Introduction to Concurrent Programming

David Marchant
Based on slides by Troels Henriksen

2023-10-09

Inspired by slides by Randal E. Bryant and David R. O'Hallaron.

Why concurrency is hard and sometimes useful

The Unix model of concurrency

Memory model for threads

Mutexes

Scalability

What is concurrent programming?

From a programming perspective, *concurrency* means multiple *logical control flows* executing simultaneously.

Logical control flow

A stream of execution where the choice of what to do next is made by the code itself (i.e. this is pretty much all code you've written so far).

- **Concurrency is almost always present.**
  - E.g. the code rendering this slide runs *concurrently* with various system-level maintenance tasks, or perhaps a web browser.
  - These concurrent processes are isolated and do very different things.
  - Gets interesting when multiple control flows *interact*, e.g. by modifying shared data.

We need to use words deliberately

Before we even get into any programming or background, we need to establish two **key** words.

Parrallel

Two or more logical flows of events happening or existing at literally the same time.

Concurrent

Two or more logical flows of events that may happen in any order, and even be interleved.

We have already used these concepts

- We have already seen this with processes (see 2-3 lectures ago).
- We can have multiple processes executing concurrently, where part of one is progressing, then part of another.
- We can have two physical processors opperating in parallel, each at literally the same time.
- We can execute concurrently on a single processor, not in parallel. But 2 processors means we can execute 2 things in parallel.

The core idea

- This bit ain't complicated
- We have n many resources, not just one. We should be able to do n many things at once
- For instance a 4 core laptop should be doing 4 things in *parallel* (many more *concurrently*)

Single processing running for 10 minutes

Two processes each running for 5 minutes

As we will frequently see, this idealised result is seldom possible.

High level example of shared state

A

```
1  x = 1;
2  y = 2;
3  x = y + x;
```

B

```
1  x = 2;
2  y = 1;
3  x = y - x;
```

- If **A** runs first, then **B**: `x=-1,y=1`.
- If **B** runs first, then **A**: `x=3,y=2`.
- **But any interleaving is also possible:**
  - **A1**, **B1**, **B2**, **B3**, **A2**, **A3**: `x=4,y=2`
- Ordering is preserved *within* each control flow, but unpredictable across control flows due to scheduling.

We will return *frequently* to the issue of synchronisation and nondeterministic execution.

Motivation for concurrency: modularity

Sometimes multiple control flows is a natural way to express computation.

Examples

- **Video games:** distinct control flows for
  - Rendering graphics
  - Computing physics
  - AI for actors
  - Multiplayer communication
- **Browsers**
  - A control flow per tab.
  - Maybe a control flow per resource (images etc) when downloading page.
- **Network servers**
  - Control flow per user request.

Note

- This is useful even on small or old single-core processors.

Motivation for concurrency: performance

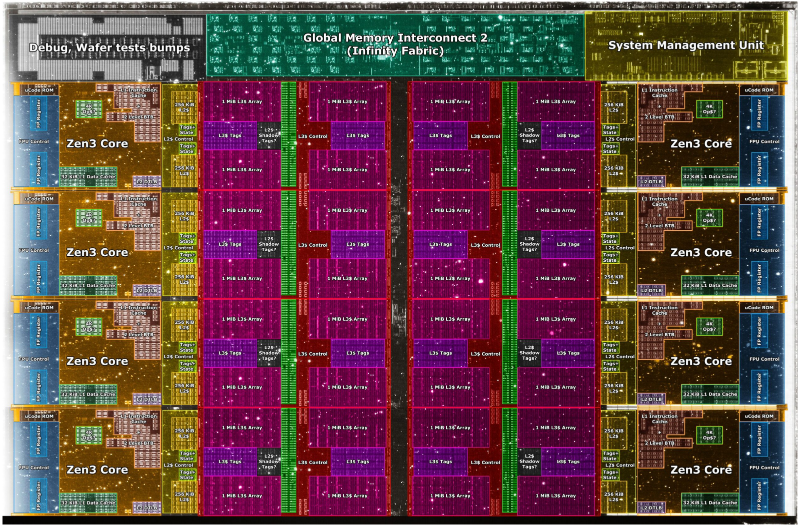
A single control flow occupies only a single *CPU core*.

