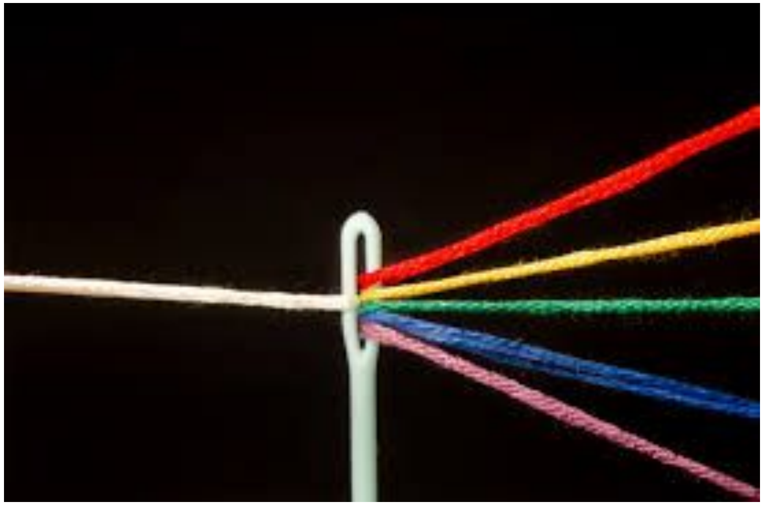
CS310 Operating Systems

Lecture 12: Thread Implementation



References

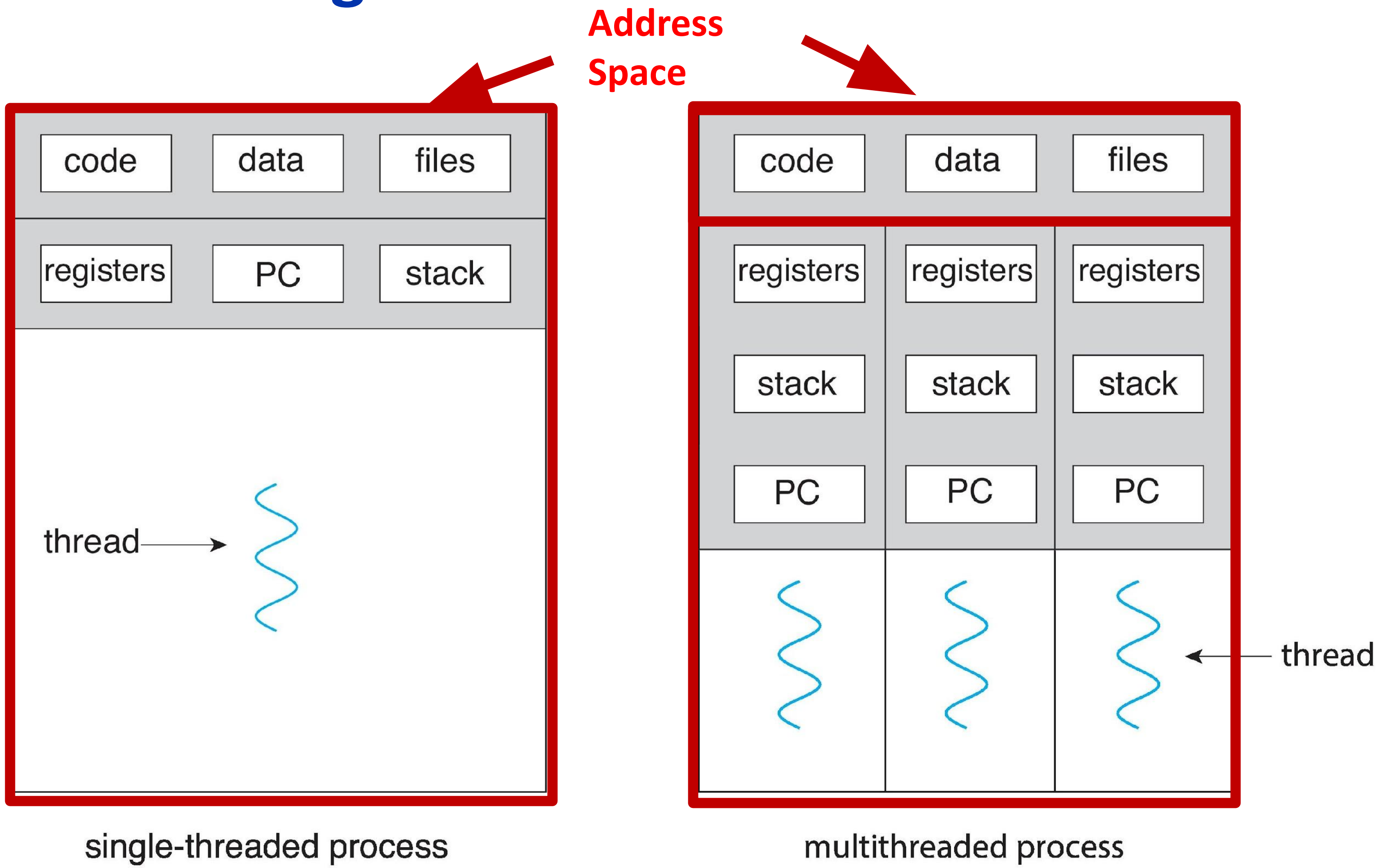
- Book: Operating Systems: Principles and Practice: Thomas Anderson and Michael Dahlin, Part 1 and Part 2
- CS162, Operating Systems and Systems Programming, University of California, Berkeley
- CS4410, Operating Systems, Course, Cornell University, Spring 2019, Lecture on Threads
- Operating Systems: Three Easy Pieces, by Remzi and Andrea Arpaci-Dusseau,
  available for free online
- Book: Modern Operating Systems, Andrew Tenenbaum, and Herbert Bos, 4th Edition, Pearson

