Computação em Larga Escala

Concurrency

Eurico Pedrosa
António Rui Borges

Universidade de Aveiro - DETI

2025-02-22

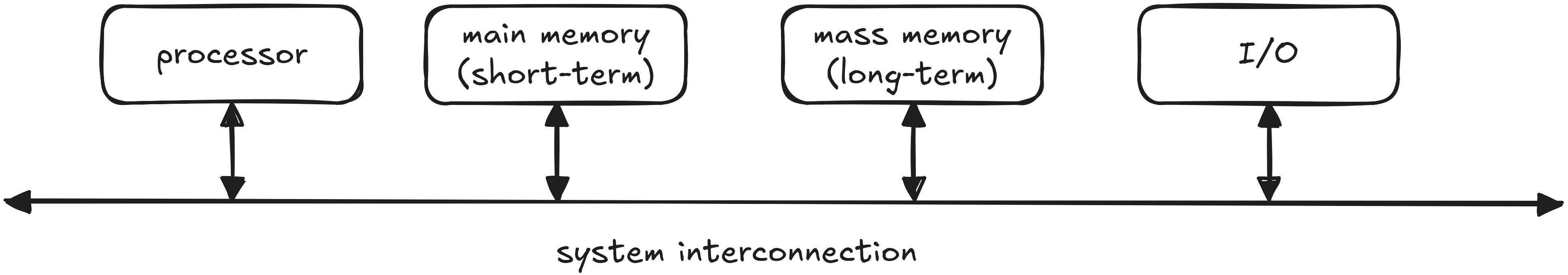
Summary

- Computer Architecture
- Program vs. Process
- Processes vs. Threads
- Suggested Reading

Computer Architecture

Overview

- Top level overview of a computer architecture



- **Processor (CPU - Central Processing Unit):** Controls the computer's operations and executes data processing tasks.
- **Main Memory:** Temporarily stores data during processing; it is volatile.

Overview

- **Mass Storage:** Retains data between processing runs, enabling large-scale data retrieval and updates; it is non-volatile and functions as a specialized I/O device.
- **Input/Output (I/O):** Manages data transfer between the computer and external devices.
- **System Interconnect:** Facilitates communication among components, typically implemented as a bus.

To execute a specific task, a computer must be provided with a set of instructions, collectively forming a **program**. A fundamental question arises: **how should instructions be represented?**

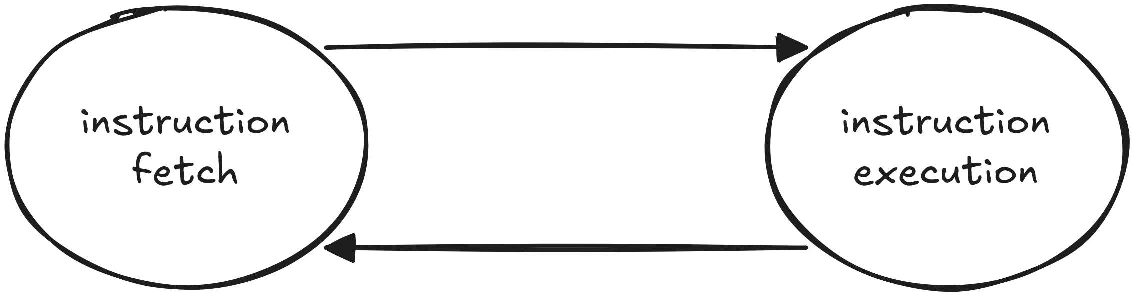
Overview

A key innovation was the idea of representing instructions in a format that could be **stored in main memory** alongside data. This effectively makes instructions a special kind of data. With this approach, a computer can **fetch and execute instructions directly from memory**, allowing programs to be loaded, modified, and executed dynamically.

This concept, independently developed by **John von Neumann** and **Alan Turing**, is known as the **stored-program architecture**. It has become the foundation of modern computing, leading to systems commonly referred to as **von Neumann machines**.

Overview

Despite its complexity, a computer system can be understood as a digital system that continuously alternates between two fundamental internal states:

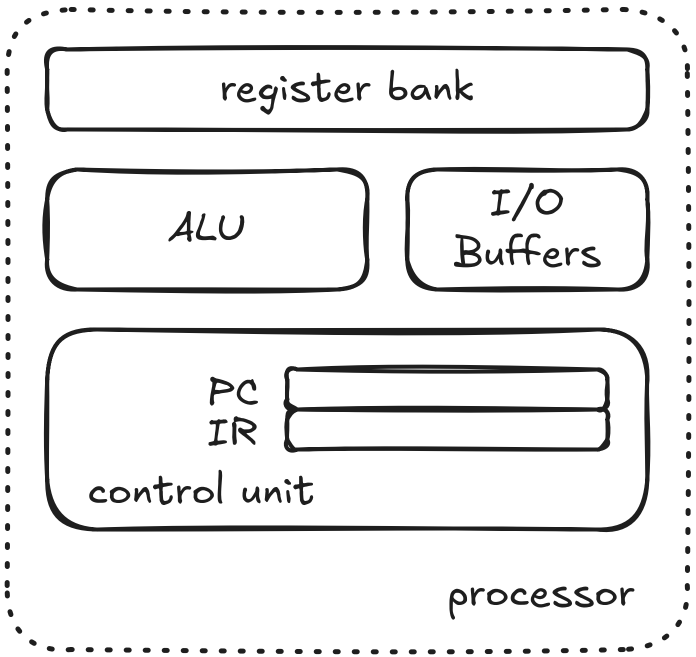


1. **Instruction Fetch:** The processor retrieves the next instruction from memory.
2. **Instruction Execution:** The processor decodes the fetched instruction and carries out the corresponding operation.

This cyclical process forms the foundation of a computer's operation, enabling it to execute programs systematically.

