Bluejay

swissChili

Jan 01, 2023

CONTENTS:

1 Building Bluejay **3**
- 1.1 Common Issues . . . 3

2 JMK 2 Build System **5**

3 Architecture **7**
- 3.1 Multi tasking . . . 7
- 3.2 Drivers . . . 7
  - 3.2.1 PCI Device Drivers . . . 7

4 Bluejay Filesystem **9**
- 4.1 Virtual Filesystem . . . 9
- 4.2 Filesystem Drivers . . . 9
- 4.3 Creating a Virtual Drive in QEMU . . . 9

5 Lisp Standard Library **11**
- 5.1 Top-level primitives . . . 11
- 5.2 Functions . . . 12

6 Kernel Logging **15**

CHAPTER

ONE

BUILDING BLUEJAY

Bluejay uses the home-grown `Jmk` build system, which is basically just a GNU m4 script that generates makefiles from `Jmk` files — makefiles with some custom macros.

To build a fresh clone of Bluejay the first thing you will need to do is run `bin/jmk` to generate your makefiles for you. You should get some output like this:

```
Processing ./boot/initrd/Jmk
Processing ./src/kernel/dri/ata_pio/Jmk
Processing ./src/kernel/dri/ahci/Jmk
Processing ./src/kernel/dri/pci/Jmk
Processing ./src/kernel/Jmk
Processing ./src/mkinitrd/Jmk
Processing ./src/lisp/Jmk
```

Then just build using `src/kernel/Makefile`. There are a few additional targets for your convenience:

- `qemu` builds and launches the kernel using QEMU's SeaBIOS
- `qemu-iso` builds a GRUB ISO and launches using QEMU
- `install` builds a GRUB ISO and installs it to `boot/bluejay.iso`
- `debug` launches kernel in QEMU and launches GDB in the terminal.
- `debug-wait` launches kernel in QEMU and starts a GDB server on `localhost:1234`. This is recommended if you want to debug since you can connect to it from vscode or any other IDE. `.vscode/launch.json` is set up to work with this so you can debug the kernel very easily.

In order to build Bluejay you will need the following dependencies:

```
gcc gcc-multilib nasm qemu-system-i386 make m4 python3 awk
```

There are some additional dependencies for building a GRUB ISO but I don't remember them at the time of writing.

1.1 Common Issues

If when launching Bluejay in QEMU with `make qemu` you see the message

```
[ DEBUG ] mb.mods_addr = <whatever>, 0x<whatever>
```

Followed by a page fault (`#PF`), your QEMU is out of date. You can either run the GRUB ISO (with `make qemu-iso`), which is slower, or upgrade your QEMU to at least version 6.0.

This is because prior to 6.0 QEMU's integrated bootloader did not support multiboot modules. This message is caused by the kernel attempting to read a module that was loaded incorrectly.

CHAPTER

TWO

JMK 2 BUILD SYSTEM

JMK2 is a rewrite of the JMK build system. I am slowly porting Bluejay to JMK2 instead of the legacy M4-based JMK build system.

JMK2 is used to generate makefiles for each project in Bluejay. A project is a directory with a `Jmk2` file (case sensitive). Each project produces a single output based on some sources.

The script `bin/jmk2` looks in the source tree for `Jmk2` files, and process each one into the corresponding Makefile. It accepts option definitions with the `-D` flag, eg `./bin/jmk2 -DSOME_OPTION=123`. You can also specify the C compiler, assembler, and linker to use with the `-c`, `-a`, and `-l` flags, respectively.

Here is an example Jmk2 file:

```
init hello # hello is the name of the project
srcs hello.c world.c # the source files this project uses
type executable # the preset type of project this is
```

Each line consists of a command (`init`, `srcs`, `type`) and its arguments. The commands are documented here:

`init name [target]`

Initializes the project with a given name. The `name` is currently unused, but should be set to a descriptive identifier.

`target` is the name of the target that the project generates. By default it is the same as `name`. For an executable, this could be `hello` or `hello.exe`. For a shared library, `libhello.so`, etc.

`preset preset_name`

Applies the preset `preset_name`. A preset is a function defined in the `::presets` namespace which makes some changes to the project state.

These are the default presets:

- `freestanding` Changes the `cflags` to build a freestanding binary (without linking the standard library).
- `optimize` Changes the `cflags` and `asmflags` to enable compile-time optimizations.
- `32` Tells the compilers to produce a 32 bit build.
- `debug` Tells the compilers to enable debug information in the resulting builds (enables DWARF symbols).
- `warn` Enables useful warnings and `-Werror`.
- `nasm` Sets `nasm` as the default assembler.

