# 第一章 操作系统内核构建概述

## 1.1 什么是操作系统（参考rcore讲体系结构，学xv6穿一下后续章节的核心概念）

### 1.1.1 体系结构

### 1.1.2 系统调用（抽象概念）

### 1.1.3 进程和内存 97

### 1.1.4 I/O与文件系统

## 1.2 NPUcore操作系统（参考xv6写法，大致讲一下，参考rcore“操作系统抽象”部分）（列出NPUcore全部系统调用）

## 1.3 预备知识及技能

### 1.3.1 RISC-V和LoongArch指令集介绍

### 1.3.2 Rust语言及其主要特性

### 1.3.3 如何查资料

# 第二章 内核构建初探

## 2.1 NPUcore内核运行

### 2.1.1 本地运行NPUcore

### 2.1.2 在线运行NPUcore

## 2.2 基于GDB内核调试

### 2.2.1 认识GDB

### 2.2.2 基于GDB的内核调试 10页

### 2.2.3 git bisect——快速问题定位

## 2.3 NPUcore内核代码结构及内核构建目标

### 2.3.1 NPUcore内核代码树（补充文件目录，参考Titanix）

### 2.3.2 NPUcore学习路线

## 2.4 实验

\section{基于GDB内核调试}

在软件开发过程中，调试是不可避免的。一个程序往往不会一开始就按照程序员预期的方式运行, 对操作系统这样的复杂巨系统而言尤其如此。本节主要讲解调试软件GDB的使用方法。

\subsection{认识GDB}

GNU调试器（英语：GNU Debugger，缩写：GDB），是GNU软件系统中的标准调试器，此外GDB也是个具有移携性的调试器，经过移携需求的调修与重新编译，如今许多的类UNIX操作系统上都可以使用GDB，而现有GDB所能支持调试的编程语言有C、C++、Pascal以及FORTRAN。

\textbf{为什么需要GDB}

当程序出现错误时，开发者需要快速地找出错误的原因，并修复它们。很多人会倾向于使用"人工静态分析"(也就是目测法和冥想法)解决Bug。

然而，程序的运行时状态往往非常复杂，有时很难在代码中准确地定位错误。 具体来说, 目测的以下缺点导致开发者往往会百思不得其解进而无功而返:

\textbf{1.无法检测运行时问题：}代码静态分析只能检测静态代码问题，例如语法错误、类型错误等，它无法检测代码的运行时问题。例如，它无法检测到由于代码在特定环境下执行而引起的问题，如内存泄漏、死锁等。

\textbf{2.误报和漏报：}静态分析可能会误会遗漏问题。例如，可能会将某些无害的代码标记为错误，或者忽略某些实际上是错误的代码。这可能会导致开发人员浪费时间和精力来调查错误的根本原因。

\textbf{3.对高质量代码的依赖：}代码静态分析需要高质量的代码才能进行准确的分析。如果代码质量不好，例如缺乏注释、变量名不规范、代码冗余等，那么静态分析可能会产生误报或漏报。

\textbf{4.难以发现复杂的问题：}静态分析工具通常使用各种分析技术来分析代码，但这些技术很难发现复杂的问题，例如多线程问题、分布式系统问题等。这些问题通常需要动态调试或其他更高级的技术来解决。

同样, 也有人会尝试插入LOG打印部分状态, 但是这种方法除了费时费力, 且暴露状态不够精确的问题之外, 在OS中, 某些LOG会产生系统状态的改变, 进而影响结果, 导致debug失败。

这时候，调试工具就显得非常重要了。调试工具可以帮助开发者在运行时监视程序的状态，跟踪代码的执行流程，查看变量的值，以及定位错误的位置。 这正是GDB的用途。

GDB 最初由Richard Stallman在他的GNU Emacs 系统稳定后于1986年编写，并设计作为他的GNU系统的一部分。GDB是根据GNU通用公共许可证(GPL)发布的免费软件。它是在Berkeley Unix发行版附带的DBX调试器之后建模的。从1990 年到 1993 年，它由John Gilmore维护。现在由自由软件基金会任命的 GDB 指导委员会维护。