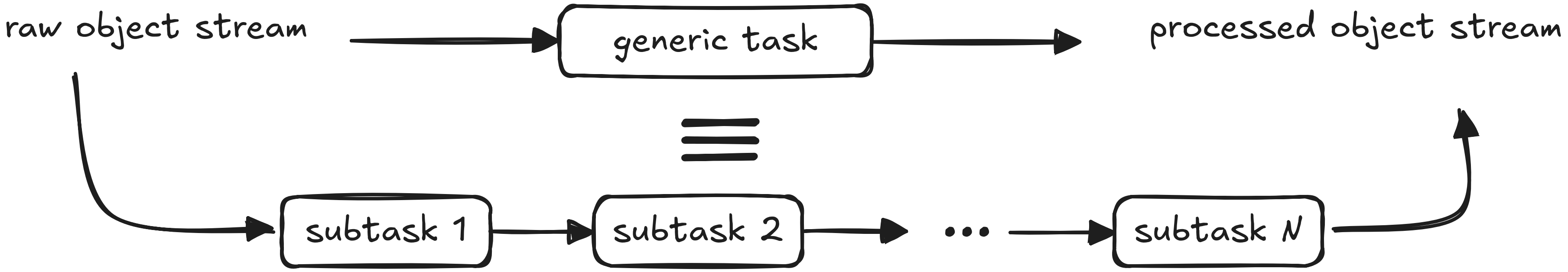
Overview

In a simplified manner, the processor can be considered as comprising: a **control unit**, which primarily handles the instruction fetch and decoding phases; an **arithmetic/logic unit (ALU)**, responsible for executing the prescribed operations; a **register bank**, which stores temporary data; and **I/O buffers**, which facilitate communication with other components of the computer system.

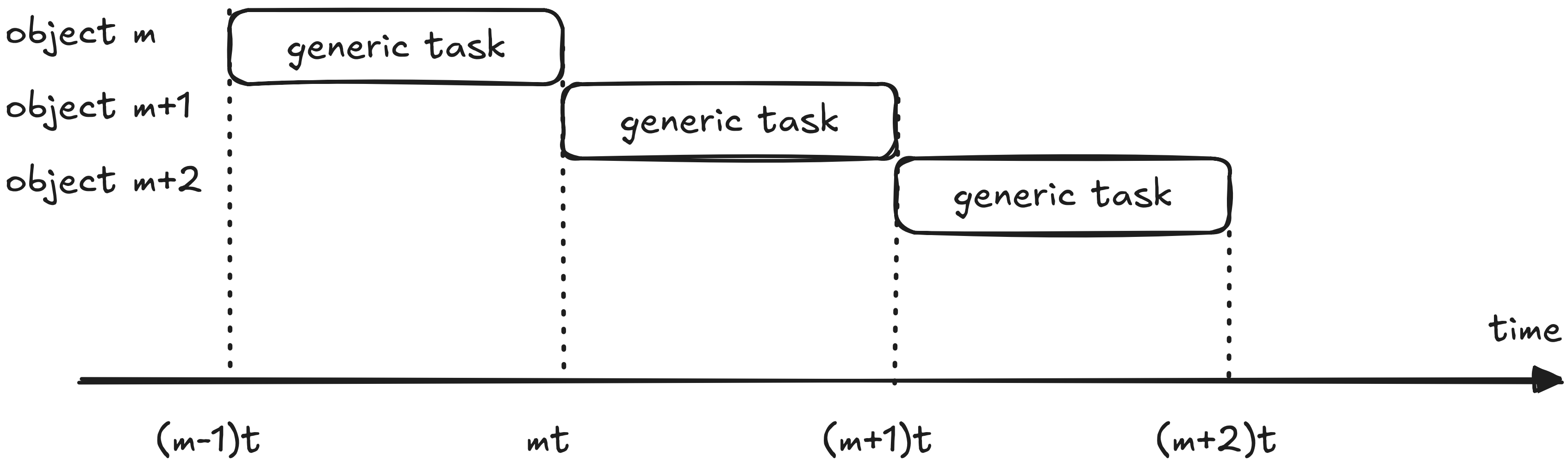


Overview

Pipelining is an implementation technique that transforms the execution of a generic task on a stream of objects into a sequence of independent subtasks, which operate simultaneously on successive objects in the stream. Each subtask, known as a **pipeline stage** or **segment**, is executed in sequence and represents a specific portion of the overall task. The combined, ordered execution of these stages is functionally equivalent to performing the original task on each object in the stream.

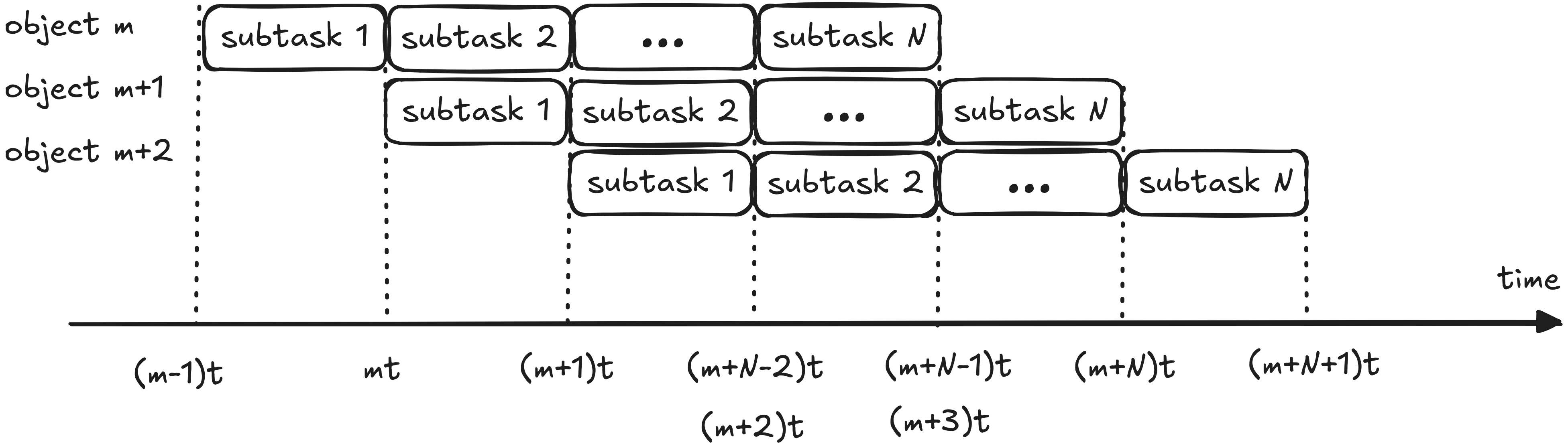


Non-pipelined version



- Throughput = $\frac{1}{t}$
- Execution time for a m object stream = mt

N-stage pipelined version



- Execution time for subtask n: t_n, with $n = 1, 2, ..., N$
- Throughput = $\frac{1}{t}$, where $t = \max(t_1, t_2, ..., t_N)$
- Execution time for a m object stream = $(N - 1 + M) \cdot t$

Overview

The speedup obtained from executing a task using an **N-stage pipeline** compared to a **non-pipelined execution** is given by the formula:

$$S = \frac{t_{\text{non-pipelined}}}{t_{\text{pipelined}}}$$

For a generic task, the execution time in a **non-pipelined** version is:

$$t_{\text{non-pipelined}} = m \cdot t_{\text{cycle}}$$

where t_{cycle} is the clock cycle time, and m is the number of objects stream.

