Lecture 1: introduction to multithreading

concurrency, parallelism, agents, threads, scheduler, Amdahl's law, race condition, deadlock, wait-for graph

Alexander Filatov
filatovaur@gmail.com

https://github.com/Svazars/parallel-programming/blob/main/slides/pdf/l1.pdf

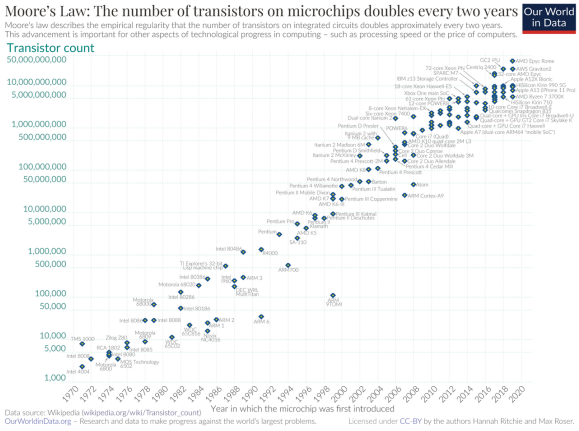
Lecture plan

1. Motivation
2. Concurrency and parallelism
3. Threads and processes
4. Scheduler
5. Concurrent problems
   - Embarrassingly parallel problems and Amdahl's law
   - Race condition
   - Data race
   - Visibility
   - Deadlock
   - Priority inversion
6. Summary

Motivation

Moore's law

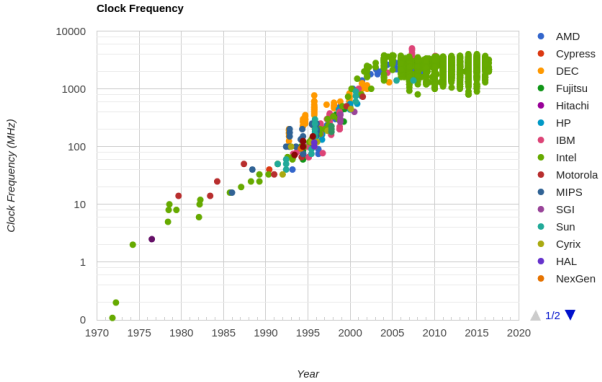
`https://en.wikipedia.org/wiki/Moore%27s_law#/media/File:Moore's_Law_Transistor_Count_1970-2020.png`



Motivation

Clock frequency

http://cpudb.stanford.edu/visualize/clock_frequency.html



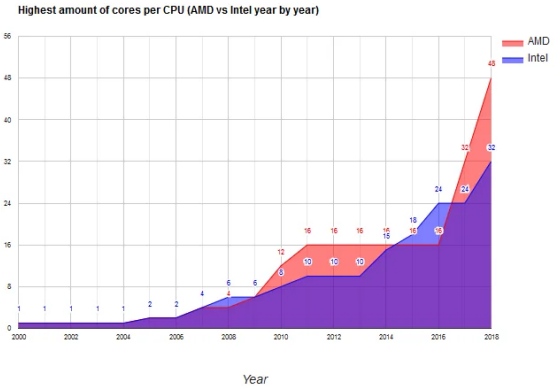
Question time

Question: Where are all these transistors?

Motivation

Core per CPU

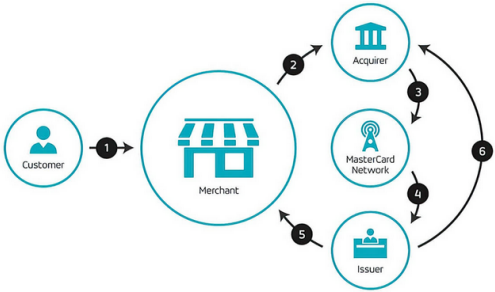
https://www.reddit.com/r/Amd/comments/6cu5ss/highest_amount_of_cores_per_cpu_amd_vs_intel_year



Motivation

Multi-agent systems

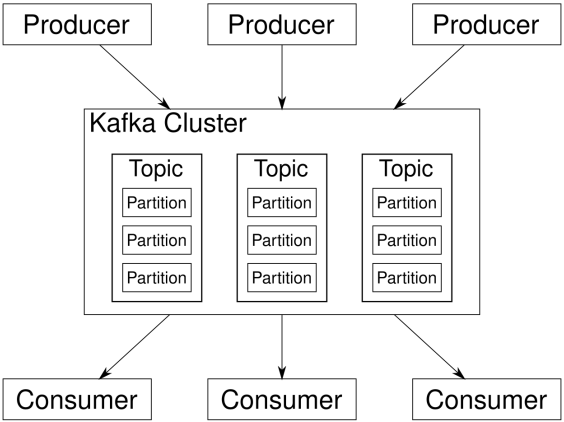
https://sea.mastercard.com/en-region-sea/business/merchants/start-accepting/payment-process.html



Motivation

Multi-agent systems

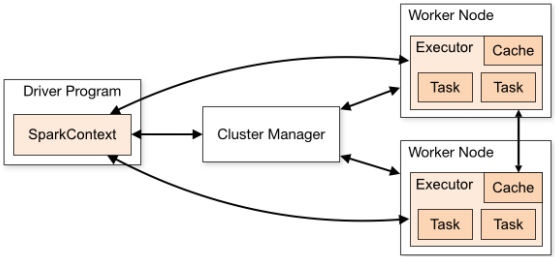
https://en.wikipedia.org/wiki/Apache_Kafka#/media/File:Overview_of_Apache_Kafka.svg



Motivation

Multi-agent systems

https://spark.apache.org/docs/latest/cluster-overview.html



Motivation

Multi-agent systems

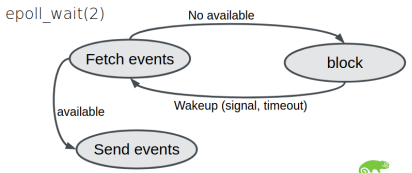
https://events19.linuxfoundation.org/wp-content/uploads/2018/07/dbueso-oss-japan19.pdf

Locking Rules

struct eventpoll
spinlock_t lock
mutex lock
wait_queue_head_t wq
wait_queue_head_t poll_wait
list_head rdlist
rb_root rbr
struct epitem *ovflist
wakeup_source *ws
user_struct *user
file *file
int visited
list_head visited_list_link

Mutex: serialization while transferring events to userspace
copy_to_user might block.
Protect epoll_ctl(2) operations, file exit, etc.

