拆弹实验：命令行工具教程

本教程会对拆弹实验中所需使用的一些命令行工具进行介绍，包括为什么要使用这些工具、宏观功能介绍以及部分工具的详细介绍，对于感兴趣的同学，我们也列出了一些更详细的资料供进一步学习。本教程中介绍的工具仅代表我们推荐的，在拆弹和后续ChCore实验中的常用命令行工具，但我们不限定同学们在完成实验时所使用的具体工具或方式，同学们可以使用任何其他自己熟悉的工具来完成实验。

本教程中所介绍的工具和命令，均假设在Linux shell环境（如bash等）下执行。如果同学们希望使用其他操作系统或平台，请自行查阅相关工具的安装方法以及可能的命令语法不同之处。

TL;DR

- `<参数>`代表需要被替换的参数，请将其替换为你需要的实际值（包括左右两侧尖括号）。
- tmux操作
  - 创建新会话：`tmux new -s <会话名>`
  - 进入会话：`tmux attach -t <会话名>`
  - 临时退出会话（会话中程序保持在后台继续运行）：`Ctrl-b d`
  - 关闭会话及其中所有程序：`tmux kill-session -t <会话名>`
  - 水平分屏：`Ctrl-b "`
  - 垂直分屏：`Ctrl-b %`
- gdb操作
  - 不输入任何命令，直接按下回车键：重复上一条命令
  - `break *address`：在地址`address`处打断点
  - `break <函数名>`：在函数入口处打断点
  - `continue`：触发断点后恢复执行
  - `info breakpoints`：列出所有断点
  - `delete <NUM>`：删除编号为`NUM`的断点
  - `stepi`：触发断点后单步执行一条指令
  - `print <expr>`：打印表达式`<expr>`的求值结果，可以使用部分C语法，例如`print *(int *)0x1234`可将地址`0x1234`开始存储的4个字节按32位有符号整数解释输出
  - `print/x <expr>`：以16进制打印`<expr>`的求值结果
  - `lay asm`：使用TUI汇编视图
  - `lay src`：使用TUI源代码视图（要求被调试可执行文件包含调试信息，且本地具有相应源代码文件）
  - `tui disable`：退出TUI
- objdump操作
  - `objdump -dS <可执行文件> > <输出文件>`：反汇编可执行文件中的可执行section（不含数据sections），并保存到输出文件中。在可能的情况下（如有调试信息和源文件），还会输出汇编指令所对应的源代码。
  - `objdump -dsS <可执行文件> > <输出文件>`：将可执行文件中的所有sections的内容全部导出，但仍然只反汇编可执行sections，且在可能情况下输出源代码。

tmux

tmux是什么

如今大部分同学开始使用电脑接触的就是Windows与GUI（图形用户界面）。在GUI下，我们可以通过桌面和窗口的方式，管理显示器上的二维空间（桌面空间）。许多情况下，桌面空间对于单个应用程序来说，有些太大了，只使用一个应用程序不能有效地利用显示空间；又或者，我们想要同时处理多个任务，比如在写文档或者论文时，希望能非常方便地看到自己的参考资料……部分同学可能已经掌握如何使用分屏以及虚拟桌面来解决这些问题。的确，不论Windows10/11还是macOS抑或是Linux下的桌面环境，甚至众多基于Android的操作系统，分屏与虚拟桌面已经成为构建高效的GUI工作环境的基本功能。

分屏解决的是如何有效利用桌面空间的问题。通过分屏功能的辅助，我们可以快速将多个应用程序的窗口紧密排列在桌面空间上，且可以更方便地调整这些窗口所占据的区域，避免自己手动排列带来的混乱和不便，让我们可以同时使用多个相关的应用程序，最大化利用桌面空间与提高效率。

虚拟桌面则解决的是如何有效隔离多种使用场景的问题，它是分屏功能的进一步衍生。我们经常需要同时处理几种不同性质的任务，例如，一边在编写文档或论文，一边还可能需要看QQ、微信和回复消息。如果只有一个桌面空间，要么不时切出和最小化QQ微信的窗口，要么把它们也使用分屏与文档窗口排列在一起，但这两种使用方式，或多或少都会影响需要专注的文档编写任务。此时，我们可以使用虚拟桌面。虚拟桌面是在物理桌面空间上“虚拟”出多个互不相干的桌面空间，每个桌面空间内都可以有自己的窗口布局。虽然同时只能使用一个虚拟桌面，但我们可以在多个虚拟桌面间快速切换。使用虚拟桌面后，我们可以将比较相关的一类程序的窗口放在同一个虚拟桌面中，其余不相干的程序则放在其他虚拟桌面中，如此，可以有效减少其他程序对于当前工作任务的干扰，同时又能在多种不同工作环境中快速切换。

