CPSC 4240/5240: Parallel Programming Techniques

Lecture 7: Loop Scheduling (continued) and Threads and Concurrency

Lecturer: Quanquan C. Liu

Spring 2026

Some slides from Steve Tarzia, EECS-343, Northwestern

OpenMP Loop Scheduling

- OpenMP has more fine-grained control over **loop scheduling**
- **Loop scheduling** determines how iterations of a parallel loop are divided among available threads
- Helps with balancing workloads, reducing latency (idle time), maximize CPU utilization, in some cases, helps with correctness
- #pragma omp for schedule(...)

OpenMP Scheduling: A Quick Aside

- **OpenMP Default Scheduler for loops**
  - Allows compilers to define their default scheduling policy
  - Can vary between different compilers and different versions of the same compiler
  - Static scheduling is the default across most compilers: GCC, Clang, Intel's ICC
  - Loop iterations divided into equal-sized chunks, each thread assigned a specific chunk before loop begins
- Other types of schedulers: **work-stealing** (discussed in later lectures!)

OpenMP Loop Scheduling

- Types of schedules: #pragma omp for schedule(...)
- **Static scheduling**
  - Divides loop iterations into chunks of a fixed size and assigns them to threads in round-robin fashion before loop starts
  - Hence it is static

Default chunk size when no chunk_size passed in

```
#pragma omp for schedule(static)
#pragma omp for schedule(static, chunk_size)
```

OpenMP Loop Scheduling

- Types of schedules: #pragma omp for schedule(...)
- **Static scheduling**
  - Divides loop iterations into chunks of a fixed size and assigns them to threads in round-robin fashion before loop starts
  - Hence it is static

```
#pragma omp for schedule(s
#pragma omp for schedule(s
```

Default chunk size when no chunk_size passed in

Default chunk size is:

$$\frac{Total\ Iters}{Num\ Threads}$$

OpenMP Loop Scheduling

- **Static scheduling**
  - **Pros**:
    - Predictable work distribution: each thread receives roughly equal number of iterations
    - Low overhead for runtime decision-making
    - Better cache performance
  - **Cons**:
    - Potential load imbalance for irregular workloads

Best for regular workloads!

OpenMP Loop Scheduling

- **Static scheduling**
  - **Pros**:
    - Predictable work distribution: each thread receives roughly equal number of iterations
    - Low overhead for runtime decision-making
    - Better cache performance
  - **Cons**:
    - Potential load imbalance for irregular workloads

Best for regular workloads! Typically you don't have to assign the chunk_size yourself

OpenMP Loop Scheduling

- **Static scheduling**
  - **Pros**:
    - Predictable work distribution: each thread receives roughly equal number of iterations
    - Low overhead for runtime decision-making
    - Better cache performance
  - **Cons**:
    - Potential load imbalance for irregular workloads

Best for regular workloads! Typically you don't have to assign the chunk_size yourself

OpenMP Loop Scheduling

- **Dynamic Scheduling:**
  - Divides loop iterations into chunks of a fixed size and assigns them to threads **on-the-fly** as the threads become available
  - Hence it is dynamic

```
#pragma omp for schedule(dynamic)
#pragma omp for schedule(dynamic, chunk_size)
```

OpenMP Loop Scheduling

- **Dynamic Scheduling:**
  - Divides loop iterations into chunks of a fixed size and assigns them to threads **on-the-fly** as the threads become available
  - Hence it is dynamic

```
#pragma omp for schedule(dy
#pragma omp for schedule(dy
```

Default chunk size is determined by compiler for dynamic scheduling

Default for most compilers is 1

OpenMP Loop Scheduling

- **Dynamic Scheduling:**
  - Divides loop iterations into chunks of a fixed size and assigns them to threads **on-the-fly** as the threads become availa
  - Hence

```
#pra
#pra
```

Default chunk size is determined by compiler for dynamic scheduling

Default for most compilers is 1

Why is it different? One is **extreme for uniform workloads** the other **extreme for irregular workloads**