Spinlock: serialization inside IRQ context, cannot sleep.
Protects ready and *overflow* list manipulation.
(Must already hold the ep->mutex)



Common problems

- Many **independent** parts
- Different **speed**
- Need to **communicate** with each other
- Need to **coordinate** execution
- May **fail** spuriously

Agents

This course is about **communicating agents**. We will see

- Alice & Bob
- Processes in operating system
- Threads in operating system
- Separate computers that use networking to pass messages
- ...

Agents will try to:

- Decide who is in charge
- Pass data without corruption
- Coordinate execution of several steps in complicated task
- Find out how many agents take part in the communication
- ...

Different worlds

There are a lot of basic concepts behind the communication of independent agents.
But worlds are *different*:

- timings
  - minutes (conversation)
  - seconds (networking)
  - milliseconds (shared memory)
  - nanoseconds (CPU)
- delivery
  - possible distortion (misunderstanding)
  - possible loss (UDP packet)
  - visibility issue (not-yet-updated memory cell/stale value in CPU cache)
- denial-of-service
  - several people do phone call to the same person
  - network go offline
  - shared memory of single process in consistent (yet weakly ordered)
- ...

Lecture plan

Question time

Question: "concurrency", "parallelism", who could translate both words to Russian?

Concurrency vs parallelism

Situations

Sequential and interruptible = concurrent[1]

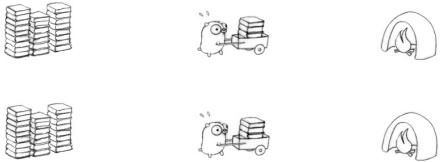
[1] https://go.dev/talks/2012/waza.slide#12

Concurrency vs parallelism

Situations

Sequential and interruptible = concurrent[1]

Independent and simultaneous = parallel[2]



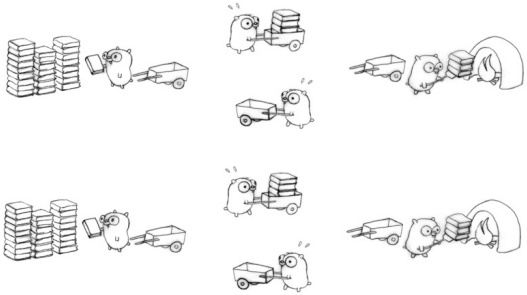
[1] https://go.dev/talks/2012/waza.slide#12

[2] https://go.dev/talks/2012/waza.slide#15

Concurrency vs parallelism

Two sides of the same medal

Real life is mixed[3]



Fun fact: real people in industry/academia may "interpret" parallelism/concurrency in different ways

[3] https://go.dev/talks/2012/waza.slide#22

Concurrency vs parallelism

English

- Parallel
- Concurrent
- Independent
- Simultaneous
- At the same time

Question time

Question: "Parallel", "Concurrent", "Independent", "Simultaneous", "At the same time". Use one Russian word which would be the best translation for all 5 concepts.

Concurrency vs parallelism

English

- Parallel
- Concurrent
- Independent
- Simultaneous
- At the same time

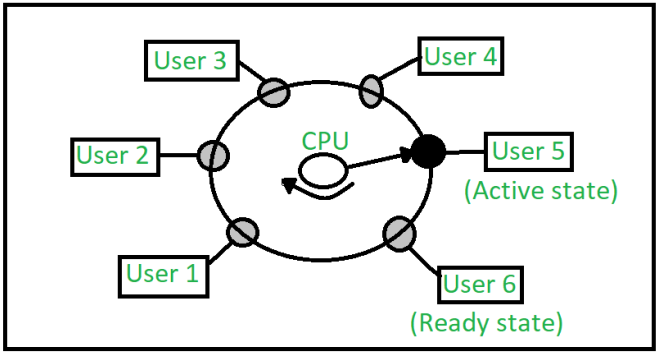
Parallel – *could* execute simultaneously (independent, applicable to different executors)
Concurrent – *could* execute interchangeably (interruptible, applicable to the same executor)
Any real-life task combines parallel and concurrent parts[4].

[4] https://stackoverflow.com/questions/1050222/what-is-the-difference-between-concurrency-and-parallelism

Concurrency vs parallelism

Time-sharing for users

Single CPU, many users[5]

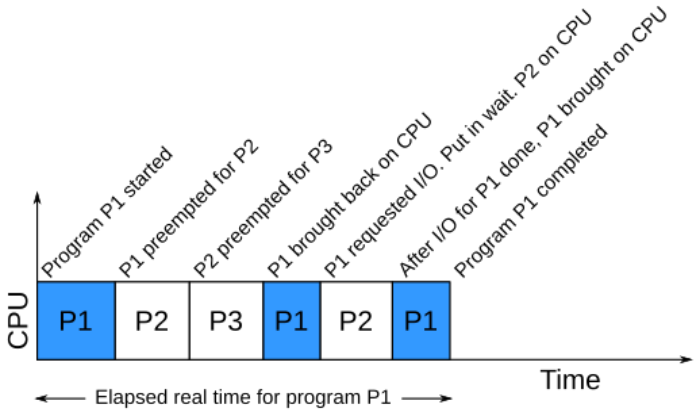


[5] https://www.geeksforgeeks.org/time-sharing-operating-system/

Concurrency vs parallelism

Time-sharing for tasks

Single CPU, pre-emptible tasks[6]

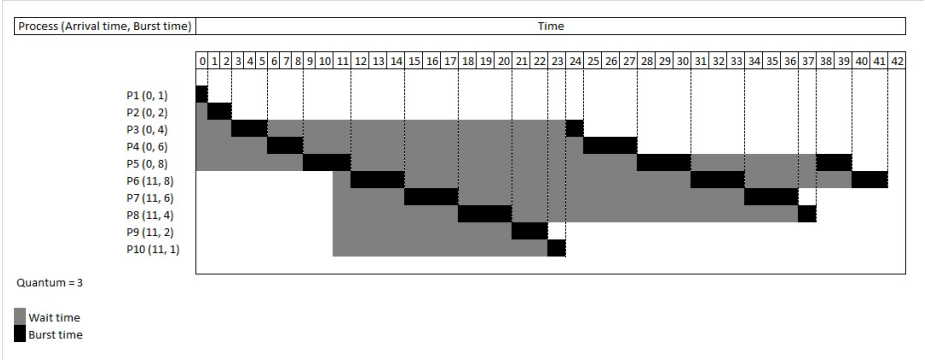


[6] https://en.wikipedia.org/wiki/CPU_time

Concurrency vs parallelism

Time-sharing scheduling strategies

Round-robin scheduling[7]



