

Multithreading

Types of parallel computations

Plan

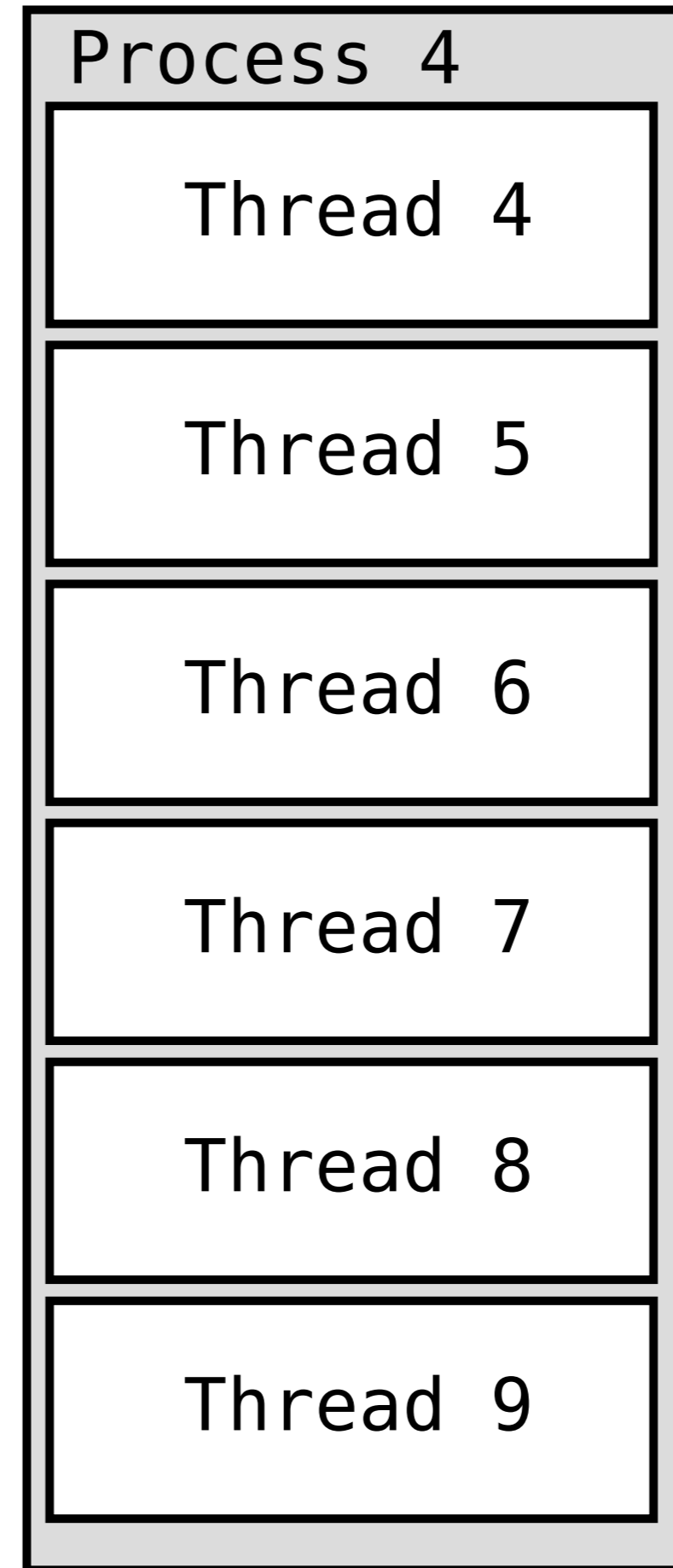
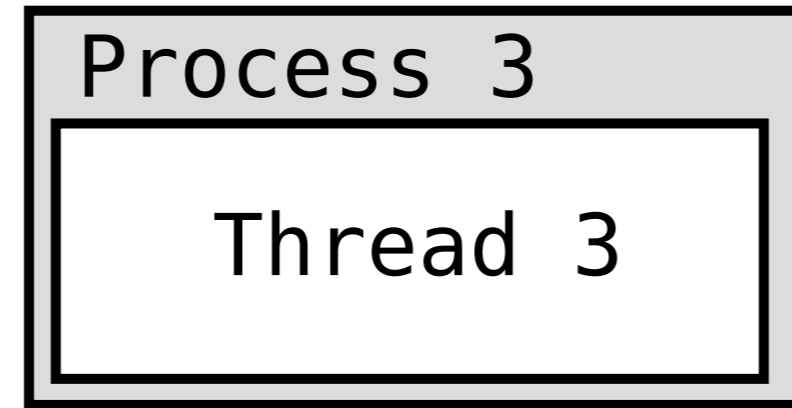
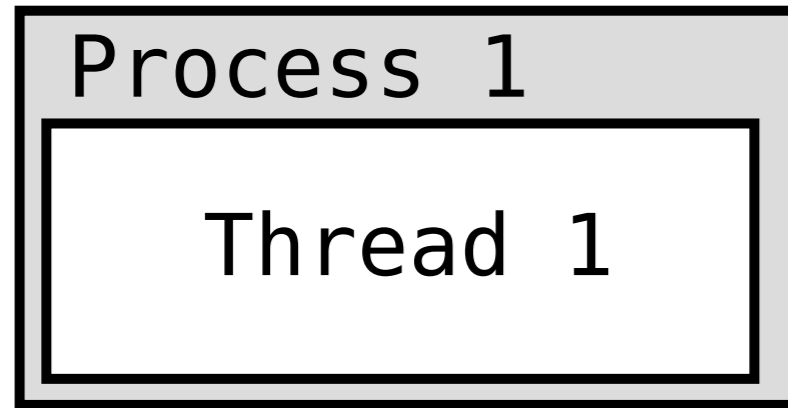
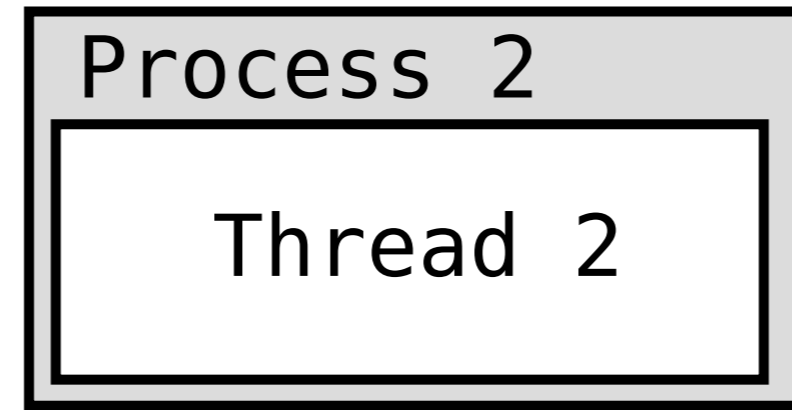
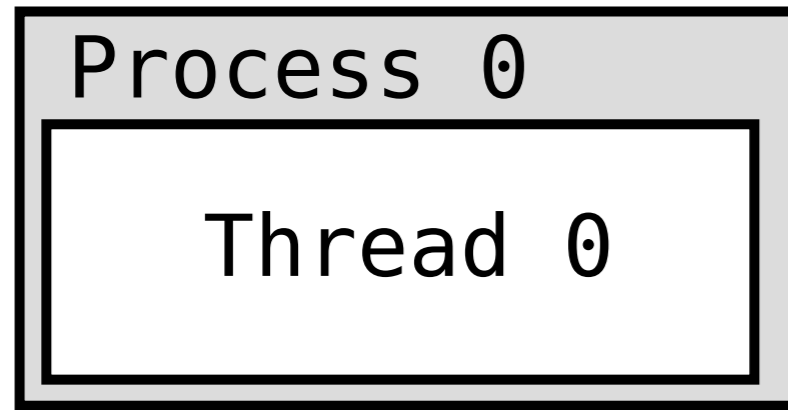
0. Parallelism that does not require programmer intervention
1. SIMD
2. Thread-level concurrency
3. Distributed computing
4. Hardware acceleration