`presets preset_a [preset_b]...`

Applies all the given presets in order. Identical to calling `preset` once for each argument.

`cflag string`

Adds `string` to the `::cflags` variable, which will be passed to the C compiler.

`cflags string_a [string_b]...`

Adds multiple strings to the `::cflags` variable, the same as calling `cflags` repeatedly.

`asmflag, asmflags`

Same as `cflag`, `cflags` but for the `::asmflags` variable.

`option name default_value`

If the option `name` has not been specified when invoking `bin/jmk2`, sets the value of the option to `default_value`. Options can be read with `::options(option_name)`.

TODO: finish!

CHAPTER

THREE

ARCHITECTURE

This document seeks to provide a brief overview of Bluejay architecture. This should be a good starting point for understanding the code.

Bluejay is exclusively a multiboot kernel, it neither provides nor supports alternative bootloaders.

The bootloader (probably GRUB) will initially run the code in `boot.s`. This is where it all begins. This code sets up segmentation and paging and maps the higher-half of virtual memory (everything above `0xC0000000`) to the kernel. At first it only maps 8 megabytes, more memory can be mapped on request.

After moving to high memory the kernel jumps to C code and enters `kmain` in `main.c`. This is the highest level procedure in the kernel, which sets up kernel services and drivers one at a time.

This includes VGA, keyboard, and PCI drivers, as well as paging and preemptive multi tasking.

3.1 Multi tasking

Multi tasking is handled by code in `task.c`. It is first initialized in `init_tasks`, which sets up the initial task. Once this is called kernel threads can be spawned at will.

Every clock tick an interrupt is triggered (see `clock.c` for timing) which causes a task switch to occur. Bluejay uses a simple round-robin scheduler, and there is no way for tasks to voluntarily give up their processing time (even in the case of blocking IO operations). `task.c` contains the implementation of the scheduler.

3.2 Drivers

So far drivers must be written either using plain `in` and `out` instructions or on top of the existing PCI driver.

3.2.1 PCI Device Drivers

PCI device drivers must register a `struct pci_device_driver` in order to interface with a certain device (or class of devices). See `include/kernel/dri/pci/pci.h` for details.

A PCI device driver must pass an instance of this structure to `pci_register_device_driver` (in `include/kernel/dri/pci/pci.h`. If `supports` returns true, (for example, if the class and subclass of the `struct pci_device` are supported by teh driver) `init` will be called. At this point the driver may do whatever it wishes with the PCI device, although all blocking operations should be done in another thread (using `spawn_thread` in `include/kernel/task.h` for example).

CHAPTER

FOUR

BLUEJAY FILESYSTEM

Filesystem drivers are still a work in progress. To test a file system you will want to create and mount a virtual block device. The makefile in `src/kernel` will generate an `hd0_ext2.img` EXT2 disk image for you automatically. The default size is 32 megabytes, but you can create your own of any size if you want. Once the image has been created it will be loaded by QEMU automatically.

In order to write to the virtual hard disk from your host operating system you should mount it. The `make mount` command in `src/kernel` mount the image to `$(BLUEJAY_ROOT)/mnt`. If you are using an EXT2 filesystem you should probably change the owner of that directory once it is mounted so that you can write to it.

4.1 Virtual Filesystem

The Bluejay VFS is heavily inspired by UNIX. It relies on inodes and a tree of file nodes. The source can be found in `src/kernel/vfs.c`. This also exports a very low-level API for dealing with files – including the usual read(), write(), readdir(), etc – but this should not be used for much longer. A high level API utilizing file descriptors will be implemented to make this simpler.

4.2 Filesystem Drivers

The current filesystem driver(s) available in Bluejay are:

- **`ext2`**
  - Read-only support, write support is in progress

4.3 Creating a Virtual Drive in QEMU

By default `make qemu` will load `hd0_$(FS).img` as the virtual hard drive for Bluejay. `FS` defaults to `ext2` but can be set in your `Jmk.options` to any value. If this file does not exist it will be created using `mkfs.$(FS)`, ie `mkfs.ext2` by default. The default size of the file system is 35 megabytes, although you can create one of any size manually if you want. 35 megabytes is plenty for testing though.