[7] https://en.wikipedia.org/wiki/Round-robin_scheduling

Concurrency vs parallelism

Conclusion

Concurrency and parallelism are two edges of the same phenomena.
In computer science (and in our course)

- parallelism
- concurrency
- interruptibility
- independence

have more formal meaning than in "usual" communication. Be aware.

Lecture plan

Question time

Question: Difference between "threads" and "processes" in one word?

Threads vs processes

Blast from the past

Process[8]

- instance of computer program (program code and other stuff like .data, .rodata, .bss ...)
- owns resources (RAM, CPU, network ...) allocated by OS
- has logical and physical access permissions (filesystem, user capabilities ...)
- described by state (context)
- managed by OS
- **isolated** from other processes (virtual memory, namespaces ...)

[8] https://en.wikipedia.org/wiki/Process_(computing)

Threads vs processes

Blast from the past

Thread[9]

- part of process (executes some code in some context)
- owns thread-specific resources (stack, TLS ...)
- has specific metadata (priority, TID ...)
- described by state (context)
- managed by OS scheduler
- **shares** memory with other threads

[9] https://en.wikipedia.org/wiki/Thread_(computing)

Threads vs processes

Bird-eye view

Processes and threads are **agents**

- different speed
- could communicate
- need coordination

Question time

Question: Threads need *more* or *less* coordination compared to processes?

Threads vs processes

Bird-eye view

Processes and threads are **agents**

- different speed
- could communicate
- need coordination

Threads vs processes

Bird-eye view

Processes and threads are **agents**

- different speed
- could communicate
- need coordination

In some sense, threads need *less* coordination (memory is already shared).

Threads vs processes

Bird-eye view

Processes and threads are **agents**

- different speed
- could communicate
- need coordination

In some sense, threads need *less* coordination (memory is already shared).
In some sense, threads need *more* coordination (memory is already shared).

Threads vs processes

Examples: process

```
echo "Sequential"
wc 50_000_000.txt
```

```
real    0m 4,900s
user    0m 4,212s
sys     0m 0,240s
```

```
echo "Parallel"
{ ( head -n 25000000 50_000_000.txt | wc) & }
{ ( tail -n 25000000 50_000_000.txt | wc) & }
wait
```

```
real    0m 2,323s
user    0m 4,576s
sys     0m 1,084s
```

Threads vs processes

Examples: process

```
echo "Sequential"
wc 50_000_000.txt
```

```
real    0m 4,900s
user    0m 4,212s
sys     0m 0,240s
```

```
echo "Parallel"
{ ( head -n 25000000 50_000_000.txt | wc) & }
{ ( tail -n 25000000 50_000_000.txt | wc) & }
wait
```

```
real    0m 2,323s
user    0m 4,576s
sys     0m 1,084s
```

- Create processes
- Wait completion
- Pass data via pipes, files, advanced IPC

Threads vs processes

Examples: pthreads[10].

Step 1. Define and allocate utility struct

```
struct thread_info {    /* Used as argument to thread_start() */
  pthread_t thread_id;  /* ID returned by pthread_create() */
  int       thread_num; /* Application-defined thread # */
};

struct thread_info * tinfo_a = malloc(sizeof(struct thread_info));
if (tinfo_a == NULL) handle_error("malloc");
```

[10] POSIX threads: https://en.wikipedia.org/wiki/Pthreads

Threads vs processes

Examples: pthreads

Step 2. Initialize thread attributes

```
pthread_attr_t attr;
s = pthread_attr_init(&attr);
if (s != 0) handle_error("pthread_attr_init");
```

Step 3. Define thread body

```
static void * thread_A_start(void *arg) {
  struct thread_info *tinfo = arg;
  printf("Thread A id = %d: hello world!\n", tinfo->thread_num);
  return NULL;
}
```

Threads vs processes

Examples: pthreads

Step 4. Initialize utility struct and start new thread

```
tinfo_a->thread_num = 1;
s = pthread_create(&(tinfo_a->thread_id), &attr, &thread_A_start, tinfo_a);
if (s != 0) handle_error("pthread_create");
```

Step 5. Join thread

```
void *res;
s = pthread_join(tinfo_a->thread_id, &res);
if (s != 0) handle_error("pthread_join");
printf("Joined with thread A, result= %p\n", (char *) res);
```

Threads vs processes

Examples: pthreads

```c
struct thread_info {
  pthread_t thread_id;
  int       thread_num;
};
```

```c
static void * thread_A_start(void *arg) {
  struct thread_info *tinfo = arg;
  printf("Thread A id = %d: hello world!\n",
    tinfo->thread_num);
  return NULL;
}
```

```c
// ...
tinfo_a->thread_num = 1;
pthread_create(&(tinfo_a->thread_id), &attr, &thread_A_start, tinfo_a);
pthread_join(tinfo_a->thread_id, &res);
printf("Joined with thread A, result= %p\n", (char *) res);
```

Threads vs processes

Examples: Java threads

```
static class ThreadA extends Thread {
    private int idx;
    ThreadA(int i) { this.idx = i; }
    @Override
    public void run() {
        System.out.printf("Thread A, idx = %d: hello world\n", this.idx);
    }
}
public static void main(String... args) throws Exception {
    Thread t = new ThreadA(1);
    t.start(); // !!! not t.run !!!
    t.join();
}
```

Question time

Question: Why somebody would use Java for parallel programming course, not C or C++?

Java threads and exceptions

Thread **A** starts thread **B** and `joins`. Uncaught exception happens in thread **B**.

- What happens in thread **B**?
- What happens in thread **A**?
- What if thread **C** `joins` thread **B** *after* exception happened?
- What if thread **D** `joins` thread **A**?