Read the following:

- Book: Operating Systems: Principles and Practice (2nd Edition) Anderson and Dahlin
  - Volume 2, Concurrency
    - Chapter 4: Concurrency and Threads
- Book: Modern Operating Systems: Tenenbaum and Bos
  - Chapter 2: Processes and Threads

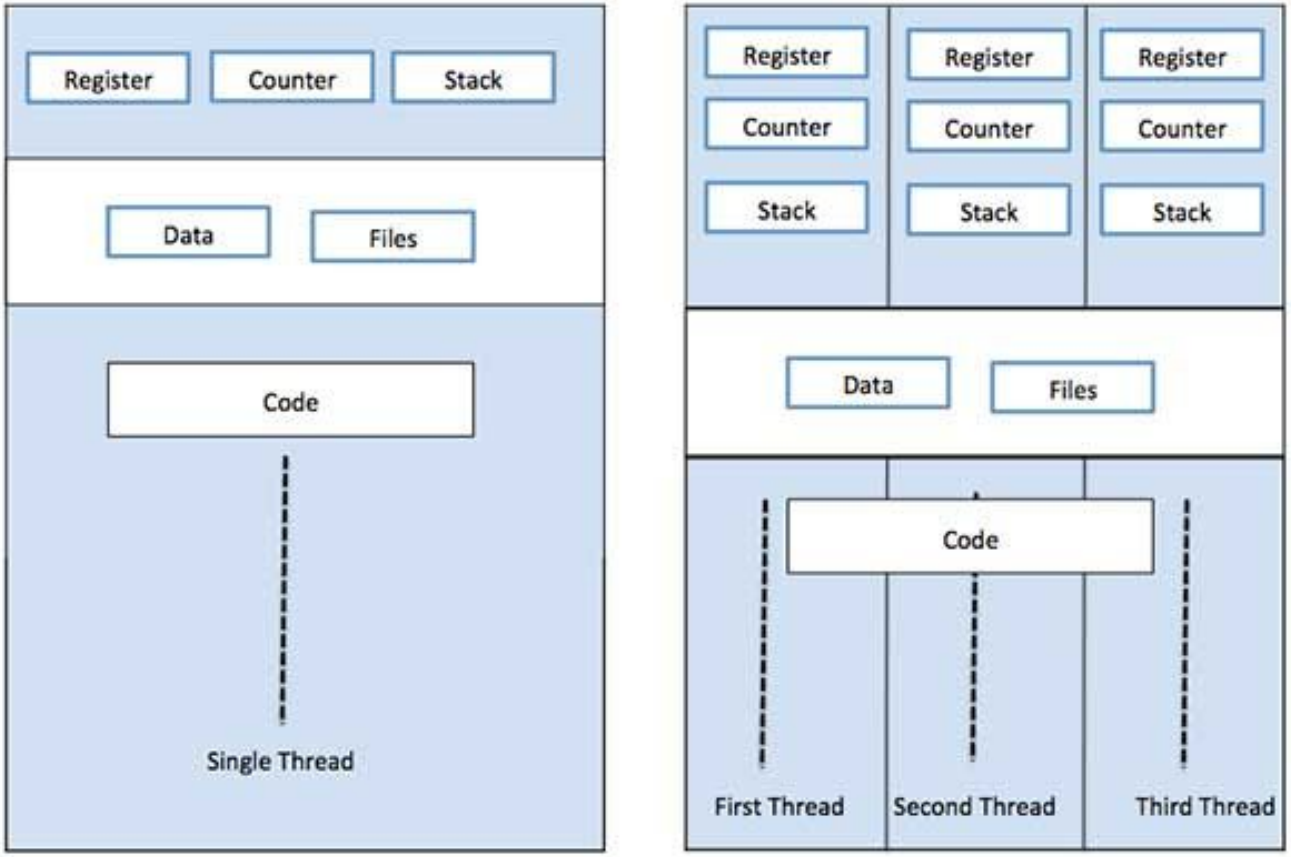
We have learnt so far ..

