
Process Management

Process Abstraction

Lecture 2

Overview

- Introduction to Process Management
- Process Abstraction:
 - Memory Context
 - Code & Data
 - Function call
 - Dynamically allocated memory
 - Hardware Context
 - OS Context
 - Process State
 - Process Control Block and Process Table
- OS interaction with Process

Recap: Efficient Hardware Utilization

- OS should provide efficient use of the hardware resource:
 - By managing the programs executing on the hardware
- Observation:
 - If there is only **one program executing at any point in time**, how can we utilize hardware resources effectively?
- Solution:
 - Allow **multiple programs to share the hardware**
 - e.g., Multiprogramming, Time-sharing

Introduction to Process Management

- As the OS, to be able to switch from running program **A** to program **B** requires:
 1. Information regarding the execution of program **A** needs to be stored
 2. Program **A**'s information is replaced with the information required to run program **B**
- Hence, we need:
 - An **abstraction** to describe a running program
 - aka **process**

Key Topics

Process Abstraction

- Information describing an executing program

Process Scheduling

- Deciding which process get to execute

Inter-Process Communication & Synchronization

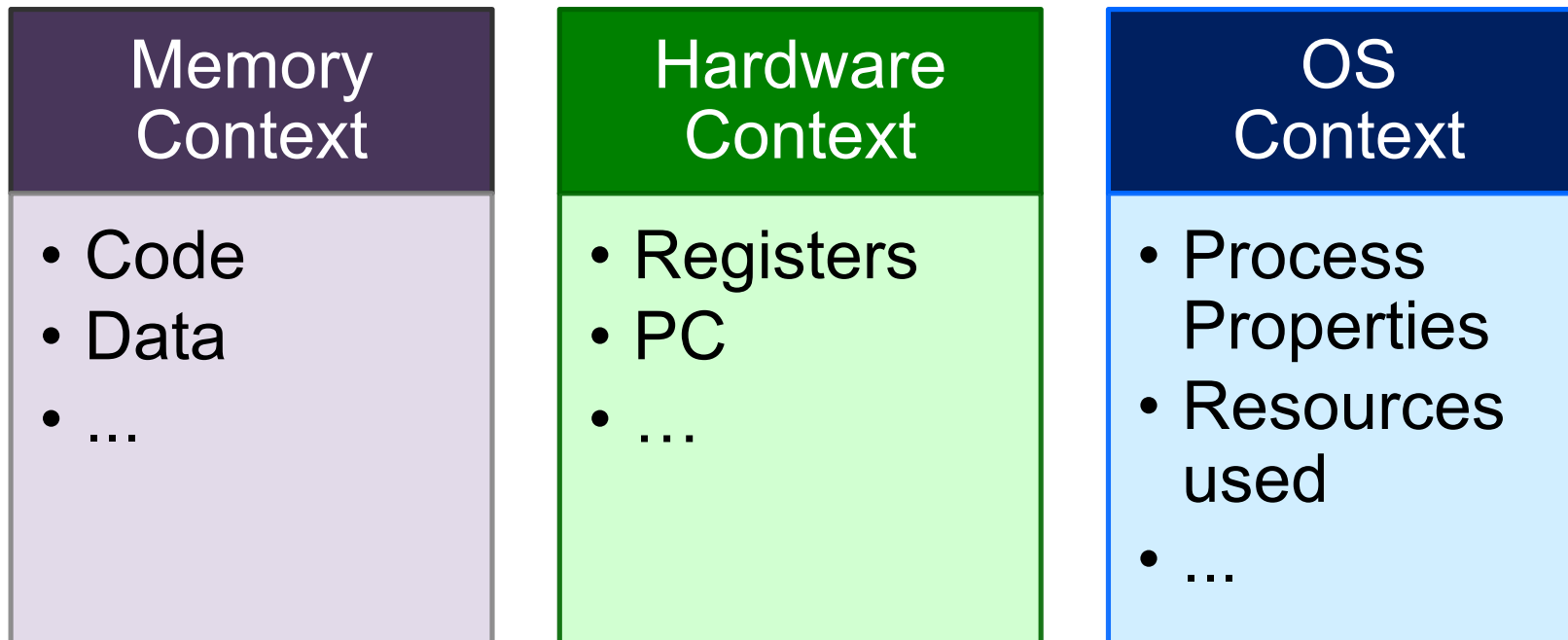
- Passing information between processes

Alternative to Process

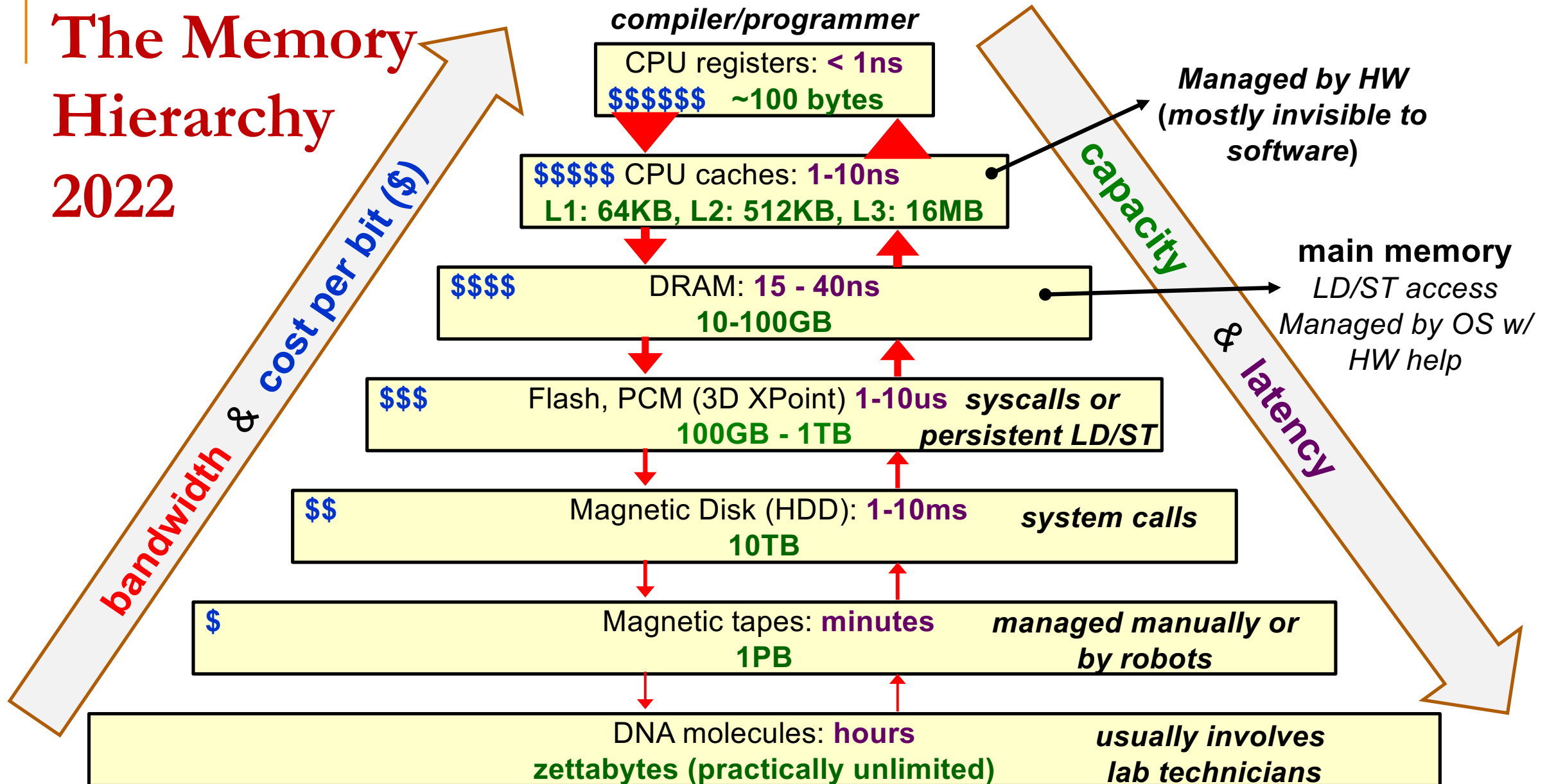
- Light-weight process aka Thread

Process Abstraction

- **(Process / Task / Job)** is a dynamic abstraction for executing program
 - The OS representation of a process includes **all information required** to describe a **running program**



The Memory Hierarchy 2022



Recap: C Sample Program and Assembly Code

```
int i = 0;
```

```
i = i + 20;
```

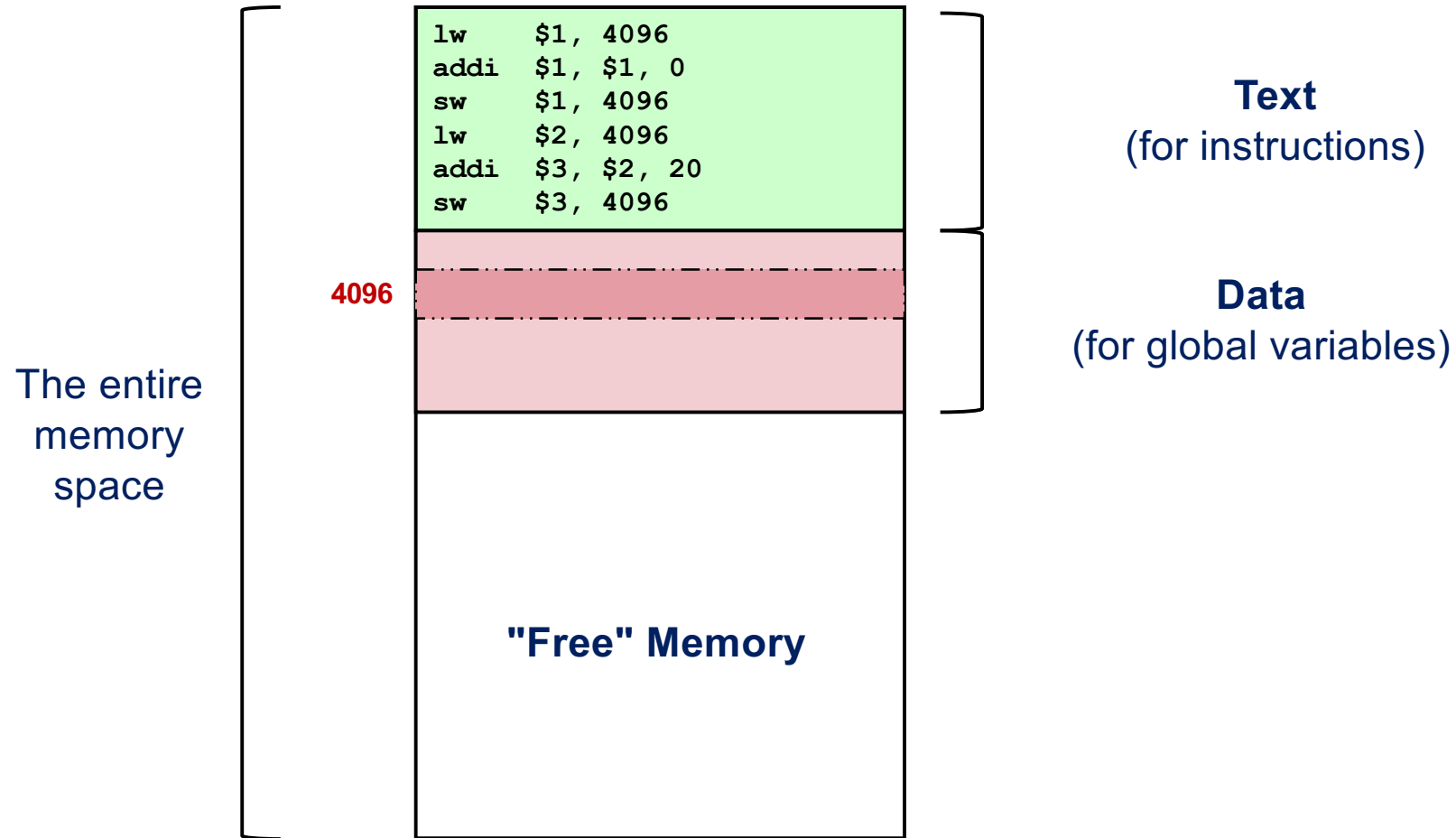
C Code Fragment

```
lw    $1, 4096    //Assume address of i = 4096
addi   $1, $0, 0   //register $1 = 0
sw     $1, 4096    //i = 0

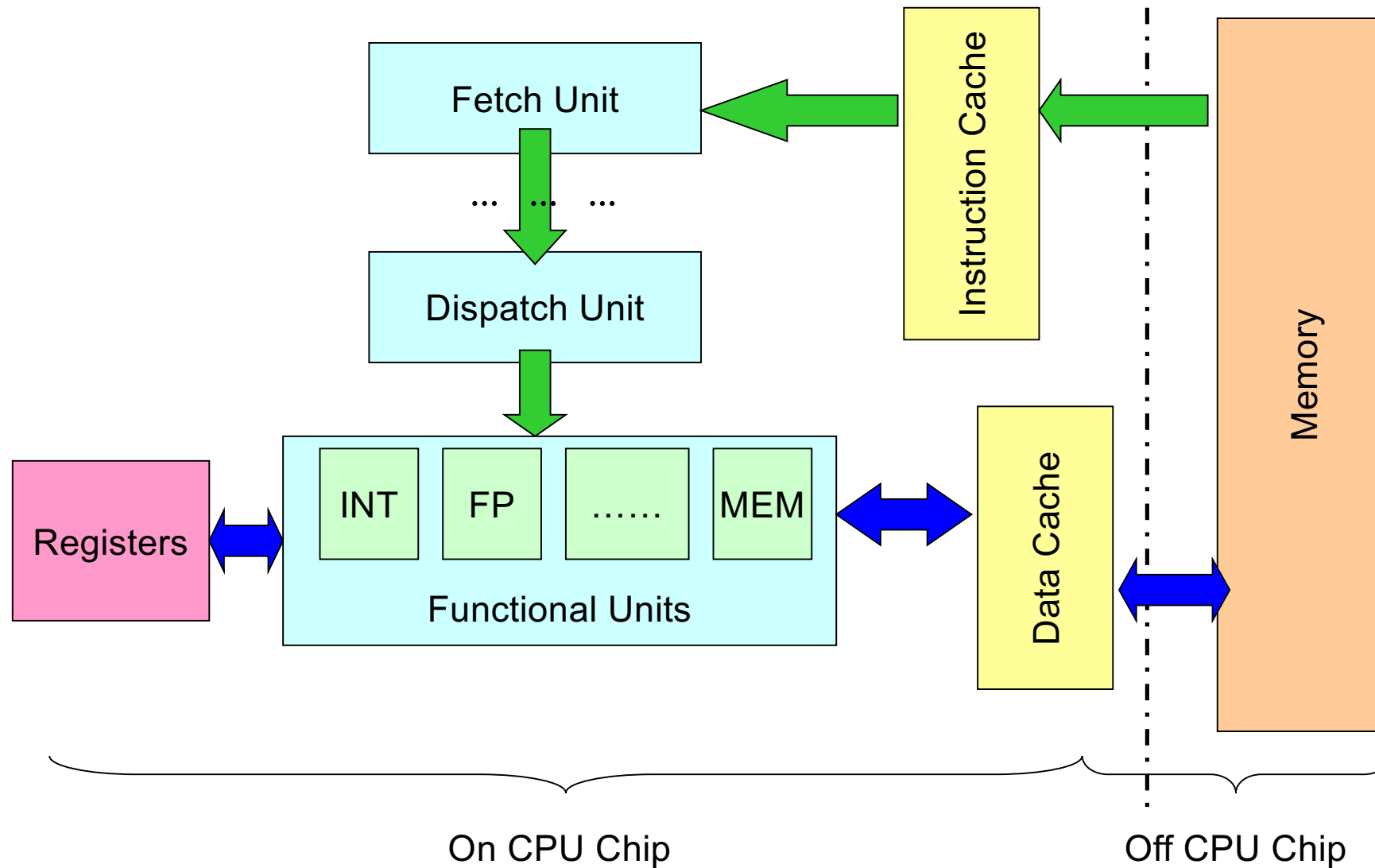
lw     $2, 4096    //read i
addi   $3, $2, 20   // $3 = $2 + 20
sw     $3, 4096    //i = i + 20
```