Java threads and exceptions

Thread **A** starts thread **B** and `joins`. Uncaught exception happens in thread **B**.

- What happens in thread **B**?
- What happens in thread **A**?
- What if thread **C** `joins` thread **B** *after* exception happened?
- What if thread **D** `joins` thread **A**?

Homework, mail

Task 1.1 Create 3 Java programs for these cases, explain results in several sentences.

Threads vs processes

Conclusion

From our course perspective:

- processes hold all the context and resources
- processes use many different parts of OS
- threads use ultra-fast shared memory for communication
- threads are minimal units eligible for OS scheduler

Almost all lectures will be about

- inter-process multi-threading
- inside high-level programming language
- run by multiprocessor hardware with MMU
- on non-real-time OS with pre-emptive multitasking

Threads vs processes

Conclusion

From our course perspective:

- processes hold all the context and resources
- processes use many different parts of OS
- threads use ultra-fast shared memory for communication
- threads are minimal units eligible for OS scheduler

Almost all lectures will be about

- inter-process multi-threading
- inside high-level programming language
- run by multiprocessor hardware with MMU
- on non-real-time OS with **pre-emptive multitasking**

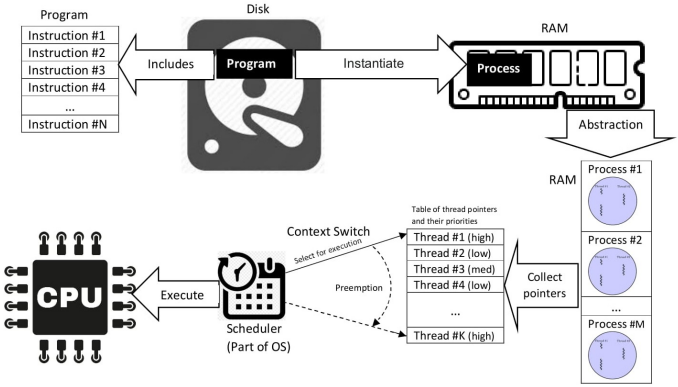
Lecture plan

1. Motivation
2. Concurrency and parallelism
3. Threads and processes
4. Scheduler
5. Concurrent problems
   - Embarrassingly parallel problems and Amdahl's law
   - Race condition
   - Data race
   - Visibility
   - Deadlock
   - Priority inversion
6. Summary

Scheduler

Problem overview

https://en.wikipedia.org/wiki/Process_(computing)#/media/File:Concepts-_Program_vs._Process_vs._Thread.jpg



Scheduler

Problem overview

Scheduler – magic black box

- "knows and owns" every unit of scheduling (thread)
- may pre-empt execution of any unit (save state from CPU to memory)
- may enable execution of any pending unit (load state from memory to CPU)
- implements some scheduling policy (round-robin, shortest remaining time ...)

Scheduler

Problem overview

Scheduler – magic black box

- "knows and owns" every unit of scheduling (thread)
- may pre-empt execution of any unit (save state from CPU to memory)
- may enable execution of any pending unit (load state from memory to CPU)
- implements some scheduling policy (round-robin, shortest remaining time ...)

Save state / load state == switch CPU context of execution

Question time

Question: What is context switch on modern OS (e.g. Linux) and modern hardware with MMU (e.g. x86_64 intel Skylake)?

Scheduler

Context switch

Do not forget all the important data

- registers (general-purpose, floating-point, other)
- stack pointer
- instruction pointer
- segmentation/page tables
- translation lookaside buffer flush
- ...

Requires user-space -> kernel space transition.

Scheduler

Context switch

Do not forget all the important data

- registers (general-purpose, floating-point, other)
- stack pointer
- instruction pointer
- segmentation/page tables
- translation lookaside buffer flush
- ...

Requires user-space -> kernel space transition.
Needs *amortization*.

Scheduler

Context switch

Do not forget all the important data

- registers (general-purpose, floating-point, other)
- stack pointer
- instruction pointer
- segmentation/page tables
- translation lookaside buffer flush
- ...

Requires user-space -> kernel space transition.
Needs *amortization*.
Scheduling quantum.

Scheduler

Scheduling quantum

Quantum may be arbitrary function of `(current task, other tasks, state of OS)`.
Do not forget about

- frequency of context switches
- "price" of context switch
- thread priority
- process priority
- "price" of CPU migration
- occupation of other processors
- ...

It is hard to trust timings[11][12].

[11] `https://shipilev.net/blog/2014/nanotrusting-nanotime`

[12] `https://pzemtsov.github.io/2017/07/23/the-slow-currenttimemillis.html`

Scheduler

Pre-emptive and cooperative multitasking

When to context switch?
Anytime anywhere:

- flexible
- easy-to-use model
- requires additional care
- has counter-examples
- relies on heuristics

When allowed to:

- fine-grained control
- nontrivial behaviour
- requires additional care
- has counter-examples
- relies on heuristics

Scheduler

Pre-emptive and cooperative multitasking

When to context switch?
Anytime anywhere:

- flexible
- easy-to-use model
- requires additional care
- has counter-examples
- relies on heuristics

When allowed to:

- fine-grained control
- nontrivial behaviour
- requires additional care
- has counter-examples
- relies on heuristics

In real environments it is always a hybrid solution[13].

