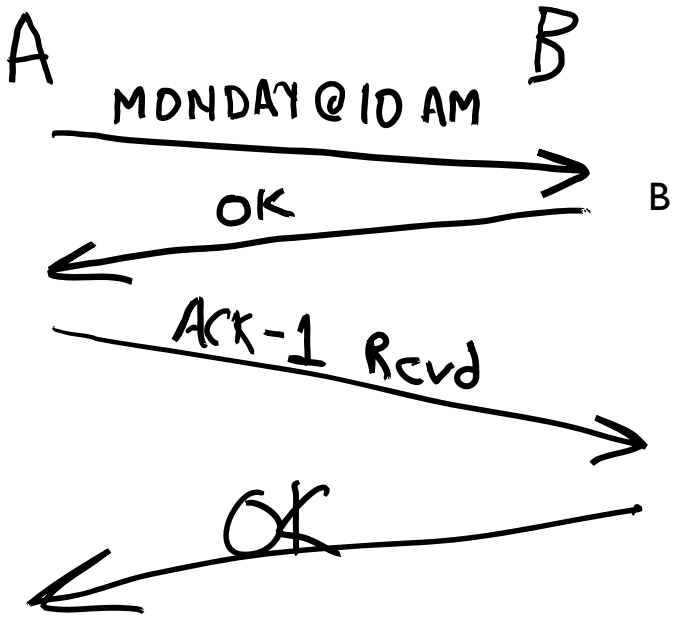
Two Generals Problem

- Two Roman Generals want to co-ordinate an attack on the enemy
  - Both must attack simultaneously. Otherwise, both will lose
- Only way to communicate is via a messenger
  - But messengers can get captured/lost.
  - Perfectly-reliable communication system not available

Task: Design a protocol that ensures the two generals always attack simultaneously



Two generals problem, continued



A does not know if B knows that A knows

B does not know if A knows about the agreement

Impossibility Proof of Two Generals Problem

- Claim: There is no non-trivial protocol that guarantees that the two generals will always attack simultaneously
- Proof by induction on the number of messages
- Let d messages be delivered at the time of attack
- Base case: d=0. Claim holds (Impossible without any delivered messages)
- Suppose impossibility claim holds for d=n. Then, we'll show for d=n+1
- Consider message n+1
  - Sender attacks without knowing if message is delivered or not
  - Receiver must then attack too, even if msg not received
  - So the last message (n+1) was irrelevant, and n messages suffice
  - But that's a contradiction: since n+1 was supposed to be the smallest number of messages

Common Knowledge

- Solving the Two Generals Problem requires common knowledge
- Common knowledge cannot be achieved with unreliable communication channels

Building Blocks Of Distributed Systems

What Are Distributed Systems Composed Of?

- Collection of nodes
- What are nodes?
  - Servers of different sizes (Raspberry Pis, 128 core large servers, etc.)
  - Conventional OS processes

OS Recap

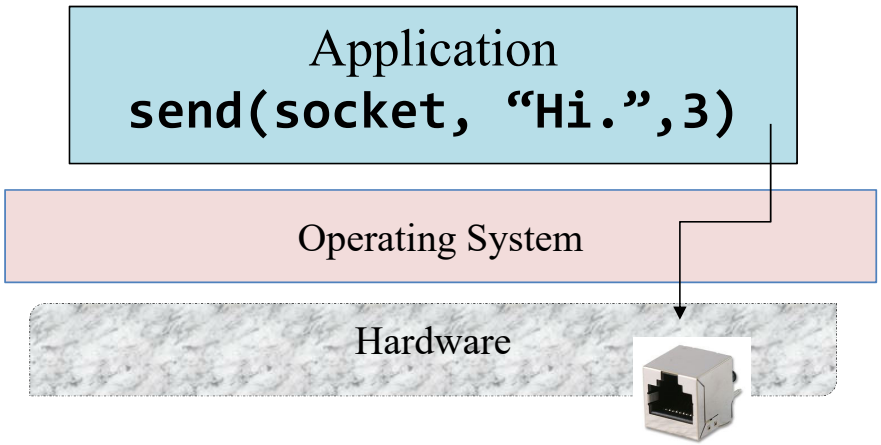
- Operating Systems: Easier to run applications
- OS provides a convenient interface to run multiple programs in a secure manner
- Portability: Decouple applications from hardware
  - Changing your USB keyboard => No need to rewrite and recompile programs
- Resource allocation and multiplexing
- OS provides all these features by:
  - Different abstractions & services
  - Interfacing with hardware features designed to help OS

OS Services

What services and features do Operating Systems provide?
Why do we use Operating Systems?

