Multiplix: using D for kernel development

Zachary Yedidia

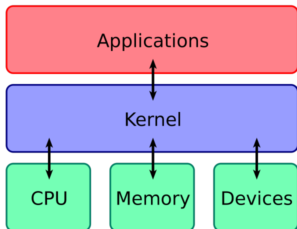
Stanford University

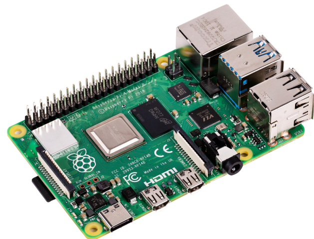
Why build a kernel?

It is fun and educational to build a kernel.

I am a researcher, and want to experiment with new ideas and architectures for operating system design. Having a small kernel to experiment with is helpful.

Cheap single-board computers (SBCs) are widely available and fun to play with!





Multiplix: how to write a small kernel in D

Multiplix is a small Unix-like kernel I have been developing over the last 10 months.

- Around 6,500 lines of D.
- Supports multiple architectures (ARMv8, RISC-V) and boards (Raspberry Pi, VisionFive).
- It will serve as the foundation for future projects I am interested in.

Multiplix: how to write a small kernel in D

Multiplix is a small Unix-like kernel I have been developing over the last 10 months.

- Around 6,500 lines of D.
- Supports multiple architectures (ARMv8, RISC-V) and boards (Raspberry Pi, VisionFive).
- It will serve as the foundation for future projects I am interested in.

In this talk:

Part 1: Why I chose D and how to get started with it for bare-metal programming.

Part 2: My experiences working on Multiplix.

Part 3: Future directions and what I intend to build on top of Multiplix next.

Why D for bare-metal programming?

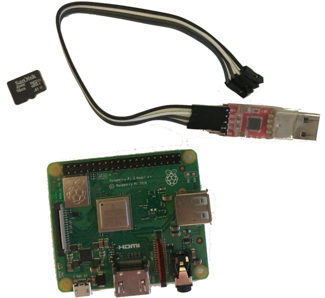
A crowded market of systems languages: D, C, C++, Rust, Zig, Hare, and more...

Familiar: converting and interfacing with existing C code is easy.
Ergonomic: particularly compared to Rust for common structures like doubly linked lists and for unsafe components.
Some safety: some safety features such as bounds-checked slices and stronger type safety compared to C.
Mature: multiple compiler implementations and a stable core set of features.

Easy to get started with, and fun to use!

Tutorial: writing a bare-metal Raspberry Pi program in D

Software: QEMU, and a bare-metal toolchain with LDC or GDC.



Hardware: all you need is a Raspberry Pi, micro SD card, and a serial connector.

See https://zyedidia.github.io/blog/posts/1-d-baremetal/ for a writeup (targeting QEMU RISC-V).

I have pre-built toolchains available at https://github.com/zyedidia/build-gdc

What is bare-metal programming?

Normally your program runs in a “hosted” environment. OS provides:

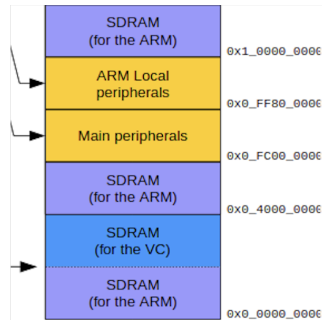
- Display capabilities.
- Ability to send and receive data over ports.
- A file system.
- Multiple processes.

When running bare-metal, hardware is the only environment:

- The hardware will begin executing your program at a pre-defined address.
- Hardware devices (display, output ports, disk) can be controlled by reading/writing a special region of memory.

Raspberry Pi setup

1. GPU loads the firmware binary from the SD card and places it at 0x0.
2. GPU loads the kernel binary from the SD card and places it at 0x80000.
3. CPU begins execution at 0x0.
4. Firmware sets up various control registers, and then jumps to 0x80000.
5. Our code begins running!