[13] The long road to lazy preemption: `https://lwn.net/Articles/994322/`

Scheduler

Goals

- In theory, there exist a *perfect* schedule
- In practice, almost any scheduling problem is NP-complete
- In applications, workload is dynamically generated on-the-fly

Scheduler

Goals

- In theory, there exist a *perfect* schedule
- In practice, almost any scheduling problem is NP-complete
- In applications, workload is dynamically generated on-the-fly

Realistic goals:

- Max *utilization*, not max *efficiency*
- Reasonable *latency*
- Soft/hard *real-time* constraints
- *Energy efficiency*

Question time

Question: Describe situation when *max utilization* scheduling contradicts *max efficiency* scheduling

Lecture plan

1. Motivation
2. Concurrency and parallelism
3. Threads and processes
4. Scheduler
5. Concurrent problems
   - Embarrassingly parallel problems and Amdahl's law
   - Race condition
   - Data race
   - Visibility
   - Deadlock
   - Priority inversion
6. Summary

Concurrent blocks you know so far

1. Create new thread
2. Modify fields concurrently
3. Join thread

Concurrent blocks you know so far

1. Create new thread
2. Modify fields concurrently
3. Join thread

Combining **1** and **2**:

- Solve "easy" parallel problems
- Encounter logic errors (race conditions)
- Encounter concurrent memory access effects (data races)

Concurrent blocks you know so far

1. Create new thread
2. Modify fields concurrently
3. Join thread

Combining **1** and **2**:

- Solve "easy" parallel problems
- Encounter logic errors (race conditions)
- Encounter concurrent memory access effects (data races)

Using **3**:

- Encounter visibility bugs (stale value)
- Encounter no-progress blocking (deadlock)
- Observe slow-progress issues (priority inversion)

Lecture plan

1. Motivation
2. Concurrency and parallelism
3. Threads and processes
4. Scheduler
5. Concurrent problems
   - Embarrassingly parallel problems and Amdahl's law
   - Race condition
   - Data race
   - Visibility
   - Deadlock
   - Priority inversion
6. Summary

Embarrassingly parallel problems

Introduction

- Embarrassingly parallel
- Embarrassingly parallelizable
- Perfectly parallel
- Delightfully parallel
- Pleasingly parallel

Key properties:

- Granularity
- Minimal coordination

Question time

Question: Any ideas for embarrassingly parallel problems?

Embarrassingly parallel problems

Examples

- function applied to collection of items (*map*)
  - ray tracing
  - word count in independent files
  - proof-of-work crypto
  - bioinformatics search (BLAST etc)
  - Discrete Fourier transform
- associative operation applied to large array (order-independent *reduce*)
  - sum of elements in integer array
  - word count for all books in e-library
- ...

Amdahl's law

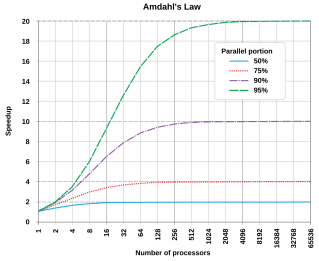
Key ideas

Any problem consists of *serial* part and *parallel* part.

Speed-up of parallel part depends on computational resources. Serial part is "fixed work".

- **Diminishing Returns**: beyond certain point, adding more processors doesn't significantly increase speedup
- **Limited Speedup**: problem cannot be solved faster than serial part

https://en.wikipedia.org/wiki/Amdahl%27s_law



Amdahl's law

Examples

- Programmer enhances a part of the code that represents 10% of the total execution time. This part starts to work 10 000 times faster. What is total speedup of a program?
- Problem could be split into two parts: A and B. A is serial, B is parallel. On single CPU, A takes 50 minutes, B takes 250 minutes. How many CPUs do you need to solve the problem in 100 minutes (achieve 3x speed-up)?
- Prepare an example where 10x speed-up could be achieved by using 100 CPUs only.

Amdahl's law

Examples

- Programmer enhances a part of the code that represents 10% of the total execution time. This part starts to work 10 000 times faster. What is total speedup of a program?
- Problem could be split into two parts: A and B. A is serial, B is parallel. On single CPU, A takes 50 minutes, B takes 250 minutes. How many CPUs do you need to solve the problem in 100 minutes (achieve 3x speed-up)?
- Prepare an example where 10x speed-up could be achieved by using 100 CPUs only.

Homework, mail

Task 1.2 Provide answers and explanation for all three problems

Embarrassingly parallel problems

Piece of advice

Such problems usually quite interesting and practically-oriented.
They do **not** require non-trivial coordination so we will not cover them in this course.
In many domains, most of "performance issues" are (almost) embarrassingly parallel.
There are plenty of

- easy-to-use
- straightforward
- safe
- stable
- performant

solutions to such problems. Just use them and get speed-up predicted by Amdahl's law.
If you go real concurrency, it will be **harder**.

Lecture plan

1. Motivation
2. Concurrency and parallelism
3. Threads and processes
4. Scheduler
5. Concurrent problems
   - Embarrassingly parallel problems and Amdahl's law
   - Race condition
   - Data race
   - Visibility
   - Deadlock
   - Priority inversion
6. Summary

Race condition

Example

```
public static void main(String... args) throws Exception {
    Thread t1 = new Thread(() -> { System.out.println("Thread A"); });
    Thread t2 = new Thread(() -> { System.out.println("Thread B"); });
    t1.start(); t2.start();
    t1.join(); t2.join();
}
```

Race condition

Example

```
public static void main(String... args) throws Exception {
    Thread t1 = new Thread(() -> { System.out.println("Thread A"); });
    Thread t2 = new Thread(() -> { System.out.println("Thread B"); });
    t1.start(); t2.start();
    t1.join(); t2.join();
}
```

Execution 1: `Thread A Thread B`

Execution 2: `Thread B Thread A`

Race condition

Key idea

- Every single operation in program is OK (properly synchronized, atomic)
- Program as a whole suffers from non-deterministic behaviour
- The only reason to this is different speed of execution

Race condition

Key idea

- Every single operation in program is OK (properly synchronized, atomic)
- Program as a whole suffers from non-deterministic behaviour
- The only reason to this is different speed of execution

Homework, mail

Task 1.3 Open `https://deadlockempire.github.io`, pass all levels up to "Confused counter", inclusive.

Race condition

Conclusion

Race condition

- at least 2 threads involved
- behaviour of system depends on sequence of events (e.g. timings)
- leads to non-deterministic results

Race condition

Conclusion

Race condition

- at least 2 threads involved
- behaviour of system depends on sequence of events (e.g. timings)
- leads to non-deterministic results

Really **suspicious**

- complicates analysis of algorithm (correctness, performance, resource utilization)
- may lead to rarely reproducible bugs
- may lead to random denial-of-service events

Race condition

Conclusion

Race condition

- at least 2 threads involved
- behaviour of system depends on sequence of events (e.g. timings)
- leads to non-deterministic results

Really **suspicious**

Race condition

Conclusion

Race condition

- at least 2 threads involved
- behaviour of system depends on sequence of events (e.g. timings)
- leads to non-deterministic results

Really **suspicious**
Not **necessarily** a problem

- Server counts number of incoming requests.
- Search engine crawls page graph using several workers. Many pages visited several times, deduplication makes result consistent.
- Torrent client downloads several chunks of a file simultaneously.