Recall: Single and Multithreaded Processes



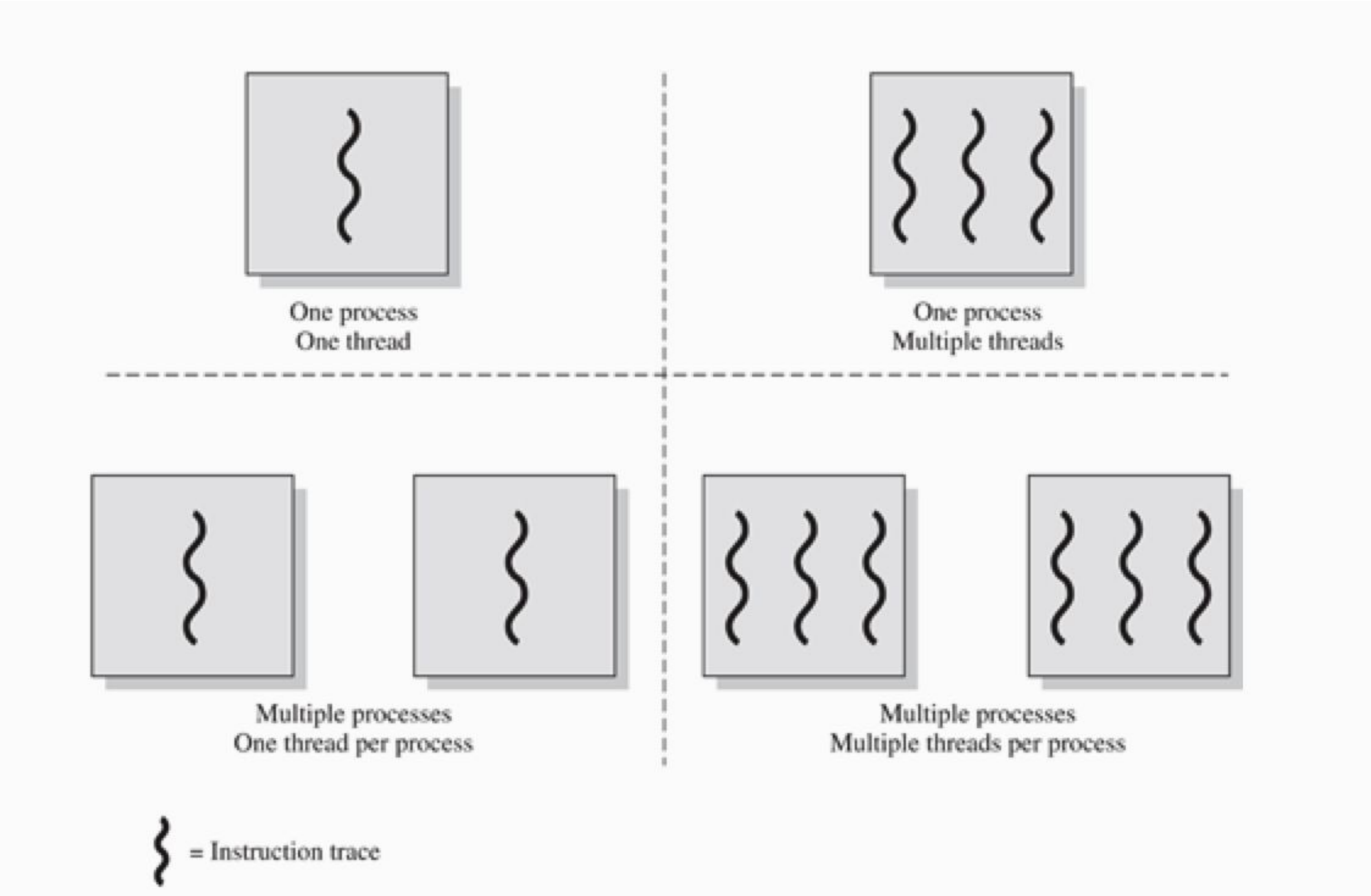
- Address spaces encapsulate protection: Passive Part
- Threads share code, data, files, heap

Thread Abstraction



Single Process P with single thread

Single Process P with three threads



Source: OS book by Stallings

Recall: Why Threads?

- To express a natural program structure
  - Updating the screen, fetching new data, receiving user input — different tasks within the same address space
- To exploit multiple processors
  - Different threads may be mapped to distinct processors
- To maintain responsiveness
  - Splitting commands, spawn threads to do work in the background
- To mask long latency of I/O devices
  - Do useful work while waiting
  - Instead of waiting, the program may do something else
    - CPU can perform other computation, or
- Threads are the natural way to avoid getting stuck
  - Why not processes instead of threads?
    - Threads are light weight: Share data and address space
    - Processes are sound choice when using logically separate tasks

Recall: Why Threads?

- Threads are lighter weight than processes
  - Easier to create and destroy than processes
    - Creating thread is 10-100 times faster than creating a process
    - Specially when the number of threads needed changes rapidly and dynamically,
- Parallel programs must parallelize for performance
- Programs with user interface need threading to ensure responsiveness
- Network and disk bound programs use threading to hide network/disk latency

- A multithreaded program is a generalization of the same basic programming model
  - Each individual thread follows a single sequence of steps (eg loops, call/return from functions, conditions etc)
  - A program can have several such threads in execution at the same time

Recall: Thread

- All threads within a process share
  - Heap
  - Global/static data
  - Libraries
- Each thread has a separate
  - Program Counter
  - Stack
  - registers

