Porting OpenBSD to RISC-V ISA

A Project Report

Presented to

The Faculty of the Department of Computer Engineering

San José State University

In Partial Fulfillment

of the Requirements for the Degree

Master of Science in Computer Engineering

Master of Science in Software Engineering

by

Brian Bamsch, Wenyan He, Mengshi Li, Shivam Waghela

May 2020

APPROVED

M. Larkin

Michael Larkin, Project Advisor

David Bruck, Director, MS Computer Engineering

Dan Harkey, Director, MS Software Engineering

Xiao Su, Department Chair

ABSTRACT

PORTING OPENBSD TO RISC-V ISA

by Brian Bamsch, Wenyan He, Mengshi Li, Shivam Waghela

The RISC-V ISA shows significant promise as an upcoming, flexible ISA for general purpose computing. However, adoption of RISC-V is largely hindered by lack of software support, particularly among popular operating systems. Only a few operating systems, including Linux and FreeBSD, have been updated to support RISC-V. Many other operating systems, including OpenBSD, do not yet run on RISC-V hardware. This project makes the first step towards a port of OpenBSD by introducing support for the RISC-V ISA within the OpenBSD kernel. As part of this effort, a deep-dive is performed into the machine-dependent aspects of bootstrap, memory, trap, process, and device subsystems of the OpenBSD kernel. The ported OpenBSD kernel has implemented early bootstrap logic in assembly, adapted PMAP to RISC-V MMU in *Sv39*-mode, developed trap handlers for interrupts and exceptions, implemented routines for process fork and context switch, and developed drivers to probe and attach machine-dependent devices. With these efforts, the OpenBSD kernel can boot on a QEMU-emulated RISC-V hardware target up through entry into the `init` process.

ACKNOWLEDGMENTS

We would like to thank our advisor, Michael Larkin. His expertise with OpenBSD has been essential to the team's success in both the implementation and bring-up of the OpenBSD operating system on RISC-V.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

1 INTRODUCTION

1.1 RISC-V

RISC-V is an Instruction Set Architecture (ISA) whose development was started by UC Berkeley in 2010 [1]. The RISC-V ISA is open-source and royalty-free and does not carry the same restrictive licensing burden as other ISAs, including x86 and ARM. Developer manuals and documentation for the RISC-V ISA are made available for free from the RISC-V Foundation which owns, maintains, and publishes documents for all intellectual property pertaining to the RISC-V ISA.

The RISC-V ISA follows a modular design philosophy which aims to be both flexible and versatile for a broad range of different applications. The RISC-V ISA defines two frozen base instruction sets[1], `RV32I` and `RV64I`, which provide only a limited set of integer instructions. All other ISA features come as extensions [2] to the base instruction set. These extensions provide general purpose features such as multiplication and division (`M`), atomic instructions (`A`), single-precision floating-point (`F`), and double-precision floating-point (`D`). The set of features supported by a particular implementation of the RISC-V ISA are identified by its implementation identifier.

ISA-level modularity gives control back to vendors to choose an implementation that best fits their specific application. Embedded platforms might implement `RV32EC` — the 32-bit base instruction set optimized for embedded systems with support for compressed format instructions — to optimize for a smaller silicon footprint and lower power consumption. More powerful machines intended to run a full-fledged operating system might implement `RV64GC` which provides a useful set of features for general purpose computing.

1. The RISC-V ISA also defines two additional base instruction sets: `RV32E` and `RV128I`. The former reduces the number of registers from the `RV32I` base instruction set for embedded platforms and the latter includes 128-bit support. Unlike `RV32I` and `RV64I`, these two base instruction sets are not yet ratified.

The RISC-V ISA has an advantage over older ISAs due to the benefit of hindsight. RISC-V incorporates ideas learned from other popular instruction sets that have had multiple decades to mature in the marketplace [3]. Unlike more mature instruction sets, the RISC-V ISA is not bogged down by decades-worth of backward compatibility over multiple generations of hardware.

The RISC-V ISA also benefits from its openness and transparency. Unlike other ISAs whose implementation details are obscured behind non-disclosure agreements, the RISC-V ISA adopts a model which encourages collaboration. A number of open-source implementations of the RISC-V ISA are already available for experimentation [4]. Open-source hardware implementations benefit from the diverse perspectives of their individual contributors which help harden the hardware against security threats over time. Combining a hardened open-source hardware platform with a hardened operating system kernel (e.g., OpenBSD) will lead to the development of noticeably more secure systems.

The RISC-V ISA is not without its weaknesses, of which its newness is a major one. Unlike well-established ISAs, the RISC-V ISA is just starting to gain traction in commercial applications and has not yet seen widespread adoption. The lack of adoption can be attributed to both the newness of the architecture and lack of software support for RISC-V. Only a handful of operating systems, including FreeBSD and Linux, have been updated to support the RISC-V ISA. The majority of operating systems do not support RISC-V yet. OpenBSD is one of them.

1.2 OpenBSD

OpenBSD is a Unix-like operating system which descends from the Berkeley Software Distribution (BSD) series of operating systems. OpenBSD was forked in 1995 from NetBSD by Theo de Raadt. Since then, OpenBSD has become renowned for its strong security principles and contributions to open-source software, for example, OpenSSH.

OpenBSD puts decades worth of security research and development into practice and prides itself on its strong security record. OpenBSD touts itself as one of the most secure operating system kernels available on the market today. Its strong security record is maintained through the practice and application of a number of methodologies, including:

- Comprehensive file-by-file audits of all critical software components.
- Proactive endeavors to identify and address bugs even if not proven exploitable.
- Applying strong security settings in the default operating system install.
- Limiting the default set of active services to reduce attack surface.

Despite OpenBSD's persistent focus on security, a number of relatively recent hardware-level security issues, including Spectre [5] and Meltdown [6], have shown that software systems are only demonstrably as secure as their underlying hardware platforms. This brings into question the security practices of existing hardware platforms and whether existing architectures can adapt to a changing security landscape.

1.3 Justification

The rise in prevalence of obscure hardware-level security problems in recent years has come to show that security is not a single faceted problem and cannot be completely addressed from within a single context. The security of software platforms such as the OpenBSD operating system depends heavily on the security of their underlying hardware platforms. The RISC-V ISA's promise to remain truly open and transparent is expected to help lead the development of security-hardened hardware platforms. This project takes the first step in that direction by porting a full OpenBSD operating system to the RISC-V ISA, starting with the OpenBSD kernel.

2 TECHNOLOGIES DESCRIPTION

2.1 Toolchain

2.1.1 LLVM

The LLVM Project is a collection of modular and reusable compiler and toolchain technologies [7]. Both LLVM and the GNU Compiler Collection (GCC) support the RISC-V ISA as of their latest releases. LLVM is leveraged in this project in large part due to a general trend seen across the BSDs moving away from GCC in favor of LLVM[2]. More importantly, RISC-V has been promoted from an experimentally-supported target to an officially-supported target starting with LLVM 9.

The LLVM Project toolchain provides a vast suite of build tools such as a C compiler (Clang), linker (LLD), debugger (LLDB), and more. This section focuses on only the subset of these tools which were critical to successfully porting the OpenBSD kernel.

2.1.1.1 Clang: The front-end compiler infrastructure of the LLVM Project for C and C-based languages [10]. The Clang compiler differs from GCC, its GNU counterpart, in its modular approach to the three-phase compiler pipeline. A three-phase compiler consists of a front end, an optimizer, and a back end as shown in Figure 1.

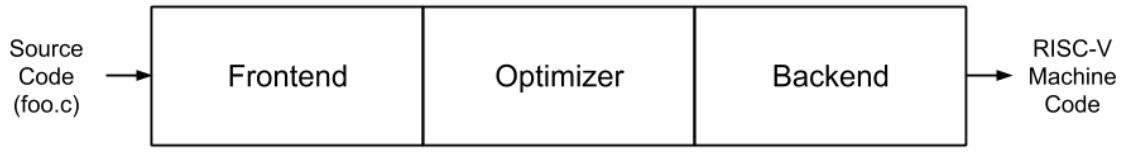

Fig. 1. Components of three-phase compiler.

The LLVM utilizes a modular approach to break the pipeline into three independent phases. Clang is one example of a language front end for the LLVM compiler toolchain. It is responsible for transforming C-based source code into LLVM Intermediate

2. FreeBSD fully transitioned to LLVM in 2012 [8] and OpenBSD's x86 variant transitioned to LLVM in 2017 [9].

Representation (IR). LLVM IR is a low-level RISC-like virtual instruction set which represents a program as it passes through the optimizer and the back end of the LLVM compiler pipeline. IR is the key to LLVM's modularity. A front end can be written to leverage the existing LLVM common optimizer and back end. Similarly, a new back end can be written to generate machine code for a new target machine with a specific architecture, such as the RISC-V ISA.

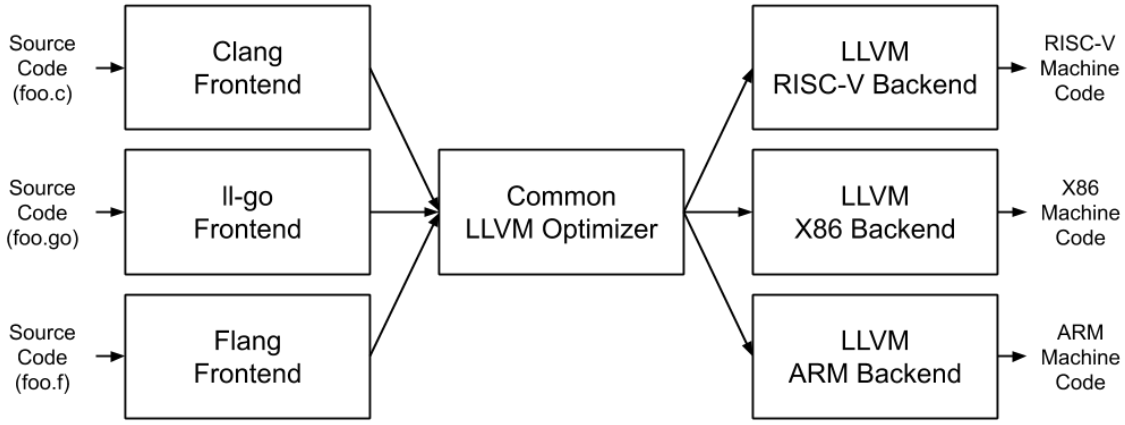

Fig. 2. LLVM's modular approach to the three-phase compiler.

In this project, LLVM is leveraged as the cross-toolchain for building the OpenBSD kernel for the RISC-V architecture. The LLVM toolchain compilation procedure and patches added in this project are further detailed in Section 5.1.2.

2.1.1.2 LLD: The platform-independent linker provided by the LLVM compiler toolchain. LLD is responsible for combining the machine code modules and libraries produced by the Clang front-end compiler into an executable format. It is designed as a drop-in replacement for the GNU linker which makes it easy to be inserted into the build process.

LLD is leveraged in this project to combine all machine-independent and machine-dependent kernel modules into a single compiled OpenBSD kernel file (`bsd`) as shown in Figure 3.

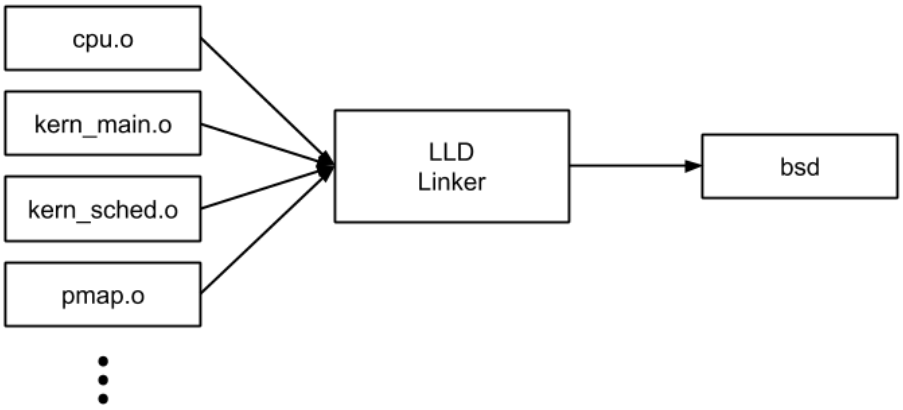

Fig. 3. LLD combines OpenBSD kernel modules into an executable binary.

2.1.2 QEMU

QEMU is a flexible open source machine emulator and virtualizer [11]. QEMU can perform a full system emulation of non-native hardware targets. For RISC-V, QEMU supports emulating either a generic virtual hardware platform or some commercially available developer kits such as the SiFive Freedom Unleashed U540 platform. QEMU support for RISC-V architecture is described in more detail in [12].

Hardware virtualization is essential to success of this project due to lack of access to physical RISC-V hardware. Whereas 64-bit x86 processors are easily accessible via cloud providers, RISC-V hardware is not yet widely available through consumer channels. Even though companies such as SiFive sell RISC-V hardware development kits, these kits are still much more expensive than mass-produced hardware based on x86 or ARM. Hence QEMU is utilized as a substitute for real RISC-V hardware to run and debug the ported OpenBSD kernel.

2.1.3 GDB

The GNU Debugger (GDB) is a powerful debugging tool that supports a number of target architectures. GDB provides useful commands to inspect, single-step, and debug applications. Most importantly, GDB offers functionality to perform remote debugging which enables debugging virtual RISC-V targets via the QEMU GDB stub as shown in Figure 4.

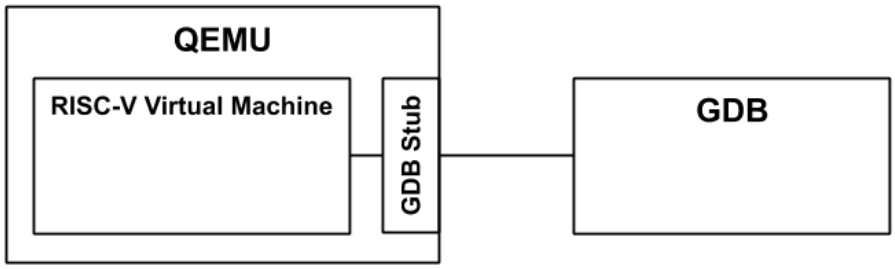

Fig. 4. Remote debugging QEMU via the GDB stub.

GDB was selected as the remote debugger for this project despite the heavy investment in the LLVM toolchain. In practice, GDB proved to be more reliable than the LLVM equivalent for remote debugging the OpenBSD kernel via the QEMU GDB stub.

2.2 RISC-V

2.2.1 Privilege Modes

Privilege modes are a hardware-level security mechanism provided by the RISC-V ISA. This mechanism plays an important role in providing isolation between different software layers. The *RISC-V Privileged Architecture Specification* defines three privilege modes: *Machine (M) mode*, *Supervisor (S) mode*, and *User (U) mode*.

Not all implementations of the RISC-V ISA support all these three privilege modes. An embedded system, for example, typically only implements one or two of these privilege modes as shown in Table 1. Systems intending to run a full Unix-like operating system such as OpenBSD need to implement all three privilege modes.

Table 1
Supported Combinations of RISC-V Privilege Modes

Number of Levels	Supported Modes	Intended Usage
1	M	Simple embedded systems
2	M, U	Secure embedded systems
3	M, S, U	Systems running Unix-like operating systems

The privilege modes defined by the RISC-V ISA are strictly ordered. A higher privilege mode has access to a superset of features and permissions granted by all lower privilege modes. For example, Machine-level instructions can only be executed in machine-mode. However, user-level instructions can be executed in machine-mode, supervisor-mode, and user-mode. Virtual memory address translation is the only exception to this rule in that it is supported in supervisor-mode and user-mode but not machine-mode. The role of each privilege mode within the context of a system running a traditional operating system is illustrated in Figure 5.

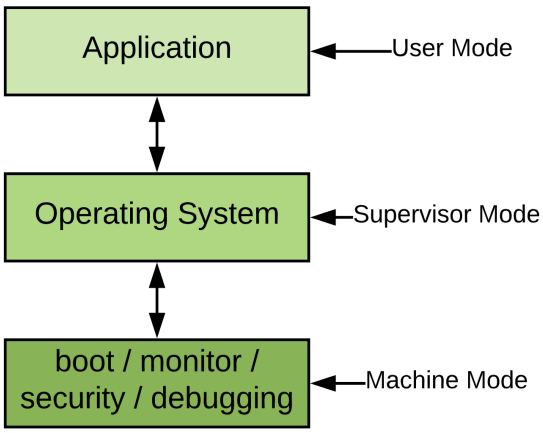

Fig. 5. HW-OS-App three layer hierarchy.

2.2.1.1 Machine-Mode: The highest privilege mode intended for bare-metal / first-stage bootloader usage. Machine-mode provides unfettered access to the entire machine. All RISC-V HARTs (short for HARdware Thread) start in machine-mode and

can drop into lower privilege modes when entering lesser-privileged software layers. Within the scope of this project, the Berkeley Boot Loader (BBL) runs in machine-mode and is responsible for dealing with traps to machine-mode triggered by the lower privilege modes. BBL loads the OpenBSD kernel into system memory and drops into supervisor-mode upon entry into the OpenBSD kernel.

2.2.1.2 Supervisor-Mode: The middle privilege mode intended for operating system usage. Supervisor-mode provides access to most privileged instructions and system I/O functions and supports changing the virtual memory address translation scheme. Within the scope of this project, the OpenBSD kernel runs in supervisor-mode and is responsible for dealing with traps triggered by user-mode processes. The OpenBSD kernel loads user programs into system memory and drops into user-mode when entering a user process.

2.2.1.3 User-Mode: The lowest privilege mode intended for conventional application usage. User-mode is restrictive and is largely controlled by the operating system kernel which runs in supervisor-mode. User-mode applications are given access to unprivileged instructions and access system devices indirectly via the operating system kernel. Within the scope of this project, user-mode processes spawned by the OpenBSD kernel run in user-mode. The OpenBSD kernel running in supervisor-mode is given fine-grained control over user-mode processes.

2.2.2 Execution Environments

An execution environment refers to the context under which some piece of software runs. A piece of software interacts with its execution environment via a binary interface, which defines the set of instructions and higher-level functions provided by the execution environment. RISC-V software remains portable across execution environments which implement a specific binary interface.

Porting an operating system to RISC-V involves dealing with two execution environments: the Supervisor Execution Environment (SEE) and the Application Execution Environment (AEE). The relationship between these environments and the operating system is shown in Figure 6.

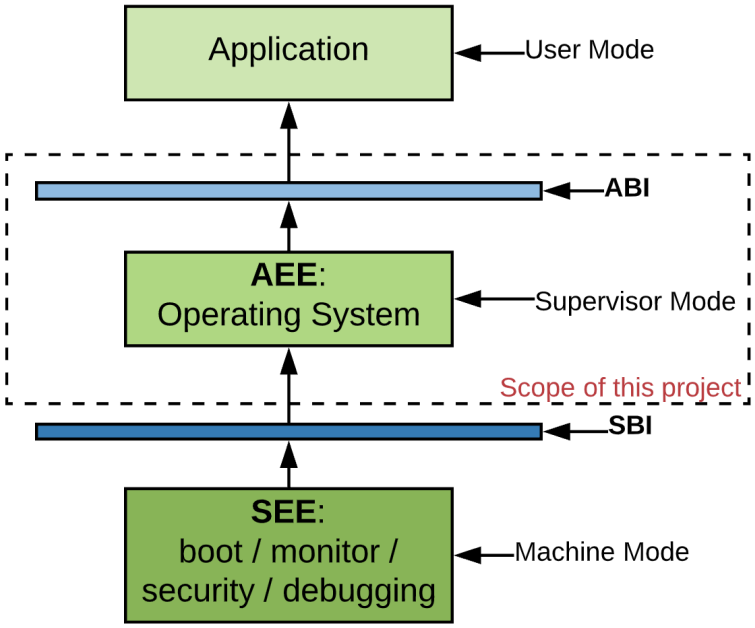

Fig. 6. SEE-AEE-App three layer hierarchy.

2.2.2.1 Supervisor Execution Environment: The supervisor execution environment is the context under which an operating system executes. The SEE can be a simple bootloader or a hypervisor-provided virtual machine. An operating system interacts with its SEE via the Supervisor Binary Interface (SBI) which comprises the user-level and supervisor-level ISA along with a defined set of SBI functions [13]. The SBI represents the compatibility layer between an operating system and its SEE. An operating system binary compiled for a single SBI can be run on any SEE as long as the SEE exposes the correct SBI. In this project, the OpenBSD kernel is adapted to the SBI exposed by the Berkeley Boot Loader SEE.

2.2.2.2 Application Execution Environment:

The application execution environment is the context under which user-mode programs execute. AEE is normally provided by an operating system such as OpenBSD. Similar to SEE, AEE exposes an Application Binary Interface (ABI) which comprises the user-level ISA and a defined set of ABI function calls, often referred to as system calls (or syscalls for short). The ABI varies across operating systems but is generally stable between releases of an operating system. The portability of an application binary is limited only by the target ABI. The OpenBSD AEE and ABI are touched upon in the implementation of the machine-dependent aspects of the OpenBSD kernel system call handler.

3 PROJECT OVERVIEW

3.1 Development Environment

- A host machine running OpenBSD for source code editing and compilation.
- A cross-toolchain to compile kernel source code to non-native RISC-V targets.
- A virtual or physical RISC-V hardware platform with Memory Management Unit (MMU) support for `Sv39`[3].
- A bootloader providing the supervisor execution environment for the OS kernel.
- A debugger to remotely debug a RISC-V hardware target and inspect the kernel.

