Operating Systems: The Basics

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

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, files, etc.
- Processes are one of the main abstractions provided by the operating system
- 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

Processes as Distributed System Components

- Processes are isolated from each other, and thus "independent and autonomous"
- Each process is running its own code, with its own memory address space (local variables etc)
  - We will assume that the only way to communicate is explicit messages
    - Using networking protocol
  - Reading/writing to any shared object is communication!
    - Any variables/data structures in memory
    - Or files on disk
- If you don't share (too much) state, then it doesn't matter where they run
- For most assignments, all processes will be running on the same machine (for convenience)
  - But, your design should work even if the processes run on different machines!

Concurrent Execution

```
# main.py . Driver program
import os, subprocess

p1 = subprocess.Popen('python3 alice.py', shell=True)
p2 = subprocess.Popen('python3 bob.py', shell=True)
```

```
# Alice.py
import os,sys,time

while True:
    time.sleep(1)
    print("Alice here!")
```

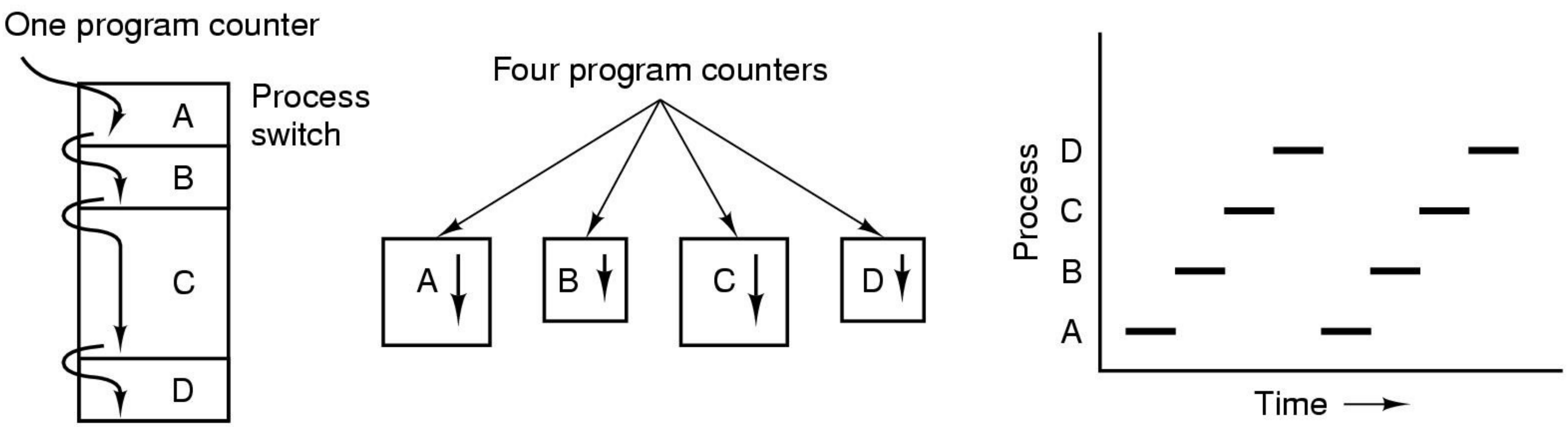
```
# Bob.py
import os,sys,time

while True:
    time.sleep(1)
    print(”Bob here!")
```

- Popen will launch in background and will not block
  - Wait for p1 to finish using p1.wait()
  - Can also grab output of p1 using capture_output
  - See subprocess documentation!!