2. Thread-level concurrency

Processes and threads

- When the OS runs an executable, it gets its own **process**
- A single executable (if run multiple times) can have multiple independent processes
- Memory is virtualized: **each process has its own view of the memory it owns**

- A process can create (“spawn”) multiple **threads**
- Like processes, each thread is an individual task from the point of view of the scheduler
- Within a process, **threads share a same view of the process memory**



- **Pro:** Communication between threads is extremely efficient
 - Just write something to memory,
 - let other threads read it through the same pointer
- **Con:** Because memory is shared, **synchronizing** threads is **very complex**
 - It is difficult for humans to reason about
 - The C language was not designed for multithreading
 - no semantics for concurrent read/write to shared variables until C11

Wrong code (1)

```
int ready = 0;    // one if there is some data in the buffer, zero otherwise
int buffer = 0;  // data in the buffer

// Every push()ed element must be pop()ed exactly once.
// - push() will block until the buffer is empty/available/"not ready"
// - pop() will block until the buffer is nonempty/"ready"
void push(int value)
{
    while (ready == 1) {
        // wait
    }

    buffer = value;
    ready = 1;
}

int pop()
{
    while (ready == 0) {
        // wait
    }

    ready = 0;
    return buffer;
}
```

The C compiler is free to reorder this:

```
void push(int value)
{
    while (ready == 1) {
        // wait
    }

    buffer = value;
    ready = 1;
}
```

into this:

```
void push(int value)
{
    buffer = value;

    while (ready == 1) {
        // wait
    }

    ready = 1;
}
```

```
while (ready == 1) {  
    // wait  
}
```

The C compiler can also notice that this loop has either

- zero iterations, or
- infinitely many iterations without side effects (UB!)

thus remove the loop!

The `volatile` keyword

An object that has `volatile`-qualified type [C23, p119]

- may be modified in ways `unknown to the implementation`
- or have other unknown `side effects`.

Example:

```
volatile int ready = 0;    // one if there is some data in the buffer, zero otherwise
volatile int buffer = 0;  // data in the buffer
```

Wrong code (2)

```
volatile int ready = 0;    // one if there is some data in the buffer, zero otherwise
volatile int buffer = 0;  // data in the buffer

void push(int value)
{
    while (ready == 1) {
        // wait
    }

    buffer = value;
    ready = 1;
}

int pop()
{
    while (ready == 0) {
        // wait
    }

    ready = 0;
    return buffer;
}
```

Thread 0

```
int pop()
{
    while (ready == 0) {
        // wait
    }

    ready = 0;

    return buffer;
}
```

// ready = 1 buffer = 'A'

// ready = 1 buffer = 'A'
// ready = 0 buffer = 'A'

// ready = 0 buffer = 'B'
// ready = 1 buffer = 'B'

Thread 1

```
void push(int value)
{
    while (ready == 1) {
        // wait
    }

    buffer = value;
    ready = 1;
}
```

// push('B')

Wrong code (3)

```
volatile int ready = 0;      // one if there is some data in the buffer, zero otherwise
volatile int buffer = 0;    // data in the buffer

void push(int value)
{
    while (ready == 1) {
        // wait
    }

    buffer = value;
    ready = 1;
}

int pop()
{
    while (ready == 0) {
        // wait
    }

    int b = buffer;
    ready = 0;
    return b;
}
```

Thread 0

```
void push(int value) // push('A')
{
    // ready = 0    buffer = 'X'
    while (ready == 1) {
        // wait
    }

    // ready = 0    buffer = 'X'

    // ready = 0    buffer = 'B'
    // ready = 1    buffer = 'B'

    buffer = value;    // ready = 1    buffer = 'A'
    ready = 1;        // ready = 1    buffer = 'A'
}
```

Thread 1

```
void push(int value) // push('B')
{
    while (ready == 1) {
        // wait
    }

    buffer = value;
    ready = 1;
}
```

Solution

- **low-level:** compiler intrinsics for “**atomic**” operations:
combined operations that are performed as a single unit
no thread will ever see the memory in an intermediate state
- **high-level:** use libraries that correctly implement some primitives:
locks, queues, etc.
 - C11 threads
 - Win32 threads (Windows)
 - Posix threads (“pthreads”; Linux, MacOS)
 - OpenMP (Open Multi-Processing; higher-level, portable)

C11 threads

- Introduced in C11
- Was initially **defective** and thus amended in C17
- Support is optional for compilers writers
- Very little documentation besides the C standard

Other multithreading APIs

- Win32 threads: [official documentation](#).
- Posix threads: [official documentation](#), [current reference](#).
- OpenMP: [tutorials](#), [specification](#).