Overview

In a **pipelined** version, once the pipeline is filled, a new subtask completes execution every cycle, meaning:

$$t_{\text{pipelined}} = (m + (N - 1)) \cdot t_{\text{cycle}}$$

where N is the total number of subtasks.

For a large number of subtasks $(N \gg m)$, the speedup approaches:

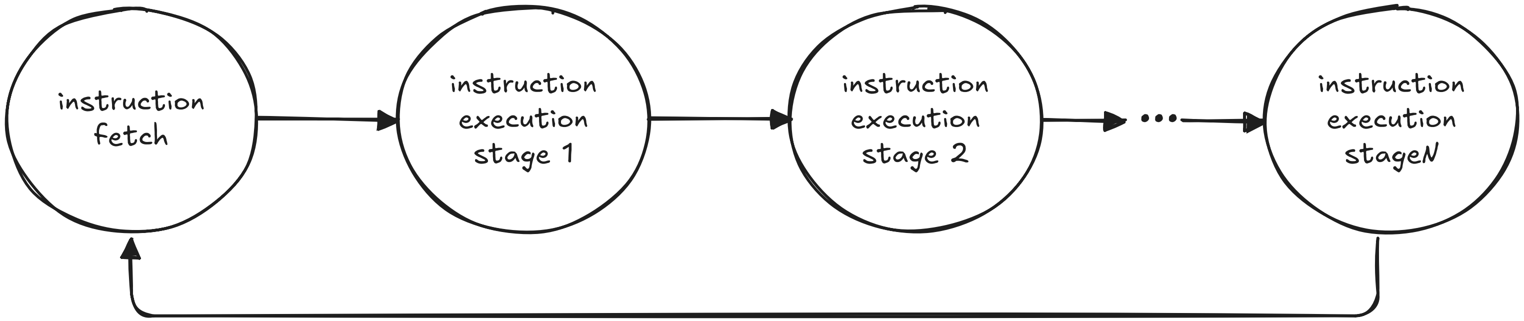
$$S \approx m$$

which means an **N-stage pipeline ideally provides an N-fold speedup** over a non-pipelined version, assuming perfect conditions.

Overview

Since 1985, all processors have incorporated **pipelining** as a technique to overlap instruction execution and enhance performance. This **overlapping of instruction execution** is known as **Instruction-Level Parallelism (ILP)** because it allows multiple instructions to be processed concurrently through the decomposition of their execution into independent stages.

A common approach to achieving this is by **decomposing the instruction execution phase** of the processor cycle into multiple **stages**, allowing different parts of multiple instructions to be processed simultaneously.



Overview

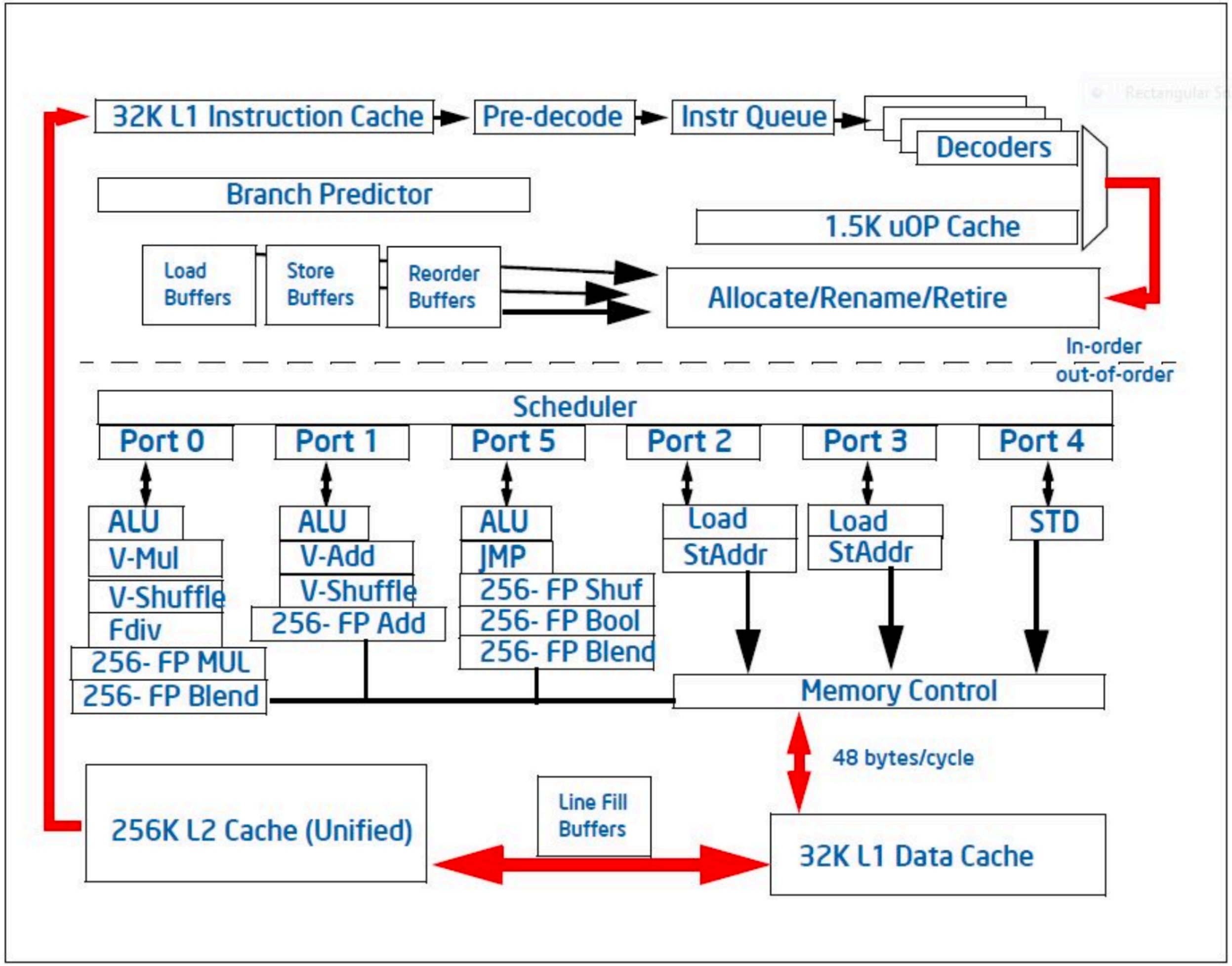


Figure 2-1. Intel microarchitecture code name Sandy Bridge Pipeline Functionality

