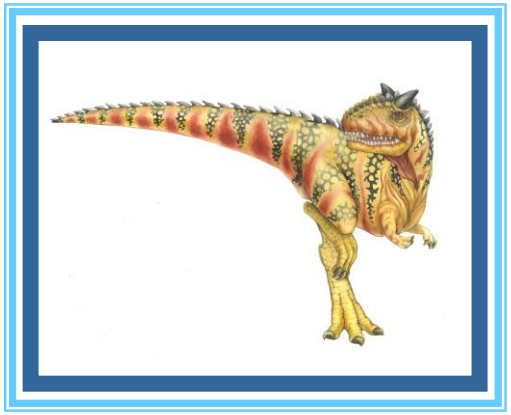
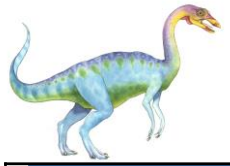
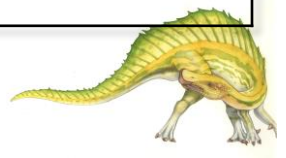
Chapter 3: Processes





Comparison: BIOS vs. OS

Aspect	BIOS	Operating System (OS)
Location	Firmware stored on the motherboard	Software stored on a storage device (e.g., HDD, SSD)
Primary Function	Hardware initialization and boot process	Manages hardware resources and provides a platform for applications
Execution Time	Runs when the computer is powered on	Runs after the BIOS has loaded the bootloader and continues until shutdown
User Interaction	Limited (BIOS setup utility)	Extensive (GUI, CLI, application interfaces)
Hardware Control	Basic, low-level control	Full control, with support for advanced features and multitasking
Security Features	Basic password protection	Comprehensive security management, including access control, firewalls, and encryption
Memory Management	None (minimal memory checks during POST)	Comprehensive memory management, including allocation, paging, and segmentation
File System Management	None	Provides and manages file systems for data storage
Networking	None (except for basic PXE boot capabilities)	Full networking stack for local and internet connectivity

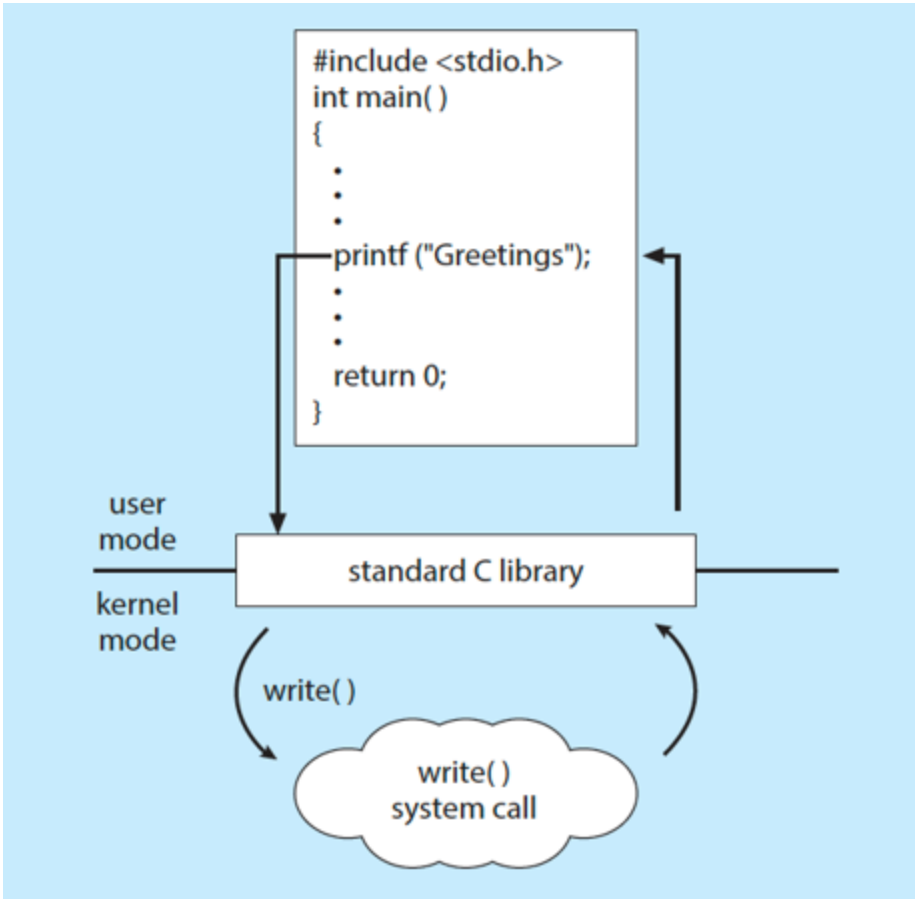


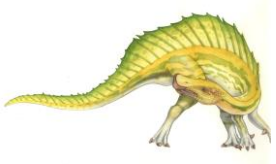
Summary

- **BIOS** is a low-level firmware that initializes hardware and boots the operating system. It provides basic input/output functions and system configuration options but lacks advanced features.

- **Operating System (OS)** is a more complex and comprehensive software layer that takes over from the BIOS, managing all aspects of the computer's operation, including hardware, memory, processes, files, security, and user interfaces. The OS provides the environment in which applications run and ensures the efficient and secure operation of the entire system.

Kernel vs User Mode





Kernel vs User Mode

```python
class OperatingSystem:
    def __init__(self):
        self.is_kernel_mode = False

    def system_call(self, operation, *args):
        """Simulate a system call which switches to kernel mode."""
        print("Switching to kernel mode...")
        self.is_kernel_mode = True

        if operation == "open_file":
            self._open_file(*args)
        elif operation == "allocate_memory":
            self._allocate_memory(*args)
        elif operation == "send_data":
            self._send_data(*args)
        else:
            print(f"Unknown operation: {operation}")

        print("Switching back to user mode...")
        self.is_kernel_mode = False

    def _open_file(self, filename):
        print(f"Kernel: Opening file '{filename}'")

    def _allocate_memory(self, size):
        print(f"Kernel: Allocating {size} bytes of memory")

    def _send_data(self, data):
        print(f"Kernel: Sending data '{data}'")
```

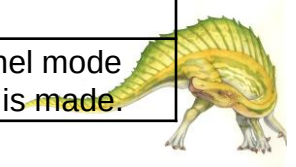
The kernel mode is where the operating system has unrestricted access to all hardware and can execute any instruction

The user mode is where application code runs with limited privileges, preventing direct access to hardware or critical system data.

```python
# Example user-mode functions
def user_mode_application(os):
    print("User Mode: Requesting to open a file")
    os.system_call("open_file", "example.txt")

    print("User Mode: Requesting memory allocation")
    os.system_call("allocate_memory", 1024)

    print("User Mode: Requesting to send data over the network")
    os.system_call("send_data", "Hello, World!")

# Example usage
os = OperatingSystem()
user_mode_application(os)
```

Kernel vs User Mode

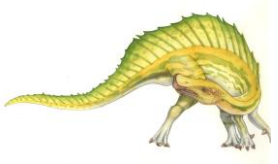
Aspect	Kernel Mode	User Mode
Definition	The mode in which the operating system executes with full access to all hardware resources.	The mode in which application software executes with restricted access to system resources.
Access Level	Full access to all hardware and system resources, including CPU, memory, and I/O devices.	Limited access; cannot directly interact with hardware or critical system areas.
Execution Context	Runs critical OS components such as device drivers, kernel, and core system processes.	Runs user applications and non-critical services.
Memory Access	Can access both kernel and user-space memory.	Can only access memory within its own process; cannot access kernel memory.
Stability	Kernel mode errors can cause the entire system to crash (e.g., Blue Screen of Death in Windows).	User mode errors typically only crash the affected application without affecting the entire system.
System Calls	Handles system calls and interacts with hardware directly.	Makes system calls to request services from the kernel (e.g., file operations, memory management).
Performance	Higher performance for critical tasks due to direct access to hardware and fewer restrictions.	Slower for hardware access due to the need for system calls to transition to kernel mode.
Process Isolation	Kernel processes are not isolated; a fault in one can affect others.	User processes are isolated from each other; a fault in one does not directly affect others.
Examples	Device drivers, memory management, process scheduling.	Web browsers, text editors, games, office applications.
Mode Switching	Switching from kernel to user mode occurs after a system call is processed.	Switching from user to kernel mode occurs when a system call is made.



OS structures

Monolithic Structure

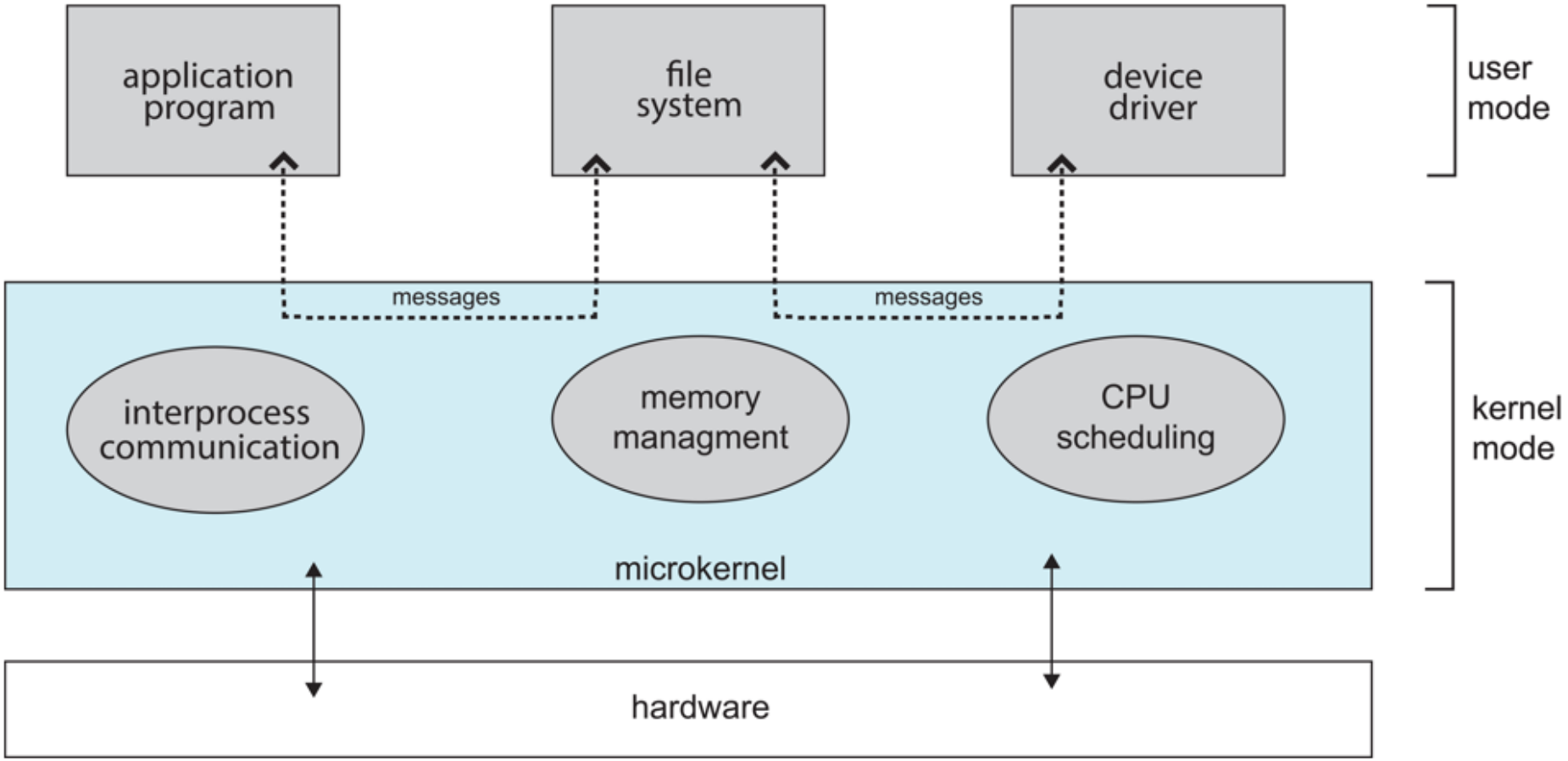
```
class MonolithicOS:
    def __init__(self):
        self.file_system = FileSystem()
        self.memory_manager = MemoryManager()
        self.process_manager = ProcessManager()

    def boot(self):
        print("Booting Monolithic OS...")
        self.file_system.initialize()
        self.memory_manager.initialize()
        self.process_manager.initialize()

class FileSystem:
    def initialize(self):
        print("File System Initialized")

class MemoryManager:
    def initialize(self):
        print("Memory Manager Initialized")

class ProcessManager:
    def initialize(self):
        print("Process Manager Initialized")

# Example usage
os = MonolithicOS()
os.boot()
```



Layered Structure

```
class LayeredOS:
    def __init__(self):
        self.layer1 = HardwareLayer()
        self.layer2 = KernelLayer(self.layer1)
        self.layer3 = ApplicationLayer(self.layer2)
    def boot(self):
        print("Booting Layered OS...")
        self.layer3.start()

class HardwareLayer:
    def run_hardware(self):
        print("Hardware Running")

class KernelLayer:
    def __init__(self, hardware_layer):
        self.hardware_layer = hardware_layer
    def run_kernel(self):
        print("Kernel Running")
        self.hardware_layer.run_hardware()

class ApplicationLayer:
    def __init__(self, kernel_layer):
        self.kernel_layer = kernel_layer
    def start(self):
        print("Starting Application Layer")
        self.kernel_layer.run_kernel()

# Example usage
os = LayeredOS()
os.boot()
```

Microkernel System Structure



Microkernel Structure

```
import queue
import threading
class Microkernel:
    def __init__(self):
        self.services = {}
        self.message_queue = queue.Queue()
    def register_service(self, name, service):
        self.services[name] = service
    def send_message(self, service_name, message):
        self.message_queue.put((service_name, message))
    def run(self):
        while not self.message_queue.empty():
            service_name, message = self.message_queue.get()
            if service_name in self.services:
                self.services[service_name].handle_message(message)
class FileSystemService:
    def handle_message(self, message):
        print(f"FileSystemService: {message}")
class NetworkService:
    def handle_message(self, message):
        print(f"NetworkService: {message}")
# Example usage
microkernel = Microkernel()
fs_service = FileSystemService()
network_service = NetworkService()
microkernel.register_service("FileSystem", fs_service)
microkernel.register_service("Network", network_service)
microkernel.send_message("FileSystem", "Read File")
microkernel.send_message("Network", "Send Data")
microkernel.run()
```

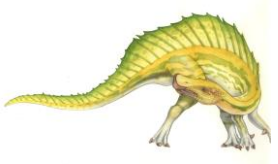
Modular Structure

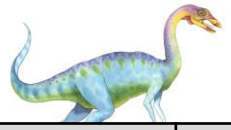
```
class ModularOS:
    def __init__(self):
        self.modules = {}
    def load_module(self, name, module):
        self.modules[name] = module
        print(f"Module {name} loaded")
    def unload_module(self, name):
        if name in self.modules:
            del self.modules[name]
            print(f"Module {name} unloaded")
    def execute_module(self, name):
        if name in self.modules:
            self.modules[name].execute()
        else:
            print(f"Module {name} not found")
class FileSystemModule:
    def execute(self):
        print("Executing File System Module")
class NetworkModule:
    def execute(self):
        print("Executing Network Module")
# Example usage
os = ModularOS()
fs_module = FileSystemModule()
network_module = NetworkModule()
os.load_module("FileSystem", fs_module)
os.load_module("Network", network_module)
os.execute_module("FileSystem")
os.execute_module("Network")
os.unload_module("FileSystem")
```

Hybrid Structure

Example of creating a system where:

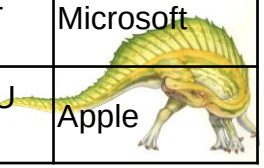
- **Core Services (like memory management and process scheduling)** are implemented in a **monolithic** fashion within a single kernel module for performance reasons.
- **Additional Services (like file system and network management)** are implemented as separate user-space services, similar to the **microkernel** approach, communicating with the core through a message-passing mechanism.