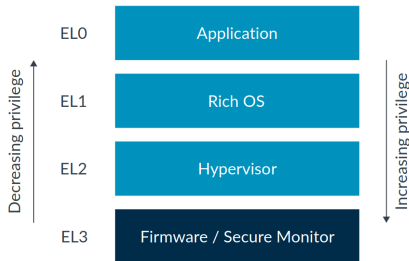


ARM view of the Address Map
in “Low Peripheral” mode

Quick overview of ARMv8

The CPU ultimately runs machine code (e.g., ARM):

- 31 general-purpose registers (`x0`-`x30`).
- Stack pointer (`sp`) and zero register (`xzr`).
- Load-store architecture: `ldr xN, [xM]`, and `str xN, [xM]`.
- Four exception levels: EL0 (user), EL1 (kernel), EL2 (hypervisor), EL3 (firmware).



Setting up a minimal bare-metal environment

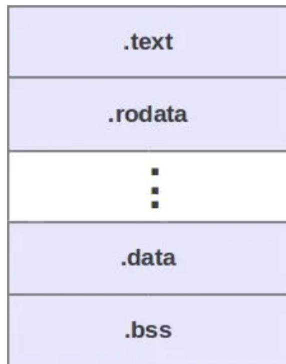
What does D code need?

1. A valid stack pointer.
2. An initialized Block Start Segment (BSS).

Link `_start` at 0x80000.

`start.s`:

```
.section ".text.boot"
.globl _start
_start:
      ldr    x1, =_start // stack below _start
      mov    sp, x1
      ldr    x0, =_bss_start // zero the BSS
      ldr    x1, =_bss_end
cmpr: cmp    x0, x1
      bcc    loop
      bl     kmain
halt: b      halt
loop: str    wzr, [x0], 4
      b      cmpr
```



`main.d`:

```
extern (C) void kmain() {}
```

Using D with a custom runtime

D compilers require the presence of `object.d`, which contains definitions that are always available. Usually it contains the core definitions for the D runtime.

Make your own empty `object.d`:

```
module object;
```

As the project expands, you can add more features:

- Basic types (`string`, `size_t`, ...).
- Support for assertions (`_assert()`).
- Support for builtins (atomics, volatile load/store, ...)
- Support for array operations (`_d_array_slice_copy(...)`, ...)
- Whatever features of the D runtime you want to include.

Generating output

We can transmit bytes over the serial connector by using the Pi's UART device.

The datasheet tells us how to configure the UART and how to transmit/receive data.

`0x7e215000`.

Offset	Name	Description
0x00	AUX_IRQ	Auxiliary Interrupt status
0x04	AUX_ENABLES	Auxiliary enables
0x40	AUX_MU_IO_REG	Mini UART I/O Data

Datasheet: https://www.scs.stanford.edu/~zyedidia/docs/rpi/bcm2711.pdf

Generating output

We can transmit bytes over the serial connector by using the Pi's UART device.

The datasheet tells us how to configure the UART and how to transmit/receive data.

Bits	Name	Description	Type	Reset
31:8	Reserved.	-	-	-
7:0	LS 8 bits Baudrate read/write, DLAB=1	Access to the LS 8 bits of the 16-bit baudrate register. (Only if bit 7 of the line control register (DLAB bit) is set)	RW	0x00
7:0	Transmit data write, DLAB=0	Data written is put in the transmit FIFO (Provided it is not full) (Only if bit 7 of the line control register (DLAB bit) is clear)	WO	0x00
7:0	Receive data read, DLAB=0	Data read is taken from the receive FIFO (Provided it is not empty) (Only if bit 7 of the line control register (DLAB bit) is clear)	RO	0x00

Generating output

We can transmit bytes over the serial connector by using the Pi's UART device.

