Multitasking with D

Ali Çehreli

DConf 2016, May 4 • Berlin, Germany
Contents

- Definitions
- (Parallelism)
- CPU and OS internals
- Fibers
- Concurrency
- Asynchronous input and output
Confusing related terms

- Multitasking
- Concurrency
- Parallelism
- Multithreading
- Fibers (coroutine, green thread, greenlet, light-weight thread, etc.)
- etc.
Multitasking

Performing multiple tasks, not sequentially (i.e. concurrently, likely in an interleaved fashion).

Multitasking is

• *not* parallelism
  ◦ *not* data parallelism (SIMD)
  ◦ *not* instruction-level parallelism (CPU pipelining)
  ◦ *not* memory-level parallelism (CPU cache, TLB, prefetching, etc.)

• *not* multithreading (but uses threads)

```c
// An impractical and sub-optimal multitasking example
task_1_step_1();
task_2_step_1();
task_1_step_2();
task_2_step_2();
// ...
```
Parallelism

Executing operations simultaneously to make the program run faster.
Especially good for *embarrassingly parallel* operations.
std.parallelism.parallel

If the following takes 4 seconds

```plaintext
auto images = [ Image(1), Image(2), Image(3), Image(4) ];

foreach (image; images) {
    // ... lengthy operations ...
}
```

The following takes 1 second on 4 cores

```plaintext
import std.parallelism;

foreach (image; images.parallel) {
    // ... lengthy operations ...
}
```
std.parallelism module

- **parallel**: Operates on a range in parallel; good with `foreach` with lengthy operations
- **asyncBuf**: Iterates a range semi-eagerly in parallel; good with range algorithms with lengthy iterations
- **map**: Operates on a range semi-eagerly in parallel
- **amap**: Operates on a range eagerly in parallel
- **reduce**: Does calculations on a range eagerly in parallel
- **task**: Creates tasks to be executed in parallel (blurs the parallelism-concurrency boundary)
Operating system and CPU internals

- Call stack
- CPU registers IP and SP
- Thread
- CPU caches
- MMU and TLB
Call stack

Stack frame: Local state of a function call

Call stack: Stack frames of all currently active function calls (aka stack)

```c
void main() {
    int a;
    int b;
    int c = foo(a, b);
}

int foo(int x, int y) {
    bar(x + y);
    return 42;
}

void bar(int param) {
    string[] arr;
    // ...
}

The call stack grows as function calls get deeper.
```

![Diagram showing stack frames with variables and function calls]
Call stack is especially useful in recursion

The call stack takes care of execution state automatically.

```plaintext
import std.array;

int sum(int[] arr, int currentSum = 0) {
    if (arr.empty) {
        return currentSum;
    }
    return sum(arr[1..$], currentSum + arr.front);
}

void main() {
    assert(sum([1, 2, 3]) == 6);
}
```

**Note:** Use `std.algorithm.sum` instead.

**Note:** "Tail-call optimization" can eliminate stack frames.
CPU registers

Ultimately, everything happens on CPU registers.

- Instruction pointer (IP): what to execute next (aka program counter (PC))
- Stack pointer (SP): the local context
- ...
- ...

... more (usually dozens) ...

Plug: Even the Mill, a revolutionary CPU with no conventional general-purpose registers, have equivalents of IP and SP: http://millcomputing.com/
Thread

An execution context:
• IP register determines the execution
• SP register determines the context (other pieces are involved as well)

A simplification of a thread for the rest of this presentation:
Two threads

Two processes to be executed concurrently:

```plaintext
// A
import std.stdio;
void main() {
    writeln("Hello, world.");
}
```

```plaintext
// B
import std.stdio;
void main() {
    writeln("Hello, Mars.");
}
```

The OS loads each process into memory and allocates a stack for each:

Memory:

```
... ───┬────────────┬─── ··· ───┬────────────┬─── ···
     ▼                        ▼
    ┌────┐   │                        │                 ┌────┐
 A's main │ IP │───┘                        └─────────────────│ IP │ B's main
 thread │ SP │─┐                                          ┌─│ SP │ thread
      └────┘ │                                          │ └────┘
            │                │    │
            │                │    │
            │    │                │    │
            │    │                │    │
            └────┤                ├────┤
                └──────▶          └────┘
                      A's stack     B's stack
```

Three threads

Two processes, three threads:

// A
import std.stdio;
import std.concurrency;

void greetMoon() {
    writeln("Hello, moon.");
}

void main() {
    spawn(&greetMoon);
    writeln("Hello, world.");
}

// B
import std.stdio;

void main() {
    writeln("Hello, Mars.");
}

A's main thread IP SP

A's greetMoon thread IP SP

B's main thread IP SP
OS concurrency

Potentially thousands of threads on e.g. 4 cores:

The OS uses special thread scheduling algorithms relying on

- Process priority
- Thread priority
- IO-bound versus CPU-bound
- Time-slice fully used last time or not (Linux)
- Process foreground versus background (Windows)
- etc.
OS thread scheduler

The **goal**: No core should be idle if there are runnable threads.

*(A number of performance issues with the Linux scheduler has recently been reported. (See "The Linux Scheduler: a Decade of Wasted Cores" by Lozi and others.))*

Each thread is placed on a core and given a slice of time to run:

- Either it uses the entire time slice before being *preempted*
- Or stops early because it is
  - waiting for IO
  - waiting for a synchronization primitive
  - paused intentionally
Partially unused time slice

Performance issue: Actual execution time is abandoned.
CPU and its caches

An imaginary 4-core CPU with 3 levels of hierarchical cache.

c1, c2, c3, c4: Cores
- Orange: Level 1 cache, ~1 clock cycle
- Brown: Level 2 cache, ~20 clock cycles
- Maroon: Level 3 cache, ~80 clock cycles
- Black: Physical memory, ~200 clock cycles
Virtual memory

Every process (program) sees memory as a contiguous storage space (e.g. all of the 64-bit space of a 64-bit CPU):

\[0x0000_0000_0000_0000 - 0xFFFF_FFFF_FFFF_FFFF\]

<table>
<thead>
<tr>
<th>Process</th>
<th>Variable</th>
<th>Virtual address</th>
<th>Physical address</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>x</td>
<td>0x1000</td>
<td>0x1234</td>
</tr>
<tr>
<td>B</td>
<td>y</td>
<td>0x1000</td>
<td>0x5678</td>
</tr>
</tbody>
</table>

This requires a *translation* at runtime

- from virtual addresses
- to physical addresses
Memory management unit (MMU)

- Accesses memory for the CPU
- Does virtual-to-physical address translation

Virtual-to-physical translation table is too large to be on-chip; may even be swapped to disk. What is on-chip is the TLB.

Translation lookaside buffer (TLB), ~1 clock cycle hit, ~100 clock cycles miss
Context switch

Placing another thread on a core (i.e. replacing IP and SP with a different thread's)

Reasons:

• Thread consumed the entire time slice (good!)
• Waiting for input or output (IO)
• Waiting for exclusive access to a piece of critical code section (e.g. locking a mutex)
• Paused intentionally

Potential performance issues:

• Part of execution time-slice may be unused
• CPU's instruction and data caches may be flushed
• TLB may be flushed
Fibers
Same fringe problem

"Two binary trees have the same fringe if they have exactly the same leaves reading from left to right." Richard P. Gabriel at http://www.dreamsongs.com/10ideas.html

"Write a samefringe program that does not consume a lot of storage."