- Careful around full pathnames
  - Best practice: os.getcwd()+’alice.py’
  - Shell=True passes envmt variables

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

The process abstraction

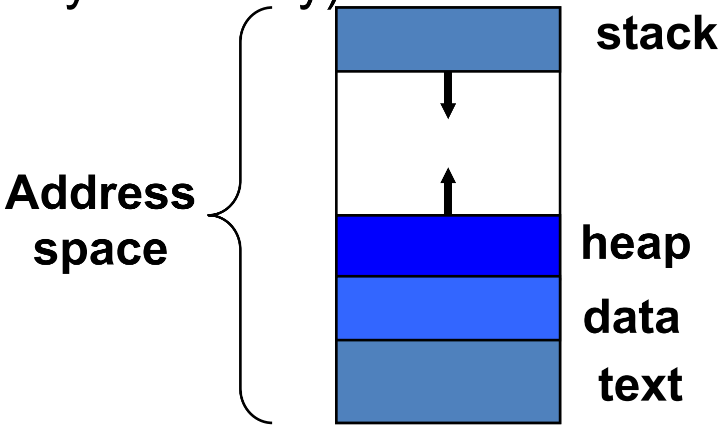


(a) (b) (c)

- 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 a 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();
   }
…
```

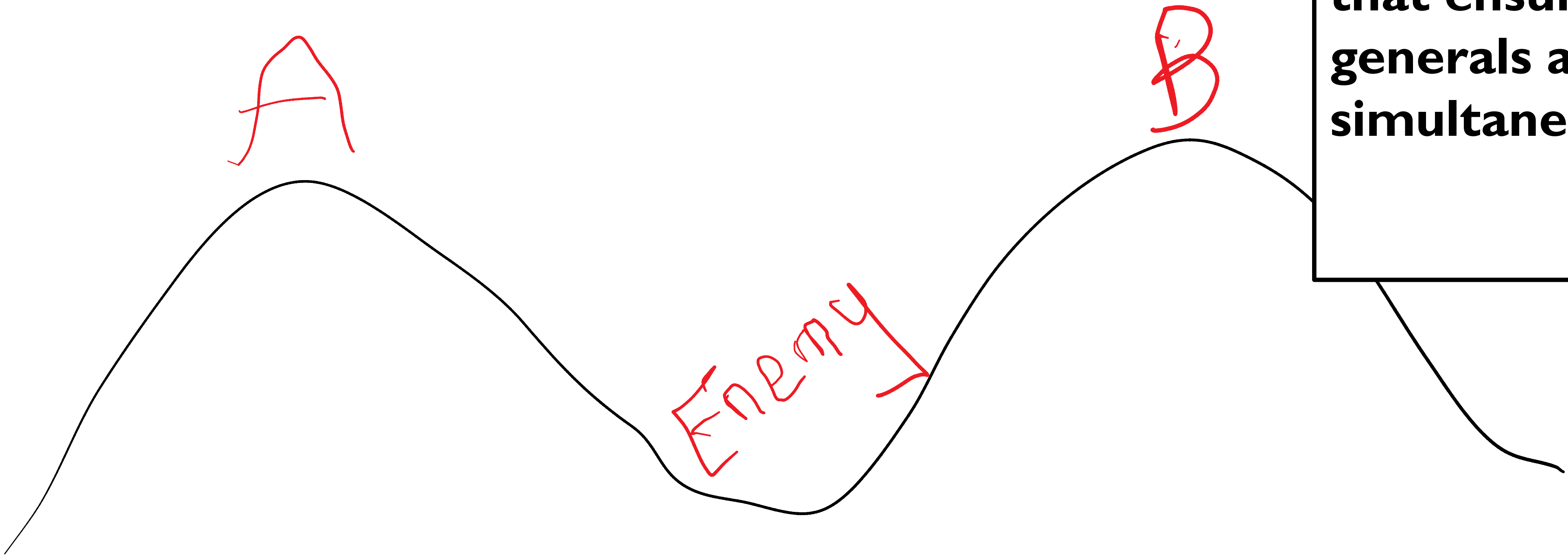
Building Distributed Programs With Processes

- 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.