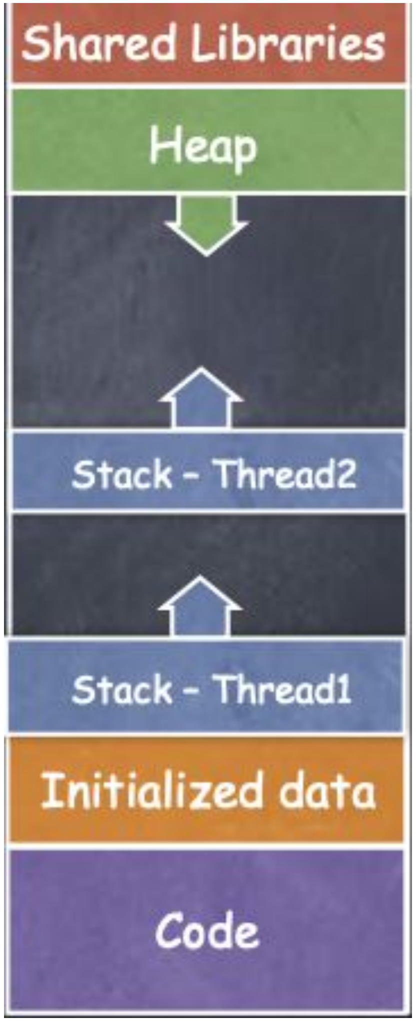
Any stack-allocated variables, parameters, return values, and other things that we put on the stack

Process Address Space

Recall: Threads

- All threads within a process share
  - Heap
  - Global/static data
  - Libraries
- Each thread has a separate
  - Program Counter
  - Stack
  - registers

Process Address Space

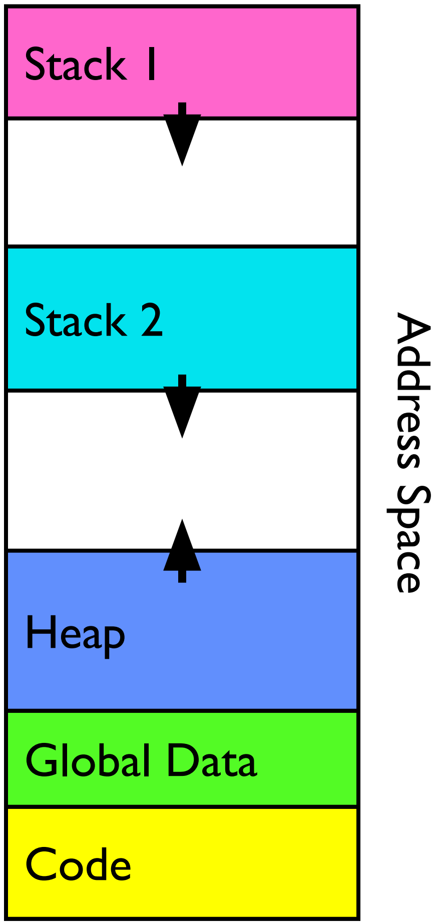


Suspension and termination

- Suspending a process involves suspending all threads of the process since all threads share the same address space
- Termination of a process, terminates all threads within the process

Memory Footprint: Two-Threads

- If we stopped this program and examined it with a debugger, we would see
  - Two sets of CPU registers
  - Two sets of Stacks
- Questions:
  - How do we position stacks relative to each other?
  - What maximum size should we choose for the stacks?
  - What happens if threads violate this?



In this class we will study

- Thread Perspective
- Process Vs Thread
- Thread Life Cycle
- Thread Implementation – Structures
- User level thread - introduction

Thread Perspective

Process Vs Thread

Thread Perspective

- Thread's
  - PC: keeps track of next instruction to be executed
  - Registers: holds current working variables
  - Execution Stack: contains execution history – one frame for each procedure called but not returned

- Multithreading
  - Multiple threads per process
  - When a multithreaded process runs on a single CPU, the threads take turns running
    - Illusion of threads running in parallel (with fast context switching)
- Different threads are not as independent as different processes
  - They share the same address space – share global variables – they can read, write, or even wipe out other thread's stack

Adapted from Tenenbaum's book: Modern Operating Systems

Threads must cooperate

- There is no protection between threads (of a process)
  - Note that all threads belong to one process – no need for privacy – all threads of a process are owned by one user
  - While different processes may belong to different users
- Threads must cooperate with each other (not fight)
- Threads also share
  - the same set of open files
  - Child processes
  - Alarms and signals, and so on

Per-Process Items (process properties)

Address Space

Global Variables

Open files

Pending alarms

Signals and signal handler

Accounting information

Per-thread Items (thread properties)

Program Counter

Registers

Stack

State

- If a thread opens a file, it is visible to all threads. They can read and write it
- With thread, we are trying to achieve
  - Ability for multiple threads to share a set of resources so that they can work together closely to perform some tasks

Process

- Have data/code/heap and other segments
- Include at least one thread
- If a process dies, its resources are reclaimed and its threads die
- Inter-process communication via OS and data copying
- Each process has its own address space; isolated from other processes'
- Each process can run on a different processor
- Expensive creation and context switch

Thread

- No data segment or heap - specific to a thread
- Needs to live in a process
- More than one can be in a process.
- If a thread dies, its stack is reclaimed
- Inter-thread communication via memory
- Have own stack and registers, but no isolation from other threads in the same process
- Each thread can run on a different processor
- Inexpensive creation and context switch

Processes Vs Threads

	Process	**Thread with a process**
Overhead	Expensive to create (fork creates clone of a process)	low
Context Switching	Expensive	Context switch between two threads is low cost – swap basic CPU registers
Virtual Memory	All processes have their won page table	All threads have the same virtual memory
Security	High; Each process had diff address space	Less