CPU core

A piece of hardware that executes instructions—contains registers and one or more levels of cache.

- When we previously talked about "the CPU" we really meant "a CPU core".
- A modern CPU has several of these.
- **Each core executes a single logical control flow.**
- If we want to *fully utilize* the CPU (meaning: go fast), we need *a control flow per core*.

AMD Ryzen 5000 (Zen 3 architecture)



https://wccftech.com/amd-ryzen-5000-zen-3-vermeer-undressed-high-res-die-shots-close-ups-pictured-detailed/

Problem: concurrent programming is hard!

- The human mind is sequential.
- Thinking about all possible orderings of events in a concurrent system is at least error prone and usually impossible.

Problem: concurrent programming is hard!

- The human mind is sequential.
- Thinking about all possible orderings of events in a concurrent system is at least error prone and usually impossible.

The human brain is almost unchanged since this was the most complex problem it had to solve.

Classic problems of concurrent programming

Races: Outcome depends on arbitrary scheduling decisions elsewhere.

- **Example:** who gets the last seat on the airplane?

Classic problems of concurrent programming

Races: Outcome depends on arbitrary scheduling decisions elsewhere.

- **Example:** who gets the last seat on the airplane?

Deadlock: Resource allocation prevents forward progress.

- **Example:** traffic gridlock. (And programs generally cannot reverse!)

Classic problems of concurrent programming

Races: Outcome depends on arbitrary scheduling decisions elsewhere.

- **Example:** who gets the last seat on the airplane?

Deadlock: Resource allocation prevents forward progress.

- **Example:** traffic gridlock. (And programs generally cannot reverse!)

Starvation: External events or scheduling prevents forward progress.

- **Example:** someone always jumping ahead in line.
- Also known as *livelock* or *fairness*.

Classic problems of concurrent programming

Races: Outcome depends on arbitrary scheduling decisions elsewhere.

- **Example:** who gets the last seat on the airplane?

Deadlock: Resource allocation prevents forward progress.

- **Example:** traffic gridlock. (And programs generally cannot reverse!)

Starvation: External events or scheduling prevents forward progress.

- **Example:** someone always jumping ahead in line.
- Also known as *livelock* or *fairness*.

This is a field that has more terms for how things can go *wrong* than go *right*.

- By far the most difficult form of programming I know of.
- Our own brain is poorly suited for this kind of thinking.
  - We shall see C is not much better.
- Many aspects are outside the scope of CompSys.
  - But not all!

Why concurrency is hard and sometimes useful

The Unix model of concurrency

Memory model for threads

Mutexes

Scalability

We have already seen multiple control flows

- *Processes* are an example of concurrent execution.
  - Generally *just work* because they are isolated from each other.
  - Interleaved execution often not noticeable.
  - Process isolation makes close collaboration inefficient and awkward.
- **Instead we use *threads*.**

Our traditional model of a process

A process consists of three parts:

1. A *virtual memory space*.
   - Contains stack, code, heap, data, etc.
2. A *kernel context*.
   - PID, open files, signal mask, parent PID, list of children, etc.
3. An *execution context*.
   - Registers (including special ones like the program counter).

A multi-threaded process model

A process still consists of three parts:

1. A *virtual memory space*.
   - Contains stack, code, heap, data, etc.
2. A *kernel context*.
   - Process ID (PID), open files, signal mask, parent PID, list of children, etc.
3. One or more *threads*, each containing.
   3.1 An *execution context*
      - Registers (including special ones like the program counter).