分屏与虚拟桌面有效提高了GUI下的窗口管理效率。但是，窗口和桌面的概念，并非只能局限于GUI中。利用除了字母数字外的各种字符和颜色，我们同样可以在命令行用户界面（CLI）下“绘制”窗口，相较于通过命令行参数，窗口这种交互方式对用户更友好，更直观。同样地，在CLI下的窗口中，分屏和虚拟桌面需要解决的这些效率问题同样是存在的，也一样有着解决这些问题的需求。tmux（terminal multiplexer）项目则是目前在CLI环境下这些问题的主要解决方案。顾名思义，它是一个“终端多路复用器”，如果说分屏和虚拟桌面是有效利用GUI中的桌面空间，tmux则主要是有效利用终端中的空间。这里的终端，可以是GUI下的终端模拟器，比如Windows Terminal，iTerm2等，也可以是运行在命令行模式下的Linux的显示器空间等等。

tmux vs 多个终端模拟器窗口

- 同一窗口内部布局自由构建（部分终端模拟器也可实现）
- 统一管理多个窗口、便捷切换（开多个终端容易混淆，不便于随意切换）
- tmux可以在不使用时将进程保持在后台继续运行（detach，而终端模拟器一旦关闭就会杀死其中所有进程）
- tmux还是一个服务器，可以通过网络连接，如果有需要，可以允许其他人通过网络连接到你的tmux界面中，实现网络协作（终端模拟器不支持）
- tmux支持高度自定义的配置，且有丰富的插件生态

session/window/pane/

在理解了GUI下为什么需要有分屏和虚拟桌面后，类比GUI下的概念，可以很容易地理解tmux中的相关概念。

- Pane

  pane相当于GUI下的一个窗口。只不过相较于GUI下窗口可以自行自由移动，也可以使用分屏辅助排列，CLI下窗口还是基于字符的，所以tmux下的pane只能实现类似于GUI分屏的紧密排列，不能自由移动，也不能实现pane之间的重叠。

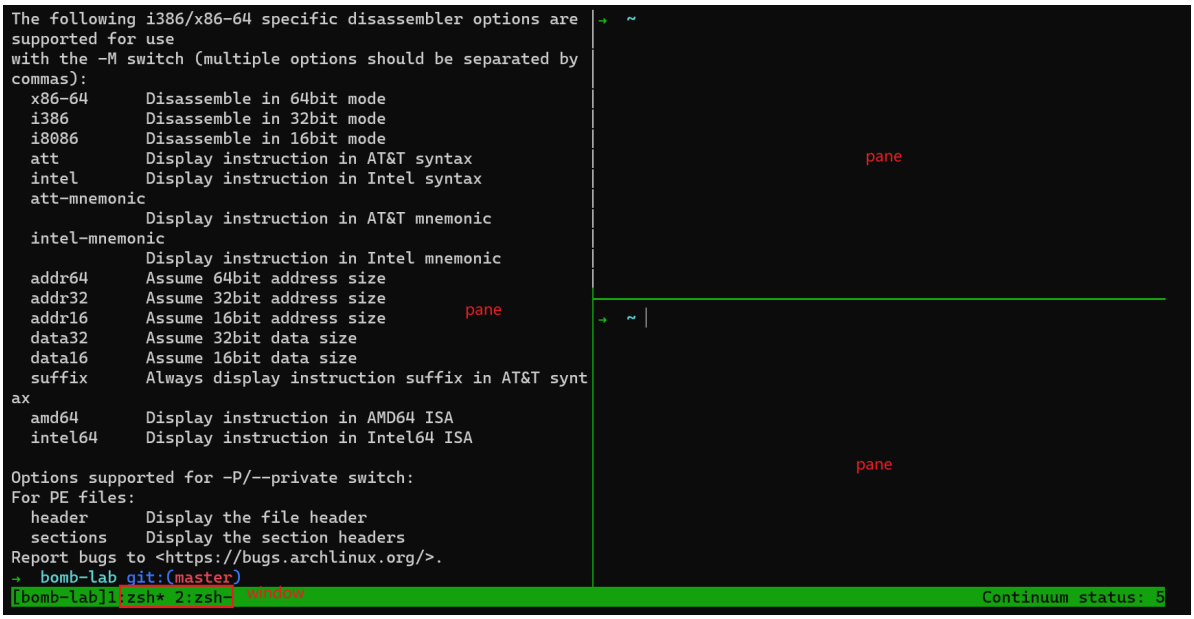
- Window

window则相当于GUI下的虚拟桌面。一个window是一组pane的集合，不同的window拥有独立的pane以及pane的布局，且可以在多个window间通过快捷键快速切换。

- Session

  session是tmux特有的概念，它是一组window的集合，代表一个完整的工作环境。一般来说，不论我们通过显示器、Linux桌面环境下的终端模拟器、Windows或macOS上的终端模拟器+ssh……等方式访问Linux命令行，首先都是进入一个shell中，而并不能直接进入tmux。因此，在tmux中，相较于“打开”和“关闭” tmux，我们更常说"attach","detach"到session。attach指的是从当前运行的shell进入一个tmux session的过程，而detach则是从tmux session离开，回到单个shell的过程。相较于“打开”和“关闭”，session+attach/detach有下列好处：

  - 在同一个终端中，也可以方便切换不同工作环境（这是最基础的功能）
  - 避免同一个session中有过多不相干的window，降低切换效率
  - detach不是关闭session，detach后，session中运行的所有程序会在后台继续运行，且tmux会负责收集它们产生的输出，我们随时可以重新attach到某一个session，就可以看到其中程序最新的运行情况和历史输出，这对于某些需要长时间后台运行的任务是非常方便的。
  - 可以有多个终端同时attach到一个session，搭配tmux的服务器功能，可以实现向他人共享你的工作界面和环境，以及协作工作。



常用快捷键与子命令

tmux支持丰富的快捷键，同时，也可以通过tmux命令的一组子命令来与tmux进行交互。事实上，如果研究tmux的配置文件，可以看到其中重要的一项内容就是将快捷键绑定到相应的子命令上。