Race condition

Conclusion

Race condition

- at least 2 threads involved
- behaviour of system depends on sequence of events (e.g. timings)
- leads to non-deterministic results

Really **suspicious**
Not **necessarily** a problem

Race condition

Conclusion

Race condition

- at least 2 threads involved
- behaviour of system depends on sequence of events (e.g. timings)
- leads to non-deterministic results

Really **suspicious**
Not **necessarily** a problem
Main headache of our course:

- **Race condition may be a bug or intentional behaviour depending on programmer's intention**

In general, cannot be detected by static code analysis/dynamic instrumentation tool.

Question time

Question: Some tools detect race conditions in concurrent software. What is "false positive" detection of race condition?

Lecture plan

1. Motivation
2. Concurrency and parallelism
3. Threads and processes
4. Scheduler
5. Concurrent problems
   - Embarrassingly parallel problems and Amdahl's law
   - Race condition
   - Data race
   - Visibility
   - Deadlock
   - Priority inversion
6. Summary

Data race

Example 1

```
static volatile int x = 0;
public static void main(String... args) throws Exception {
    Thread t = new Thread(() -> { x++; });
    t.start();
    System.out.println(x);
    t.join();
}
```

Data race

Example 1

```
static volatile int x = 0;
public static void main(String... args) throws Exception {
    Thread t = new Thread(() -> { x++; });
    t.start();
    System.out.println(x);
    t.join();
}
```

Execution 1: `x = 0`

Execution 2: `x = 1`

Data race

Example 2

```
static volatile int x = 0;
public static void main(String... args) throws Exception {
    Thread t = new Thread(() -> { x++; });
    t.start();
    x++;
    System.out.println(x);
    t.join();
}
```

Data race

Example 2

```
static volatile int x = 0;
public static void main(String... args) throws Exception {
    Thread t = new Thread(() -> { x++; });
    t.start();
    x++;
    System.out.println(x);
    t.join();
}
```

Execution 1: `x = 1`
Execution 2: `x = 2`

Data race

Example 2.5

```
static volatile int x = 0;
public static void main(String... args) throws Exception {
    Thread t = new Thread(() -> { x++; });
    t.start();
    x++;
    t.join();
    System.out.println(x);
}
```

Data race

Example 2.5

```
static volatile int x = 0;
public static void main(String... args) throws Exception {
    Thread t = new Thread(() -> { x++; });
    t.start();
    x++;
    t.join();
    System.out.println(x);
}
```

Execution 1: `x = 2`

Data race

Example 2.5

```
static volatile int x = 0;
public static void main(String... args) throws Exception {
    Thread t = new Thread(() -> { x++; });
    t.start();
    x++;
    t.join();
    System.out.println(x);
}
```

Execution 1: `x = 2`
Execution 2: `x = 1`

Data race

Example 2.5

```
static volatile int x = 0;
public static void main(String... args) throws Exception {
    Thread t = new Thread(() -> { x++; });
    t.start();
    x++;
    t.join();
    System.out.println(x);
}
```

Execution 1: `x = 2`

Execution 2: `x = 1`

`x++ <=> int tmp = x; tmp = tmp + 1; x = tmp`

Data race

Key idea

Actually, formal definition exists and is language-specific. More on this in next lectures.
Data race:

- at least 2 threads
- at least one of threads is writing data
- operating on the same *memory location* (Lecture 9)
- *simultaneously* (Lecture 6)
- without *proper synchronization* (Lecture 10)

Any data race is race condition. Not every race condition is data race.

Data race

Compound data example

```
static volatile ArrayList<String> list = new ArrayList<>();
public static void main(String... args) throws Exception {
    Thread t1 = new Thread(() -> { list.add("x"); });
    Thread t2 = new Thread(() -> { list.add("y"); });
    t1.start(); t2.start();
    t1.join(); t2.join();
    for (String s : list) {
      System.out.println(s);
    }
}
```

Data race

Compound data example

```
static volatile ArrayList<String> list = new ArrayList<>();
public static void main(String... args) throws Exception {
    Thread t1 = new Thread(() -> { list.add("x"); });
    Thread t2 = new Thread(() -> { list.add("y"); });
    t1.start(); t2.start();
    t1.join(); t2.join();
    for (String s : list) {
      System.out.println(s);
    }
}
```

What is the size of `list`? **1** or **2**?

Data race

Compound data example

```
static volatile ArrayList<String> list = new ArrayList<>();
public static void main(String... args) throws Exception {
    Thread t1 = new Thread(() -> { list.add("x"); });
    Thread t2 = new Thread(() -> { list.add("y"); });
    t1.start(); t2.start();
    t1.join(); t2.join();
    for (String s : list) {
      System.out.println(s);
    }
}
```

What is the size of `list`? **1** or **2**?

Data race may lead to **inconsistent** state even if every thread executed only "valid" operations.

Data race

Word tearing

```
static volatile long data = 0;
public static void main(String... args) throws Exception {
    Thread t1 = new Thread(() -> { data = (1 << 0) + (1 << 33); });
    Thread t2 = new Thread(() -> { data = (2 << 0) + (2 << 33); });
    t1.start(); t2.start();
    System.out.println("data = " + data);
    t1.join(); t2.join();
}
```

Data race

Word tearing

```
static volatile long data = 0;
public static void main(String... args) throws Exception {
    Thread t1 = new Thread(() -> { data = (1 << 0) + (1 << 33); });
    Thread t2 = new Thread(() -> { data = (2 << 0) + (2 << 33); });
    t1.start(); t2.start();
    System.out.println("data = " + data);
    t1.join(); t2.join();
}
```

Is it possible to see `data == (1 « 0) + (2 « 33)`?

Data race

Word tearing

```
static volatile long data = 0;
public static void main(String... args) throws Exception {
    Thread t1 = new Thread(() -> { data = (1 << 0) + (1 << 33); });
    Thread t2 = new Thread(() -> { data = (2 << 0) + (2 << 33); });
    t1.start(); t2.start();
    System.out.println("data = " + data);
    t1.join(); t2.join();
}
```

Is it possible to see `data == (1 « 0) + (2 « 33)`?
Depends on hardware, compiler and language.