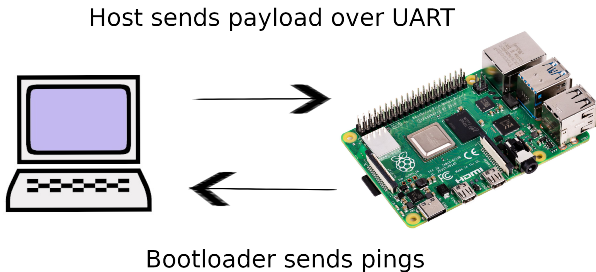
The datasheet tells us how to configure the UART and how to transmit/receive data.

```
extern (C) void kmain() {
    foreach (b; "hello!") {
        volatileStore(cast(ubyte*) 0x7e215040, b);
    }
}
```

Next step: wrap this into a device driver with a better interface.

The bare-metal development experience

1. Use QEMU for prototyping and quick testing.
2. Use a UART bootloader to quickly send new programs to the Pi.



What is Multiplix?

A simple Unix-like kernel I have been developing for several months.

- Supports ARMv8 and RISC-V 64, and runs on real hardware: Raspberry Pi 3/4, VisionFive 1/2.
- Preemptive processes.
- Simple set of system calls for user processes.
- Unix v6 file system.
- Partial multicore support (in-progress).
- Written in D! Can be built with LDC or GDC.

Next: how I use D to develop Multiplix.

The style of D that I use

A “kernel” style of D: **keep it simple**.

Most important features (compared to C):

- Modules.
- Built-in slices with bounds-checking.
- RAII or `scope(exit)` for resource management.
- No preprocessor, contracts, more sane than C, ...

Library code additionally makes use of:

- Templates (for the allocator, data structures, generic bitwise operations).
- Destructors (for resource management and concurrency primitives).
- Iterators (for pagetables and other data structures).

Sticking to BetterC by choice (no classes, module constructors, exceptions, etc...).

The style of D that I use

A "kernel" style of D: **keep it simple**. But there are some exceptions.

Most important features (compared to C):

- Modules.
- Built-in slices with bounds-checking.
- RAII or `scope(exit)` for resource management.
- No preprocessor, contracts, more sane than C, ...

Library code additionally makes use of:

- Templates (for the allocator, data structures, generic bitwise operations).
- Destructors (for resource management and concurrency primitives).
- Iterators (for pagetables and other data structures).

Sticking to BetterC by choice (no classes, module constructors, exceptions, etc...).

Exception: mixins for system registers

System registers control machine configuration.

They are read/written with a special MRS/MSR instruction.

Exception: mixins for system registers

```
const char[] GenSysRegWrOnly(string name) =
`pragma(inline, true) ` ~
`static void ` ~ name ~ `(uintptr v) {
    asm {
        "msr ` ~ name ~ `, %0" : : "r"(v);
    }
}`;


struct SysReg {
    mixin(GenSysReg!("spsr_el2"));
    mixin(GenSysReg!("hcr_el2"));
    // ...
}
```

Exception: mixins for system registers

```
// Set spsr to return to EL1h.
SysReg.spsr_el2 = Spsr.a | Spsr.i | Spsr.f | Spsr.el1h;
// Configure EL1 to run in aarch64 mode.
SysReg.hcr_el2 = Hcr.rw_aarch64;
// Enable all debug exceptions in kernel mode.
SysReg.mdscr_el1 = SysReg.mdscr_el1 | Mdscr.mde;
// Route debug exceptions to EL2.
SysReg.mdcr_el2 = SysReg.mdcr_el2 | Mdcr.tde;
```

Very convenient!

C and Rust would use macros for this and still end up with a worse interface.

How to handle shared memory

Shared memory is a source of bugs, and D can help us to avoid some problems.

Approaches for handling managing global data:

- Make the data CPU-local (don't share it!).
- Force data access to go through a lock (Rust-style).
- Use manual locks with RAII or `scope(exit)`.
- Only use the data in a single-threaded context.

Per-CPU data

Per-CPU data is not the same as thread-local data: it is not preserved across a thread migration.