Common Knowledge

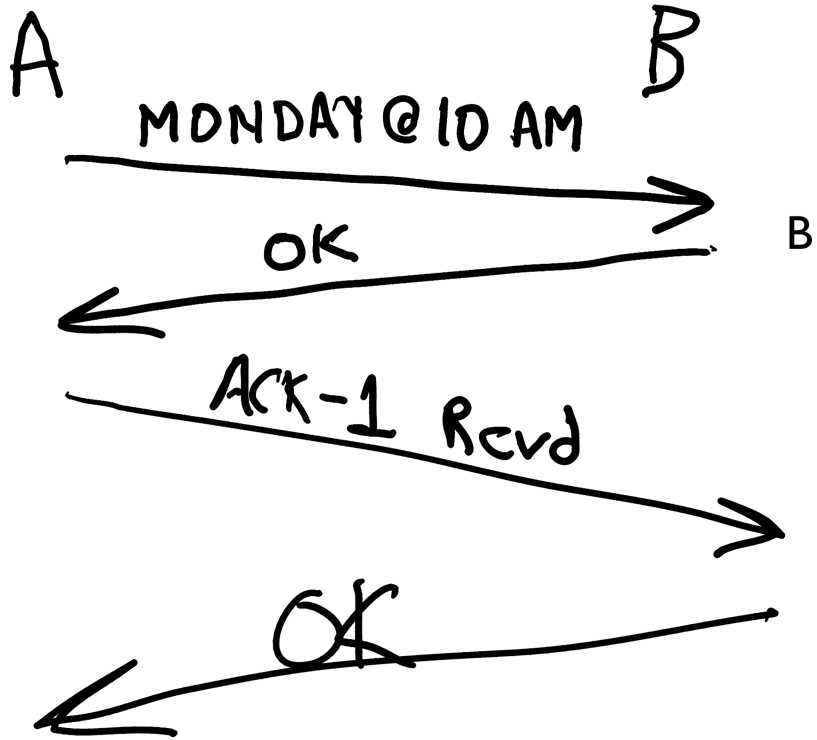
Two Generals Problem

- 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



B does not·know if A knows about the agreement

A does not know if B knows that A knows