Comparing the different operating system structures

Structure	Description	Advantages	Disadvantages	Example System	Company
Monolithic	Single, large kernel with all OS services running in kernel mode.	High performance due to fast system calls and minimal overhead. Simplicity in design and implementation.	• Stability and security risks due to lack of isolation between components. • Difficult to maintain and update due to large, interconnected codebase.	Linux	Linux Foundation
				Unix (traditional)	AT&T (historically)
Layered	OS is divided into multiple layers, each built on top of the previous one.	Modularity makes it easier to develop, test, and debug. Security and stability due to isolation of functions.	• Performance overhead due to inter-layer communication. • Complexity in designing layers and managing dependencies.	THE (Technische Hogeschool Eindhoven)	N/A (research)
				Windows NT (conceptually)	Microsoft
Microkernel	Minimal kernel; most OS services run in user space.	High reliability and security due to minimal kernel. Easy to update and maintain services independently.	• Higher performance overhead due to context switching. • Complexity in implementing robust IPC mechanisms.	macOS (Darwin kernel)	Apple
				QNX	BlackBerry Limited
Modular	Kernel with dynamically loadable modules that can be added or removed at runtime.	Flexibility to update or replace modules without rebooting. Combines performance benefits of monolithic kernels with flexibility.	• Complexity in managing dependencies and interactions between modules. • Potential instability if modules are not managed carefully.	Linux (modern distributions)	Linux Foundation
				Solaris	Oracle Corporation
Hybrid	Combines elements of monolithic and microkernel architectures.	Balances performance, modularity, and flexibility. Can efficiently scale and support diverse hardware.	• Complexity in design and implementation. • Potential for system bloat due to the combination of approaches.	Windows NT	Microsoft
				macOS (XNU kernel)	Apple

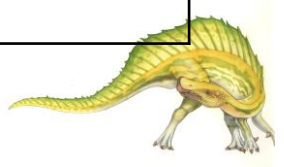


Emulation vs Virtualization

Aspect	Emulation	Virtualization
Definition	Simulates the hardware of a different system.	Creates virtual instances of the same hardware on which multiple OS can run.
Performance	Slower due to instruction translation and hardware simulation.	Near-native performance, especially with hardware-assisted virtualization.
Use Cases	Running software for different architectures, preserving legacy systems.	Running multiple OS on one machine, server consolidation, cloud computing.
Hardware Dependency	Not dependent on host hardware matching guest hardware.	Generally requires the same architecture between host and guest.
Complexity	More complex due to the need to accurately simulate hardware.	Less complex in terms of hardware simulation, but hypervisor management can be complex.

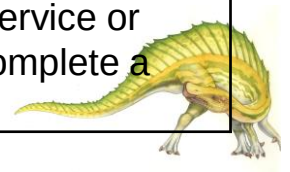
Linker vs Loader

Aspect	Linker	Loader
Function	Combines object files and resolves symbol references to create an executable.	Loads the executable into memory and prepares it for execution.
Operation Time	Operates during the compilation phase, after source code is compiled into object files.	Operates during the execution phase, when the program is about to run.
Output	Generates an executable file (e.g., .exe, .out).	Loads the executable file into memory and initializes the program's runtime environment.
Tasks	- Symbol resolution - Address binding - Code and data relocation - Merging multiple object files - Handling library linking	- Loading executable into memory - Setting up the memory space - Resolving runtime addresses (for dynamic linking) - Starting the program execution by passing control to the entry point
Types	- Static linker - Dynamic linker	- Absolute loader - Relocating loader - Dynamic loader
Examples	- GNU ld, gold (Linux/Unix) - link.exe (Windows)	- OS kernel loader (loads executables into memory) - Dynamic linker/loader (ld.so on Unix/Linux)



System Call vs System Service

Aspect	System Call	System Service
Definition	A mechanism that allows user-level processes to request services from the kernel.	Functions or operations provided by the operating system to perform tasks.
Functionality	Provides an interface for the process to interact with the operating system kernel.	Implements the actual operation or service requested by a system call.
Usage	Used by applications to perform tasks that require access to hardware or other privileged operations.	Executes tasks like file operations, process control, memory management, etc.
Examples	read(), write(), open(), fork(), exec()	File system management, process scheduling, memory allocation, I/O management.
Execution Context	Initiated by the user-space process, but executed in kernel space.	Typically executes in kernel space, responding to system calls.
Security Level	Requires switching from user mode to kernel mode, with higher privilege.	Operates in kernel mode, with full access to system resources.
Overhead	Involves overhead due to context switching between user space and kernel space.	Runs within the operating system, often optimized for efficiency.
Accessibility	Directly accessible to user applications through specific APIs or functions.	Not directly accessible; invoked through system calls.
Purpose	Acts as a gateway between user programs and the operating system's services.	Provides the actual service or function needed to complete a system call.



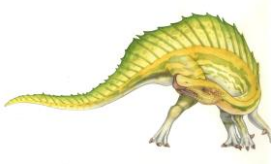
System Call vs System Service

```python
class OperatingSystem:
    def __init__(self):
        self.is_kernel_mode = False

    def system_call(self, operation, *args):
        """Simulate a system call which switches to kernel mode."""
        print("Switching to kernel mode...")
        self.is_kernel_mode = True

        if operation == "open_file":
            self._open_file(*args)
        elif operation == "allocate_memory":
            self._allocate_memory(*args)
        elif operation == "send_data":
            self._send_data(*args)
        else:
            print(f"Unknown operation: {operation}")

        print("Switching back to user mode...")
        self.is_kernel_mode = False

    def _open_file(self, filename):
        print(f"Kernel: Opening file '{filename}'")

    def _allocate_memory(self, size):
        print(f"Kernel: Allocating {size} bytes of memory")

    def _send_data(self, data):
        print(f"Kernel: Sending data '{data}'")
```

```python
# Example user-mode functions
def user_mode_application(os):
    print("User Mode: Requesting to open a file")
    os.system_call("open_file", "example.txt")

    print("User Mode: Requesting memory allocation")
    os.system_call("allocate_memory", 1024)

    print("User Mode: Requesting to send data over the network")
    os.system_call("send_data", "Hello, World!")

# Example usage
os = OperatingSystem()
user_mode_application(os)
```

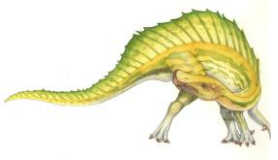
Outline Chapter 3

- Process Concept
- Process Scheduling
- Operations on Processes
- Interprocess Communication
- IPC in Shared-Memory Systems
- IPC in Message-Passing Systems
- Examples of IPC Systems
- Communication in Client-Server Systems



Objectives

- Identify the **separate components** of a process and illustrate how they are represented and scheduled in an operating system.
- Describe **how processes are created and terminated** in an operating system, including developing programs using the appropriate system calls that perform these operations.
- Describe and contrast **interprocess communication** using shared memory and message passing.
- Design programs that uses pipes and POSIX shared memory to perform interprocess communication.
- Describe client-server communication using sockets and remote procedure calls.
- Design kernel modules that interact with the Linux operating system.

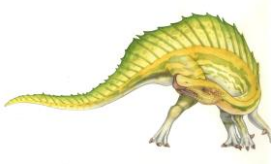


Process Concept

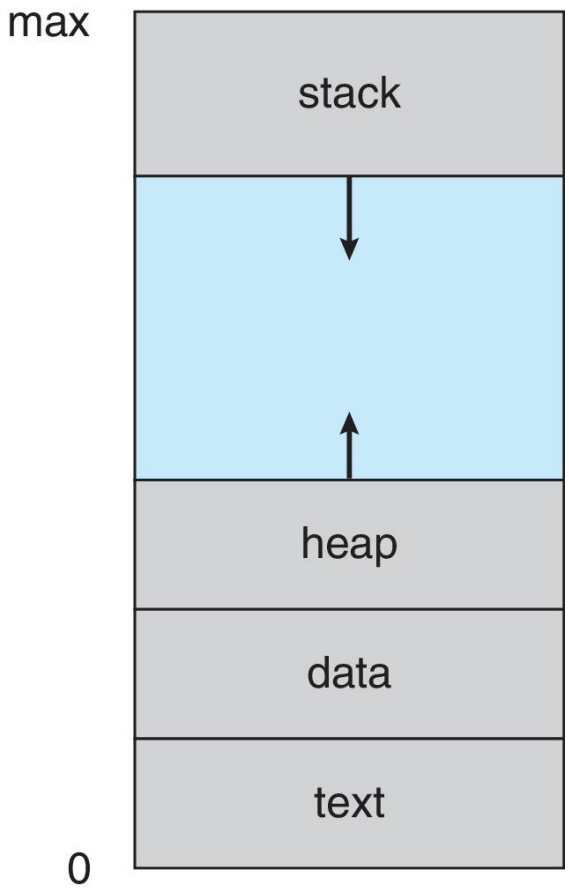
- An operating system executes a variety of programs that run as a process.
- **Process** – a program in **execution**; process execution must progress in sequential fashion. No parallel execution of instructions of a single process
- Multiple parts
  - The program code, also called **text section**
  - Current activity including **program counter**, processor registers
  - **Stack** containing temporary data
    - Function parameters, return addresses, local variables
  - **Data section** containing global variables
  - **Heap** containing memory dynamically allocated during run time

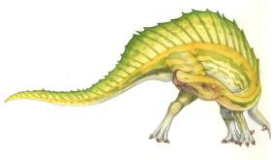
Process Concept (Cont.)

- Program is **passive** entity stored on disk (**executable file**); process is **active**
  - Program becomes process when an executable file is loaded into memory
- Execution of program started via GUI mouse clicks, command line entry of its name, etc.
- One program can be several processes
  - Consider multiple users executing the same program

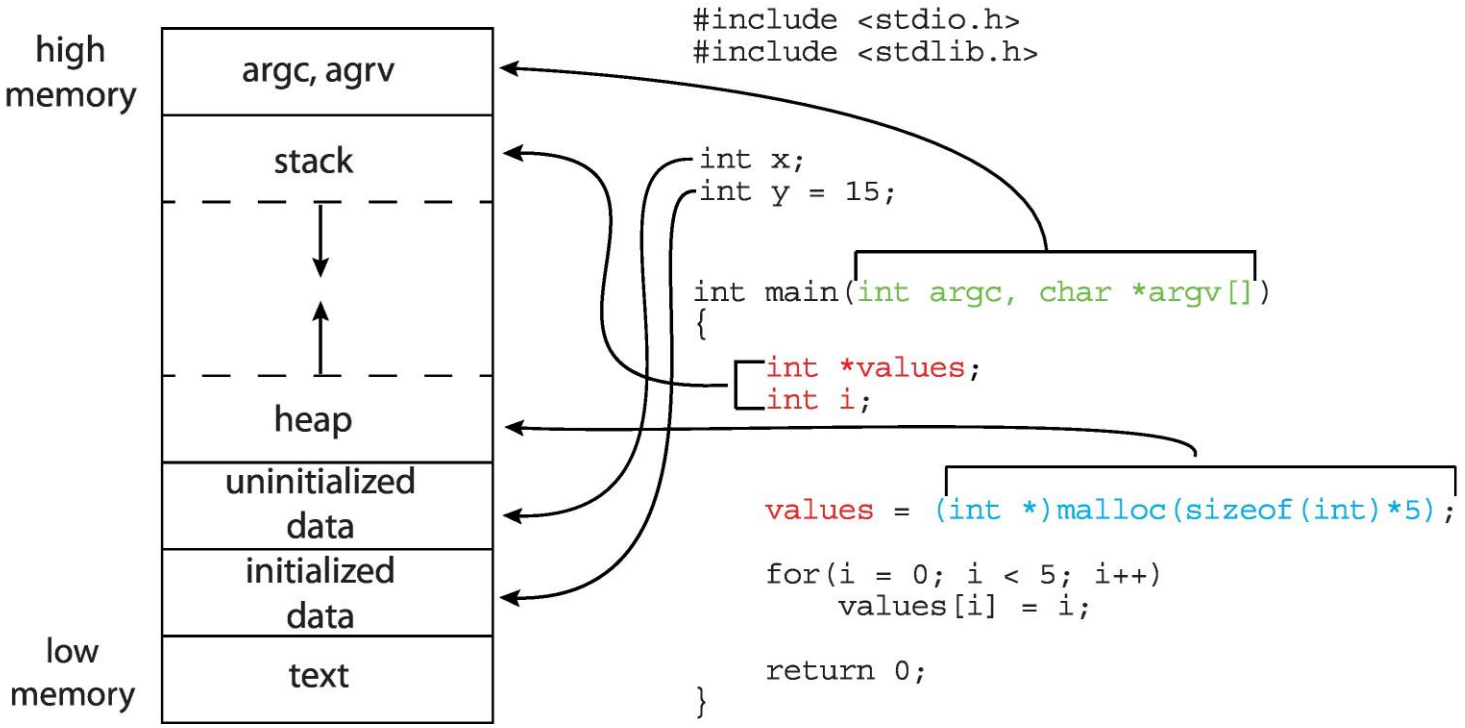


Process in Memory



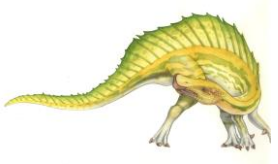


Memory Layout of a C Program



For 32-bit processor: 2^{32} = 4 billion (10^9) addresses

For 64-bit processor: 2^{64} = 18 quadrillion (10^{18}) addresses



Process State

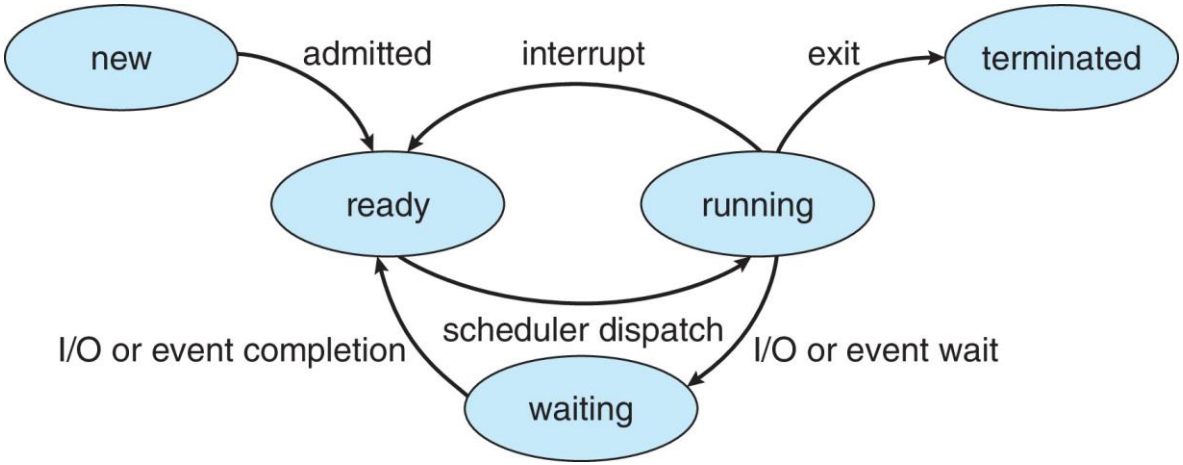
- As a process executes, it changes **state**
  - **New**: The process is being created
  - **Running**: Instructions are being executed
  - **Waiting**: The process is waiting for some event to occur
  - **Ready**: The process is waiting to be assigned to a processor
  - **Terminated**: The process has finished execution

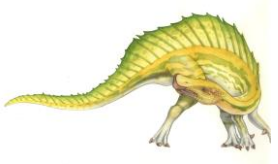
Specific implementations may include variations or additional states depending on the requirements of the operating system:

- **Suspended** (or Swapped Out): The process is not currently in memory, and its execution is paused until it is brought back into memory. This state is used in systems that implement swapping or paging.

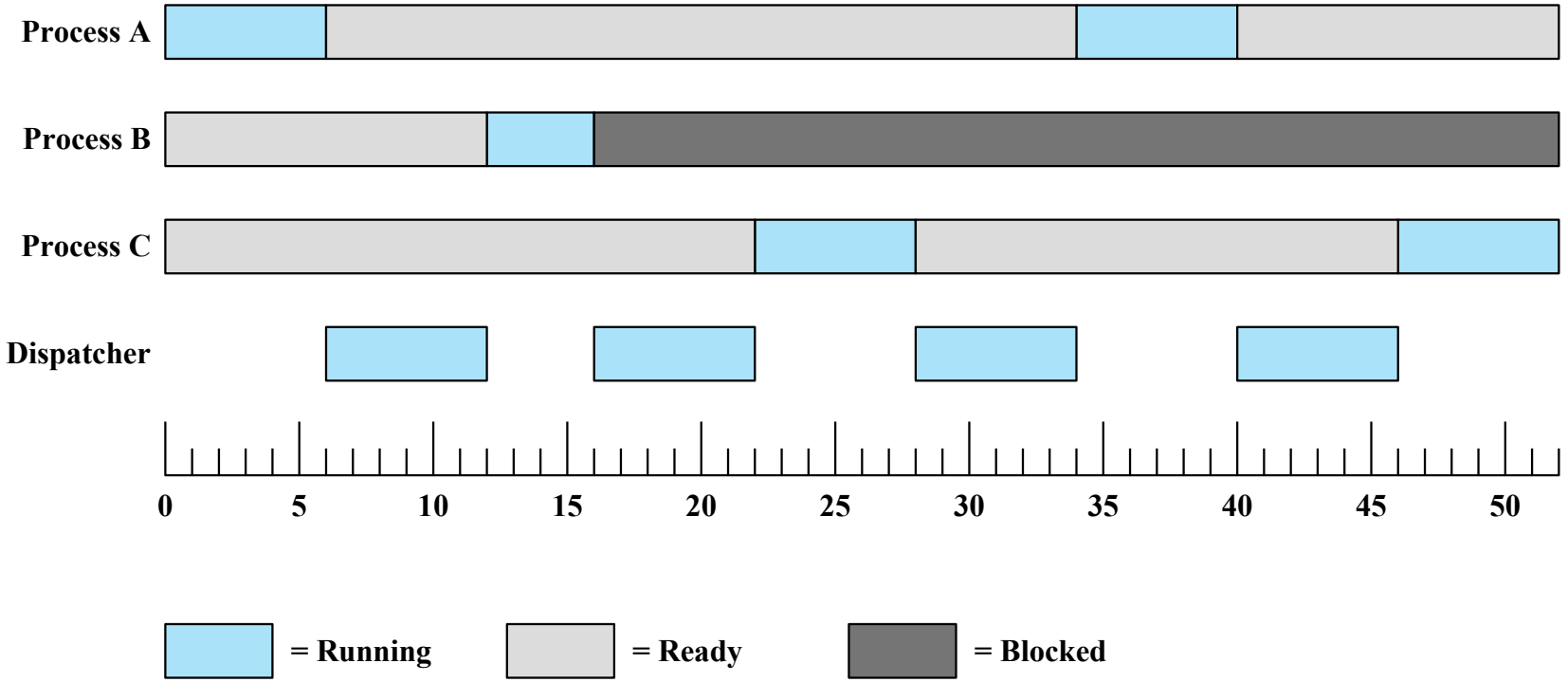
- **Zombie**: The process has terminated, but its parent process has not yet retrieved its exit status. It remains in the process table as a "zombie."