This data changes when a thread is migrated, so we cannot make it thread-local.

```
struct PerCpu(T) {
    T[Machine.ncores] vals;
    // note: interrupts must be disabled
    ref T val() shared {
        return *cast(T*) &this.vals[rdcpu()];
    }
    alias val this;
}
```

```
shared PerCpu!(int) x;
void foo() {
    x = bar();
}
```

Rust/Fearless-style locks

```
shared Mutex!(int) foo = Mutex!(int)(10);

void bar() {
    auto foo = typeof(foo).Guard(&foo);
    printf("%d\n", foo.val);
}
```

Rust/Fearless-style locks

```
shared Mutex!(int) foo = Mutex!(int)(10);

void bar() {
    auto foo = foo.lock();
    printf("%d\n", foo.val);
}
```

Note: this version relies on Named Return Value Optimization.

Rust/Fearless-style locks

```
shared Mutex!(int) foo = Mutex!(int)(10);

void bar() {
    auto foo = foo.lock();
    printf("%d\n", foo.val);
}
```

Note: this version relies on Named Return Value Optimization.

Built on top of a spinlock primitive implemented with atomics.

Rust/Fearless-style locks

```
struct Mutex(T) {
    private T val;
    private Spinlock lock;

    this(T v) {
        val = v;
    }

    static struct Guard {
        private shared Mutex!(T)* mutex;
        this(shared Mutex!(T)* m) {
            mutex = m;
            mutex.lock.lock();
        }
        ref T val() {
            // cast away the 'shared'
            return *cast(T*) &mutex.val;
        }
        alias val this;
        ~this() {
            mutex.lock.unlock();
        }
        @disable this();
        @disable this(this);
        @disable void opAssign(Guard);
    }
}
```

Rust/Fearless-style guards

```
static struct Guard {
    private shared Mutex!(T)* mutex;
    this(shared Mutex!(T)* m) {
        mutex = m;
        mutex.lock.lock();
    }
    ref T val() {
        // cast away the 'shared'
        return *cast(T*) &mutex.val;
    }
    alias val this;
    ~this() { mutex.lock.unlock(); }
    @disable this();
    @disable this(this);
    @disable void opAssign(Guard);
}
```

Rust/Fearless-style guards

```
static struct Guard {
    private shared Mutex!(T)* mutex;
    this(shared Mutex!(T)* m) {
        mutex = m;
        mutex.lock.lock();
    }
    ref T val() {
        // cast away the 'shared'
        return *cast(T*) &mutex.val;
    }
    alias val this;
    ~this() { mutex.lock.unlock(); }
    @disable this();
    @disable this(this);
    @disable void opAssign(Guard);
}
```

Rust/Fearless-style guards

```
static struct Guard {
    private shared Mutex!(T)* mutex;
    this(shared Mutex!(T)* m) {
        mutex = m;
        mutex.lock.lock();
    }
    ref T val() {
        // cast away the 'shared'
        return *cast(T*) &mutex.val;
    }
    alias val this;
    ~this() { mutex.lock.unlock(); }
    @disable this();
    @disable this(this);
    @disable void opAssign(Guard);
}
```

Rust/Fearless-style guards

```
static struct Guard {
    private shared Mutex!(T)* mutex;
    this(shared Mutex!(T)* m) {
        mutex = m;
        mutex.lock.lock();
    }
    ref T val() {
        // cast away the 'shared'
        return *cast(T*) &mutex.val;
    }
    alias val this;
    ~this() { mutex.lock.unlock(); }
    @disable this();
    @disable this(this);
    @disable void opAssign(Guard);
}
```

The infamous __gshared

Sometimes `__gshared` is useful, but the compiler cannot enforce safety.

- The data is manually protected by a lock.
- The data is only accessed by a single core (e.g., during boot when only one core is running).

Annoyance: the gshared double-underscore.