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

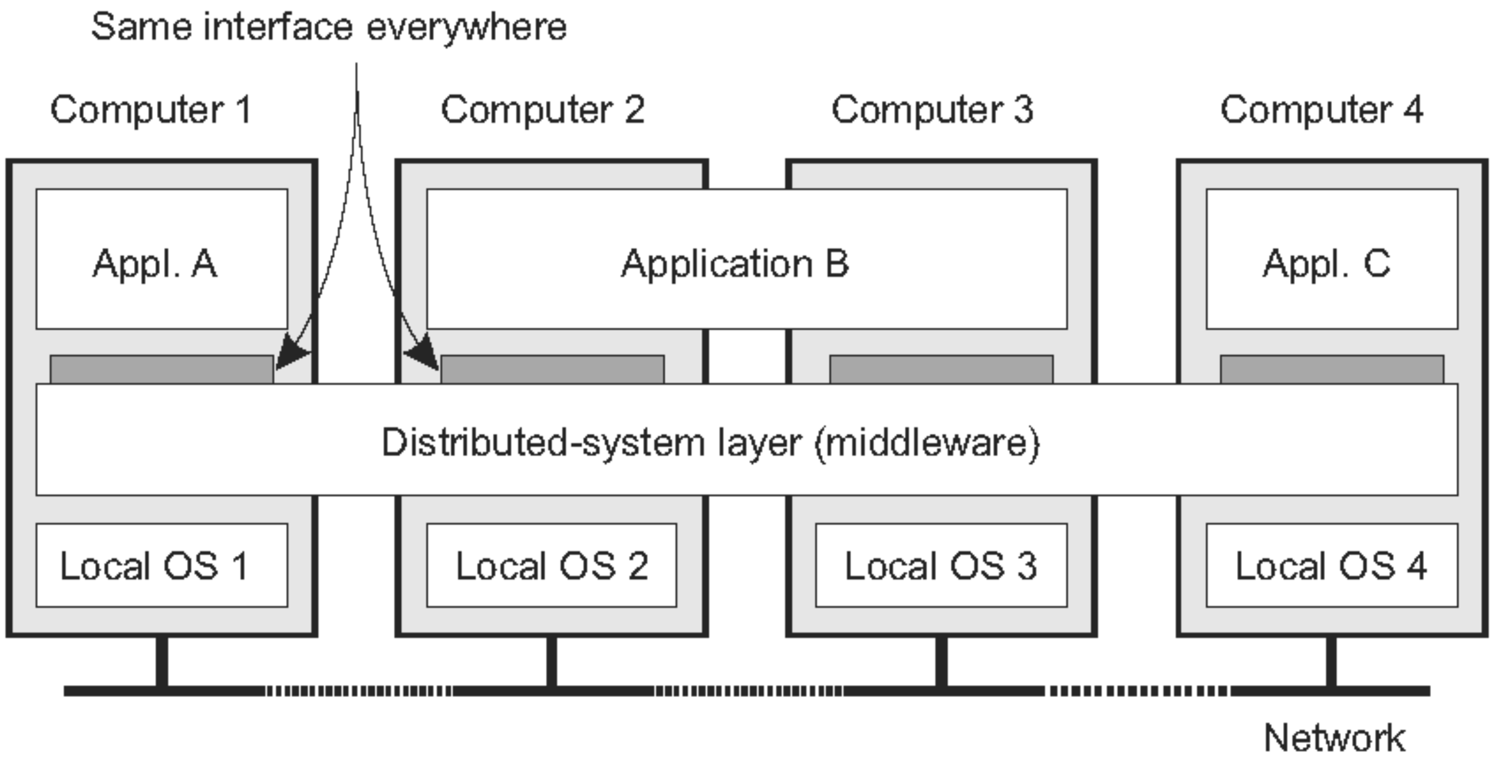
Common Knowledge

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

Distributed Operating Systems