Corresponding MIPS-like Assembly Code

Recap: Program Execution (Memory)



Recap: Generic Computer Organization



Recap: Component Description

■ Memory:

- ❑ Storage for instruction and data

■ Cache:

- ❑ Duplicate part of the memory for faster access
- ❑ Usually split into instruction cache and data cache

■ Fetch unit:

- ❑ Loads instruction from memory
- ❑ Location indicated by a special register: **P**rogram **C**ounter (PC)

Recap: Component Description (cont)

■ Functional units:

- ❑ Carry out the instruction execution
- ❑ Dedicated to different instruction type

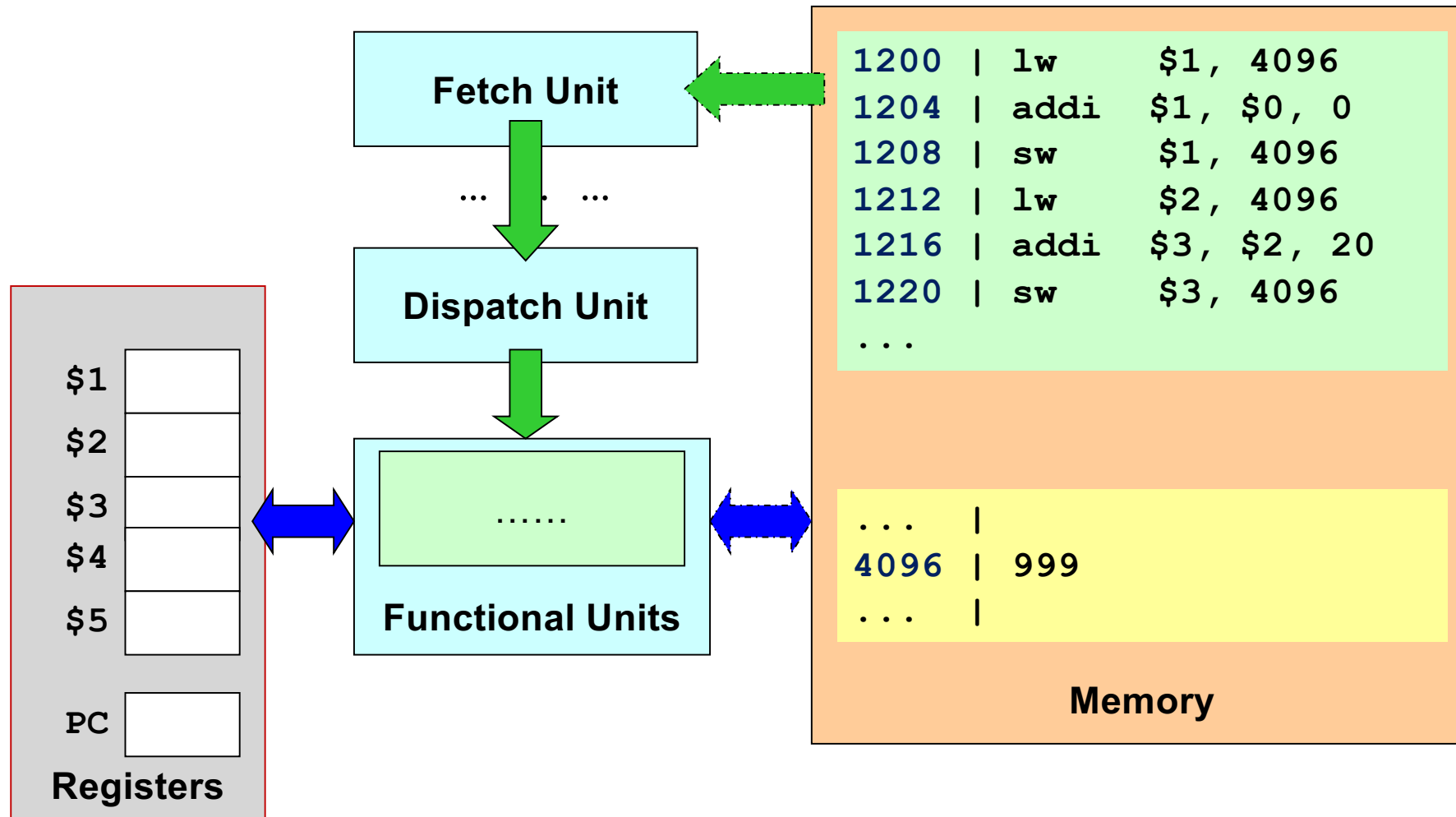
■ Registers:

- ❑ Internal storage for the fastest access speed
- ❑ General Purpose Register (GPR):
 - Accessible by user program (i.e., visible to compiler)
- ❑ Special Registers:
 - **Program Counter** (PC)
 - **Status Register** (a.k.a. **Program Status Word** (PSW))
 - etc.

Recap: Basic Instruction Execution

- Instruction X is fetched
 - Memory location indicated by **P**rogram **C**ounter
- Instruction X dispatched to the corresponding Functional Unit
 - Read operands if applicable
 - Usually from memory or GPR
 - Result computed
 - Write value if applicable
 - Usually to memory or GPR
- Instruction X is completed
 - PC updated for the next instruction

Basic Instruction Execution



Recap: What you should know 😊

- An executable (binary) consists of two major components:
 - Instructions and Data
- When a program is **under execution**, there are **more information**:
 - Memory context:
 - **Text and Data, ...**
 - Hardware context:
 - **General purpose registers, Program Counter, ...**
- Actually, there are **other types of memory usage** during program execution
 - Coming up next

Memory Context

Function Calls

What if $\mathbf{f}()$ calls $\mathbf{u}()$ calls $\mathbf{n}()$?

Function Calls : Challenges

```
int i = 0;  
  
i = i + 20;
```

C Code Fragment

VS

```
int g(int i, int j)  
{  
    int a;  
  
    a = i + j;  
    return a;  
}
```

C Code with Function

- Consider:
 - ❑ How do we allocate memory space for variables `i`, `j`, and `a`?
 - Can we just make use of the "**data**" memory space?
 - ❑ What are the key issues?

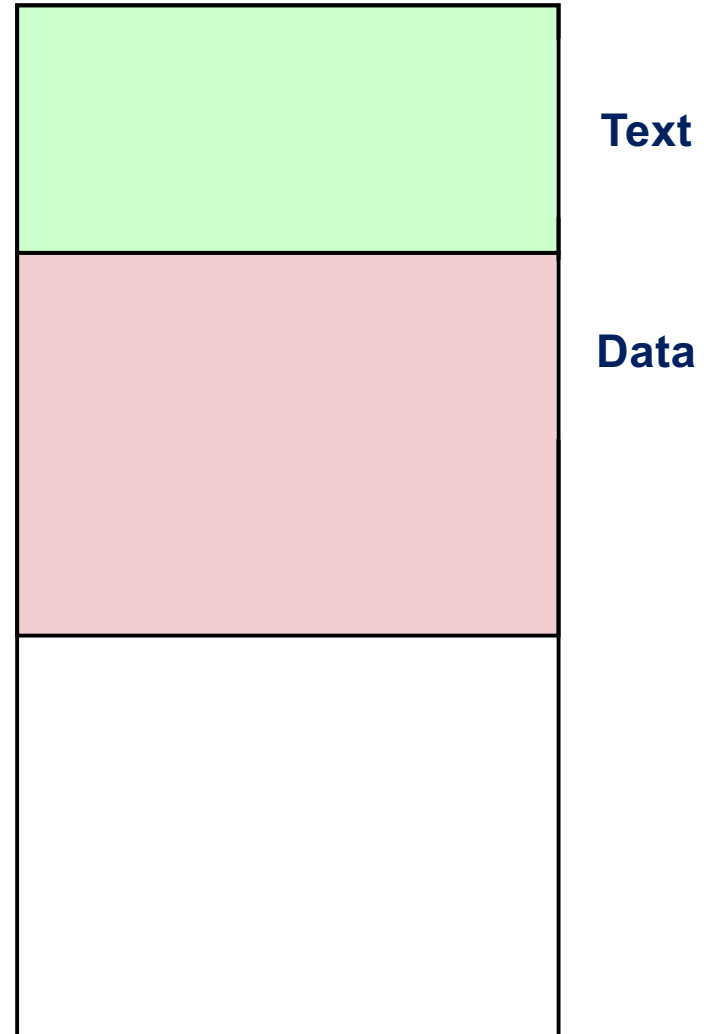
Function Call : Challenges

```
int i = 0;  
...  
i = i + 20;
```

C Code Fragment

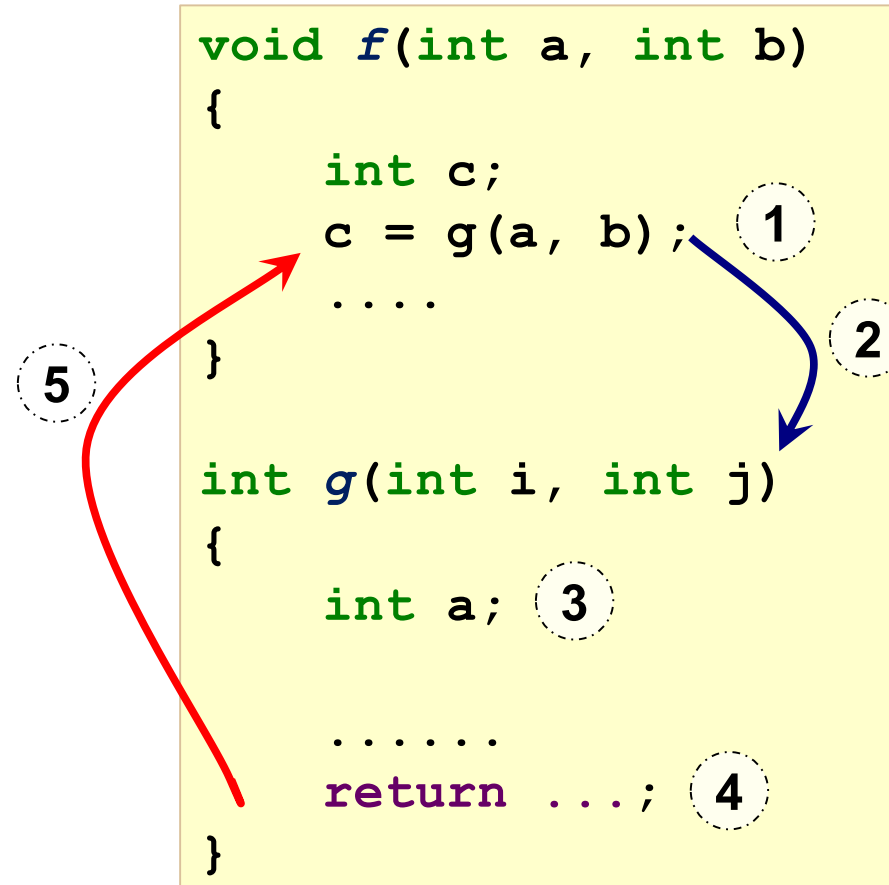
```
int g(int i, int j)  
{  
    int a;  
  
    a = i + j;  
    return a;  
}
```

C Code with Function



Function Calls : Control Flow and Data

- **f ()** calls **g ()**
 - ❑ **f ()** is the **caller**
 - ❑ **g ()** is the **callee**
- Important Steps:
 1. Setup the parameters
 2. Transfer control to callee
 3. Setup local variable
 4. Store result if applicable
 5. Return to caller



Function Calls : Control Flow and Data

■ Control Flow Issues:

- ❑ Need to jump to the function body
- ❑ Need to resume when the function call is done
- ➔ Minimally, need to store the PC of the caller

■ Data Storage Issues:

- ❑ Need to pass parameters to the function
- ❑ Need to capture the return result
- ❑ May have local variables declaration

➔ Need a **new region of memory** that dynamically used by function invocations

Introducing **Stack Memory**

- **Stack Memory Region:**

- The new memory region to store information function invocation

- Information required for a function invocation is contained in the **stack frame**

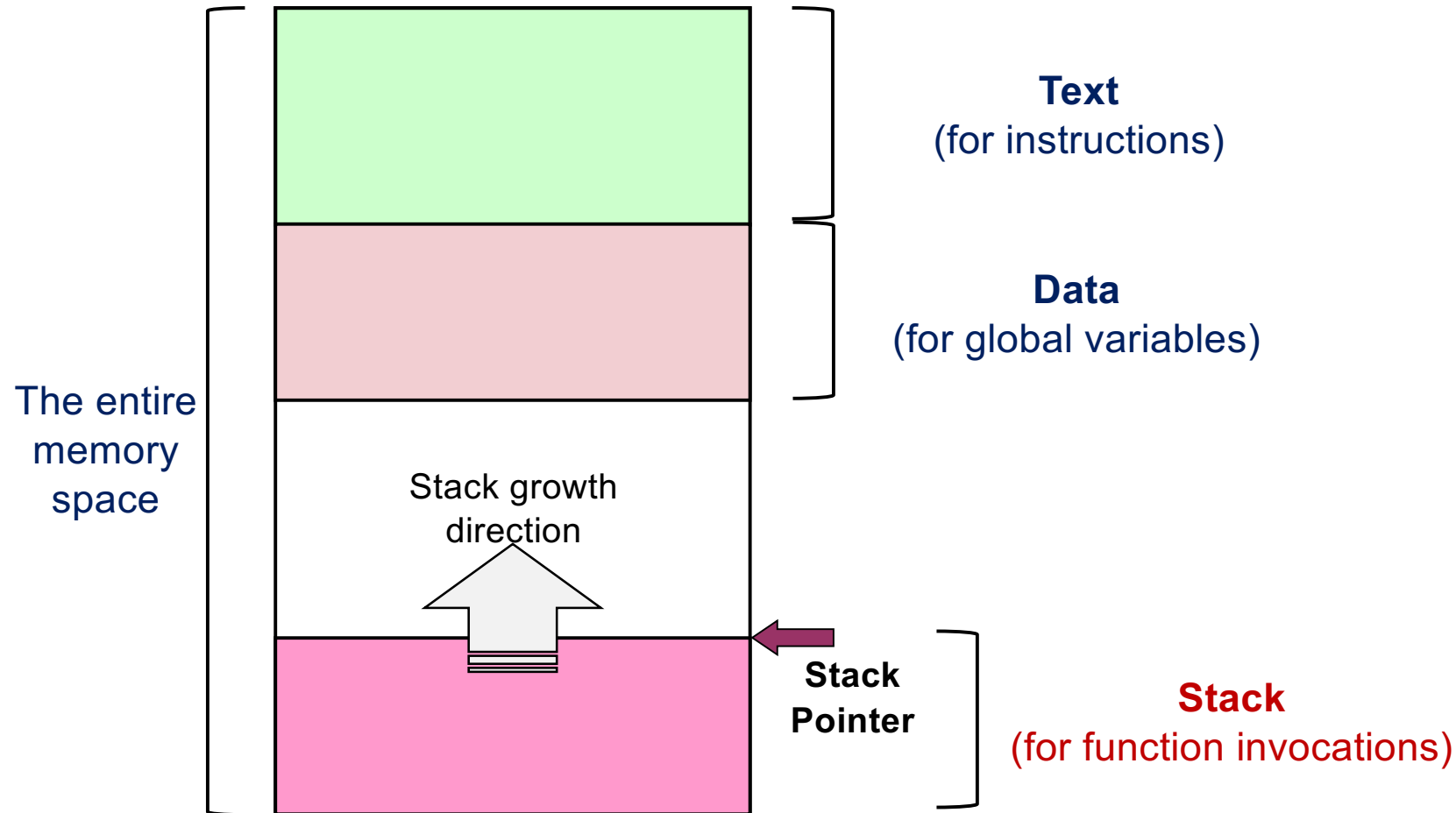
- Stack frame contains:

- Return address of the caller
- Arguments (Parameters) for the function
- Storage for local variables
- Other information.... (more later)

Stack Pointer

- The top of stack region (first unused location) is logically indicated by a **Stack Pointer**:
 - ❑ Most CPU has a specialized register for this purpose
 - ❑ Stack frame is added on top when a function is invoked
 - Stack “grows”
 - ❑ Stack frame is removed from top when a function call ends
 - Stack “shrinks”

Illustration: Stack Memory



- The memory layout on some systems is flipped, i.e. stack on top, text on the bottom

Illustration: Stack Memory Usage (1 / 5)

```
void f()  
{  
    ...  
    g();  
    ...  
}  
  
void g()  
{  
    h();  
    ...  
}  
  
void h()  
{  
    ...  
}
```

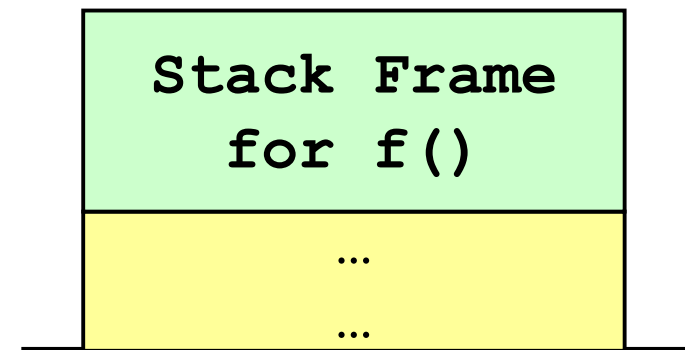

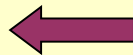


Illustration: Stack Memory Usage (2 / 5)

```
void f()  
{  
    ...  
    g();  
    ...  
}
```

```
void g()  
{  
    h();  
    ...  
}
```

```
void h()  
{  
    ...  
}
```



At this
point

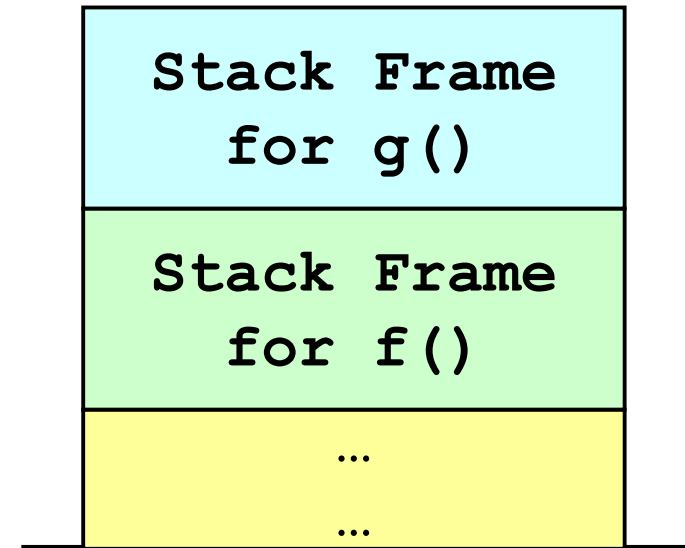


Illustration: Stack Memory Usage (3 / 5)

```
void f()  
{  
    ...  
    g();  
    ...  
}
```

```
void g()  
{  
    h();  
    ...  
}
```

```
void h()  
{  
    ...  
}
```



At this
point

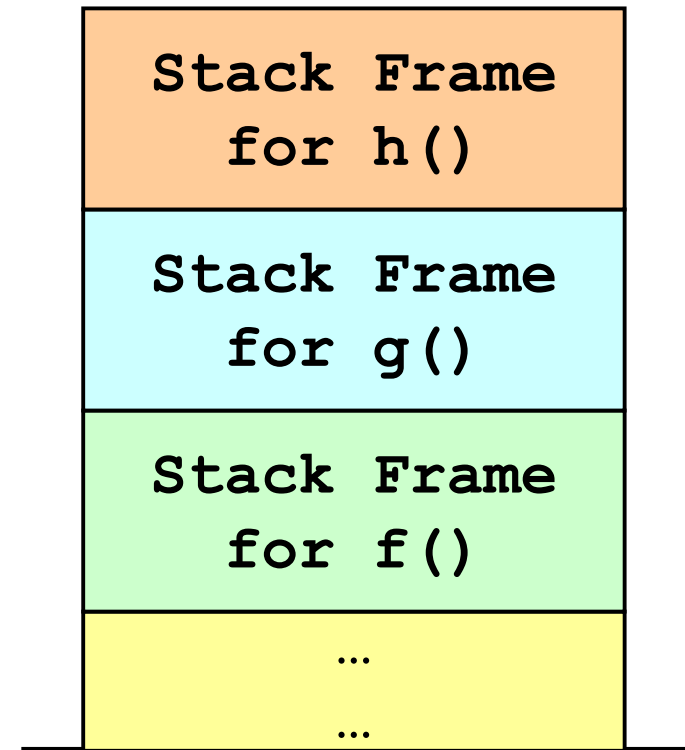


Illustration: Stack Memory Usage (4 / 5)

```
void f()  
{  
    ...  
    g();  
    ...  
}
```

```
void g()  
{  
    h();  
    ...  
}
```

At this
point

```
void h()  
{  
    ...  
}
```

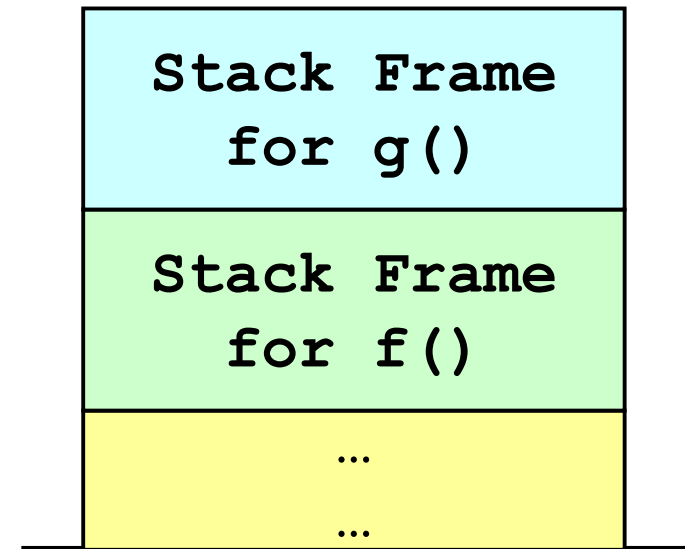


Illustration: Stack Memory Usage (5 / 5)

```
void f()  
{  
    ...  
    g();  
    ...  
}  
  
void g()  
{  
    h();  
    ...  
}  
  
void h()  
{  
    ...  
}
```

At this
point

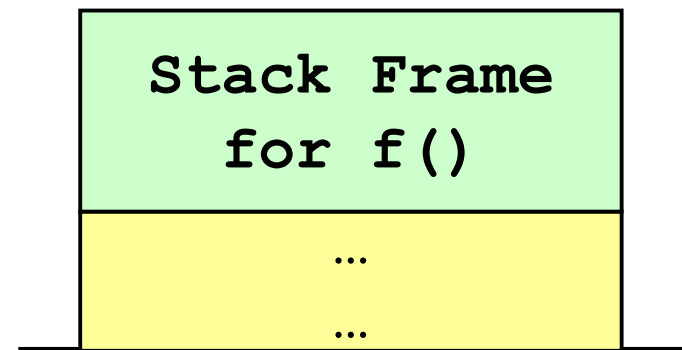
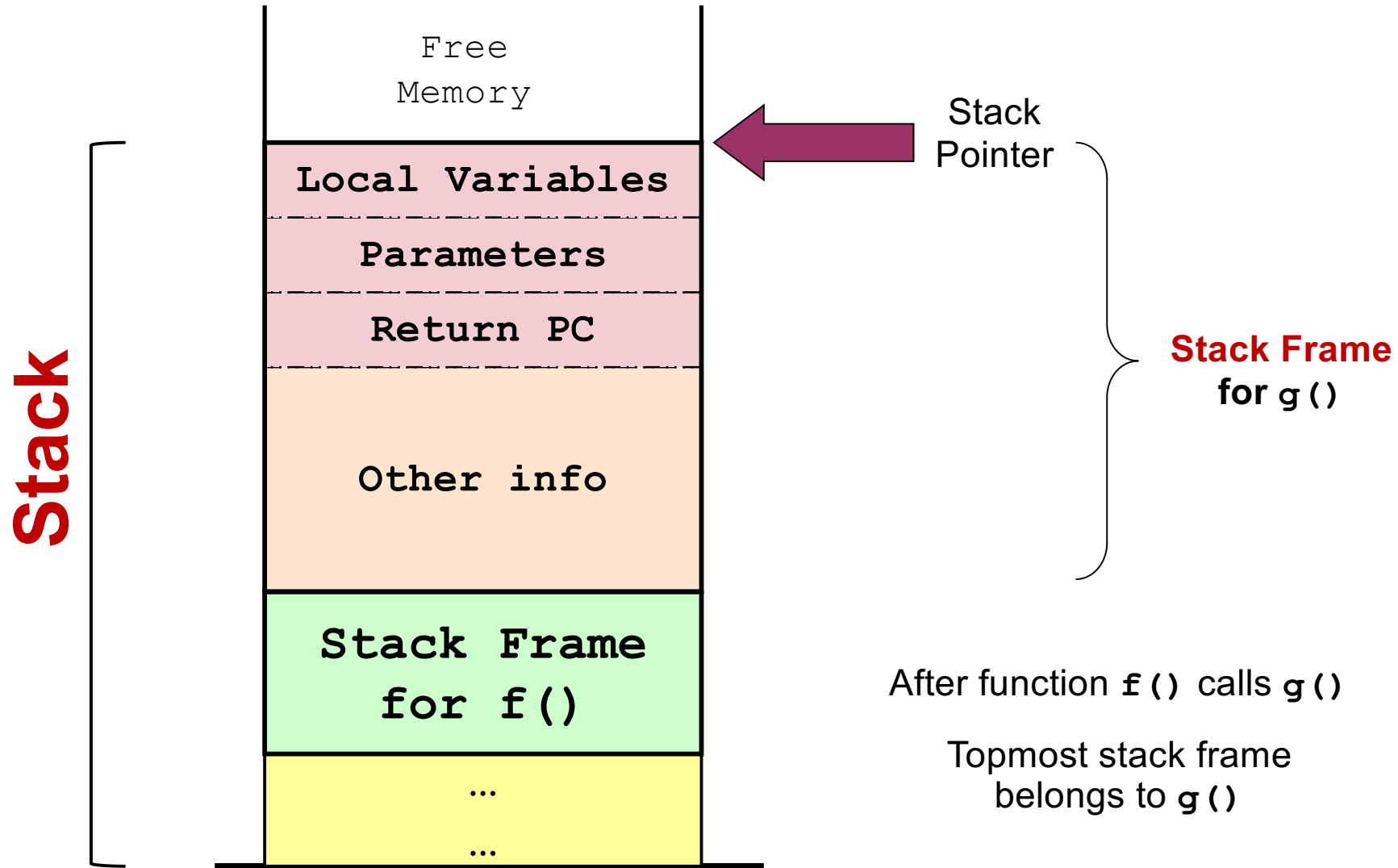