Applications, OS, and Hardware

- Programs: Sequence of CPU instructions
  - Mov, add, jmp,…
- Programs often build on top of and make use of other programs (“libraries”)
- OS provides a wide range of services to applications



Operating System Services

- One set of functions of the OS provides services to the user:
  - User interface - Almost all operating systems have a user interface (UI)
    - Command-Line (CLI), Graphical User Interface (GUI), Batch
  - Program execution - The system must be able to load a program into memory and to run that program, end execution, either normally or abnormally (indicating error)
  - I/O operations - A running program may require I/O, which may involve a file or an I/O device.
  - File-system manipulation - The file system is of particular interest. Obviously, programs need to read and write files and directories, create and delete them, search them, list file Information, permission management.

Operating System Services (Cont.)

- OS services for the user (con't):
  - Communications – Processes may exchange information, on the same computer or between computers over a network
    - Communications may be via shared memory or through message passing (packets moved by the OS)
  - Error detection – OS needs to be constantly aware of possible errors
    - May occur in the CPU and memory hardware, in I/O devices, in user program
    - For each type of error, OS should take the appropriate action to ensure correct and consistent computing
    - Debugging facilities can greatly enhance the user's and programmer's abilities to efficiently use the system

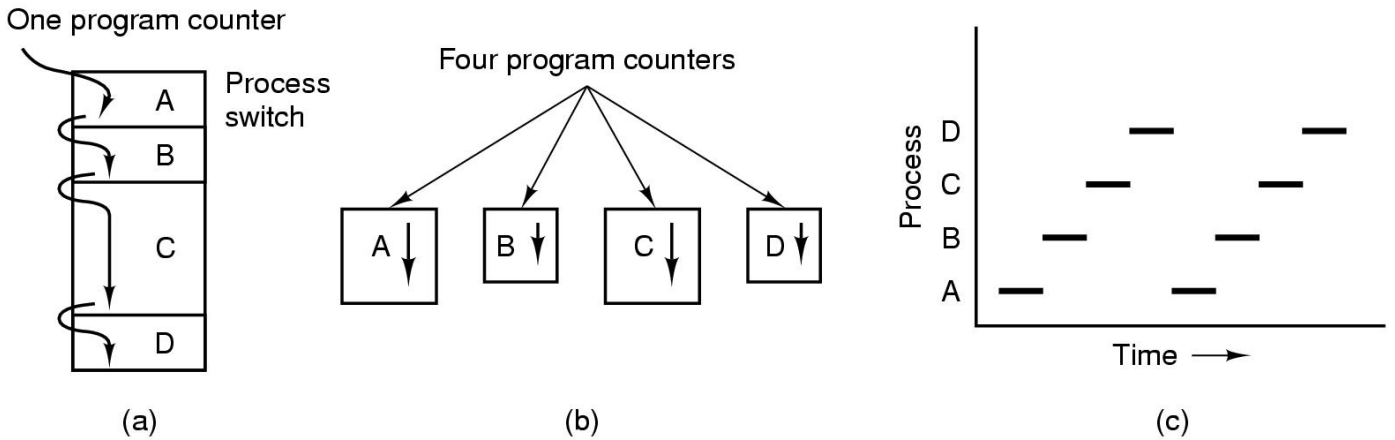
Operating System Services (Cont.)

- Another set of OS functions exists for ensuring the efficient operation of the system itself via resource sharing
  - **Resource allocation -** When multiple users or multiple jobs running concurrently, resources must be allocated to each of them
    - Many types of resources - Some (such as CPU cycles, main memory, and file storage) may have special allocation code, others (such as I/O devices) may have general request and release code.
  - **Accounting -** To keep track of which users use how much and what kinds of computer resources
  - **Protection and security -** The owners of information stored in a multiuser or networked computer system may want to control use of that information, concurrent processes should not interfere with each other
    - **Protection** involves ensuring that all access to system resources is controlled
    - **Security** of the system from outsiders requires user authentication, extends to defending external I/O devices from invalid access attempts
    - If a system is to be protected and secure, precautions must be instituted throughout it. A chain is only as strong as its weakest link.