Data race

Word tearing

Java Language Specification[14]

> *For the purposes of the Java programming language memory model, a single write to a non-volatile long or double value is treated as two separate writes: one to each 32-bit half. This can result in a situation where a thread sees the first 32 bits of a 64-bit value from one write, and the second 32 bits from another write.*
>
> *Writes and reads of volatile long and double values are always atomic.*
>
> *Writes to and reads of references are always atomic, regardless of whether they are implemented as 32-bit or 64-bit values.*

[14] https://docs.oracle.com/javase/specs/jls/se23/html/jls-17.html#jls-17.6

Data race

Word tearing

Java Language Specification[14]

> *For the purposes of the Java programming language memory model, a single write to a non-volatile long or double value is treated as two separate writes: one to each 32-bit half. This can result in a situation where a thread sees the first 32 bits of a 64-bit value from one write, and the second 32 bits from another write.*
> *Writes and reads of volatile long and double values are always atomic.*
> *Writes to and reads of references are always atomic, regardless of whether they are implemented as 32-bit or 64-bit values.*

Data race may lead to **unexpected** state even if every thread executed only "valid" operations.

[14] https://docs.oracle.com/javase/specs/jls/se23/html/jls-17.html#jls-17.6

Data race

Benign data race

Data race that *does not* affect correctness.
To be sure about this, you need to

- understand all possible interleavings (Lecture 5) and memory ordering effects (Lecture 9)
- ensure particular language memory model allows such hacks (Lecture 10)
- trust language compiler developers and language runtime engineers (Lecture 13)

In our course **any** data race considered harmful.

Data race

Conclusion

Data race is a specific variation of race condition

- concurrent and not properly synchronized
- write access
- to shared data

Always **problematic**

- inconsistent shared state
- unexpected/impossible computation results
- forbidden by language implementation

Could be formalized and avoided

- Strict coding policy (e.g. enforced by linter/compiler)
- Static analysis of source code
- Static analysis of execution traces
- Dynamic instrumenting race finders

Question time

Question: Some tools detect data races in concurrent software. Why do we still have systems with data races?

Lecture plan

1. Motivation
2. Concurrency and parallelism
3. Threads and processes
4. Scheduler
5. Concurrent problems
   - Embarrassingly parallel problems and Amdahl's law
   - Race condition
   - Data race
   - Visibility
   - Deadlock
   - Priority inversion
6. Summary

Visibility

Introduction

It is one of the most subtle notions in concurrency.

Visibility

Introduction

It is one of the most subtle notions in concurrency.
For now, we should expect that

- compiler could aggressively reorder/eliminate field operations
- threads may see stale values, not the "newest" values written by other threads

unless there is "synchronization point" between two threads

Visibility

Introduction

It is one of the most subtle notions in concurrency.
For now, we should expect that

- compiler could aggressively reorder/eliminate field operations
- threads may see stale values, not the "newest" values written by other threads

unless there is "synchronization point" between two threads

- `Thread.start`
- `Thread.join`

Visibility

Incorrect: message passing via plain fields

```
static boolean jobDone = false;
public static void main(String... args) throws Exception {
    Thread t = new Thread(() -> { jobDone = true; });
    t.start();
    while (jobDone == false) { /* loop */ }
    t.join();
}
```

Visibility

Incorrect: message passing via plain fields

```
static boolean jobDone = false;
public static void main(String... args) throws Exception {
    Thread t = new Thread(() -> { jobDone = true; });
    t.start();
    while (jobDone == false) { /* loop */ }
    t.join();
}
```

May hang because:

- compiler optimization: `if (!jobDone) while(true);`
- stale value, `jobDone` remains `false` in main thread forever

Lecture plan

1. Motivation
2. Concurrency and parallelism
3. Threads and processes
4. Scheduler
5. Concurrent problems
   - Embarrassingly parallel problems and Amdahl's law
   - Race condition
   - Data race
   - Visibility
   - Deadlock
   - Priority inversion
6. Summary

Blocking methods

Wait-for graph

Blocking method: execution of current thread is *suspended* until some condition is met.

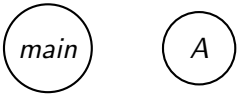
```
Thread A = new Thread(() -> {
    Thread B = new Thread(() -> { Thread.sleep(10_000); });
    Thread C = new Thread(() -> { B.join(); Thread.sleep(10_000); });
    B.start(); C.start(); B.join(); C.join()
}); A.start(); A.join();
```

Blocking methods

Wait-for graph

Blocking method: execution of current thread is *suspended* until some condition is met.

```
Thread A = new Thread(() -> {
    Thread B = new Thread(() -> { Thread.sleep(10_000); });
    Thread C = new Thread(() -> { B.join(); Thread.sleep(10_000); });
    B.start(); C.start(); B.join(); C.join()
}); A.start(); A.join();
```

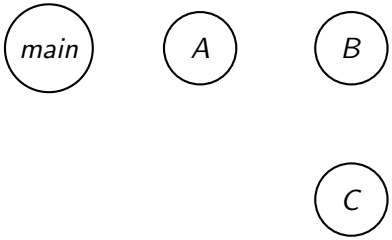


Blocking methods

Wait-for graph

Blocking method: execution of current thread is *suspended* until some condition is met.

```
Thread A = new Thread(() -> {
    Thread B = new Thread(() -> { Thread.sleep(10_000); });
    Thread C = new Thread(() -> { B.join(); Thread.sleep(10_000); });
    B.start(); C.start(); B.join(); C.join()
}); A.start(); A.join();
```

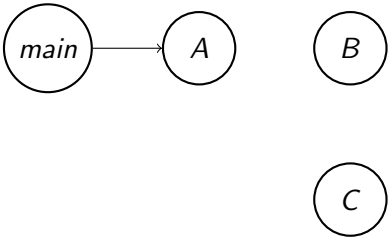


Blocking methods

Wait-for graph

Blocking method: execution of current thread is *suspended* until some condition is met.

```
Thread A = new Thread(() -> {
    Thread B = new Thread(() -> { Thread.sleep(10_000); });
    Thread C = new Thread(() -> { B.join(); Thread.sleep(10_000); });
    B.start(); C.start(); B.join(); C.join()
}); A.start(); A.join();
```