- 前缀(prefix table)快捷键与根(root table)快捷键
  - 许多CLI应用程序都会定义自己的快捷键。作为更底层的程序，tmux需要尽可能避免自己的快捷键和上层应用程序的快捷键冲突。前缀键就是为了解决这个问题引入的。
  - 前缀快捷键：需要先按下前缀键（默认是`Ctrl+b`），然后再按下相应的快捷键。
  - Root快捷键：直接按下对应的快捷键，不需要前缀键。这些通常是一些不与通常使用的终端快捷键冲突的键。
- 命令行模式
  - `<prefix>` `:`，可以进入tmux的内部命令行。在该命令行中可以直接输入tmux子命令（而不是`tmux <子命令>`），可以用于使用未绑定快捷键的命令。
- pane管理
  - 切换pane：`<prefix>`方向键，tmux命令版本是`select-pane -[UDLR]`，U代表上，D代表下，L代表左，R代表右。同时只能有一个pane接收输入，利用这个命令可以切换当前接收输入的pane。

    - 分割当前pane，创建两个新的pane：`<prefix>` `"` (上下分割)，`<prefix>` `%` (左右分割)，tmux命令版本是`split-window` (默认是上下分割)，`split-window -h` (左右分割)。在初始情况下，每个window只有一个占满整个window的pane，使用这个命令可以分割当前接收输入的pane，创建两个pane，类似于GUI下的分屏功能。
    - 关闭pane：`<prefix>` `x`，然后按`y`确认。tmux命令版本是`kill-pane`。
- window管理
    - 创建Window：`<prefix>` `c`，tmux命令版本是`new-window`。
    - 切换Window：`<prefix>` `n` (下一个Window)，`<prefix>` `p` (上一个Window)，`<prefix>` `l` (最后一个活跃Window)，tmux命令版本是`next-window`，`previous-window`，`last-window`。
    - 切换到指定Window：`<prefix>` `窗口序号`，tmux命令版本是`select-window -t :窗口序号`。
- session管理（部分命令没有快捷键，因为不是在session内部进行操作）
    - 创建新session：`tmux new-session -s mysession`，`mysession`是新session的名字。
    - 列出所有session：`tmux list-sessions`。
    - attach到一个session：`tmux attach -t mysession`，`mysession`是目标session的名字。
    - detach当前session：`<prefix>` `d`，tmux命令版本是`detach`。
    - 终止某个session：`tmux kill-session -t mysession`，`mysession`是你想要终止的session的名字。
- 在某些GUI终端模拟器中，还可以通过鼠标与tmux交互，例如可以通过鼠标拖拽pane的边界来调整各个pane的大小，点击window的名字来切换到指定window等。

扩展阅读

- http://man.openbsd.org/OpenBSD-current/man1/tmux.1
- https://github.com/rothgar/awesome-tmux
- https://github.com/gpakosz/.tmux
- 如果你希望通过tmux提高自己的工作效率，我们强烈建议你编写适合自己使用习惯的tmux配置文件。

gdb

什么是gdb？为什么要使用gdb?

gdb是目前最常用的**动态调试**工具之一。所谓动态调试，指的是在程序运行的过程中对程序进行观测或施加干预的过程，一种常见的动态调试方法是断点，通过插入断点使得程序在特定点位暂停运行，让动态调试器可以对程序的状态进行进一步的观测。与动态调试相对的是**静态分析**。静态分析不需要实际运行程序，甚至不需要编译程序，而是在程序源代码或指令的层级进行一系列分析甚至“枚举”，预测程序的可能执行情况，寻找潜在的问题。某种程度上来说，对于存在问题的程序，程序员直接阅读代码或汇编指令并分析问题，也可以被视为一种静态分析。

相较于静态分析，动态调试可以真实地反映程序运行的实际情况，包括各种数据的实际值、程序的实际执行路径等。在某些场景下，使用动态调试寻找程序的问题或理解程序的行为，比直接阅读程序源码要简单许多，例如使用动态调试器单步运行程序，在每一步运行的前后观察程序的相关状态，可以非常直观地找到导致问题的程序指令或代码。当然，静态分析也有着广泛的应用，许多静态分析工具在无需编译或运行程序的情况下便可分析程序的潜在问题，这可以有效节约程序运行的时间，同时静态分析可以尽可能地枚举程序的可能执行路径，有助于发现实际运行程序时不会出现或非常罕见的问题。

源码级调试 vs 汇编级调试

动态调试是在程序的运行过程中施加干预和观测状态，问题在于，如何干预程序的运行？又该观测程序的什么状态？以打一个断点为例，应该在什么地方打断点，程序触发断点后，又该检查程序的哪些状态？动态调试器本身只是为程序员提供了完成上述工作的能力，但如何运用这些能力，仍然需要程序员本身对于程序的了解。站在程序员的角度，无疑希望能直接在程序执行到某一行源代码时触发断点，触发断点后，又可以直接检查程序中某个变量的值甚至复杂对象的内容。许多同学此前可能已经接触过各种IDE自带的调试功能，它们大多都允许程序员在源代码中设置断点，并且可以在触发断点时直接看到各个变量和对象的内容。这种程序员直接站在源代码的层级，使用源代码级的概念（代码行、变量、对象等）进行调试的过程称为源码级调试。