The `make mount` command will mount the current virtual hard drive in `$(ROOT)/mnt` (where `$(ROOT)` is the root directory of the Bluejay sources, not `/`). This command requires superuser privileges. If you want to give your (host) user account write permissions use `chown -R user:group /path/to/mnt` where `user` and `group` are the user and group you want to own the files.

Currently Bluejay ignores file permissions so it doesn't matter who you set the owner to.

CHAPTER

FIVE

LISP STANDARD LIBRARY

This provides documentation for every built-in function in the Lisp standard library. It is not auto-generated, please update this documentation if you change the API in any way.

In general every user-facing API in the standard library should be documented here.

- `(x ...)` represents a list `x`.
- `& body` means that the rest of the list is represented by `body`.
- `[something]` means that `something` is optional.

5.1 Top-level primitives

These are "functions" that can only appear at the top-level of the program. This means they can't be nested in any other expressions.

`(defun function-name (args ...) & body)`

Defines a function `function-name` that takes `args` and evaluates `body`. `function-name` is quoted, not evaluated.

```
(defun say-hi (name)
  (print "Hi, ")
  (print name))

(say-hi "Joe")
; "Hi,"
; "Joe"
```

`(defmacro macro-name (args ...) & body)`

`defmacro` is to macros as `defun` is to functions. When `macro-name` is called, whatever it evaluates to will be compiled.

Note that internally this compiles a function the same way all other functions are compiled, meaning you can call **any** lisp function from a macro definition and it will work as expected.

```
(defun double (n)
  (+ n n))

(defmacro call-with-4 (whatever)
  (print "this was run at **compile time**")
  (print whatever)
```

(continues on next page)

(continued from previous page)

```
  ;; ``whatever`` expands to the form passed to this macro, in this case
  ;; ``double``.
  (list whatever 4))

(print (call-with-4 double))
; "this was run at **compile time**"
; 'double
; 8
```

5.2 Functions

(if condition true-condition [false-condition])

Evaluates `condition`, if it is truthy (non-`nil`) `true-condition` is evaluated. Otherwise `false-condition` is evaluated. If `false-condition` is not provided and `condition` is `nil`, `if` will evaluate to `nil`.

```
(print (if (= 2 3)
           "2 = 3"
           "2 /= 3"))
; 2 /= 3
```

(let1 (variable binding) & body)

Evaluates `binding` and binds it to `variable`, then evaluates `body`. After `body` is evaluated `variable` is unbound.

```
(let1 (greeting (greet "John"))
  (do-something greeting)
  (print greeting))
; greeting is no longer bound
```

(gc)

Force the garbage collector (GC) to run.

(car pair)

Return the first item in `pair`.

```
(car (cons 'a 'b)) ;=> 'a
```

(cdr pair)

Return the second (last) item in `pair`.

```
(cdr (cons 'a 'b)) ;=> 'b
```

(cons a b)

Return a cons-pair containing `a` and `b`.

(print val)

Print out `val` to standard output. This will not be formatted as an s-expression, but in a manner more similar to the internal representation.

(list & items)

Returns a cons-list of items.

```
(list 1 2 3)
; is the same as
(cons 1 (cons 2 (cons 3 nil)))
```

(quote form)

Returns form without evaluating it.

```
'(cons a b)
; or
(quote cons a b)
; is the same as
(list 'cons 'a 'b)
```

(lambda (args ...) & body)

Creates an anonymous function (closure). This function uses **lexical scope** meaning that any free variables (variables bound outside this lambda definition) are "captured" by the closure. You can call this function with `funcall` (to be implemented) or `apply`.

```
(let1 (number 3)
  (let1 (adds-number-to (lambda (n)
                          (+ n number)))
    (print (apply adds-number-to '(5)))))
; 8
```

(apply function (args ...))

Call `function` with `args` and return the result. Note that since this is a Lisp-2 (i.e. functions and variables do not share the same namespace) you need to pass a **function object** (i.e. a lambda or quoted function).

CHAPTER

SIX

KERNEL LOGGING

Drivers and other kernel components may write log messages to the default output (currently only VGA since it is the only display target implemented) using `kprintf` in `include/kernel/log.h`. Additional defines in the same file may help differentiate different types of log messages (i.e. errors, debug information, etc).

```
kprintf(OKAY "Something succeeded\n");
kprintf(ERROR "Something failed :(\n");
// etc, see log.h for details
```

Bluejay is an operating system inspired by UNIX and early Lisp machines. Currently it only targets x86. There are no plans to port to other platforms.

This documentation should provide an introduction to compiling, developing, and using Bluejay.