3.2 Functional Requirements

3.2.1 Essential

- Provide "*config(8)*"-syntax based kernel configuration.
- Provide "*files.config(5)*"-based file lists for Makefile generation.
- Construct machine-dependent kernel components to provide machine-independent kernel components with an interface to hardware, including:
  - System Bootstrap
  - System Memory Management
  - Kernel and User Process Management
  - Interrupt and Exception Handling
  - Device Configuration and Management
- Boot OpenBSD kernel to copyright message on virtual RISC-V hardware platform.

3.2.2 Desired

- Support Symmetric Multi-Processing (SMP) for multi-processor RISC-V hardware.
- Boot OpenBSD kernel to copyright message on physical RISC-V hardware platform.

3. `Sv39` refers to the page-based virtual memory address translation mode that supports 39 addressable bits of virtual memory address space. `Sv48`, which appears later in this report, refers to the page-based virtual memory address translation mode that supports 48 addressable bits of virtual memory address space.

3.2.3 Optional

- Cross-compile OpenBSD userland components (libc, libcompiler_rt, etc.)
- Boot OpenBSD kernel through to multi-user login prompt on RISC-V hardware.
- Bootstrap build toolchain inside OpenBSD running on RISC-V hardware.
- Provide working OpenBSD RISC-V install media.
- Publish one or more compiled software binaries for OpenBSD on RISC-V.
- Add device support for non-essential RISC-V hardware devices.

3.3 Non-functional Requirements

3.3.1 Essential

- Standardization
  - Adhere to *POSIX* and *ANSI* standards.
- Quality
  - Discourage use of hacks and workarounds.
  - Adhere to OpenBSD source code auditing guidance.
  - Provide well-documented source code.

3.3.2 Desired

- Stability
  - OpenBSD kernel remains stable under steady-state operation.
- Security
  - Apply security best practices while implementing kernel code.

3.3.3 Optional

- Performance
  - Keep OpenBSD kernel overhead to a minimum, if possible.
- Consistency
  - Keep the ”look and feel” of OpenBSD on RISC-V consistent with other ISAs.

3.4 Milestones

The primary goal of this project is to port the OpenBSD kernel to the RISC-V ISA. A number of smaller milestones pave the way to this goal and beyond. These milestones track work items from early development environment setup through stretch goals which identify nice-to-have features that are not considered critical to a successful OpenBSD kernel port. The milestones are enumerated below:

1) Development Environment
   a) Configure and build Clang+LLVM toolchain with support for the RISC-V ISA.
   b) Configure and build QEMU with support for emulating RISC-V hardware.
   c) Configure and build BBL to serve as bootloader for the OpenBSD kernel.
   d) Configure and build GDB to remotely debug the OpenBSD kernel on QEMU.
2) OpenBSD Kernel Source Tree
   a) Populate OpenBSD tree with minimum viable RISC-V kernel build boilerplate.
   b) Stub out required machine-dependent headers and source files for kernel build.
   c) Define, derive, or otherwise write out machine-dependent code, including:
      - Early machine initialization code (*locore.S*)
      - Standard system data types
      - RISC-V machine registers
      - System exception & interrupts handlers
      - Per-CPU information structure
      - Process control structures
      - Stack frames for trap, exception, signal, context switch, etc.
      - Memory management structures
      - RISC-V hardware device drivers
3) OpenBSD Kernel Cross-Compilation & Linking
   a) Resolve remaining unresolved kernel symbols during initial kernel compilation.

b) Generate OpenBSD RISC-V kernel binary using RISC-V cross-toolchain.

c) Build BBL with OpenBSD RISC-V kernel payload.

4) OpenBSD Kernel Bring-up

a) Hack at kernel implementation until OpenBSD kernel boots to copyright message.

b) Boot OpenBSD through `autoconf` (device probing and configuration)

c) Bring up additional system devices, including:

- Virtual block devices (e.g., VIRTIO BLK, the VirtIO Block Driver)
- Virtual network device (e.g., VTNET, the VirtIO Ethernet Driver)

d) Fork and exec to `init` process

5) OpenBSD Multi-User *[Stretch Goal]*

a) Build OpenBSD userland components

b) Support Simultaneous Multi-Processing (Optional)

6) OpenBSD RISC-V Installer & Bootloader *[Stretch Goal]*

a) Write Installer for the RISC-V OpenBSD Port

b) Write Bootloader for the RISC-V OpenBSD Port

3.5 Dependencies

Porting the OpenBSD kernel to RISC-V carries a number of external dependencies, particularly around the availability of tooling that supports the RISC-V ISA. The RISC-V ISA is incredibly new relative to the other mainstream ISAs. The Intel x86 family of ISAs goes back as far as 1978 and the ARM ISA dates back to 1985. These ISAs have had plentiful time to mature and grow their market share over a few decades. Support for the RISC-V ISA, while growing, is still largely experimental across various virtualization products and build tools.

The success of this project depended heavily on support for the RISC-V ISA in virtualization tools such as QEMU. Companies including SiFive have made great strides in building out hardware platforms for RISC-V development. Unfortunately, these

development platforms are far too expensive to justify for purchase within the scope of this project. Instead, QEMU allows for emulation of virtual RISC-V hardware targets which are sufficient for kernel bootstrapping. While efforts to support the RISC-V ISA in QEMU only began in early 2018, other BSD-based operating systems including FreeBSD were successful in bootstrapping their operating system kernels on QEMU's virtual RISC-V hardware. As such, it is optimistic that QEMU's support for virtual RISC-V hardware targets would also be suitable in the work to bootstrap the OpenBSD kernel on RISC-V.

In addition to virtualization tools, the success of this project depended on support for the RISC-V ISA in build toolchains such as LLVM. The decision to bootstrap with the LLVM instead of GNU toolchain did carry some risk at the early stages of the project as LLVM support for the RISC-V ISA was still considered experimental until September 2019. The availability of the GNU toolchain as a fallback to LLVM helped to reassure that this project would not come to a complete standstill due to toolchain issues.

4 PROJECT ARCHITECTURE

4.1 Overview

Operating systems are responsible for interfacing high-level applications with the low-level hardware resources provided by a computer system. The OS kernel serves user-mode applications by exposing a well-documented set of functions via an application binary interface which sits atop the machine-independent (MI) layer of the kernel. The MI layer of the OS kernel relies on a machine-dependent (MD) layer which provides abstractions over the underlying hardware resources. This multi-layer hierarchy is depicted in Figure 7.

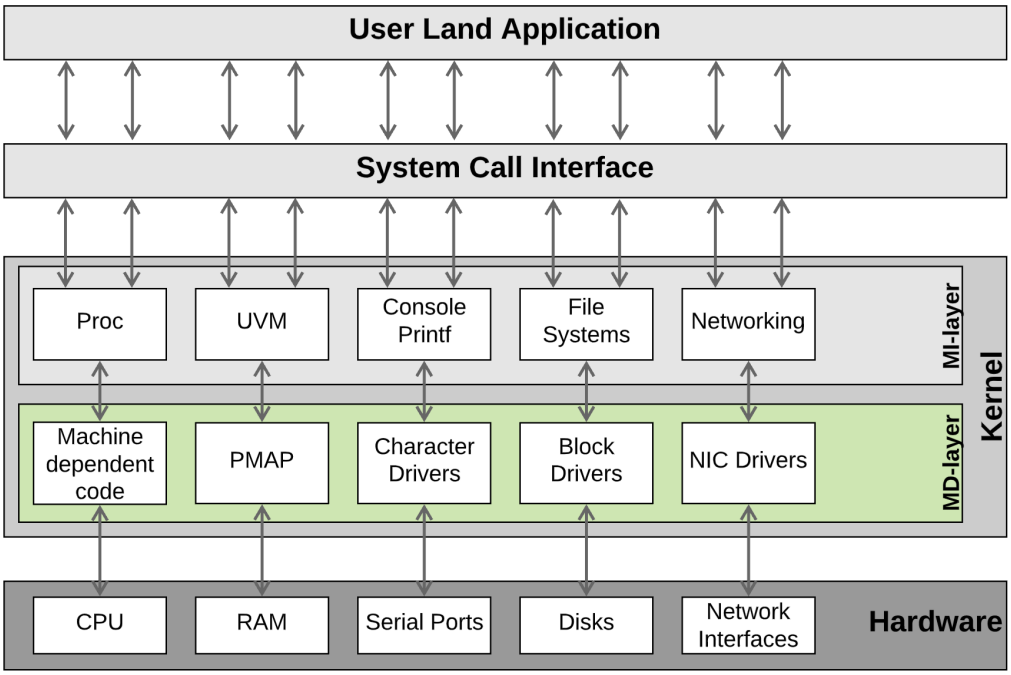

Fig. 7. OS connects hardware and applications.

Porting an existing OS to a new architecture is a very different problem than writing an OS from scratch. The former focuses almost entirely on the MD layer of the OS kernel. The basic strategy involves rewriting the architecture-specific routines and drivers of the MD layer to provide an interface to the hardware for the MI layer. The MI layer remains

largely unchanged. OpenBSD's MD layer can be divided into five subsystems, as shown in Figure 8. These subsystems are summarized in the sections that follow. Fine-grained implementation details for each are presented in Section 5.

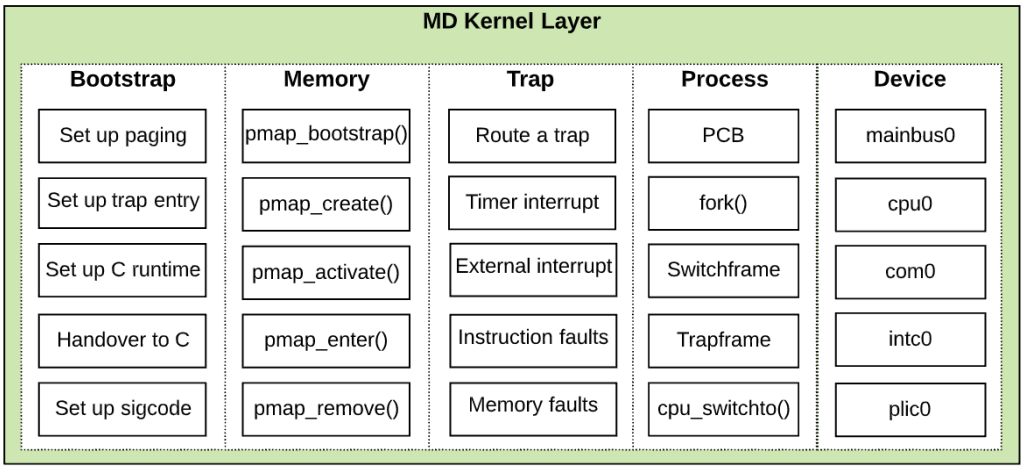

Fig. 8. Decomposition of machine-dependent kernel layer.

4.2 Bootstrap Subsystems

Booting a Unix-like operating system on a RISC-V machine involves a multi-stage bootstrap process. For the OpenBSD operating system, the bootstrap process finishes upon entry into the MI OS kernel initialization routine. This bootstrap procedure can be divided into three stages:

1) Stage 1 Bootstrap: Hardware Reset
2) Stage 2 Bootstrap: Berkeley Boot Loader
3) Stage 3 Bootstrap: OS Kernel Bootstrap (OpenBSD)

4.2.1 Stage 1 Bootstrap: Hardware Reset

Stage 1 of system bootstrap begins from the moment that power is applied to a RISC-V processor. Applying power to the RISC-V circuitry generates a reset signal which places the hardware into a known starting state. For virtual hardware provided by

QEMU, this hardware reset is emulated in software. The routine that implements this, `riscv_cpu_reset()`, performs the following actions:

- Set privilege level to machine-mode.
- Set `mstatus.MIE` to 0 to disable interrupts.
- Set `mstatus.MPRV` to 0 to turns off address translation.
- Initialize `mcause` register to 0.
- Set `pc` to `0x00001000` which targets the trampoline code in system ROM.

The trampoline code in system ROM acts as the first-stage bootloader. This small bit of code builds the Flattened Device Tree (FDT) to describe system hardware and loads the second-stage bootloader (BBL) into memory at address `0x80000000`. The first-stage then yields control to the second-stage bootloader, passing along the address of the FDT.

4.2.2 Stage 2 Bootstrap: Berkeley Bootloader

The second stage of system bootstraps begins upon entry into the Berkeley Bootloader. BBL runs entirely in machine-mode, the highest RISC-V privilege mode, and provides the supervisor execution environment for the OS kernel. As described in paragraph 2.2.2.1, the combination of SEE and SBI is what allows a compiled kernel binary to remain portable across RISC-V hardware platforms. BBL abstracts away the underlying hardware and, instead, exposes a number of SBI functions which allow the OS kernel to interface with the machine-mode layer. BBL provides the following functionalities:

- Emulation of illegal instructions.
- A machine-mode trap handler for machine-level exceptions and interrupts.
- Routines to manage periodic timer interrupts.
- Chain loading and access to the initial console device.

As mentioned previously, the first stage of the system bootstrap process loads BBL at base address `0x80000000`. The base address also serves as the entry point into BBL. Within the BBL executable, a `reset_vector` is mapped to this base address which

kicks off the BBL bootstrapping process. This bootstrapping procedure performs roughly the following sequence of events:

1) Selects one RISC-V HART to serve as the main HART for BBL. All other RISC-V HARTs are temporarily put to sleep until BBL is ready to transfer control to the OS kernel.
2) Filters the FDT received from the previous-stage bootloader. This removes references to some hardware and other machine-specific details that are irrelevant during supervisor-mode operation.
3) Wakes up all other HARTs from sleep. All HARTs join in executing the remaining steps up through OS kernel entry.
4) Reads `mhartid` Control and Status Register (CSR). The value read from this register is later passed to the OS kernel to uniquely identify each RISC-V HART.
5) Configures physical memory permission (PMP) for supervisor-mode to allow full access to system memory resources.
6) Configures machine-mode trap handlers and stack. The BBL-provided trap handler code deals with machine-mode exceptions and interrupts.
7) Drops the HART into supervisor-mode before jumping to the start of the BBL payload. In this project, the payload is a compiled OpenBSD kernel.

4.2.3 Stage 3 Bootstrap: OS Kernel Bootstrap

Stage 3 of system bootstrap begins upon entry into the OS kernel starting with *locore.S*. The OS kernel runs in supervisor-mode within the SEE provided by BBL. The kernel bootstrap procedure handles the bulk of the workload in preparing the system to initialize the OpenBSD kernel. This process includes:

- Setting up initial kernel page tables.
- Performing the switch into paged virtual memory mode.
- Probing and attaching hardware devices.

- Configuring the supervisor-mode trap handler.
- Initializing the global pointer and stack pointer.
- Propagating kernel arguments to the MI kernel.
- Jumping into the MI kernel initialization routine.

4.3 Memory Subsystem

OpenBSD uses the UVM virtual memory system to manage system memory resources across multiple virtual memory address spaces [14]. UVM provides a high-level machine-independent view of system memory to the kernel as well as user-mode processes. UVM is a multi-layer system consisting of a large machine-independent layer, which manages virtual address spaces, and a lightweight machine-dependent layer, which serves as the interface to the system's Memory Management Unit.

The UVM external interface exposes high-level memory operations such as allocating pageable or non-pageable memory regions, setting permissions on those memory regions, and adjusting the size of allocated memory regions. Internally, the machine-independent layer of UVM manages software structures which represent the layout of the virtual address space for each process on the system. As necessary, the machine-independent layer delegates to the machine-dependent layer to insert or remove virtual-to-physical address mappings.

The machine-dependent layer modifies in-memory page tables which affect the virtual memory layout exposed by the MMU. The machine-dependent layer must understand the memory address translation schemes supported by the target architecture. The relationship between user-mode applications, the kernel, and the sub-layers of the memory management system is illustrated in Figure 9.

The 64-bit variant of the RISC-V ISA supports three memory address translation modes. As of publication of "*RISC-V Privileged Architecture Specification v1.11*", these modes include: `Bare`, `Sv39`, and `Sv48`. The specification also reserves two additional

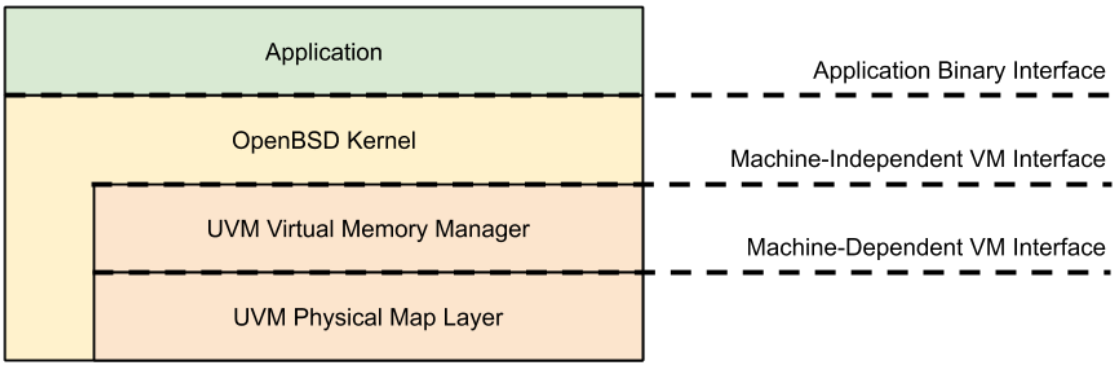

Fig. 9. Multi-layer memory management system.

modes for future use: `Sv57` and `Sv64`. The `Bare` mode provides a simple identity mapping between virtual and physical memory addresses without memory protection. The `Sv39` and `Sv48` modes are 4 KiB page-based memory address translation schemes.

In porting the OpenBSD kernel, only `Sv39` mode is supported to keep a tight bound on complexity in the memory subsystem. `Sv39` mode is similar in nature to the page-based memory schemes used in other architectures, such as 64-bit ARM. `Sv39` mode uses a 3-level page table walk to translate a virtual address to a physical address. The virtual to physical address translation procedure for `Sv39` is portrayed in Figure 10.

As shown in Figure 10, a virtual address with 39 addressable bits maps to a physical address of 56 addressable bits. During translation, a page fault can occur if an invalid entry is encountered or if a permissions check fails. Page faults which arise during user-mode operation will trigger a trap to the supervisor trap handler, which further delegates to UVM to resolve the fault. A page fault may be resolved by mapping a missing entry into the page table, or by terminating the process which triggered the fault. Assuming the virtual to physical address translation does not cause a page fault, the translation of a virtual address to a physical address can be visualized as in Figure 11.

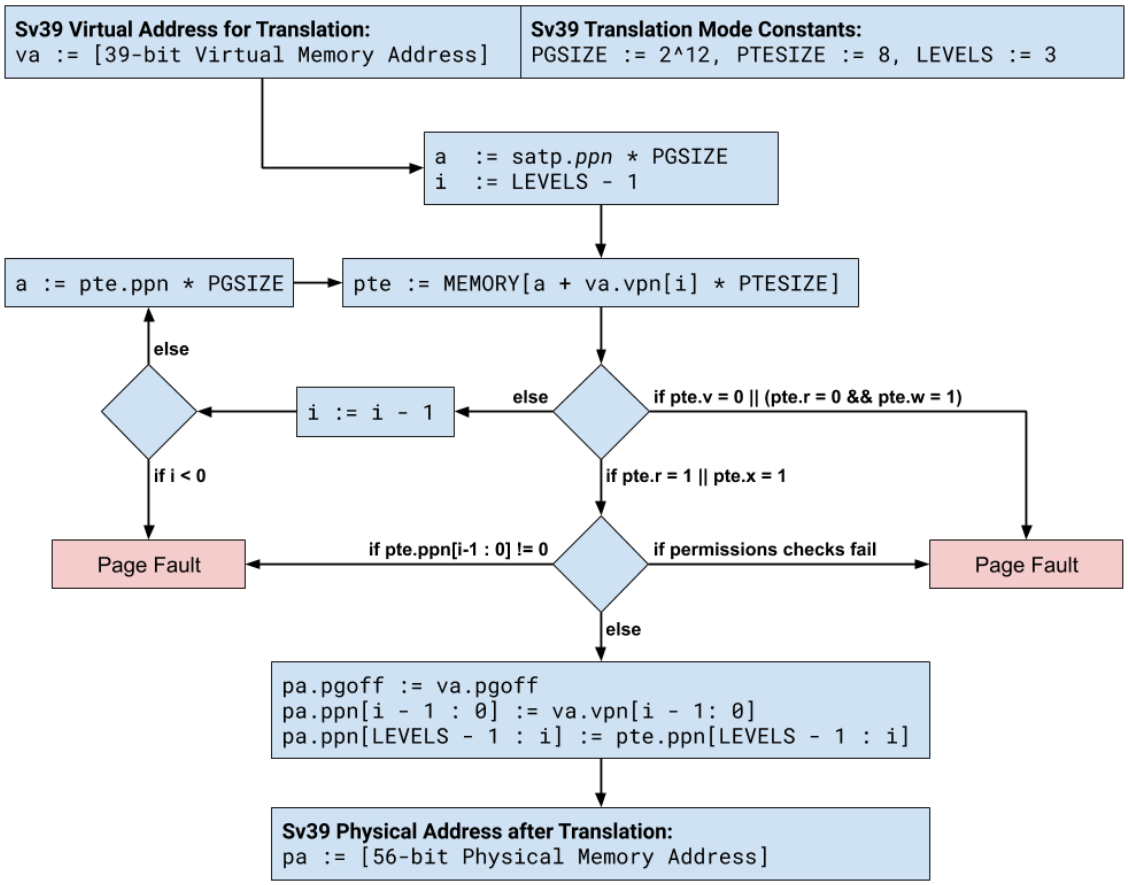

Fig. 10. The Sv39 memory address translation state machine.