GDB允许用户查看一个程序在执行时“内部”的执行过程—，或者查看程序在崩溃时的内部状态。这些被调试的程序可以与 GDB 在同一台机器上（本地）、另一台机器（远程）或模拟器上执行。GDB 可以在大多数流行的 UNIX 和 Microsoft Windows 变体以及 Mac OS X 上运行。具体而言，目前 GDB 支持以下程序：Ada、Assembly、C、C++、D、Fortran、Go、Objective-C、OpenCL、Modula-2、Pascal、Rust 等。

GDB 默认只有命令行接口（CLI）可用，而不具备较能亲合上手、直觉操作的图形用户界面（GUI），不过此一弱处也已经有几个前端程序为其补强，例如DDD、GDBtk／Insight （页面存档备份，存于互联网档案馆）以及Emacs中的“GUD 模式”等，有了这些补强后，GDB在功效使用的便利性上就能够与“集成发展环境中的调试功效使用”相接近。

\textbf{GDB的启动}

显然, 启动gdb有不同的方法, 在终端中输入gdb是最简单的, 但NPUcore在RISC-V上构建, 因此不能直接使用本机的gdb(除非你使用一台RISC-V64计算机),因此我们推荐安装并使用gdb-multiarch(这里需要Ubuntu环境):

\begin{lstlisting}[language={Rust}, label={code:forktest},

caption={forktest.rs}]

$ sudo apt-get install gdb-multiarch

# 然后启动：

$ gdb-multiarch

\end{lstlisting}

注意, 这里实际上有"工作目录"的概念, 也就是你的当前目录实际上最好在项目或者源代码的路径上， 否则会需要手工加载源代码路径（方法见下文）。

\subsection{基于GDB的内核调试}

\textbf{QEMU虚拟机的相关命令}

介绍GDB为什么要先介绍虚拟机呢？因为正是QEMU与GDB合作， 才给了我们方便地进行大部分系统软件调试的机会。

作为一款全虚拟化虚拟机, QEMU能彻底模拟CPU的内部状态, 包括寄存器和其他部分, 因此很适合进行调试。

具体来说, QEMU配合GDB提供了单步执行、断点调试、内存监视、寄存器查看等。

用户可以使用调试功能逐步执行代码，查看每一步的运行结果和寄存器状态，同时还可以设置断点，方便定位问题所在。

利用QEMU提供的远程调试功能，允许用户在另一台计算机上通过网络连接到QEMU的调试接口进行调试。这个功能可以方便地在不同的计算机之间进行协作开发和调试。

这里我们只介绍本地的远程调试。

为了方便, 在os文件夹中, 使用下列make命令直接进行gdb调试:

\begin{lstlisting}[language={Rust}, label={code:forktest},

caption={forktest.rs}]

make gdb

\end{lstlisting}

  其后端执行实际命令是:

\begin{lstlisting}[language={Rust}, label={code:forktest},

caption={forktest.rs}]

gdb:

@qemu-system-riscv64 -machine virt -nographic -bios $(BOOTLOADER) -device loader,\

file=target/riscv64gc-unknown-none-elf/debug/os,addr=0x80200000 -drive \

file=$(U\_FAT32),if=none,format=raw,id=x0 \

-device virtio-blk-device,drive=x0,bus=virtio-mmio-bus.0 -smp threads=$(CORE\_NUM) -S -s

\end{lstlisting}

  和do-run的内容相比,

\begin{lstlisting}[language={Rust}, label={code:forktest},

caption={forktest.rs}]

do-run:

@qemu-system-riscv64 \

-machine virt \

-nographic \

-bios $(BOOTLOADER) \

-device loader,file=$(KERNEL\_BIN),addr=$(KERNEL\_ENTRY\_PA) \

-drive file=$(U\_FAT32),if=none,format=raw,id=x0 \

-device virtio-blk-device,drive=x0,bus=virtio-mmio-bus.0\

-smp threads=$(CORE\_NUM)

\end{lstlisting}

不难发现最主要的差别在于后面多出的"-S -s"。 前面的S代表STOP, 意思是设置完虚拟机直接挂起, 停止一切执行, 直到接收到外部的continue信息为止.