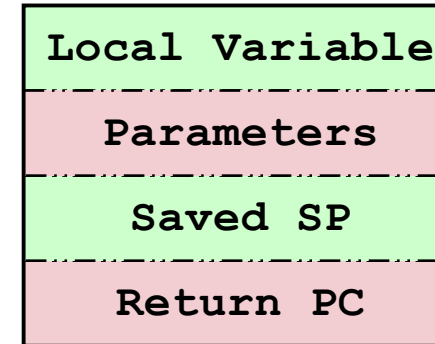
Illustration: Stack Frame v1.0



Function Call Convention

- Different ways to setup stack frame:
 - Known as **function call convention**
 - Main differences:
 - What information is stored in stack frame or registers?
 - Which portion of stack frame is prepared by caller / callee?
 - Which portion of stack frame is cleared by caller / callee?
 - Who between caller / callee to adjust the stack pointer?
- No universal way
 - Hardware and programming language dependent
- An example scheme is described next

Stack Frame Setup



■ Prepare to make a function call:

- ❑ **Caller:** Pass parameters with registers and/or stack
- ❑ **Caller:** Save Return PC on stack

❑ **Transfer Control from Caller to Callee**

- ❑ **Callee:** Save the old Stack Pointer (SP)
- ❑ **Callee:** Allocate space for local variables of callee on stack
- ❑ **Callee:** Adjust SP to point to new stack top

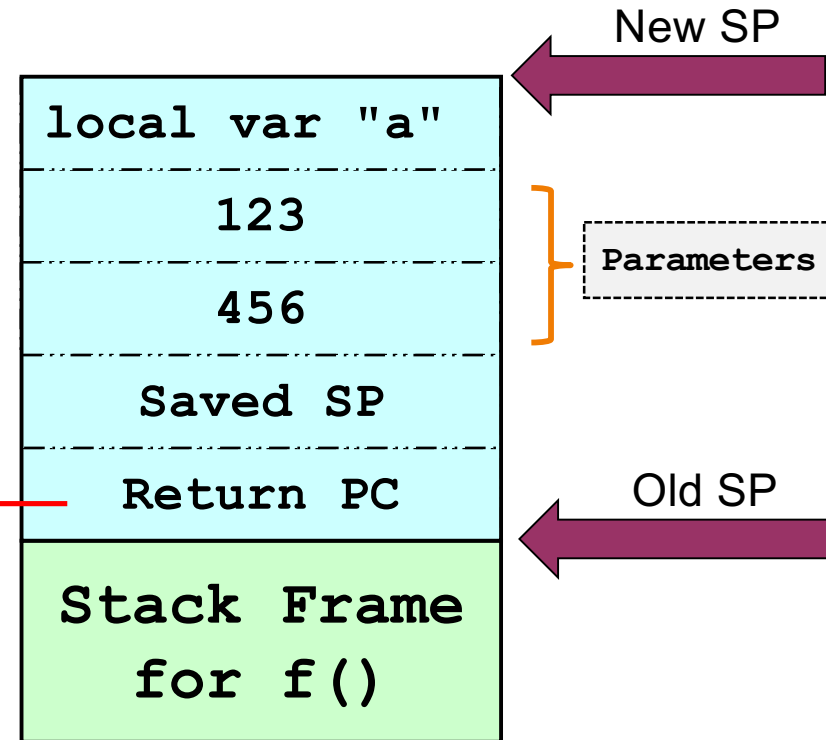
Illustration: Calling function *g* ()

```
void f(int a, int b)
{
    int c;

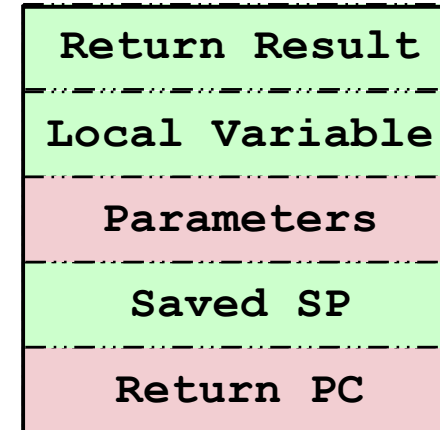
    a = 123;
    b = 456;
    c = g(a, b);
    ....
}

int g(int i, int j)
{
    int a;

    a = i + j;
    return a * 2;
}
```



Stack Frame Teardown



■ On returning from function call:

- ❑ **Callee:** Place return result on stack (if applicable)

- ❑ **Callee:** Restore saved Stack Pointer

- ❑ Transfer control back to caller using saved PC

- ❑ **Caller:** Utilize return result (if applicable)

- ❑ **Caller:** Continues execution in caller

Illustration: Function *g* () finishes

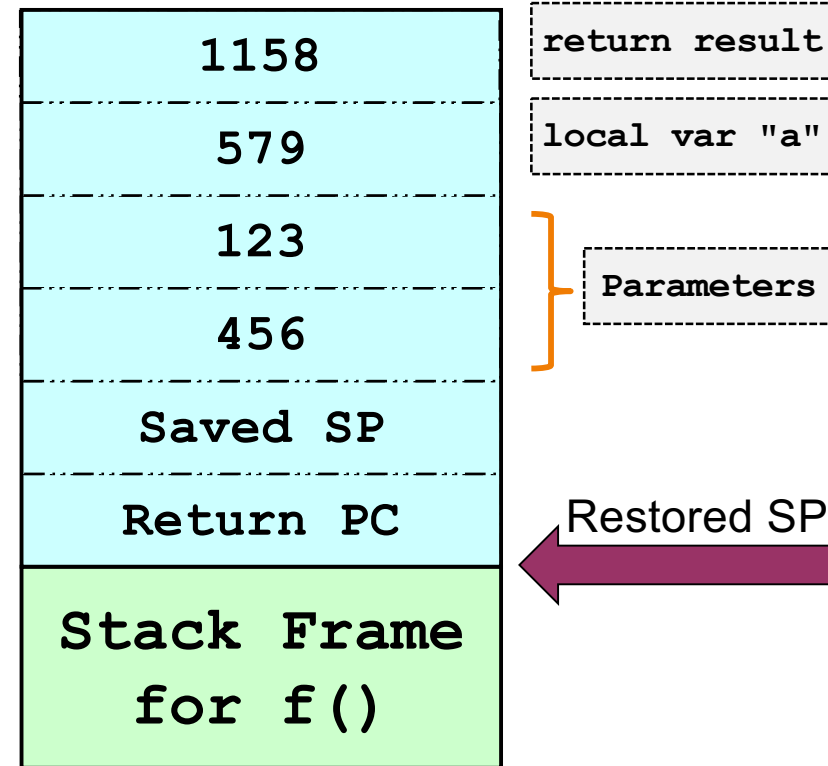
```
void f(int a, int b)
{
    int c;

    a = 123;
    b = 456;
    c = g(a, b);
    ....
}
```

Execution
resumes here

```
int g(int i, int j)
{
    int a;

    a = i + j;
    return a * 2;
}
```



Other Information in Stack Frame

- We have described the basic idea of:
 - Stack frame
 - Calling Convention: Setup and Teardown
- Let us look at a few common additional information in the stack frame:
 - Frame Pointer
 - Saved Registers

Frame Pointer

- To facilitate the access of various stack frame items:
 - Stack Pointer is hard to use as it can change
 - ➔ Some processors provide a dedicated register ***Frame Pointer***
- The frame pointer points to a fixed location in a stack frame
 - Other items are accessed as a displacement from the frame pointer
- The usage of FP is platform dependent

Saved Registers

- The number of general purpose register (GPR) on most processors are very limited:
 - E.g. MIPS has 32 GPRs, x86 has 16 GPRs
- When GPRs are exhausted:
 - Use memory to temporary hold the GPR value
 - That GPR can then be reused for other purpose
 - The GPR value can be restored afterwards
 - known as **register spilling**
- Similarly, a function can spill the registers it intend to use before the function starts
 - Restore those registers at the end of function

Saved Registers

- **GPRs** on most processors are very limited:
 - E.g., MIPS has 32 GPRs, x86 has 16 GPRs
- When GPRs are exhausted:

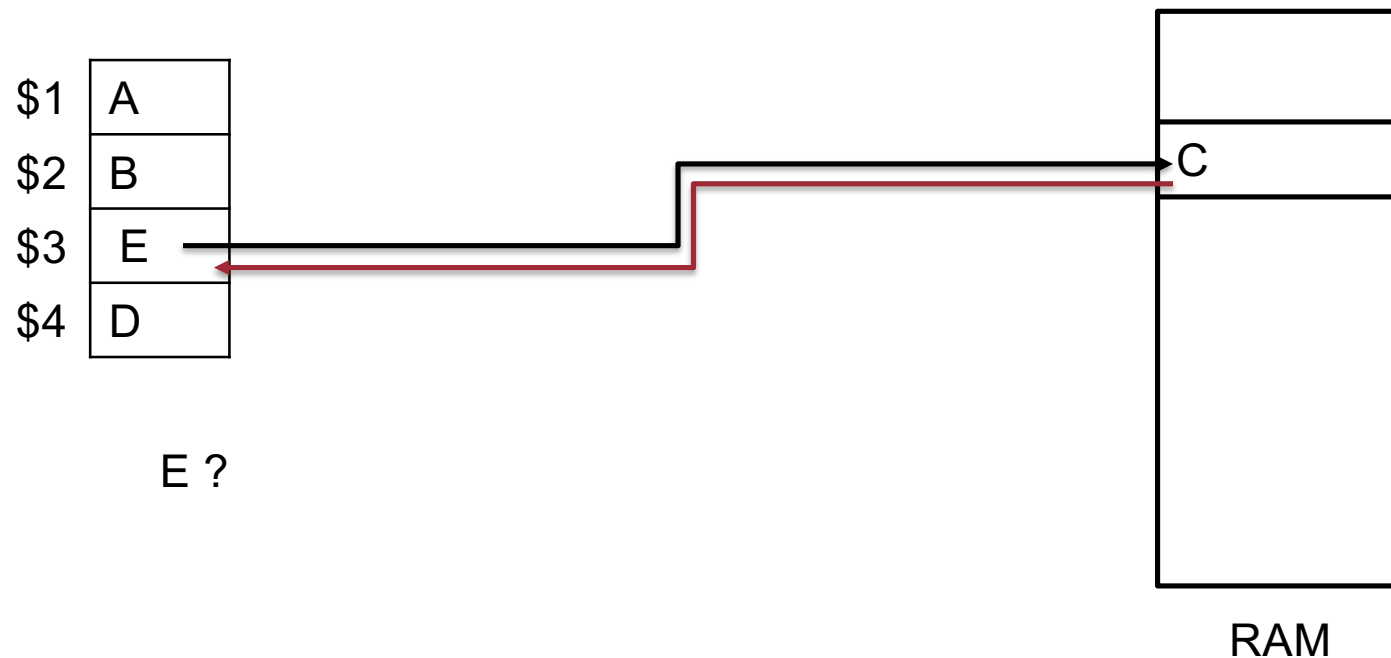
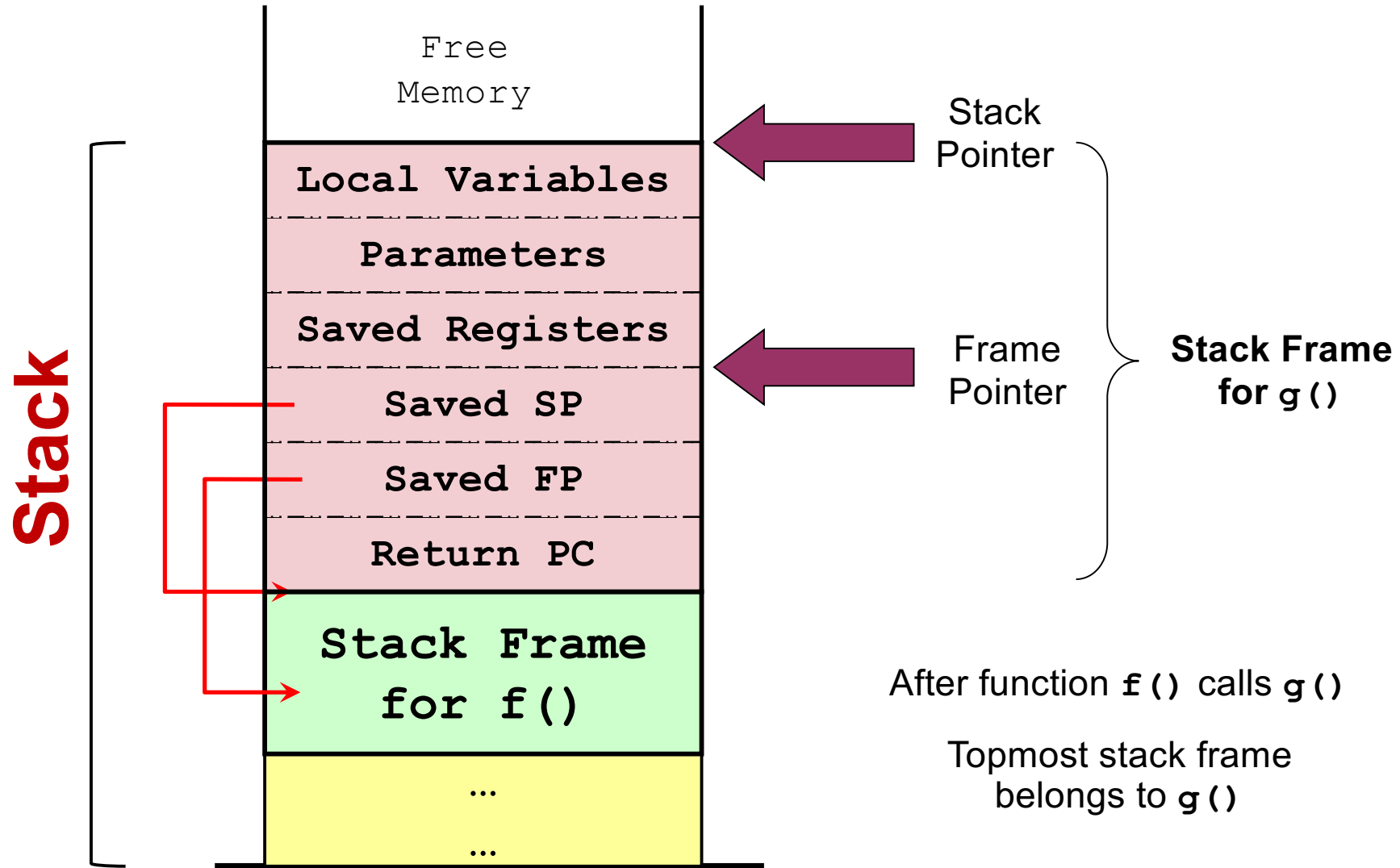