然而，从动态调试器的功能而言，要支持源码级调试并非仅有动态调试器即可做到。这是因为，CPU本身只能执行二进制形式的机器指令，不论是编译执行或是解释执行的高级编程语言，最终在程序运行时，动态调试器能观察和控制的只是最终的机器指令、寄存器和内存地址。例如，仅使用动态调试器本身，我们只能指定在某一条指令暂停执行，也不能直接检查变量或者对象的内容，因为在机器指令的层面并没有变量和对象的概念，只有寄存器和内存。这种只使用机器执行过程中直接可见的概念（指令、寄存器、内存）进行调试的过程称为汇编级调试/机器级调试。造成上述问题的原因是，在从源代码到可执行程序的编译或解释过程中，许多信息都丢失了，因为这些信息对于程序的最终执行并无任何帮助：从程序执行的角度来说，CPU不需要理解某条指令对应源代码中的哪个文件的哪一行，也不需要理解某个寄存器在某一时刻存储的是哪个变量的值。但是，这些信息的丢失，就给调试带来了较大的困难，因为高级语言翻译成汇编指令的方式非常多样，并且存在各种复杂的细节，从而使得汇编级调试并不直观，往往需要远多于源码级调试的时间精力才有可能定位和理解程序存在的问题。

如果想要进行源码级调试，就需要在程序可执行文件中加入一系列的额外信息，来弥补编译/解释过程中损失的信息，让动态调试器可以把指令地址、寄存器、内存地址“还原”为源代码级的概念如源代码行、变量、对象等等。但是，嵌入这些信息会使得程序可执行文件的体积增大，所以如果不是在编译时使用特定的选项，程序往往是没有这些额外信息的。以Linux下最常见的可执行文件格式ELF为例，若要支持源码级调试，需要ELF文件存在符号表和专门的调试信息。其中符号表可以用于将一些内存地址还原回函数或全局变量等，除了调试之外，还有许多其他用途，而调试信息则是专门为了将机器级概念还原到源代码级存在的。如果一个ELF文件只有符号表，没有调试信息，那么绝大多数源码级调试功能也都是不可用的，但是也可以支持一定程度的源码级调试，例如在函数的入口打断点，检查全局变量的值等等。

对于解释型语言，情况还要更复杂一些。这是因为，gdb等动态调试器，都是把程序视为一个“黑盒”，它们并不理解一个程序是在完成自身的工作，还是在作为解释器，为一种更高级的语言（如Python）提供支持。以Python为例，即使Python解释器程序本身有包含调试信息，从gdb的角度来看，也只能看到解释器本身的工作情况，例如它是如何解析Python字节码的，这个过程中它调用了自身的哪些函数，修改了自己内部的哪些变量等等。但是从Python程序员的角度来说，往往假设Python解释器本身是正确的，问题在于自己编写的Python代码，比起理解解释器内部的执行情况，更关注的是Python语言层级的概念。但gdb等通用动态调试器是无法在Python语言层级进行调试的。对于使用解释执行的语言，需要解释器本身支持调试功能，往往还需要使用专门的调试器。

在本次实验中，我们提供的炸弹程序是使用非常短的C代码编译而成的，且没有使用过高的优化等级，汇编指令与原始C代码是高度对应的；程序保留了符号表，但移除了调试信息。这是因为在操作系统中不可避免地存在无法使用高级语言，必须使用汇编语言编写的部分。因此我们希望通过一个复杂度有限的汇编程序，提高同学们对于汇编语言以及C语言编译到汇编语言过程的理解，同时增强同学们对gdb的熟悉程度和调试能力，为后续的实验打下基础。

gdb使用简介

- 设置被调试目标
    - gdb支持调试本地运行的进程，也支持通过网络等方式远程调试其他机器上运行的进程。在Linux机器上调试本地进程时，gdb依赖ptrace这个syscall，它从操作系统层面为gdb控制其他进程的运行提供了基础支持。一般而言，出于安全性考虑，各个Linux发行版都对ptrace syscall的调用进行了程度不等的权限控制。例如只允许通过ptrace调试子进程等。