Allocator API

```
ubyte[] kalloc(usize sz);              // alloc 'sz' bytes
ubyte[] kzalloc(usize sz);             // alloc 'sz' bytes zeroed
T* knew(T, Args...)(Args args);        // alloc and construct a 'T'
T[] kallocarray(T)(usize nelem);       // alloc slice of 'T's
void kfree(T)(T* ptr);                 // free pointer to 'T'
void kfree(T)(T[] arr);                // free a slice of 'T's
```

Simple, typed, and tracks size information automatically.

The allocator does not need to store per-object allocation sizes.

Allocator API

```
ubyte[] kalloc(usize sz);             // alloc 'sz' bytes
ubyte[] kzalloc(usize sz);            // alloc 'sz' bytes zeroed
T* knew(T, Args...)(Args args);       // alloc and construct a 'T'
T[] kallocarray(T)(usize nelem);      // alloc slice of 'T's
void kfree(T)(T* ptr);                // free pointer to 'T'
void kfree(T)(T[] arr);               // free a slice of 'T's
```

Simple, typed, and tracks size information automatically.

The allocator does not need to store per-object allocation sizes.

Future: possible to enable garbage collection for kernel?

Using both LDC and GDC

D has two compilers that can target aarch64/riscv64, so we want to use both of them!

Mostly compatible, some differences:

- Builtins for atomics.
- Inline assembly for passing arguments to system/firmware calls.
- Sanitizers.

LDC:

```
import ldc.llvmasm;
return __asm!(usize)(
    "hvc 0",
    "={x0},{x7},{x0},{x1},{x2},~{memory}",
    fn, arg0, arg1, arg2
);
```

GDC (requires version 13):

```
import gcc.attributes;
@register("x7") usize x7 = fn;
@register("x0") usize x0 = arg0;
@register("x1") usize x1 = arg1;
@register("x2") usize x2 = arg2;
asm {
    "hvc 0" : "+r"(x0) : "r"(x7), "r"(x1), "r"(x2) :
      "memory";
}
return x0;
```

Modules and conditional compilation

Architecture-specific code is placed in the `arch/riscv64` or `arch/aarch64` directory. Modules within `arch` import the appropriate architecture-specific package depending on version tags.

```
module plix.arch.vm;

version (RISCV64) {
    public import plix.arch.riscv64.vm;
} else version (AArch64) {
    public import plix.arch.aarch64.vm;
}
```

Focus: increasing correctness with automated checking

I have a research interest in tools for automated bug-finding.

What kind of analysis tools are can be used with Multiplix?

- Static analysis: find bugs by examining the code before running it.
- Dynamic analysis: find bugs while the code is running (usually only enabled for test runs).

Static and dynamic analysis

Static analysis:

- D-Scanner.
- GCC analyzer.

Usage:

- Unused variables, return values, etc...
- Unnecessary qualifiers.
- Unsafe pointer accesses.

Negative: limited information; false positives.

Dynamic analysis:

- GCC sanitizer (undefined, address).
- Custom tools.

Usage:

- Memory management bugs.
- Undefined behavior.
- Race conditions.
- TLB/I-cache mismanagement[a].
- Bugs in device code.

Negative: bug must run to be caught.

[a] Even found a minor bug in Linux!

Dynamic analysis: GCC sanitizer

GDC has `-fsanitize=undefined` and `-fsanitize=address` for finding undefined behavior and memory errors.

We can use `-fsanitize=kernel-address` instead, and define our own callbacks.

```
void __asan_load1_noabort(uintptr addr);
void __asan_load2_noabort(uintptr addr);
void __asan_store1_noabort(uintptr addr);
void __asan_store2_noabort(uintptr addr);
...
```

Basic custom address sanitizer:

- On load/store: check `addr` against shadow memory.
- Update shadow memory in `kalloc`/`kfree`.

memory	shadow
free	0
free	0
allocated	1
allocated	1
allocated	1
allocated	1
free	0
free	0

Custom dynamic analysis with debug hardware

Dynamic analysis often requires complicated tools:

- Compiler instrumentation (GCC sanitizer).
- Dynamic re-compilation (Valgrind).