Diagram of Process State





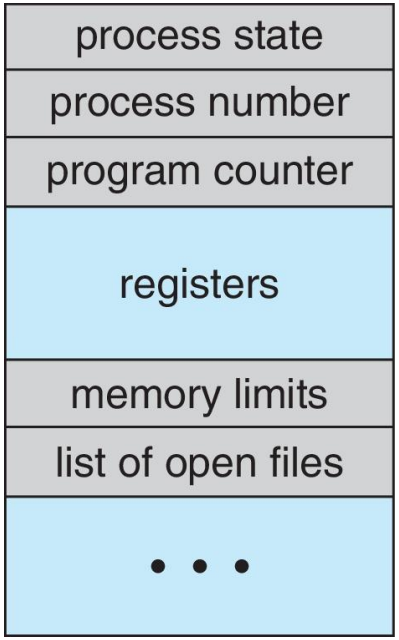
Process States

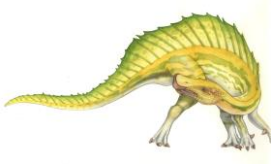


Process Control Block (PCB)

Information associated with each process(also called **task control block**)

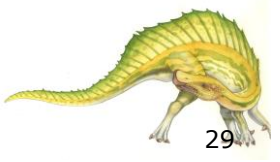
- **Process state** – running, waiting, etc.
- **Program counter** – location of instruction to next execute
- **CPU registers** – contents of all process-centric registers
- **CPU scheduling** information- priorities, scheduling queue pointers
- **Memory-management information** – memory allocated to the process
- **Accounting information** – CPU used, clock time elapsed since start, time limits
- **I/O status information** – I/O devices allocated to process, list of open files





PCB (Process Control Block)

- **Process Identification**: Tracks the unique ID for each process.
- **Process Status**: Tracks whether the process is in user mode or kernel mode.
- **Process State**: Defines the current state (running, waiting, etc.).
- **Process Status Word**: CPU status information like the program counter and interrupt status.
- **Register Contents**: Saves CPU registers for context switching.
- **Main Memory**: Manages the memory allocation for the process.
- **Resources**: Tracks system resources (files, devices) that the process is using.
- **Process Priority**: Determines the order of execution based on importance.
- **Accounting**: Tracks usage statistics like CPU time and I/O operations.

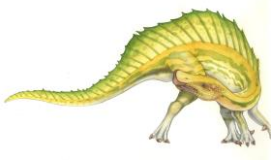


Threads

- So far, process has a single thread of execution
- Consider having multiple program counters per process
  - Multiple locations can execute at once
    - Multiple threads of control -> **threads**
- Must then have storage for thread details, multiple program counters in PCB
- Explore in detail in Chapter 4

TCB (Thread Control Block)

- Thread identification
- Thread state
- CPU information:
  - Program counter
  - Register contents
  - Thread priority
- Pointer to process that created this thread
- Pointers to all other threads created by this thread



Motivational Example for Threads

- Imagine the following C program:

```
main() {
    ComputePI("pi.txt");
    PrintClassList("classlist.txt");
}
```

- What is the behavior here?
  - Program would never print out class list
  - Why? `ComputePI` would never finish

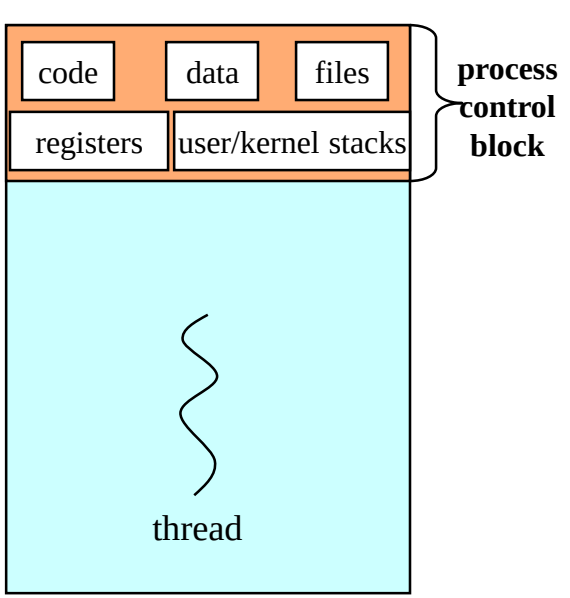
Adapted from: Prof. John Kubiatowicz
http://cs162.eecs.Berkeley.edu

Threading in Python

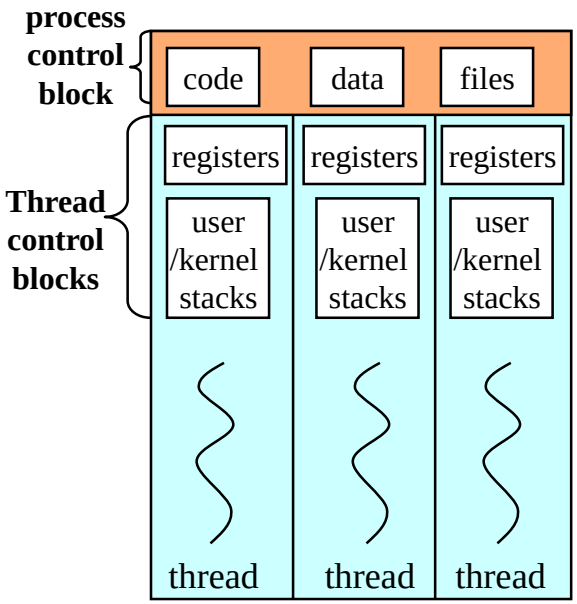
```
import threading
import time

def print_numbers():
    for i in range(1, 6):
        print(f"Number: {i}")
        time.sleep(1)  # Simulate a time-consuming task

def print_letters():
    for letter in ['A', 'B', 'C', 'D', 'E']:
        print(f"Letter: {letter}")
        time.sleep(1.5)  # Simulate a time-consuming task

# Create two threads
thread1 = threading.Thread(target=print_numbers)
thread2 = threading.Thread(target=print_letters)

# Start the threads
thread1.start()
thread2.start()

# Wait for both threads to complete
thread1.join()
thread2.join()

print("Both threads have finished execution.")
```

```
Number: 1Letter: A

Number: 2
Letter: B
Number: 3
Letter: C
Number: 4
Number: 5
Letter: D
Letter: E
Both threads have finished execution.
```

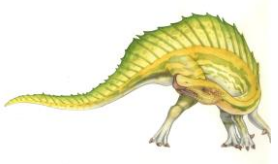
Single and multithreaded processes



single threaded

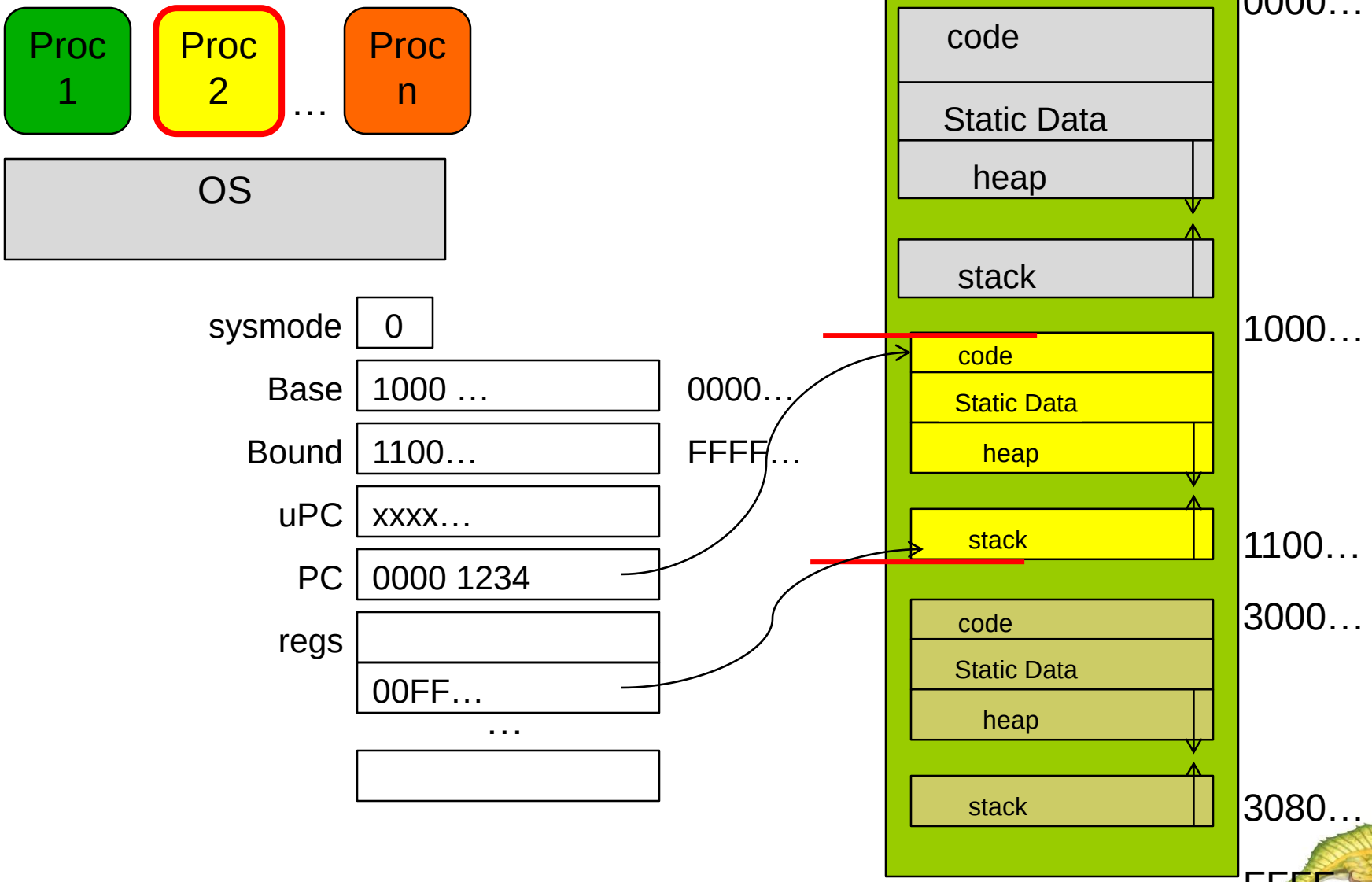


multithreaded

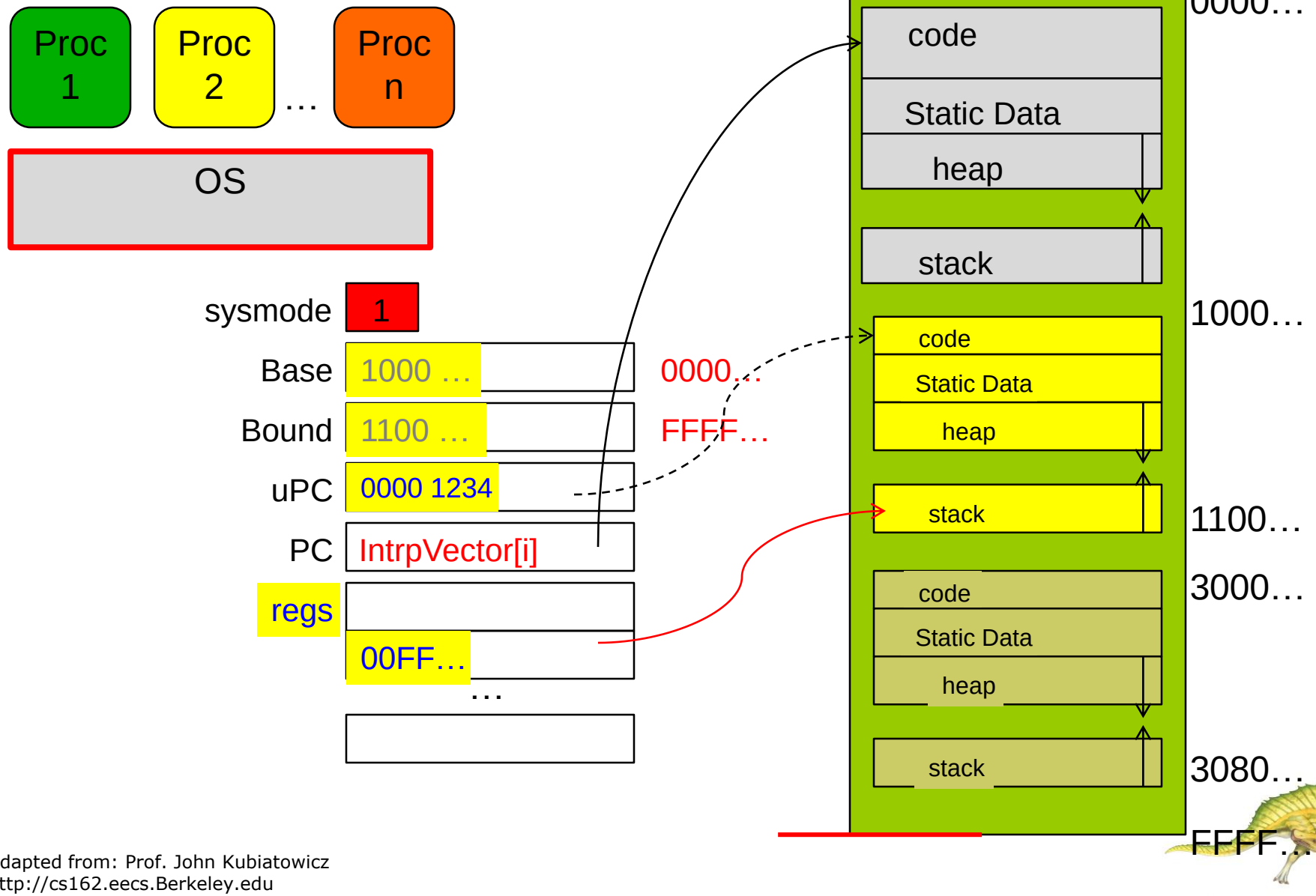


Feature	Process	Thread
Definition	An independent program in execution with its own memory space	A unit of execution within a process that shares memory and resources with other threads
Memory	Each process has its own memory space (isolated)	Threads within the same process share memory space
Communication	Communication between processes is complex and slower (e.g., inter-process communication mechanisms like pipes or message queues)	Communication between threads is easy and fast as they share memory
Creation Time	Slower to create due to memory allocation and initialization	Faster to create as threads share resources of the parent process
Overhead	Higher overhead due to separate memory space and resource allocation	Lower overhead since threads share memory and resources
Context Switching	More expensive due to separate memory and context	Less expensive as threads share the same memory space
Execution	Each process can run independently	Threads within the same process must cooperate and share resources
Crash Impact	A crash in one process does not affect other processes	A crash in a thread can potentially crash the entire process
Resource Sharing	Processes do not share resources unless explicitly done via IPC	Threads share resources like data, memory, and file descriptors within the same process
Multitasking	Enables multitasking across different programs	Allows concurrent execution within a single program