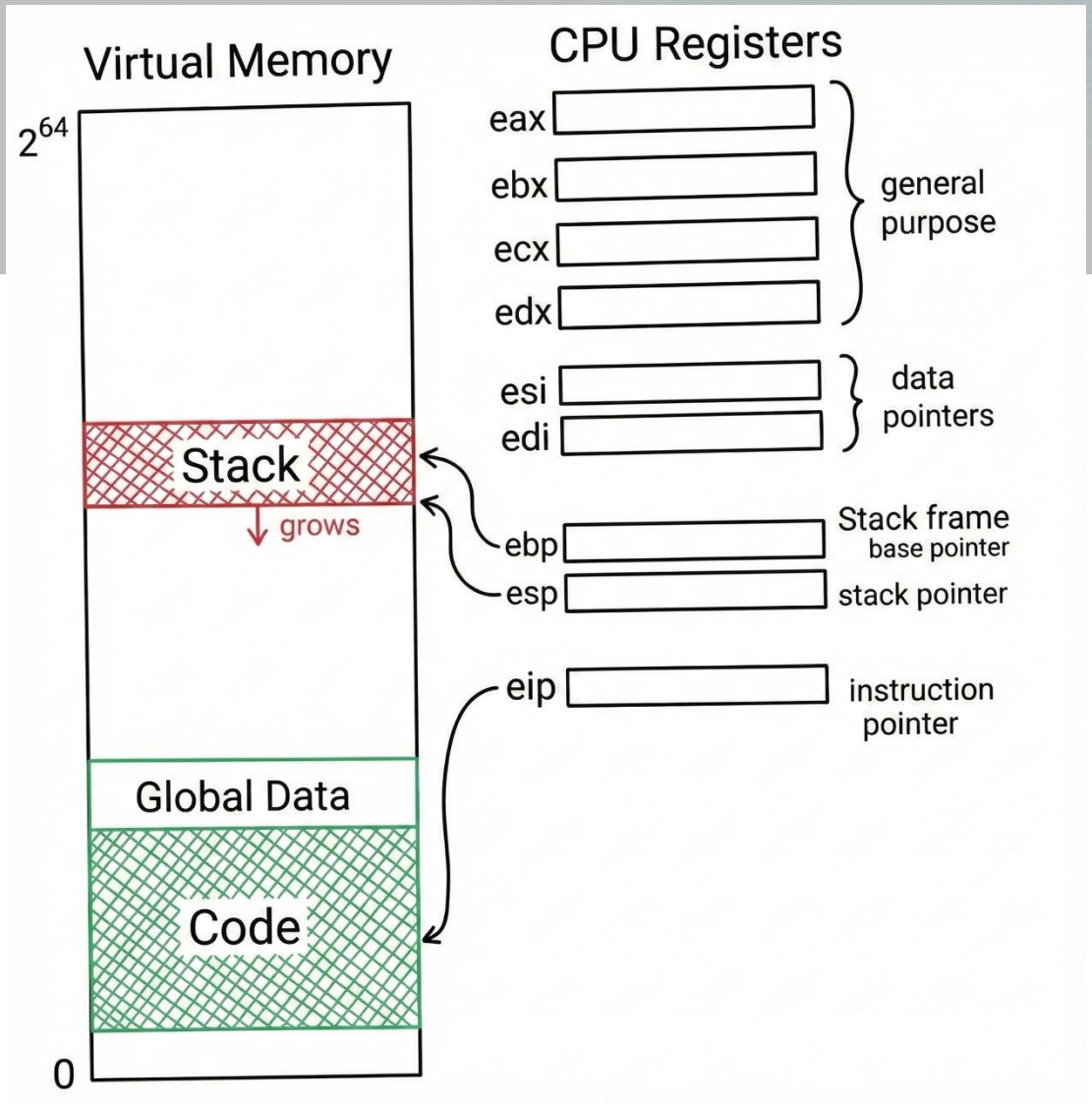
OpenMP Loop Scheduling

- **Dynamic Scheduling:**
  - **Pros**:
    - Better load balancing for irregular workloads
    - Reduced risk of load imbalance
  - **Cons**:
    - Increased overhead
    - Nondeterministic execution order
    - Potential memory overhead for task queues and scheduling structures