Programs and processes

- A program is a series of instructions
  - code for a single "process" of control
- Process: running program + state
- State: Input, output, memory, code, file, etc.
- A Thread is an execution context with register state, a program counter (PC) and a stack
  - "Thread of execution"
- Multiple processes can be running the same program, even sharing the code in the same memory space
  - reduces memory overhead, which is important in limited memory environments like embedded OSes

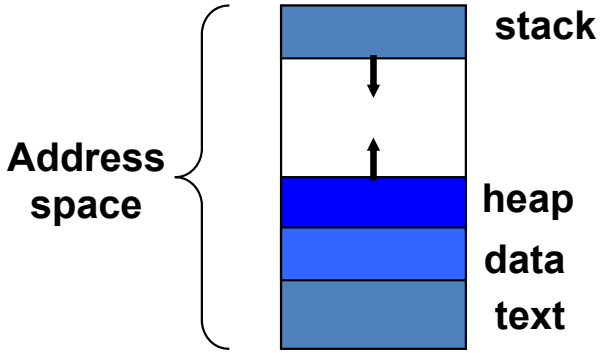
The process abstraction



- Multiprogramming of four programs in the same address space
- Conceptual model of 4 independent, sequential processes
- Only one program active at any instant

UNIX Process Address Space

- Memory locations process is allowed to address
- Each process runs in its own virtual memory *address space* that consists of:
  - *Stack space* – used for function and system calls
  - *Data space* – static variables, initialized globals
  - *Heap space* – dynamically allocated variables
  - *Text* – the program code (usually read only)



- Invoking the same program multiple times results in the creation of multiple distinct address spaces

UNIX Process Creation

- Parent processes create child processes, which, in turn create other processes, forming a tree of processes
- Resource sharing options
  - Parent and children share all resources
  - Children share subset of parent's resources
  - Parent and child share no resources
- Execution options
  - Parent and children execute concurrently
  - Parent waits until children terminate

UNIX Process Creation (Cont.)

- Address space
  - Child duplicate of parent
  - Child has a program loaded into it
- UNIX examples
  - **fork** system call creates new process
  - **exec** system call used after a **fork** to replace the process' memory space with a new program

CPU Virtualization

- Processes create the illusion of multiple "virtual" CPUs that programs fully control
- Process PCB contains program counter and other register state, allowing it to be "resumed"
- Timesharing: OS switches process running on physical CPU at high frequency (context switch)
- Virtualization is a key OS principle
  - Applies to CPU, memory, I/O, …

Example: process creation in UNIX

sh (pid = 22)

```
…

pid = fork()
if (pid == 0) {
   // child…
   …
   exec();
   }
else {
   // parent
   wait();
   }
…
```

Process creation in UNIX example

sh (pid = 22)

```
…
pid = fork()
if (pid == 0) {
   // child…
   …
   exec();
   }
else {
   // parent
   wait();
   }
…
```

sh (pid = 24)

```
…
pid = fork()
if (pid == 0) {
   // child…
   …
   exec();
   }
else {
   // parent
   wait();
   }
…
```