Virtual to physical address translation allows the OpenBSD kernel to share limited memory resources among a large number of resident processes. Each resident process is assigned a dedicated virtual address space identifier. The layout of each virtual address space is dictated by the layout as defined in the machine-dependent kernel source, and the layout is baked into the compiled kernel. This porting work inherits the virtual memory layout adopted by FreeBSD's RISC-V port. The virtual address space layout is depicted in Figure 12.

The `Sv39` memory address translation mode provides 512 GiB of addressable virtual address space. The lowest 256 GiB of virtual address space is reserved for user-mode

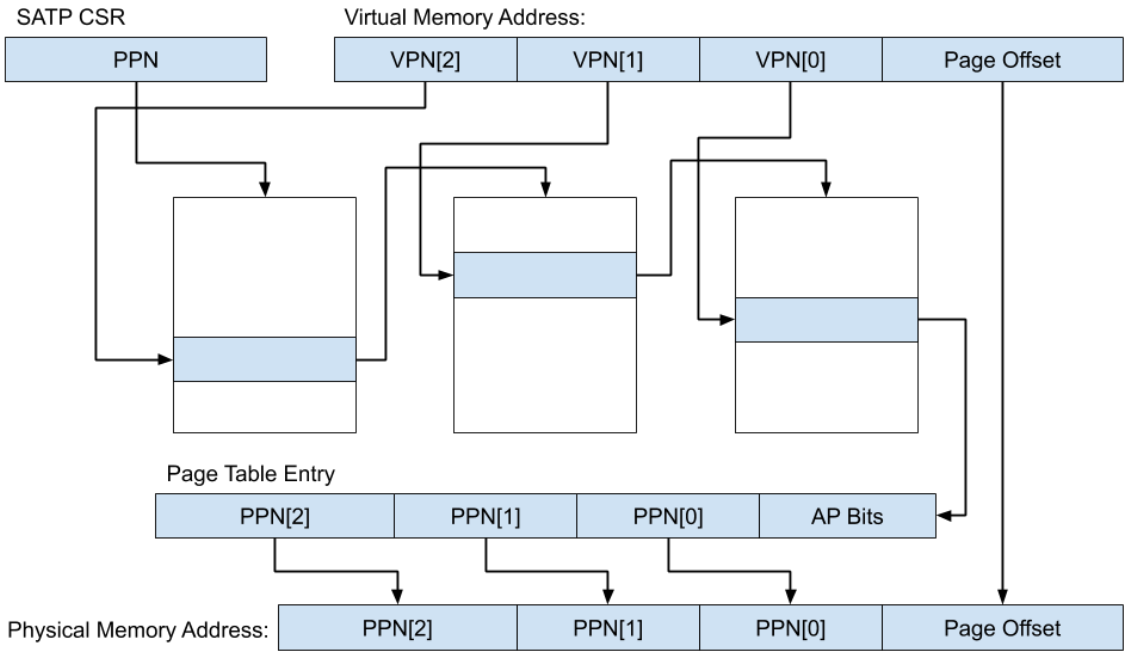

Fig. 11. Sv39 memory address translation for a 4 KiB page.

processes. The highest 256 GiB of virtual address space is reserved for the kernel. 32 GiB of the kernel-reserved space, starting from `0xFFFFFFC000000000`, is assigned to the *Kernel Map* to hold kernel code and data. Another 128 GiB of the kernel-reserved space, starting from `0xFFFFFFD000000000`, is assigned to the *Direct Map* which is an offset-identity mapping of physical memory. The remaining 96 GiB of kernel-reserved space is unused.

4.4 Trap Subsystem

A trap is a form of synchronous event which causes a RISC-V HART to halt its current activities and immediately transfer control to a trap handler. This document focuses exclusively on supervisor-mode traps which are handled by the OpenBSD kernel. This document uses the terms *exception*, *interrupt*, and *trap* in alignment with the

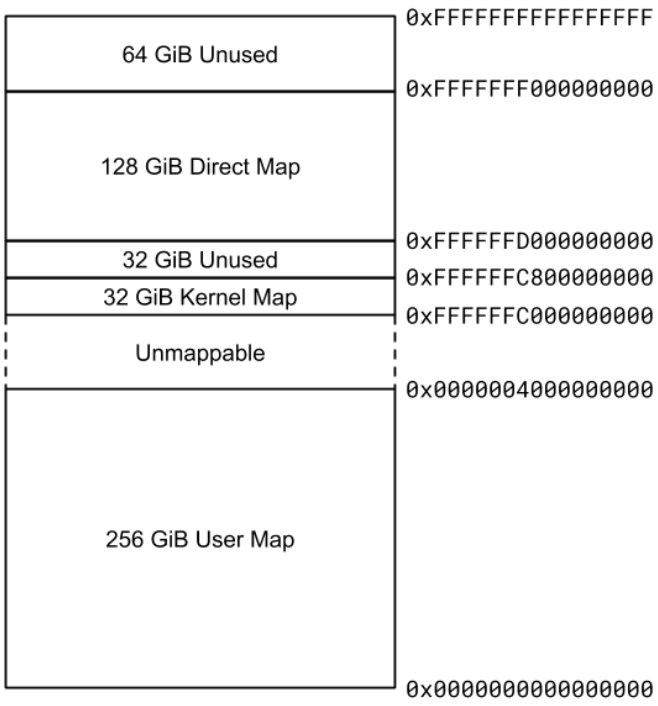

Fig. 12. Virtual address space layout (Sv39) for the ported OpenBSD.

definitions provided by *The RISC-V Instruction Set Manual, Volume I: User-Level ISA* [15]:

- "we use the term *exception* to refer to an unusual condition occurring at run time associated with an instruction in the current RISC-V thread."
- "we use the term *interrupt* to refer to an external asynchronous event that may cause a RISC-V HART to experience an unexpected transfer of control."
- "we use the term *trap* to refer to the transfer of control to a trap handler caused by either an exception or an interrupt."

4.4.1 Trap CSRs

The CSRs provided by the RISC-V ISA that are used for supervisor-mode trap configuration and handling are listed below:

- The Supervisor Status (`sstatus`) Register contains some bits for controlling system trap behavior. This includes bits for controlling whether interrupts are enabled (`SIE`, `UIE`), bits for indicating whether interrupts were enabled prior to entering the supervisor-mode trap handler (`SPIE`, `UPIE`), and a bit for indicating the privilege mode prior to entering the supervisor-mode trap handler (`SPP`).
- The Supervisor Cause (`scause`) Register indicates the cause for a trap. This register dedicates one bit for distinguishing between traps caused by exceptions and traps caused by interrupts. The remaining bits are used to uniquely identify the exact type of exception or interrupt that occurred. This register takes on one of the states as enumerated in Table 2.
- The Supervisor Trap Value (`stval`) Register holds an exception-specific value that can be used by the trap handler to service the exception. During a misaligned instruction exception, for example, this register will hold the virtual address which triggered the exception.
- The Supervisor Trap Vector (`stvec`) Register holds the base address of trap vector and the trap vectoring mode. The trap vectoring supports *Direct*-mode and *Vectored*-mode.
  - *Direct*-mode is the default value. In *Direct*-mode, `stvec` holds the base address of the overall trap handler entry, and software is responsible for figuring out the source of exception/interrupt and acting accordingly.
  - *Vectored*-mode introduces a method to create a vector table that hardware uses to achieve lower interrupt handling latency. When an interrupt occurs in *Vectored*-mode, the `pc` will get assigned by the hardware to the address stored in the vector table entry corresponding to the interrupt ID.

- The Supervisor Interrupt Enable (`sie`) Register allows for specific subsets of interrupts to be enabled or disabled. These interrupts include software interrupts, timer interrupts, and external interrupts.
- The Supervisor Interrupt Pending (`sip`) Register contains information on pending interrupts. As with `sie`, these interrupts include software interrupts, timer interrupts, and external interrupts.
- The Supervisor Exception Program Counter (`sepc`) Register preserves the program counter (`pc`) at the moment the trap occurs. This register can be modified during trap handling to adjust the program counter upon return from the trap handler.

4.4.2 Trap Entry and Exit

RISC-V ISA specification aims to keep trap handling simple. The state of the hardware is largely unchanged upon entry into the supervisor trap handler. Both hardware and software benefit from this. Implementing trap handling in hardware is not only easier, but supervisor software does not have to work around hardware-level assumptions.

All traps to supervisor-mode arrive at the OS kernel via the kernel trap entry point installed in `stvec` during early bootstrap, as described in Section 5.3.5. Upon trap to the supervisor, the hardware needs to complete the following tasks [16]:

- Save `pc` to `sepc`.
- Save privilege mode to `sstatus.SPP`.
- Save `sstatus.SIE` to `sstatus.SPIE`.
- Set `sstatus.SIE` to 0.
- Set `scause` to trap reason.
- Set `stval` with exception-specific value, if applicable.
- Set `pc` to trap handler base address which has been installed to `stvec`.

After making the state changes above, the hardware immediately begins executing the kernel trap handler code. The initial entry point into the kernel trap handler is

Table 2
RISC-V Supervisor Cause Register (`scause`) after Trap

Interrupt	Code	Description
1	0	User software interrupt
1	1	Supervisor software interrupt
1	2-3	*Reserved for future standard use*
1	4	User timer interrupt
1	5	Supervisor timer interrupt
1	6-7	*Reserved for future standard use*
1	8	User external interrupt
1	9	Supervisor external interrupt
1	10-15	*Reserved for future standard use*
1	$\geq$16	*Reserved for platform use*
0	0	Instruction address misaligned
0	1	Instruction access fault
0	2	Illegal instructions
0	3	Breakpoint
0	4	Load address misaligned
0	5	Load access fault
0	6	Store/AMO address misaligned
0	7	Store/AMO access fault
0	8	Environment call from U-mode
0	9	Environment call from S-mode
0	10-11	*Reserved for future standard use*
0	12	Instruction page fault
0	13	Load page fault
0	14	*Reserved for future standard use*
0	15	Store/AMO page fault
0	$\geq$16	*Reserved*

machine-dependent code provided by the OS kernel which performs the transition into the kernel. Since the changes to hardware state are limited, the kernel trap handler gets to decide what state is saved upon entry into the kernel.

In addition to supervisor-level trap CSRs, the Supervisor Scratch (`sscratch`) Register is often helpful in retaining state between entry into and exit from the kernel. The `sscratch` register has no ISA-designated purpose, and thus the OS kernel can use

it for any purpose. Within the OpenBSD trap handler, this register is used to quickly swap between the user stack and the kernel stack upon entry into and exit from the kernel.

Once the state of the active process is saved and the kernel stack is switched onto the HART, the trap handler attempts to route the trap within the kernel. The kernel reads the `scause` register to identify the exact cause of the trap and delegates to the appropriate sub-handler.

Upon completion of trap handling, control may return to the faulting process. The state of the faulting process is restored onto the HART and an `sret` instruction triggers the hardware to switch back to the process at its appropriate privilege mode. This equates to the following steps:

- Restore privilege mode from `sstatus.SPP`.
- Restore `sstatus.SIE` from `sstatus.SPIE`.
- Restore `pc` from `sepc`.

4.5 Process Subsystem

The OpenBSD process subsystem is responsible for managing the life cycle of processes within the operating system. Each process is an instance of an executable program and operates in its own virtual address space. A process is represented in a software structure, an instance of "`struct process`" which retains relevant information about the process. The structure includes, among other things, a pointer to the virtual address space structure used by the memory subsystem to manage virtual to physical address mappings for the process.

Processes spawn from parent processes in response to certain syscalls provided by the kernel. Child processes are typically spawned by some combination of the `fork()` + `exec()` syscalls. An example of this is shown in Figure 13. The call to `fork()` generates a new child process which is nearly identical to its parent and is almost always

followed up by a call to `exec()` to load an executable and trigger Address Space Layout Randomization (ASLR).

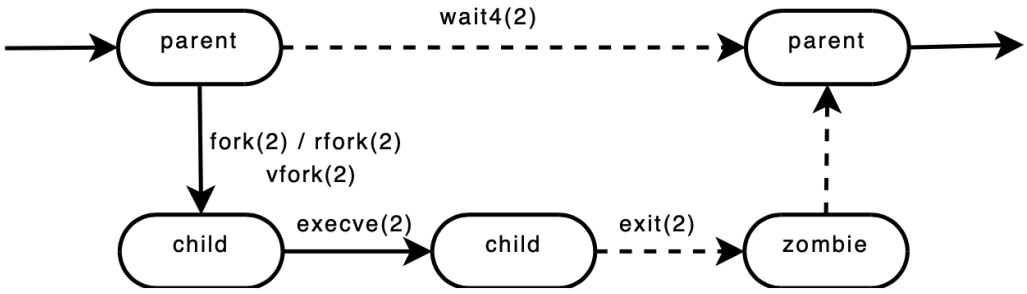

Fig. 13. Managing processes via system calls.

The OS kernel is responsible for managing access to hardware resources among processes. On a given system, there might be a hundred or more processes competing for access to just a handful of HARTs. To share HARTs between processes, the OS kernel time-slices access to the HARTs with help from a periodic hardware timer interrupt. Before a process is assigned to a HART, the OS kernel ensures that the periodic timer is primed so that a trap will eventually force the process to yield the HART back to the kernel scheduler. When the scheduling-related trap occurs, this signals to the kernel to swap out the active process for another process as seen in Figure 14.

The trap causes the state of the HART to be saved into the Process Control Block (PCB) to make way for a new process. The kernel scheduler selects another process to take control of the RISC-V HART. The previous state of the selected process is restored on the HART to complete the context switch into the new process. The machine-dependent aspects of the context switch are explored in more detail in Section 5.6.2.

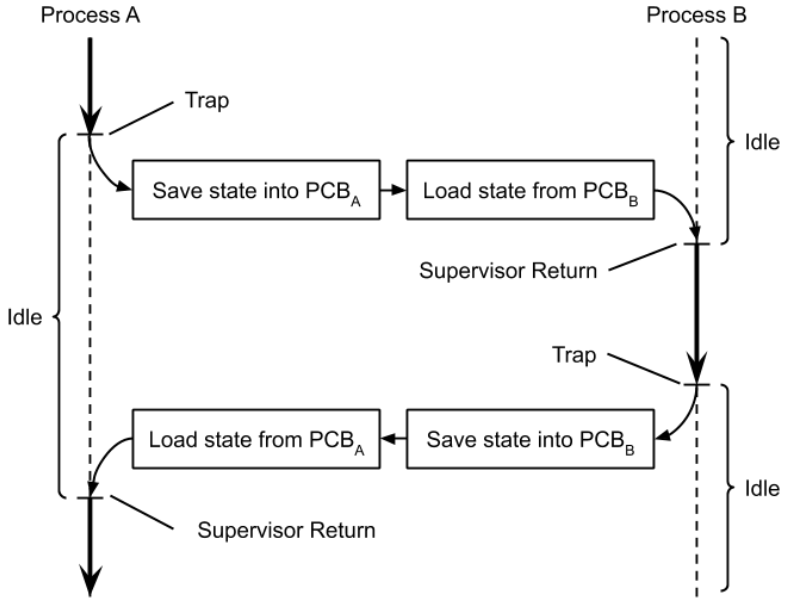

Fig. 14. Save and restore execution states in a context switch.

4.6 Device Subsystem

During system startup, the early system bootloaders enumerate system devices into a structure known as the FDT. The FDT describes system hardware including the RISC-V HARTs, the memory map, and the attached devices.

During OpenBSD initialization, the kernel runs through a process called *autoconf*. The FDT serves as a guide for the OS kernel to properly discover and configure hardware devices. During the *autoconf* process, the OS kernel traverses the FDT to match hardware devices with compatible drivers using the system configuration table, which is generated based on device definition in *files.riscv64* and device configuration in a file named *GENERIC*. The system configuration table describes the expected attachment points for various devices and associates them with an appropriate driver.

The *autoconf* process is recursive in nature. The process begins by traversing devices nodes attached at the root of the FDT. As device nodes are attached, their child nodes are

expanded and traversed. The key routines for the *autoconf* process are enumerated in Snippet 1.

```
void *
config_search(cfmatch_t func, struct device *parent, void *aux);

struct device *
config_attach(struct device *parent, void *match, void *aux, cfprint_t print)

struct device *
config_found_sm(struct device *parent, void *aux, cfprint_t print,
  cfmatch_t submatch);
```

Snippet 1. `autoconf` key routines.

Given a parent device, `config_search()` identifies the best-matched child device under a specific matching criteria. `config_attach()` will then attach the child device to its parent in three steps. First, it allocates memory for the device's software configuration structure (`config_make_softc()`). This structure is then filled with this device's hardware information as parsed from FDT. Lastly, the device-specific initialization logic is performed as provided by the device driver. At this stage, this device's child device will get attached recursively.

`config_found_sm()` locates the configuration data for a device and attaches it. This boils down to making calls to `config_search()` and `config_attach()`, consecutively. `autoconf` supports two modes of configuration: *direct* mode and *indirect* mode. When a child device is provided via the `aux` argument, *direct* mode is used and the device is attached directly. Otherwise, *indirect* mode is used which scans the bus to match and attach a compatible device.

The device attachment hierarchy of this project is demonstrated in Figure 15. This project provides definition, configuration, and driver-level code for machine-dependent devices identified in this hierarchy. This is presented in Section 5.7.

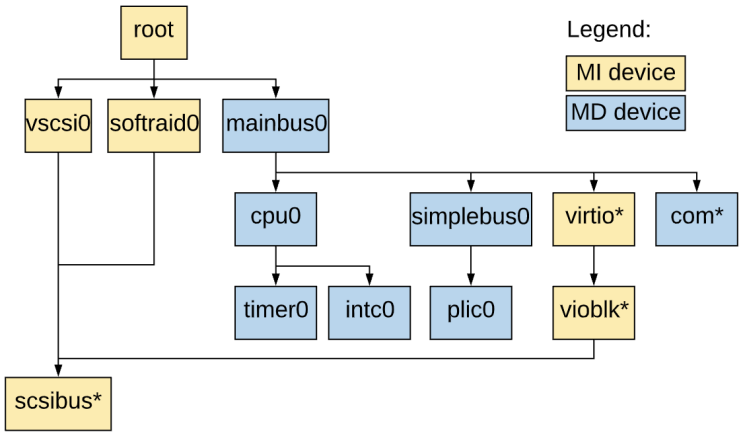

Fig. 15. The device attachment hierarchy.

5 PROJECT IMPLEMENTATION

5.1 Development Environment Setup

This section briefly describes how to set up a development environment for the purpose of porting OpenBSD to RISC-V ISA. An OpenBSD host machine is required for source code editing and kernel cross compilation. Cross-compilation tools are mandatory for building the OpenBSD kernel on an `x86` host for a `riscv64` target. The cross-compiled kernel is encapsulated as payload into BBL and runs on a QEMU virtual machine. GDB can then be attached to QEMU to single-step the OpenBSD kernel and inspect its functionality.

5.1.1 OpenBSD Machine

The OpenBSD machine can be bare-metal or a virtual machine. This machine is used for cross-compiling the OpenBSD kernel. The OpenBSD kernel building supports incremental compilation which reduces the time necessary to recompile the kernel for incremental updates to source code. However, some machine-dependent headers, such as *cpu.h*, are widely-referenced across the OpenBSD source tree. Modifications to widely-used headers can trigger re-compilation of large portions of the OpenBSD kernel which can drastically increase build time.

Running OpenBSD as a virtual machine on a cloud-provider allows for flexibility in tuning CPU and memory resources. The virtual machine can be scaled up for speed when running a full kernel build and scaled down for cost-saving when running incremental kernel builds. In this project, a customized OpenBSD installation image is created to install OpenBSD on a cloud-resident virtual machine [17].

5.1.2 RISC-V Cross-Toolchain

LLVM is selected as the RISC-V cross-toolchain for this project. The pre-compiled LLVM binaries in trusted `OpenBSD-current` software sources do not include stable

support for RISC-V. Similarly, the latest version of LLVM in the OpenBSD ports repository predates the inclusion of RISC-V as a stable target. LLVM 9 is the first LLVM release to support RISC-V as an officially-supported, non-experimental target. In this project, LLVM is compiled from source to obtain a working LLVM 9 toolchain with support for RISC-V[4]. The build configuration used to produce the LLVM toolchain used in this project is shown in Snippet 2.

```
# Run from within ${LLVM_PROJECT_ROOT}/build
$ cmake \
  -DLLVM_ENABLE_FFI:Bool=False \
  -DLLVM_ENABLE_TERMINFO:Bool=False \
  -DLLVM_ENABLE_RTTI:Bool=False \
  -DCMAKE_DISABLE_FIND_PACKAGE_LibXml2:Bool=True \
  -DCMAKE_DISABLE_FIND_PACKAGE_Z3:Bool=True \
  -DLLVM_TOOL_LLDB_BUILD:Bool=True \
  -DLLVM_BUILD_LLVM_DYLIB:Bool=True \
  -DLLVM_LINK_LLVM_DYLIB:Bool=True \
  -DGO_EXECUTABLE=GO_EXECUTABLE-NOTFOUND \
  -DCMAKE_DISABLE_FIND_PACKAGE_Backtrace:Bool=True \
  -DLLVM_TARGETS_TO_BUILD:String="X86;RISCV" \
  -DLLVM_ENABLE_PROJECTS:String="clang" \
  -DCMAKE_BUILD_TYPE=Release \
  -DCMAKE_INSTALL_PREFIX=/usr/local/riscv \
  -DCMAKE_CXX_FLAGS="-mno-retpoline" \
  -G Ninja \
  ../llvm
$ ninja -j8
```

Snippet 2. CMake configuration for the LLVM cross-toolchain.