Threads and Concurrency

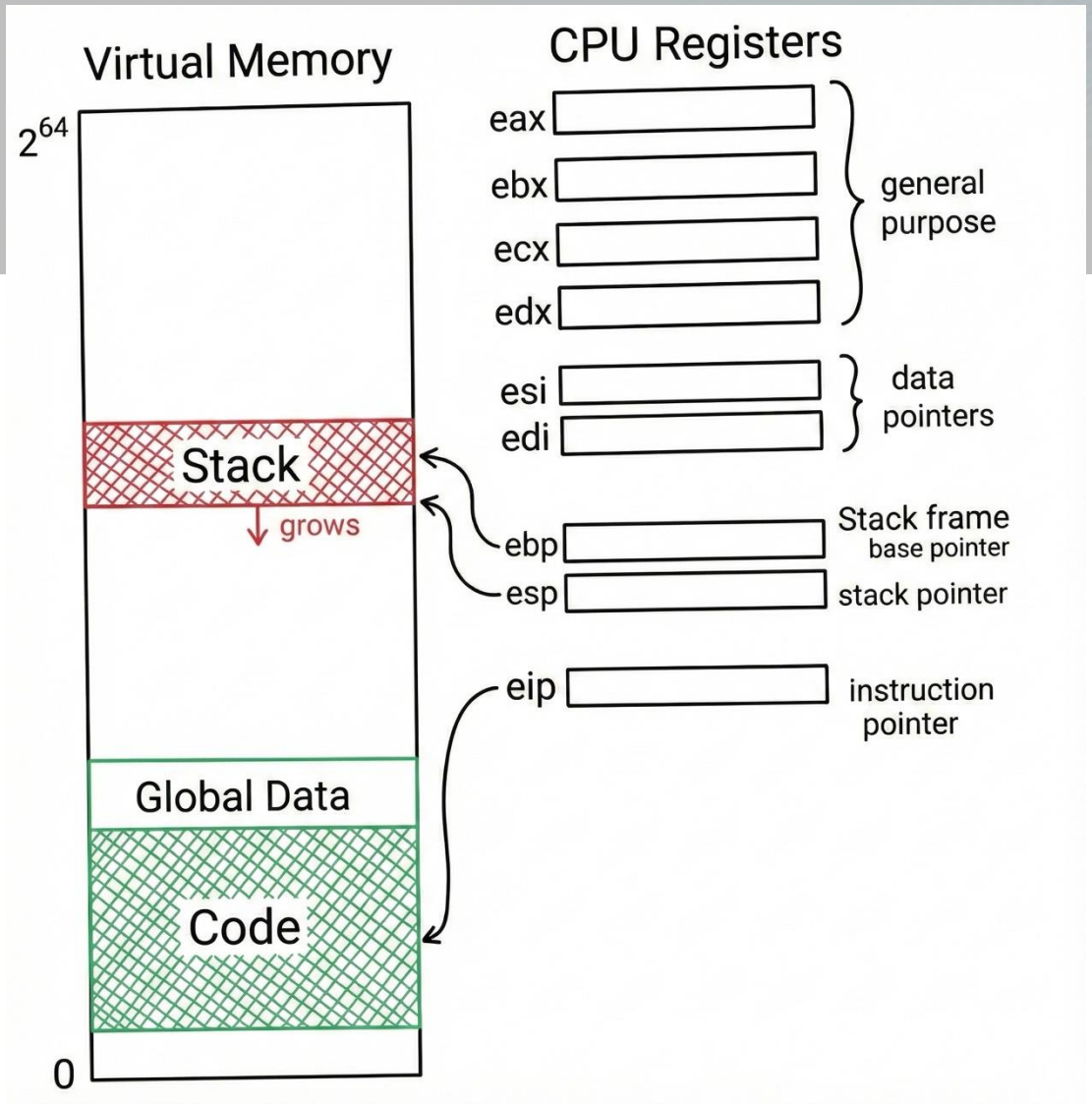
Threads

- So far, we have discussed ***threaded*** processes
- A thread is a part of a process indicating ***where*** it's executing
- OS scheduler actually schedules threads, not processes



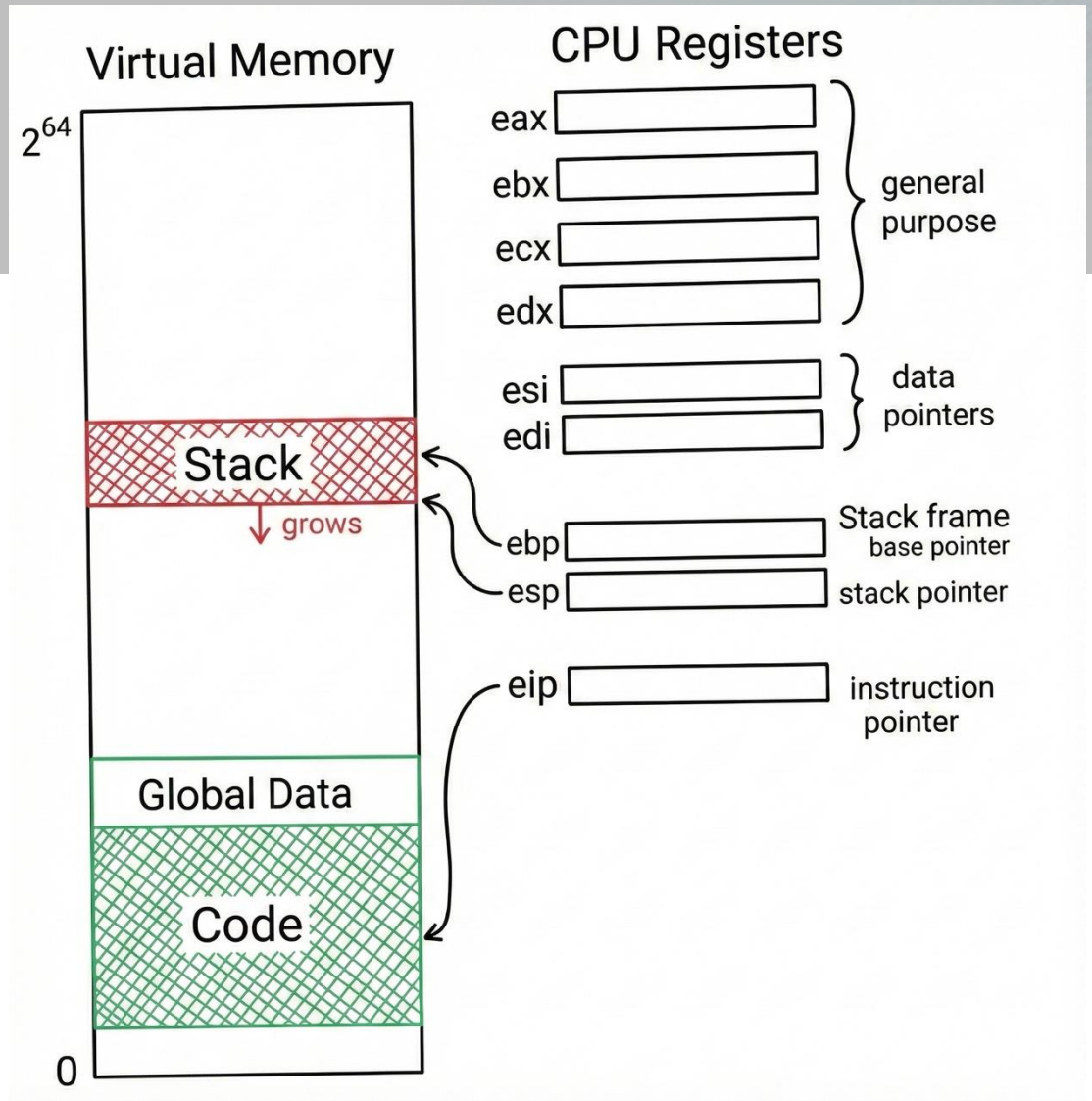
Threads

- Classic memory layout of a **single-threaded process**
  - **The Address Space**: The markings 0 at the bottom and 2^{64} at the top indicate this is a 64-bit address space
  - **Code**: The bottom section contains the actual machine instructions of your compiled program. Read-only memory