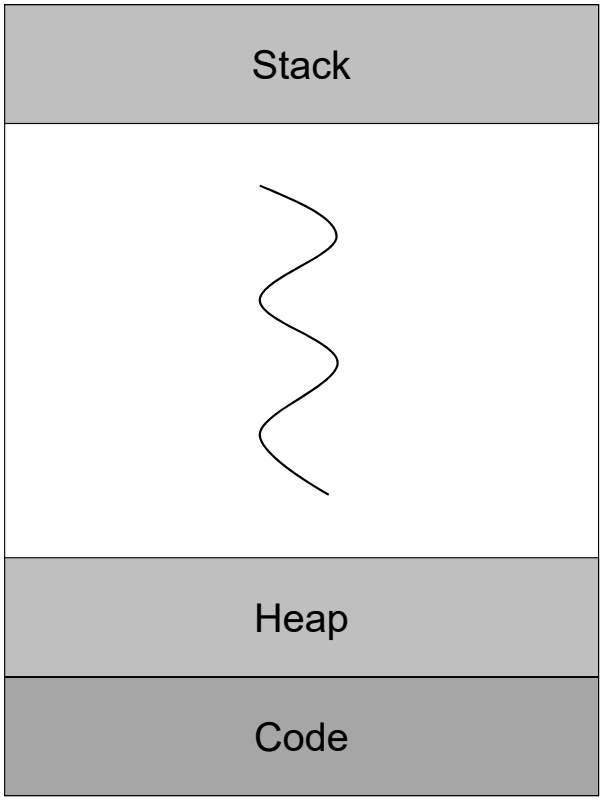
Process Creation in UNIX/Bash

- `>./my-program.o &`
- #This creates a process that runs my-program.o, and runs it in the background
- Typical setup: spawn multiple processes :
- `>./dist-program node-id=1 type=primary-node &`
- `>./dist-program node-id=2 type=primary-node &`
- `>./dist-program node-id=3 type=secondary-node &`
- Exercise: Get comfortable with process creation and termination in your language/environment
  - Python subprocess

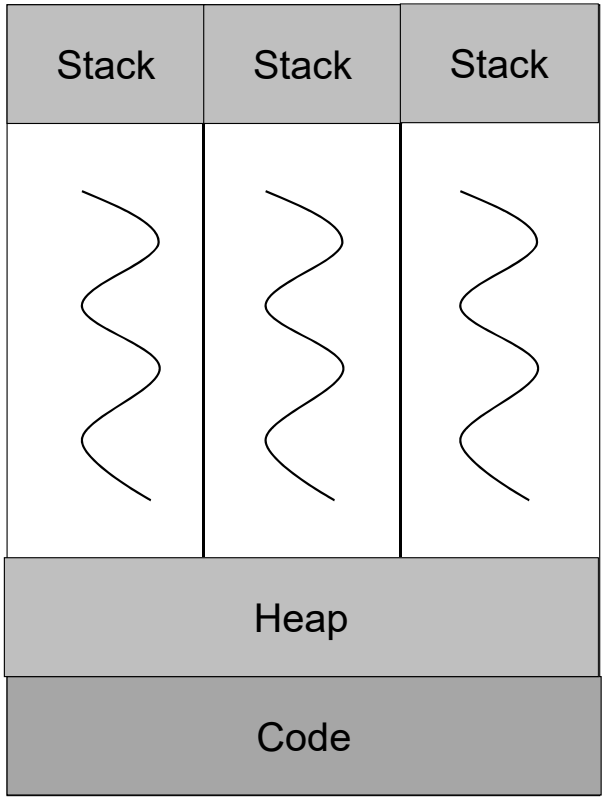
UNIX Threads

- Creation of a process using `fork()` is expensive (time and machine effort)
  - Memory copying to create a copy of the process
    - In many cases just to call `exec()` and replace it
    - There are ways to mitigate creating a complete copy
  - Coordinating activities across process boundaries requires effort
- Threads are sometimes called *lightweight processes*
  - What we have called a process is sometimes considered a *heavyweight* process
  - A thread contains the necessary state for a distinct activity (process in the most general sense)

Single and Multithreaded Processes



One Thread



Multiple Threads

Benefits of Threads

- Efficiency / economy
  - Less memory, fewer system resources
- Responsiveness
  - Lower startup time
- Easier resource sharing
  - Natural sharing of memory, open files, etc.
  - With caveats that we will discuss
- Concurrency
  - Utilization of multiple processors or cores
- You can use threads as distributed system nodes, as long as you don’t use shared memory