A patch to the LLVM toolchain was later applied to the LLVM toolchain to address missing pre-processor defines during initial kernel compile. The missing pre-processor defines were a side-effect of the LLVM toolchain not fully parsing the OpenBSD-specific compile target: `--target=riscv64-unknown-openbsd6.7`. This patch, shown in Snippet 3, was introduced to local copies of the LLVM toolchain source code to address this issue.

4. The toolchain used in the project was later upgraded to LLVM 10.

```
From ebd07b299e0c5e4d9cd0779861969b846fc8f2a6 Mon Sep 17 00:00:00 2001
From: Brian Bamsch <bbamsch@google.com>
Date: Mon, 14 Apr 2020 20:59:09 -0700
Subject: [PATCH] [RISCV] Add Target for OpenBSD

---
 clang/lib/Basic/Targets.cpp    | 2 ++
 clang/test/Preprocessor/init.c | 3 +++
 2 files changed, 5 insertions(+)

diff --git a/clang/lib/Basic/Targets.cpp b/clang/lib/Basic/Targets.cpp
index c063f8ca447..bacc2149fbe 100644
--- a/clang/lib/Basic/Targets.cpp
+++ b/clang/lib/Basic/Targets.cpp
@@ -389,2 +389,4 @@ TargetInfo *AllocateTarget(const llvm::Triple &Triple,
      return new LinuxTargetInfo<RISCV64TargetInfo>(Triple, Opts);
+    case llvm::Triple::OpenBSD:
+      return new OpenBSDTargetInfo<RISCV64TargetInfo>(Triple, Opts);
     default:
diff --git a/clang/test/Preprocessor/init.c b/clang/test/Preprocessor/init.c
index 6966698549a..647563d92a9 100644
--- a/clang/test/Preprocessor/init.c
+++ b/clang/test/Preprocessor/init.c
@@ -9558,2 +9558,4 @@
 // RUN:   | FileCheck -match-full-lines -check-prefixes=RISCV64,RISCV64-LINUX
    ↪ %s
+// RUN: %clang_cc1 -E -dM -ffreestanding -fgnuc-version=4.2.1 -triple=riscv64-
   ↪ unknown-openbsd < /dev/null \
+// RUN:   | FileCheck -match-full-lines -check-prefixes=RISCV64,RISCV64-
   ↪ OPENBSD %s
 // RISCV64: #define _LP64 1
@@ -9760 +9762,2 @@
 // RISCV64-LINUX: #define unix 1
+// RISCV64-OPENBSD: #define __OpenBSD__ 1
--
2.26.0
```

Snippet 3. Diff of changes made to LLVM source to support ‘`riscv64-unknown-openbsd6.7`’.

5.1.3 QEMU

QEMU provides the virtual RISC-V hardware target used to bootstrap the initial OpenBSD kernel port in this project. QEMU exposes a GDB stub which provides a convenient mechanism to single-step the kernel and debug issues during kernel initialization.

As explained in Section 4.2, supervisor-mode software such as the OpenBSD kernel is compiled against the standard supervisor binary interface to increase portability. The

OpenBSD kernel has to run within an SEE that exposes certain SBI functions. In this project, BBL is responsible for providing the SEE to the OpenBSD kernel. The BBL executable embeds the compiled OpenBSD kernel and sets up the appropriate SEE before handing over control to the kernel payload. This BBL executable is passed to QEMU via the `kernel` argument to boot the OS kernel.

5.1.4 GDB

The version of GDB shipped with OpenBSD-current does not support the RISC-V ISA. Therefore, GDB needs to be built from source. Unfortunately, building GDB on OpenBSD has been proved to be quite cumbersome. This issue is sidestepped by instead running GDB on a different machine and debugging QEMU remotely. This project's work repository [17]: '*02-Set up RISCV-GNU tool chain on Linux & OpenBSD*' provides a step-by-step guide for this.

Figure 16 demonstrates a possible environment setup, where the OpenBSD machine is dedicated to source code editing and kernel compilation while a remote Ubuntu machine is utilized for building BBL, running QEMU, and debugging the kernel with GDB. BBL building and QEMU can also run on the OpenBSD host machine, which is functionally equivalent to the configuration in Figure 16.

5.2 Machine-Dependent Headers

The machine-dependent kernel components interface directly with the RISC-V hardware and provide the necessary bindings to the machine-independent kernel components. This section highlights a few of the relatively large set of machine-dependent interfaces that have been implemented to support the RISC-V ISA.

5.2.1 Types (_types.h)

The Types header defines the machine-dependent data types. More specifically, it assigns the exact-width, minimum-width, fastest minimum-width, and greatest-width

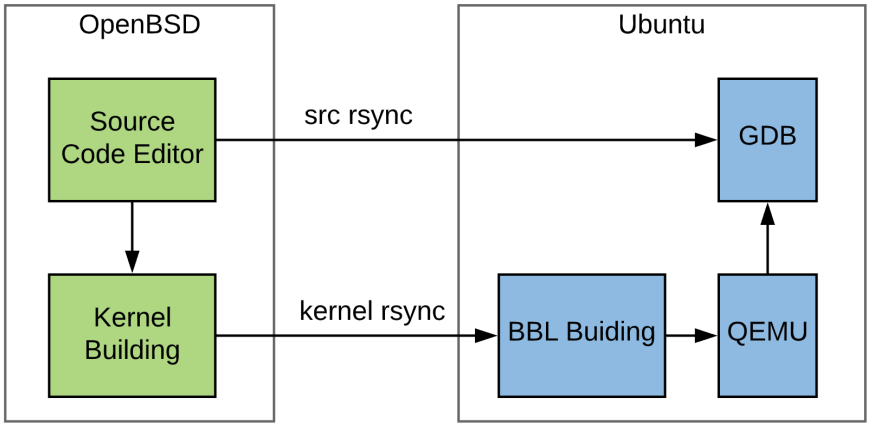

Fig. 16. One possible setup of the develop environment.

integers to their appropriate underlying data types. This file also assigns types for register-sized data, physical memory addresses, and virtual memory addresses. The types defined in this file are used widely across the OpenBSD kernel.

5.2.2 Atomics (atomic.h)

The Atomics header defines a set of functions which are used to provide atomic updates to memory. In particular, this header exposes `atomic_setbits_int` and `atomic_clearbits_int`, which read a location from memory, set or clear bits present in an integer-sized mask, and write the value back to the same location in memory. This kernel header also provides memory barriers including: a barrier to wait on all pending reads from memory, a barrier to wait on all pending writes to memory, as well as a barrier to wait on all pending memory operations.

5.2.3 Endianness (endian.h)

The Endianness header interface provides a small set of functions to swap the endianness of values in memory. The RISC-V baseline architecture uses little-endian byte order but does not rule out the possibility for big-endian byte order implementations nor implementations which support both in a configurable manner. While the baseline ISA

does not allow for switching between processor endianness, the big-endian order is still useful when supporting big-endian hardware.

Unfortunately, there is no finalized RISC-V specification for bit manipulation. Implementing an endian swap currently requires manually performing swaps on individual bytes. Snippet 4 shows the 64-bit endianness swap for RISC-V without bit manipulation instruction support. Bit manipulation instructions will be introduced with the *B-extension* to the RISC-V ISA.

```
static __inline __uint64_t
__swap64md(__uint64_t _x)
{
        __uint64_t ret;

        ret = (_x >> 56);
        ret |= ((_x >> 40) & 0xff00);
        ret |= ((_x >> 24) & 0xff0000);
        ret |= ((_x >>  8) & 0xff000000);
        ret |= ((_x <<  8) & ((__uint64_t)0xff << 32));
        ret |= ((_x << 24) & ((__uint64_t)0xff << 40));
        ret |= ((_x << 40) & ((__uint64_t)0xff << 48));
        ret |= (_x << 56);

        return (ret);
}
```

Snippet 4. 64-bit endianness swap for RISC-V without bit manipulation instructions.

5.2.4 Trap Frames (frame.h)

The Trap Frames header defines various trap frames used by the kernel when switching between execution contexts. These structures closely mimic the set of registers defined by the RISC-V ISA. When an exception or signal occurs, the trap frame is pushed onto the stack and all of the registers for the active process are saved. This allows for the execution state of a process to be restored once control is handed back by the kernel.

The 64-bit RISC-V ISA defines 32 distinct 64-bit wide integer registers. In the trap frame shown in Snippet 5, space is allocated for 31 of the 32 registers as the `x0` register

is hard coded to value 0 and does not need to be saved. In addition to registers, the trap frame also includes a small number of CSRs which are useful during exception handling.

```
/*
 * Exception/Trap Stack Frame
 */
#define clockframe trapframe
typedef struct trapframe {
        /* Standard Registers */
        register_t tf_ra;
        register_t tf_sp;
        register_t tf_gp;
        register_t tf_tp;
        register_t tf_t[7];
        register_t tf_s[12];
        register_t tf_a[8];
        /* Supervisor Trap CSRs */
        register_t tf_sepc;
        register_t tf_sstatus;
        register_t tf_stval;
        register_t tf_scause;
} trapframe_t;
```

Snippet 5. Trap frame structure.

5.2.5 RISC-V Registers (riscvreg.h)

The RISC-V Registers header enumerates constants and functions useful for dealing with RISC-V CSRs. It contains definitions for a number of RISC-V CSRs including the `sstatus` CSR, which exposes information about the processor state.

5.2.6 Process Control Block (pcb.h)

The PCB header defines the process control block structure used by the kernel for process management. It is closely related to context switch. It holds process state such as the state of CPU registers at the time when a process was switched out of a HART. As context switching is normally triggered by a trap, the PCB data structure *struct pcb* in Snippet 6 contains a trap frame to hold the process state saved during trap to the kernel.

5.3 Early Bootstrap Implementation

Section 4.2 describes how the first two bootstrap stages prepare the system for the OS kernel. This section continues describing the kernel bootstrap logic as implemented in

```
struct pcb {
        u_int           pcb_flags;
#define PCB_FPU         0x00000001      /* Process had FPU initialized */
#define PCB_SINGLESTEP  0x00000002      /* Single step process */
        struct          trapframe *pcb_tf;

        register_t      pcb_sp;         // stack pointer of switchframe

        caddr_t         pcb_onfault;    // On fault handler
        struct fpreg    pcb_fpstate;    // Floating Point state */
        struct cpu_info *pcb_fpcpu;

        void            *pcb_tcb;
};
```

Snippet 6. Process control block structure.

locore.S. Specifically, this section only covers the booting process from the first kernel instruction until the jump to `initriscv()`, the machine-dependent kernel initialization routine.

The OpenBSD kernel runs in supervisor-mode. The control flow of *locore.S* is summarized in Figure 17.

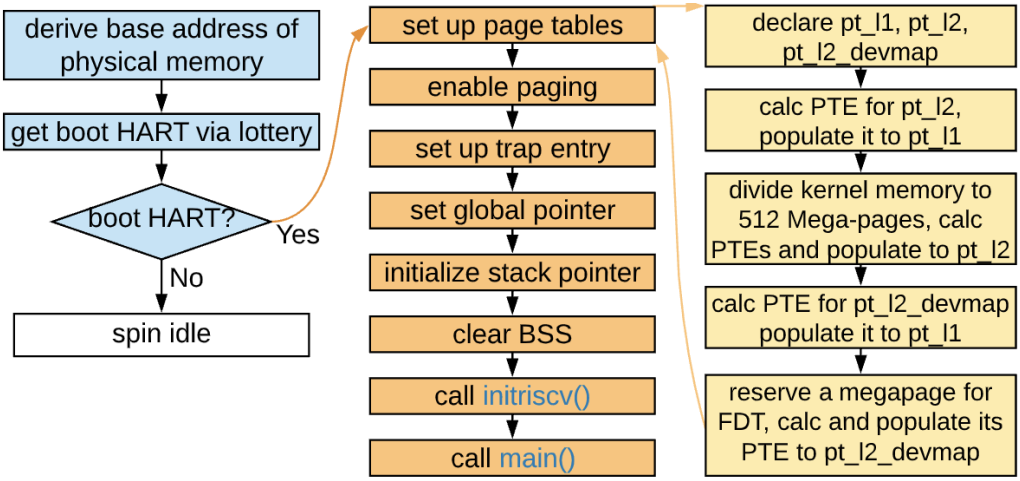

Fig. 17. OpenBSD early bootstrap control flow on RISC-V.

5.3.1 Starting State

When the OpenBSD kernel boots, it expects the system to be in the following state:

1) Register `a0` contains a unique HART identifier.
2) Register `a1` contains a pointer to the FDT.
3) System memory is identity-mapped.
4) The OS Kernel ELF image is loaded into system memory.

5.3.2 Resolve Kernel Physical Base Address

The kernel physical base address is resolved with the help of `virt_map`, which is a quad-word that stores `virt_map`'s virtual address. This virtual address is specified at link time. By subtracting `virt_map`'s physical address from its virtual address, an offset is calculated. This offset can be used to resolve a physical address from a virtual address which falls in the kernel virtual address space. The offset is applied to `KERNBASE`, the kernel virtual base address, to resolve the physical base address of kernel, which is stored in register `s9` for further reference.

5.3.3 Determine Boot HART

The OpenBSD kernel expects a single HART to boot the kernel. However, RISC-V HARTs enter the kernel in an arbitrary order from BBL. To handle this impedance mismatch, a very short Atomic Memory Operation (AMO) sequence is executed to select a boot HART. All HARTs attempt to atomically swap a '1' into a location in memory. The HART which successfully swaps its '1' into this memory address, stealing the '0' value held in this location, wins the HART lottery and is chosen to boot the OpenBSD kernel. All other HARTs wait until the boot HART signals that the kernel is ready for multi-processor entry.

5.3.4 Set up Page Table

During kernel compile, regions within the `.data` section are reserved for the initial L1 page table, L2 page table, and L2 device map page table. The kernel builds the initial page tables in these pre-allocated regions of memory. Page Table Entries (PTEs) are

calculated and inserted into these page tables to map the kernel image and FDT into the kernel virtual address space. The detailed procedure for this is shown in the right-most column of Figure 17. The resultant page table structure is shown in Figure 18.

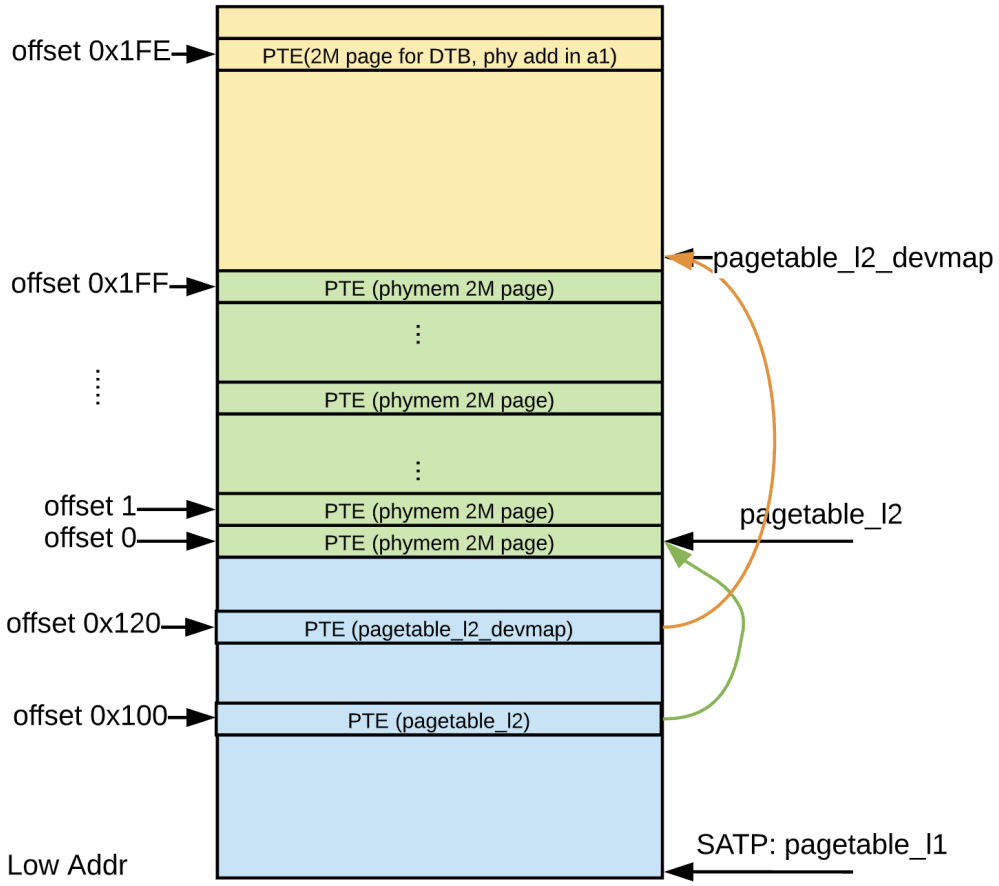

Fig. 18. The initial page table at bootstrap stage.

Once the initial kernel page tables are constructed, a Physical Page Number (PPN) is calculated from the physical address of `pagetable_l1`. The PPN is combined with the `Sv39` virtual address translation mode identifier and written to the supervisor address translation and protection (`satp`) register to enable paging.

5.3.5 Set up Supervisor Trap Vector

The supervisor trap handler is configured by writing the address of `cpu_trap_handler` to the `stvec` CSR. This function serves as the entry point into the kernel for all interrupts and exceptions. Interrupt and exception handling is described in Section 5.5.

5.3.6 Jump to C Routine

The kernel prepares for entry into the C runtime:

1) Set up Global Pointer `gp`.
2) Set up Stack Pointer `sp`.
3) Clear BSS.
4) Fill boot parameters to stack.

Once the C runtime is ready, the boot HART enters into `initriscv()` which performs more machine-dependent initialization. Afterwards, the entry into `main()` triggers machine-independent kernel initialization. Later, in the machine-independent kernel initialization, all other HARTs are configured via `init_secondary()`.

5.4 Memory Subsystem Implementation

OpenBSD relies on the UVM virtual memory system to manage memory resources. The UVM system consists of two layers: a machine-dependent layer and a machine-independent layer. The machine-independent layer is the larger of these two and is shared across all supported architectures. It exposes memory resources at a higher level abstraction over the memory scheme provided by the underlying hardware. The MI layer manages memory of the underlying hardware indirectly through the machine-dependent layer known as the physical map (PMAP). The PMAP layer understands the virtual memory scheme provided by a specific target architecture and acts on requests from the MI layer to map virtual memory addresses to physical memory addresses.

Supporting the RISC-V ISA within the OpenBSD memory subsystem involves adapting the PMAP layer to build and modify in-memory page table structures to control the virtual memory layout exposed by RISC-V MMU. This section provides a deep dive into the machine-dependent aspects of managing the RISC-V MMU. It includes both initial MMU configuration early in kernel boot as well as the adaptation of the PMAP layer to manage in-memory page table structures.

5.4.1 Bootstrapping the MMU

The default memory address translation scheme at system boot is `Bare` which provides a simple identity mapping between virtual and physical memory addresses. In this mode, the virtual memory address `0x80000000` maps to the physical memory address `0x80000000`. The other supported memory address translation modes are 4 KiB page-based virtual addressing schemes. `Sv39` and `Sv48` differ in the number of addressable bits in their virtual address space. `Sv39` mode has 39 addressable bits and `Sv48` has 48 addressable bits.

This default `Bare` memory address scheme is appropriate during kernel bootstrap as it gives an unrestricted view of physical memory. The kernel bootstrapping code operates in this mode to manually build the initial kernel page table structures using pre-allocated regions of memory. These initial kernel page tables are configured to define the virtual address space layout for the `Sv39` page-based memory address translation scheme. The initial kernel page tables map the OpenBSD kernel image, located at physical address `0x80000000`, to two locations in the kernel's virtual address space. The first location is its long-term home at virtual address `0xFFFFFFC000000000`. The second location is a temporary identity-mapping at virtual address `0x80000000`.

Snippet 7 provides a glimpse into how kernel bootstrapping code computes the appropriate offset into the L1 kernel page table in order to insert the identity-mapped kernel entry. This identity-mapped entry is a leaf node as identified by its protection bits.