Overview

Instruction-Level Parallelism (ILP) is facilitated by several key mechanisms that enable overlapping and parallel execution of instructions:

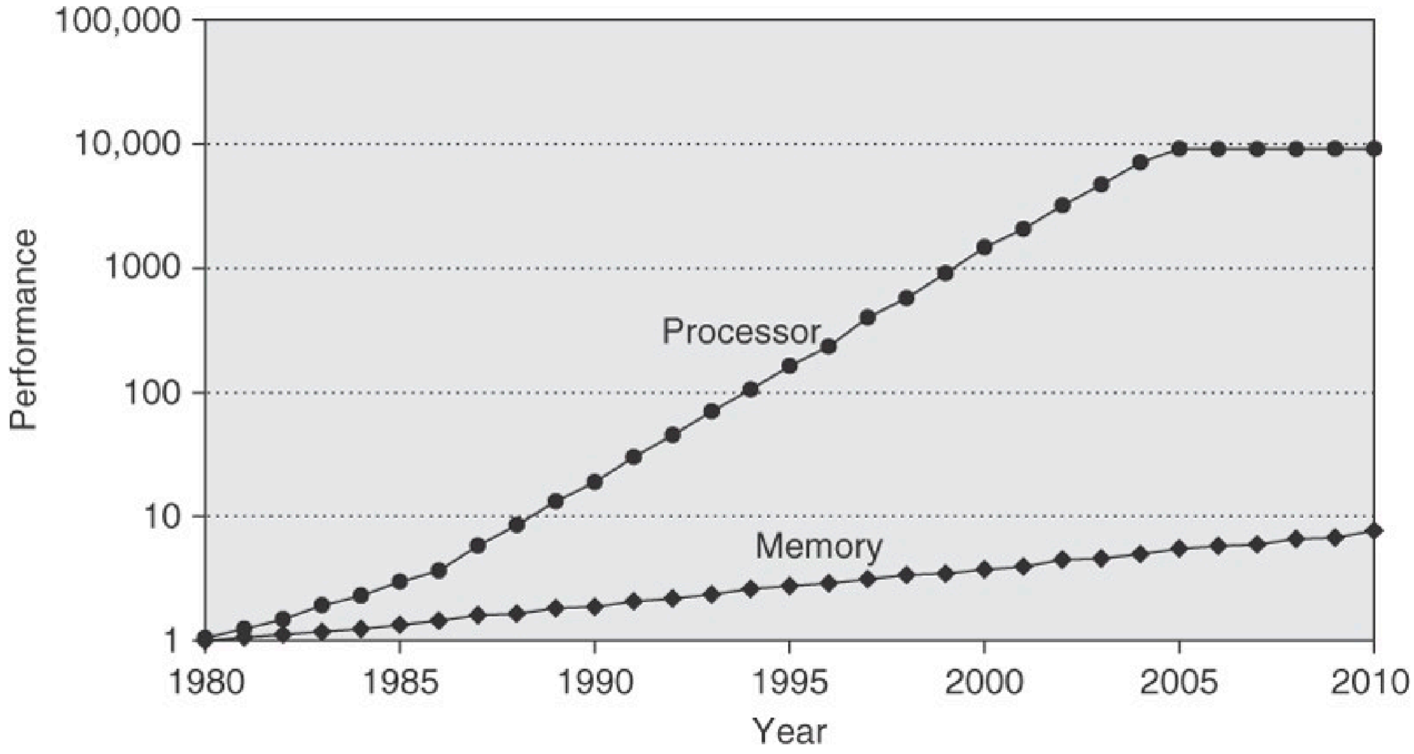
- **Multiple-issue** – Independent instructions can be initiated simultaneously, increasing throughput.
- **Pipelining** – Arithmetic units process multiple operations concurrently at different stages of execution.
- **Branch prediction and speculative execution** – The processor anticipates the outcome of conditional instructions and executes them speculatively to reduce stalls.
- **Out-of-order execution** – Instructions can be dynamically rearranged for optimal efficiency, provided there are no dependencies.
- **Prefetching** – Data is retrieved speculatively before an instruction explicitly requests it, improving memory access latency.

Overview

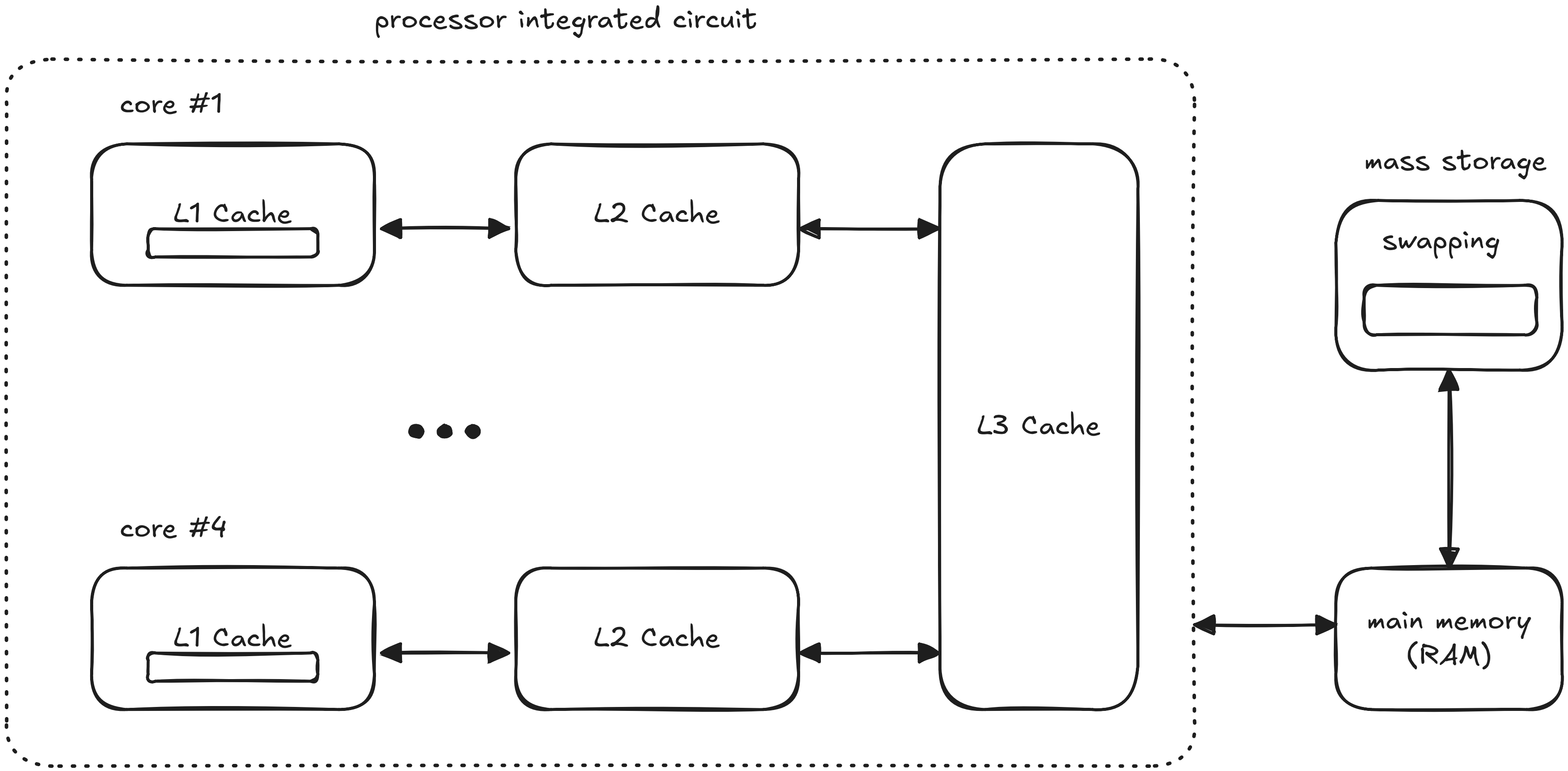
Over time performancee variation of single processor vs. memory