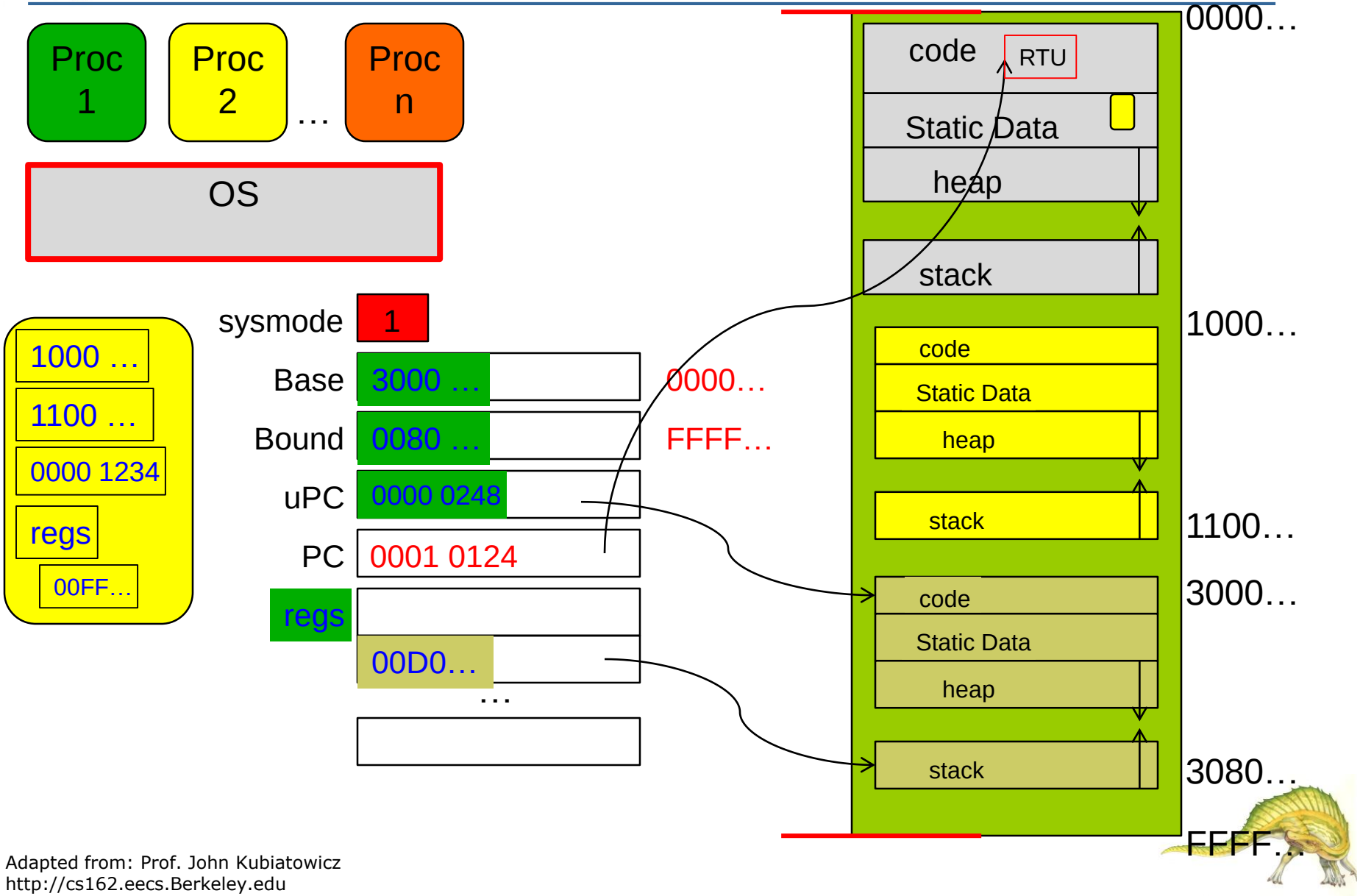
Simple Base & Bound: User => Kernel



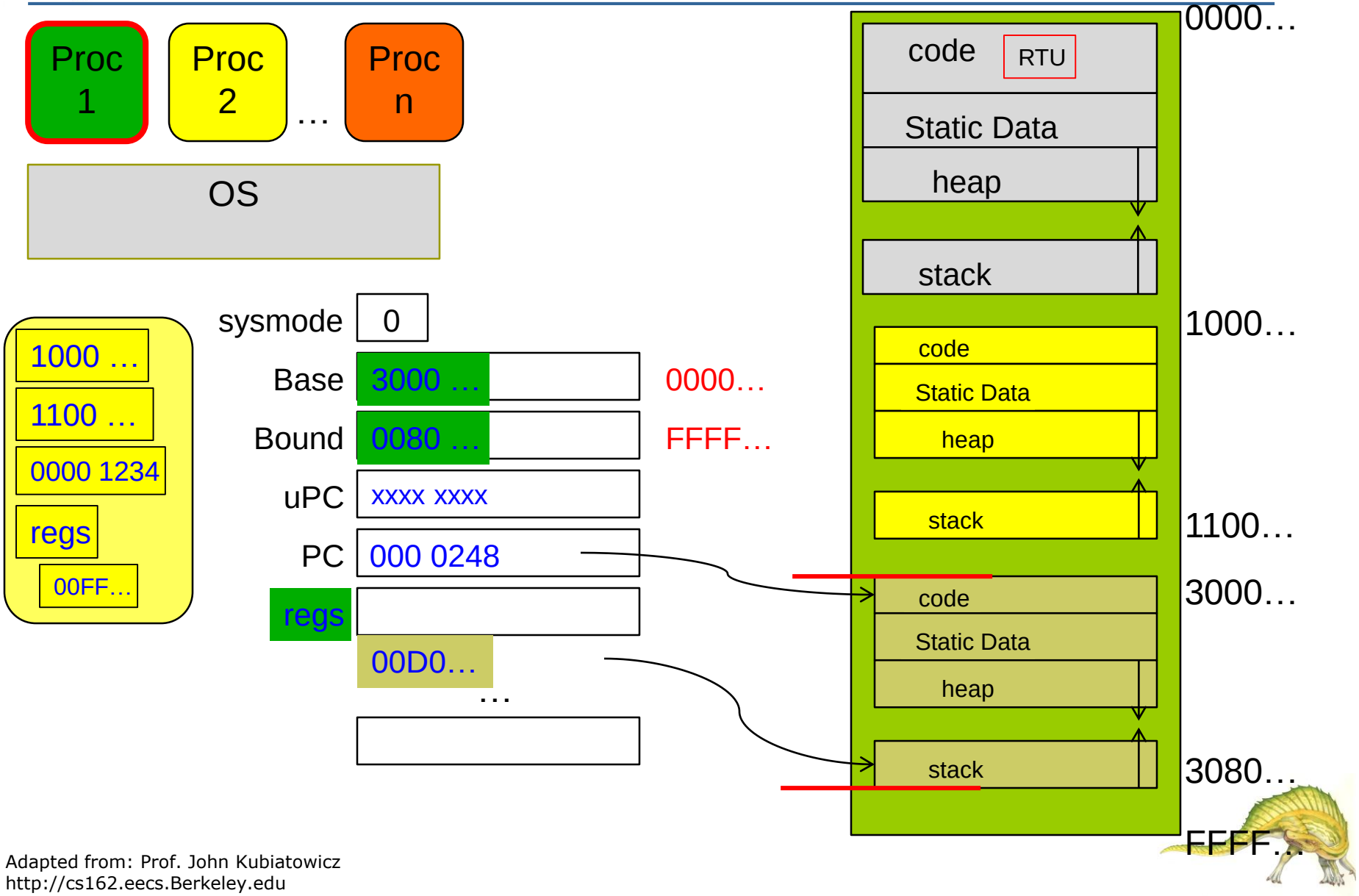
Simple B&B: Interrupt

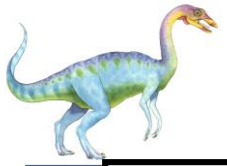


Simple B&B: Switch User Process



Simple B&B: “resume”



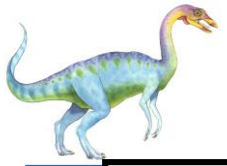


Execution Stack Example

```
        A(int tmp) {
  A:      if (tmp<2)
A+1:        B();
A+2:      printf(tmp);
        }
        B() {
  B:      C();
B+1:    }
        C() {
  C:      A(2);
C+1:    }
        A(1);
exit:
```

- Stack holds temporary results
- Permits recursive execution
- Crucial to modern languages

Adapted from: Prof. John Kubiatowicz
http://cs162.eecs.Berkeley.edu



Execution Stack Example

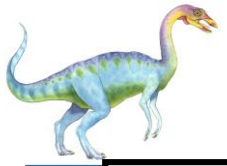
```
        A(int tmp) {
A:        if (tmp<2)
A+1:        B();
A+2:      printf(tmp);
        }
        B() {
B:        C();
B+1:    }
        C() {
C:        A(2);
C+1:    }
        A(1);
exit:
```

Stack Pointer →

```
A: tmp=1
   ret=exit
```

- Stack holds temporary results
- Permits recursive execution
- Crucial to modern languages

Adapted from: Prof. John Kubiatowicz
http://cs162.eecs.Berkeley.edu



Execution Stack Example

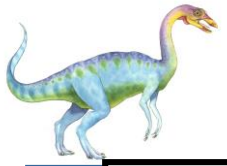
```
      A(int tmp) {
A:      if (tmp<2)
A+1:      B();
A+2:    printf(tmp);
      }
      B() {
B:      C();
B+1:  }
      C() {
C:      A(2);
C+1:  }
      A(1);
exit:
```

Stack Pointer →

```
A: tmp=1
   ret=exit
```

- Stack holds temporary results
- Permits recursive execution
- Crucial to modern languages

Adapted from: Prof. John Kubiatowicz
http://cs162.eecs.Berkeley.edu



Execution Stack Example

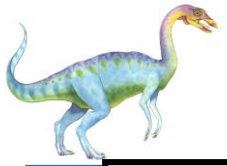
```
        A(int tmp) {
A:        if (tmp<2)
A+1:        B();
A+2:      printf(tmp);
        }
        B() {
B:        C();
B+1:    }
        C() {
C:        A(2);
C+1:    }
        A(1);
exit:
```

Stack Pointer →

```
A: tmp=1
   ret=exit
```

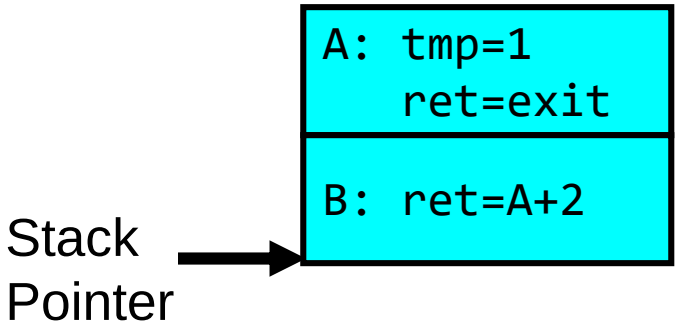
- Stack holds temporary results
- Permits recursive execution
- Crucial to modern languages

Adapted from: Prof. John Kubiatowicz
http://cs162.eecs.Berkeley.edu



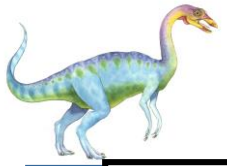
Execution Stack Example

```
        A(int tmp) {
A:        if (tmp<2)
A+1:        B();
A+2:      printf(tmp);
        }
        B() {
B:        C();
B+1:    }
        C() {
C:        A(2);
C+1:    }
        A(1);
exit:
```



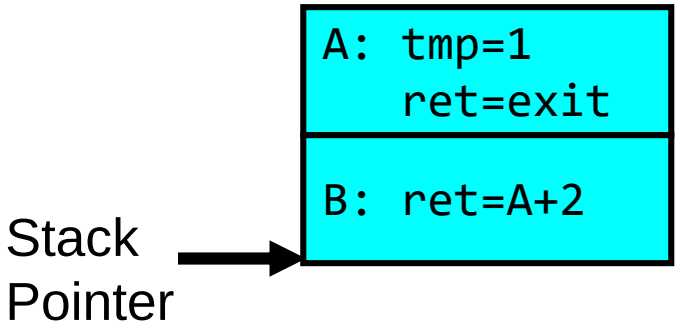
- Stack holds temporary results
- Permits recursive execution
- Crucial to modern languages

Adapted from: Prof. John Kubiatowicz
http://cs162.eecs.Berkeley.edu



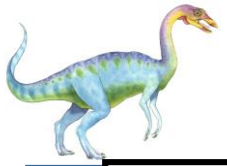
Execution Stack Example

```
        A(int tmp) {
A:        if (tmp<2)
A+1:        B();
A+2:      printf(tmp);
        }
        B() {
B:        C();
B+1:    }
        C() {
C:        A(2);
C+1:    }
        A(1);
exit:
```



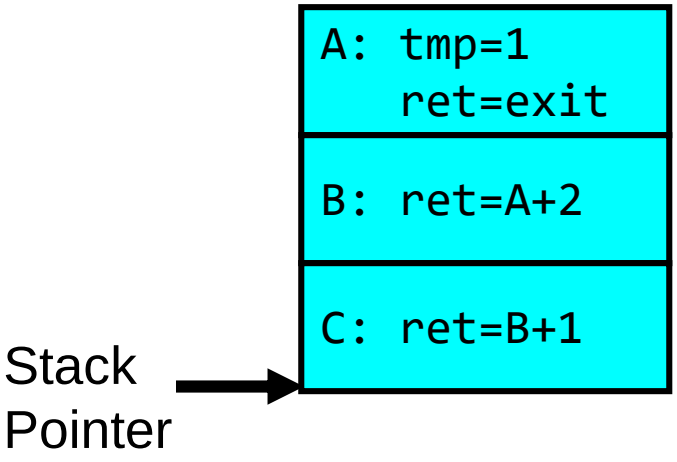
- Stack holds temporary results
- Permits recursive execution
- Crucial to modern languages

Adapted from: Prof. John Kubiatowicz
http://cs162.eecs.Berkeley.edu



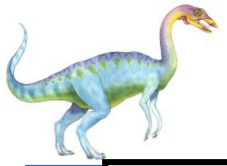
Execution Stack Example

```
        A(int tmp) {
A:        if (tmp<2)
A+1:        B();
A+2:      printf(tmp);
        }
        B() {
B:        C();
B+1:    }
        C() {
C:        A(2);
C+1:    }
        A(1);
exit:
```



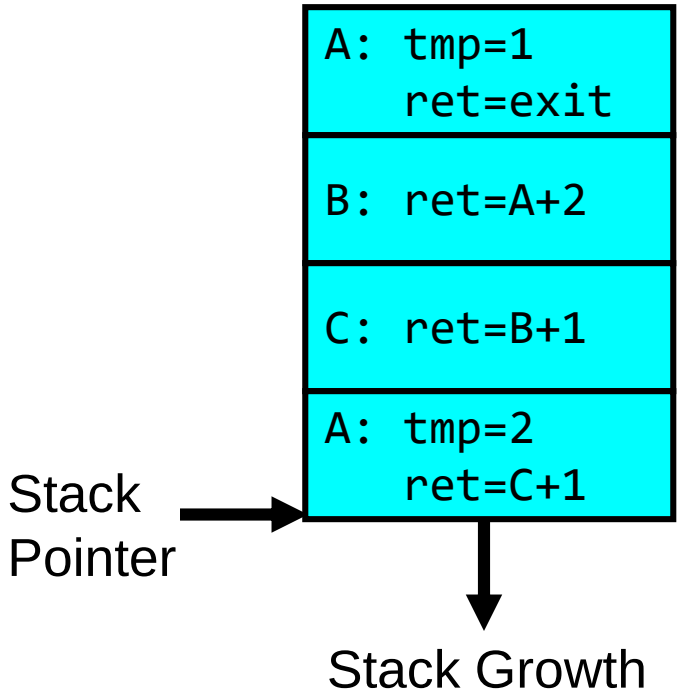
- Stack holds temporary results
- Permits recursive execution
- Crucial to modern languages

Adapted from: Prof. John Kubiatowicz
http://cs162.eecs.Berkeley.edu



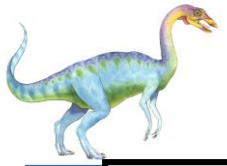
Execution Stack Example

```
        A(int tmp) {
A:        if (tmp<2)
A+1:        B();
A+2:      printf(tmp);
        }
        B() {
B:        C();
B+1:    }
        C() {
C:        A(2);
C+1:    }
        A(1);
exit:
```



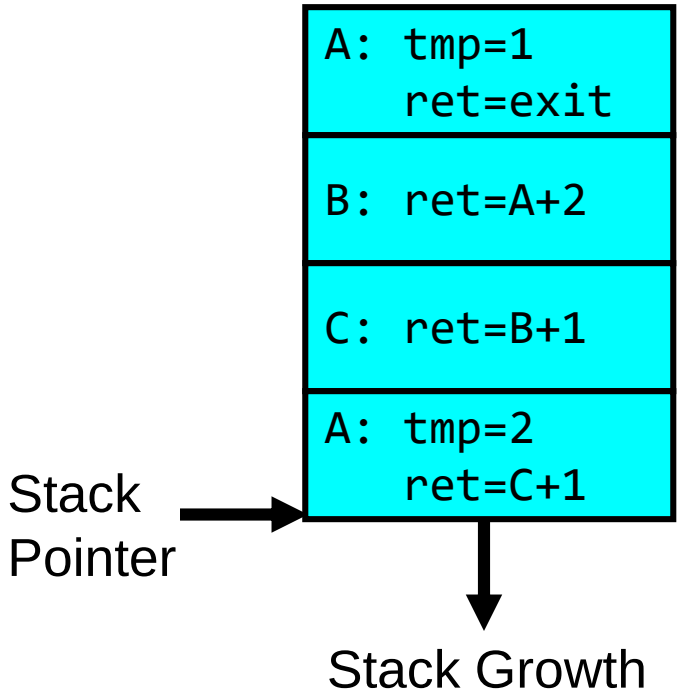
- Stack holds temporary results
- Permits recursive execution
- Crucial to modern languages

Adapted from: Prof. John Kubiatowicz
http://cs162.eecs.Berkeley.edu



Execution Stack Example

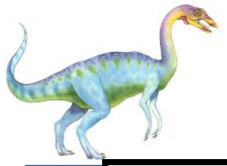
```
        A(int tmp) {
A:        if (tmp<2)
A+1:        B();
A+2:      printf(tmp);
        }
        B() {
B:        C();
B+1:    }
        C() {
C:        A(2);
C+1:    }
        A(1);
exit:
```