Leaf page table entries inserted into L1 page table are GigaPage (1 GiB) mappings in `Sv39` mode.

```
/* step 0) Identity map kernel @ 0x80000000 */
/* Build PTE for 1 GiB identity-mapped GigaPage */
lla     s1, pagetable_l1        // Physical Address
mv      s2, s9                  // Physical Address
srli    s2, s2, PAGE_SHIFT      // Shift 12 -- a 4K page
slli    t5, s2, PTE_PPN0_S      // Shift 10 -- ppn[2:0] = pte[53:10]
li      t4, (PTE_KERN | PTE_X)  // Initialize Protection Bits
or      t6, t4, t5              // Build PTE for a 1 GiB GigaPage

/* Calculate VPN[2] for 1 GiB identity-mapped gigapage */
mv      a5, s9              // VA -- identity mapped
srli    a5, a5, L1_SHIFT    // Shift 30, va[63:30] remain
andi    a5, a5, 0x1ff       // Only use va[38:30] as VPN[2]

/* Store L1 PTE entry */
li      a6, PTE_SIZE    // Each Page Table Entry is 8 bytes
mulw    a5, a5, a6
add     t0, s1, a5      // Calculate address within L1 Page Table
mv      s8, t0          // Store address in L1 Page Table to unmap later
sd      t6, (t0)        // Write the PTE for Identity-Mapped Kernel
```

Snippet 7. Kernel bootstrap inserts an identity-mapped GigaPage to simulate `Bare` memory mode identity mapping during the transition into `Sv39` memory mode.

The temporary identity mapping serves to persist the existing memory layout in the small time span between when the MMU is enabled and when the program counter jumps to the newly-mapped virtual address space. Without this temporary mapping, the transition into `Sv39` mode would cause the program counter to point to an unmapped region of virtual memory. With the identity-mapped kernel in place, the HART can successfully fetch and decode the instruction immediately following the transition into `Sv39` mode which instructs the HART to relocate the program counter to the appropriate location in the new virtual address space.

Once the kernel bootstrapping code has successfully jumped into the virtual address space, the kernel identity mapping becomes obsolete and is unmapped from the L1 kernel page table as shown in Snippet 8.

```
/* Unmap the identity mapped kernel GigaPage */
sd      x0, (s8)     // s8 holds the address of PTE for identity mapped kernel
sfence.vma           // Flush the TLB. Goodbye identity mapped kernel!
```

Snippet 8. The identity-mapped kernel GigaPage is removed from the L1 Kernel page table immediately following the transition into the newly-mapped kernel virtual address space.

Following the switch into `Sv39` mode, the HART sees the kernel virtual address space defined by the initial kernel page tables constructed during bootstrap as shown in Figure 18. The initial kernel page tables are not managed by the UVM memory management system. The UVM memory management system is bootstrapped later, starting with the physical map.

5.4.2 Initializing the Physical Map

The machine-dependent PMAP layer of the UVM memory management system is initialized during kernel bootstrapping to provide a mechanism for managing spare regions of physical memory. Up until the physical map is initialized, kernel bootstrap code operates entirely in pre-allocated regions of memory.

The PMAP is initialized in the call to `pmap_bootstrap()`. Immediately prior to the call to `pmap_bootstrap()`, the kernel bootstrap code reads from the FDT to identify the full range of physical memory and provides it to the PMAP. This is important because, from this point onward, the PMAP layer becomes responsible for keeping track of available physical memory regions until UVM takes over. In addition to the full physical memory range, kernel bootstrap code also passes information on pre-allocated regions of memory, including the bootloader, the kernel image, and the FDT. Snippet 9 shows how the PMAP initializes itself to control the allocation of available memory resources.

The kernel can begin to allocate memory once available regions are set up within the PMAP layer. PMAP initialization continues by allocating memory to build the kernel

```
// Example parameters for Physical Map Bootstrap in QEMU
// kvo = 0x4080000000
// l1pt = 0xffffffc000811000
// kernelstart = 0xffffffc000200000
// kernelend = 0xffffffc0008b5c28
// fdt_start = 0x80a00000
// fdt_end = 0x80a00eb5
// ram_start = 0x80000000
// ram_end = 0xa0000000
vaddr_t
pmap_bootstrap(long kvo, vaddr_t l1pt, vaddr_t kernelstart, vaddr_t kernelend,
    paddr_t fdt_start, paddr_t fdt_end, paddr_t ram_start, paddr_t ram_end)
{
        (...) // Variable declarations omitted for conciseness

        pmap_setup_avail(ram_start, ram_end, kvo);

        // Remove the Bootloader physical address range
        printf("removing %lx-%lx\n", ram_start, kernelstart+kvo);
        pmap_remove_avail(ram_start, kernelstart+kvo);

        // Remove the Kernel Image physical address range
        printf("removing %lx-%lx\n", kernelstart+kvo, kernelend+kvo);
        pmap_remove_avail(kernelstart+kvo, kernelend+kvo);

        // Remove the FDT physical address range
        printf("removing %lx-%lx\n", fdt_start+kvo, fdt_end+kvo);
        pmap_remove_avail(fdt_start, fdt_end);

        (...) // Physical Map Bootstrap continues
}
```

Snippet 9. Early in the PMAP bootstrap process, the physical map stores the full extent of system physical memory and removes regions that were pre-allocated to the bootloader, kernel image, and FDT.

page table structures that will be managed by PMAP. The kernel page table structure initially consists of an L1 page table, an L2 page table, and 512 L3 page tables with each consuming 8 KiB of physical memory. The PMAP consumes 2 pages (8 KiB) per page table rather than a single page (4 KiB) as required by the `Sv39` translation mode. This is because the PMAP layer also stores another 4 KiB (512 x 8 Bytes) of pointers for easier traversal in software for page table management operations. After allocating physical memory for the kernel page table, all previously allocated regions of memory regions are populated into the page tables in a call to `pmap_map_stolen()`. The newly allocated kernel page tables are fully managed by the physical map layer.

At this point in PMAP initialization, the RISC-V MMU is still configured to use the initial kernel page tables that were constructed earlier during kernel bootstrap. PMAP bootstrap updates the `satp` register, which leads the MMU to switch from the initial kernel page tables to the PMAP-managed page tables. The active physical map is also stored to the CPU context as shown in Snippet 10. Upon returning from `pmap_bootstrap()`, the kernel memory address space is fully managed by the physical map layer of the OpenBSD memory subsystem.

```
vaddr_t
pmap_bootstrap(long kvo, vaddr_t l1pt, vaddr_t kernelstart, vaddr_t kernelend,
    paddr_t fdt_start, paddr_t fdt_end, paddr_t ram_start, paddr_t ram_end)
{
    (...) // Kernel PMAP is initialized

        // Switch to new kernel page table
        uint64_t satp = pmap_kernel()->pm_satp;
        __asm __volatile("csrw satp, %0" :: "r" (satp) : "memory");

        printf("all mapped\n");

    // Store current pmap to CPU
        curcpu()->ci_curpm = pmap_kernel();

        return vstart;
}
```

Snippet 10. The Physical Map bootstrap process is finalized by configuring the RISC-V MMU to perform translation based on the newly allocated, managed kernel page tables.

Later, once all machine-dependent bootstrapping has finished, the OpenBSD kernel brings up the machine-independent layer of UVM. The machine-independent UVM layer exposes system memory to the kernel at a higher-level abstraction to mask any machine-specific memory management details within the kernel. The machine-independent UVM layer delegates all modifications to in-memory page tables to the machine-dependent PMAP layer. The PMAP exposes a number of operations to the machine-independent UVM layer, allowing it to indirectly manage the RISC-V MMU.

The machine-independent portion of the OpenBSD memory subsystem is not delved into as it is identical across architectures. Instead, the page table management operations provided by the machine-dependent physical map layer is focused on in this project.

5.4.3 Creating a Page Table

Once the memory subsystem is initialized, the life cycle of any new page table starts with a call to `pmap_create()`. This call is typically executed as part of a call to `fork()` either within the kernel or within a user process (via a syscall). The PMAP's responsibility during a call to `pmap_create()` is to create a new page table identical in structure to the kernel page tables. The PMAP layer only deals with creating a bare minimum clone of the kernel page table because PMAP does not have an understanding of higher-level memory objects mapped into the page table by the machine-independent layer. For this reason, PMAP alone cannot feasibly clone an entire page table without potentially compromising address space isolation between the parent and child processes. Therefore, PMAP is only responsible for mapping in the kernel address space.

```
pmap_t
pmap_create(void)
{
        pmap_t pmap;
        pmap = pool_get(&pmap_pmap_pool, PR_WAITOK | PR_ZERO);
        mtx_init(&pmap->pm_mtx, IPL_VM);

        pmap_pinit(pmap); ❶
        if (pmap_vp_poolcache == 0) {
                pool_setlowat(&pmap_vp_pool, 20);
                pmap_vp_poolcache = 20;
        }
        return (pmap);
}
```

Snippet 11. Page tables are created by a call to `pmap_create()`

The call to `pmap_create()` allocates enough memory to hold an instance of the page table management structure (`pmap_t`) which holds a pointer to the page table and

other page table metadata. Initialization of the page table management structure is delegated to `pmap_pinit()` as demonstrated by ❶ in Snippet 11.

Within `pmap_pinit()`, the top-level (L1) page table entry is allocated and PMAP copies L1 page table entries for mapped regions in the kernel address space. For a 64-bit HART in `Sv39` mode, the mapped regions consist of the kernel image (`0xFFFFFFC000000000` – `0xFFFFFFC7FFFFFFFF` ❷) and direct mapped region (`0xFFFFFFD000000000` – `0xFFFFFFFEFFFFFFFF` ❸) as shown in Snippet 12.

```
void
pmap_pinit(pmap_t pm)
{
        struct pmapvp1 *vp1, *kvp1;
        vaddr_t l1va;
        uint64_t l1pa;

        /* Allocate a full L1 table. */
        while (pm->pm_vp.l1 == NULL) {
                pm->pm_vp.l1 = pool_get(&pmap_vp_pool,
                    PR_WAITOK | PR_ZERO);
        }

        vp1 = pm->pm_vp.l1; /* top level is l1 */
        l1va = (vaddr_t)vp1->l1;

        // Fill Kernel Entries ❷
        kvp1 = pmap_kernel()->pm_vp.l1;
        memcpy(&vp1->l1[L1_KERN_BASE], &kvp1->l1[L1_KERN_BASE],
                        L1_KERN_ENTRIES * sizeof(pt_entry_t));
        memcpy(&vp1->vp[L1_KERN_BASE], &kvp1->vp[L1_KERN_BASE],
                        L1_KERN_ENTRIES * sizeof(struct pmapvp2 *));

        // Fill DMAP PTEs ❸
        memcpy(&vp1->l1[L1_DMAP_BASE], &kvp1->l1[L1_DMAP_BASE],
                        L1_DMAP_ENTRIES * sizeof(pt_entry_t));
        memcpy(&vp1->vp[L1_DMAP_BASE], &kvp1->vp[L1_DMAP_BASE],
                        L1_DMAP_ENTRIES * sizeof(struct pmapvp2 *));

        pmap_extract(pmap_kernel(), l1va, (paddr_t *)&l1pa);
        pmap_set_ppn(pm, l1pa);
        pmap_set_mode(pm);
        pmap_allocate_asid(pm);
        pmap_reference(pm);
}
```

Snippet 12. The Physical Map allocates a new L1 page table and maps in the kernel image and direct mapped regions of the kernel address space.

PMAP uses the kernel page table to resolve the physical base address of the newly-allocated top-level page table. This physical address is combined with the `Sv39` translation mode identifier (`0x8`) and a newly allocated address space identifier (ASID) to form a value that is written to the `satp` register when switching to the newly-allocated page table.

PMAP returns the newly-created page table management structure to the machine-independent layer of UVM. The newly-created page table is not yet activated on a HART until it is attached to a process and explicitly activated as described in Section 5.4.4.

5.4.4 Activating a Page Table

Activating a page table on a HART is achieved by calling the physical map's `pmap_activate()` function. This function accepts a process structure which provides the PMAP layer with the page table management structure. This structure holds the page table that describes the virtual address space for the target process. Changing the virtual address space exposed by the RISC-V MMU is achieved by writing `satp` as shown by ❹ in Snippet 13.

```
void
pmap_activate(struct proc *p)
{
        pmap_t pm = p->p_vmspace->vm_map.pmap;
        int sie;

        sie = disable_interrupts();
        if (p == curproc && pm != curcpu()->ci_curpm)
                pmap_set_satp(p); ❹
        restore_interrupts(sie);
}
```

Snippet 13. Calling `pmap_activate()` switches to a new virtual address space.

The `satp` register is written in the call to `pmap_set_satp()`. The function uses inline assembly to devise an instruction for writing `satp`, which is immediately followed

by an address translation memory fence. Such a memory fence forces subsequent address translations to reflect the new virtual address space. The new page table is also saved to the current RISC-V HART state in software.

```
void
pmap_set_satp(struct proc *p)
{
        struct cpu_info *ci = curcpu();
        pmap_t pm = p->p_vmspace->vm_map.pmap;

        __asm __volatile("csrw satp, %0" :: "r"(val));
        __asm __volatile("sfence.vma");
        ci->ci_curpm = pm;
}
```

Snippet 14. The `pmap_set_satp()` function writes `satp` to switch the virtual address space and stores a pointer to the new page table management structure.

Note that updates to hardware and software state are not atomic during a virtual address space transition. It is important that interrupts are disabled during this transition to prevent the hardware and software from falling out of sync. Once execution returns to the caller of `pmap_activate()`, the new page table is activated on the HART and the kernel can later hand off control to the new process which will run in its own virtual address space.

5.4.5 Updating a Page Table

The kernel isolates processes running on the system by providing them with separate virtual address spaces and managing the permissions enforced on page table entries. When a new page table is created by the PMAP layer, it only has kernel page table entries mapped. The machine-independent UVM layer is responsible for mapping in user-mode relevant regions of the virtual address space.

The machine-independent UVM layer needs to update its internal structures to map in user-mode regions of the virtual address space. These regions are lazily-mapped into the page table. Loading an executable into the virtual address space does not necessarily

entail mapping the file into physical memory immediately. When a file is mapped into memory, it is only reflected in the MI UVM layer. The page tables which are manipulated by the PMAP layer are not yet updated. Later when the HART tries to fetch the first instruction of this executable, a page fault is triggered, which leads UVM to load the page into physical memory and insert it into the page tables via the PMAP layer.

The machine-independent UVM layer inserts mappings into physical memory by making calls to the `pmap_enter()` function provided by the PMAP layer. This function is responsible for initializing a new page table entry and inserting it into the page table. To prevent concurrent updates from corrupting the page table, a lock on the page table management structure must be held while the page table is being updated.

During page table entry initialization, the UVM layer provides the PMAP layer with a set of permissions desired for the new virtual to physical address mapping. This information is stored in the software descriptor for this page table entry by the `pmap_fill_pte()` function, as shown in Snippet 15. The software descriptor for a page table entry describes the permissions requested by the kernel memory subsystem. The permissions written into the software descriptor may differ slightly from the permissions enforceable by the underlying hardware. The translation between software permissions and hardware permissions is handled during PTE insertion.

The PMAP layer will attempt to insert the page table entry into the in-memory page table structure. A page table entry will be inserted if it is a wired mapping or has one of the read, write, or execute permissions set on the page table entry descriptor. The `pmap_pte_insert()` function as shown in Snippet 16 handles translating the software page table entry descriptor into the in-memory page table format supported by the underlying hardware.

The translation from software page table entry descriptor to the hardware-specific page table entry format is delegated to the `pmap_pte_update()` function identified

```
void
pmap_fill_pte(pmap_t pm, vaddr_t va, paddr_t pa, struct pte_desc *pted,
    vm_prot_t prot, int flags, int cache)
{
        pted->pted_va = va;
        pted->pted_pmap = pm;

        switch (cache) {
        case PMAP_CACHE_WB:
        case PMAP_CACHE_WT:
        case PMAP_CACHE_CI:
        case PMAP_CACHE_DEV:
                break;
        default:
                panic("pmap_fill_pte:invalid cache mode");
        }

        pted->pted_va |= cache;
        pted->pted_va |= prot & (PROT_READ|PROT_WRITE|PROT_EXEC);

        if (flags & PMAP_WIRED) {
                pted->pted_va |= PTED_VA_WIRED_M;
                pm->pm_stats.wired_count++;
        }

        pted->pted_pte = pa & PTE_RPGN;
        pted->pted_pte |= flags & (PROT_READ|PROT_WRITE|PROT_EXEC);
}
```

Snippet 15. The `pmap_fill_pte()` function initializes a page table entry descriptor which describes the expected behavior and permissions requested by the memory subsystem.

```
void
pmap_pte_insert(struct pte_desc *pted)
{
        /* put entry into table */
        pmap_t pm = pted->pted_pmap;
        pt_entry_t *pl3;

        if (pmap_vp_lookup(pm, pted->pted_va, &pl3) == NULL) {
                panic("%s: have a pted, but missing a vp"
                    " for %lx va pmap %p", __func__, pted->pted_va, pm);
        }

        pmap_pte_update(pted, pl3); ❺
}
```

Snippet 16. The `pmap_pte_insert()` function takes a page table entry descriptor and transforms it into the page table entry format supported by the hardware.

by ❺ in Snippet 16. The implementation of this function is shown in Snippet 17. The software permissions requested by the UVM layer are mapped to hardware-supported permissions by `ap_bits_kern` ❻ and `ap_bits_user` ❼, which assign permissions for kernel pages and user-mode pages, respectively.

```
void
pmap_pte_update(struct pte_desc *pted, uint64_t *pl3)
{
        pt_entry_t pte, access_bits;
        pmap_t pm = pted->pted_pmap;

        if (pm->pm_privileged)
                access_bits = ap_bits_kern[pted->pted_pte & PROT_MASK]; ❻
        else
                access_bits = ap_bits_user[pted->pted_pte & PROT_MASK]; ❼

        pte = VP_Lx(pted->pted_pte) | access_bits;
        *pl3 = pte;
}
```

Snippet 17. The `pmap_pte_update()` function translates the permissions requested by the UVM layer into permissions supported by the hardware.

As mentioned earlier, the software descriptor may describe a slightly different set of permissions than what can be enforced by the hardware. The RISC-V architecture, for example, does not support page table entries which are writable but not readable [13]. The permissions set in hardware are either equivalent to, if supported, or less restrictive than those permissions requested by the software. The translation between software permissions and hardware permissions are shown in Snippet 18.

The hardware-supported permissions derived from either `ap_bits_kern` or `ap_bits_user` are written into the in-memory page table. Ignoring any potential memory ordering issues which could arise from updating active page tables, the page table structure at this point will allow for successfully translating the virtual address `va` to the kernel-assigned physical address `pa`. The PMAP layer returns control to the machine-independent UVM layer following some cleanup work and unlocking the page

```
// For RISC-V Machines, write without read permission is not a valid
// combination of permission bits. These cases are mapped to R+W instead.
// PROT_NONE grants read permissions because r = 0 | w = 0 | x = 0 is
// reserved for non-leaf page table entries.
const pt_entry_t ap_bits_kern[8] = {
  [PROT_NONE]                       = PTE_A|PTE_R,
  [PROT_READ]                       = PTE_A|PTE_R,
  [PROT_WRITE]                      = PTE_A|PTE_R|PTE_W,
  [PROT_WRITE|PROT_READ]            = PTE_A|PTE_R|PTE_W,
  [PROT_EXEC]                       = PTE_A|PTE_X,
  [PROT_EXEC|PROT_READ]             = PTE_A|PTE_X|PTE_R,
  [PROT_EXEC|PROT_WRITE]            = PTE_A|PTE_X|PTE_R|PTE_W,
  [PROT_EXEC|PROT_WRITE|PROT_READ]  = PTE_A|PTE_X|PTE_R|PTE_W,
};

const pt_entry_t ap_bits_user[8] = {
  [PROT_NONE]                       = PTE_U|PTE_A|PTE_R,
  [PROT_READ]                       = PTE_U|PTE_A|PTE_R,
  [PROT_WRITE]                      = PTE_U|PTE_A|PTE_R|PTE_W,
  [PROT_WRITE|PROT_READ]            = PTE_U|PTE_A|PTE_R|PTE_W,
  [PROT_EXEC]                       = PTE_U|PTE_A|PTE_X,
  [PROT_EXEC|PROT_READ]             = PTE_U|PTE_A|PTE_X|PTE_R,
  [PROT_EXEC|PROT_WRITE]            = PTE_U|PTE_A|PTE_X|PTE_R|PTE_W,
  [PROT_EXEC|PROT_WRITE|PROT_READ]  = PTE_U|PTE_A|PTE_X|PTE_R|PTE_W,
};
```

Snippet 18. Permissions required by software are translated to permissions supported by the hardware by the `ap_bits_kern` and `ap_bits_user` structures.

table management structure. This process is repeated as many times as necessary to map all user-mode pages into the page table structure.

5.5 Trap Subsystem Implementation

This section expands upon the overview of trap CSRs and trap entry/exit presented in Section 4.4, providing further details into how traps are routed and handled within the kernel.

5.5.1 Routing a Trap

As shown in Snippet 19, `stvec` is set to `cpu_trap_handler`, where all interrupts and exceptions are first routed to. `cpu_trap_handler` further delivers the trap to its specific handler.