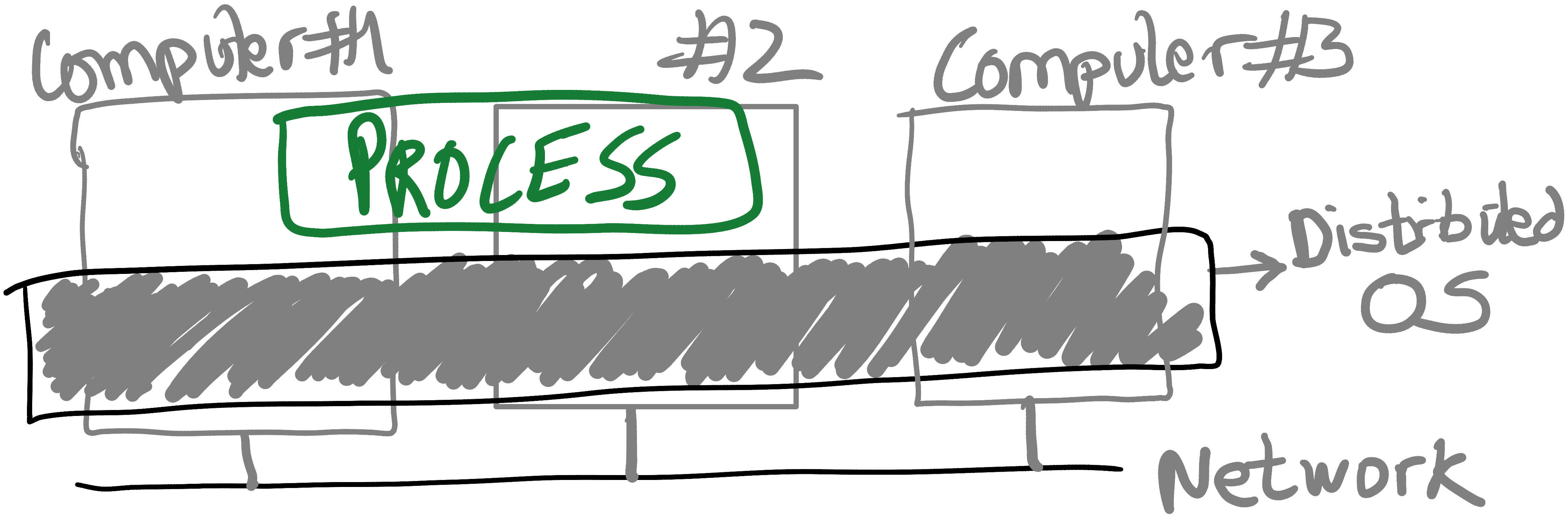
Middleware: The OS of Distributed Systems

- Commonly used components and functions for distributed applications



Distributed Operating System

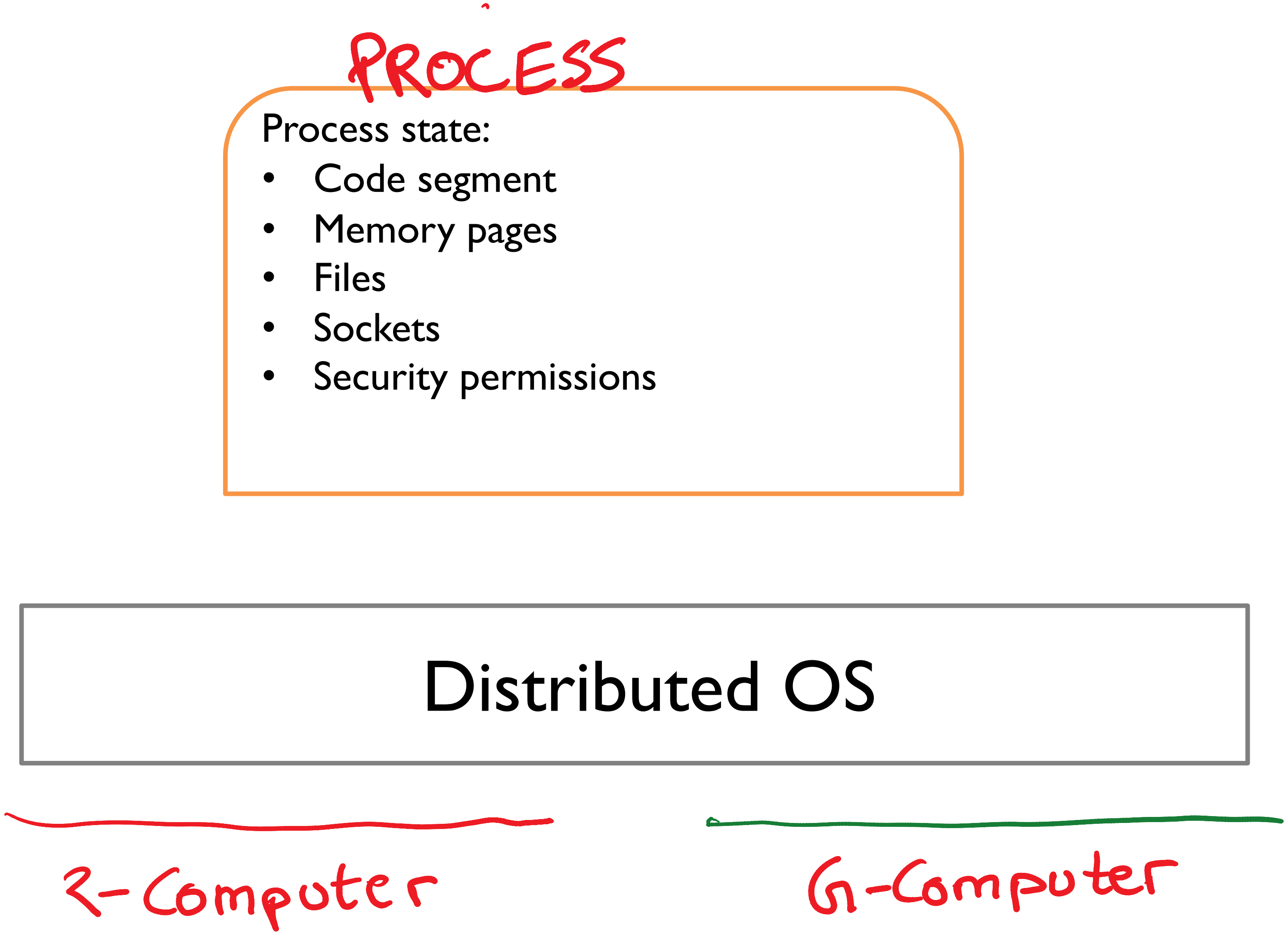