Thread Life Cycle

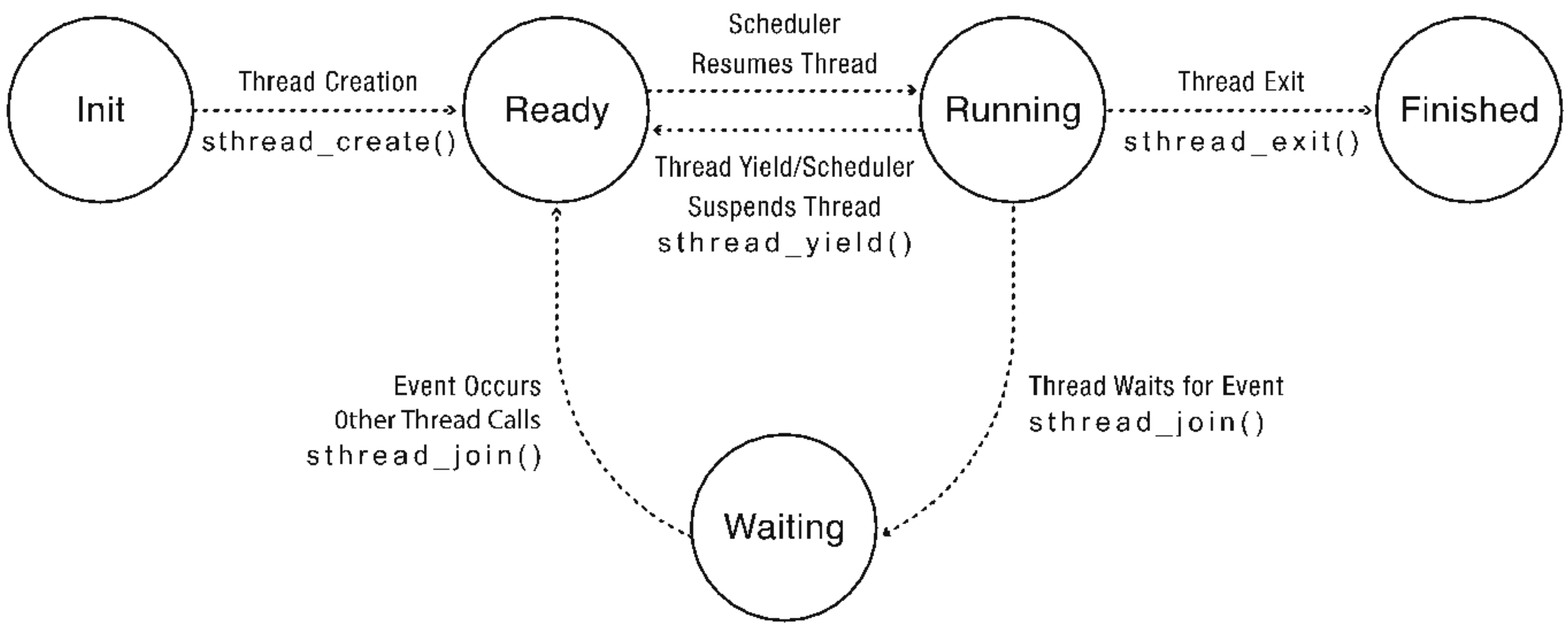
Thread State

- Thread's
  - PC: keeps track of next instruction to be executed
  - Registers: holds current working variables
  - Execution Stack: contains execution history

- State shared by all threads in process/address space
  - Content of memory (global variables, heap)
  - I/O state (file descriptors, network connections, etc)

- State private to each thread
  - Kept in TCB (Thread Control Block)
    - CPU registers (including, program counter)
  - Thread's Execution stack – to keep variables
  - Thread Metadata: Thread id, Owner, scheduling priority

Thread Life Cycle



Thread Life Cycle

- INIT State
  - Initializes and allocates per-thread data structure
- READY State
  - Thread is available in ready list
  - It is available to be run but yet not scheduled
- RUNNING State
  - Transition from READY to RUNNING by copying it's register values from its TCB to the processor's registers
  - Thread is running (executing) on the processor
  - Transitions possible to: READY state, WAITING state, or FINISHED state
    - Will discuss these

Transition from Running State to Ready State

- Preemption by Scheduler
  - OS saves the thread's registers to it's TCB
  - Switching the processor to run the next thread on the ready list
- Voluntarily Relinquishing the Processor
  - By calling `thread_yield` in the thread library