Posix threads

Using with `libpthread`

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

Link with `-lpthread`:

```
clang -o executable obj0.o obj1.o obj2.o -lpthread
```

Creating threads

```
#include <pthread.h>

typedef void *thread_fn_t(void *arg);

int pthread_create(pthread_t *thread, const pthread_attr_t *attr, thread_fn_t *fn, void *arg);
```

- `thread` is a pointer to a `pthread_t` (in practice: an integer of some size).
On success, it is set to a thread identifier (for the just-created thread).
- `attr` allows overriding default parameters (stack size and address, thread priority, ...).
Set to `NULL` for defaults.
- `fn` is a pointer to a function that will run the thread code.
(When the function returns, the thread terminates.)
- `arg` is a pointer passed as an argument to `fn`.
(Allows differentiating threads if they run the same `fn`).

```
#include <pthread.h>

typedef void *thread_fn_t(void *arg);

int pthread_create(pthread_t *thread, const pthread_attr_t *attr, thread_fn_t *fn, void *arg);
```

Example:

```
void *thread_fn(void *arg)
{
    int data = *((int *)arg);
    printf("Thread %d\n", data);
    return NULL;
}

int main()
{
    int data0 = 0;
    int data1 = 1;
    pthread_t id0, id1;

    if (pthread_create(&id0, NULL, thread_fn, &data0))
        return 1;

    if (pthread_create(&id1, NULL, thread_fn, &data1))
        return 1;

    // ...
}
```

Terminating threads

```
void pthread_exit(void *retval);
```

- `pthread_exit()` is an alternative to returning from the thread function
- It exits as if the thread function returned `retval`

Waiting for a thread to terminate

```
int pthread_join(pthread_t thread, void **retval);
```

We pass:

- the thread identifier `thread`, and
- a pointer to a void pointer `retval`.

(the void pointer will be set to the return value of the thread function)

```

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <pthread.h>

int x = 0;

void *mythread(void *arg)
{
    x = 1;
    printf("Child process set x=1\n");
    return NULL;
}

int main(int argc, char *argv[])
{
    pthread_t tid;
    int e0 = pthread_create(&tid, NULL, mythread, NULL);

    if (e0) {
        fprintf(stderr, "pthread_create(): %s\n", strerror(e0));
        exit(EXIT_FAILURE);
    }

    void *vp;
    int e1 = pthread_join(tid, &vp);

    if (e1) {
        fprintf(stderr, "pthread_join(): %s\n", strerror(e1));
        exit(EXIT_FAILURE);
    }

    printf("Returned pointer: %p\n", vp);
    printf("Parent process sees x=%d\n", x);
    return EXIT_SUCCESS;
}

```

```

Child process set x=1
Returned pointer: (nil)
Parent process sees x=1

```

Synchronizing primitives

- mutex
- condition variable

Mutexes

Mutexes

“**Mutex**” stands for mutual exclusion.

Mutexes have two states:

- **locked** and “owned” by a specific thread
- **unlocked**

Attempting to lock an already-locked mutex **blocks** until it is unlocked first

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

- `pthread_mutex_lock()` locks the given mutex
 1. if `mutex` was unlocked,
it becomes locked and owned by the calling thread
returns immediately
 2. if `mutex` is already locked by another thread
suspends the calling thread until `mutex` is unlocked.
when `mutex` is unlocked, see a.
- `pthread_mutex_unlock()` unlocks the given mutex
(currently locked and owned by the calling thread)

Both functions return zero in normal operation.

Example

```
char shared_buffer[16777216];
pthread_mutex_t shared_buffer_mutex = PTHREAD_MUTEX_INITIALIZER;

void *thread_function(void *arg)
{
    while (1) {
        if (pthread_mutex_lock(&shared_buffer_mutex))
            abort();

        process_data(shared_buffer);

        if (pthread_mutex_unlock(&shared_buffer_mutex))
            abort();
    }

    return NULL;
}
```

Implementation of blocking functions

Busy waiting:

```
while (*mutex_locked == 0) { }
```

- Best possible **latency**
- Wastes **resources**:
 - Keeps a CPU busy, no other thread can run
 - CPU consumes power and generates heat while doing nothing

System call:

Kernel scheduler suspends thread and either puts CPU to sleep, or runs another thread