- An OS that spans multiple computers
- Same OS services, functionality, and abstractions as single-machine OS



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

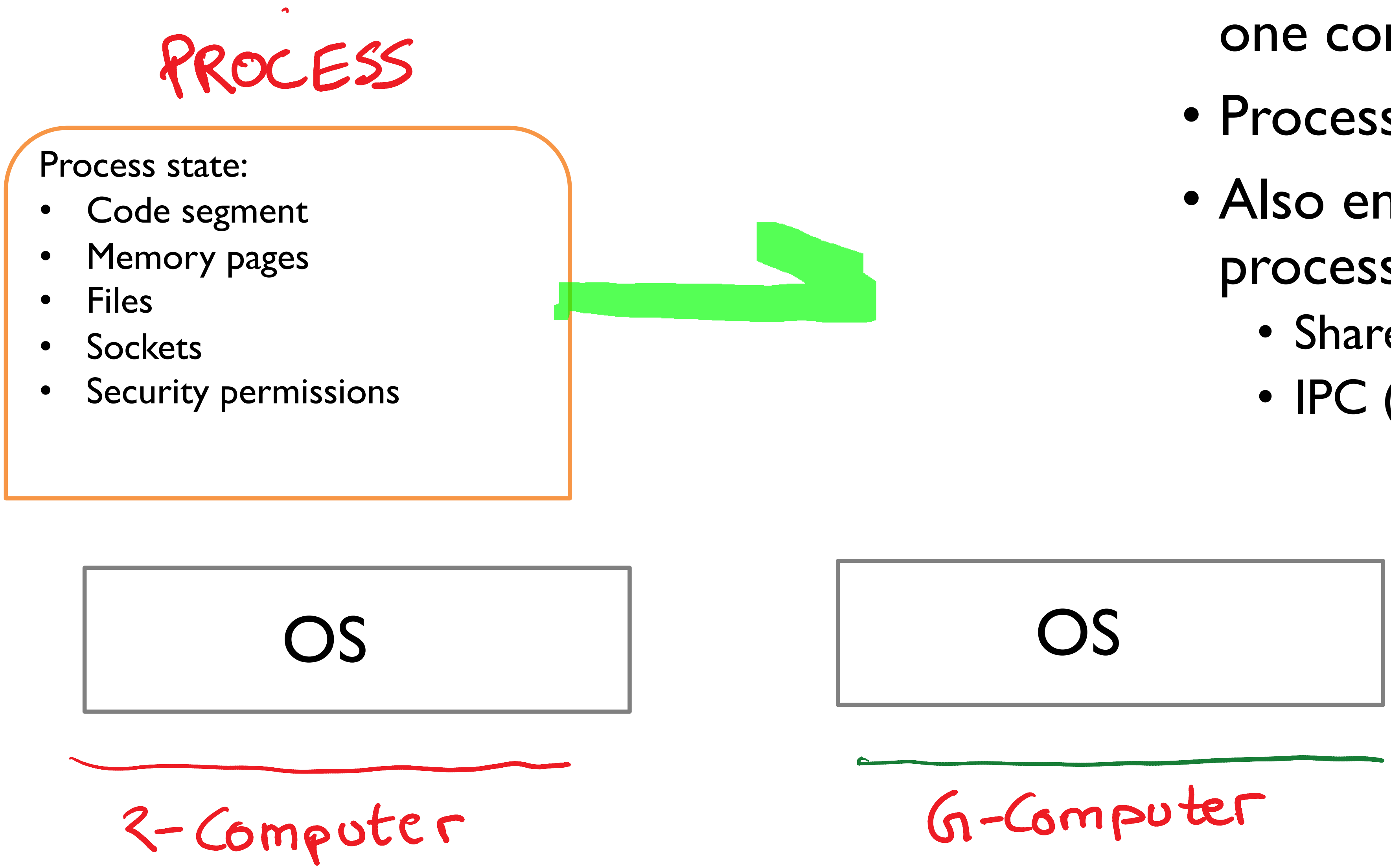
Distributed OS