Waiting State and Finished State

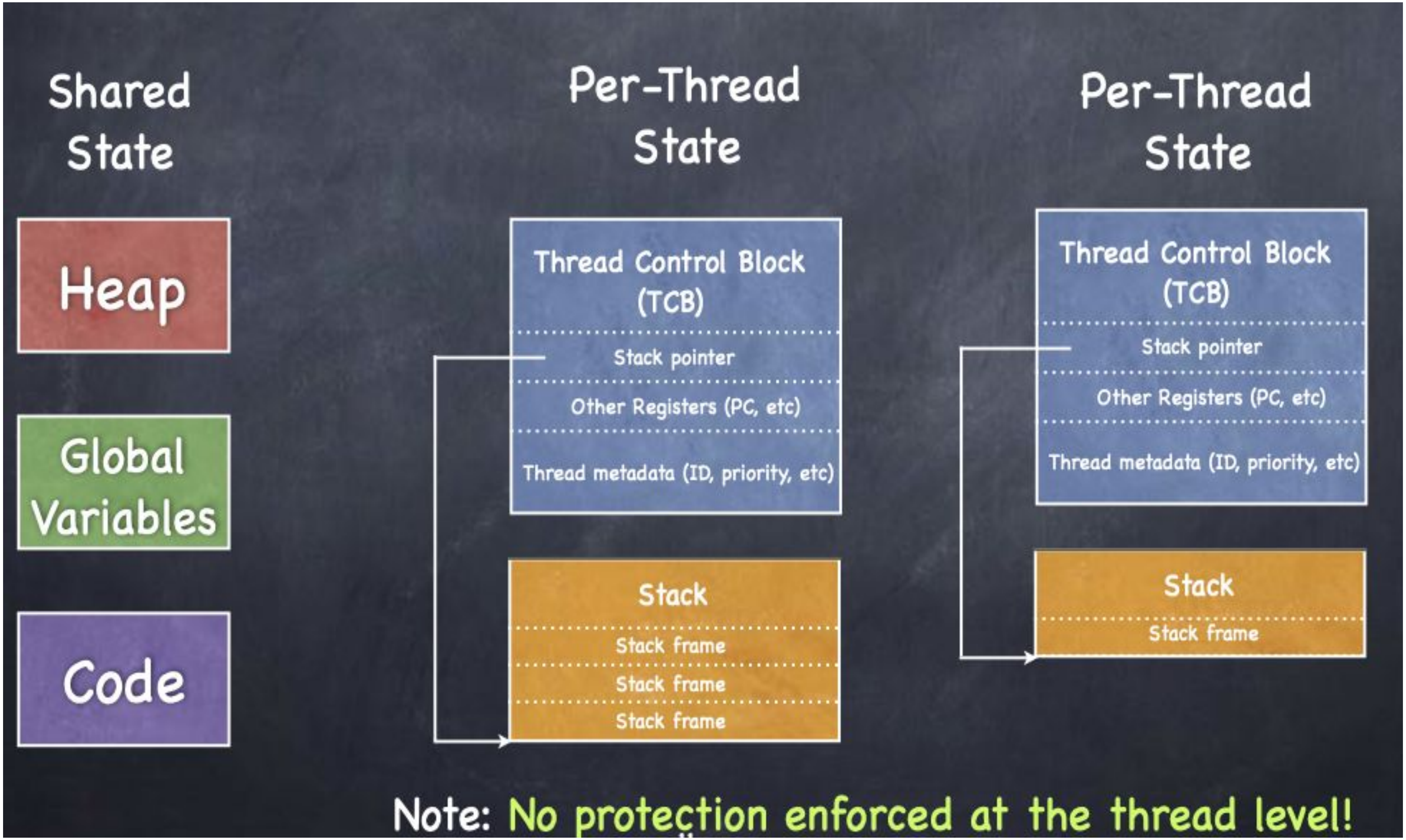
- WAITING State
  - Thread is waiting for some event
  - A thread in WAITING state is placed into READY state by another thread – once the event happens
    - Use of synchronization variable
    - Example: a thread will be in WAITING state by calling `thread_join` for it's child
- FINISHED State
  - A thread in the FINISHED state never runs again
  - System can free up some or all of it's state (registers etc) for other uses
  - OS may keep TCB in finished list for some time
    - When a thread's state is no longer needed (eg after exit value has been read by the join call), it is deleted

Location of thread's per-thread state for different Life cycle stages

State of Thread	Location of Thread Control Block (TCB)	Location of Registers
INIT	Being Created	TCB
READY	Ready List	TCB
RUNNING	Running List	Processor
WAITING	Synchronization Variable's Waiting List	TCB
FINISHED	Finished List then Deleted	TCB or Deleted