- High **latency**
 - direct costs: 1000+ cycles (“context switch”: restoring registers & usermode page table)
 - indirect costs: pipeline and cache invalidation

- Best **resource usage** (if not too frequent)

Consider the following task:

- We have a shared buffer:

```
pthread_mutex_t buff_mutex = PTHREAD_MUTEX_INITIALIZER;  
char *buff_data;  
size_t buff_size;  
size_t buff_used;
```

- We have a thread with a single mission:
When the buffer is full, it calls a function `buffer_flush()`

```
pthread_mutex_t buff_mutex = PTHREAD_MUTEX_INITIALIZER;
char *buff_data;
size_t buff_size;
size_t buff_used;
```

```
void *flusher_thread(void *arg)
{
    while (1) {
        // wait until buffer is full
        while (1) {
            if (pthread_mutex_lock(&buff_mutex))
                abort();

            if (buff_used == buff_size) {
                // stop waiting, while owning the mutex
                break;
            }

            if (pthread_mutex_unlock(&buff_mutex))
                abort();
        }

        // flush buffer (while owning the mutex)
        buffer_flush();

        // unlock mutex
        if (pthread_mutex_unlock(&buff_mutex))
            abort();
    }

    return NULL;
}
```

- This implementation has high *lock contention*:
flusher_thread() will keep trying own buff_mutex all the time
- If it succeeds in doing so without waiting, it is essentially **busy waiting**.

Solution: **condition variables**

Condition variables

Condition variables

A **condition variable** allows efficiently **blocking until an arbitrary condition** becomes true.

Other threads cooperatively **signal** it when the condition **may** have become true.

```
pthread_cond_t cond = PTHREAD_COND_INITIALIZER;  
  
int pthread_cond_wait(pthread_cond_t *cond, pthread_mutex_t *mutex);  
int pthread_cond_signal(pthread_cond_t *cond);
```

- `pthread_cond_wait()`:
 - unlocks mutex
 - waits for cond to be signaled
 - re-locks mutex
- `pthread_cond_signal()` restarts one thread waiting cond
 - if no thread is waiting, nothing happens
 - if several threads are waiting, exactly one is restarted (it is not specified which)

```
pthread_mutex_t buff_mutex = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t buff_cond = PTHREAD_COND_INITIALIZER;
char *buff_data;
size_t buff_size;
size_t buff_used;
```

```
void *flusher_thread(void *arg)
{
    while (1) {
        // wait until buffer is full
        if (pthread_mutex_lock(&buff_mutex))
            abort();

        while (buff_used < buff_size) {
            if (pthread_cond_wait(&buff_cond, &buff_mutex))
                abort();
        }

        // flush buffer (while owning the mutex)
        buffer_flush();

        // unlock mutex
        if (pthread_mutex_unlock(&buff_mutex))
            abort();
    }

    return NULL;
}
```

```
void *filler_thread(void *arg)
{
    while (1) {
        // lock mutex
        if (pthread_mutex_lock(&buff_mutex))
            abort();

        // fill buffer
        size_t old_used = buff_used;

        buffer_fill();

        if (buff_used > old_used) {
            if (pthread_cond_signal(&buff_cond))
                abort();
        }

        // unlock mutex
        if (pthread_mutex_unlock(&buff_mutex))
            abort();
    }

    return NULL;
}
```

Resources

Resources

- “Perf Book”:
Is Parallel Programming Hard, And, If So, What Can You Do About It?
<https://mirrors.edge.kernel.org/pub/linux/kernel/people/paulmck/perfbook/perfbook.html>
- Book about (among other things) concurrency:
Operating Systems: Three Easy Pieces
<https://pages.cs.wisc.edu/~remzi/OSTEP/>
- pthreads tutorial:
<https://www.cs.cmu.edu/afs/cs/academic/class/15492-f07/www/pthreads.html>

Tutorial 6

Tutorial 6

1. Write a multithreaded C program in which each thread does the following in a loop: (a) print the value of a global integer x , then (b) increment the value of x . Do not use synchronisation primitives.
2. Add a synchronisation primitive to the above program so that the printing and incrementing of x in one thread is viewed as atomic from the others.
3. Enable multithreading in the OpenBLAS implementation of square fp32 matrix multiplication of Tutorial 5.
4. Write a multithreaded version of the filtering exercise in Tutorial 5.