Source: Computer Architecture: A Quantitative Approach

processor: avg request per second, **memory**: inverse DRAM access latency



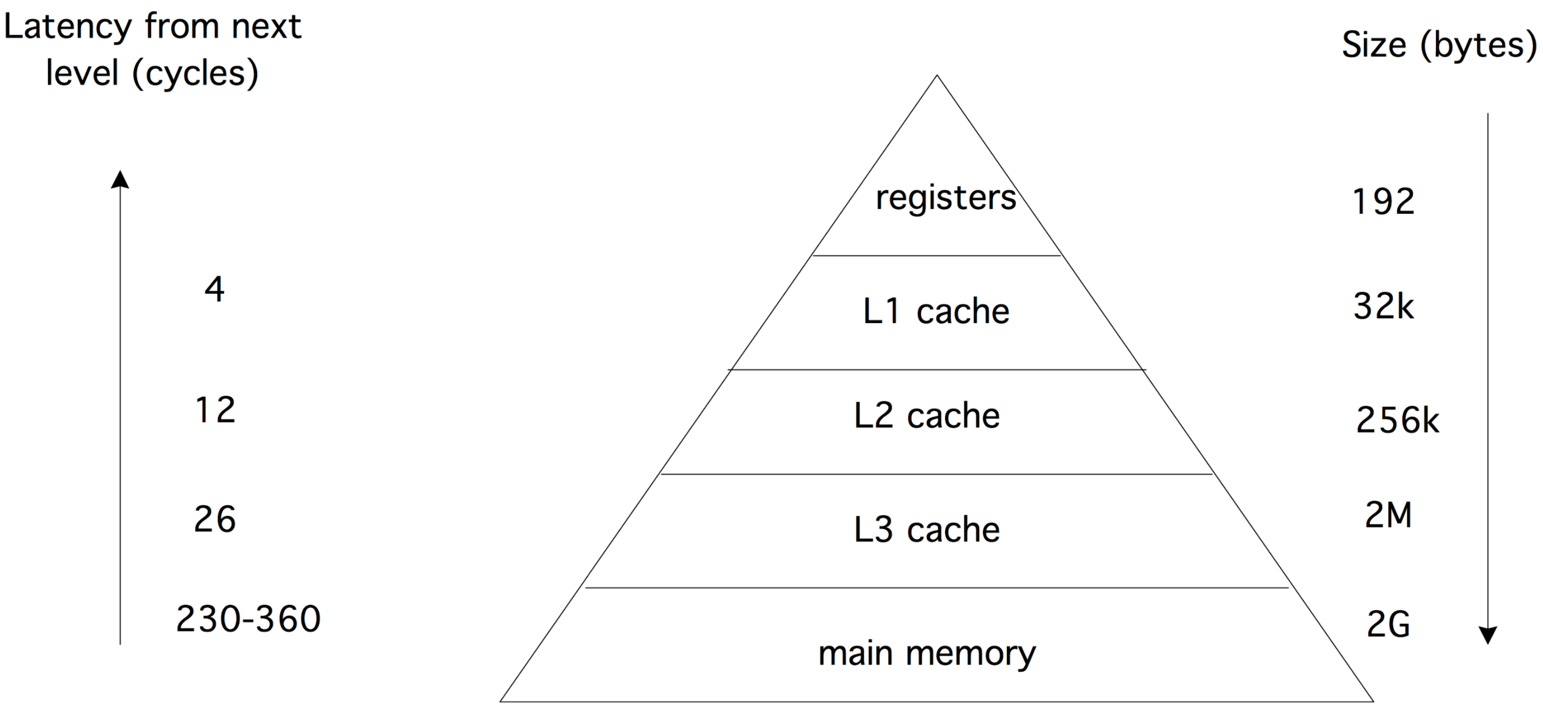
Memory Hierarchy for a 4-core processor system



Overview

Memory hierarchy of an Intel Sandy Bridge, characterized by speed and size

Source: The Art of High Performance Computing, volume 1



Overview

In a **multicore processor**, where multiple simultaneous processes may compete for access to shared data—whether protected by critical regions or not—an additional challenge arises: **cache coherence**. In such cases, copies of the same memory block may reside in **level 1 (L1) or level 2 (L2) cache lines** of different processor cores, potentially leading to inconsistencies if modifications are not properly managed.

To ensure that all processors consistently access the most up-to-date data, a **write-through policy** should be implemented for **level 1 (L1) and level 2 (L2) caches**. When a write occurs, all copies at these cache levels should be marked as **stale**, ensuring that any subsequent read triggers a transfer from the **level 3 (L3) cache**, which holds the most recent version of the data.

Overview

Version 1

```c
#define N 20000
int x[N];   // random values
int y[N];   // set to zero
int a[N][N];

int i, j;
for (j = 0; j < N; j++)
    for (i = 0; i < N; i++)
        y[i] += a[i][j] * x[j];
```

Version 2

```c
#define N 20000
int x[N];   // random values
int y[N];   // set to zero
int a[N][N];

int i, j;
for (i = 0; i < N; i++)
    for (j = 0; j < N; j++)
        y[i] += a[i][j] * x[j];
```

Can you find the difference?