Output: `>2`

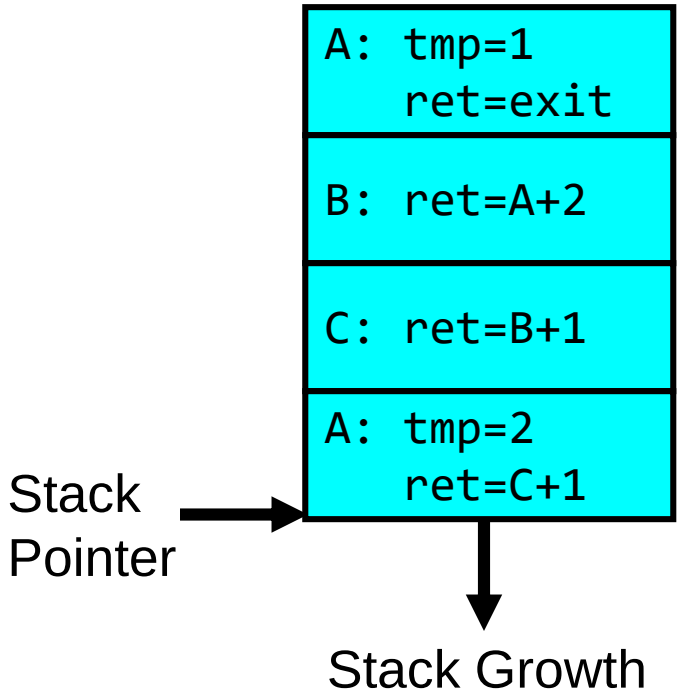
- Stack holds temporary results
- Permits recursive execution
- Crucial to modern languages

Adapted from: Prof. John Kubiatowicz
http://cs162.eecs.Berkeley.edu



Execution Stack Example

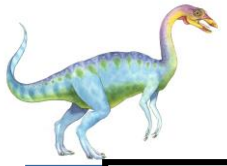
```
        A(int tmp) {
A:        if (tmp<2)
A+1:        B();
A+2:      printf(tmp);
        }
        B() {
B:        C();
B+1:    }
        C() {
C:        A(2);
C+1:    }
        A(1);
exit:
```



Output: `>2`

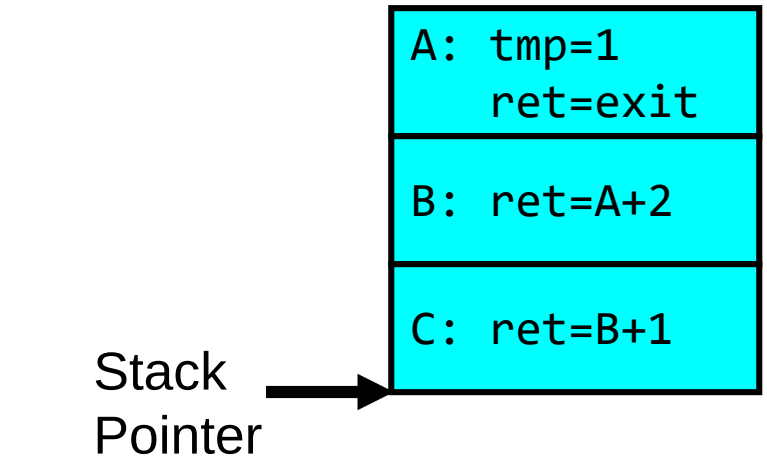
- Stack holds temporary results
- Permits recursive execution
- Crucial to modern languages

Adapted from: Prof. John Kubiatowicz
http://cs162.eecs.Berkeley.edu



Execution Stack Example

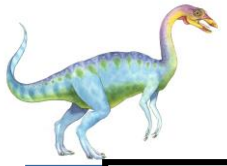
```
        A(int tmp) {
  A:      if (tmp<2)
A+1:        B();
A+2:      printf(tmp);
        }
        B() {
  B:      C();
B+1:    }
        C() {
  C:      A(2);
C+1:    }
        A(1);
exit:
```



Output: >2

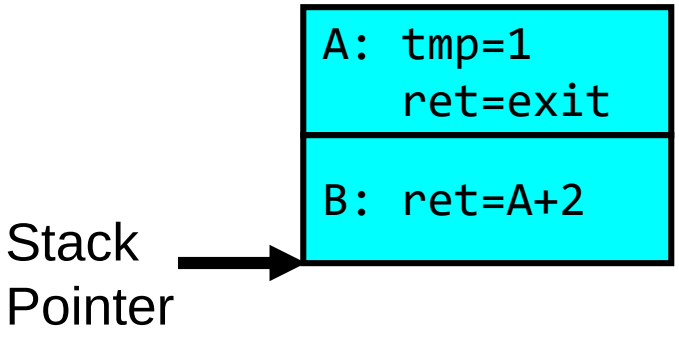
- Stack holds temporary results
- Permits recursive execution
- Crucial to modern languages

Adapted from: Prof. John Kubiatowicz
http://cs162.eecs.Berkeley.edu



Execution Stack Example

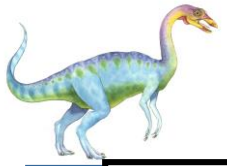
```
        A(int tmp) {
A:        if (tmp<2)
A+1:        B();
A+2:      printf(tmp);
        }
        B() {
B:        C();
B+1:    }
        C() {
C:        A(2);
C+1:    }
        A(1);
exit:
```



Output: `>2`

- Stack holds temporary results
- Permits recursive execution
- Crucial to modern languages

Adapted from: Prof. John Kubiatowicz
http://cs162.eecs.Berkeley.edu



Execution Stack Example

```
        A(int tmp) {
A:        if (tmp<2)
A+1:        B();
A+2:      printf(tmp);
        }
        B() {
B:        C();
B+1:    }
        C() {
C:        A(2);
C+1:    }
        A(1);
exit:
```

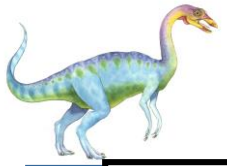
Stack Pointer →

```
A: tmp=1
   ret=exit
```

Output: `>2 1`

- Stack holds temporary results
- Permits recursive execution
- Crucial to modern languages

Adapted from: Prof. John Kubiatowicz
http://cs162.eecs.Berkeley.edu



Execution Stack Example

```
        A(int tmp) {
A:        if (tmp<2)
A+1:        B();
A+2:      printf(tmp);
        }
        B() {
B:        C();
B+1:    }
        C() {
C:        A(2);
C+1:    }
        A(1);
exit:
```

Stack Pointer →

```
A: tmp=1
   ret=exit
```

Output: `>2 1`

- Stack holds temporary results
- Permits recursive execution
- Crucial to modern languages

Adapted from: Prof. John Kubiatowicz
http://cs162.eecs.Berkeley.edu

Execution Stack Example

```
A(int tmp) {
  if (tmp<2)
    B();
  printf(tmp);
}
B() {
  C();
}
C() {
  A(2);
}
A(1);
```

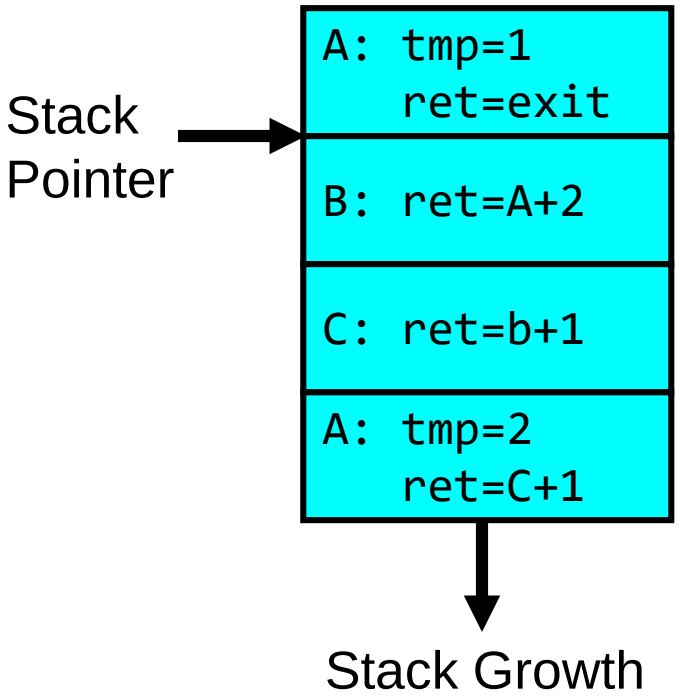
Output: `>2 1`

- Stack holds temporary results
- Permits recursive execution
- Crucial to modern languages

Adapted from: Prof. John Kubiatowicz
http://cs162.eecs.Berkeley.edu

Execution Stack Example

```
A(int tmp) {
  if (tmp<2)
    B();
  printf(tmp);
}
B() {
  C();
}
C() {
  A(2);
}
A(1);
```



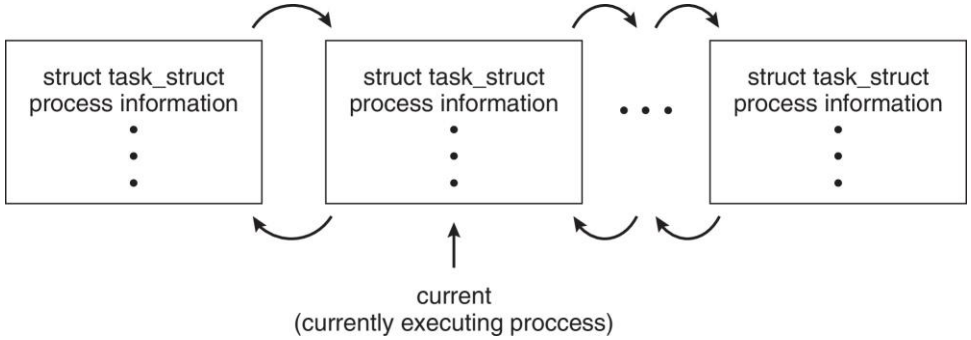
- Stack holds temporary results
- Permits recursive execution
- Crucial to modern languages

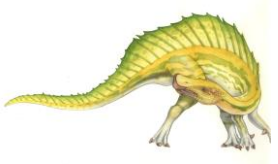
Adapted from: Prof. John Kubiatowicz
http://cs162.eecs.Berkeley.edu

Process Representation in Linux

Represented by the C structure `task_struct`

```
pid t_pid;                    /* process identifier */
long state;                   /* state of the process */
unsigned int time_slice       /* scheduling information */
struct task_struct *parent;/* this process's parent */
struct list_head children; /* this process's children */
struct files_struct *files;/* list of open files */
struct mm_struct *mm;         /* address space of this process */
```





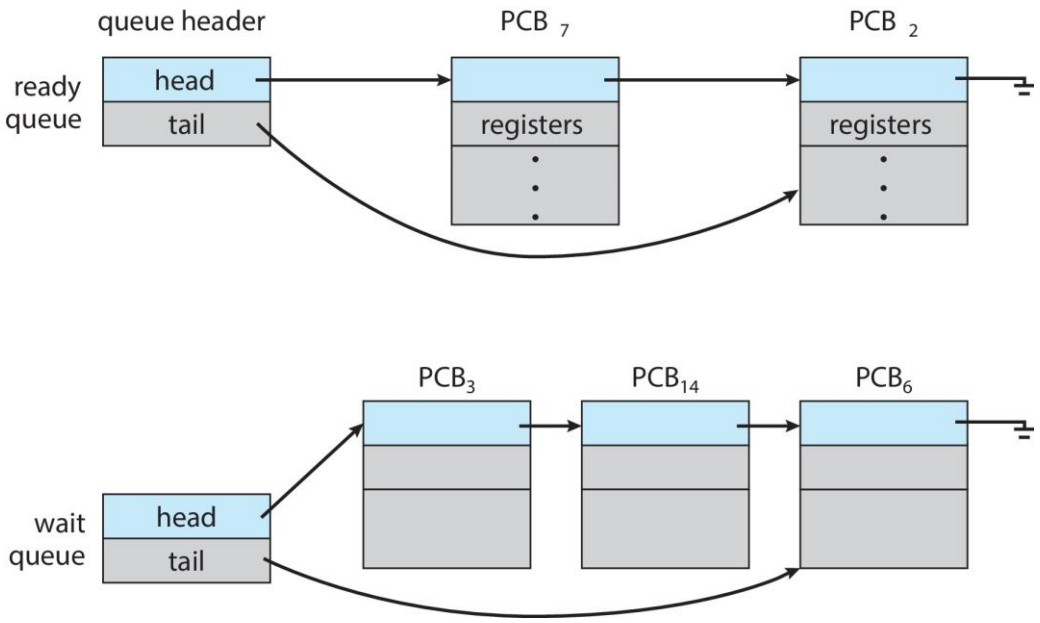
Python Process Representation

```
class TaskStruct:
    def __init__(self, t_pid, state, time_slice, parent=None):
        self.t_pid = t_pid                  # Process identifier
        self.state = state                  # State of the process
        self.time_slice = time_slice        # Scheduling information
        self.parent = parent                # Pointer to the parent process
        self.children = []                  # List of children processes
        self.files = []                     # List of open files (can be a list of file descriptors)
        self.memory_space = None            # Memory management information (address space)

    def add_child(self, child_process):
        """Adds a child process to the children list."""
        self.children.append(child_process)
        child_process.parent = self
```

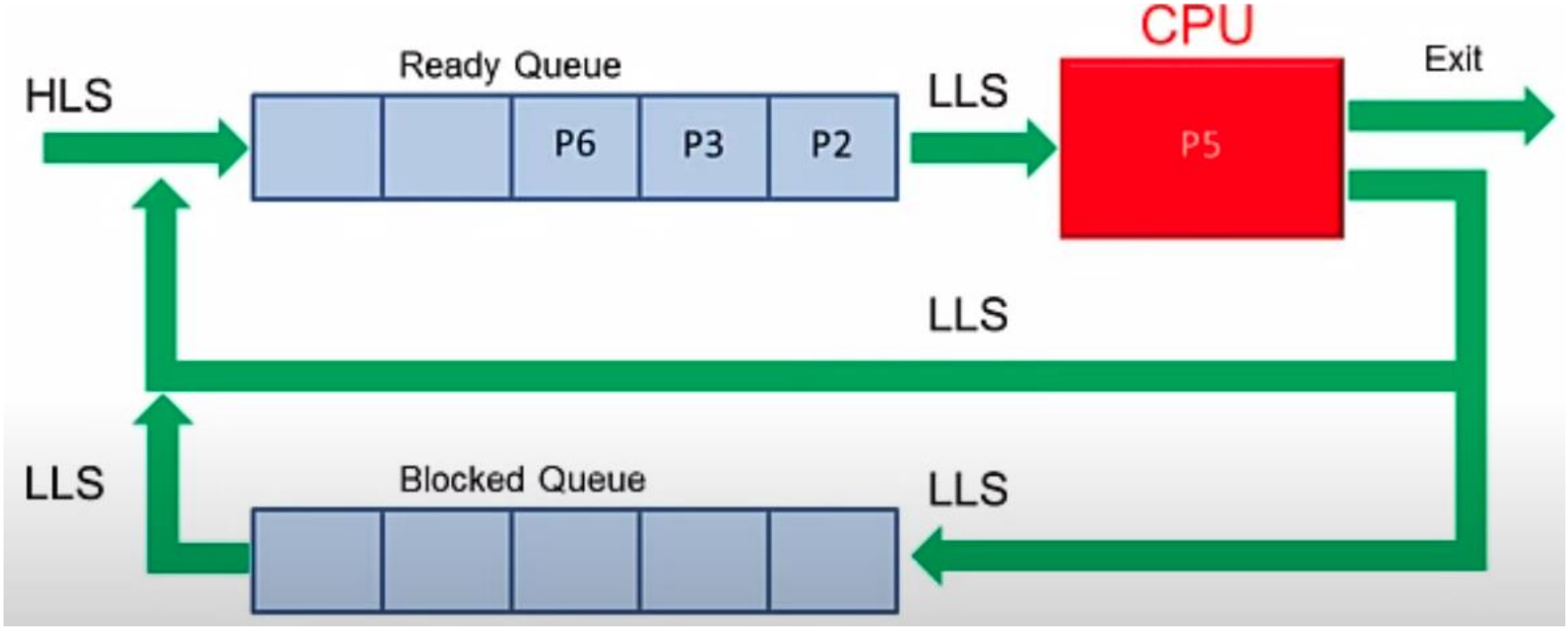
Process Scheduling

- **Process scheduler** selects among available processes for next execution on CPU core
- Goal -- Maximize CPU use, quickly switch processes onto CPU core
- Maintains **scheduling queues** of processes
  - **Ready queue** – set of all processes residing in main memory, ready and waiting to execute
  - **Wait queues** – set of processes waiting for an event (i.e., I/O)
  - Processes migrate among the various queues