Problem: these tools are often too large and complex to make custom checkers for kernel code.

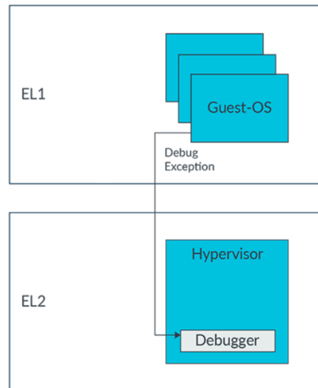
Custom dynamic analysis with debug hardware

Another approach: use hardware breakpoints and watchpoints to do dynamic analysis.

Automated checkers run in the monitor (EL2) and catch events with traps via breakpoints, watchpoints, and single-stepping.

Orders of magnitude less code!

Slow but simple (enabled only for debugging runs).

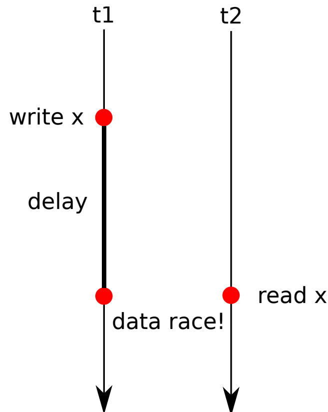


Dynamic analysis with debug hardware: Data Collider

Finding bugs in multi-threaded code (Data Collider[1] technique).

Approach: induce data races to happen.

1. Trap loads/stores with a range watchpoint.
2. When a load/store is executed mark the address in a table.
3. Delay for a small amount of time.
4. If a separate core traps a load/store with an address in the table: data race!



[1] Erickson et al. *Effective Data-Race Detection for the Kernel*, OSDI 2010.

Dynamic analysis with debug hardware: I-cache checker

How does an OS load a process? (at a high level)

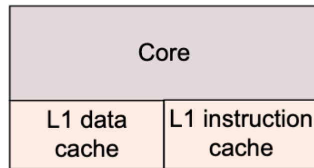
- Write the binary into memory.
- Jump to its entrypoint (somewhere in that memory).

Dynamic analysis with debug hardware: I-cache checker

How does an OS load a process? (at a high level)

- Write the binary into memory.
- Jump to its entrypoint (somewhere in that memory).

RISC-V specification: *RISC-V does not guarantee that stores to instruction memory will be made visible to instruction fetches on a RISC-V hart until that hart executes a FENCE.I instruction*

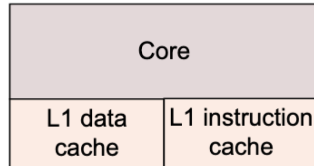


Dynamic analysis with debug hardware: I-cache checker

How does an OS load a process? (at a high level)

- Write the binary into memory.
- Execute FENCE.I
- Jump to its entrypoint (somewhere in that memory).

RISC-V specification: *RISC-V does not guarantee that stores to instruction memory will be made visible to instruction fetches on a RISC-V hart until that hart executes a FENCE.I instruction*

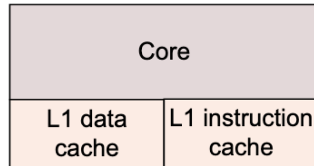


Dynamic analysis with debug hardware: I-cache checker

Automated checker:

1. Record writes to memory into a table.
2. Clear the table when a FENCE.I executes.
3. If the program counter is ever in the table: ERROR!

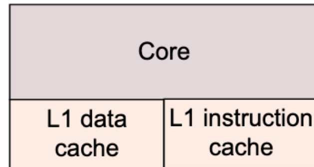
We can use single-stepping to hook these events.



Dynamic analysis with debug hardware: I-cache checker

Automated checker:

1. Record writes to memory into a table.
2. Clear the table when a FENCE.I executes.
3. If the program counter is ever in the table: ERROR!



We can use single-stepping to hook these events.

Similar checker for Translation Lookaside Buffer (TLB) consistency.