- 启动为子进程：`gdb <program>`
  - 其中，`<program>`为需要被调试的程序名，可以是一个完整的可执行文件路径，也可以只提供程序自身的可执行文件的名称。对于后一种情况，gdb会搜索$PATH环境变量来找到可执行文件的实际路径。
  - 执行该命令后，会进入**gdb命令行**。此时，gdb并不会立即开始执行被调试的程序。但此时gdb已经载入了可执行文件中的符号表和调试信息（如有），可以在实际执行程序前就设置一些断点等，以便调试程序运行早期的代码或不接收输入的程序等等。
  - 确认完成准备工作后，在gdb命令行中执行`run`命令，作为gdb的子进程运行被调试程序。
- attach到运行中进程
  - 一般而言，在常见Linux发行版上，直接attach到运行中进程可能需要root权限。
  - `sudo gdb -p <pid>`
- 远程调试
  - gdb支持通过网络、串口等调试其他机器上运行的程序。这里我们以网络远程调试为例说明远程调试的基本原理。如果想使用网络远程调试，需要在实际运行被调试程序的机器上启动一个gdbserver，或任何实现了gdbserver协议的代理程序（以下统称为gdbserver）。gdb将会连接到这个gdbserver，并通过网络向gdbserver发送命令，以及通过gdbserver，读取运行中程序的信息等。在这个架构下，实际控制程序运行的是gdbserver，而不是gdb，但gdbserver又受到gdb的控制。远程调试的主要优点是提供了一种通用的将调试和程序的运行解耦的方法，而不是必须将被调试程序作为gdb的子进程运行。例如，远程调试可以允许程序在其他机器上运行，这对于一些必须在内网中运行或是依赖特殊硬件等的程序很有用；此外，远程调试基于gdbserver协议，任何程序只要实现了该协议，都可以"表现为"一个gdbserver，如此，程序可以主动将自己的一些内部信息暴露给gdb。理论上而言，这可以用于实现Python、Go等由虚拟机运行的程序的调试（虽然实际上这些语言的调试不是这么实现的），也可以用于调试由qemu运行的程序（详见下文）。
  - 使用`target remote <ip|domain name>:<port>`，即可将gdb的调试目标设为远程的gdbserver。
  - 如果通过gdb进行远程调试，在运行gdb时是否还需要指定`<program>`？事实上，答案并非是完全不需要。如果只想进行纯粹的汇编级调试，的确可以不需要指定`<program>`，但如果想要进行源码级调试，则仍需要指定`<program>`。在这种情况下，`<program>`的主要作用是让gdb读取其中的符号表和调试信息。gdb不会直接从gdbserver中读取这些信息，因为gdbserver不一定能提供这些信息，gdb只会和gdbserver交换汇编级的信息，它依赖本地`<program>`文件中的符号表和调试信息将源码级信息翻译回汇编级信息。此外，还需要注意，在触发一个断点后，如果希望看到当前所执行到的程序源代码，则还需要本地保存了程序的源代码。这是因为，调试信息（以ELF所使用的DWARF格式为例）只保存了某条指令所对应的源代码的文件路径与行数，并没有直接保存源代码，gdb所做的工作是根据调试信息中的路径和行数，读取本地的代码文件并显示相应源代码。如果本地相应路径没有源代码文件或内容有误，那么gdb将无法正确显示源代码。
  - 如果运行gdb时没有指定`<program>`，还可以通过`add-symbol-file`命令指定`<program>`。如果程序不是在本地编译的，那么源代码的绝对路径可能与本地保存源代码的路径不同，可以用`set-substitute-path`命令进行替换。

以下命令均在**gdb命令行**中执行。

- 断点控制
  - `break <expr>`: 在`<expr>`处创建一个断点。`<expr>`是一个最终可以求值为某条指令的地址的表达式。例如，如果想直接在某个地址处设置断点，可以写成`*<address>`，如果被调试程序有符号表，可以直接使用函数名。如果被调试程序有调试信息，可以指定在某个源文件的某一行设置断点，甚至可以使用更复杂的C表达式，更详细的信息可以参考手册。但需要注意，

不论使用何种表达式，最终其实都是求值到一条指令的地址，gdb只是利用调试信息并帮助程序员简化了这一步骤。
    - `info breakpoints`: 列出当前的所有断点
    - `disable/enable <NUM>`: 禁用/启用编号为`<NUM>`的断点，断点编号可以通过前述`info`命令查看。
    - `delete <NUM>`: 删除编号为`<NUM>`的断点。
- 执行控制
    - 不输入任何命令，直接按下回车键：重复上一条命令
    - 在触发断点后，可以通过下列命令控制如何恢复程序的执行
    - `continue`: 恢复程序执行，直到触发下一个断点
    - `continue <NUM>`: 恢复程序执行，且忽略此断点`<NUM>`次
    - `kill`: 终止程序执行
    - `quit`: 退出gdb
    - `stepi`: 执行下一条机器指令，随后继续暂停执行（单步调试）
    - `stepi <NUM>`: 执行接下来`<NUM>`条指令
    - `step`: 执行下一条语句，这属于源代码级调试，需要调试信息
    - `nexti`: 执行下一条指令，且不跟踪（step through）函数调用。与`stepi`不同，如果当前指令是一条`bl`等指令，那么`stepi`会在被调用函数的第一条指令暂停（step in），而`nexti`会在`bl`指令之后的那条指令，即被调用函数返回后的那条指令上暂停。
    - `nexti <NUM>`，`next`: 与`stepi`，`step`类似，只是不跟踪函数调用。
    - `finish`: 恢复执行直到当前被调用的函数返回
- 显示信息
    - `backtrace`: 显示栈跟踪信息，可以看到当前函数是如何一步步被调用到的。但需要注意，由于编译器的优化等因素，如果没有调试信息，栈跟踪信息可能是不准确的，甚至无法提供栈跟踪信息。
    - `print<mods> <expr>`: 打印表达式`<expr>`执行的值。其中`<expr>`可以是寄存器，如`$pc`，`$sp`，`$x0`，也可以是C表达式等。如果没有调试信息，则所使用的C表达式不能涉及到程序中变量的值，但仍可以使用常量或寄存器中的值，例如，可以直接将某个内存地址作为数字打印：`print *(int *)0x1234`，或使用寄存器中的值参与运算等：`print $x0+$x1`。`<mods>`是可选的修饰符，可以用来控制打印的格式，详细信息可以参考手册。表达式求值时，寄存器、变量、内存中的值都基于触发断点时程序的状态。
    - `x/NFU <address>`: 打印内存中指定长度的内容。与`print`命令不同，该命令只能打印内存中的内容，且不需要如print一样将内存地址转换为指针并求值。这是因为此命令只是将内存视为字节数组，打印出其中给定数目的字节的内容，而不对字节内容做任何解释。相反，`print`在打印指针求值表达式时，会根据指针的类型信息和`<mods>`来计算需要打印多少字节，且会对多字节的内容进行解释，例如将它们显示为一个完整的整数，而非整数在内存中的各个字节。此命令的修饰符中，`N`表示需要打印的单元数目，`U`的取值可以为`b`,`h`,`w`,`g`，分别表示一个单元大小为1，2，4，8字节。`F`表示每个单元的打印格式，如`i`表示作为指令反汇编，`x`表示十六进制，`d`表示十进制等，详细信息可以查看手册。
    - `display<mods> <expr>`: 在每次程序暂停执行时，自动根据`<mods>`打印`<expr>`的值，适用于一些频繁更改的表达式。利用这条命令可以在单步调试时自动查看所需关注的值，而无需反复输入print命令。
    - `info display`: 列出所有的自动打印。
    - `delete display <NUM>`: 删除编号为`<NUM>`的自动打印。
- TUI
    - TUI(text UI)是gdb内置的高级命令行界面功能。TUI在gdb内部引入了类似窗口的概念，允许用户在使用gdb命令的同时查看被调试程序的汇编指令、源代码等，也允许用户设置自定义的UI布局，提高工作效率。