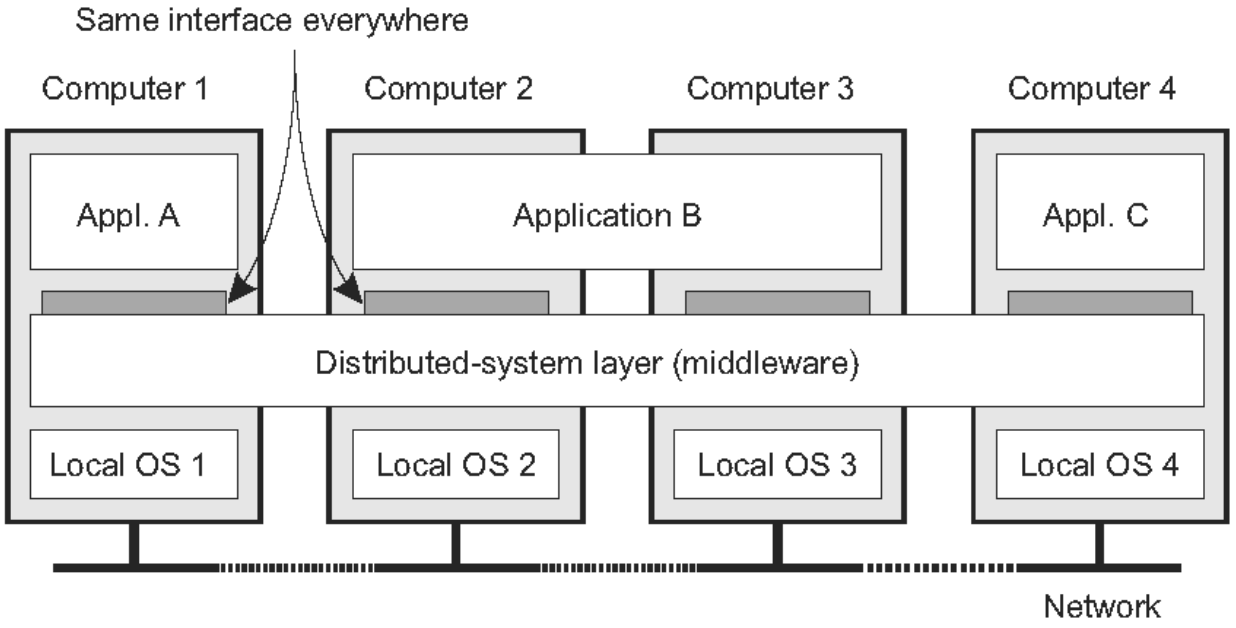
Building Distributed Programs With Processes

- Remember that process === node
- Each process must have some "global" id === (machine-id, process-id)
  - Machine-id === (ip-address, [port])
- Processes communicate through well-defined communication channels
  - Network sockets (covered in next class)
- Be careful with process management
  - When to start/stop processes
  - Clean-up state on termination/failure : Temporary files, open sockets, etc.

Distributed Operating Systems

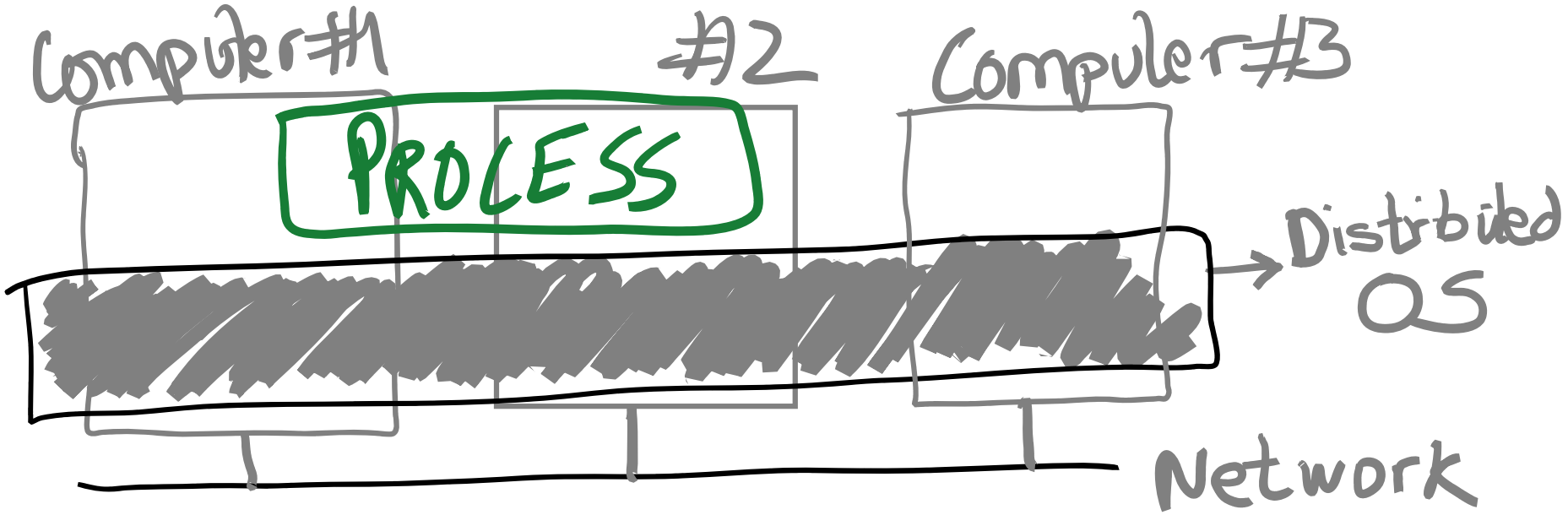
Middleware: The OS of Distributed Systems