The implementation of the `cpu_trap_handler` is shown in Snippet 20. The trap handler uses the `sscratch` register to distinguish whether the trap originates from the

```
/* Set up supervisor trap vector */
la      t0, cpu_trap_handler
csrw    stvec, t0
```

Snippet 19. Assembly code to set up supervisor trap vector `stvec`.

kernel or a user-mode program. Based on the originating privilege mode, the trap is forwarded to `cpu_trap_handler_supervisor` or `cpu_trap_handler_user`, respectively. These two handlers are very similar to each other in their prologue and epilogue, which encapsulates the privilege-mode-specific handlers, as shown in locations ❽ and ❾ of Snippet 21, respectively.

```
ENTRY(cpu_trap_handler)
        csrrw   sp, sscratch, sp
        beqz    sp, 1f
        /* User-mode detected */
        j       cpu_trap_handler_user
1:
        /* Supervisor-mode detected */
        csrrw   sp, sscratch, sp
        j       cpu_trap_handler_supervisor
END(cpu_trap_handler)
```

Snippet 20. Assembly code implementation of `cpu_trap_handler`.

```
ENTRY(cpu_trap_handler_supervisor)
        save_registers 1
        mv      a0, sp
        call    _C_LABEL(do_trap_supervisor)    ❽
        load_registers 1
        sret
END(cpu_trap_handler_supervisor)

ENTRY(cpu_trap_handler_user)
        save_registers 0
        mv      a0, sp
        call    _C_LABEL(do_trap_user)  ❾
        do_ast
        restore_registers 0
        csrrw   sp, sscratch, sp
        sret
END(cpu_trap_handler_user)
```

Snippet 21. Assembly code implementation of `cpu_trap_handler_supervisor` and `cpu_trap_handler_user`.

In *Direct*-mode, both `do_trap_supervisor()` and `do_trap_user()` must read the `interrupt` bit of the `scause` register to determine whether the trap originates from an interrupt or exception. The `interrupt` bit and the `code` bits of the `scause` register uniquely identify the cause of a trap, as depicted in Table 2.

Taking `do_trap_supervisor()` as an example, the check-and-act logic is implemented as shown in Snippet 22. All interrupts are routed to the generic interrupt handler entry `riscv_cpu_intr()`, while all exceptions are routed to an appropriate exception handler according to the specified exception cause, as depicted in Section 5.5.3.

```
void
do_trap_supervisor(struct trapframe *frame)
{
        uint64_t exception;
        ...

        if (frame->tf_scause & EXCP_INTR) {
                /* Interrupt */
                riscv_cpu_intr(frame);
                return;
        }
        ...

        exception = (frame->tf_scause & EXCP_MASK);
        switch(exception) {
        case EXCP_FAULT_LOAD:
        case EXCP_FAULT_STORE:
        case EXCP_FAULT_FETCH:
        case EXCP_STORE_PAGE_FAULT:
        case EXCP_LOAD_PAGE_FAULT:
            data_abort(frame, 0);
                break;
        case ...

}
```

Snippet 22. C code implementation for `do_trap_supervisor()`.

5.5.2 Interrupt Handling

RISC-V defines three types of interrupts: software interrupt, timer interrupt, and external interrupt. Software interrupts are used to perform Inter-Processor Interrupts (IPIs) for communication between processors in a multi-processor system[5]. Timer interrupts are

used by the HART to trigger regular interrupts for time-slicing which drives the kernel scheduler. External interrupts are used to communicate with peripherals, such as UART devices and PCI devices.

These interrupts are routed by a two-level nested interrupt controller architecture, as demonstrated in Figure 19. This diagram illustrates a two-HART CPU system, where each HART has a dedicated HART-Level Interrupt Controller (HLIC). The software interrupts and timer interrupts are provided by a Core-Local Interrupter (CLINT)[6]and the external interrupts are routed to HLIC via a Platform-Level Interrupt Controller (PLIC).

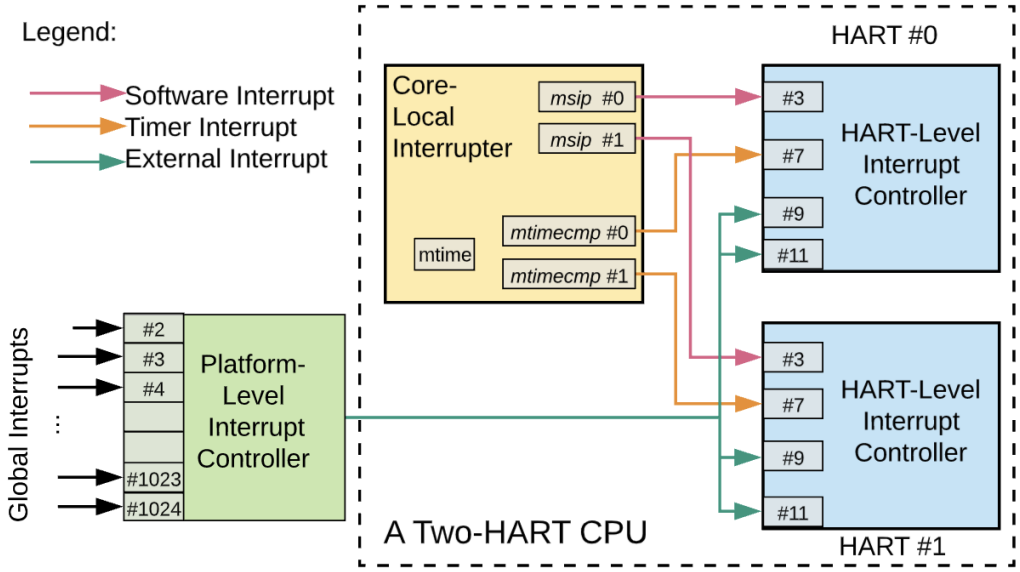

Fig. 19. A two-level nested interrupt controller architecture.

5.5.2.1 Core-Local Interrupter: CLINT serves as a simple CPU interrupter for software interrupts and timer interrupts. The CLINT integrates the `mtime` and `mtimecmp` CSRs to determine when to generate a timer interrupt. CLINT also integrates the `msip` CSR which is used for software interrupts. These three CSRs are listed below:

5. This port of OpenBSD for RISC-V does not yet support multi-processor systems, therefore this feature is currently unimplemented and is reserved as a future work.

6. The terminology Core-Local Interrupter is inherited from SiFive [16]. Despite what the name might imply, CLINT is shared among all HARTs as depicted in Figure 19.

- `mtime`: machine-mode timer register, which is expected to be implemented as an real-time clock device [2]. This clock runs at a constant frequency. There is only one `mtime` across the platform, even for a multi-HART system.
- `mtimecmp`: machine-mode timer compare register, which signals when to trigger a timer interrupt. A timer interrupt is generated when `mtime` becomes greater than or equal to `mtimecmp`. Each HART has its own dedicated `mtimecmp`.
- `msip`: machine-mode software interrupt pending register, which is used to send a software interrupt to a HART. Each HART has its own dedicated `msip` so that HART-A can write to HART-B's `msip` for inter-HART communication.

Specific to the QEMU '*virt*' board, CLINT's FDT definition is shown in Snippet 24. The `interrupts-extend` field should be interpreted as: CLINT extends interrupt `0x00000003` (machine-mode software interrupt) of interrupt controller whose `phandle` equals `0x00000002` (the HLIC). Similarly, CLINT also extends interrupt `0x00000007` (machine-mode timer interrupt) of the same HLIC.

```
soc {
    ...
    clint@2000000 {
        interrupts-extended = <0x00000002 0x00000003 0x00000002 0x00000007>;
        reg = <0x00000000 0x02000000 0x00000000 0x00010000>;
        compatible = "riscv,clint0";
};
```

Snippet 23. CLINT definition in FDT

The software interrupt and timer interrupt mechanisms provided by CLINT are exposed as machine-mode operations. These machine-mode mechanisms are filtered out from the FDT by BBL thus they are invisible to the OS kernel. Supervisor-mode does expose a `time` register which mirrors the behavior of the machine-mode `mtime` register. Reads to the supervisor-mode `time` register are intercepted and emulated by BBL, by performing the I/O to retrieve time value from CLINT's `mtime` [2]. The `mtimecmp` and `msip` registers do not have supervisor-mode counterparts and are instead exposed via SBI

functions. The `sbi_set_timer()` SBI function allows the OS kernel to set a new value to `mtimecmp` to configure the timer interrupt. Similarly, the `sbi_send_ipi()` and `sbi_clear_ipi()` SBI functions allow the OS kernel to manipulate the `msip` register to send and receive software interrupts from other HARTs.

5.5.2.2 Platform-Level Interrupt Controller: The PLIC provides a flexible system-level mechanism for dispatching global external interrupts to one or more "contexts" within the system [16]. The term "context" used here is different from that in Process Subsystem Implementation (Section 5.6). Within this section, a "context" refers to a HART under a specific privilege mode. Each global interrupt has a dedicated interrupt identifier and arrives at a HART through a single interrupt connection. The PLIC provides memory-mapped registers to program the priority for each external interrupt source, to program the priority threshold for each context, to control the connectivity of each interrupt-to-context pair, and to claim/complete an interrupt.

- PLIC **ID**: Each distinct interrupt source is assigned a unique identifier. PLIC supports up to 1024 interrupt sources, where interrupt ID 1 is tied to `global_interrupt[0]`, which means "no interrupt". This makes the first usable PLIC ID 2. PLIC ID is independent of the interrupt IDs as presented in `scause` for the core-local interrupt handler.
- PLIC **Priorities**: Each distinct interrupt source has a configurable priority level. QEMU emulates the SiFive implementation of the PLIC which supports priority values from 0 through 7, inclusively. The value '0' is the lowest priority and disables the interrupt source. Ties on priority are resolved by PLIC ID (lower PLIC ID wins).
- PLIC **Threshold**: Each context has a configurable priority threshold. The priority threshold is used to mask all interrupts with priority less than or equal to the specified threshold. Only interrupts with priority higher than the threshold are propagated to the HART.

- PLIC **Connectivity**: Connectivity between each pair of interrupt source and context is configurable. This register provides fine-grained control over where interrupt sources are routed.

PLIC extends both machine-mode and supervisor-mode external interrupts of HLIC. The OS kernel can receive supervisor-mode external interrupts from PLIC directly. To handle an external interrupt, a HART should read the `claim` register to acquire the PLIC ID. A successful claim atomically clears the pending bit in the PLIC interrupt pending register, signaling that the interrupt is being serviced [16]. Before the interrupt handler is finished and the `claim` register is written back, the PLIC cannot forward a new interrupt to the same HART. This means PLIC does not support preemption of global interrupts to an individual HART. The details to handle an external interrupt is described in paragraph 5.5.2.5. How PLIC is configured and attached is depicted in Section 5.7.7.

5.5.2.3 HART-Level Interrupt Controller: The essential software data structure in OS kernel for this HLIC is the handlers vector, `struct intrhand* intc_handler[INTC_NIRQS]`, which is referenced to route an interrupt to the correct handling entry point. For the OpenBSD kernel running in supervisor-mode, at least three handlers should be populated into this vector via a call to `riscv_intc_intr_establish()`.

- `IRQ_SOFTWARE_SUPERVISOR`.
- `IRQ_TIMER_SUPERVISOR`.
- `IRQ_EXTERNAL_SUPERVISOR`.

When an interrupt happens, `riscv_intc_irq_handler()` checks the `irqno` and uses this to index into the handlers vector and invoke the appropriate handler.

How this interrupt controller is configured and attached is depicted in Section 5.7.5.

5.5.2.4 Timer Interrupt Handling: The configuration and initialization of the timer device is depicted in Section 5.7.4. During

`riscv_timer_cpu_initclocks()`, the timer establishes its interrupt handler `riscv_timer_intr` to the corresponding entry of the HLIC's handlers vector: `intc_handler[IRQ_TIMER_SUPERVISOR]`. Later, when a timer interrupt is routed to the HART as demonstrated in Section 5.5.1, `riscv_timer_intr()` is called to:

- Calculate the time for the next timer interrupt.
- Write the next interrupt time to `mtimecmp` via SBI function `sbi_set_timer()`.
- Clear the supervisor timer interrupt pending bit (`sip.STIP`).
- Return '1' to indicate interrupt has been successfully handled.

Normally the timer interrupt frequency is set to 100Hz. This periodic interrupt is used in process scheduling, as described in Section 5.6.

5.5.2.5 External Interrupts Handling: The core data structure maintained for the PLIC is the external interrupt handlers vector `struct plic_irqsrc *sc_isrcs` as embedded in `plic_softc`. Each `plic_irqsrc` corresponds to one external interrupt source, but each interrupt could have multiple handlers (`struct plic_intrhand`). This could be because different HARTs have different handling procedures for the same interrupt.

Each device that reports interrupts to the PLIC should establish an interrupt handler with `sc_isrcs`, by a call to `plic_intr_establish_fdt()`. Later, when an external interrupt occurs, the pre-established handler will be called from `plic_irq_handler()`. Within this handler, a HART first reads the `claim` register of PLIC to obtain the pending interrupt ID. The HART then uses the interrupt ID to index into the PLIC handlers vector to invoke the corresponding handler. Once the handler is returned, the HART writes the interrupt ID back to the `claim` register, indicating the interrupt handling has been completed.

The OpenBSD kernel maintains a global System Priority Level *SPL* to block interrupts with priority less than or equal to the named level. *SPL* is enforced to PLIC's threshold register in `plic_setipl()`. Upon entry into an external interrupt's handler (`plic_irq_dispatch()`), the OS kernel should first raise *SPL* to prevent this handler from being preempted by interrupts with lower or equal priority. The *SPL* value will be restored upon the exit from this handler.

5.5.3 Exception Handling

Exceptions arise as a direct consequence of an instruction executing on a RISC-V HART. For example, an exception will occur for the illegal instruction defined as 16 consecutive zero bits (`0x0000`). The majority of the defined exception types, as shown in Table 3, arise from faults related to memory access.

Table 3
Summary of RISC-V Supervisor-Mode Exception Types

Code	Description
0	Instruction address misaligned
1	Instruction access fault
2	Illegal instruction
3	Breakpoint
4	Load address misaligned
5	Load access fault
6	Store / AMO address misaligned
7	Store / AMO access fault
8	Environment call from U-mode
9	Environment call from S-mode
10-11	*Reserved*
12	Instruction page fault
13	Load page fault
14	*Reserved for future standard use*
15	Store / AMO page fault
$\geq$16	*Reserved*

The exception types defined in Table 3 can be divided into two categories: exceptions arising from instruction faults and exceptions arising from memory faults.

5.5.3.1 Instruction Faults: Instruction faults consist of exceptions which arise as a direct result of executing an instruction excluding all memory access and alignment related faults. Of those exception types defined in Table 3, instruction faults include: *illegal instruction*, *breakpoint*, *environment call from U-mode*, and *environment call from S-mode*. These exception types arise due to issues unrelated to memory access and may instruct the kernel to take some explicit actions to remediate the fault.

The *illegal instruction* exception can arise from execution of the defined illegal instruction (`0x0000`) or from execution of instructions that are not supported by the hardware implementation. For example, an illegal instruction exception will occur when executing a floating point instruction on a RISC-V platform which does not implement the floating point ISA extension. A kernel trap handler could be instrumented to emulate instructions that are not provided by the underlying hardware, if desired, or might simply signal to terminate the offending process if there is no remediation for the faulting instruction. Within the OpenBSD kernel, the response to this type of exceptions depends on the context from which the exception arises. Illegal instructions encountered from the kernel context will cause a kernel panic, whereas illegal instructions encountered from a user-mode context will trigger an illegal instruction signal (`SIGILL`) to be issued to the offending process.

The *breakpoint* exception type occurs as a direct consequence of executing an environment breakpoint instruction (`EBREAK`) during program execution. This instruction is intended to be used by debuggers to insert hardware breakpoints to regain control of the RISC-V HART when the program reaches a certain point in the sequence of instructions. Similar to the *illegal instruction* exception type, the OpenBSD kernel's reaction to this type of exceptions depends on the context from which it is received. The OpenBSD kernel features a built-in kernel debugger (`DDB`) which handles breakpoints encountered

in the kernel context. Breakpoints encountered from user-mode contexts trigger a trap signal (`SIGTRAP`) to be issued to the offending process.

The remaining instruction-fault exceptions are *environment call* arising as a direct result of an environment call (`ECALL`) instruction from different source contexts. Within the context of the OpenBSD kernel trap handler, the *environment call from U-mode* is the only expected type of environment call exception. These exceptions are used by user-mode processes to issue *system calls* to the OpenBSD kernel. The kernel trap handler delegates system calls to the machine-dependent syscall handler (`svc_handler()`) whose implementation is partially shown in Snippet 24.

The machine-dependent syscall handler looks into the registers saved in the `trapframe` of the active process. As shown by ❿ in Snippet 24, the user-mode program stores the unique system call code in register `t0`, which the system call handler uses to identify the requested operation. The user-mode process may also pass one or more arguments as required by the system call in the argument registers. The machine-dependent system call handler takes responsibility to copy system call arguments from the user-mode process address space into the kernel address space ① before delegating it to the machine-independent system call handler ②.

5.5.3.2 Memory Faults: Memory faults are defined as those arising as a result of a memory-related operation. This includes all exception codes listed in Table 3 that are not already covered in Instruction Faults. The memory fault exceptions can be broken down further into three subcategories: Memory Alignment Faults, Memory Page Faults, and Memory Access Faults.

Memory alignment faults occur when an address is misaligned. For example, the program counter must be 2-byte aligned for instruction fetch on RISC-V implementations that support the compressed instruction format or 4-byte aligned on RISC-V

```
void
svc_handler(trapframe_t *frame)
{
    (...) // Variable declarations

        ap = &frame->tf_a[0];       // Pointer to first arg
        code = frame->tf_t[0]; ⑩    // Syscall code
        callp = p->p_p->ps_emul->e_sysent;

    (...) // Syscall bounds check & lookup

        nargs = callp->sy_argsize / sizeof(register_t);
        if (nargs <= nap) {
                args = ap;
        } else {
                KASSERT(nargs <= MAXARGS);
                memcpy(copyargs, ap, nap * sizeof(register_t));
                if ((error = copyin((void *)frame->tf_sp, copyargs + nap,
                     (nargs - nap) * sizeof(register_t)))) ①
                        goto bad;
                args = copyargs;
        }

        rval[0] = 0;
        rval[1] = frame->tf_a[1];

        error = mi_syscall(p, code, callp, args, rval); ②

        switch (error) {
        case 0:
                frame->tf_a[0] = rval[0];
                frame->tf_a[1] = rval[1];
                frame->tf_t[0] = 0;             /* syscall succeeded */
                break;

        case ERESTART:
                frame->tf_sepc -= 4;            /* prev instruction */
                break;

        case EJUSTRETURN:
                break;

        default:
        bad:
                frame->tf_a[0] = error;
                frame->tf_t[0] = 1;             /* syscall error */
                break;
        }

        mi_syscall_return(p, code, error, rval);
}
```

Snippet 24. The `svc_handler()` function handles the machine-dependent aspect of a system call before delegating it to the machine-independent system call handler.

implementations which do not support the compressed instruction format. Loads, stores, and atomic memory operations must be aligned to an appropriate byte boundary.

Memory page faults occur when a memory address translation fails due to a missing or otherwise invalid virtual to physical address mapping. Specifically, when a virtual to physical address mapping has not been cached in the Translation Lookaside Buffer (TLB), the RISC-V hardware page table walker will traverse the in-memory page table structure following the procedure outlined in Section 4.3. If the virtual to physical address mapping is missing or otherwise invalid, a page fault is thrown based on the memory access type.

Memory access faults occur when a memory access fails a Physical Memory Permissions check following virtual to physical address translation. Each virtual to physical address mapping stored in the page table is assigned PMP bits, which determine whether the memory is *readable*, *writable*, and *executable*. Attempting to access a memory page with permissions not granted by the page table entry will cause a memory access fault.

The kernel trap handler delegates recoverable memory faults to the UVM memory subsystem by a call to `data_abort()` function. The `data_abort()`, as partially represented in Snippet 25, uses the `scause` register to resolve the access type ③ and relays this information to UVM. The machine-dependent PMAP layer is given a chance to resolve any delayed mappings which may not yet be reflected in the in-memory page tables. If the PMAP layer is unable to resolve the mapping, the fault is instead issued to the machine-independent UVM layer ④. UVM reads the virtual address space and may insert the virtual to physical address mapping into the page table.

5.6 Process Subsystem Implementation

The process subsystem manages all processes within the operating system. An instance of the process structure holds on to the state of the relevant hardware. When a process is context switched onto a HART, the state of general purpose registers is restored

```
static void
data_abort(struct trapframe *frame, int usermode)
{
        (...) // Variable declarations

        if ((frame->tf_scause == EXCP_FAULT_STORE) ||
            (frame->tf_scause == EXCP_STORE_PAGE_FAULT)) {
                access_type = PROT_WRITE; ③
        } else if ((frame->tf_scasue == EXCP_FAULT_FETCH) ||
                   (frame->tf_scause == EXCP_INST_PAGE_FAULT)) {
                access_type = PROT_EXEC;
        } else {
                access_type = PROT_READ;
        }

        (...) // Resolve UVM VM Map for fault context

        if (pmap_fault_fixup(map->pmap, va, ftype, usermode))
                goto done;

        KERNEL_LOCK();
        error = uvm_fault(map, va, ftype, access_type); ④
        KERNEL_UNLOCK();

        (...) // Cleanup & Error Handling
}
```

Snippet 25. The `data_abort()` function resolves the fault type based on the process state held in `trapframe` and delegates these faults to the OpenBSD memory subsystem.

from the previous state of the process. This saved state is a snapshot of the process's execution at the moment when it was switched out of a HART. The context switch aspect of process management is described in more detail in Section 5.6.2.