- `lay asm`: 进入TUI默认汇编布局，此布局下会同时开启一个gdb命令窗口以及一个汇编指令窗口，用户可以直观看到当前正在执行的汇编指令。
- `lay src`: 进入TUI默认源代码布局，此布局下会同时开启gdb命令窗口和源代码窗口。需要具备调试信息和源代码文件。
- `tui disable`: 退出TUI。

扩展阅读

- ptrace syscall: https://man7.org/linux/man-pages/man2/ptrace.2.html
- gdb手册（如想深入了解gdb或参考命令的详细用法，非常建议阅读，此文档只是给出了一个非常简要的介绍）：https://sourceware.org/gdb/onlinedocs/gdb/index.html#SEC_Contents
- TUI: https://sourceware.org/gdb/onlinedocs/gdb/TUI.html

objdump

objdump能做什么?

objdump是GNU binutils中的标准工具之一。它的主要作用是显示**二进制**程序的相关信息，包括可执行文件、静态库、动态库等各种二进制文件。因此，它在逆向工程和操作系统开发等底层领域中非常常用，因为在这些领域中，我们所接触到的程序很可能不提供源代码，只有二进制可执行文件；抑或是所面临的问题在高级语言的抽象中是不存在的，只有深入到机器指令的层次才能定位和解决。在这些领域中，objdump最常见的应用是**反汇编**，即将可执行文件中的二进制指令，根据目标架构的指令编码规则，还原成文本形式的汇编指令，以供用户从汇编层面分析和理解程序。例如，在拆弹实验中，我们只提供了炸弹程序的极少一部分源代码，同学们需要通过objdump等工具进行逆向工程，从汇编层级理解炸弹程序的行为，进而完成实验。

GNU实现 vs LLVM实现

objdump最初作为GNU binutils的一部分发布的。由于它直接处理二进制指令，因此它的功能与CPU架构等因素强相关。在常见Linux发行版中，通过包管理安装的objdump均为GNU binutils提供的实现，且它一般被编译为用于处理当前CPU架构的二进制指令。例如，在x86_64 CPU上运行的Linux中安装objdump，它一般只能处理编译到x86_64架构的二进制文件。对于GNU binutils提供的实现，如果需要处理非当前CPU架构的二进制可执行文件，则需要额外安装为其他架构编译的objdump，例如，如果需要在x86_64 CPU上处理aarch64架构的二进制文件，一般需要使用`aarch64-linux-gnu-objdump`。

LLVM是另一个被广泛应用的模块化编译器/工具链项目集合。LLVM项目不仅提供了另一个非常常见的C/C++编译器`clang`，也提供了一组自己的工具链实现，对应GNU binutils中的相关工具。例如，LLVM也提供了自己的objdump实现，在许多Linux发行版上，你需要安装和使用`llvm-objdump`命令。LLVM所提供的工具链实现与GNU binutils中的相应工具是基本兼容的，这意味着它们的命令行参数/语法等是相同的，但LLVM工具链也提供了其他扩展功能。从用户的角度来说，LLVM工具链与GNU binutils的一个主要区别是LLVM工具链不需要为每个能处理的架构编译一个单独的版本，以`llvm-objdump`为例，不论处理x86_64还是aarch64架构中的二进制文件，均可以使用`llvm-objdump`命令（需要在编译`llvm-objdump`时开启相应的支持，从发行版包管理中安装的版本通常有包含）。

可执行文件格式

为了更好地加载和执行程序，操作系统在一定程度上需要了解程序的内部结构和一系列元数据。因此，一个二进制可执行文件并不仅仅是将所有指令和数据按顺序写到文件中即可，通常它需要以一种特定的可执行文件格式存储，使用的格式是由它将要运行在的操作系统决定的。例如，对于Unix/Linux操作系统，ELF格式是目前最主流的可执行文件格式。而Windows下主要的可执行文件格式是PE/COFF。除了反汇编之外，objdump还可以显示与可执行文件格式相关的众多信息，但需要用户对可执行文件格式自身的理解。详细分析可执行文件格式超出了本教程的范围，感兴趣的同学建议查阅与ELF相关的资料。在后续ChCore实验中，也会涉及部分关于ELF的知识。

objdump使用

本节只介绍拆弹实验中可能用到的基础用法。详细用法感兴趣的同学可以参考扩展阅读部分。

- `objdump -dS <可执行文件> > <输出文件>`: 反汇编可执行文件中的可执行section（不含数据sections），并保存到输出文件中。在可能的情况下（如有调试信息和源文件），还会输出汇编指令所对应的源代码。
- `objdump -dsS <可执行文件> > <输出文件>`: 将可执行文件中的所有sections的内容全部导出，但仍然只反汇编可执行sections，且在可能情况下输出源代码。