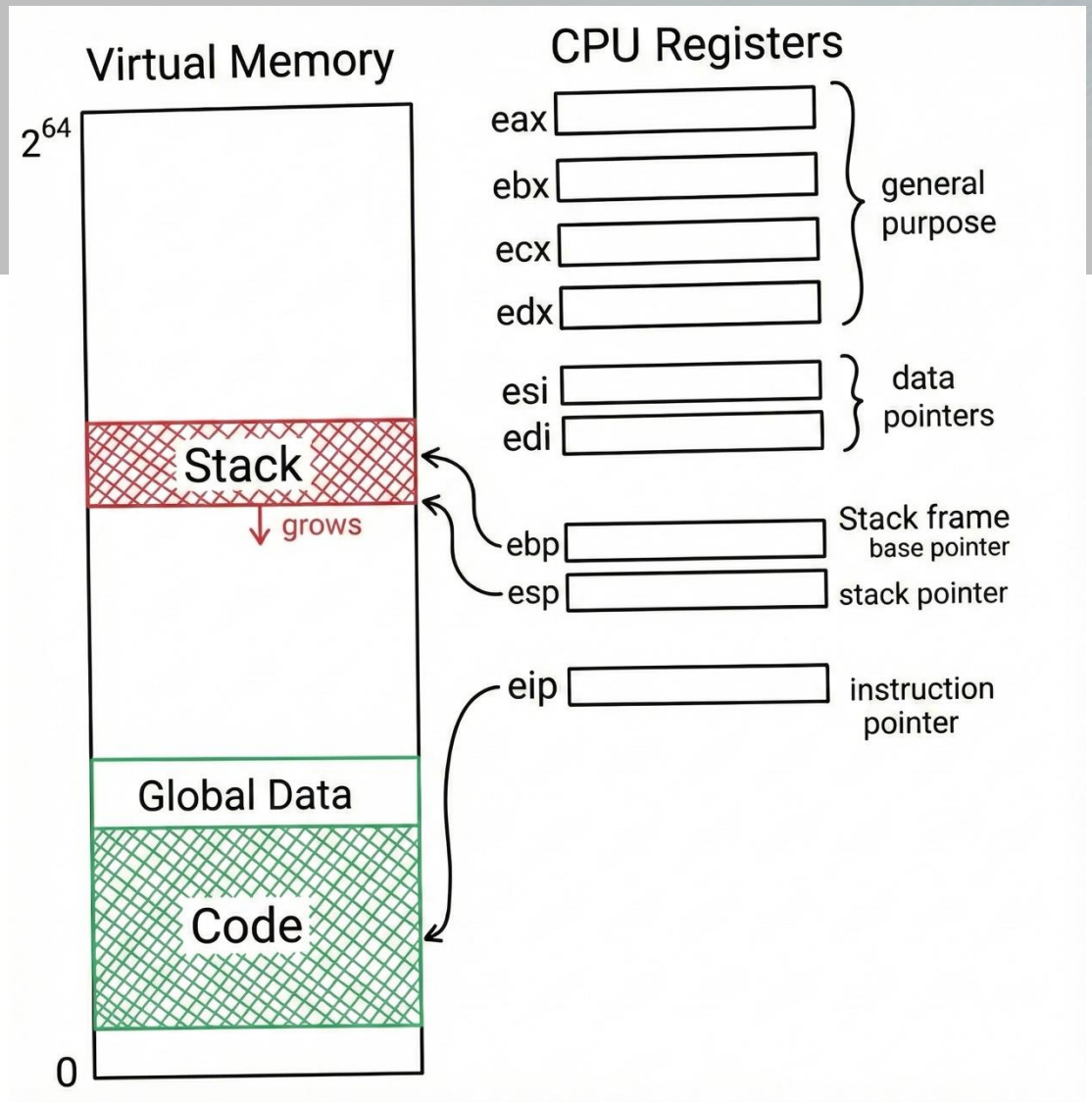
Threads

- Classic memory layout of a **single-threaded process**
  - **Global Data**: Just above the code sits the global and static variables; data that needs to persist for the entire life of the program
  - **The Stack**: This is temporary memory used for function calls, local variables, and return addresses