Thread Implementation - Structures TCB and Execution Stack

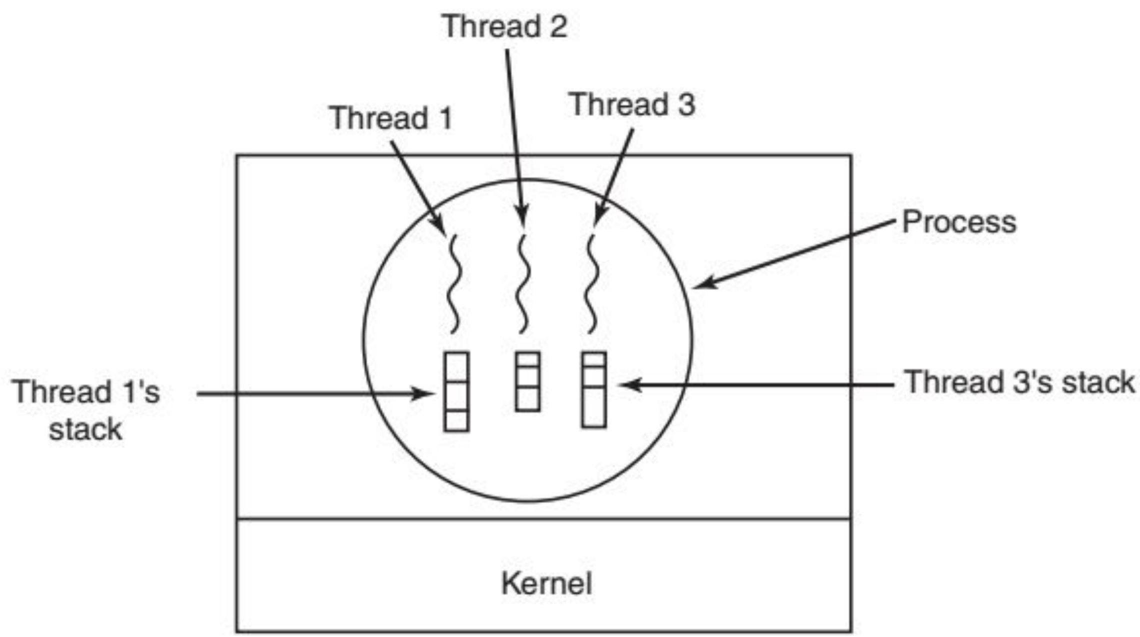
Implementing Thread Abstraction



Thread’s Execution Stack

- It is the same as that for the stack for a single-threaded (process) computation
  - Stores information needed by nested procedures – currently running
  - Example: if a thread calls foo(), foo() calls bar(), and bar() calls bas()
    - The stack will have stack frames for each of these procedures to store local variables and return address
  - Each thread has it’s own stack
- When a new thread is created, the OS allocates a new stack and stores a pointer to that stack in the thread’s TCB
- Each thread generally call different procedures and thus have different execution history

Each thread has its own execution stack



Each thread's execution stack may have different frames corresponding to different procedures called by the thread

Adapted from Tenenbaum's book: Modern Operating Systems

How big is a stack for each thread?

- Stack grows and shrinks
- The size of the stack should be large enough to accommodate the deepest nesting level allowed
- Modern OS allocate
  - Kernel stacks (for each thread)
    - Nesting depth is usually small (in Kernel threads)
    - Linux allocates 8KB stack to each kernel thread
  - User level stacks in Virtual Memory
    - No need for tight space constraints
    - Often 1MB

Backup

Recall - how Execution Stack works?
(from Computer Architecture course)

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

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

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

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:
```

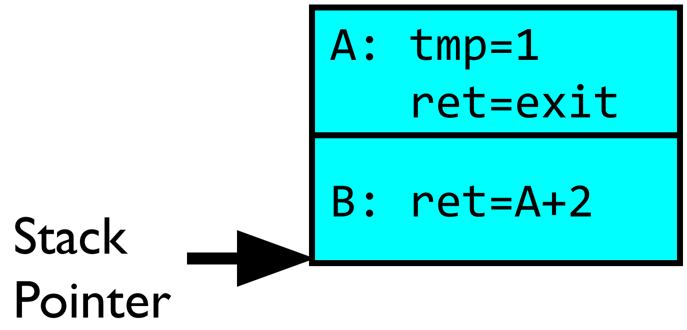
Stack Pointer →

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

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

Execution Stack Example

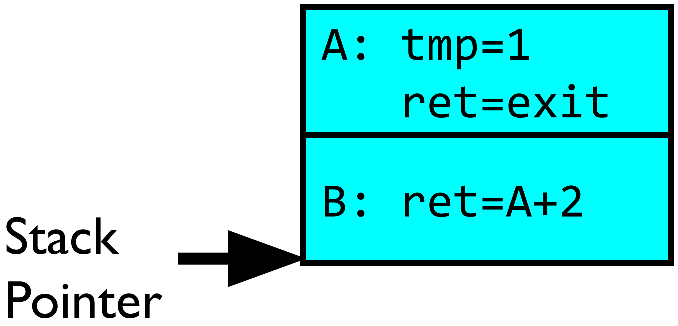
```
       A(int tmp) {
A:       if (tmp<2)
A+1:       B();
A+2:     printf(tmp);
       }
       B() {
B:       C();
B+1:   }
       C() {
C:       A(2);
C+1:   }
       A(1);
exit:
```



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

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

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:
```