扩展阅读

- objdump手册：https://man7.org/linux/man-pages/man1/objdump.1.html
- https://en.wikipedia.org/wiki/Executable_and_Linkable_Format
- https://maskray.me/blog/
- https://fourstring.github.io/ya-elf-guide/guide_zh

make

什么是make？为什么要使用make？

make是Linux操作系统上常见的较为基础的构建系统（build system）。构建系统的主要工作，是管理一个较大规模或较为复杂的项目的各个“部件”，以及如何将这些“部件”“组装”成为最终的项目产物。以C语言项目的编译过程为例。一个最简单的C语言项目，只需要有一个源文件即可，这种简单的项目，确实对构建系统没有很强的需求，只需要一条很简短的gcc命令即可完成编译。但是，随着项目规模的增加，项目的源文件也会越来越多，诚然，我们可以把所有源文件的文件名都提交给gcc命令进行编译，但是这样会导致每次都要输入一个很长的命令。或许有同学会说，命令历史记录或者写一个简单的脚本都可以简化这个操作。但是构建系统能做的不仅如此。如果每次都把所有源文件提交给gcc让它进行编译，那么gcc每次都会执行一次**完整编译**，即重新将每一个源文件编译成对象文件，再将他们进行链接。然而，事实上，在一个规模较大的项目中，我们在更新这个项目时，通常不会修改项目的所有源文件，常常只有一小部分或者一部分源文件被修改了，那么那些没有被修改的源文件就完全没有必要再重新编译，只需要将已有的**中间产物**（对象文件）与修改过的文件重新编译后的对象文件进行链接即可，也即所谓的**部分编译**。换句话说，如果每次都使用前述的一条完整gcc命令，那么相当一部分编译时间就被浪费了，因为gcc自身并不理解哪些文件需要重新编译，哪些文件不需要重新编译，而且除非添加特定参数，gcc也不会保留中间产物或利用现有的中间产物。相比于在一条命令中把所有源文件都提交给gcc，更好的做法是把整个构建过程拆散，先逐个将源文件编译为对象文件，并且保存这些对象文件作为中间产物，随后再将这些中间产物链接到一起，产生完整的程序。当然，这么做相较于一条gcc命令会复杂很多，但对于大型项目可以显著减少编译时间。由于整个流程从一条命令变成了需要以特定顺序执行的许多条命令，因此出于可维护性的考虑，应当把整个流程以文件形式保存下来，这就是构建系统中常见的**规则文件**。

如前所述，一个比较复杂的项目，其编译流程涉及多条（甚至是大量）需要按照特定顺序执行的命令。如何确定这些命令的顺序？如何在项目的结构发生变更后重新确定这些命令执行的顺序？这些问题，是单纯的shell脚本所不能解决的，因此构建系统的规则文件通常并不是简单的shell脚本。一般而言，构建系统的规则文件都包含几个基本元素（不同系统中的术语可能不同），以make为例：目标（target）、配方（recipe）、前置条件（prerequisites）。目标指的是构建系统需要负责生成的一个文件或完成的一项任务，这个文件可以是某个中间产物文件，也可以是最终的产物或任何其他文件；此外，它也可以是一项抽象的任务，例如本实验中用到的使用qemu运行炸弹程序，也被作为make的一个目标。前置条件指的是在完成这个目标前需要完成的其他目标，一个目标只有在前置目标都已完成时才能被执行。最简单的前置目标是构建这个目标所需的源文件，例如用于生成对象文件app.o的目标，其前置目标可以是其源文件app.c。构建系统可以根据文件系统中app.o和app.c两个文件的修改时间，来判断是否需要执行这个目标，如果app.c在app.o产生之后又被修改过，则认为app.o需要重新生成，否则就可以直接使用现有的app.o。配方则是定义如何完成某一个特定的目标，通常包含一系列shell命令等。从上面的

描述中不难看到，规则文件的作用实际上是定义了一个**依赖图**，图中每个目标是一个顶点，如果目标A依赖目标B，则目标B有一条边指向目标A。从原理上来说，构建系统会根据规则文件构建出依赖图，随后依照图的拓扑序执行每个顶点所定义的配方，最终就能生成完整的产物。还需要注意，构建系统是独立于编译器之外的，构建系统能做的，是给程序员提供自动生成和执行依赖图的能力，但依赖图的结构本身仍然是规则文件的编写者定义的，例如，项目的构建涉及到多少目标，每个目标的配方使用什么命令，每个目标的前置条件是什么，这些问题是不可能由构建系统自身来解决的，而是要由规则文件的编写者"告知"构建系统。

总而言之，使用构建系统而不是直接调用编译器，对于复杂C/C++项目有下列好处：

- 增强构建流程可维护性
- 简化执行部分任务（类似于脚本），例如清理构建中间产物等
- 通过部分编译/增量编译，可以显著减少大规模项目的编译时间
- 可以并发执行所有前置已被满足的目标（并发编译），有效利用多核CPU，进一步缩短编译时间