Illustration: Stack Frame v2.0



Stack Frame Setup / Teardown [Updated]

■ On executing function call:

- ❑ **Caller:** Pass arguments with registers and/or stack
- ❑ **Caller:** Save Return PC on stack
- ❑ **Transfer control from caller to callee**

- ❑ **Callee:** Save registers used by callee. Save old FP, SP
- ❑ **Callee:** Allocate space for local variables of callee on stack
- ❑ **Callee:** Adjust SP to point to new stack top

■ On returning from function call:

- ❑ **Callee:** Restore saved registers, FP, SP
- ❑ Transfer control from callee to caller using saved PC
- ❑ **Caller:** Continues execution in caller

■ Remember, just an example!

Function Call Summary

- In this part, we learned:
 - Another portion of memory space is used as a **Stack Memory**
 - Stack Memory stores the executing function using **Stack Frame**
 - Typical information stored on a stack frame
 - Typical scheme of setting up and tearing down a stack frame
 - The usage of Stack Pointer and Frame Pointer

Function Call: The BIG picture

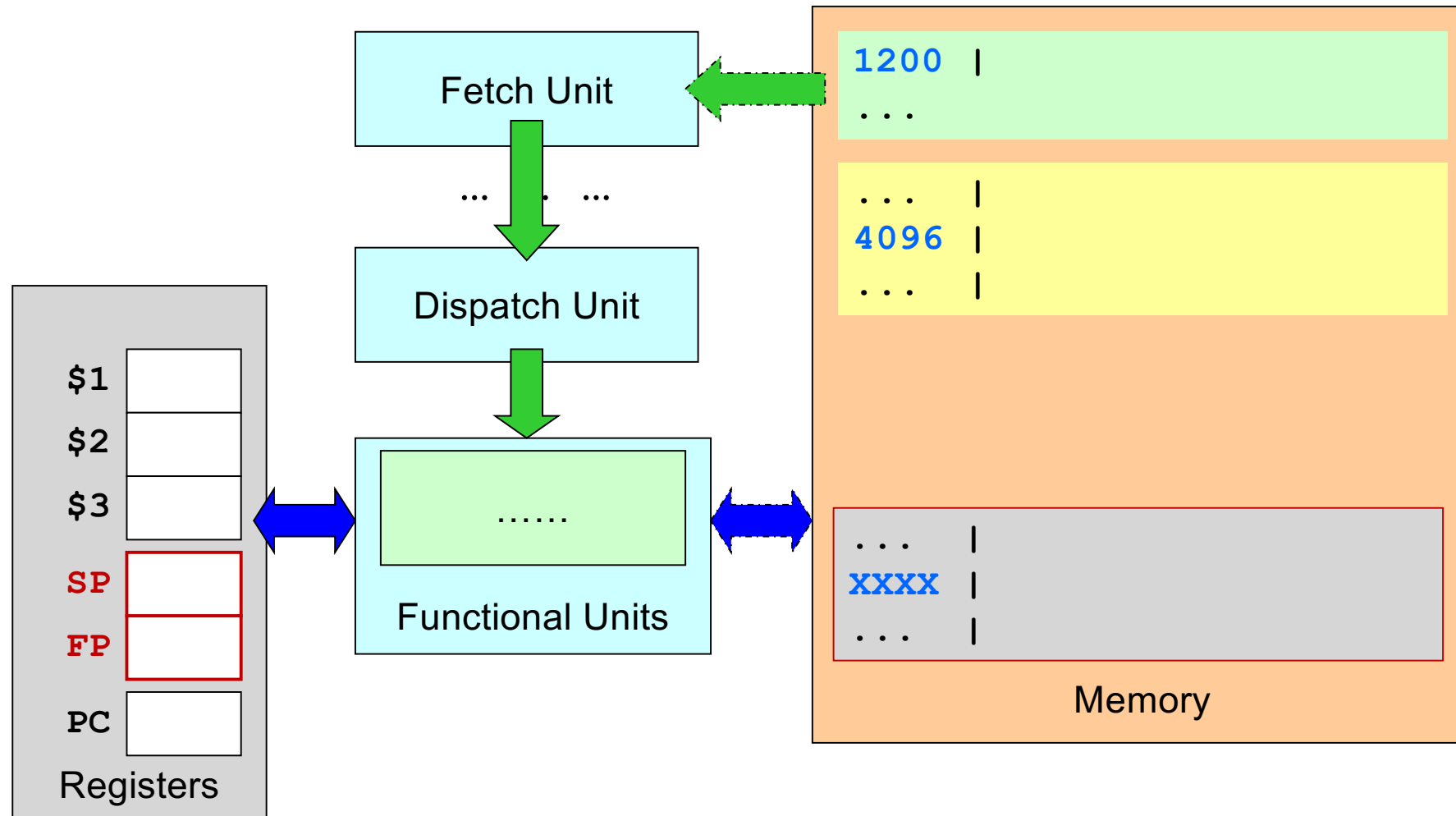
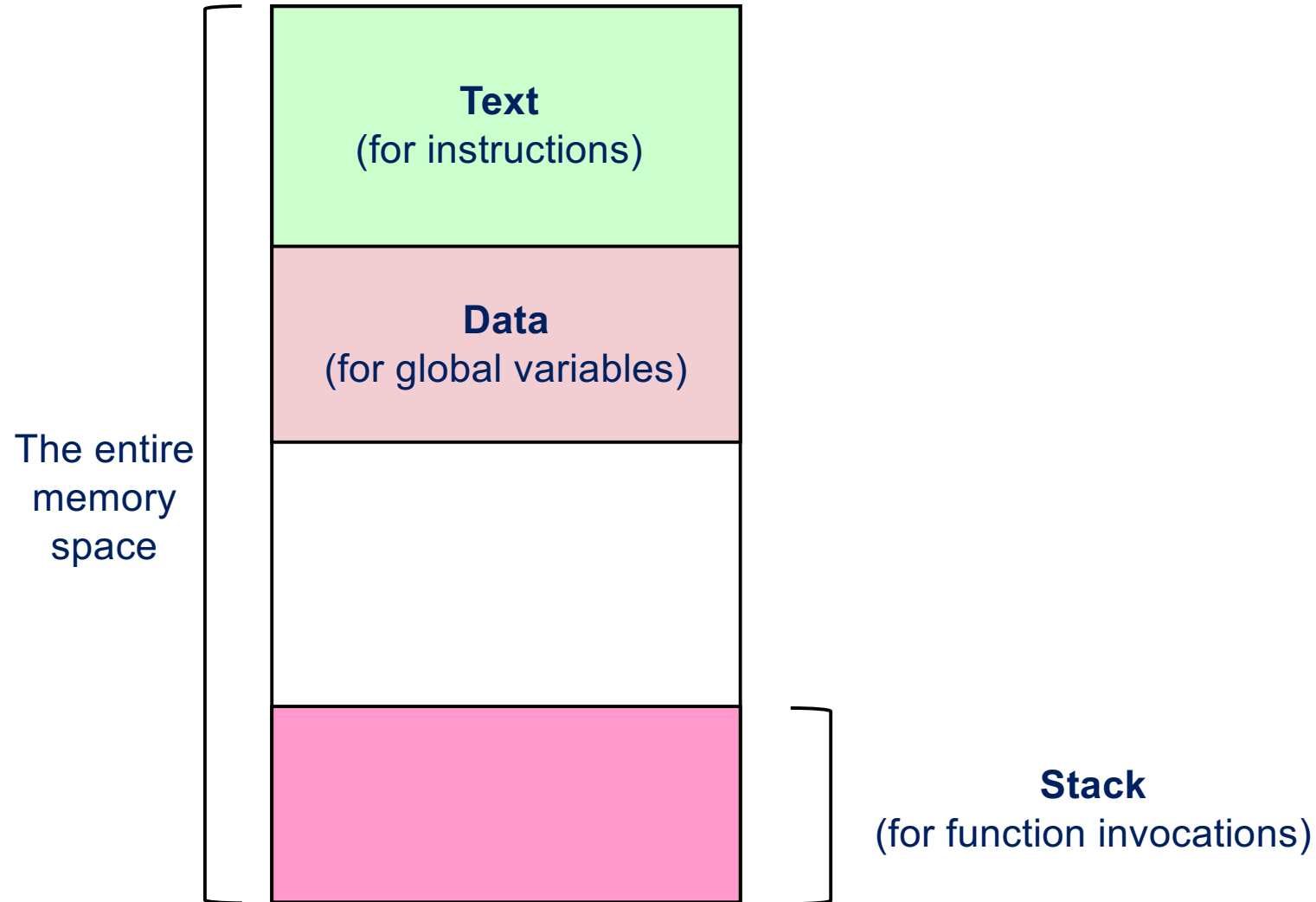


Illustration: Memory Regions So Far



Memory Context

Dynamically Allocated Memory

Hmm... I need more memory

Dynamically Allocated Memory

- Most programming languages allow dynamically allocated memory:
 - i.e. acquire memory space during **execution time**
- Examples:
 - In C, the ***malloc()*** function call
 - In C++, the ***new*** keyword
 - In Java, the ***new*** keyword
- Question:
 - Can we use the existing "Data" or "Stack" memory regions?

Dynamically Allocated Memory

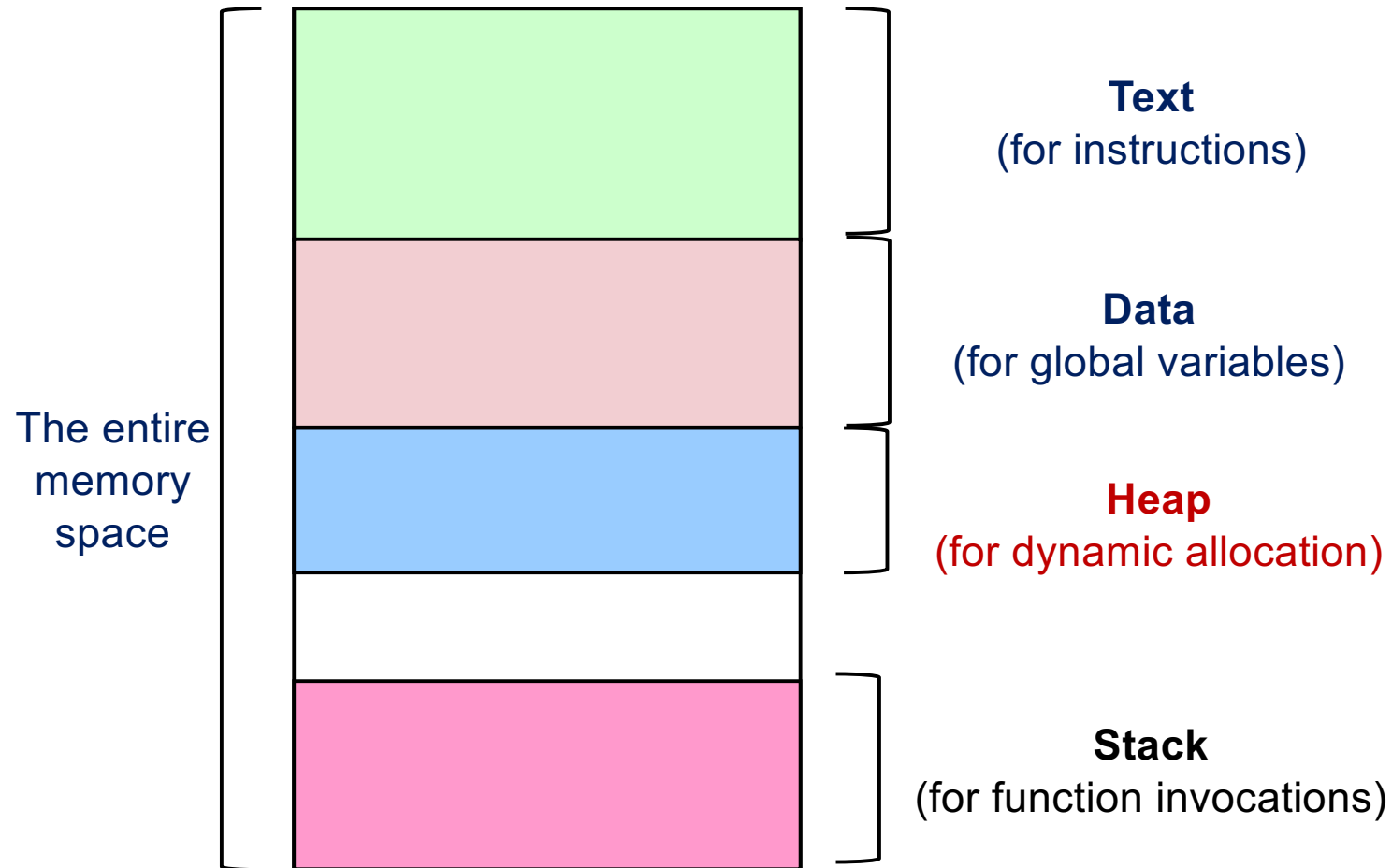
■ Observations:

- ❑ These memory blocks have different behaviors:
 1. Allocated only at runtime, i.e. size is not known during compilation time → Cannot be placed in **Data** region
 2. No definite deallocation timing, e.g. can be explicitly freed by programmer in C/C++, can be implicitly freed by garbage collector in Java → Cannot be placed in **Stack** region

■ Solution:

- ❑ Setup a separate **heap memory region**

Illustration for Heap Memory



Managing Heap Memory

- Heap memory is a lot trickier to manage due to its nature:
 - Variable size
 - Variable allocation / deallocation timing
- You can easily construct a scenario where heap memory are allocated /deallocated in such a way to create "holes" in the memory
 - Free memory block squeezed in between of occupied memory block
- We will learn more in the memory management (much) later in the course

Checkpoint: Contexts updated

- Information describing a **process**:
 - Memory context:
 - Text, Data, **Stack** and **Heap**
 - Hardware context:
 - General purpose registers, Program Counter, **Stack pointer**, **Stack frame pointer**,

OS Context

Process Id & Process State

Your ID? Give me a status report!

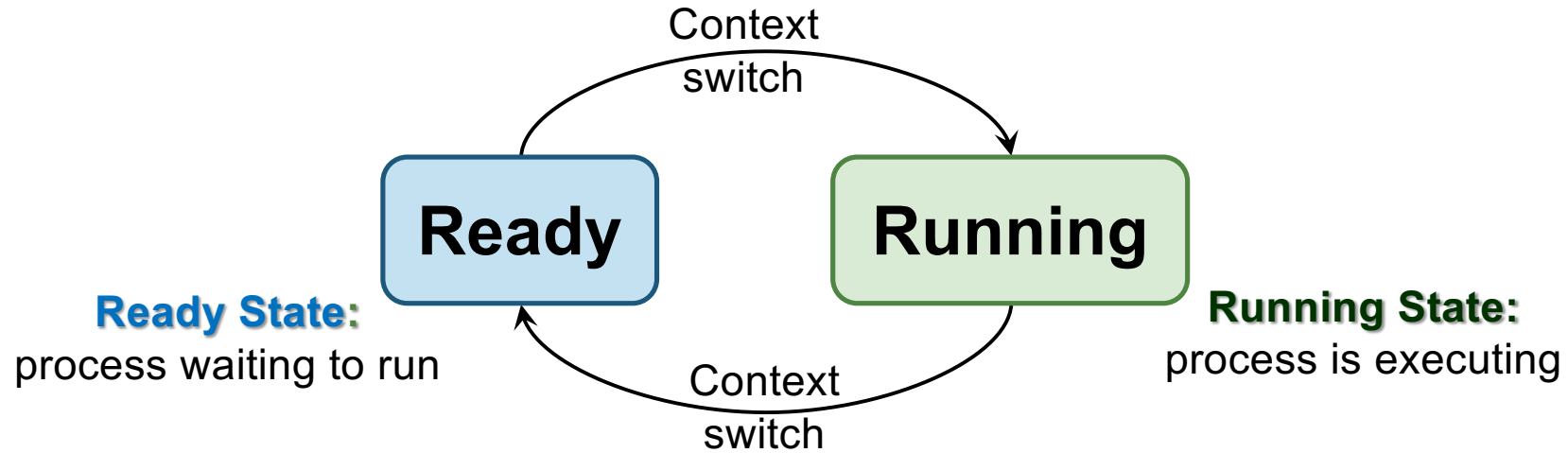
Process Identification

- To distinguish processes from each other
 - ❑ Common approach is to use process ID (**PID**)
 - Just a number
 - ❑ Unique among processes
- There are a couple of OS dependent issues:
 - ❑ Are PIDs reused?
 - ❑ Does it limit the maximum no. of processes?
 - ❑ Are there reserved PIDs?

Introducing Process State

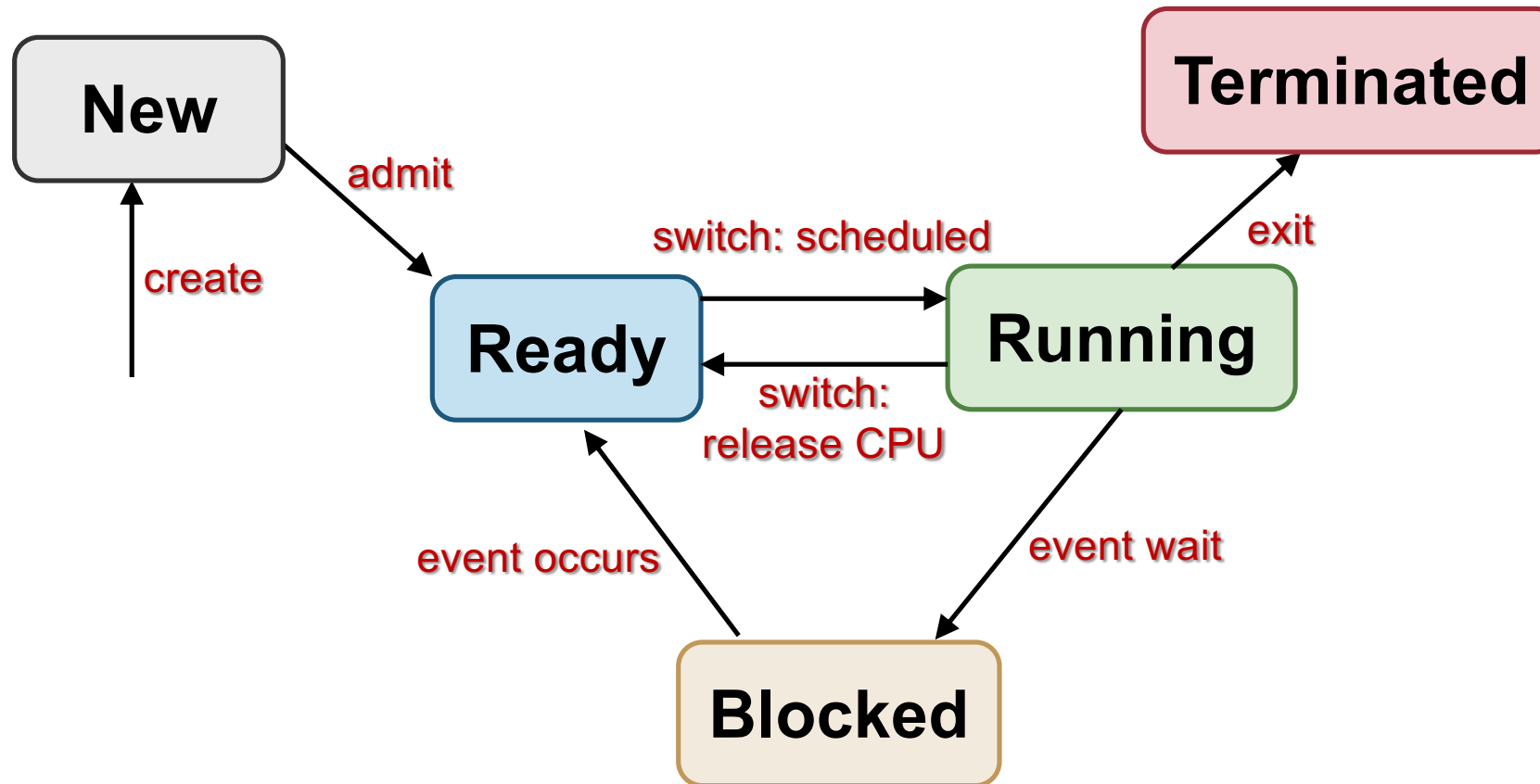
- With the multi-process scenario:
 - A process can be:
 - Running OR
 - Not-running, eg. another process running
- A process can be **ready to run**
 - But not actually executing
 - E.g. waiting for its turn to use the CPU
- Hence, each process should have a **process state**:
 - As an indication of the execution status

(Simple) Process Model State Diagram



- The set of states and transitions are known as **process model**
 - ❑ Describes the behaviors of a process

Generic 5-State Process Model



Notes: generic process states, details vary in actual OS

Process States for 5-Stage Model

■ New:

- ❑ New process created
- ❑ May still be under initialization → not yet ready

■ Ready:

- ❑ process is waiting to run

■ Running:

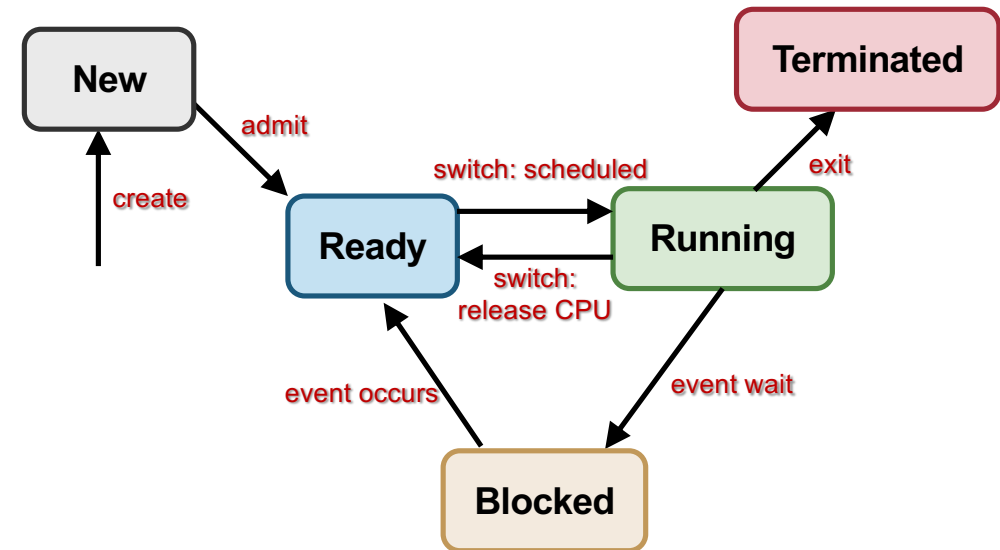
- ❑ Process being executed on CPU

■ Blocked:

- ❑ Process waiting (sleeping) for event
- ❑ Cannot execute until event is available

■ Terminated:

- ❑ Process has finished execution, may require OS cleanup



Process State Transitions in 5-Stage Model

- **Create** (nil \rightarrow New):

- New process is created

- **Admit** (New \rightarrow Ready):

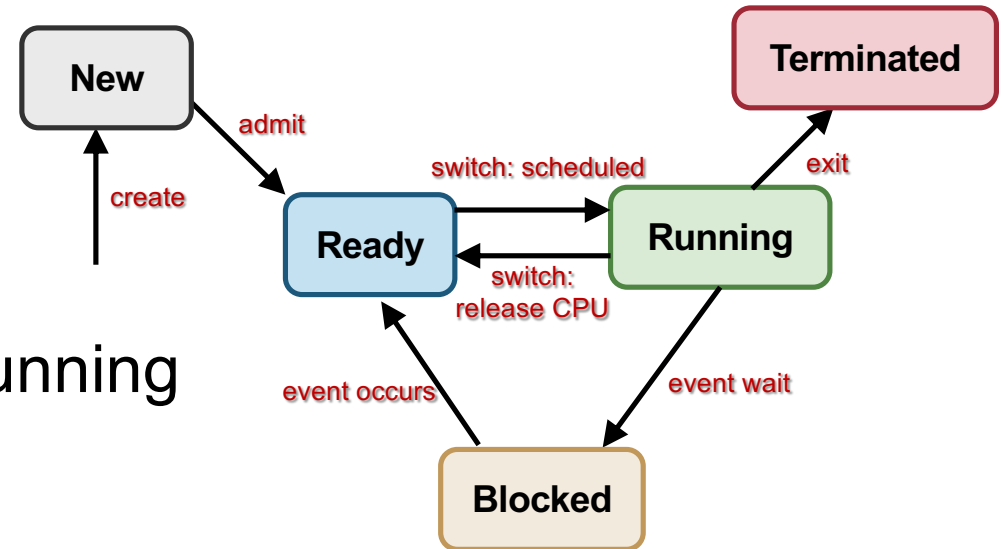
- Process ready to be scheduled for running

- **Switch** (Ready \rightarrow Running):

- Process selected to run

- **Switch** (Running \rightarrow Ready):