With apologies, changing the problem to same elements in in-order traversal:

```
Tree A          Tree B
          2               4
         / \             / \  
        1   4           2   5
       / \         / \  
      3   5   1   3
```
Recursive tree traversal

Thanks to call stack, traversing a binary tree is easy and elegant:

```c
void traverse(const Node * node, Func func) {
    if (!node) {
        return;
    }
    traverse(node.left, func);
    func(node.element);
    traverse(node.right, func);
}
```

What if there are two trees?
Surprising complexity

Implementing a range (or iterator) type for a tree is very hard especially considering how trivial it is with recursion.

```cpp
struct Tree {
    // ...

    struct InOrderRange {
        // ... What should the implementation be? ...
    }

    InOrderRange opSlice() const {
        return InOrderRange(root);
    }
}
```

Some tree iterator implementations require an additional `Node*` to point at the parent node.
Cooperative multitasking

Context switch performed by an OS thread

Time slice:

main thread starts running

fiber starts running

main thread

fiber

main thread
Fiber operations

• A fiber (and its call stack) starts with a callable entity taking no parameter, returning nothing:

```c++
void fiberFunc() { /* ... */ }
```

• Can be created as an object of the `core.thread.Fiber` class hierarchy:

```c++
auto fiber = new Fiber(&fiberFunc);
```

• Started and resumed by its `call()` member function:

```c++
fiber.call();
```

• Pauses itself by `Fiber.yield()`:

```c++
void fiberFunc() { /* ... */ Fiber.yield(); /* ... */ }
```

• The execution state of a fiber is determined by its `.state` property:

```c++
if (fiber.state == Fiber.State.TERM) { /* ... */ }
```
User threads

Context switch is the same: replace IP, SP, and a few others.
As fast as a function call (almost).
CPU cache, TLB, etc. are not disturbed.
import core.thread;

void fibonacciSeries(ref int current) {
    current = 0;  // Note: 'current' is the parameter
    int next = 1;

    while (true) {
        Fiber.yield();

        /* Next call() will continue from this point */

        const nextNext = current + next;
        current = next;
        next = nextNext;
    }
}

void main() {
    int current;
    Fiber fiber = new Fiber(() => fibonacciSeries(current));

    foreach (_; 0 .. 10) {
        fiber.call();

        import std.stdio;
        writef("%s ", current);
    }
}
import std.stdio;
import std.range;
import std.concurrency;

/* Resolve the name conflict with std.range.Generator. */
alias FiberRange = std.concurrency.Generator;

void fibonacciSeries() {
    int current = 0;  // <-- Not a parameter anymore
    int next = 1;

    while (true) {
        yield(current);

        const nextNext = current + next;
        current = next;
        next = nextNext;
    }
}

void main() {
    auto series = new FiberRange!int(&fibonacciSeries);
    writeln("%(%s %)", series.take(10));
}
Recursive tree traversal with a fiber

The only difference is \texttt{yield()} and the \texttt{func} parameter disappears:

```c
void traverse(const (Node) * node) {
  if (!node) {
    return;
  }
  traverse(node.left);
  \textbf{yield}(node.element);
  traverse(node.right);
}
```

Now there can be any number of trees, iterated any level deep.
D features that help with concurrency

- Thread-local by default; `shared, immutable, __gshared`
- Garbage collector
- Synchronization
  - `synchronized`
  - `core.sync`
- `cas, atomicOp, and others`
- `core.thread`
- `std.concurrency`
- Fibers
Sharing mutable data is problematic. In D, global and static data are thread-local by default.

• Must define data as `shared` to share data
• `immutable` is automatically shared

```d
int a;       // mutable but not shared
shared(int) b; // shared mutable (careful!)
immutable(int) c; // immutable and implicitly shared
__gshared int d; // C-style mutable global (careful!)
```

`shared` and `immutable` are overloadable function attributes
Garbage collector

No need to manage lifetimes with reference counting, etc.

```plaintext
import std.concurrency;
import std.random;
import std.range;

void worker() {
    for (; ;) {
        receive(
            (immutable(int[]) arr) {
                // ...
            });
    }
}

int[] producer(int n) pure {
    return iota(n).array;
}

void main() {
    auto w = spawn(&worker);
    foreach (_, 0 .. 100) {
        immutable arr = producer(uniform(10, 100));
        w.send(arr);
    }
}
```
Synchronization

Useful features but these involve waiting, which better be avoided:

```java
// Critical section
synchronized {
    // ...
}
```