- Commonly used components and functions for distributed applications



Distributed Operating System

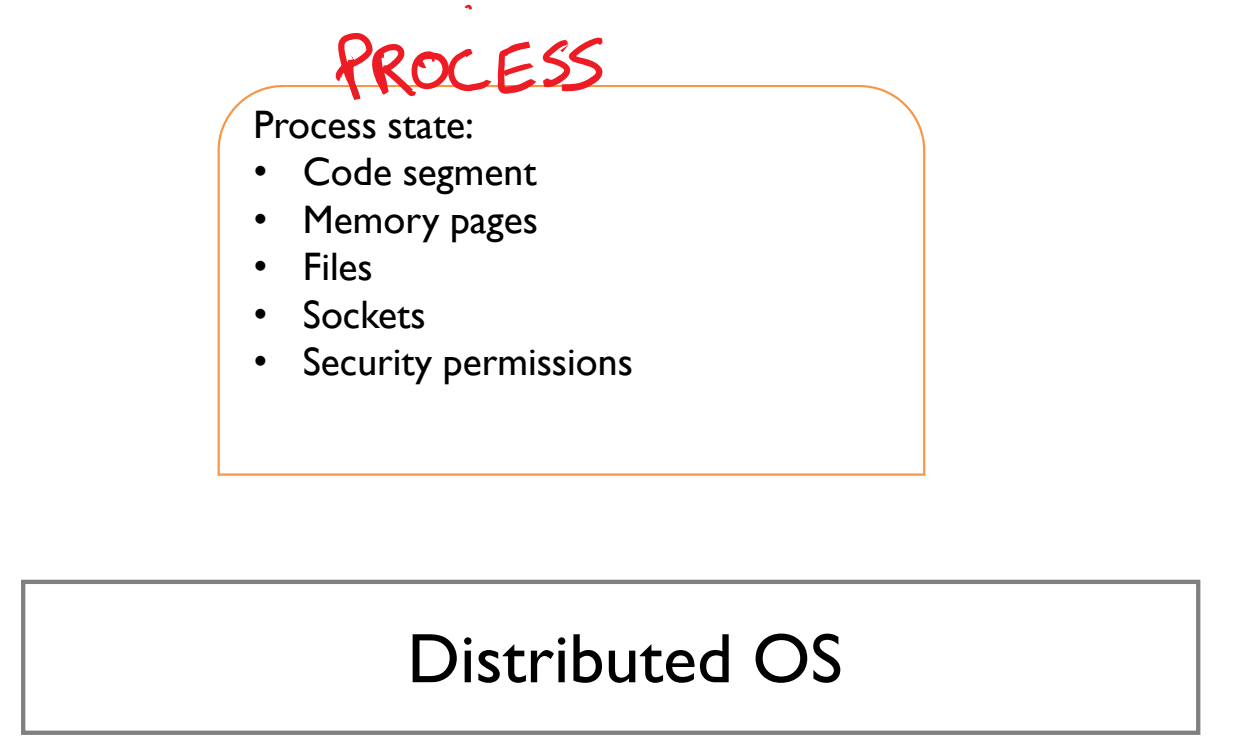
- An OS that spans multiple computers
- Same OS services, functionality, and abstractions as single-machine OS



Distributed OS Challenges

- Providing the process abstraction and resource virtualization is hard
- Resource virtualization must be transparent
  - But in distributed settings, there's always a distinction between local and remote resources
- In a single-machine OS, processes don't care where their resources are coming from:
  - Which CPU cores, when they are scheduled, which physical memory pages they use, etc.
- In fact, providing abstract, virtual resources is one of the main OS services