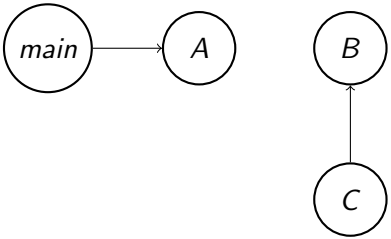


Blocking methods

Wait-for graph

Blocking method: execution of current thread is *suspended* until some condition is met.

```
Thread A = new Thread(() -> {
    Thread B = new Thread(() -> { Thread.sleep(10_000); });
    Thread C = new Thread(() -> { B.join(); Thread.sleep(10_000); });
    B.start(); C.start(); B.join(); C.join()
}); A.start(); A.join();
```

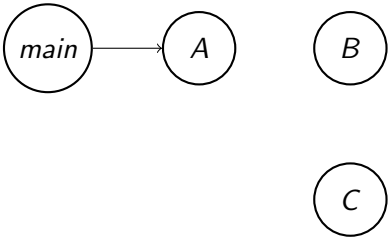


Blocking methods

Wait-for graph

Blocking method: execution of current thread is *suspended* until some condition is met.

```
Thread A = new Thread(() -> {
    Thread B = new Thread(() -> { Thread.sleep(10_000); });
    Thread C = new Thread(() -> { B.join(); Thread.sleep(10_000); });
    B.start(); C.start(); B.join(); C.join()
}); A.start(); A.join();
```



Blocking methods

Deadlock

Deadlock – cycle in wait-for graph.
Trivial single-threaded deadlock:

Blocking methods

Deadlock

Deadlock – cycle in wait-for graph.
Trivial single-threaded deadlock:

```
Thread.currentThread().join();
```

Blocking methods

Deadlock

Deadlock – cycle in wait-for graph.
Trivial single-threaded deadlock:

```
Thread.currentThread().join();
```

Classic two-thread deadlock:

```
static volatile Runnable lambda = null;
public static void main(String... args) throws Exception {
    Thread A = new Thread(() -> { lambda.run(); });
    Thread B = new Thread(() -> { A.join();     });
    lambda = () -> { B.join(); };
    A.start(); B.start();
    A.join(); B.join();
}
```

Blocking methods

Deadlock visualisation

```
java Sample &
PID="$!" && sleep 1
jstack -l $PID
```

Blocking methods

Deadlock visualisation

```
java Sample &
PID="$!" && sleep 1
jstack -l $PID
```

Output:

```
"main" #1 prio=5 os_prio=0 cpu=38,96ms elapsed=1,25s tid=0x00007fdcc0017000 nid
   java.lang.Thread.State: WAITING (on object monitor)
    at java.lang.Object.wait(java.base@14.0.1/Native Method)
    - waiting on <0x0000000095148930> (a java.lang.Thread)
    ...
    at Sample.main(Sample.java:10)
    ...
```

Blocking methods

Deadlock detection

- If API has at least one blocking method – extreme care required to avoid deadlocks
- If you do not know in which thread your code will be executed – extreme care required to avoid deadlocks
- If you can not control execution (inheritance, virtual methods, function pointers) – extreme care required to avoid deadlocks
- Design of any concurrent system should start with preparing tools for debugging and monitoring (observability API)

Blocking methods

Deadlock detection

- If API has at least one blocking method – extreme care required to avoid deadlocks
- If you do not know in which thread your code will be executed – extreme care required to avoid deadlocks
- If you can not control execution (inheritance, virtual methods, function pointers) – extreme care required to avoid deadlocks
- Design of any concurrent system should start with preparing tools for debugging and monitoring (observability API)

Homework, mail

Task 1.4 Run `jstack` on all deadlock samples from this lecture.

Question time

Question: Object-oriented languages love to "hide" implementation details using abstractions (inheritance, virtual methods, function pointers, interfaces). How would I know that user of my library/module will not call blocking method and introduce deadlock in my system?

Lecture plan

1. Motivation
2. Concurrency and parallelism
3. Threads and processes
4. Scheduler
5. Concurrent problems
   - Embarrassingly parallel problems and Amdahl's law
   - Race condition
   - Data race
   - Visibility
   - Deadlock
   - Priority inversion
6. Summary

Blocking methods

Priority inversion

```
Thread A = new Thread(() -> { do1(); B = startThread(); do2(); B.join(); });
A.start();
doCriticalWork();
A.join();
```

`main` waitsFor `A`, `A` waitsFor `B`.
Progress of `main` determined by progress of `B`.
"Soft" version of deadlock (depends-on graph).

- Some Operation Systems **do** have special support to remedy priority inversion problems
- Many Virtual Machines, concurrent libraries, multi-threaded frameworks **do not**

Blocking methods

Priority inversion

```
Thread A = new Thread(() -> { do1(); B = startThread(); do2(); B.join(); });
A.start();
doCriticalWork();
A.join();
```

`main` waitsFor `A`, `A` waitsFor `B`.
Progress of `main` determined by progress of `B`.
"Soft" version of deadlock (depends-on graph).

- Some Operation Systems **do** have special support to remedy priority inversion problems
- Many Virtual Machines, concurrent libraries, multi-threaded frameworks **do not**

If you launch rovers on Mars, be ready to debug them[15].

[15] https://www.cs.cornell.edu/courses/cs614/1999sp/papers/pathfinder.html

Summary

- We study agents with different speed that communicate with each other via shared memory
- Threads are parts of OS processes and are minimal unit of work for scheduler
- Tasks may be interrupted in pre-defined places or arbitrary allowing different levels of concurrency
- Independent tasks may be executed in parallel on different cores
- The perfect scenario: problem could be divided into many independent parts with almost zero coordination overheads
- Amdahl's law fundamentally limits possible speed-up of tasks with sequential parts
- Whenever you coordinate threads, you encounter race conditions, data races, visibility issues, deadlocks, priority inversion

Summary: homework

- answer "threads and exceptions" question (Task 1.1)
- solve amdahl's law puzzles (Task 1.2)
- `https://deadlockempire.github.io/` up to "Confused Counter" (Task 1.3)
- run `jstack` on deadlock samples (Task 1.4)