R-Computer

G-Computer

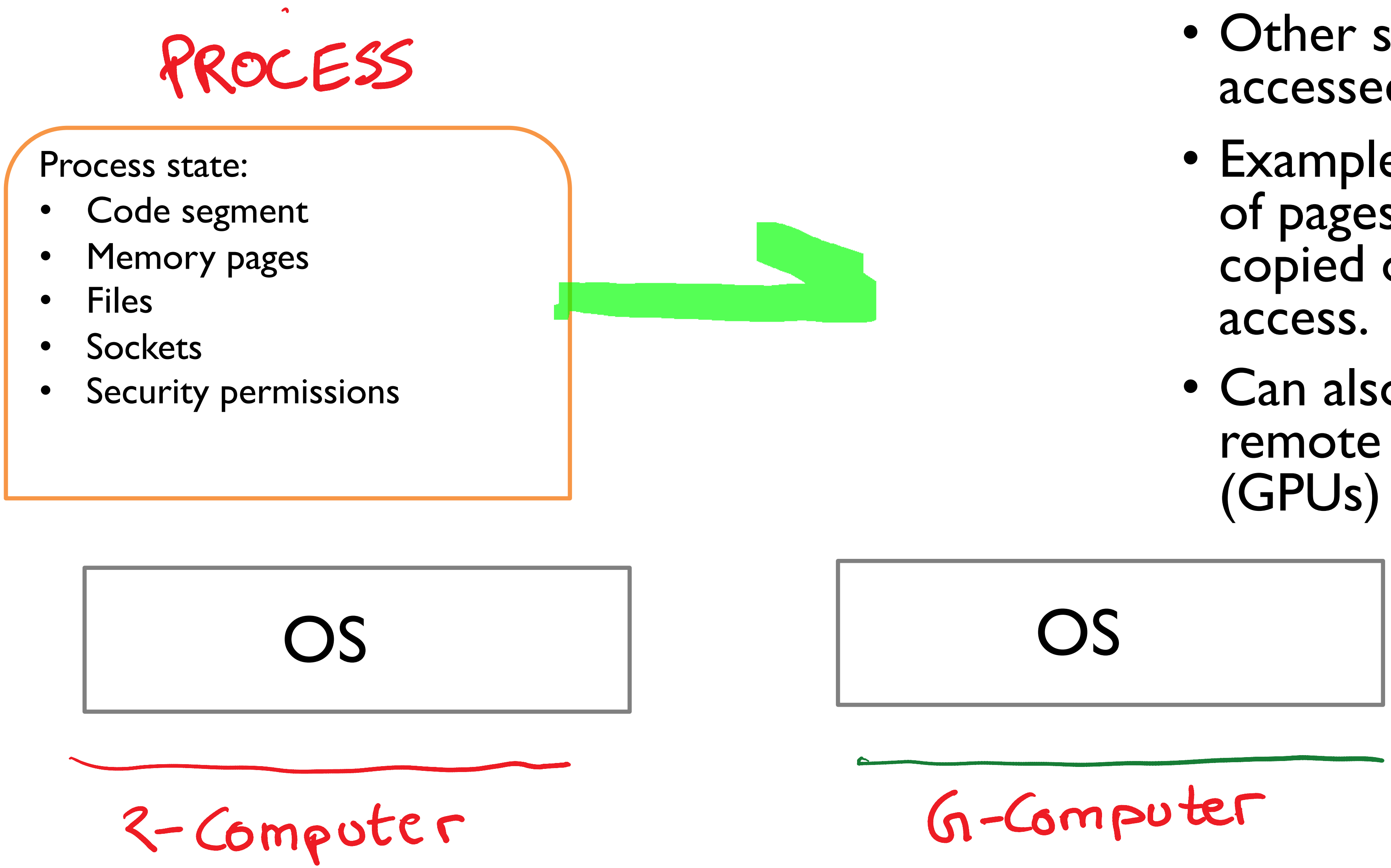
- Where does code run?
- Which memory is used?
  - Local vs. remote
- How are files accessed?

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)