Build system: Knit

I have been working on a build system called **Knit**, used for Multiplix.

Knit is not specific to D, but I have an optimization that is especially helpful for D.

The Knit-based build system for Multiplix supports:

- Multiple architectures (RISC-V, AArch64).
- Multiple boards (QEMU, Raspberry Pi, VisionFive).
- Multiple options (optimization, LTO, unified build, sanitizers, D compiler).
- **Parallel and incremental builds.**

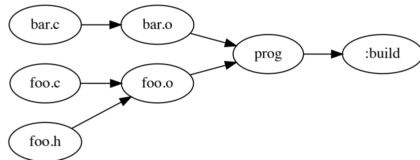
A brief introduction to Knit

Knit is a general build tool similar to Make for expressing build graphs.

Rules are written in Make-style syntax as an embedded DSL within Lua.

Example: an incremental build for a C project.

```
local knit = require("knit")
local src = knit.glob("*.c")
local obj = knit.extrepl(src, ".c", ".o")
local prog = "prog"
return b{
    $ $prog: $obj
        cc $input -o $output
    $ %.o:D[%.d]: %.c
        cc -O2 -MMD -c $input -o $output
}
```



Over-engineering the build system: parallel and incremental builds for D

For each D source module `foo.d` that changes:

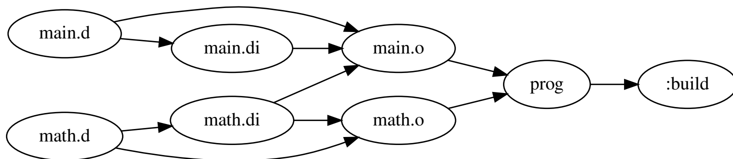
Rebuild all modules that import `foo.d` — *only if the API changes*.

The API is expressed by the `foo.di` file, which can be auto-generated.

Over-engineering the build system: parallel and incremental builds for D

What we want:

1. A D source file `math.d` changes.
2. Knit first rebuilds the `math.di` "header" file for that single source file (using `-H`).
3. **Key part**: If the header is unchanged from a prior build, rebuild only `math.o`.
4. Otherwise rebuild `math.o` and all source files that import `math` (using info from `--makedeps`/`-MD`).



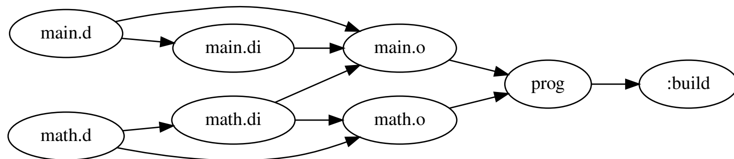
Over-engineering the build system: parallel and incremental builds for D

Possible using Knit's *dynamic task elision* feature (with D interface files):

If Knit dynamically detects that a rebuilt file is unchanged, it will skip the build steps in that subgraph.

Get the incrementality benefit of C header files without having to write them.

Note on simplicity: reducing template usage improves incremental compilation.



More info here: https://zyedidia.github.io/blog/posts/4-incremental-d-knit/

Future directions for Multiplix: process isolation

Techniques for isolating processes with software:

- Languaged-based isolation — WebAssembly, Singularity.
- Software fault isolation[2] (SFI) — Google Native Client (NaCl).

Lightweight Fault Isolation (LFI): a new SFI system I have been working on with very low overhead (reduced by 2-16x compared to WebAssembly).

LFI for process isolation: all processes run in kernel mode in a single address space (what could go wrong?)

Benefit: context switch/system call overhead reduced by 10-100x. Feasible? We'll see!

---

[2] Wahbe et al. *Efficient Software-based Fault Isolation*, SOSP 1993.

Thank you!

Check it out at https://github.com/zyedidia/multiplix.

Knit: https://github.com/zyedidia/knit.

Slides at https://www.scs.stanford.edu/∼zyedidia/docs/slides/dconf23.pdf.

Any questions?