Compilation without optimization

```
$ gcc -Wall -O0 -o testCompiler1 testCompiler1.c
$ ./testCompiler1
Elapsed time = 0.889183 s

$ gcc -Wall -O0 -o testCompiler2 testCompiler2.c
$ ./testCompiler2
Elapsed time = 0.394932 s
```

Compilation with optimization

```
$ gcc -Wall -O3 -o testCompiler1 testCompiler1.c
$ ./testCompiler1
Elapsed time = 0.880187 s

$ gcc -Wall -O3 -o testCompiler2 testCompiler2.c
$ ./testCompiler2
Elapsed time = 0.065789 s
```

Program vs. Process

Overview

A **program** is generally defined as a **sequence of instructions** that describes the execution of a specific task on a computer. However, for this task to be carried out, the corresponding program must be **executed**. The execution of a program is known as a **process**.

Overview

A **process**, representing an active instance of a program, is characterized by the following components:

- **Addressing space** – The program code and the current values of all its associated variables.
- **Processor context** – The current state of all internal processor registers.
- **I/O context** – Data currently being transferred between input and output devices.
- **Execution state** – The current status of the process, indicating its stage in the execution cycle.

Process state

A **process** can exist in different states throughout its lifecycle. The most important process states include:

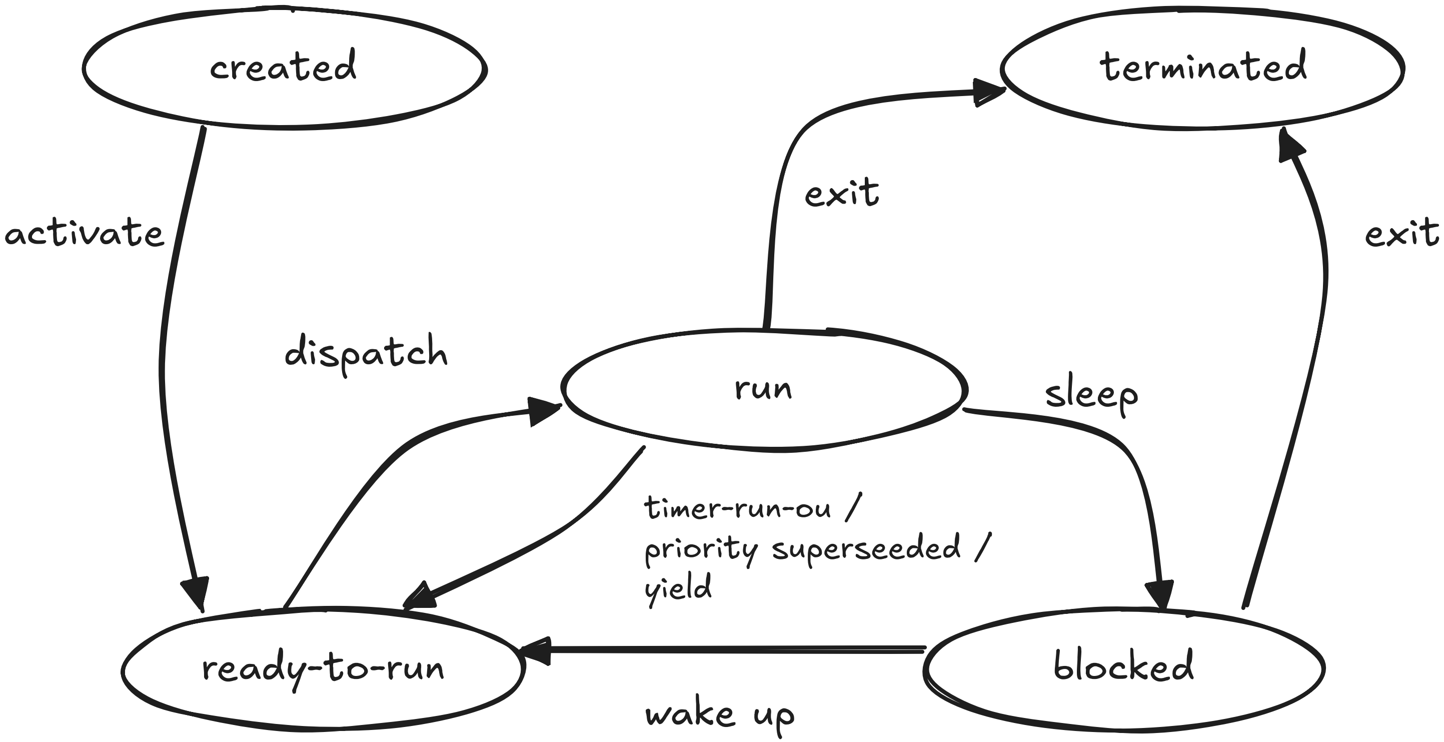
- **Running** – The process is actively executing on the processor.
- **Ready-to-run** – The process is waiting for processor assignment to begin or resume execution.
- **Blocked** – The process is temporarily halted, unable to proceed until an external event occurs (e.g., access to a resource, completion of an input/output operation).

Process state

State transitions are typically triggered by an **external source**, such as the **operating system**. However, in some cases, a process may trigger its own state transition.

The component of the operating system responsible for managing process state transitions is called the **scheduler** (specifically, the **processor scheduler** in this context). It is a fundamental part of the **kernel**, which handles **exception management** and schedules processor time and other system resources for active processes.

Process state diagram



Process state diagram

Process State Transitions

- **Activate** – A process is created and placed in the **ready-to-run queue**, awaiting scheduling for execution.
- **Dispatch** – The scheduler selects a process from the **ready-to-run queue** and assigns it to the processor for execution.
- **Timer-run-out** – The currently executing process exhausts its allocated processor time slot and is preempted (preemptive scheduling).
- **Priority superseded** – The executing process is preempted because a higher-priority process in the **ready-to-run queue** requires the processor (preemptive scheduling).
- **Yield** – The process voluntarily releases the processor, allowing other processes to execute (non-preemptive scheduling).
- **Sleep** – The process is temporarily suspended and must wait for an external event to proceed.
- **Wake up** – The external event the process was waiting for occurs, allowing it to resume execution.
- **Exit** – The process **terminates** after completing execution or being explicitly ended.