扩展阅读

- https://faculty.cs.niu.edu/~mcmahon/CS241/Notes/makefiles.html
- https://www.gnu.org/software/make/manual/html_node/index.html
- make是一个比较底层的构建系统，在使用上仍有诸多繁琐之处，于是又出现了一些用于替代make或在make更上层，更为简化的构建系统，前者的代表项目之一是ninja，后者的代表项目之一是CMake，感兴趣的同学可以进一步了解。

QEMU

QEMU能做什么?

qemu是目前广泛应用的开源模拟器和虚拟机项目。它可以在一种架构的CPU（如x86）上，模拟其他多种架构的CPU（如aarch64等），这使得可以通过qemu在x86 CPU上运行为其他CPU架构编译的程序。例如，本实验的目的是让同学们熟悉aarch64汇编，因此我们提供的炸弹程序是为aarch64 CPU编译的，如果没有qemu，则程序不能在x86 CPU上执行。此外，qemu也可以模拟一个完整的机器，包括CPU、内存、磁盘以及多种其他外部硬件设备，此时它的功能基本等价于虚拟机，在这种场景下，它还可以与KVM技术配合，使用硬件虚拟化提高虚拟机的运行性能，但此时它就不能再运行为其他架构编译的操作系统了。

进程级模拟 vs 系统级模拟

qemu的模拟粒度可以分为进程级和系统级。在进程级模拟下，qemu只负责运行一个为其他架构编译的普通程序，这个程序与当前系统中运行的程序的唯一区别是它所使用的指令集不同，qemu会负责将它所使用的指令集翻译为当前机器可以执行的指令。除此之外，该程序相当于当前系统中的一个普通进程，它仍然通过当前系统的内核来访问操作系统提供的功能。默认情况下，`qemu`命令执行进程级模拟。例如，我们提供的炸弹程序就只需要使用进程级模拟，它本质上就是一个普通的用户态程序，只是编译到了aarch64指令集而非x86指令集。除此之外，它的输入/输出等功能，仍然是调用当前操作系统提供的syscall。

`qemu-system`命令可以用于系统级模拟。在系统级模拟下，QEMU会模拟一整套硬件，包括CPU、内存、磁盘以及多种可选硬件设备，此时QEMU的功能类似于虚拟机。在系统级模拟下，QEMU不能直接运行普通的用户态程序，而是需要运行完整的操作系统，由操作系统来管理QEMU模拟出的硬件资源。系统级模拟与是否使用KVM等硬件虚拟化进行加速是正交的。如果不使用KVM，QEMU仍然通过动态指令集翻译来运行被模拟的操作系统，此时它可以运行为其他架构编译的操作系统。否则，使用硬件虚拟化可以增强性能，但QEMU不再可以运行为其他架构编译的操作系统。

QEMU+gdb

qemu实现了gdbserver协议，这使得gdb可以连接到qemu，并且给qemu发送指令，以及从qemu中读取信息。通过gdbserver，qemu可以把自己内部的信息暴露出来，或者说，可以让gdb理解它需要调试的目标是qemu中被模拟的程序（或操作系统），而不是qemu自身。例如，当gdb发送在地址`0x400000`设置一个断点的指令时，qemu可以在被模拟程序执行到`0x400000`时暂停被模拟程序的执行，并告知gdb客户端，而不是在qemu自身执行到`0x400000`时暂停。如果gdb需要读取内存中的值，qemu可以返回被调试程序的内存数据，而不是自身的内存数据。

还需要强调，由于gdb和gdbserver之间交换的信息仅停留在汇编级，而且gdb完全使用本地的可执行程序文件中的符号表和调试信息执行源码级信息到汇编级信息之间的转换，因此在使用qemu系统级模拟时，我们可能会发现一些意料之外的行为。这是因为，在系统级模拟时，QEMU自身相当于站在CPU的抽象层面，而不是操作系统内核的抽象层面，这使得QEMU自身并不能理解它内部正在运行的操作系统的相关信息。例如，如果在qemu中运行一个Linux操作系统，其中的每一个进程的虚拟地址空间都有许多重叠，那么如果此时在`0x400000`打一个断点，qemu并不能识别出这个断点是针对哪个进程的，它所能做的只是在CPU“执行”到地址为`0x400000`的指令时暂停执行，不论当前执行的到底是哪个进程。又由于gdb客户端是完全根据用户所指定的可执行文件进行源码级到汇编级的翻译的，所以可能会出现这样一种情况：

- gdb客户端从`appA`中读取符号表，其中，函数`funcA`的地址为`0x412345`，它有2个参数
- 设置断点：`break funcA`，但实际上，gdb会根据符号表把`funcA`翻译回`0x412345`，qemu的gdbserver只能接收到在`0x412345`打断点的指令。
- 在qemu模拟的Linux操作系统中，进程`appB`恰好在执行，且它的函数`funcB`（只有1个参数）的地址恰好也为`0x412345`，qemu并不能理解这一点，它同样会暂停进程`appB`的执行。
- 根据`appA`的调试信息，gdb客户端知道`funcA`有2个参数，所以它会要求gdbserver读取寄存器`x0`和`x1`的值，用于显示`funcA`的参数。但由于此时暂停执行的实际上是只有1个参数的`funcB`，所以`x1`中保存的是一个随机值，`x0`中保存的值很可能也不符合`funcA`的第一个参数的语义。于是我们会看到虽然触发了断点，但函数参数却像是乱码。
- 同学们在后续chcore用户态相关的实验中，会对这个问题有更进一步的体会和理解。

扩展阅读

- https://www.qemu.org/docs/master/