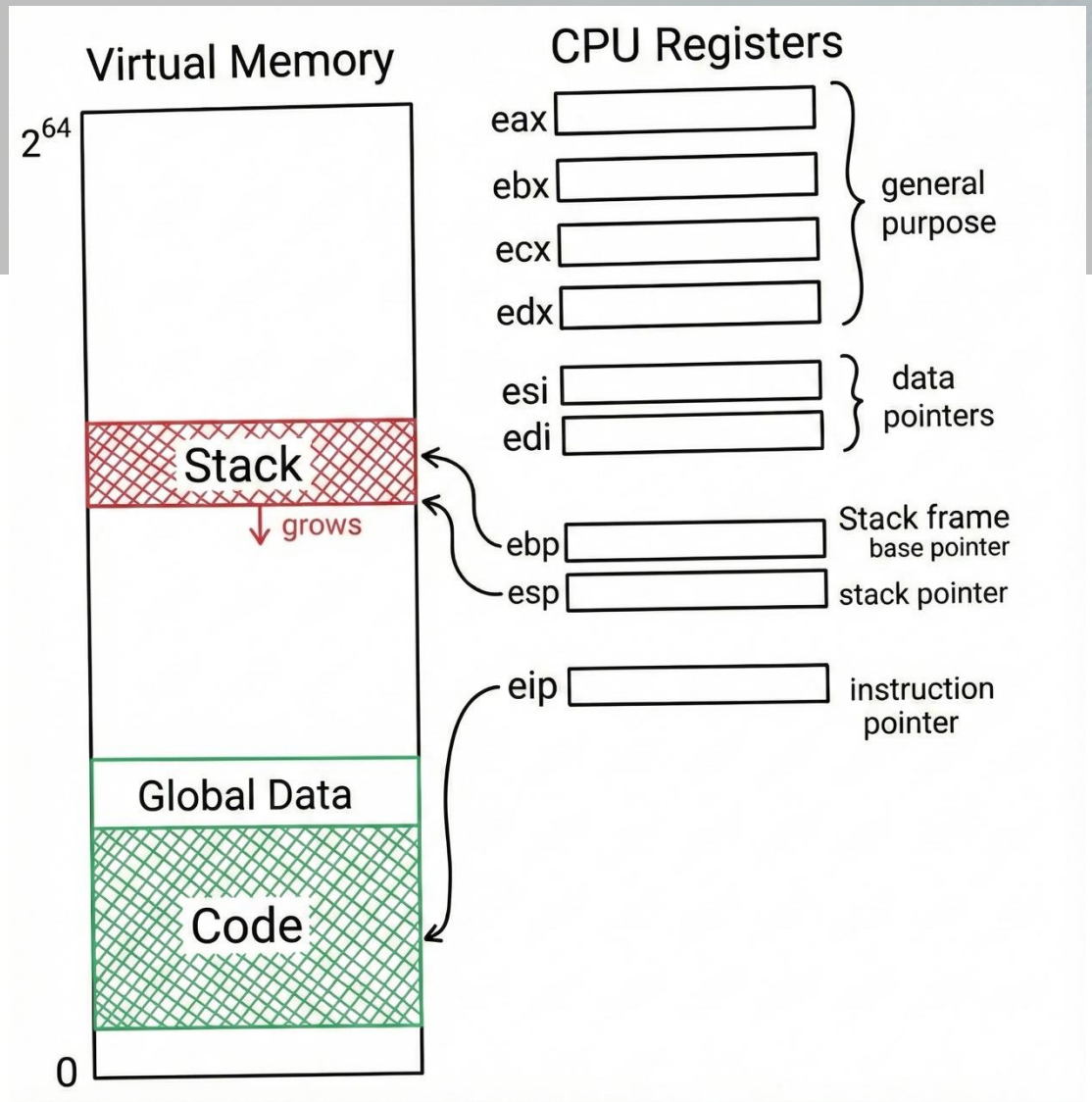
Threads

- Classic memory layout of a **single-threaded process**
  - **Stack "Grows" Down:** The red arrow indicates that as you add data to the stack (push), it expands downwards toward lower memory addresses