Processes In Distributed OS



Transparency Issues In Distributed OS

PROCESS

Process state:
- Code segment
- Memory pages
- Files
- Sockets
- Security permissions

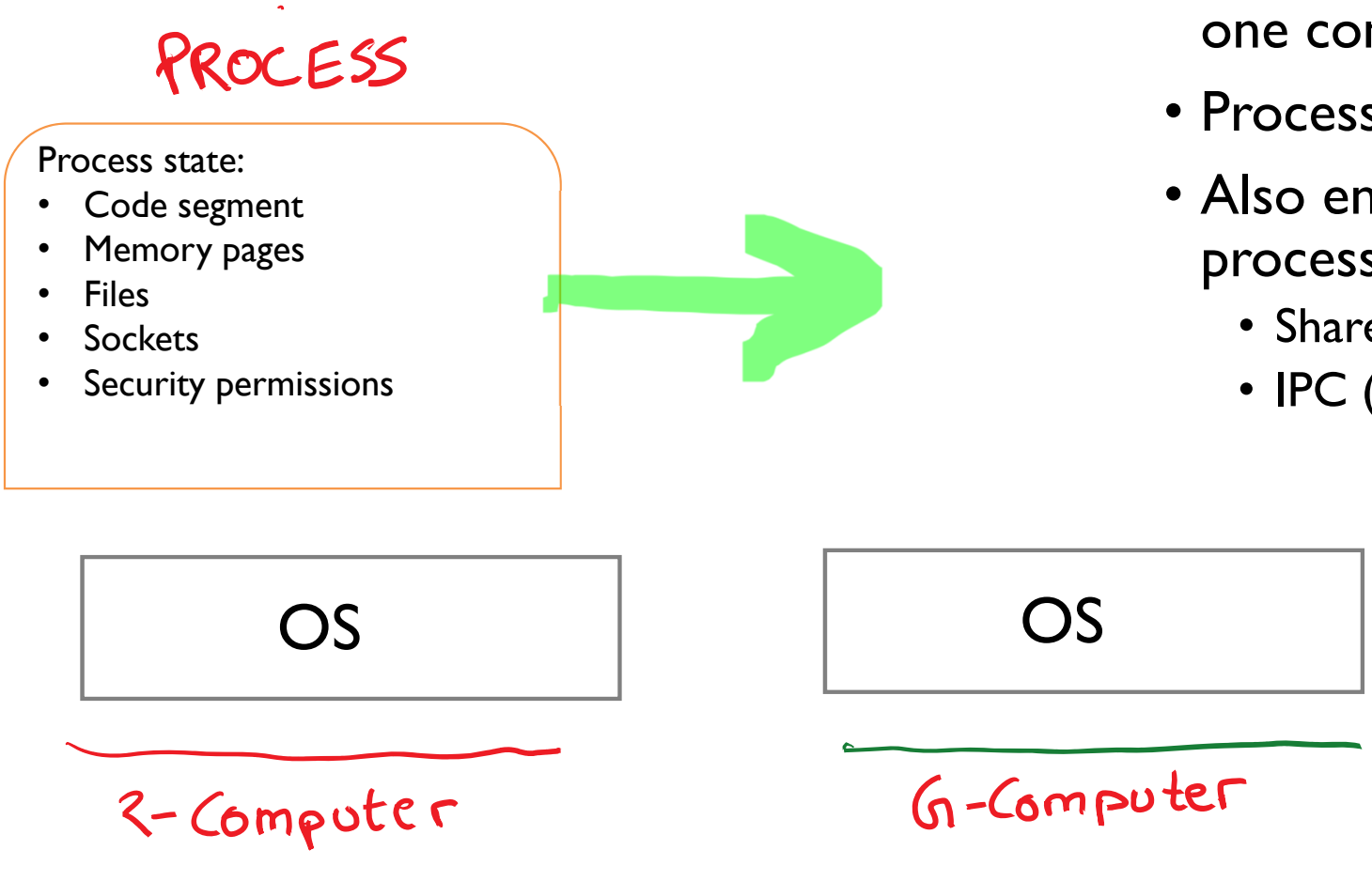
- Where does code run?
- Which memory is used?
  - Local vs. remote
- How are files accessed?