   3.2 A *kernel thread context*.
      - Thread ID (TID), and a few other things that do not matter.

Processes consist of one or more threads!

Threads and sharing

Threads in the same process

- Each thread has its own logical control flow.
- Each thread has its own stack.
- Threads share open files.
- Threads share the same virtual memory space.
- They are *peers*, there is no "main thread".

- **Implication:** threads can interact by modifying memory.
  - Even unintentionally.

Threads contra processes

Similarities

- Each has its own logical control flow.
- Each can run concurrently with others (possibly also parallel).
- Each is context switched.

Differences

- Threads share code and data.
  - Processes typically do not.
- Threads are cheaper to create and maintain than processes.
  - Take with a grain of salt; both are plenty fast on Linux.
  - But switching between threads within a process does not require switching to a new virtual address space.

- **Each process has one or more threads.**
- **Each thread belongs to exactly one process.**

POSIX threads—standard thread interface on Unix

- Creating and reaping threads:
  - `pthread_create()`
  - `pthread_join()`
- Determining your thread ID:
  - `pthread_self()`
- Terminating threads:
  - `pthread_cancel()` (using this is usually a mistake)
  - `pthread_exit()` – terminates calling thread.
  - `exit()` – terminates *call* threads.
    - Implicit when `main()` returns.
- Synchronisation:
  - `pthread_mutex_init()`
  - `pthread_mutex_lock()`
  - `pthread_mutex_unlock()`

We will add a few more functions next lecture, but this is plenty to get in trouble.

Hello World in POSIX threads

Hello World in POSIX threads

```
#include <pthread.h>
#include <assert.h>
#include <stdio.h>

void* thread(void *arg) {
  int* p = arg;
  printf("Hello world! %d\n", *p);
  return NULL;
}

int main(void) {
  int x = 42;
  pthread_t tid;
  assert(pthread_create(&tid, NULL, thread, &x) == 0);
  assert(pthread_join(tid, NULL) == 0);
}
```

Why concurrency is hard and sometimes useful

The Unix model of concurrency

Memory model for threads

Mutexes

Scalability

Example program to illustrate sharing

```
char **ptr;
int cnt = 0;
int main() {
  pthread_t tid;
  char *msgs[2] = {
    "Hello␣from␣foo",
    "Hello␣from␣bar"
  };
  ptr = msgs;
  for (int i = 0; i < 2; i++)
    pthread_create(&tid,
                   NULL,
                   thread,
                   (void *)i);
  pthread_exit(NULL);
}
```

```
void* thread(void *vargp) {
  int j = (int)vargp;

  printf("%d:␣%s␣(cnt=%d)\n",
         j, ptr[j], ++cnt);
  return NULL;
}
```

- **Global variables**—*one instance*.
- **Local variables**—*one instance per function call*.
- Variables are shared if multiple threads reference the same instance.
- **Which variables are shared here?**

Example program to illustrate sharing

```
char **ptr;
int cnt = 0;
int main() {
  pthread_t tid;
  char *msgs[2] = {
    "Hello from foo",
    "Hello from bar"
  };
  ptr = msgs;
  for (int i = 0; i < 2; i++)
    pthread_create(&tid,
                   NULL,
                   thread,
                   (void *)i);
  pthread_exit(NULL);
}
```

```
void* thread(void *vargp) {
  int j = (int)vargp;

  printf("%d: %s (cnt=%d)\n",
         j, ptr[j], ++cnt);
  return NULL;
}
```

- **Global variables**—*one instance*.
- **Local variables**—*one instance per function call*.
- Variables are shared if multiple threads reference the same instance.
- **Which variables are shared here?**
  - `ptr`, `cnt`, `msgs`.