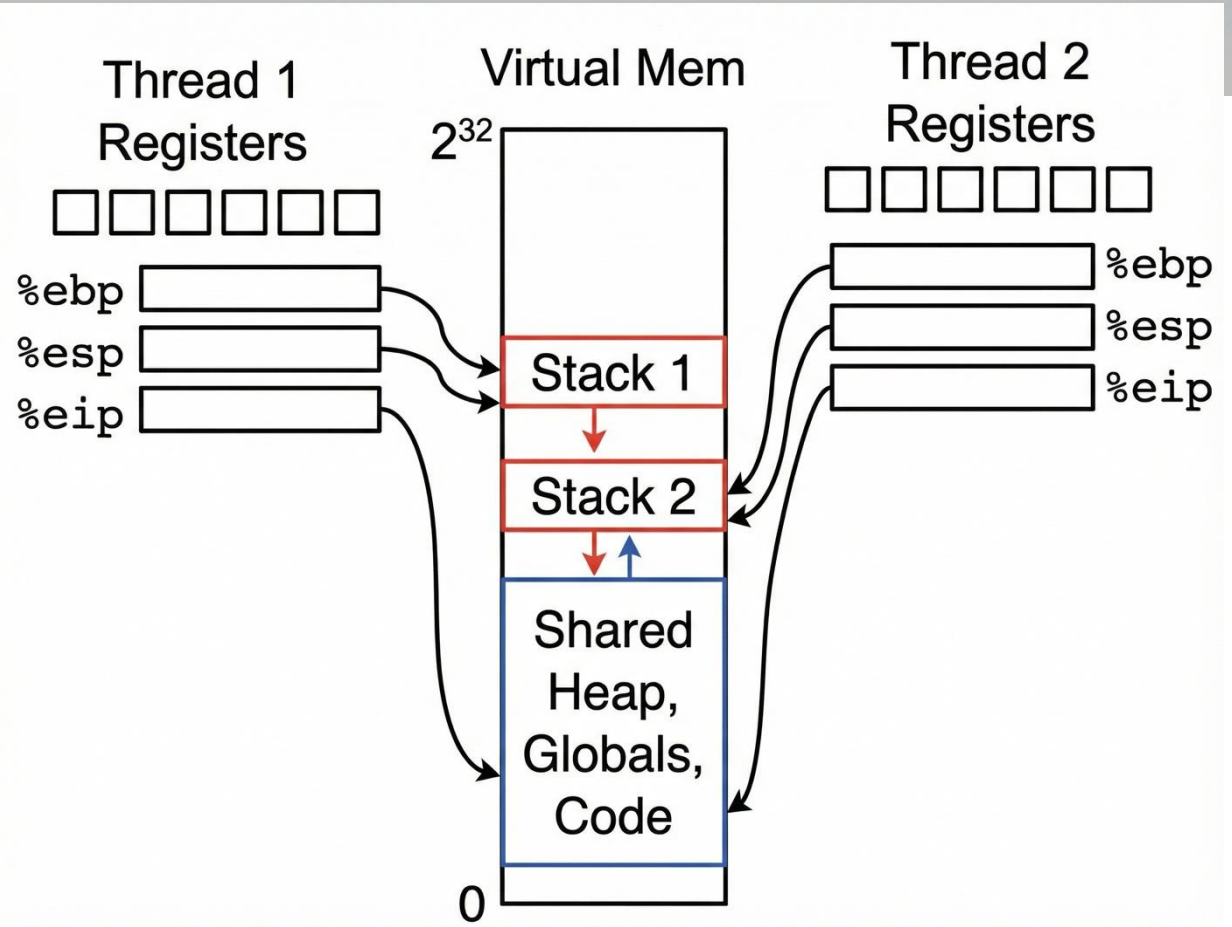
Threads

- **CPU Registers**: Ultra-fast 32-bit
  - **General Purpose**: The CPU's "scratchpad" used for arithmetic, logic, and holding temporary values during calculations
  - **Data Pointers**: Specialized registers often used for efficiently moving data blocks (like copying strings or arrays)
  - **Stack Pointers**:
    - Tracks the very top of the stack
    - Anchors the current function's "frame" (a stable reference point for finding local variables)
- **Instruction Pointer**: The most critical register for control flow. It acts as a cursor, tracking exactly which line of code the CPU is executing



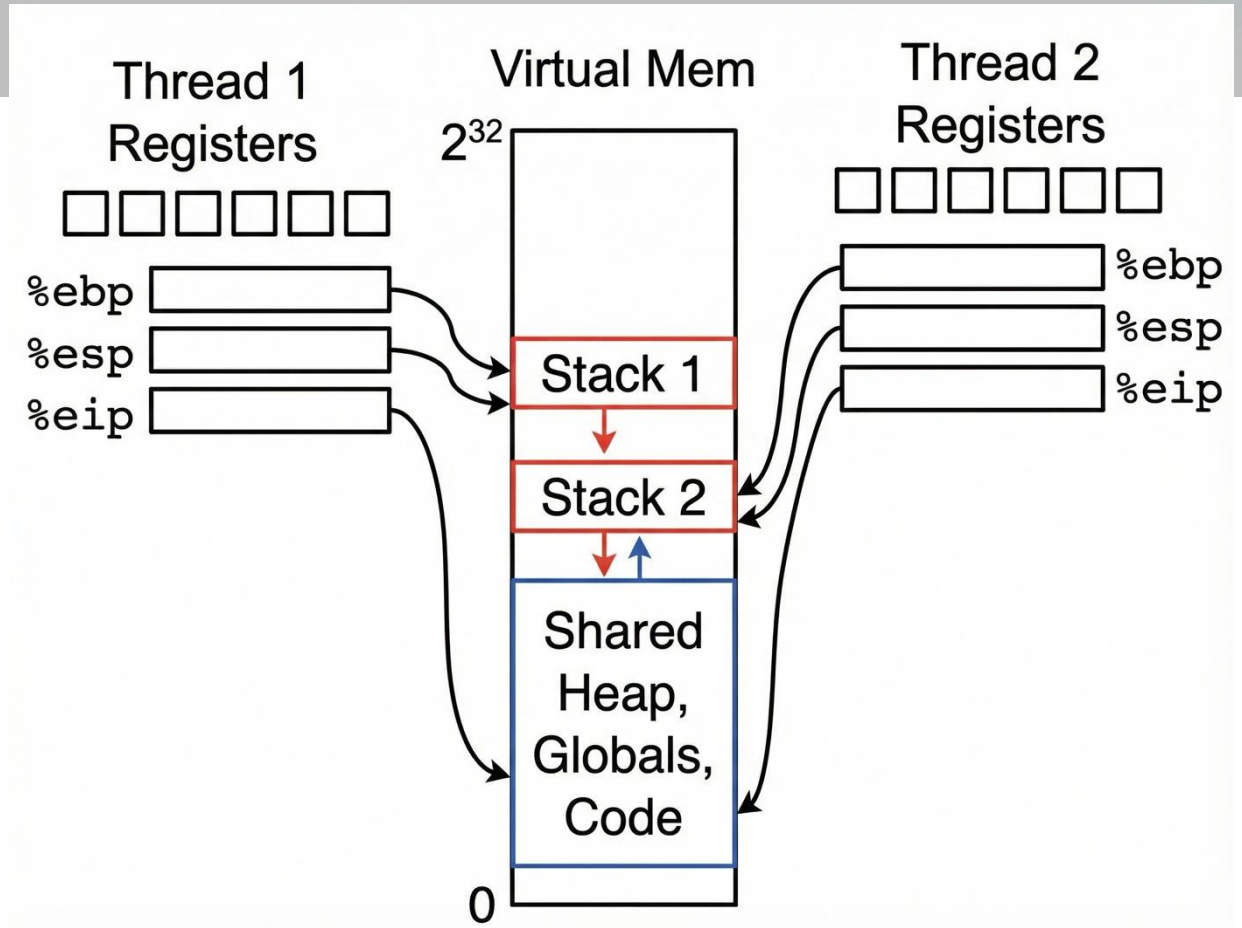
Multi-threaded processes

- A ***multi-threaded*** process can execute in ***parallel***
- A thread is like a process, but it **shares all its virtual memory** with all other threads of the same process
- Each thread has:
  - Its own set of ***register values*** (including the instruction pointer)
  - Its own ***stack***