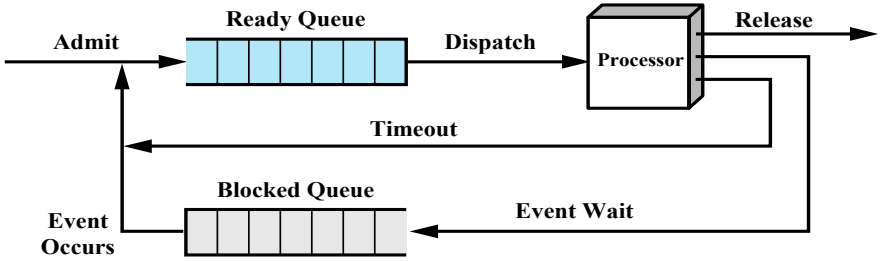
Ready and Wait Queues



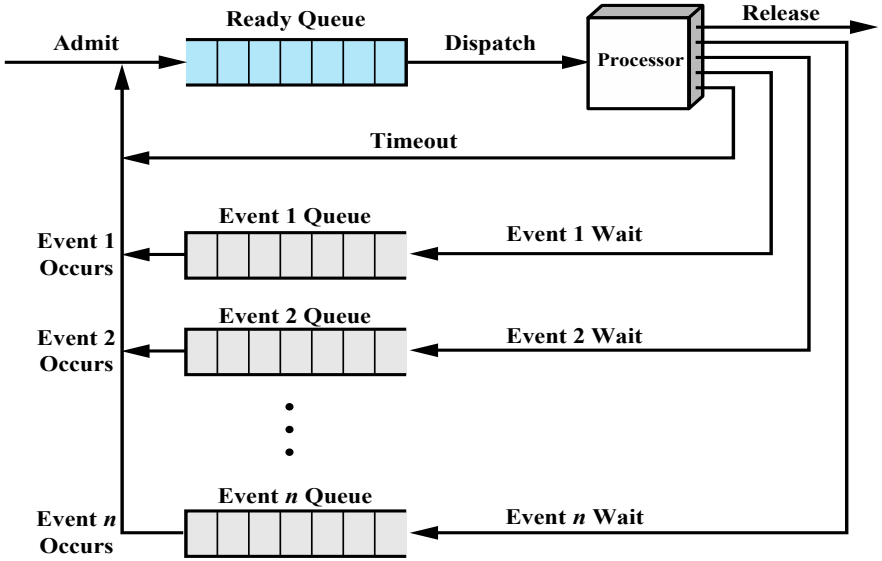
Ready and Wait Queues



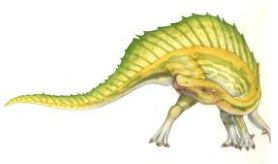
Ready and Wait Queues



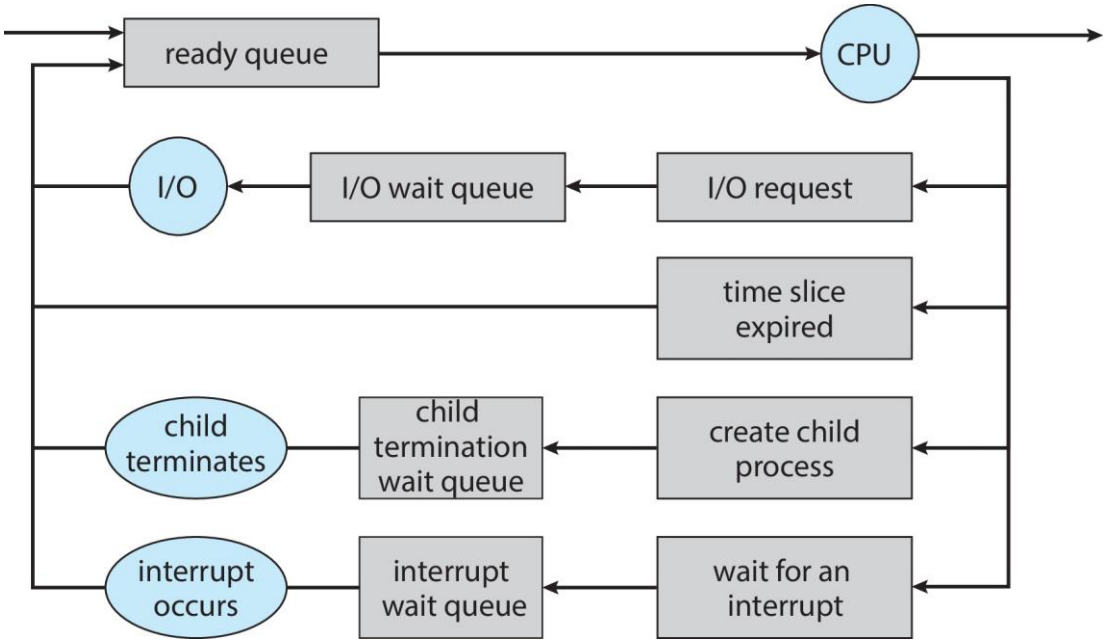
(a) Single blocked queue



(b) Multiple blocked queues

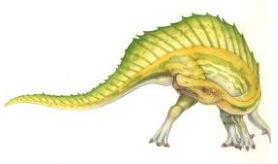


Representation of Process Scheduling



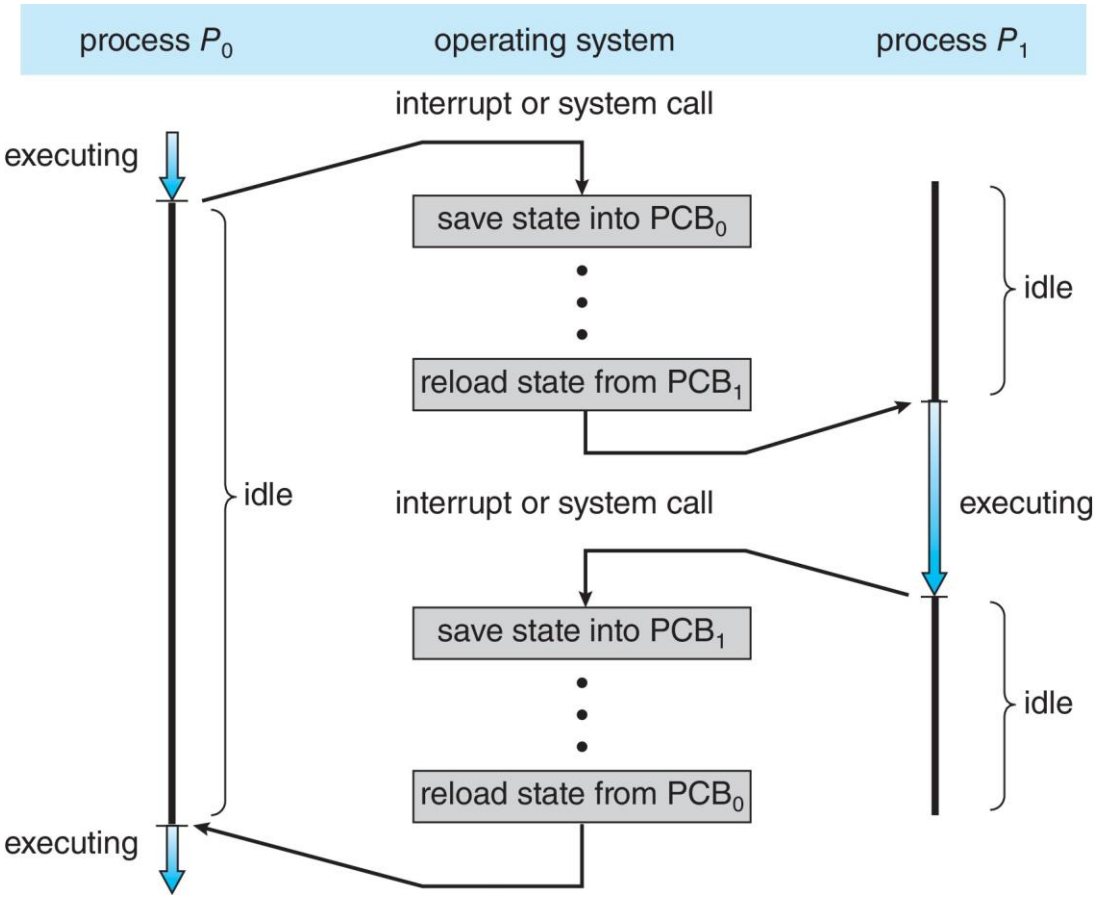
User => Kernel

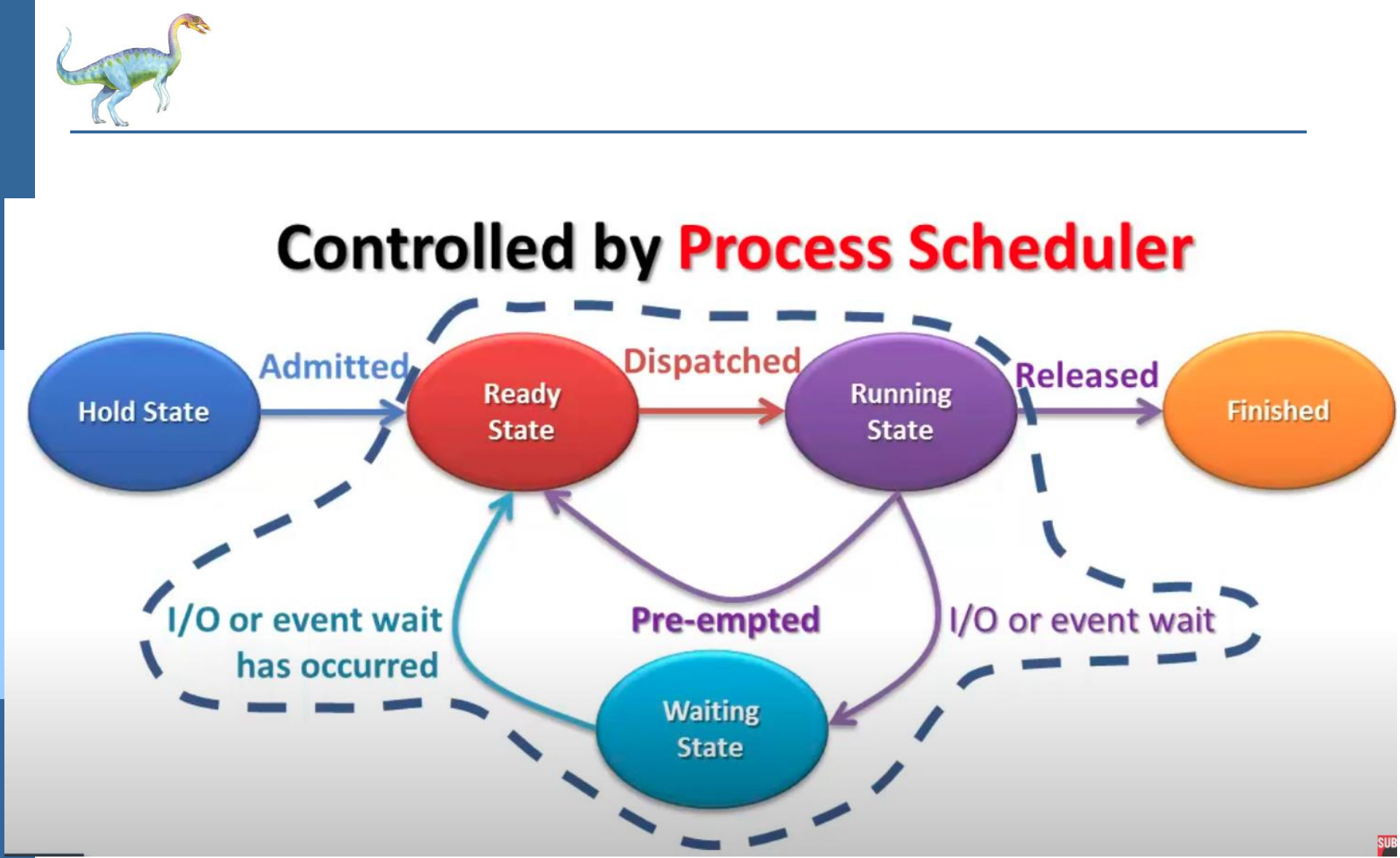
- System Call (eg. System service)
- Interrupt (eg. I/O service)
- Trap or Exception (protection violation)



CPU Switch From Process to Process

A **context switch** occurs when the CPU switches from one process to another.

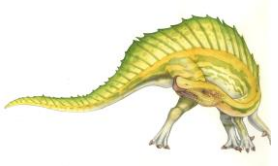



Controlled by Process Scheduler
Hold State
Admitted
Ready State
Dispatched
Running State
Released
Finished
I/O or event wait has occurred
Pre-empted
I/O or event wait
Waiting State

Context Switch

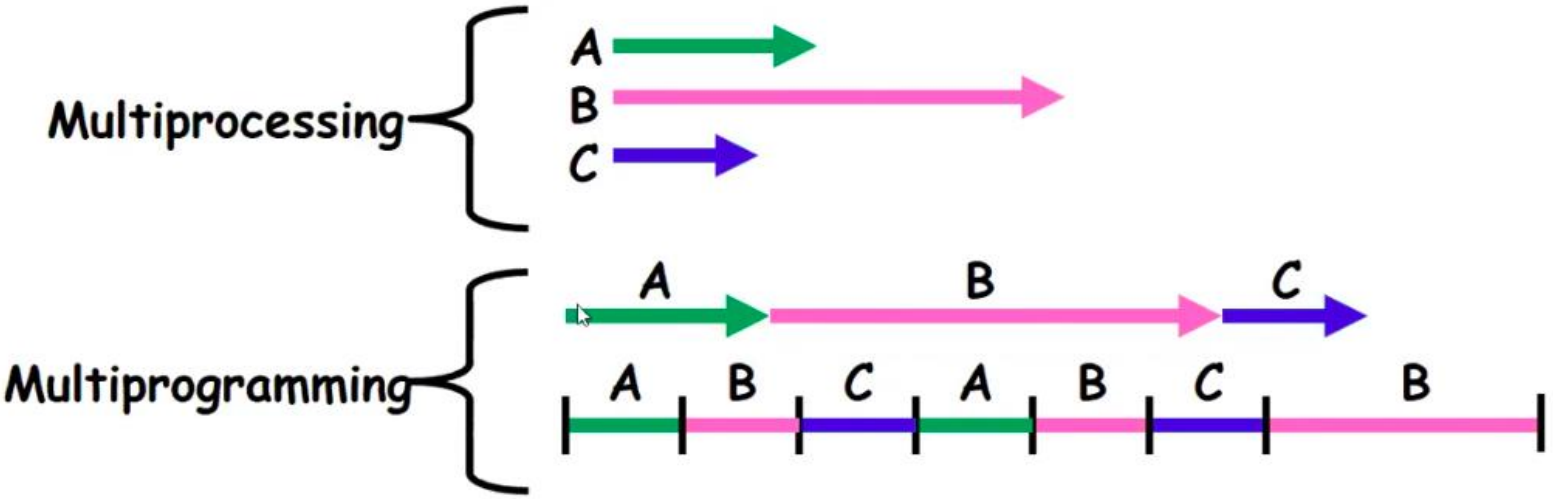
- When CPU switches to another process
  - The system must **save the state** of the old process
  - Load the **saved state** for the new process via a **context switch**
- **Context** of a process represented in the PCB
- Context-switch time is pure overhead; the system does no useful work while switching (**context switch time**)
  - The more complex the OS and the PCB ➔ the longer the context switch
- Time dependent on hardware support
  - Some hardware provides multiple sets of registers per CPU ➔ multiple contexts loaded at once

e.g. the more notes and tools when switch between assignments (more complex processes), the longer it takes to switch between them (longer context-switch time).