Distributed OS

R-Computer

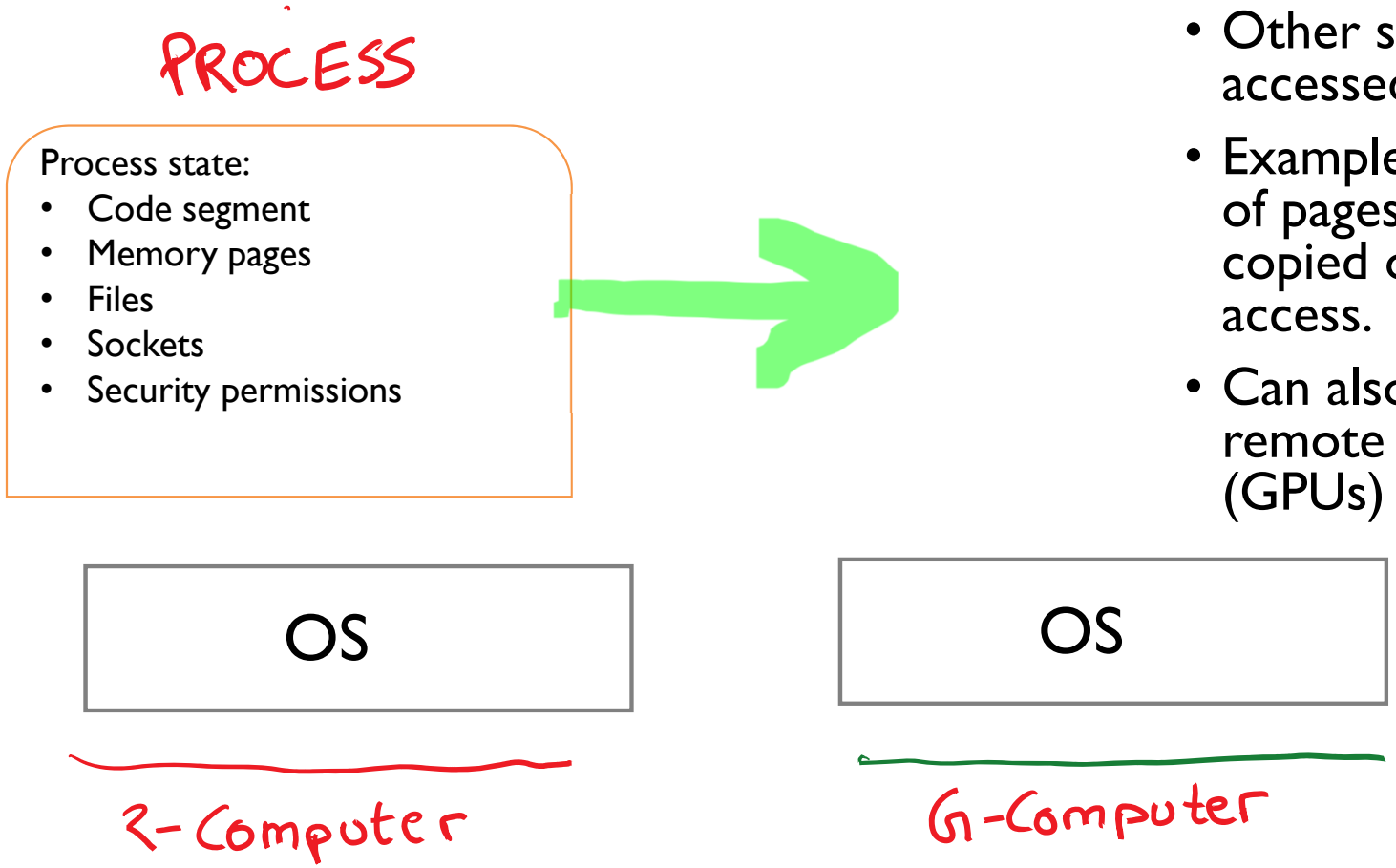
G-Computer

Process Migration



- Move all process state from one computer to another
- Process state can be vast
- Also entangled with other process states
  - Shared files?
  - IPC (pipes etc)

Partial Process Migration



- Migrate some state
- Other state, if required, is accessed over the network
- Example: migrate only fraction of pages. Other pages are copied over the network on access.
- Can also be used to access remote hardware devices (GPUs)