Multi-threaded processes

- **The "Shared" Territory (Virtual Memory)**
  - **Code & Globals:** All threads execute the same program and access global variables.
  - **Heap:** Dynamically allocated memory is visible to all threads.
- **The "Private" Territory (Per Thread)**
  - **Separate Stacks:** Each thread requires its own stack for local variables and function calls to prevent data corruption.
  - **Independent Execution:** Threads may run the same code but can be at different execution points.
- **The Context Switch (Registers)**
  - **Instruction Pointer (%eip):** Tracks *where* each thread is currently executing.
  - **Stack Pointers (%esp/%ebp):** Anchor the CPU to the specific thread's private stack.
  - *Key Takeaway:* Switching threads means swapping these register values to change the execution context.



Why use threads?

- Allows a process to work in ***parallel*** and use multiple CPU cores to get work done faster
- Allows slow tasks to be done in a ***background*** thread
  - For example:
    - Fetch an image for a website (I/O bound)
    - Save a document to disk (I/O bound)
    - Transcode a media file (CPU bound)
  - This is useful even on machines with a single CPU core
  - For GUI applications, allows main UI thread to be responsive.
    - UI thread will use little CPU time and retain high priority in MLFQ.
    - Disk/network I/O will not block the UI thread
- ***Shared memory*** allows the threads to easily coordinate
  - For example, results can be stored in a global data structure

Why do we need a stack for each thread?

- Remember that the stack stores:
  - Local function variables
  - Function parameters
  - Return addresses
- CPU needs a stack to track its progress through C-style functions
- Each thread takes its own path through the code
  - So, every thread needs its own stack
- A thread ***usually*** should not access another thread's stack
  - The stack is ***thread local*** storage

Stacks, registers and main memory (RAM)

- **Registers**:
  - **Location**: Inside the CPU
  - **Purpose**: Hold the data/instructions that the CPU is directly working on (“active workbench”)
  - **Speed**: Fastest storage the CPU can access (no wait time)
  - **Size**: Very small (only a handful of registers in typical CPUs)

Stacks, registers and main memory (RAM)

- **Stack**
  - **Location**: Part of a process's memory layout in **main memory**, though frequently accessed parts may reside in cache when in use
  - **Purpose**: Primarily used for function call management (function parameters, local variables, return addresses)
  - **Speed**: Accessed via normal memory read/write (with caching, it can be quite fast in practice, but not as fast as direct register access)
  - **Size**: Can grow/shrink within certain limits; much larger than the few CPU registers, but still constrained compared to the entire memory space

Stacks, registers and main memory (RAM)

- **CPU Cache**
  - **Location**: Still on the CPU chip (or very close by), but separate from the registers
  - **Purpose**: Keep recently or frequently used data/instructions from main memory close to the CPU to avoid slow main memory fetches
  - **Speed**: Slower than registers, but faster than main memory
  - **Size**: Larger than register sets but still limited (commonly split into multiple “levels” like L1, L2, L3); (kilobytes to a few megabytes per cache level)

Stacks, registers and main memory (RAM)

- **Main Memory (RAM)**
  - **Location**: Physically separate from the CPU (on memory modules connected to the motherboard).
  - **Purpose**: Stores the operating system, running programs, and data that the CPU can access.
  - **Speed**: Slower to access than cache, but faster than disk.
  - **Size**: Significantly larger than cache (gigabytes).

Stacks, registers and main memory (RAM)

- **Disk Memory (External Storage)**
  - **Secondary (Persistent) Storage**: Data remains stored even when the system is powered off (non-volatile).
  - **Physical Location**: A separate component connected to the system bus (e.g., via SATA or NVMe), not integrated with the CPU.
  - **Long-Term Data**: Used to store the operating system, applications, user files, and other data that must be retained.
  - **Speed**: Slower access compared to RAM and CPU cache, though SSDs are faster than traditional HDDs.
  - **Capacity**: Typically much larger than RAM or CPU cache (often hundreds of gigabytes to terabytes).

Thread creation example

Threads immediately start running on creation

Joins wait for thread to finish running

```
#include <iostream>
#include <thread>

void mythread(const char* arg) {
    std::cout << arg << std::endl;
}

int main() {
    std::cout << "main: begin" << std::endl;

    // Create two threads using std::thread
    std::thread t1(mythread, "A");
    std::thread t2(mythread, "B");

    // Wait for both threads to finish execution
    t1.join();
    t2.join();

    std::cout << "main: end" << std::endl;
    return 0;
}
```

Uses std::thread (introduced in C++11)

Thread creation vs. Fork

- **Thread creation**
- **`std::thread t(thread_func, parameter)`**
  - Creates a thread with **`thread_func, parameter`**
  - ***Shares*** all memory with all threads of the process
  - Scheduled independently of other threads
- **`t.join()`**
  - Waits for a particular thread to finish
- Can communicate by reading/writing (shared) global variables.

Concurrency

- **Definition**: Concurrency involves multiple tasks making progress at the same time.
- **Motivation**: Utilize multi-core and multi-processor systems efficiently.
- Improve throughput and responsiveness.
- **Concurrency vs. Parallelism**:
  - Concurrency: Dealing with lots of things at once conceptually.
  - Parallelism: Physically executing multiple operations simultaneously.