Multiprocessing vs. Multiprogramming

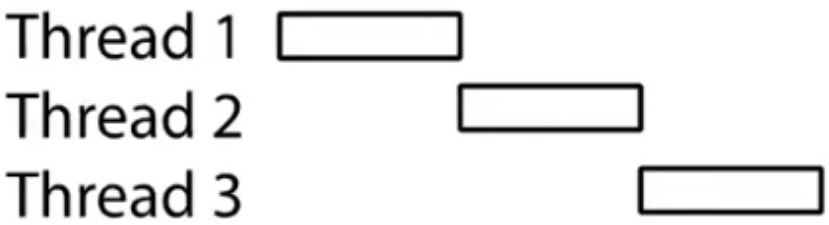
- Some Definitions:
  - Multiprocessing: Multiple CPUs(cores)
  - Multiprogramming: Multiple jobs/processes
  - Multithreading: Multiple threads/processes
- What does it mean to run two threads concurrently?
  - Scheduler is free to run threads in any order and interleaving
  - Thread may run to completion or time-slice in big chunks or small chunks



Programmer vs. Processor View

Programmer's View	Possible Execution #1	Possible Execution #2	Possible Execution #3
.	.	.	.
.	.	.	.
.	.	.	.
x = x + 1;	x = x + 1;	x = x + 1	x = x + 1
y = y + x;	y = y + x;	.....	y = y + x
z = x +5y;	z = x + 5y;	thread is suspended	.....
.	.	other thread(s) run	thread is suspended
.	.	thread is resumed	other thread(s) run
.	.	.....	thread is resumed
		y = y + x	.....
		z = x + 5y	z = x + 5y

Possible Executions



a) One execution

b) Another execution

c) Another execution

Adapted from: Prof. John Kubiatowicz
http://cs162.eecs.Berkeley.edu

Threading in Python

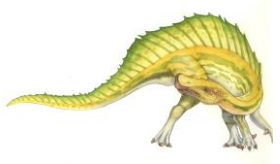
```
import threading
import time

def print_numbers():
    for i in range(1, 6):
        print(f"Number: {i}")
        time.sleep(1)  # Simulate a time-consuming task

def print_letters():
    for letter in ['A', 'B', 'C', 'D', 'E']:
        print(f"Letter: {letter}")
        time.sleep(1.5)  # Simulate a time-consuming task

# Create two threads
thread1 = threading.Thread(target=print_numbers)
thread2 = threading.Thread(target=print_letters)

# Start the threads
thread1.start()
thread2.start()

# Wait for both threads to complete
thread1.join()
thread2.join()

print("Both threads have finished execution.")
```

```
Number: 1Letter: A

Number: 2
Letter: B
Number: 3
Letter: C
Number: 4
Number: 5
Letter: D
Letter: E
Both threads have finished execution.
```

Multitasking in Mobile Systems

- Some mobile systems (e.g., early version of iOS) allow only one process to run, others suspended
- Due to screen real estate, user interface limits iOS provides for a
  - Single **foreground** process- controlled via user interface
  - Multiple **background** processes– in memory, running, but not on the display, and with limits
  - Limits include single, short task, receiving notification of events, specific long-running tasks like audio playback
- Android runs foreground and background, with fewer limits
  - Background process uses a **service** to perform tasks
  - Service can keep running even if background process is suspended
  - Service has no user interface, small memory use

iPhone 1 - Steve Jobs MacWorld keynote in 2007 - Full Presentation, 80 mins (youtube.com)



Operations on Processes

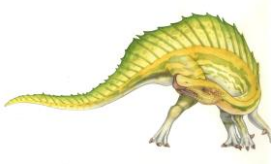
- System must provide mechanisms for:
  - Process creation
  - Process termination

Bootstrapping

- If processes are created by other processes, how does the first process start?
- First process is started by the kernel
  - Often configured as an argument to the kernel *before* the kernel boots
  - Often called the “init” process
- After this, all processes on the system are created by other processes

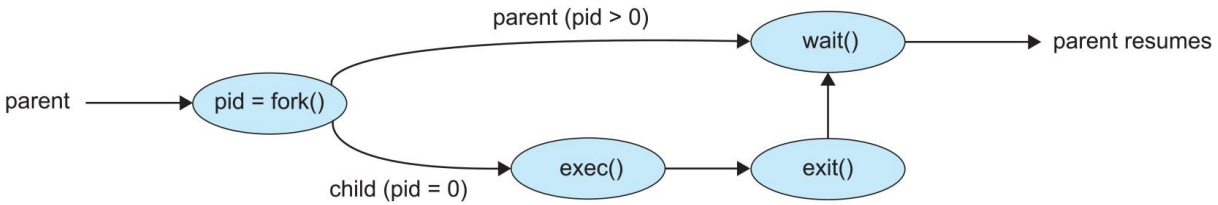
Process Creation

- **Parent** process create **children** processes, which, in turn create other processes, forming a **tree** of processes
- Generally, process identified and managed via a **process identifier** (**pid**)
- 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

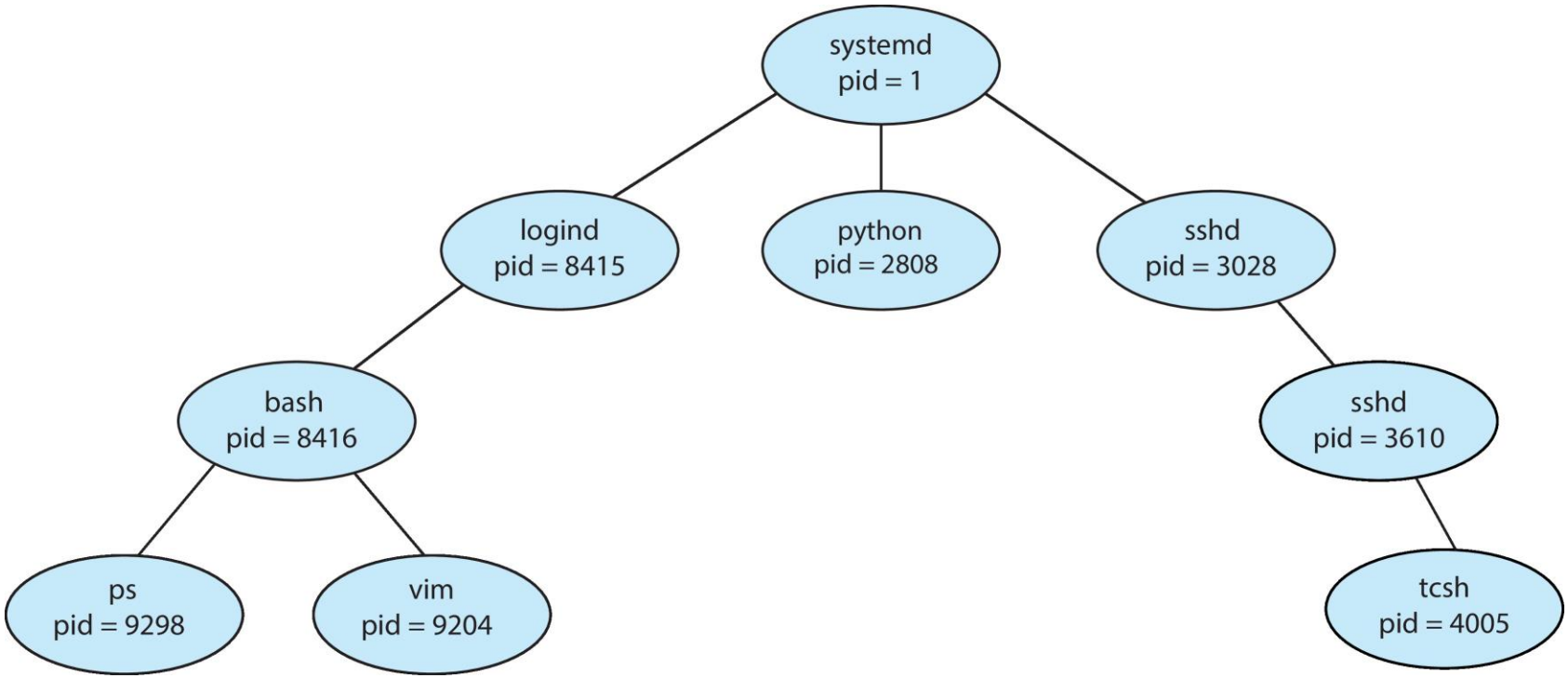


Process Creation (Cont.)

- 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
  - Parent process calls **`wait()`** waiting for the child to terminate



A Tree of Processes in Linux



Python Program Forking Separate Process

```
import multiprocessing
import os
max_level = 3  # Set the maximum depth of the process tree

def build_process_tree(tree, level, parent_pid):
    child_pid = os.getpid()
    tree.append((level, parent_pid, child_pid))

    if level < max_level:  # Continue to fork if not at max level
        # Create first child
        first_child = multiprocessing.Process(target=build_process_tree,
                                              args=(tree, level + 1, child_pid))
        first_child.start()
        first_child.join()

        # Create second child
        second_child = multiprocessing.Process(target=build_process_tree,
                                               args=(tree, level + 1, child_pid))
        second_child.start()
        second_child.join()

def print_tree(tree):
    for level, parent_pid, child_pid in tree:
        indent = "    " * level
        print(f"{indent}Parent PID: {parent_pid}, Child PID: {child_pid}")
```

Python Program Forking Separate Process

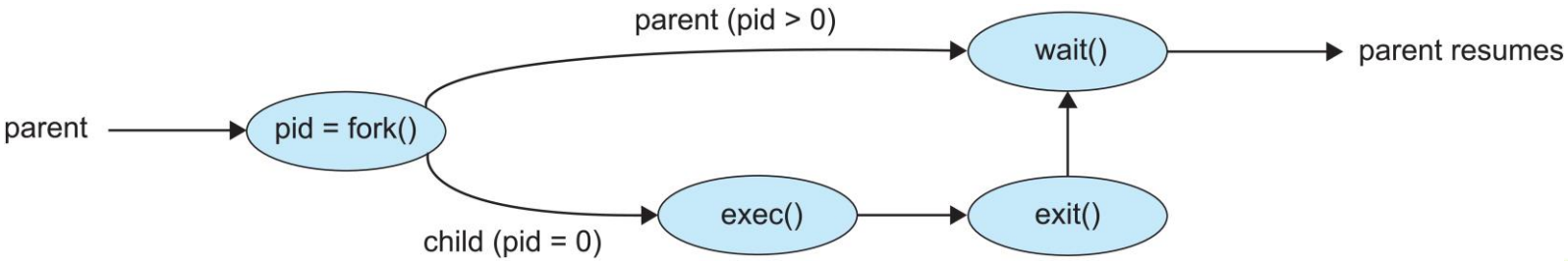
```
if __name__ == "__main__":

    manager = multiprocessing.Manager()
    tree = manager.list()  # A managed list to hold the tree structure

    root_pid = os.getpid()
    print(f"Root Parent PID: {root_pid}")

    # Build the process tree
    build_process_tree(tree, 0, root_pid)

    # Print the process tree
    print tree(tree)
```

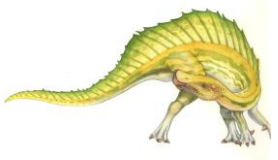
Python Program Forking Separate Process

```
Root Parent PID: 19080
Parent PID: 19080, Child PID: 19080
    Parent PID: 19080, Child PID: 23848
        Parent PID: 23848, Child PID: 18236
            Parent PID: 18236, Child PID: 1300
            Parent PID: 18236, Child PID: 15948
        Parent PID: 23848, Child PID: 16280
            Parent PID: 16280, Child PID: 23984
            Parent PID: 16280, Child PID: 26548
    Parent PID: 19080, Child PID: 24500
        Parent PID: 24500, Child PID: 17932
            Parent PID: 17932, Child PID: 26600
            Parent PID: 17932, Child PID: 13560
        Parent PID: 24500, Child PID: 15936
            Parent PID: 15936, Child PID: 25336
            Parent PID: 15936, Child PID: 6236
```

Process Termination

- Process executes last statement and then asks the operating system to delete it using the **`exit()`** system call.
  - Returns status data from child to parent (via **`wait()`**)
  - Process' resources are deallocated by operating system
- Parent may terminate the execution of children processes using the **`abort()`** system call. Some reasons for doing so:
  - Child has **exceeded** allocated resources
  - Task assigned to child is **no longer required**
  - The **parent is exiting**, and the operating systems does not allow a child to continue if its parent terminates





Process Termination

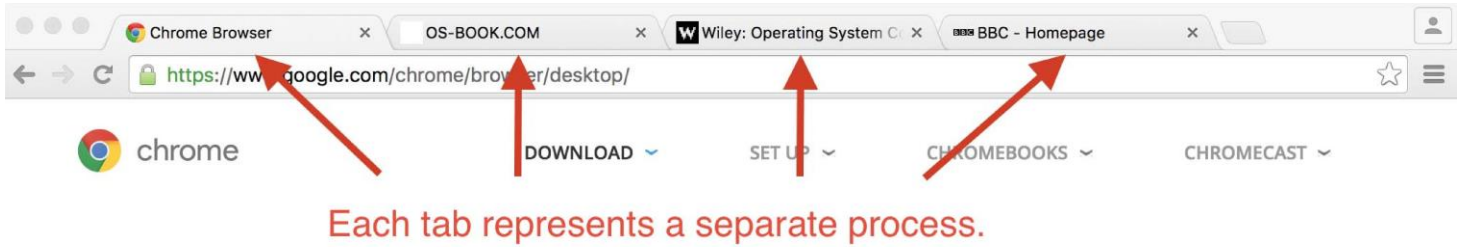
- Some operating systems do not allow child to exists if its parent has terminated. If a process terminates, then all its children must also be terminated.
  - **cascading termination.** All children, grandchildren, etc., are terminated.
  - The termination is initiated by the operating system.
- The parent process may wait for termination of a child process by using the **`wait()`** system call. The call returns status information and the pid of the terminated process

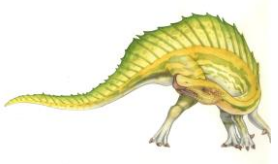
```
pid = wait(&status);
```

- If no parent waiting (did not invoke **`wait()`**) process is a **zombie:** A **process** still exists in the system but isn't doing any work (it's just waiting to be cleaned up).
- If parent terminated without invoking **`wait()`**, process is an **orphan:** A **process** continues to do its job even though its original parent is no longer around.

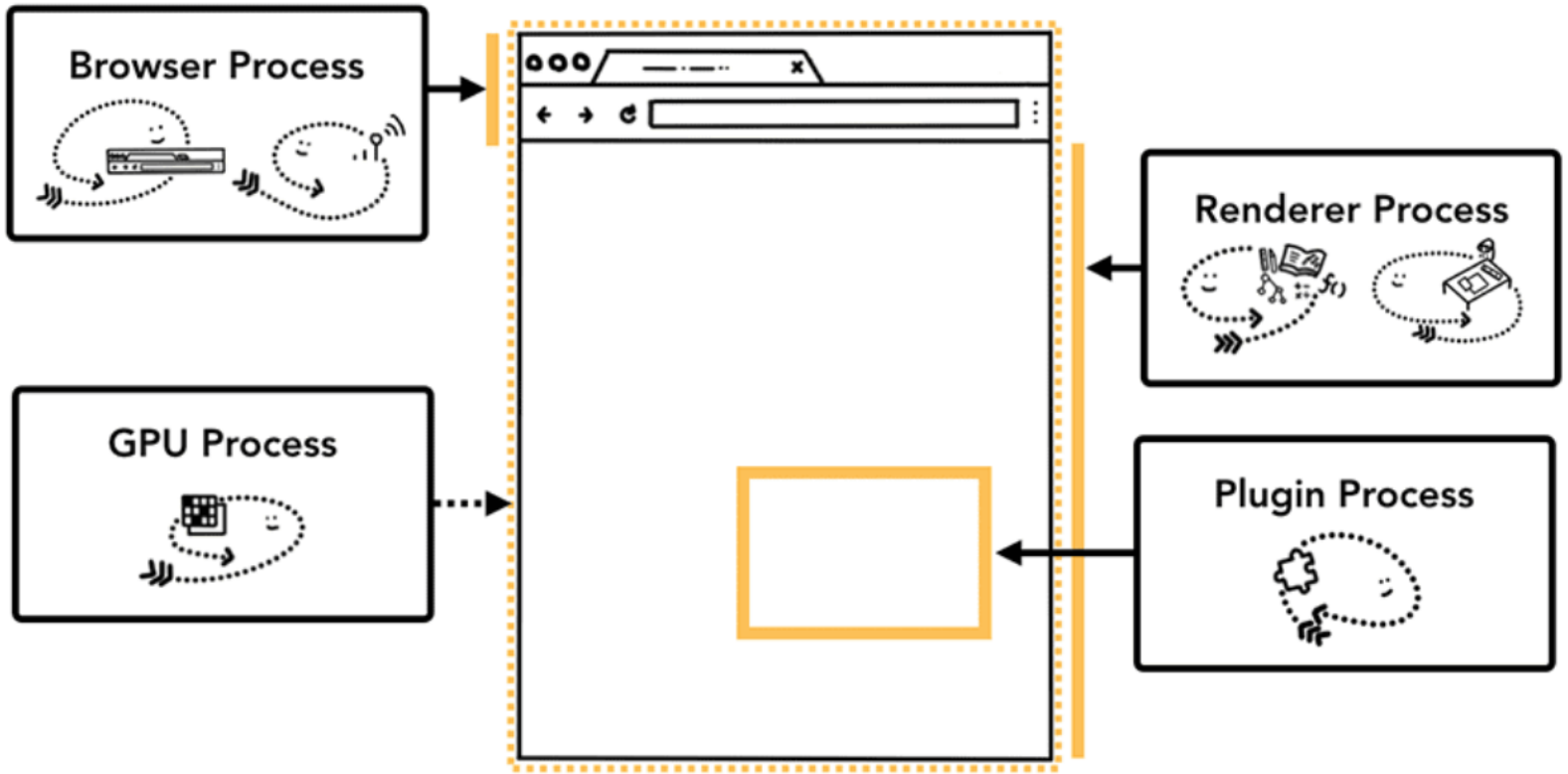
Multiprocess Architecture – Chrome Browser

- Many web browsers ran as single process (some still do)
  - If one web site causes trouble, entire browser can hang or crash
- Google Chrome Browser is multiprocess with 3 different types of processes:
  - **Browser process manages** user interface, disk and network I/O
  - **Renderer process renders** web pages, deals with HTML, Javascript. A new renderer created for each website opened
    - Runs in **sandbox** restricting disk and network I/O, minimizing effect of security exploits
  - **Plug-in** process for each type of plug-in, e.g. a video player or PDF viewer





Process and What it controls	
Browser	Controls "chrome" part of the application including address bar, bookmarks, back and forward buttons. Also handles the invisible, privileged parts of a web browser such as network requests and file access.
Renderer	Controls anything inside of the tab where a website is displayed.
Plugin	Controls any plugins used by the website, for example, flash.
GPU	Handles GPU tasks in isolation from other processes. It is separated into different process because GPUs handles requests from multiple apps and draw them in the same surface.