Synchronising threads

```c
  int n = atoi(argv[1]);
  pthread_t tid1, tid2;
  pthread_create(&tid1,
                 NULL,
                 thread,
                 &n);
  pthread_create(&tid2,
                 NULL,
                 thread,
                 &n);
  pthread_join(tid1, NULL);
  pthread_join(tid2, NULL);
  if (cnt != 2 * n)
    printf("Bad:  %d\n", cnt);
  else
    printf("Good: %d\n", cnt);
}
```

```c
int cnt = 0;

void *thread(void *vargp) {
  int n = *((int*)vargp);

  for (int i = 0; i < n; i++)
    cnt++;

  return NULL;
}
```

- Updates of shared variable `cnt` result in nondeterminism.
- But why?

Assembly code for loop

```
for (int i = 0; i < n; i++)
  cnt++;
```

```
         li t0, 0             # i = 0
   H     la t1, cnt           # address of cnt in t1
         beq t0, a0, done     # skip if nothing to do
loop:
   L     lw t2, 0(t1)         # load cnt from memory
   U     addi t2, t2, 1       # increment cnt
   S     sw t2, 0(t1)         # store cnt in memory
         addi t0, t0, 1       # i++
   T     beq t0, a0, done     # done?
         j loop               # another iteration
done:
```

Concurrent execution, when we are lucky

- **Any sequentially consistent interleaving is possible, and some give an unexpected result.**

i	instruction	`t2`$_0$	`t2`$_1$	`cnt`
0	H_0	?	?	0
0	L_0	0	?	0
0	U_0	1	?	0
0	S_0	1	?	1
1	H_1	1	1	1
1	L_1	1	1	1
1	U_1	1	2	1
1	S_1	1	2	2
1	T_1	1	2	2
0	T_0	1	2	2

Thread 0 critical section
Thread 1 critical section

Correct result!

Concurrent execution, when we are not so lucky

- **Any sequentially consistent interleaving is possible, and some give an unexpected result.**

i	instruction	`t2`$_0$	`t2`$_1$	`cnt`
0	H_0	?	?	0
0	L_0	0	?	0
0	U_0	1	?	0
1	H_1	1	0	0
1	L_1	1	0	0
0	S_0	1	0	1
0	T_0	1	1	1
1	U_1	1	1	1
1	S_1	1	1	1
1	T_1	1	1	1

Thread 0 critical section
Thread 1 critical section

Wrong result!

Critical sections and atomicity

Definition (Critical section)

A section of code that only a single thread must be executing at a time.

- The general principle is **mutual exclusion**.
- How do we ensure that critical sections are executed atomically?
- **Mutexes**!

Mutex API

```
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
```

- `pthread_mutex_t` is the pthreads *mutex type*.
- Mutexes have two states: *locked* and *unlocked*
- New mutexes start out *unlocked*.

```
int pthread_mutex_lock(pthread_mutex_t *mutex);
int pthread_mutex_unlock(pthread_mutex_t *mutex);
```

- Locking an a currently *unlocked* mutex *locks it* and returns.
  - The thread now *holds* the mutex.
- Trying to lock a currently *locked* mutex **blocks the calling thread**.
  - The thread now *waits* for the mutex.
  - When the mutex is unlocked (by some other thread), one *waiting* thread is allowed to lock the mutex.
- **Strong property:** when `pthread_mutex_lock()` returns (assuming no error), *the calling thread holds the mutex*.

Basic idea: Protecting critical sections with mutexes

- We associate each critical section with a mutex.
- **When entering the critical section:** lock the mutex.
- **When leaving the critical section:** unlock the mutex.

```
// shared mutex instance
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
```

```
pthread_mutex_lock(&mutex);
... critical section ...
pthread_mutex_unlock(&mutex);
```

Basic idea: Protecting critical sections with mutexes

- We associate each critical section with a mutex.
- **When entering the critical section:** lock the mutex.
- **When leaving the critical section:** unlock the mutex.

```
// shared mutex instance
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
```

```
pthread_mutex_lock(&mutex);
... critical section ...
pthread_mutex_unlock(&mutex);
```

- C doesn't have a language-level notion of "critical section", and typically it isn't *code* that we need to protect.
- In practice we associate mutexes with *shared variables*.
  - Sometimes a single mutex protects *multiple* variables.