Concurrency vs. Parallelism

- **Concurrency**
  - **Conceptual Focus**: Concurrency is primarily about dealing with multiple tasks (or threads) *in a way that they appear to progress at the same time*.
  - **Example**: On a single-core machine running two threads:
    - **Thread A** might run for 10 ms, then get swapped out.
    - **Thread B** then runs for 10 ms, then gets swapped out.
  - They keep alternating in such small time slices that they *seem* to be running simultaneously (to a human observer).

Concurrency vs. Parallelism

- **Parallelism**
  - **Physical Execution**: Parallelism is about *literally* running multiple operations at the *same time* on different hardware resources
  - **Hardware-Based**: To achieve parallelism, you need an actual multicore or multiprocessor system, or specialized hardware (like GPU cores). This ensures multiple instructions are executed *truly simultaneously*.
  - **Example**: On a quad-core processor running four threads:
    - Each of the four threads can be assigned to a different core.
    - All four threads can *actually execute* instructions at the same moment in time.

Challenges in Concurrency

- **Race Conditions**: Multiple threads accessing/modifying shared data without proper synchronization.
- **Deadlocks**: Two or more threads wait forever for each other's locks.
- **Livelocks**: Threads continually react to each other without progressing.
- **Starvation**: Some threads never get enough CPU or resources to make progress.
- **Data Inconsistency**: Interleaving of operations leads to incorrect or partial updates.

How to Solve These Challenges

- **Synchronization Techniques**
  - **Mutexes (Mutual Exclusion Locks)**:
    - Ensures only one thread accesses critical section at a time.
  - **Semaphores**:
    - Can allow multiple threads up to a limit to access a shared resource.
  - **Spinlocks**:
    - Busy waiting locks suitable for short lock durations.
  - **Monitors / Condition Variables**:
    - High-level constructs for safe access and signaling between threads.
  - **Atomic Operations**:
    - Hardware-level support for indivisible updates to shared data.

Concurrency in Practice

- **Programming Languages**: C/C++: Pthreads, std::thread, std::mutex, and atomic library.
- Java: java.util.concurrent package (Executors, Locks, etc.).
- Python: threading, multiprocessing (noting GIL limitations).
- .NET (C#): Task, async/await, and Parallel classes.
- **Frameworks**: OpenMP: Compiler directives for shared-memory concurrency.
- MPI: Message-passing for distributed memory systems.

Concurrency Design Patterns

- **Producer-Consumer**: One or more producers generate data, consumers process it.
  - Typically uses a blocking queue or buffer.
- **Pipeline**: Data flows through multiple stages, each in its own thread.
- **Fork-Join**: Tasks split (fork) into sub-tasks and then (join) to merge results.
- **Map-Reduce**: Map phase transforms data; reduce phase aggregates results (common in big data).

Debugging and Testing Concurrent Programs

- **Logging**: Detailed timestamps and thread identifiers.
- **Tools**: Race condition detectors (e.g., ThreadSanitizer).
  - Performance profilers (to detect hotspots and contention).
- **Testing Strategies**: Stress testing and randomization.
  - Model checking or formal verification (for safety-critical systems).

Concurrency Best Practices

- **Minimize Shared State**: Use immutable objects or local variables whenever possible.
- **Favor High-Level Abstractions**: Let frameworks handle lower-level synchronization details (e.g. OpenDP).
- **Plan for Scalability**: Make sure concurrency solutions scale as cores/processors increase.
- **Design for Fault Tolerance**: Graceful recovery if part of the system fails or times out.

Concurrency can create tricky problems

```
#include <iostream>
#include <thread>
#include <atomic>

int counter = 0;
static const int LOOPS = 1e7;

void mythread(const char* arg) {
    std::cout << arg << ": begin\n";
    for (int i = 0; i < LOOPS; i++) {
        counter++; // Unsafe increment
    }
    std::cout << arg << ": done\n";
}
```

```
int main() {
    std::cout << "main: begin (counter = " << counter << ")\n";

    // Create two threads
    std::thread t1(mythread, "A");
    std::thread t2(mythread, "B");

    // Wait for threads to finish
    t1.join();
    t2.join();

    std::cout << "main: done with both (counter = " << counter
              << ", goal was " << 2 * LOOPS << ")\n";

    return 0;
}
```

- Start two threads, each of which increments a shared global **counter** variable 10^7 times.

Concurrency can create tricky problems

```
#include <iostream>
#include <thread>
#include <atomic>

int counter = 0;
static const int LOOPS = 1e7;

void mythread(const char* arg) {
    std::cout << arg << ": begin\n";
    for (int i = 0; i < LOOPS; i++) {
        counter++; // Unsafe increment
    }
    std::cout << arg << ": done\n";
}
```

```
int main() {
    std::cout << "main: begin (counter = " << counter << ")\n";

    // Create two threads
    std::thread t1(mythread, "A");
    std::thread t2(mythread, "B");

    // Wait for threads to finish
    t1.join();
    t2.join();

    std::cout << "main: done with both (counter = " << counter
              << ", goal was " << 2 * LOOPS << ")\n";

    return 0;
}
```

Output: main: done with both (counter = 10286823, goal was 20000000)

- Start two threads, each of which increments a shared global **counter** variable 10^7 times.

What's the problem?

- You have to understand the low-level behavior to find the problem.
  - In short, the "counter++" operation is not ***atomic***.

Output: main: done with both (counter = 10286823, goal was 20000000)