- Process gives up CPU voluntarily or *preempted* by scheduler



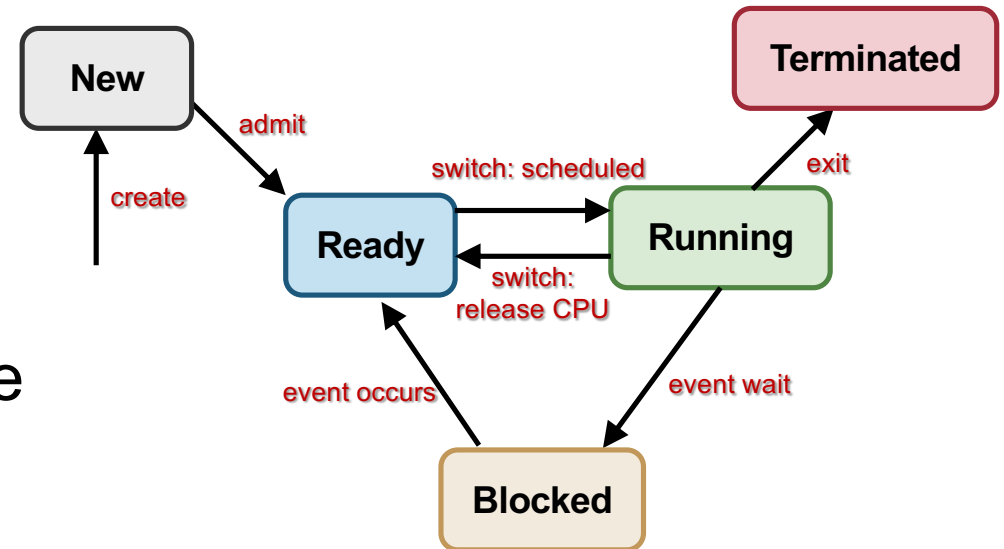
Process State Transitions

■ **Event wait** (Running → Blocked):

- ❑ Process requests event/resource/service which is not available/in progress
- ❑ Example events:
 - System call, waiting for I/O, (*more later*)

■ **Event occurs** (Blocked → Ready):

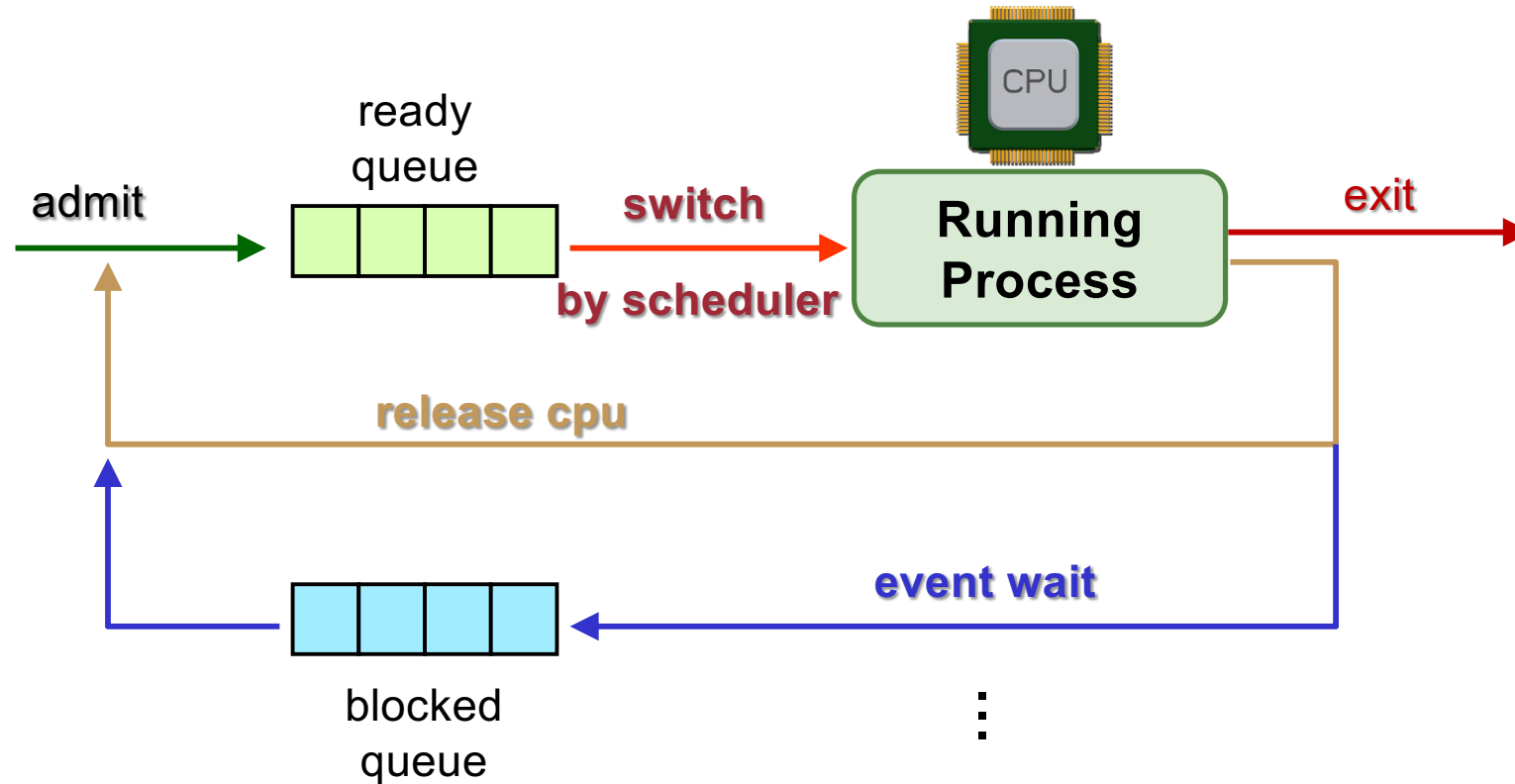
- ❑ Event occurs → process can continue



Global View of Process States

- Given n processes:
 - With 1 CPU:
 - ≤ 1 process in running state
 - conceptually 1 transition at a time
 - With m CPUs:
 - $\leq m$ process in running state
 - possibly parallel transitions
- Different processes may be in different states
 - each process may be in different part of its state diagram

Queuing Model of 5 state transition



Notes:

- More than 1 process can be in ready + blocked queues
- May have separate event queues
- Queuing model gives global view of the processes, i.e. how the OS views them

Checkpoint: Contexts updated

- When a program is **under execution**, there are **more information**:
 - Memory context:
 - Text and Data, Stack and Heap
 - Hardware context:
 - General purpose registers, Program Counter, Stack pointer, Stack frame pointer, ...
 - OS context:
 - Process ID, Process State, ...

Process Table & Process Control Block

Putting it together

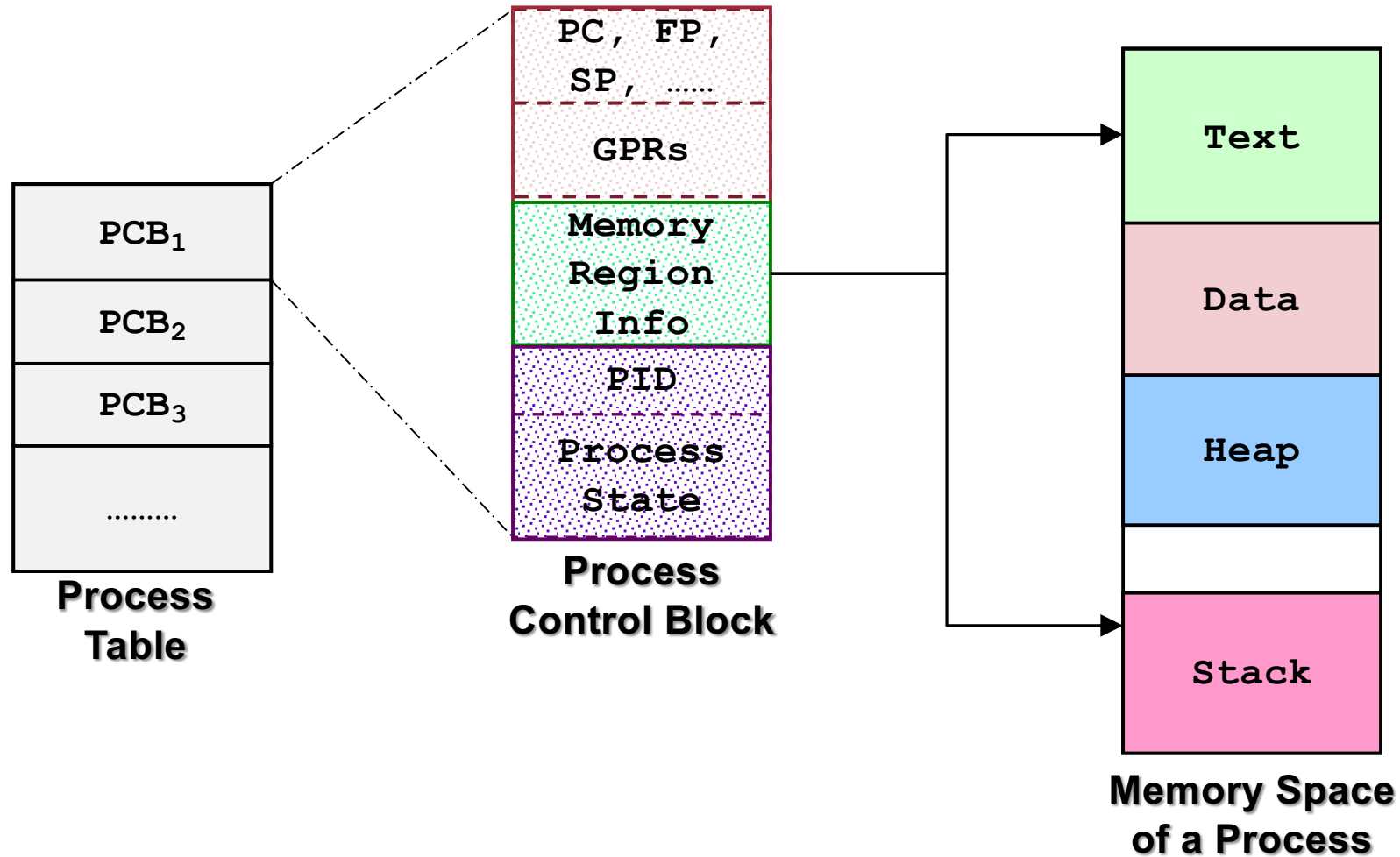
Process Control Block & Table

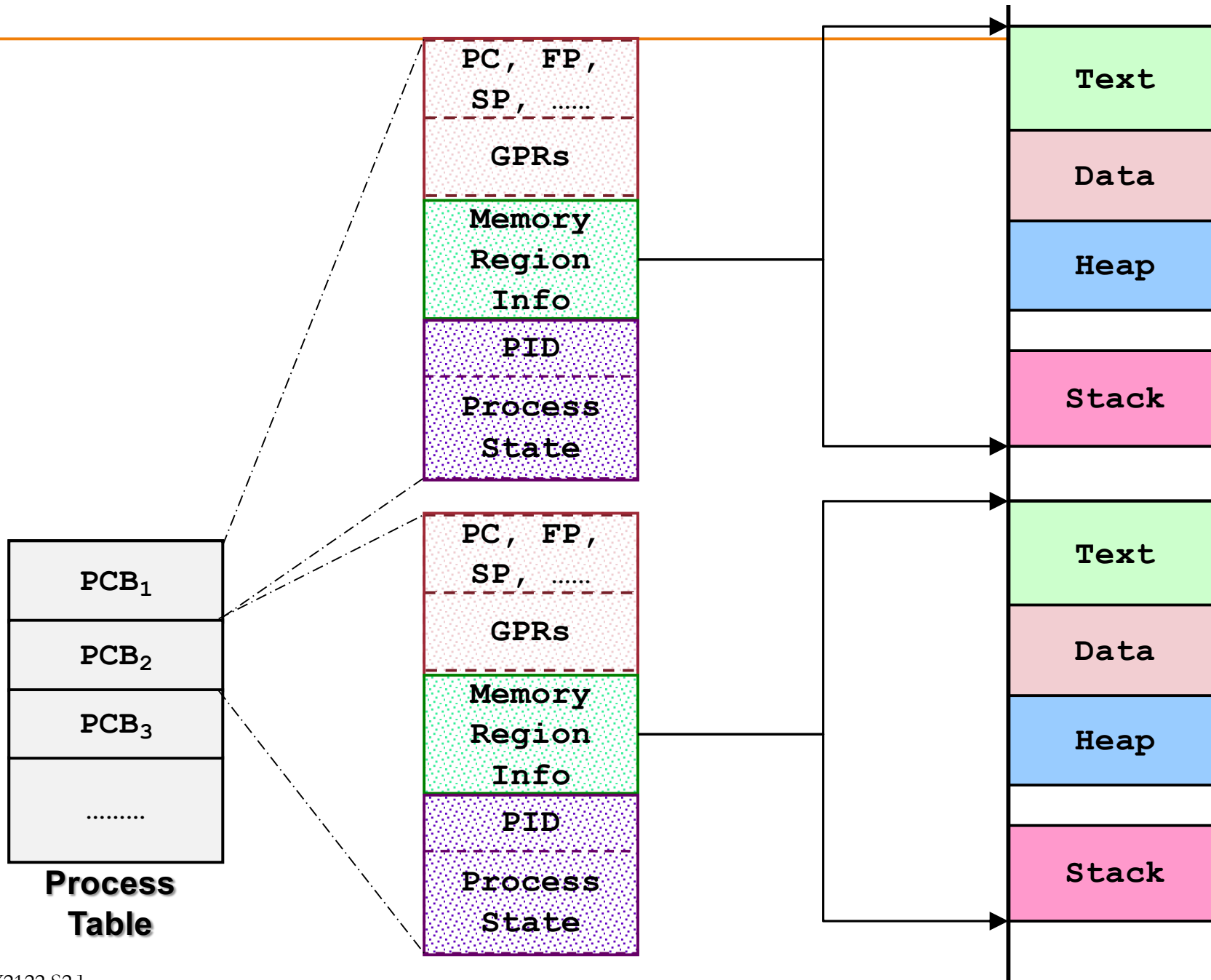
- The entire execution context for a process
 - Traditionally called **P**rocess **C**ontrol **B**lock (PCB) or **P**rocess **T**able **E**ntry
- Kernel maintains PCB for all processes
 - Conceptually stored as one table representing all processes