The OpenBSD process subsystem also includes a CPU scheduler which is responsible for deciding which of the active processes can be assigned to an idle HART. The CPU scheduler leverages the RISC-V's hardware timer to regain control of the HART from processes that do not yield before the periodical timer interrupt arrives. Timer interrupt handling is described in Section 5.5.

Kernel bootstrapping code manually constructs a process context for itself to become process 0 as part of system initialization. This manual process construction is only

utilized for process 0, as all other processes are spawned by a call to one of the variants of `fork()`.

5.6.1 Process Fork

With exception of the root process (process 0), all processes are spawned from a parent process via a call to `fork()`. The root process is the only process that does not have a parent process. Instead, it is the root of the process tree. All other processes are attached to the root process through some chain of parent processes. An example of the process hierarchy is shown in Figure 20.

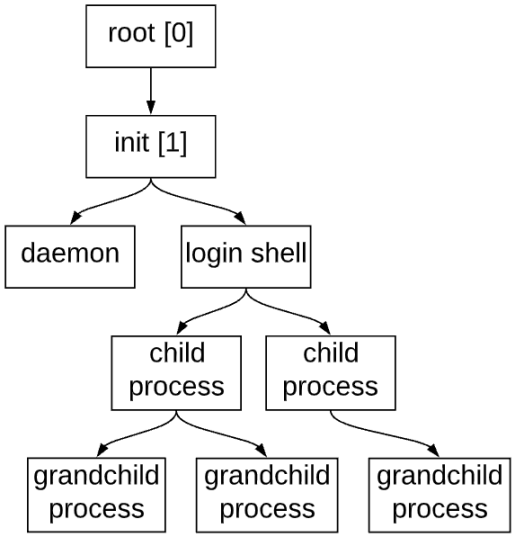

Fig. 20. Example of process hierarchy in a UNIX-like system.

When a process is forked from its parent, the child is spawned as a near-identical duplicate of the parent process. The kernel is responsible for carrying over all process state into the new process and creating a clone of the virtual address space of the parent process. The virtual address space clone is delegated to UVM. UVM's machine-dependent PMAP layer is responsible for cloning the kernel page table entries while UVM's machine-independent layer is responsible for persisting all other memory objects into the new virtual address space.

The forked child process is added to the CPU scheduler and the `fork()` call returns the child process identifier to the parent process. For kernel process forks, the parent process in the kernel controls the entry point of the new kernel process. For normal user-mode process forks, the new process returns to user-mode at the instruction immediately following the `fork()` system call in the parent process. The new child process sits in the process list until it is selected by the kernel scheduler to run either on the current HART or another HART in a multi-processor system.

5.6.2 Context Switch

The OpenBSD process subsystem perform a context switch to swap between different active processes running on the system. Context switches can happen for a number of reasons. For example, a process attempts to read from a hard disk and gets stuck waiting for the disk to return the requested data. Instead of allowing the HART to sit idly waiting on blocking I/O, the system call to read from disk may also trigger a context switch to assign another process to the HART.

A context switch starts by storing the execution state of the active process before switching to a new process. The active process's execution state is already mostly saved by the supervisor trap handler upon entry into kernel. A trap handler may signal to the kernel scheduler to yield the HART to another process depending on the circumstances. The machine-independent scheduler code delegates to `cpu_switchto()` to perform the machine-dependent aspects of the context switch. The machine-dependent context switch begins by saving the kernel execution state into a `switchframe`, as shown in Snippet 26. This step may be skipped in certain scenarios where the kernel does not care about the old process state, such as when a process is terminating.

Note that in the block of assembly identified by ⑤, the `switchframe` only stores callee-saved registers as the caller to `cpu_switchto()` has already saved important temporary registers to the stack, as required by RISC-V calling conventions. The

```
/*
 * cpu_switchto(struct proc *oldproc, struct proc *newproc)
 *      a0      'struct proc *' of the old context
 *      a1      'struct proc *' of the new context
 */
ENTRY(cpu_switchto)
        // check if old context needs to be saved
        beqz    a0, 1f

        // create switchframe ⑤
        addi    sp, sp, -SWITCHFRAME_SIZEOF
        sd      s0, (SF_S + 0 * 8)(sp)
        sd      s1, (SF_S + 1 * 8)(sp)
        sd      s2, (SF_S + 2 * 8)(sp)
        sd      s3, (SF_S + 3 * 8)(sp)
        sd      s4, (SF_S + 4 * 8)(sp)
        sd      s5, (SF_S + 5 * 8)(sp)
        sd      s6, (SF_S + 6 * 8)(sp)
        sd      s7, (SF_S + 7 * 8)(sp)
        sd      s8, (SF_S + 8 * 8)(sp)
        sd      s9, (SF_S + 9 * 8)(sp)
        sd      s10, (SF_S + 10 * 8)(sp)
        sd      s11, (SF_S + 11 * 8)(sp)
        sd      ra, SF_RA(sp)

        // store switchframe
        ld      a5, CI_CURPCB(tp)
        sd      sp, PCB_SP(a5)

        ...
END(cpu_switchto)
```

Snippet 26. The machine-dependent context switch function begins by saving the kernel execution state related to the active process.

`switchframe` is stored on the kernel stack and the `proc` and `cpu_info` structures are updated before switching to the new virtual address space of the target process, as shown in Snippet 27.

The new process' stack pointer is stored in a saved register so that it persists across the transition to the new virtual address space, which is implemented via the call to `pmap_set_satp()`. Immediately following the switch, the new process' stack pointer is restored to the `sp` register, so that the `switchframe` of the target process' kernel state can be restored from the target process' kernel stack. The registers are restored to the HART and the execution returns to the caller.

```
ENTRY(cpu_switchto)
    ...

        li      a5, SONPROC
        sb      a5, P_STAT(a1)          // Mark new on cpu
        sd      tp, P_CPU(a1)           // Store curcpu
        ld      a5, P_ADDR(a1)          // Load new pcb
        sd      a5, CI_CURPCB(tp)
        sd      a1, CI_CURPROC(tp)

        ld      s1, PCB_SP(a5)          // load new stack pointer
        mv      a0, a1
        la      t0, pmap_set_satp
        jalr    t0

        mv      a7, s0                  // move retguard random
        mv      sp, s1                  // restore stack pointer

        ...
END(cpu_switchto)
```

Snippet 27. The machine-dependent context switch function updates the `proc` and `cpu_info` structures and switches to the virtual address space of the target process.

5.7 Device Subsystem Implementation

With the `autoconf` framework as presented in Section 4.6, all devices are attached in a tree hierarchy as illustrated in Figure 15. This section elaborates on the machine-dependent devices identified in this device hierarchy. Device definition and configuration are separated into Appendix A.

5.7.1 Mainbus

`mainbus0` is the root of the FDT. It is the starting point of the `autoconf` procedure. `mainbus0` is defined in Snippet 37 at location ⑥ in Appendix A, where an attribute named `mainbus` is defined such that all its children devices can attach to this attribute. `mainbus0`'s parent is a virtual attribute `root`, which is introduced to simplify device configuration.

With this definition, `mainbus0` can be configured as in Snippet 38 at location ⑧ in Appendix A.

The device definition and configuration is synthesized by **config(8)** to generate a descriptor in *ioconf.c*. This "machine descriptor" file drives the `autoconf` matching and attaching procedure. The key data structure in *ioconf.c* is `struct cfdata cfdata[]`, which stores the descriptors for multiple devices, as shown in Snippet 39 in Appendix A. For example, the `cfdata` entry for `mainbus0` (at location ⑨ in Snippet 39) provides:

- `mainbus0`'s attachment-relevant data structures in `mainbus_ca`.
- `mainbus0`'s driver-relevant data structures in `mainbus_cd`.
- `mainbus0`'s parent as `pv+3`, which equals to `-1` (`root`).
- `mainbus0`'s unit number starting from 0.

The call stack leading to `mainbus0`'s configuration is `main()` → `cpu_configure()` → `config_rootfound("mainbus", NULL)`. The `config_rootfound()` will call `mainbus_match()` then `mainbus_attach()`. `mainbus_match()` always returns '1', which means `mainbus0` will always be attached, as it handles both FDT and non-FDT machines. `mainbus_attach()` mainly completes the following tasks:

1) Call `riscv_intr_init_fdt()` to initialize all interrupt controllers. This function discovers all interrupt controllers by iterating the FDT in a depth-first manner and matching the property '`interrupt-controller`' of each node. It then allocates memory for each newly-discovered interrupt controller and inserts it to the global interrupt controllers list.
2) Call `riscv_timer_init()` to attach `riscv_timer_delay()` to the generic stub `riscv_clock_func.delay`. This provides a dummy delay method before timer is attached.
3) Call `mainbus_attach_cpus()` to attach the primary CPU, as detailed in Section 5.7.3.

4) Iterate through each child of `mainbus0` and call `mainbus_attach_node()` to attach nodes with locator `early=1` followed by nodes with locator `early=0`.
5) Call `mainbus_attach_framebuffer()` to attach frame buffer.
6) Call `mainbus_attach_cpus()` to attach secondary CPUs.

`mainbus_attach_node()` first constructs the `fdt_attach_args` by pulling properties from FDT for the given node. It then passes this `fdt_attach_args` to `config_found_sm()` as the `aux` argument so that this child device can be attached in *direct* configuration mode.

5.7.2 UART

The UART device is presented to kernel via FDT, as defined in Snippet 28. The FDT specifies that `com0` has a baud rate of `0x384000` and `com0` reports interrupt (with `irqno=0x0000000a`) to PLIC (whose `phandle=0x00000003`).

```
uart@10000000 {
    interrupts = <0x0000000a>;
    interrupt-parent = <0x00000003>;
    clock-frequency = <0x00384000>;
    reg = <0x00000000 0x10000000 0x00000000 0x00000100>;
    compatible = "ns16550a";
};
```

Snippet 28. `com0` definition in FDT

`com0` is initialized even before `autoconf` starts. This early initialization enables `printf` during kernel bootstrap. `printf` prints one character to the console via the call stack illustrated in Figure 21. The call stack can be divided into two layers, MI layer and MD layer. The key connection between MI and MD components is `cn_tab`, which is hooked to `com_fdt_cons` in the early initialization routine `com_fdt_init_cons()`. This routine maps `com0` to the virtual address space so that it can be accessed as an MMIO device via `generic_space_read/write_1()`.

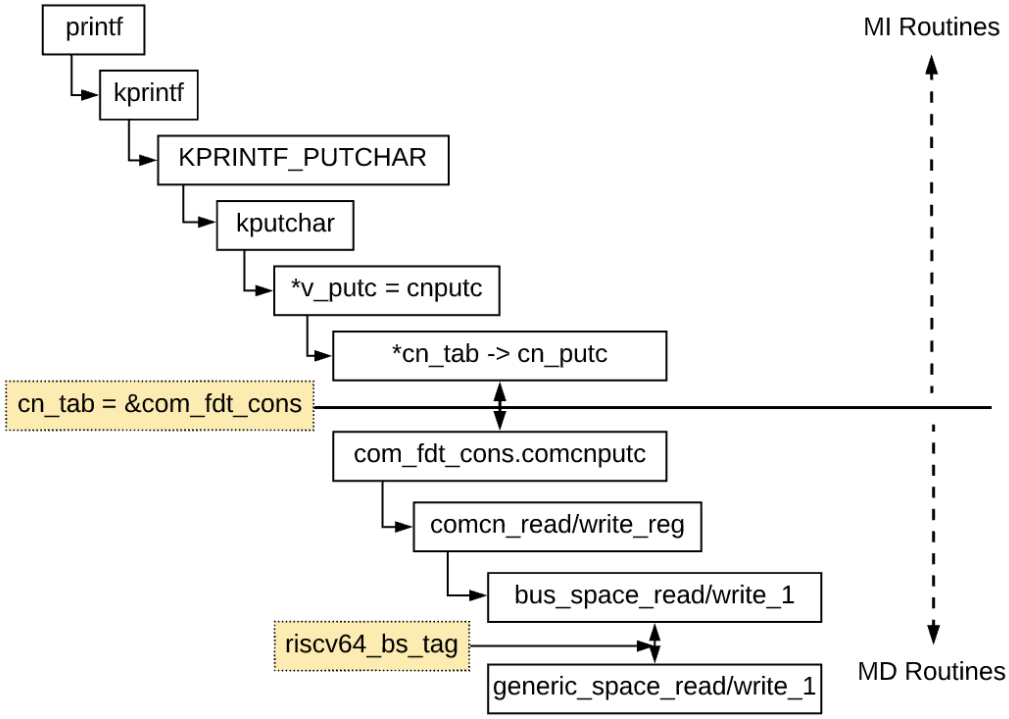

Fig. 21. The MI-MD interface for `printf`.

Different from other MD devices as defined in *files.riscv64*, the `com0` device is defined in '*/sys/conf/files*':

```
device  com: tty
file    dev/ic/com.c    com & (com | com_cardbus | com_gsc |
                                com_isapnp)      needs-flag
```

`com0` is attached to `attribute [fdt]` as `com_fdt` in '*/sys/dev/fdt/files.fdt*':

```
attach  com at fdt with com_fdt
file    dev/fdt/com_fdt.c               com_fdt
```

files.riscv64 includes *files.fdt* as shown in Snippet 37. With these definitions, `com0` can be configured as shown in Snippet 38. The resultant device descriptor for `com0` is given in Snippet 39.

com0 is formally attached as a child node of mainbus0 in *direct* configuration mode during mainbus_attach().

com_fdt_match() simply checks whether the given aux's node is compatible with ns16550a while com_fdt_attach() tackles the following tasks:

1) Enable clock if the UART device has an embedded clock.
2) Obtain clock-frequency.
3) Populate the com_softc data structure.
4) Map com0 to virtual address space via bus_space_map().
5) Call com_attach_subr() to configure com0.
6) Establish the connection of com0's interrupt handler to plic0 via fdt_intr_establish().

5.7.3 HART

cpu0 is the primary HART, which is selected to boot the kernel.

A HART is defined in FDT as shown in Snippet 29. The HART ID is stored in reg field. The supported ISA set is stored in the "riscv,isa" field. The timebase-frequency field gives the real-time clock frequency (10MHz). This property is used to match timer0.

CPU's device definition is given in Snippet 37. cpu0's configuration is listed in Snippet 38. The resultant device descriptor of cpu0 is presented in Snippet 39. Driven by this descriptor, following what was described in Section 5.7.1, mainbus_attach_cpus() first identifies the '/cpus' node in FDT, then calls mainbus_attach_node() for each child node of '/cpus'. mainbus_attach_cpus() accepts a cfmatch_t-type argument: match. The two cfmatch_t functions assigned to mainbus_match_primary() and mainbus_match_secondary() distinguishes the primary and secondary HARTs by calling the matched cfdata's cpu_match() function.

```
cpus {
    #address-cells = <0x00000001>;
    #size-cells = <0x00000000>;
    timebase-frequency = <0x00989680>;
    cpu-map {
        cluster0 {
            core0 {
                cpu = <0x00000001>;
            };
        };
    };
    cpu@0 {
        phandle = <0x00000001>;
        device_type = "cpu";
        reg = <0x00000000>;
        status = "okay";
        compatible = "riscv";
        riscv,isa = "rv64imafdcsu";
        mmu-type = "riscv,sv48";
        interrupt-controller {
            ...
        };
    };
};
```

Snippet 29. HART definition in FDT

For primary HART, `cpu_match()` returns true only if its HART ID equals `boot_hart`, which is saved during kernel bootstrap, as shown in Snippet 30.

```
    la  t0, boot_hart   //the HART kernel boots on.
    sw  a0, 0(t0)       //all above logic runs on this a0 HART.
```

Snippet 30. The boot HART ID is saved to `boot_hart` during kernel bootstrap in *locore.S*.

For secondary HART, `cpu_match()` returns true only if the total number of discovered HARTs (excluding the one that is under discovery), `ncpus`, is smaller than `MAXCPUS`. This logic is shown in Snippet 31.

`cpu_attach()` handles the following tasks:

1) Allocate memory for `cpu_info` (Snippet 32), which maintains the management information for a HART. If current HART is the `boot_hart`, it will instead use the pre-allocated `cpu_info_primary` directly. Otherwise, a `cpu_info` structure is

```
int
cpu_match(struct device *parent, void *cfdata, void *aux)
{
    ...
    if (ncpus < MAXCPUS || faa->fa_reg[0].addr == boot_hart) /* the primary cpu
        ↪  */
                return 1;
}
```

Snippet 31. CPU match function key logic as implemented in*cpu.c*.

allocated for the newly-discovered HART. All HART's `cpu_info` are organized in `cpu_info_list`.

2) Update the following fields of `struct cpu_info *ci`:
   - `ci->ci_dev`, which points to the `device` data structure for this HART and which is allocated and initialized during call stack `config_attach()`→`config_make_softc()`.
   - `ci->ci_cpuid`, which equals `dev->dv_unit`. This is the instance index of CPU's driver but *not* the HART ID.
   - `ci->ci_node`, which equals the HART node index in FDT.
   - `ci->self`, which points to this `cpu_info` itself.
3) Call `cpu_identify()` to identify the HART's vendor, part number and supported ISA set.
4) Attach current HART's child devices, e.g., `timer0` and `intc0`. Section 5.7.4 and Section 5.7.5 discuss these two child devices in greater detail.

5.7.4 Timer

The definition of device `timer0` is presented in Snippet 37. Its configuration in the *GENERIC* file is listed in Snippet 38. The resultant device descriptor of `timer0` in *ioconf.c* is given in Snippet 39. Driven by this file, attaching `timer0` happens inside `cpu_attach()`, with a call to `config_found_sm()`.

```
struct cpu_info {
        struct device           *ci_dev; /* Device corresponding to this CPU */
        struct cpu_info         *ci_next;
        struct schedstate_percpu ci_schedstate; /* scheduler state */

        u_int32_t               ci_cpuid;
        int                     ci_node;
        struct cpu_info         *ci_self;

        struct proc             *ci_curproc;
        struct pmap             *ci_curpm;
        u_int32_t               ci_randseed;

        struct pcb              *ci_curpcb;
        struct pcb              *ci_idle_pcb;

        u_int32_t               ci_ctrl; /* The CPU control register */

        uint32_t                ci_cpl;
        uint32_t                ci_ipending;
        uint32_t                ci_idepth;
#ifdef DIAGNOSTIC
        int                     ci_mutex_level;
#endif
        int                     ci_want_resched;

#ifdef MULTIPROCESSOR
        struct srp_hazard       ci_srp_hazards[SRP_HAZARD_NUM];
        volatile int            ci_flags;
        volatile int            ci_ddb_paused;
#define CI_DDB_RUNNING          0
#define CI_DDB_SHOULDSTOP       1
#define CI_DDB_STOPPED          2
#define CI_DDB_ENTERDDB         3
#define CI_DDB_INDDB            4

#endif

#ifdef GPROF
        struct gmonparam        *ci_gmon;
#endif
};
```

Snippet 32. `cpu_info` data structure definition in *cpu.h*.

`riscv_timer_match()` will return 1 if `timer0` has *not* been attached yet and the `/cpus` node in the FDT has a `timebase-frequency` property.

`riscv_timer_attach()` is called from `config_attach()` following allocation of `riscv_timer_softc` for `timer0`. `riscv_timer_attach()` first obtains the real-time clock frequency and stores it in `riscv_timer_softc`. It then

calls `riscv_clock_register()` to hook `riscv_timer_cpu_initclocks()`, `riscv_timer_delay()` and `riscv_timer_startclock()` to the generic stubs in `riscv_clock_func`.

`riscv_timer_cpu_initclocks()` is called from call stack `main()`→`initclocks()`→`cpu_initclocks()` and it needs to handle the following tasks:

1) Update `sc_ticks_per_intr` in `riscv_timer_softc`.
2) Establish timer interrupt handler to HLIC's handlers vector.
3) Calculate the time when the first timer interrupt should happen and set this value to `mtimecmp` via `sbi_set_timer()`.
4) Enable timer interrupt by setting `sie.STIE` field.

`riscv_timer_startclock()` is called from `cpu_start_secondary()`→`cpu_startclock()` to start timer interrupt for secondary HARTs by performing only the last two tasks listed above.

`riscv_timer_delay()` provides two delay mechanisms:

- If `timer0` has not been attached, delay with a dummy nested for-loop.
- If `timer0` is attached, use this timer to count the total delay cycles.

5.7.5 HART-Level Interrupt Controller

The functional feature of HLIC has been demonstrated in paragraph 5.5.2.3. This subsection mainly focuses on how `intc0` is defined, probed and configured.

The HLIC is presented to kernel via FDT as a sub-device of HART `cpu0`. Snippet 33 demonstrates a partial FDT with two HARTs, where each HART has a dedicated HLIC distinguished by `phandle`.

The definition of `intc0` is presented in Snippet 37. `intc0`'s configuration in *GENERIC* is included in Snippet 38. The resultant device descriptor of `intc0` in *ioconf.c* is presented in Snippet 39. Driven by this file, `intc0` is attached during

```
cpus {
    ...
    cpu@0 {
        ...
        interrupt-controller {
            #interrupt-cells = <0x00000001>;
            interrupt-controller;
            compatible = "riscv,cpu-intc";
            phandle = <0x00000004>;
        };
    };
    cpu@1 {
        ...
        interrupt-controller {
            #interrupt-cells = <0x00000001>;
            interrupt-controller;
            compatible = "riscv,cpu-intc";
            phandle = <0x00000002>;
        };
    };
};
```

Snippet 33. HART-Level Interrupt Controller definition in FDT

`cpu_attach()` via a call to `config_found()`. The `aux` stores `intc0`'s node index in FDT, which implies `intc0` is attached in direct-config mode.