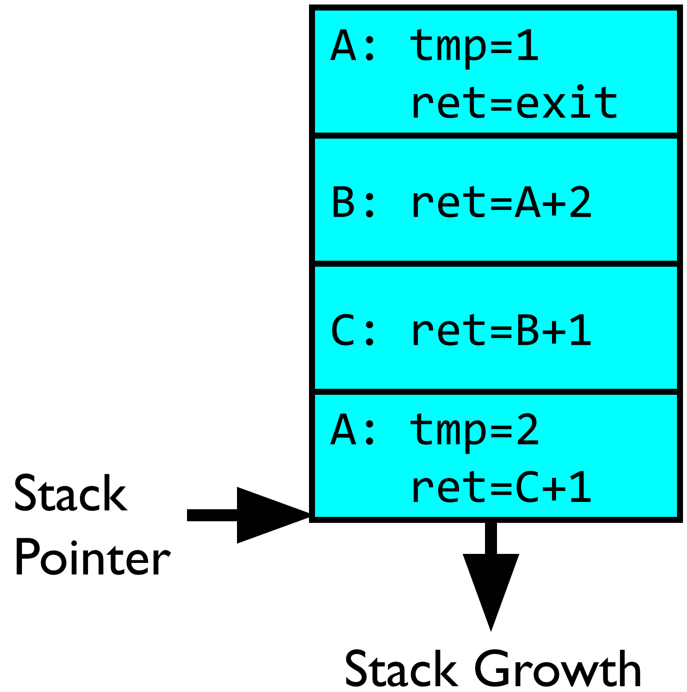
A: tmp=1 ret=exit
B: ret=A+2
C: ret=B+1

Stack Pointer

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

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:
```

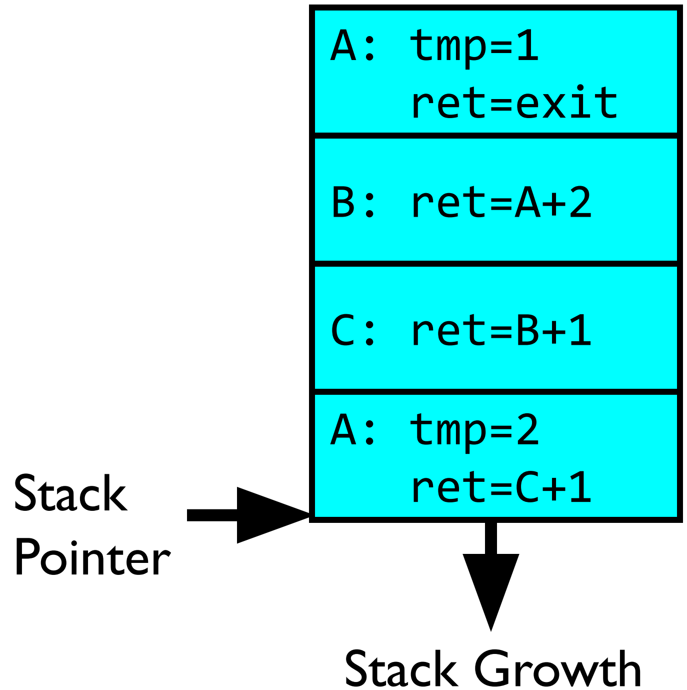


- Stack holds temporary results
- Permits recursive execution
- Crucial to modern languages

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:
```

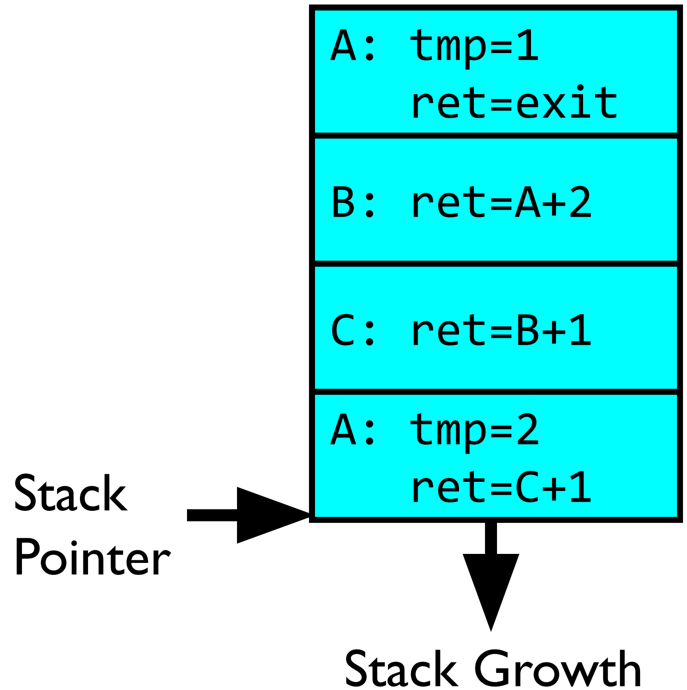


Output: >2

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

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:
```

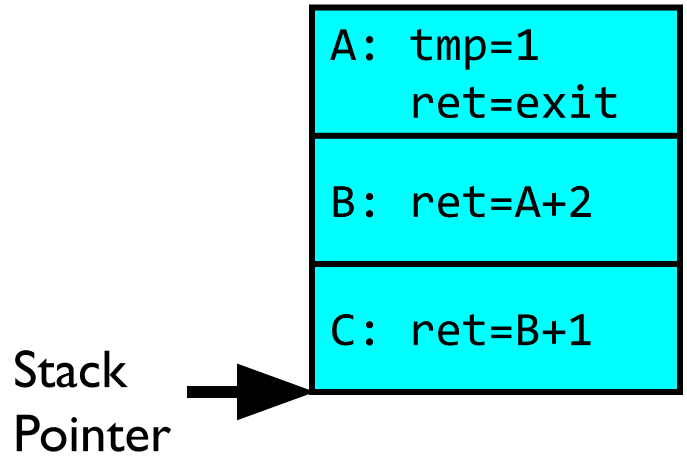


Output: >2

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

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:
```

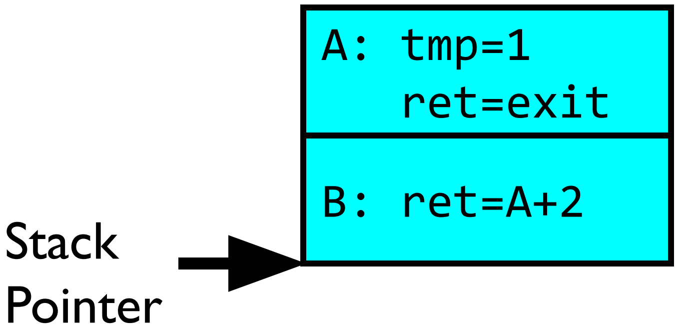


Output: >2

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

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

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

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

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