Interesting Issues:

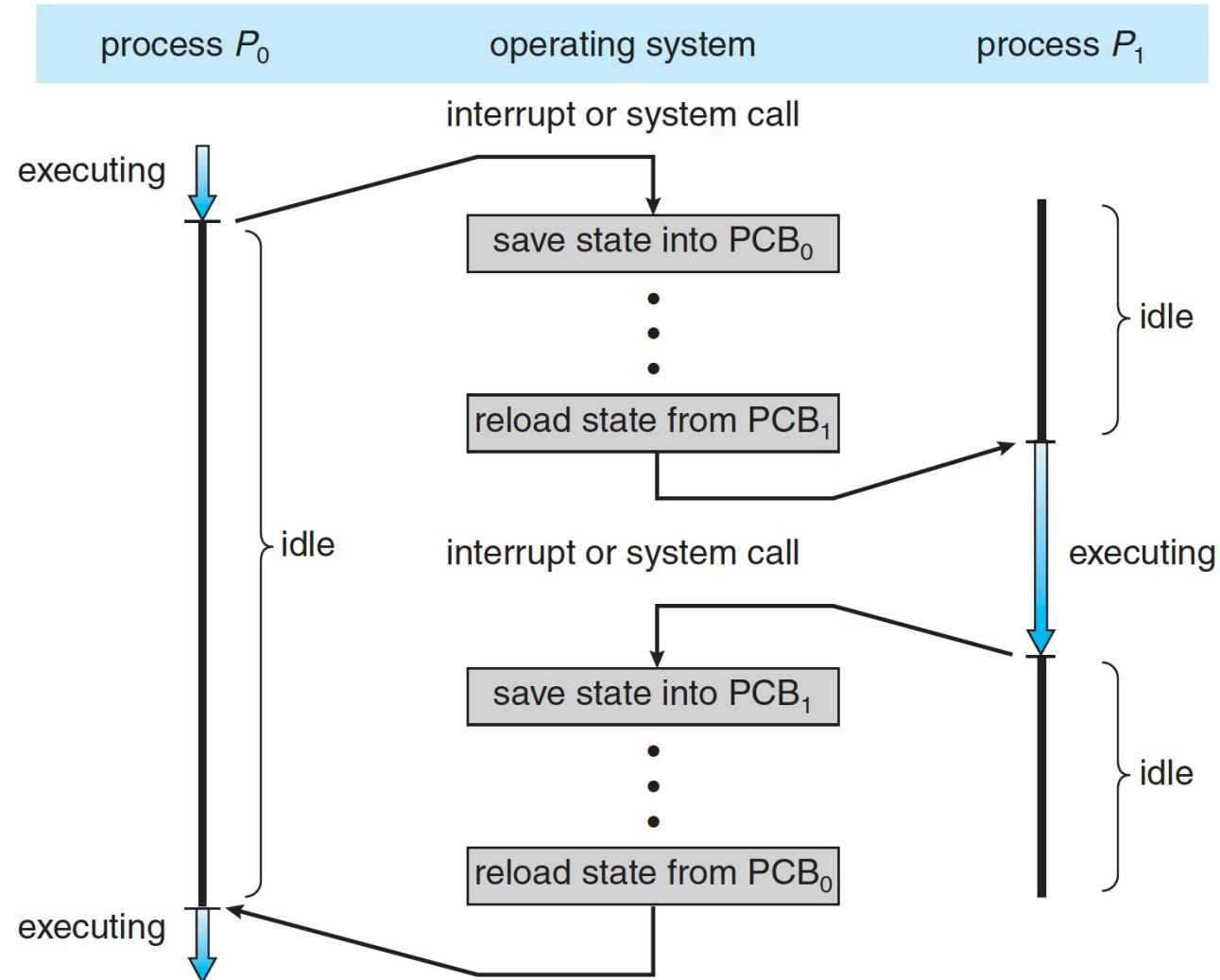
- Scalability
 - How many concurrent processes can you have?
- Efficiency
 - Should provide efficient access with minimum space wastage

Illustration of a Process Table





Context Switch



Process interaction with OS

System Calls

Can you please do this for me?

System Calls

- Application Program Interface (API) to OS
 - ❑ Provides way of calling facilities/services in kernel
 - ❑ **NOT** the same as normal function call
 - have to change from user mode to kernel mode
- Different OS have different APIs:
 - ❑ Unix Variants:
 - Most follows **POSIX** standards
 - Small number of calls: ~100
 - ❑ Windows Family:
 - Uses **Win API** across different Windows versions
 - New version of windows usually adds more calls
 - Huge number of calls:~1000

Unix System Calls in C/C++ program

- In C/C++ program, system call can be invoked *almost directly*
 - Majority of the system calls have a library version with the **same name** and the same parameters
 - The library version act as a **function wrapper**
 - Other than that, a few library functions present a more user friendly version to the programmer
 - E.g. lesser number of parameters, more flexible parameter values etc
 - The library version acts as a **function adapter**

Example

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

int main()
{
    int pid;

    /* get Process ID */
    pid = getpid();

    printf("process id = %d\n", pid);

    return 0;
}
```

Library call that
has the same
name as a
system call

Library call that
make a system
call

- System Calls invoked in this example:
 - ❑ `getpid()`
 - ❑ `write()` – made by `printf()` library call

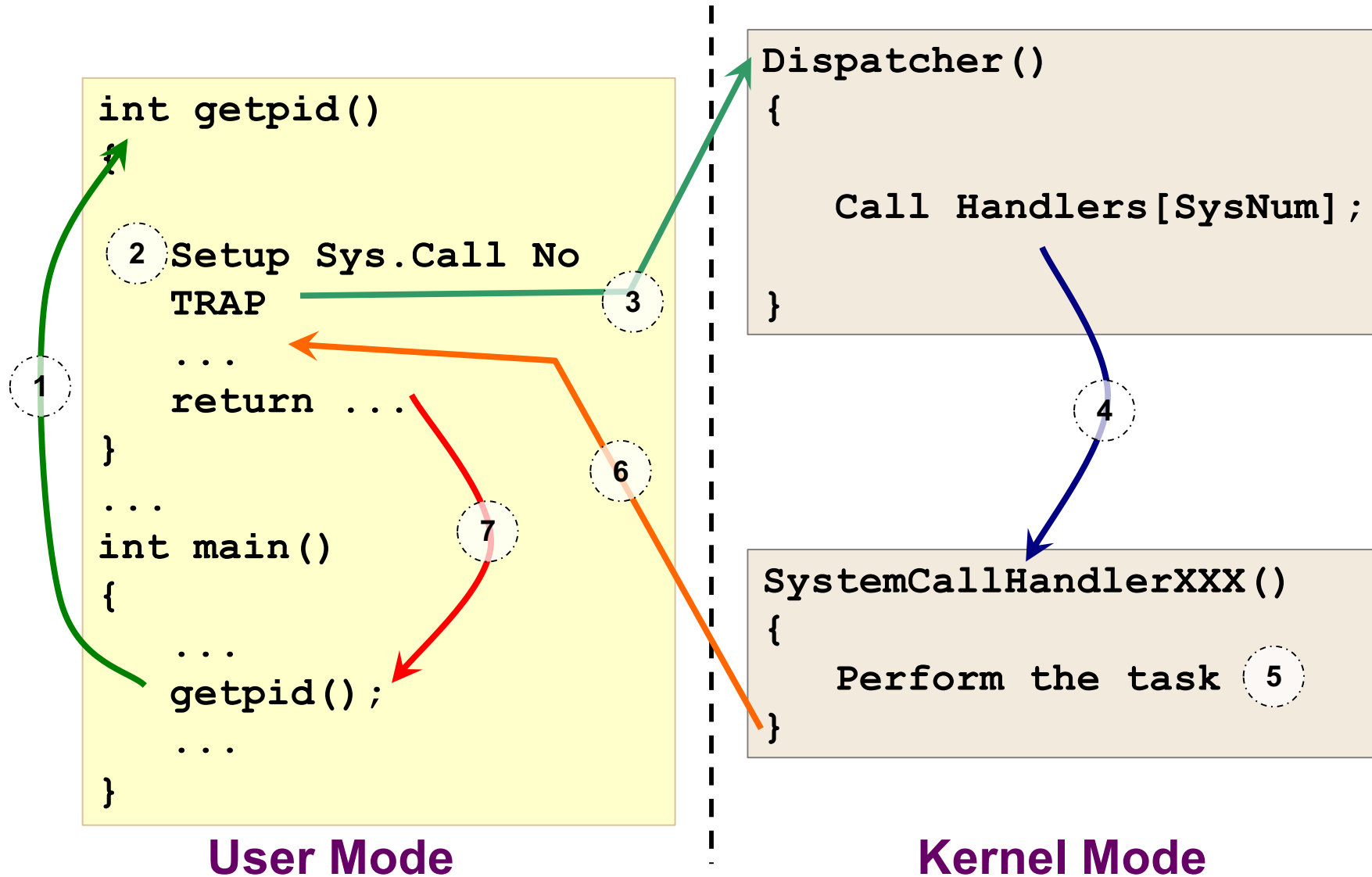
General System Call Mechanism

1. User program invokes the library call
 - Using the normal function call mechanism as discussed
2. Library call (usually in assembly code) places the **system call number** in a designated location
 - E.g. Register
3. Library call executes a special instruction to switch from user mode to kernel mode
 - That instruction is commonly known as **TRAP**

General System Call Mechanism (cont)

4. Now in kernel mode, the appropriate system call handler is determined:
 - Using the system call number as index
 - This step is usually handled by a **dispatcher**
5. System call handler is executed:
 - Carry out the actual request
6. System call handler ended:
 - Control return to the library call
 - Switch from kernel mode to user mode
7. Library call return to the user program:
 - via normal function return mechanism

Illustration: System Call Mechanism



Process interaction with OS

Exception and Interrupt

Ops!

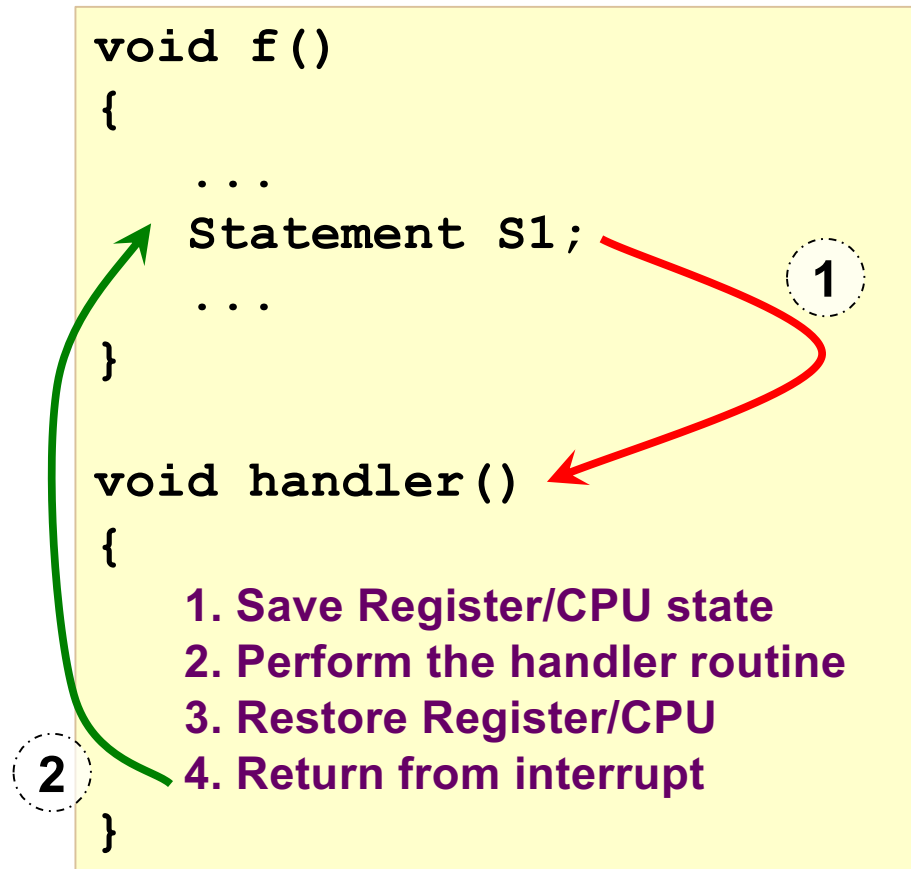
Exception

- Executing a **machine level instruction** can cause exception
- For example:
 - Arithmetic Errors
 - Overflow, Underflow, Division by Zero
 - Memory Accessing Errors
 - Illegal memory address, Mis-aligned memory access
 - Etc
- Exception is **Synchronous**
 - occur due to program execution
- Effect of exception:
 - Have to execute a **exception handler**
 - Similar to a **forced function call**

Interrupt

- External events can interrupt the execution of a program
- Usually hardware related, e.g.:
 - Timer, Mouse Movement, Keyboard Pressed etc
- Interrupt is **asynchronous**
 - Events that occurs **independent** of program execution
- Effect of interrupt:
 - Program execution is suspended
 - Have to execute an **interrupt handler**

Exception/Interrupt Handler: Illustration



1. Exception/Interrupt occurs:

- Control transfer to a handler routine **automatically**

2. Return from handler routine:

- Program execution resume
- **May** behave as if nothing happened

Summary

- Using process as an abstraction of running program:
 - Necessary information (environment) of execution
 - Memory, Hardware and OS contexts
- Process from OS perspective:
 - PCB and process table
- OS \leftrightarrow Process interactions
 - System calls
 - Exception / Interrupt

References

- Modern Operating System (3th Edition)
 - Section 2.1
- Operating System Concepts (9th Edition)
 - Section 3.1