Protecting the critical section in the counting program

```
pthread_mutex_t cnt_mutex = PTHREAD_MUTEX_INITIALIZER;
int cnt = 0;
```

```
for (int i = 0; i < n; i++) {
  pthread_mutex_lock(&cnt_mutex);
  cnt++;
  pthread_mutex_unlock(&cnt_mutex);
}
```

Gotta go fast

```c
int local_cnt = 0;
for (int i = 0; i < n; i++) {
  local_cnt++;
}

pthread_mutex_lock(&cnt_mutex);
cnt += local_cnt;
pthread_mutex_unlock(&cnt_mutex);
```

How mutexes are implemented

Real mutexes contain more features, but this captures the essence:

```
struct mutex {
  int locked; // 0 if unlocked, 1 if locked.
};

void lock(struct mutex *mutex) {
  while (mutex->locked != 0) {
    // try again
  }
  mutex->locked = 1;
}
```

Problem?

How mutexes are implemented

Real mutexes contain more features, but this captures the essence:

```
struct mutex {
  int locked; // 0 if unlocked, 1 if locked.
};

void lock(struct mutex *mutex) {
  while (mutex->locked != 0) {
    // try again
  }
  mutex->locked = 1;
}
```

Problem?

Not atomic.

The compare-and-swap (CAS) operation

```
int cas(int* p, int expected, int desired) {
  // This should be atomic!
  if (*p == expected) {
    *p = desired;
    return 0;
  } else {
    return 1;
  }
}
void lock(struct mutex *mutex) {
  while (1) {
    // try to switch from 0 to 1
    int switched = cas(&mutex->locked, 0, 1);
    if (switched) { break; }
  }
}
```

OK, but we still have the same problem.

Compare-and-swap

```c
int cas(int* p, int expected, int desired) {
  // This should be atomic!
  if (*p == expected) {
    *p = desired;
    return 0;
  } else {
    return 1;
  }
}
```

- This function cannot be implemented in C.
- In fact, is only possible if the architecture provides it as a primitive (as x86 does), or some even more basic primitive (RISC-V *conditional loads/stores*).

Compare-and-Swap in RISC-V

```
# a0 holds address of memory location
# a1 holds expected value
# a2 holds desired value
cas:  lr.w t0, (a0)       # load original value from a0 into t0
      bne  t0, a1, fail # value != expected, so fail
      sc.w t0, a2, (a0) # if (a0) untouched, write a2 to a0
      bnez t0, fail       # store-conditional failed, so CAS failed
      li   a0, 0          # set return to success
      jr   ra             # return
fail: li   a0, 1          # set return to failure
      jr   ra             # return
```

- Real implementations are actually a little more complicated.
- Students who wish to derail their life can study the details at `https://five-embeddev.com/riscv-isa-manual/latest/a.html`

Why concurrency is hard and sometimes useful

The Unix model of concurrency

Memory model for threads

Mutexes

Scalability

Speedup

Definition (Speedup in latency)

If T_1, T_2 are the runtimes of two programs P_1, P_2, then the *speedup in latency* of P_2 over P_1 is

$$\frac{T_1}{T_2}$$

Definition (Speedup in throughput)

If Q_1, Q_2 are the throughputs of two programs P_1, P_2, then the *speedup in throughput* of P_2 over P_1 is

$$\frac{Q_2}{Q_1}$$

Scalability

Definition (Strong scaling)

How the runtime varies with the number of processors for a fixed problem size.

Definition (Weak scaling)

How the runtime varies with the number of processors for a fixed problem size *relative to the number of processors*.