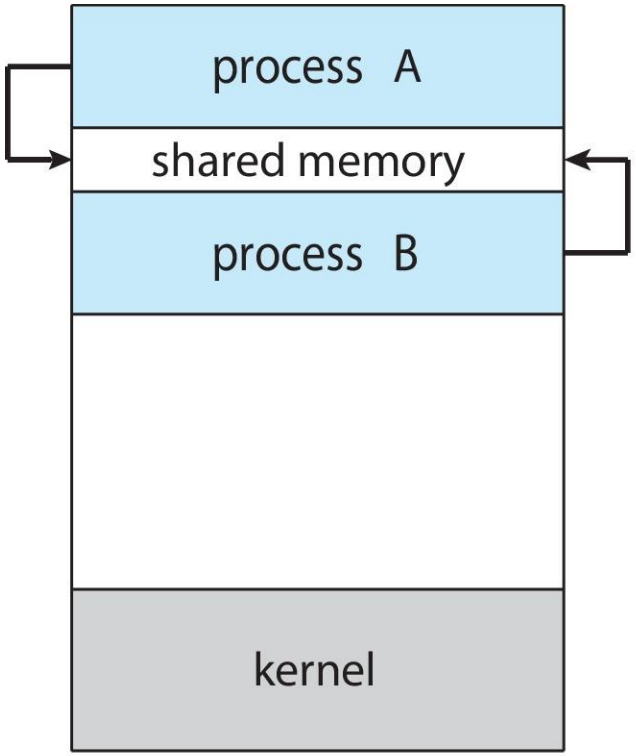


Interprocess Communication (IPC)

- Processes within a system may be ***independent*** or ***cooperating***
- Cooperating process can affect or be affected by other processes, including **sharing data**
- Reasons for cooperating processes:
  - Information sharing
  - Computation speedup
  - Modularity
  - Convenience
- Cooperating processes need **interprocess communication** (**IPC**)
- Two models of IPC
  - **Shared memory**
  - **Message passing**

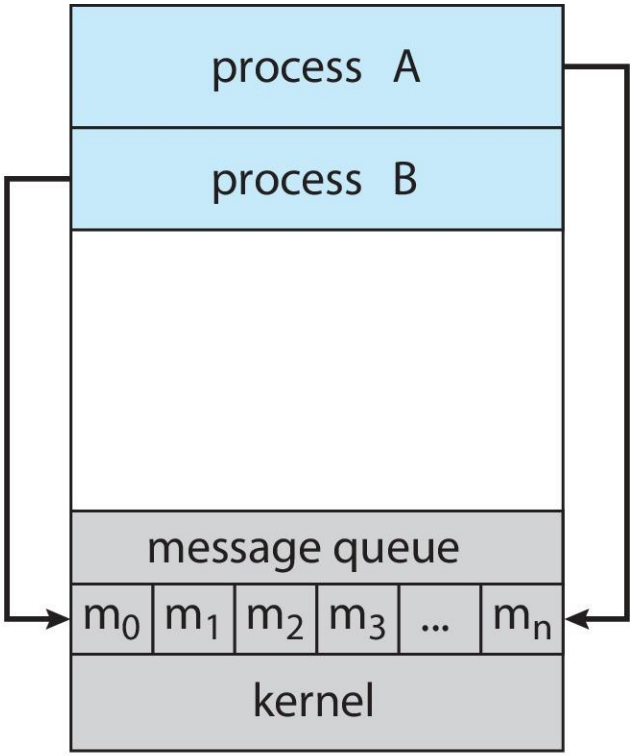
Communications Models

(a) Shared memory.



(a)

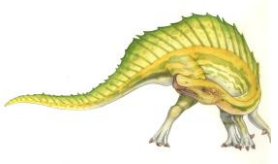
(b) Message passing.



(b)

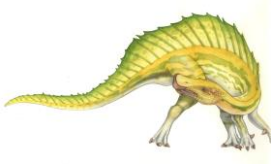
Producer-Consumer Problem

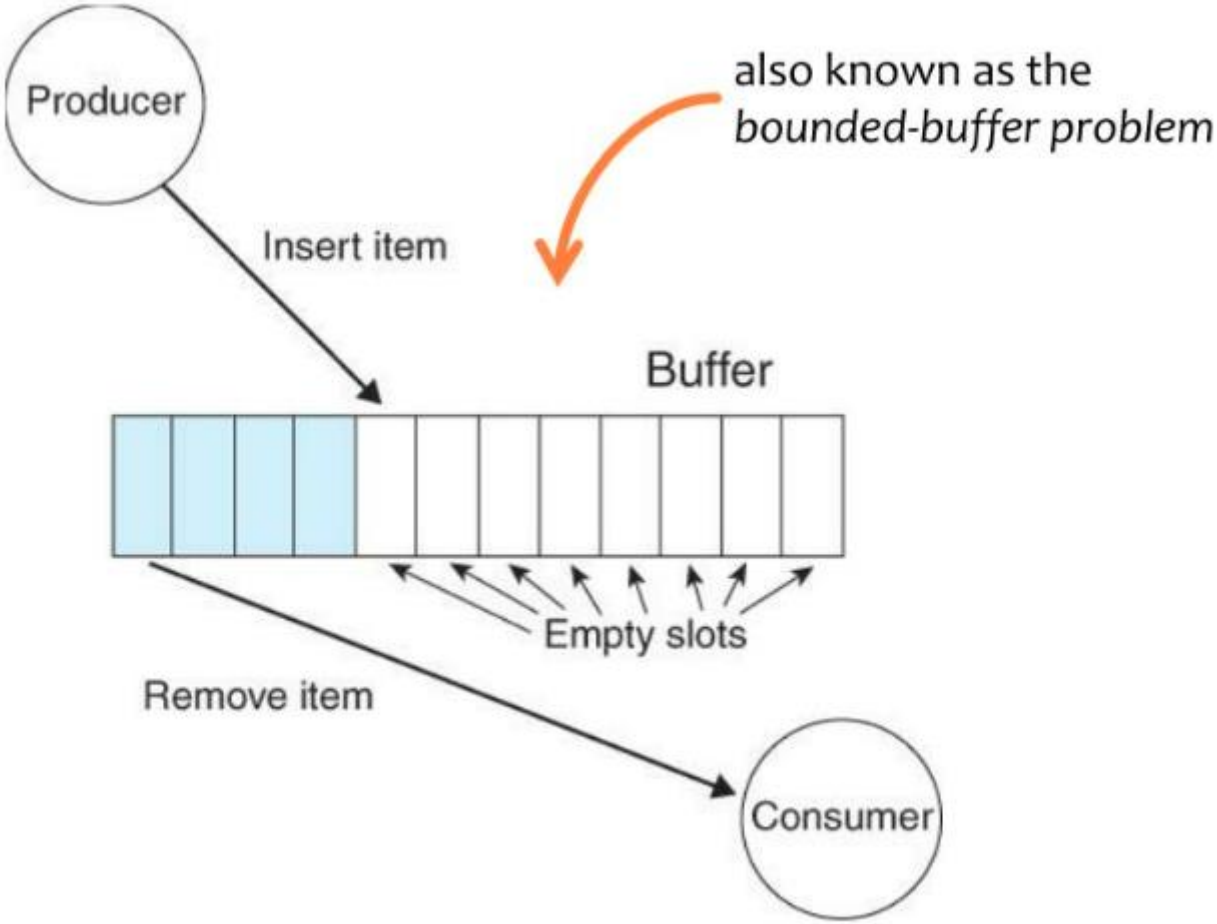
- Paradigm for cooperating processes:
  - *producer* process produces information that is consumed by a *consumer* process
- Two variations:
  - **unbounded-buffer** places no practical limit on the size of the buffer:
    - Producer never waits
    - Consumer waits if there is no buffer to consume
  - **bounded-buffer** assumes that there is a fixed buffer size
    - Producer must wait if all buffers are full
    - Consumer waits if there is no buffer to consume



IPC – Shared Memory

- An area of memory shared among the processes that wish to communicate
- The communication is under the control of the users processes not the operating system.
- Major issues is to provide mechanism that will allow the user processes to synchronize their actions when they access shared memory.
- Synchronization is discussed in great details in Chapters 6 & 7.




Producer
also known as the
bounded-buffer problem
Insert item
Buffer
Empty slots
Remove item
Consumer

Bounded-Buffer – Shared-Memory Solution

- Shared data

```
#define BUFFER_SIZE 10
typedef struct {
  . . .
} item;

item buffer[BUFFER_SIZE];
int in = 0;
int out = 0;
```

- Solution is correct, but can only use **`BUFFER_SIZE-1`** elements

Producer Process – Shared Memory

```
item next_produced;

while (true) {
   /* produce an item in next produced */
   while (((in + 1) % BUFFER_SIZE) == out)
        ; /* do nothing */
   buffer[in] = next_produced;
   in = (in + 1) % BUFFER_SIZE;
}
```

Consumer Process – Shared Memory

```
item next_consumed;

while (true) {
        while (in == out)
                ; /* do nothing */
        next_consumed = buffer[out];
        out = (out + 1) % BUFFER_SIZE;

        /* consume the item in next consumed */
}
```

What about Filling all the Buffers?

- Suppose that we wanted to provide a solution to the consumer-producer problem that fills **all** the buffers.
- We can do so by having an integer **`counter`** that keeps track of the number of full buffers.
- Initially, **`counter`** is set to 0.
- The integer **`counter`** is incremented by the producer after it produces a new buffer.
- The integer **`counter`** is decremented by the consumer after it consumes a buffer.

Producer

```
while (true) {
        /* produce an item in next produced */

        while (counter == BUFFER_SIZE)
                ; /* do nothing */
        buffer[in] = next_produced;
        in = (in + 1) % BUFFER_SIZE;
        counter++;
}
```

Consumer

```
while (true) {
        while (counter == 0)
                ; /* do nothing */
        next_consumed = buffer[out];
        out = (out + 1) % BUFFER_SIZE;
         counter--;
        /* consume the item in next consumed */
}
```

Race Condition

- **`counter++`** could be implemented as

```
register1 = counter
register1 = register1 + 1
counter = register1
```

- **`counter--`** could be implemented as

```
register2 = counter
register2 = register2 - 1
counter = register2
```

- Consider this execution interleaving with "count = 5" initially:

S0: producer execute `register1 = counter` {register1 = 5}
S1: producer execute `register1 = register1 + 1` {register1 = 6}
S2: consumer execute `register2 = counter` {register2 = 5}
S3: consumer execute `register2 = register2 - 1` {register2 = 4}
S4: producer execute `counter = register1` {counter = 6 }
S5: consumer execute `counter = register2` {counter = 4}

Race Condition (Cont.)

- Question – why was there no race condition in the first solution (where at most N – 1) buffers can be filled?
- More in Chapter 6.

Race Conditions

- Initially `x == 0` and `y == 0`

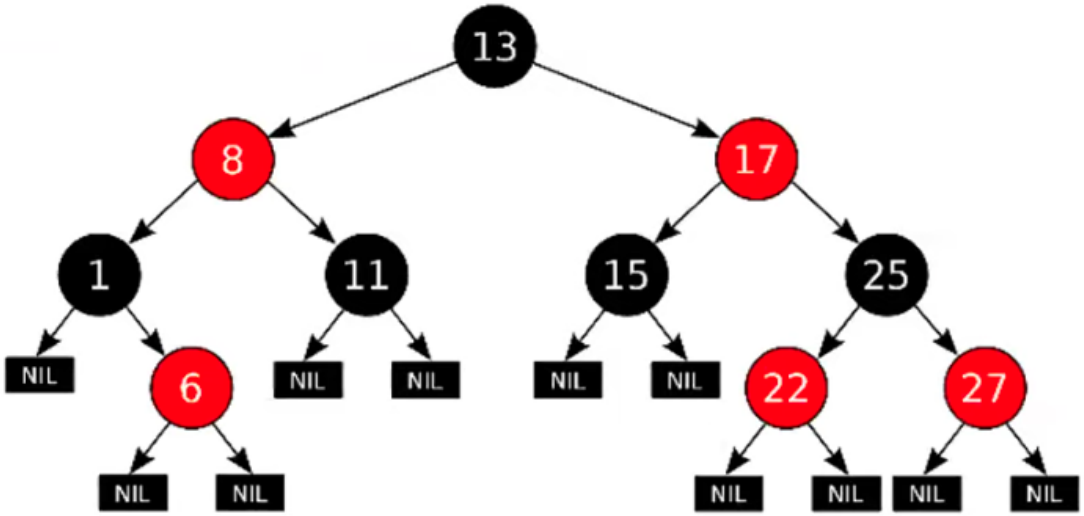
Thread A	Thread B
`x = y + 1;`	`y = 2;`
	`y = y * 2;`

- What are the possible values of x below?
- 1 or 3 or 5 (non-deterministically)
- Race Condition: Thread A races against Thread B!

Example: Shared Data Structure

Thread A

`Insert(3)`

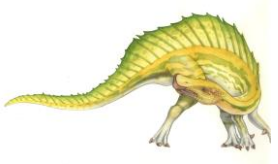


Thread B

`Insert(4)`

`Get(6)`

Tree-Based Set Data Structure



IPC – Message Passing

- Processes communicate with each other without resorting to shared variables
- IPC facility provides two operations:
  - **`send`**(*message*)
  - **`receive`**(*message*)
- The *message* size is either fixed or variable

Message Passing (Cont.)

- If processes *P* and *Q* wish to communicate, they need to:
  - Establish a ***communication link*** between them
  - Exchange messages via **send/receive**
- Implementation issues:
  - How are links established?
  - Can a link be associated with more than two processes?
  - How many links can there be between every pair of communicating processes?
  - What is the capacity of a link?
  - Is the size of a message that the link can accommodate fixed or variable?
  - Is a link unidirectional or bi-directional?

Implementation of Communication Link

- Physical:
  - Shared memory
  - Hardware bus
  - Network
- Logical:
  - Direct or indirect
  - Synchronous or asynchronous
  - Automatic or explicit buffering

Direct Communication

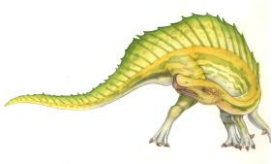
- Processes must name each other explicitly:
  - **`send`** (*P, message*) – send a message to process P
  - **`receive`**(*Q, message*) – receive a message from process Q
- Properties of communication link
  - Links are established automatically
  - A link is associated with exactly one pair of communicating processes
  - Between each pair there exists exactly one link
  - The link may be unidirectional, but is usually bi-directional

Indirect Communication

- Messages are directed and received from mailboxes (also referred to as ports)
  - Each mailbox has a unique id
  - Processes can communicate only if they share a mailbox
- Properties of communication link
  - Link established only if processes share a common mailbox
  - A link may be associated with many processes
  - Each pair of processes may share several communication links
  - Link may be unidirectional or bi-directional

Indirect Communication (Cont.)

- Operations
  - Create a new mailbox (port)
  - Send and receive messages through mailbox
  - Delete a mailbox
- Primitives are defined as:
  - **`send`**(*A, message*) – send a message to mailbox A
  - **`receive`**(*A, message*) – receive a message from mailbox A



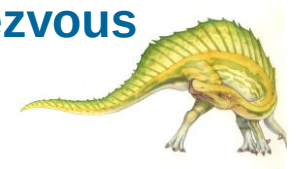
Indirect Communication (Cont.)

- Mailbox sharing
  - P_1, P_2, and P_3 share mailbox A
  - P_1, sends; P_2 and P_3 receive
  - Who gets the message?
- Solutions
  - Allow a link to be associated with at most two processes
  - Allow only one process at a time to execute a receive operation
  - Allow the system to select arbitrarily the receiver. Sender is notified who the receiver was.

Synchronization

Message passing may be either blocking or non-blocking

- **Blocking** is considered **synchronous**
  - **Blocking send** -- the sender is blocked until the message is received
  - **Blocking receive** -- the receiver is blocked until a message is available
- **Non-blocking** is considered **asynchronous**
  - **Non-blocking send** -- the sender sends the message and continue
  - **Non-blocking receive** -- the receiver receives:
    - A valid message, or
    - Null message
- Different combinations possible
  - If both send and receive are blocking, we have a **rendezvous**



Producer-Consumer: Message Passing

- Producer

```
message next_produced;
while (true) {
  /* produce an item in next_produced */

  send(next_produced);
}
```

- Consumer

```
message next_consumed;
while (true) {
  receive(next_consumed)

  /* consume the item in next_consumed */
}
```

Buffering

- Queue of messages attached to the link.
- Implemented in one of three ways
  1. Zero capacity – no messages are queued on a link.
     Sender must wait for receiver (rendezvous)
  2. Bounded capacity – finite length of *n* messages
     Sender must wait if link full
  3. Unbounded capacity – infinite length
     Sender never waits

Communication Mechanism Comparison

Mechanism	Description	Type of Communication	Platform
POSIX Message Queues	Allows asynchronous communication in UNIX systems, sending and receiving messages between processes.	Asynchronous, Message Passing	UNIX
Windows Named Pipes	Enables communication between processes on the same machine or over a network, can be unidirectional or bidirectional.	Synchronous or Asynchronous, IPC	Windows
Sockets (TCP/UDP)	Widely used for inter-process communication over networks, with TCP for reliable communication and UDP for faster, connectionless communication.	Synchronous/Asynchronous, Network Communication	Cross-platform (TCP/IP)
Mach	Microkernel-based system where processes communicate via message passing, used for inter-process and system service communication.	Asynchronous, IPC and System Services	Mach-based systems
Java Remote Method Invocation (RMI)	Allows Java objects running in different JVMs to invoke methods on each other, used for distributed object communication.	Synchronous, Remote Method Invocation	Java Virtual Machines (JVM)

End of Chapter 3