第二个小写s等价于"-gdb tcp::1234",指的是开启远程调试,其在localhost(本机)的1234端口侦听GDB的信号, 等待连接。 然后, 就是我们的下一个工具GDB的任务和工作范畴了。

\subsection{git bisect——快速问题定位}

  大家一定听过二分查找的算法, 如果我们发现某个Bug出现, 其实也可以通过二分查找定位到出错的版本。

git也自带了这个功能。使用git bisect, 我们可以在Git版本控制系统中进行二分查找，在版本历史中快速定位错误引入的位置。

\textbf{一般步骤}

  我们以这些图文为例给出一个错误的处理

\begin{lstlisting}[language={Rust}, label={code:forktest},

  caption={forktest1.rs}]

  $ git bisect start

  //运行 git bisect start 命令来启动一个二分查找会话。

  $ git bisect bad

  //用 git bisect bad 命令告诉 Git 当前版本存在问题。

  $ git bisect good HEAD~10

  //用 git bisect good 命令告诉 Git 一个知道没有问题的提交， 是这个提交的哈希值或分支名。git会给出估计的剩余步骤数

  Bisecting: 4 revisions left to test after this (roughly 2 steps)

  [f55d253527e3a72f730b06e6fbfe5e64f8594a27] fix: Change ELF related AuxV data alignment to repr(C), fixing the LPF.

  //Git 会自动切换到一个介于上述两个提交之间的提交，你需要在该提交上运行你的程序，检查问题是否存在。

  如果有问题，使用 git bisect bad 命令告诉 Git，否则使用 git bisect good 命令告诉 Git。

  你可以使用Git bisect run 切换提交后自动执行的命令(注意, git bisect run后仍然在原地, 这时候需要git bisect next才能进入下一个, 否则会冲突)

  Git 会根据你的反馈自动切换到下一个介于两个提交之间的提交，重复上述步骤，直到找到引入问题的提交。

  最后，使用 git bisect reset 命令退出二分查找会话。

\end{lstlisting}

\begin{figure}[htb]

  \centering

  \includegraphics[width=\textwidth]{figures/02-02-运行实例.png}

  \caption{

  运行实例

  }

  \label{fig:运行实例}

\end{figure}

 \textbf{log与冲突回撤}

   如果发现某个提交被错误标记(比如bad被错误标记为good), 尝试

 \begin{lstlisting}[language={Rust}, label={code:forktest},

  caption={forktest2.rs}]

  git bisect bad

  你会得到以下错误:

  ba246b7c5294eeabcdca705fafe8ea1015e6fd6e was both good and bad

  $ git bisect log//导出日志:

  git bisect start

  # good: [ba246b7c5294eeabcdca705fafe8ea1015e6fd6e] add: Expect script and clauses in Makefile for automation

  git bisect good ba246b7c5294eeabcdca705fafe8ea1015e6fd6e

  # bad: [ba246b7c5294eeabcdca705fafe8ea1015e6fd6e] add: Expect script and clauses in Makefile for automation

  git bisect bad ba246b7c5294eeabcdca705fafe8ea1015e6fd6e

  //重定向到文件

  $ git bisect log > bis.log

  //考虑修改错误

  $ git bisect start

  会得到以下信息：

  # bad: [ba246b7c5294eeabcdca705fafe8ea1015e6fd6e] add: Expect script and clauses in Makefile for automation

  $ git bisect bad ba246b7c5294eeabcdca705fafe8ea1015e6fd6e

  $ git bisect replay bis.log

  //回复到之前的状态

 \end{lstlisting}