Definition (Amdahl's Law)

If p is the proportion of execution time that benefits from parallelisation, then $S(N)$ is maximum theoretical speedup achievable by execution on N threads, and is given by

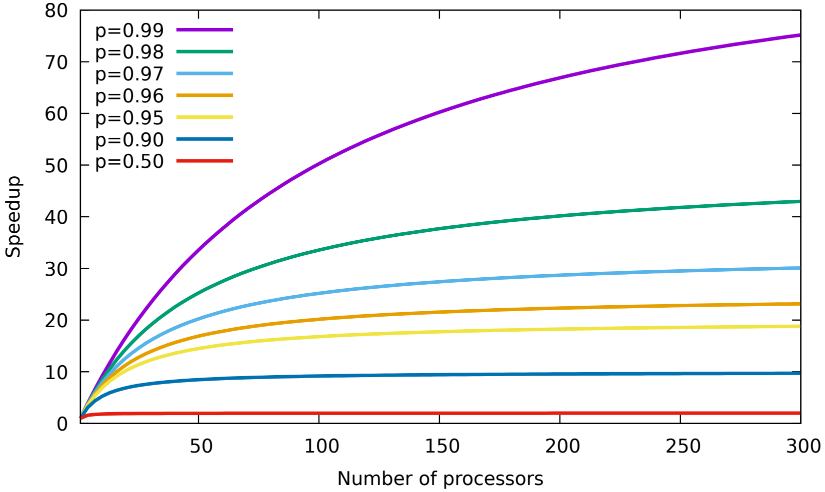
$$S(N) = \frac{1}{(1-p) + \frac{p}{N}}$$

Note that

$$S(N) \leq \frac{1}{1-p}$$

- Potential speedup by optimising part of system is bounded by proportion of part in overall runtime.
- **We should optimise the parts that take a long while to run.**
- Predicts *strong scaling*.

Amdahl's law is pessimistic



Gustafson's Law

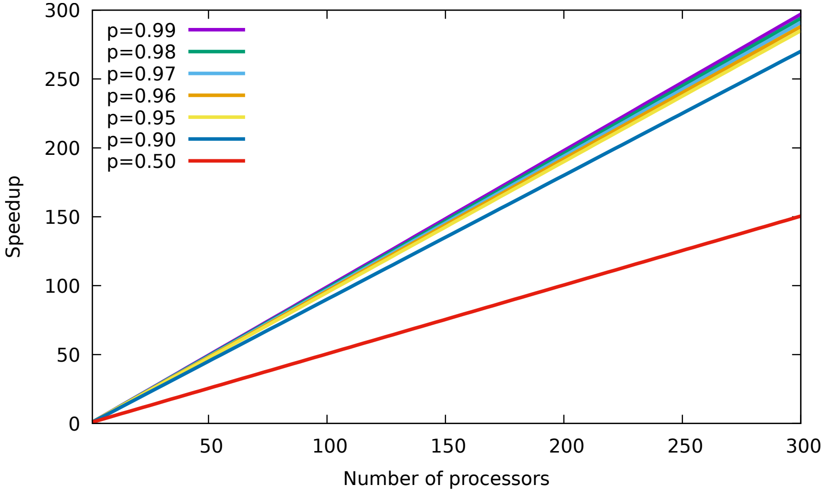
Definition (Gustafson's Law)

If $s = 1 - p$ is the proportion of execution time that must be sequential, then $S(N)$ is maximum theoretical speedup achievable by execution on N threads, and is given by

$$S(N) = N + (1 - N) \times s$$

- Predicts *weak scaling*.

Gustafson's law is optimistic



Summary

- Concurrency has two goals: modularity and parallel execution (performance).
  - The latter is increasingly important.
- Concurrency causes nondeterministic execution.
  - *Hard* to reason about.
  - The machine is certain to betray you.
- Use mutexes to ensure *mutual exclusion* to variables or critical sections.
  - Ultimately based on tiny operations that the hardware guarantees are atomic.