Processes vs. Threads

Overview

A **process** is characterized by the following key properties:

- **Resource ownership** – A private **address space** and a dedicated set of **communication channels** for interacting with input and output devices.
- **Thread of execution** – Includes:
  - A **program counter (PC)** that points to the next instruction to be executed.
  - A **set of internal registers** containing the current values of active variables.
  - A **stack** that maintains execution history, storing a **frame** for each subroutine that has been called but not yet completed.

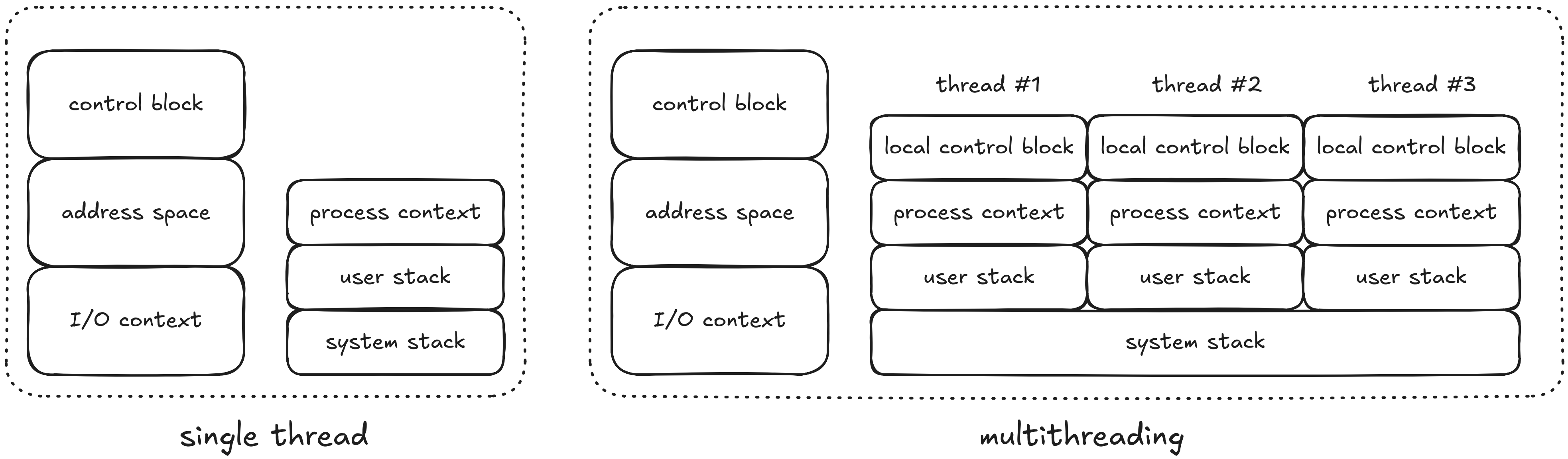
Overview

Although these properties are typically bundled together within a process, the execution environment can **treat them separately**. In such cases, a **process** is viewed as a collection of **resources and threads**.

A **thread**, also known as a **lightweight process (LWP)**, represents an **independent runnable entity** within the context of a single process.

Multithreading refers to an execution environment that supports the creation of **multiple threads of execution** within the same process, enabling **concurrent execution** and **efficient resource sharing**.

Overview


control block
address space
I/O context
process context
user stack
system stack
single thread
control block
address space
I/O context
thread #1
thread #2
thread #3
local control block
local control block
local control block
process context
process context
process context
user stack
user stack
user stack
system stack
multithreading