`riscv_intc_match()` returns 1 if the given node has property `interrupt-controller` and is compatible with "`riscv,cpu-intc`".

`riscv_intc_attach()` firstly needs to hook the HART-specific interrupt handler `riscv_intc_irq_handler` to the generic stub `riscv_cpu_intr` via `riscv_set_intr_handler()`. After this establishment, an interrupt can trap to a HART via the following step: `stvec` → `cpu_trap_handler()` → `cpu_trap_handler_supervisor()` → `do_trap_supervisor()` → `riscv_cpu_intr()` → `riscv_intc_irq_handler()`.

`riscv_intc_attach` secondly needs to register itself to the global interrupt controllers list via a call to `riscv_intr_register_fdt()`.

5.7.6 *Simplebus*

The SoC (system on a chip) device in FDT (as shown in Snippet 34) is abstracted to simplebus0. It is the parent of PCI devices, PLIC, and CLINT.

```
soc {
    #address-cells = <0x00000002>;
    #size-cells = <0x00000002>;
    compatible = "simple-bus";
    ranges;
    pci@30000000 {
        ...
    };
    interrupt-controller@c000000 {
        ...
    };
    clint@2000000 {
        ...
    };
};
```

Snippet 34. simplebus0 definition in FDT

The definition of simplebus0 is shown in Snippet 37. The configuration of simplebus0 is given in Snippet 38. The device descriptor for simplebus0 is shown in Snippet 39.

The call stack to attach simplebus0 is from: mainbus_attach() → mainbus_attach_node() → config_found_sm(), with locator early=1. The SoC node in FDT is passed to mainbus_attach_node() so that all SoC's information is parsed and stored into fdt_attach_args. This fdt_attach_args is then passed to config_found_sm() as aux, proving that simplebus0 is configured in direct-mode. config_found_sm() will ultimately call simplebus_match() and simplebus_attach() for matching and attaching, respectively.

simplebus_match() simply checks if the FDT node buried in argument aux is compatible with simple-bus.

`simplebus_attach()` is called after `config_attach()` allocates `simplebus_softc` and configures the embedded `device` structure. `simplebus_attach()` further populates the `simplebus_softc` data structure. After that, it scans the FDT tree to attach all child devices of `simplebus0` via `simplebus_attach_node()`. Devices with locator `early=1` are attached first.

`simplebus_attach_node()` is similar to `mainbus_attach_node()`. It first constructs the `fdt_attach_args` by parsing and pulling in properties for the given FDT node. It then passes this `fdt_attach_args` to `config_found_sm()` as its `aux` argument to recursively attach its children devices.

5.7.7 Platform-Level Interrupt Controller

This device is the global interrupt controller, PLIC. Its functional feature has been fully discussed in paragraph 5.5.2.2. This subsection further explains how `plic0` is configured and attached.

PLIC is presented to kernel via FDT. Its definition is given in Snippet 35.

```
soc {
    ...
    interrupt-controller@c000000 {
        phandle = <0x00000003>;
        riscv,ndev = <0x00000035>;
        reg = <0x00000000 0x0c000000 0x00000000 0x04000000>;
        interrupts-extended = <0x00000002 0x0000000b 0x00000002 0x00000009>;
        interrupt-controller;
        compatible = "riscv,plic0";
        #interrupt-cells = <0x00000001>;
        #address-cells = <0x00000000>;
    };
    ...
};
```

Snippet 35. PLIC definition in `autoconf`

PLIC's device definition is listed in Snippet 37. `plic0`'s configuration is in Snippet 38. The resultant descriptor for `plic0` is shown in Snippet 39. `plic0` is

attached as a child device of `simplebus0`. A summarized call stack to attach `plic0` is shown in Snippet 36 to demonstrate that `autoconf` is essentially a recursive process.

```
config_rootfound("mainbus")
  |
  |-config_rootsearch("mainbus")
  |
  |-config_attach(parent=ROOT)
     |
     |-mainbus_attach(parent=ROOT)
        |
        |-mainbus_attach_node(node=simplebus0)
          |
          |-config_found_sm(parent=mainbus0, aux=simplebus0)
             |
             |-config_search
             |    |
             |    |-simplebus_match
             |
             |-config_attach
                |
                |-simplebus_attach
                    |
                    |-simplebus_attach_node(node=plic0)
                       |
                       |-config_found_sm(parent=simplebus0,
                           |                       aux=plic0)
                           |-config_search
                           |    |
                           |    |-plic_match
                           |
                           |-config_attach
                               |
                               |-plic_attach
```

Snippet 36. Demonstration the call stack to `plic_attach` to show that `autoconf` is essentially a recursive process.

`plic_match()` simply checks the given FDT node's compatibility with "`riscv,plic0`" or "`sifive,plic-1.0.0`".

`plic_attach()` needs to complete the following tasks:

1) Determine the number of external devices that report interrupt to `plic0`.
2) Map `plic0` to the virtual address space, so that all its memory-mapped registers can be accessed via the generic `bus_space_write_4()` and `bus_space_read_4()` operations.

3) Allocate memory for `PLIC_MAX_IRQS` number of `plic_irqsrc` and initialize each `plic_irqsrc`'s interrupt handler list.
4) Prepare the *enable* and *context* register offset for each context (essentially a HART in supervisor-mode).
5) Call `plic_calc_mask()` to:
   - Update each interrupt source's maximum and minimum priority. This is because each interrupt source could have multiple handlers attached, and each handler could require different priorities corresponding to its own executing context.
   - Enforce the `min_priority` to `plic0`'s `priority` register for this interrupt source.
   - Update the current context (`curcpu()`) `threshold` register to `min_priority - 1`.
   - Set the intersection of this interrupt source and current context to `IRQ_ENABLE`.
6) Establish external interrupts' overall trap entry `plic_irq_handler` to the HLIC's `intc_handler[IRQ_EXTERNAL_SUPERVISOR]`.
7) Update `riscv_intr_func` and substitute the previous default `SPL`-relevant routines to `PLIC`-specific ones. This is required as a new `SPL` must be enforced to `plic0`.
8) Hook `plic_intr_establish_fdt()` to `plic0`'s `softc`, so that devices can use this routine to establish their interrupt handler to `plic0`.
9) Register `plic0` to the global interrupt controllers list.

6 PROJECT TESTING AND VERIFICATION

This section describes the end-to-end test flow and test methodology applied throughout the project for quality assurance. The overall test flow is shown in Figure 22. The methodology includes test model, test criteria, test coverage, and test tools.

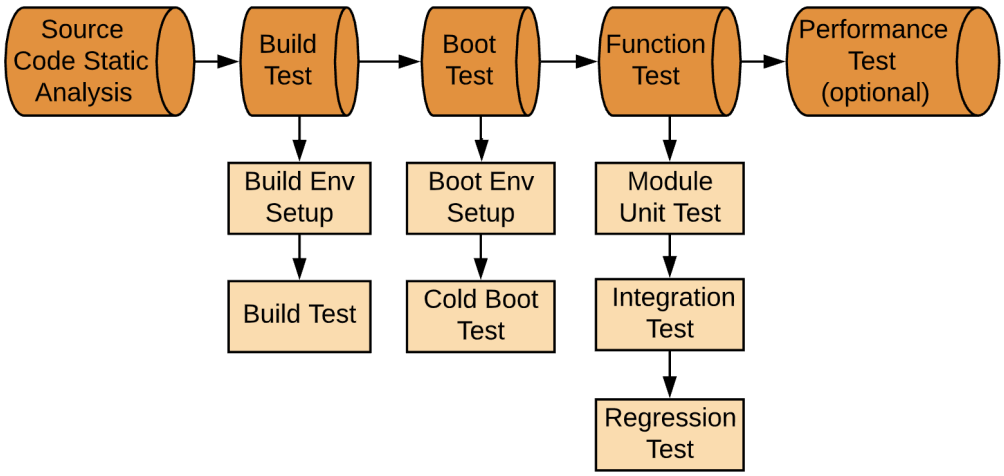

Fig. 22. Project quality assurance flow.

6.1 Build Test

This test focus on the build tool availability, source code syntax correctness, and library readiness.

The build environment involves:

- Host OS: OpenBSD.
- Cross toolchain: Clang/LLVM with support for RISC-V ISA.
- OpenBSD source tree.

Kernel cross-compile build should be able to compile, assemble, and link to generate an integrated kernel image. The cross-compiled kernel image should have appropriate ELF headers and appear reasonably formed when run through `objdump` or `readelf`.

6.2 Boot Test

Boot test verifies the stability, and consistency for both bootloader and OpenBSD ported kernel. The boot test environment involves:

- OpenBSD host machine.
- QEMU with support for *qemu-system-riscv64*.
- BBL source tree.
- Pre-built OpenBSD kernel.

The boot test steps include:

1) Build BBL with OpenBSD kernel as its payload.
2) Start kernel by executing '*qemu-system-riscv64 -kernel bbl [other arguments]*'.
3) Check QEMU console output to verify kernel boot progress.
4) Attach GDB debugger to perform single-step debugging, as necessary.

6.3 Function Test

Function test is divided into module unit test and system integration test.

6.3.1 Module Unit Test

Module unit test took up most of the testing effort. Unit test mainly uses white box testing methodology, which is based on source code structures, data, and internal program logic. White box test is executed by software developers.

- Test model: Program flow graph model
- Basic path testing, branch testing, and condition-based testing
  - Basic path test derives linearly independent paths from program flow graph with each independent path corresponding to a test case.
  - Cyclomatic complexity determines the number of minimum basic path test cases.
  - Branch test focuses on predicate node and branch link, and validates each branch.
  - Condition-based test further decomposes compound Boolean expressions into truth tables/sub-branches.

- Coverage:
  - Program source code line coverage
  - Program predicate node coverage
  - Program branch coverage
  - Program control path coverage
  - Program data flow coverage
  - Program logic coverage

6.3.2 System Integration Test

This project natively fits bottom-up system integration test. Porting an OS kernel primarily focuses on the machine-dependent layer, which sits at the bottom level. As the high-level machine-independent functions remain intact, they can serve as the test driver for the low-level function blocks directly.

System integration test is challenging because the machine-dependent modules are tightly coupled with each other. For instance, the kernel bootstrap module designed in this spec will affect any other module that requires access to the page table, such as memory management module, process management module, etc.

After the source code update, regression tests should also be conducted.

7 SUMMARY, CONCLUSIONS, AND RECOMMENDATIONS

7.1 Summary

With the target to port a secured operating system (OpenBSD) onto a secured architecture (RISC-V), the following milestones have been reached in this project.

A develop environment has been established that incorporates kernel source code editing, kernel image compilation, and kernel functionality verifying. The develop environment consists of an `x86`-based OpenBSD host machine, an LLVM cross-compilation toolchain, a bootloader (BBL), an emulated RISC-V hardware platform on QEMU, and a remote debugger GDB. With these tools, the kernel source code is adapted and cross-compiled onto a `riscv64` target. The resultant OpenBSD kernel is then encapsulated into BBL as its payload to run on the QEMU-emulated `riscv64` machine. QEMU provides the GDB stub which can be remotely connected to GDB to probe the kernel's execution.

The ported OpenBSD kernel can complete an early-stage bootstrap within the Supervisor Execution Environment provided by QEMU and BBL. The kernel bootstrap is implemented to construct an initial page table, to enable `Sv39`-mode paging, to set up the overall trap entry, to construct a C runtime and jump to the C routine to continue the bootstrap procedure.

The ported OpenBSD kernel can manage 512 GiB virtual address space with the UVM memory subsystem. Specifically, the OS kernel can create, activate, and update in-memory page tables after adapting the machine-dependent physical map layer to the RISC-V's MMU, which operates in `Sv39`-mode.

The ported OpenBSD kernel can manually construct a process context for itself to become process 0. From this process 0, the kernel can spawn more processes and organize them in a tree hierarchy. The ported kernel implements the machine-dependent

context switch routine so that it can schedule multiple processes to time-slice a single RISC-V HART.

The ported OpenBSD kernel can handle traps that arise due to both interrupts and exceptions. Timer interrupts and external interrupts can be routed and processed via a two-level nested interrupt controller architecture. Exceptions due to environment call from user-mode can be utilized to implement system calls. Exceptions due to memory faults are generally handled by aborting the data access.

The ported OpenBSD kernel can probe and configure devices including HART, UART, timer, and interrupt controllers into a tree hierarchy by leveraging the FDT and `autoconf` framework and providing the device definition, configuration, and descriptor source code.

7.2 Conclusions

To port OpenBSD to RISC-V ISA, this project has implemented assembly initialization for kernel early bootstrap, adapted PMAP to RISC-V MMU in Sv39 mode, implemented routines for process creation and context switch, developed trap handlers for interrupts and exceptions, and developed drivers to probe and attach MD devices.

7.3 Recommendations for Further Development

Despite the milestones achieved in this project to port the OpenBSD kernel to RISC-V, there is still significant work to be done to get the OpenBSD operating system into a stable stable for general purpose use. To bound project complexity, a number of features have yet to be implemented. Potential future work to bring the OpenBSD operating system to a stable state for general purpose use include:

- Separate out parts of `locore.S` into `locore0.S`.
- Support for the `Sv48` virtual memory address translation scheme.
- Audit of RISC-V MD kernel source for bugs and other vulnerabilities.
- Build userland components (`libc`, `libcompiler_rt`, and others).

- Generate a root file system image of OpenBSD userland components.
- Support for Simultaneous Multi-Processing.
- Support for Multi-user mode.

References

[1] RISC-V Foundation, "RISC-V history." RISCV.org. https://riscv.org/risc-v-history/ (accessed Apr. 22, 2019).

[2] H. H. Porter III, "RISC-V: An overview of the instruction set architecture," Jan 2018. [Online]. Available: http://web.cecs.pdx.edu/~harry/riscv/RISCV-Summary.pdf.

[3] D. Patterson, "50 years of computer architecture: From the mainframe CPU to the domain-specific TPU and the open RISC-V instruction set," in *2018 IEEE International Solid-State Circuits Conference (ISSCC)*, pp. 27–31, Feb. 2018.

[4] RISC-V Foundation, "RISC-V cores." RISCV.org. https://riscv.org/risc-v-cores/ (accessed Apr. 22, 2019).

[5] P. Kocher, J. Horn, A. Fogh, D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher, M. Schwarz, and Y. Yarom, "Spectre attacks: Exploiting speculative execution," in *2019 IEEE Symposium on Security and Privacy (SP)*, pp. 1–19, May 2019.

[6] M. Lipp, M. Schwarz, D. Gruss, T. Prescher, W. Haas, A. Fogh, J. Horn, S. Mangard, P. Kocher, D. Genkin, Y. Yarom, and M. Hamburg, "Meltdown: Reading kernel memory from user space," in *27th USENIX Security Symposium (USENIX Security 18)*, pp. 973–990, Aug. 2018.

[7] LLVM Developer Group, "The LLVM compiler infrastructure: LLVM overview." LLVM.org. https://llvm.org (accessed Apr. 6, 2020).

[8] M. Larabel, "Clang to become the default FreeBSD compiler soon." phoronix.com. https://www.phoronix.com/scan.php?page=news_item&px=MTE4NDQ (accessed Apr. 6, 2020).

[9] M. Larabel, "OpenBSD switches to Clang compiler for i386/AMD64." phoronix.com. https://www.phoronix.com/scan.php?page=news_item&px=OpenBSD-Default-Clang (accessed Apr. 6, 2020).

[10] LLVM Developer Group, "Clang: A C language family frontend for LLVM." LLVM.org. https://clang.llvm.org/ (accessed Apr. 7, 2020).

[11] QEMU Team, “QEMU: The FAST! processor emulator.” QEMU.org. https://www.qemu.org/ (accessed Apr. 10, 2020).

[12] M. Clark, “RISC-V QEMU part 1: Privileged ISA v1.10, HiFive1 and virtio.” SiFive.com. https://www.sifive.com/blog/risc-v-qemu-part-1-privileged-isa-hifive1-virtio (accessed Apr. 10, 2020).

[13] A. Waterman and K. Asanović, “The RISC-V instruction set manual, volume II: Privileged architecture,” tech. rep., RISC-V Foundation, June 2019. [Online]. Available: https://riscv.org/specifications/privileged-isa/.

[14] C. D. Cranor and G. M. Parulkar, “The UVM virtual memory system,” in *Proceedings of the USENIX Annual Technical Conference*, pp. 117–130, June 1999.

[15] A. Waterman and K. Asanović, “The RISC-V instruction set manual, volume I: Unprivileged ISA,” tech. rep., RISC-V Foundation, Dec. 2019. [Online]. Available: https://riscv.org/specifications/isa-spec-pdf/.

[16] SiFive, “Sifive interrupt cookbook, version 1.2.” SiFive, Inc., Feb. 2020. [Online]. Available: https://sifive.cdn.prismic.io/sifive/d1984d2b-c9b9-4c91-8de0-d68a5e64fa0f_sifive-interrupt-cookbook-v1p2.pdf.

[17] M. Li, “Environment setup notes for porting OpenBSD onto RISC-V.” github.com. https://github.com/MengshiLi/openbsd-riscv-notes (accessed Apr. 10, 2020).

Appendix A

DEVICE DEFINITION AND CONFIGURATION SOURCE CODE

A.1 Device Definition Code

The device definition source code in *files.riscv64* is written using a "machine description" language. The definition for machine-dependent devices in this project is presented in Snippet 37.

```
# mainbus ⑥
define  mainbus {[early = 0]}
device  mainbus: fdt
attach  mainbus at root
file    arch/riscv64/dev/mainbus.c

# cpu
define  cpu {}
device  cpu
attach  cpu at mainbus
file    arch/riscv64/riscv64/cpu.c

# timer
device  timer
attach  timer at cpu
file    arch/riscv64/dev/timer.c

# HART-Level Interrupt Controller
device  intc
attach  intc at cpu
file    arch/riscv64/dev/riscv_cpu_intc.c

# simplebus
define  simplebus {[early = 1]}
device  simplebus
attach  simplebus at mainbus
file    arch/riscv64/dev/simplebus.c

# PLIC
device  plic
attach  plic at simplebus
file    arch/riscv64/dev/plic.c

...

# Machine-independent FDT drivers ⑦
include "dev/fdt/files.fdt"
```

Snippet 37. Device definition in *files.riscv64* for Mainbus, HART, Timer, HLIC, Simplebus, and PLIC

A.2 Device Configuration Code

The device configuration code is presented in *GENERIC* as shown in Snippet 38, which specifies the device's attachment point, and whether or not the device should be attached early in the `autoconf` procedure.

```
# mainbus    ®
mainbus0          at root

# cpu0
cpu0              at mainbus0

# timer0
timer0            at cpu0

# intc0
intc0             at cpu0

# NS16550 compatible serial ports
com*              at mainbus0 early 1

# simplebus0
simplebus*        at mainbus0 early 1

# Platform Level Interrupt Controller
plic*             at simplebus? early 1

# virtio
virtio*           at mainbus0
vioblk*           at virtio?

scsibus*          at scsi?
sd*               at scsibus?
```

Snippet 38. Device configuration in *GENERIC* for `mainbus0`, `cpu0`, `timer0`, `intc0`, `com*`, `simplebus0`, and `plic0`

A.3 Auto-generated Device Configuration Table

From the device definition and configuration as presented above, an *ioconf.c* source file will be automatically generated by the **config(8)** framework. Snippet 39 lists the key data structure in *ioconf.c*, which describes each device's driver and attachment rules.

```
struct cfdata cfdata[] = {
    /* attachment   driver  unit  state loc flags parents nm starunit1 */
/*  2: mainbus0 at root */      ⑨
    {&mainbus_ca,    &mainbus_cd,     0, NORM,     loc,    0, pv+ 3, 0,    0},
/*  3: cpu0 at mainbus0 early 0 */
    {&cpu_ca,        &cpu_cd,         0, NORM, loc+  3,    0, pv+ 4, 1,    0},
/*  4: timer0 at cpu0 */
    {&timer_ca,      &timer_cd,       0, NORM,     loc,    0, pv+12, 2,    0},
/*  5: intc0 at cpu0 */
    {&intc_ca,       &intc_cd,        0, NORM,     loc,    0, pv+12, 2,   0},
/*  6: simplebus* at mainbus0 early 1 */
    {&simplebus_ca, &simplebus_cd,   0, STAR, loc+  2,    0, pv+ 4, 1,    0},
/*  7: plic* at simplebus* early 1 */
    {&plic_ca,       &plic_cd,        0, STAR, loc+  2,    0, pv+ 6, 3,   0},
/*  8: vioblk* at virtio* */
    {&vioblk_ca,     &vioblk_cd,      0, STAR,     loc,    0, pv+10, 4,    0},
/*  9: virtio* at mainbus0 early 0 */
    {&virtio_mmio_ca,  &virtio_cd,   0, STAR, loc+  3,    0, pv+ 4, 1,   0},
/* 10: com* at mainbus0 early 1 */
    {&com_fdt_ca,    &com_cd,         0, STAR, loc+  2,    0, pv+ 4, 1,    0},

    ...
    {(struct cfattach *)-1}
};
```

Snippet 39. Device descriptions in *ioconf.c* for `mainbus0`, `cpu0`, `timer0`, `intc0`, `simplebus0`, `plic0`, and `com*`, etc.