Deadlock prevention by automatic ordering of locks:

```java
synchronized (lockA, lockB) {
    // ...
}
```

Also see:

- `core.sync.barrier`
- `core.sync.condition`
- `core.sync.mutex`
- `core.sync.rwmutex`
- `core.sync.semaphore`
Direct modification of shared data is deprecated

```plaintext
import core.thread;
import std.stdio;
import std.concurrency;

shared(int) i;

void incrementor(size_t n) {
    foreach (_) 0 .. n {
        ++i;       // deprecated and wrong
    }
}

void main() {
    foreach (_) 0 .. 100 {
        spawn(&incrementor, 1_000_000);
    }
    thread_joinAll();
    writeln(i);
}
```

Deprecation: read-modify-write operations are not allowed for shared variables. Use `core.atomic.atomicOp!"+"(i, 1)` instead.
core.atomic.atomicOp

shared(int) i;  
// ...

    ++i;              // deprecated and wrong

import core.atomic;  
// ...

    atomicOp!"+="(i, 1);  // correct

Also see atomicStore, atomicLoad, etc.
core.atomic.cas

Compare-and-swap enables lock-free mutations:
1. Get the current value
2. Attempt to mutate if it has not been changed since step 1
3. Repeat from step 1 if unsuccessful

```c
int current_i;

do {
    current_i = i;
} while (!cas(&i, current_i, current_i + 1));
```

Meaning: "Set to \texttt{current\_i + 1} if it still has the value \texttt{current\_i}".

\texttt{cas} enables lock-free data structures. (See Tony Van Eerd's entertaining "Lock-free by Example" presentation to see how difficult it is to achieve.)

Issue: \texttt{cas} supports up-to 128-bit data; so, bit-packing can be used to mutate more than one data atomically.
Message-passing; a manageable form of concurrency but can be slow because `receive()` waits. (Also see `receiveTimeout()`.)

```
import std.concurrency;

void main() {
    auto worker = spawn(&func);

    worker.send(42);        // note different types of messages
    worker.send("hello");
    worker.send(Terminate());
}

struct Terminate {}

void func() {
    bool done = false;

    while (!done) {
        receive(
            (int msg) { /* ... */ },
            (string msg) { /* ... */ },
            (Terminate msg) { done = true; });
    }
}```
core.thread.Thread

Should be avoided because this is too low-level. Likely, you will invent `std.parallelLims, std.concurrency`, event loop, etc.

```cpp
auto worker = new Thread(&foo).start;
```
IO handling

Input and output can be a lot slower than other operations. Waiting for IO completion kills performance.

- Blocking synchronous
- Non-blocking synchronous; returns immediately but the result may or may not be ready (e.g. `read()` may return less than the requested number of bytes)
- Asynchronous; result is handled when IO is complete
Event loop

A single-thread that waits for events and then dispatches their handlers.

- Reactor pattern; synchronous
  - The callback is for an event (e.g. "there is data")
  - Event loop calls the callback and the callback does the read
- Proactor pattern; asynchronous, better
  - The callback is for completion
  - The OS does the read and calls the callback when it completes
libasync

"written completely in D, features a cross-platform event loop and enhanced connectivity and concurrency facilities for extremely lightweight asynchronous tasks"

http://code.dlang.org/packages/libasync

Used by vibe.d and asynchronous
vibe.d framework

Has **everything** (everything!)

"Asynchronous I/O that doesn’t get in your way, written in D"

http://vibed.org/
asynchronous library

"provides infrastructure for writing concurrent code using coroutines, multiplexing I/O access over sockets and other resources, running network clients and servers, and other related primitives"

"implements most of the python 3 asyncio API"

"is a library and not a framework"

http://code.dlang.org/packages/asynchronous
More asynchronous libraries

• **collie**: An asynchronous event-driven network framework written in D.
  http://code.dlang.org/packages/collie

• **future**: "asynchronous return values and related functionality"
  http://code.dlang.org/packages/future

• **simple_future**: "Simple asynchronous functions"
  http://code.dlang.org/packages/simple_future

• etc.