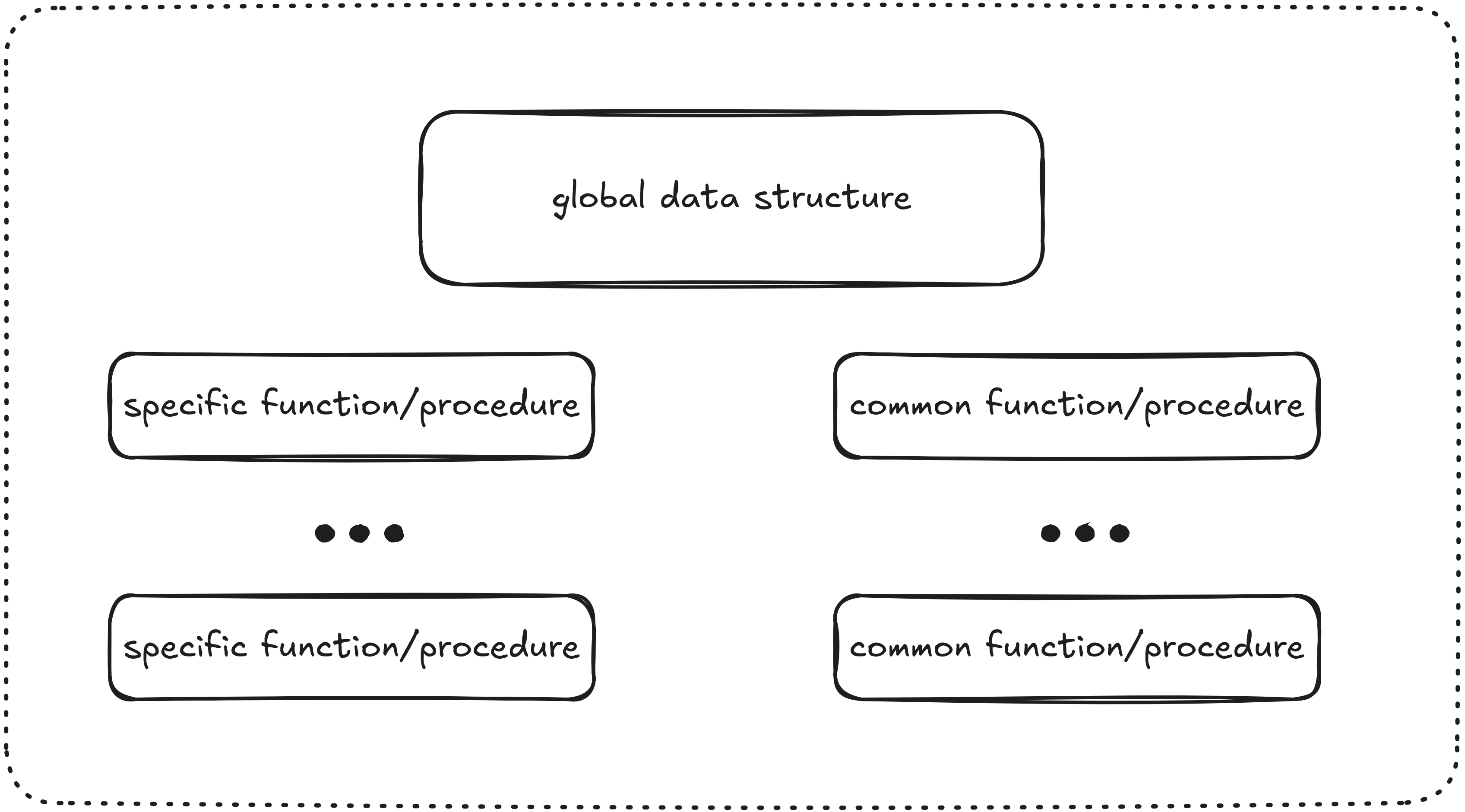
Advantages of multithreading

- **Simplified solution decomposition and enhanced modularity** – Programs involving multiple activities or servicing multiple requests are easier to design and implement in a **concurrent** model compared to a purely **sequential** approach.
- **Improved resource management** – Sharing the **address space** and **I/O context** among the threads of an application **reduces the complexity** of managing **main memory usage** and access to **input/output devices**.

Advantages of multithreading

- **Increased efficiency and execution speed**
  - A thread-based approach, compared to a process-based one, requires fewer **operating system resources**, making operations such as **thread creation, termination, and context switching** significantly lighter and more efficient.
  - In **symmetric multiprocessing (SMP)** systems, multiple threads of the same application can be **executed in parallel**, further enhancing performance and reducing execution time.

Organization of a multithread program



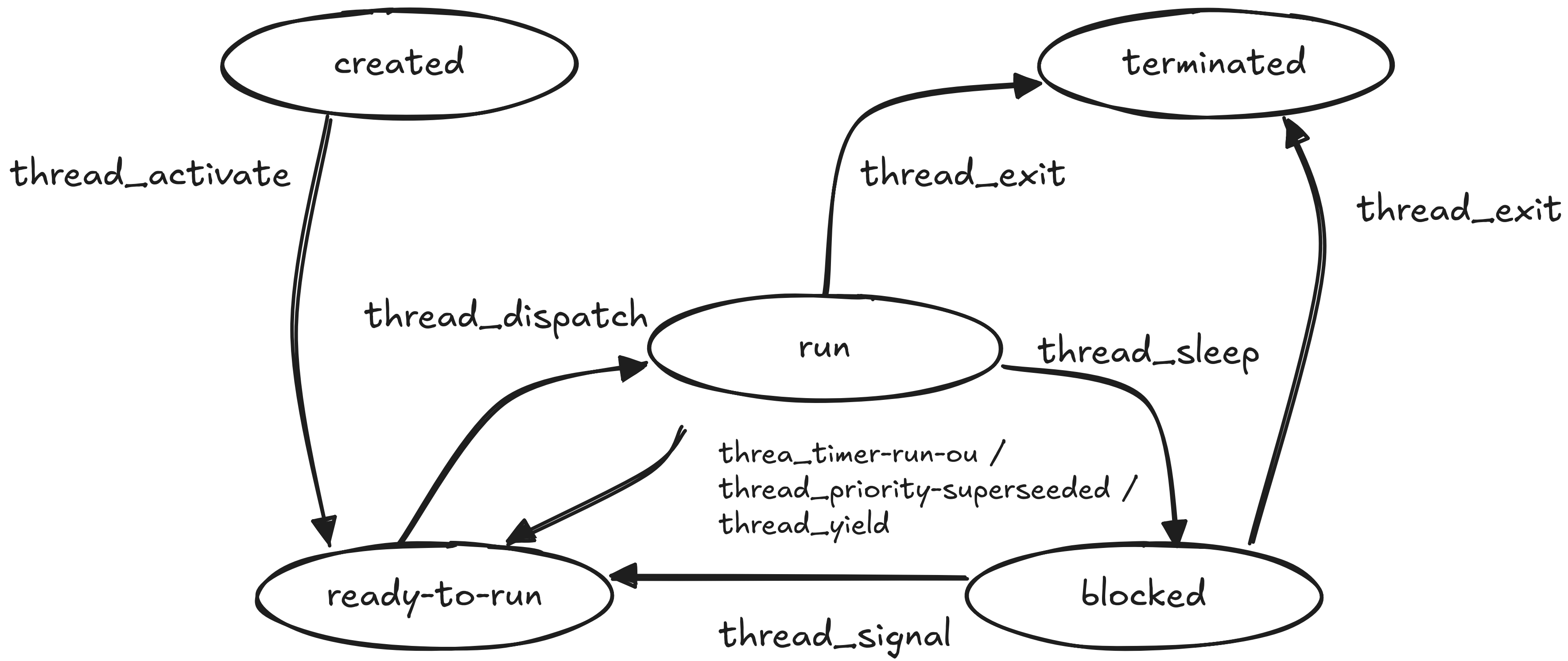
Organization of a multithread program

- **Function-based execution** – Each thread is typically associated with the execution of a **function or procedure** responsible for performing a specific task or activity.

- **Global data structure as a shared space** – The **global data structure** serves as a **shared information space**, consisting of **variables and communication channels** for input/output operations. Multiple threads that coexist at a given time can access this shared space for both **reading and writing**.

- **Main program and thread lifecycle** – The **main program**, represented by a function or procedure performing a specific activity, is the **first thread created** in the execution environment. It is typically **the last thread to terminate**, ensuring that all other threads complete their execution before the program concludes.

Thread state diagram



Types of Threads

- **User-Level Threads (ULTs)** – Threads are implemented via a **user-level library**, which provides support for **thread creation, management, and scheduling** without kernel involvement. This approach is highly **versatile and portable** but relatively **inefficient** because the **kernel only perceives processes, not threads**. Consequently, if a **single thread** makes a **blocking system call**, the **entire process** is blocked, even if other threads within the process are ready to execute.

Types of Threads

- **Kernel-Level Threads (KLTs)** – Threads are managed **directly by the kernel**, which provides built-in support for **thread creation, scheduling, and management**. While this implementation is **operating system-specific**, it has significant advantages:
  - If one thread **blocks**, the **remaining threads** within the process can still be **dispatched for execution**.
  - **Parallel execution** is possible on **multicore processors**, improving overall system efficiency.

Suggested Reading

Computer Organization and Architecture: Designing for Performance, Stalling W., 9th Edition, Prentice Hall, 2013

- Chapter 1: Introduction
- Chapter 2: Computer Evolution and Performance

The Art of High Performance Computing, Volume 1, 3rd edition 2022, formatted April 2, 2024

- Chapter 1: Single-processor Computing
- Chapter 2: Parallel Computing