Секция № 1

Инструменты

Содержание

1. Make . . . 1
   1.1. Некоторые факты про `make` . . . 1
2. Git . . . 2
   2.1. Команды для запоминания . . . 2
3. GDB . . . 3
   3.1. Команды для изучения . . . 3
4. Отладка программ в Linux . . . 5
   4.1. strace и ltrace . . . 5
   4.2. catchsegv . . . 5
   4.3. top . . . 6
   4.4. ps . . . 6
   4.5. pidof и pgrep . . . 7
   4.6. proc . . . 7
   4.7. lsof . . . 8
   4.8. netstat и ip . . . 8
   4.9. kill, pkill и killall . . . 8

Инструменты важны для каждого разработчика. Если ты тратишь время на то, чтобы разобраться с правильными инструментами, на самом деле ты экономишь время, которое будешь тратить на написание и отладку кода. В этой секции будут рассмотрены основные полезные инструменты.

1. Make

Программа `make` обычно используется для построения других програм. Когда ты запускаешь `make`, программа ищет в текущей директории файл `Makefile` и выплняет правила записаные в нём на особом языке.

```
my_first_makefile_rule:
    echo "Hello world"
```

Строительными блоками в `make` служат **правила**. Представленное правило состоит из **цели** (`my_first_makefile_rule`) и **рецепта** (`echo "Hello world"`). Если нужно получить от `make` цель, то нужно запустить в терминале `make my_first_makefile_rule`, это приведёт к тому, что `make` отыщет в `Makefile` запрошеное правило и выполнит соответствующий рецепт — напечатает `Hello world`. Рецепты состоят из команд терминала, можешь попробовать выполнить отдельно `echo "Hello world"` и посмотреть что получится.

Правила могут содержать **зависимости** — те цели, которые необходимо выполнить до выполнения текущего правила. В следующем примере правило `task_two` имеет единственную зависимоть: `task_one`. Если выполнить в терминале `make task_two`, то `make` сначала выполнит `task_one`, а затем `task_two`.

```
task_one:
    echo 1
task_two: task_one
    echo 2
```

1.1. Некоторые факты про `make`

- Если запустить `make` не указывая цель, то из `Makefile` будет выбрана первая цель.
- По соглашению названия целей обычно соответствуют именам файлов. Если файл-цель существует и время его модификации больше, чем время модификации *всех* его зависимостей, то `make` пропустит рецепт из этого правила. Если файл-цель не существует, `make` будет считать, что время модификации этой цели «начало времен» (минимально возможная в данной системе дата и время). Однако временная метка цели, это время последней модификации соответствующего файла.
- Если запустить `make clean` (служебное действие по удалению созданых файлов), то рецепт будет исполняться каждый раз, поскольку файл `clean` никогда не будет создан. (Для более устойчивой работы попробуй использовать `.PHONY` — интернет в помомщь.)

- Отступы в `Makefile` должны быть табами, а не пробелами.
- Рецепты можно запускать параллельно (одновременно) — `make -j 4` (в данном случае четыре параллельных задачи).
- `make` может создавать правила автоматически, если иное не указано. Например, можешь создать файл `my_program.c` и даже без `Makefile` программа `make` сможет его скомпилировать — `make my_program`.
- Язык правил `Makefile` довольно разнообразен, зачатую можно встретить специальные переменные `$@` и `$<`. Посмотри в интернете зачем они.

2. Git

Git — распределенная система контроля версий и система управления исходным кодом (SCM), нацеленная на скорость работы, целостность данных и поддержке распределенных, нелинейных вариантов взаимодействия разработчиков. GitHub — это севис хостинга git-репозиториев, которая помимо собственно git предоставляет множество дополнительных возможностей для взаимодействия пользователей (по факту GitHub — социальная сеть для разработчиков).

В этом курсе git и GitHub будут использоваться для управления программным кодом. Важно изучить git не теоретически, а практически, то есть руками. Попрактикуйся в онлайн-визуализаторе вот здесь: http://git-school.github.io/visualizing-git/.

2.1. Команды для запоминания

`git init`

Создание репозитория в текущей директории.

`git clone <url>`

Склонировать репозиторий в новую директорию.

`git status`

Показывает текущее состояние рабочего дерева.

`git pull <репозиторий> <ветвь>`

Скачать изменения из одного репозитория в другой или в локальную ветку.

`git push <репозиторий> <ветвь>`

Отправить локальные изменения в удаленный репозиторий.

`git add <файл(ы)>`

Добавить содержимое файлов в индекс.

`git commit -m <Сообщение об изменениях>`

Запись изменений в репозиторий с соответствующми сообщением.

`git branch`

Получение списка или удаление веток.

`git checkout`

Переключение на ветку или пути в рабочем дереве.

`git merge`

Соединение двух или более веток.

`git rebase`

Перемещение изменений к другой точке в дереве.

`git diff [--staged]`

Построчное сравнение между текущей директорией и индексом (или между индексом и `HEAD`, при наличии `--staged`).

`git show` **[--format=raw] <коммит, ветвь, тэг>**

Вывод подробной информации о чём угодно.

`git reset` **[--hard] <коммит, ветвь, тэг>**

Сброс текущего состояния репозитория.

`git log`

Список коммитов в текущей ветви.

`git reflog`

Изменения в локальном репозитории.

Отмтим, что git особенно полезен при одновременной работе нескольких людей с одной и той же кодовой базой.

3. GDB

Программа `gdb` — отладчик (дебаггер), поддерживающий C, C++ и другие языки. Без хорошего понимания работы `gdb` тебе будет тяжело эффективно отлаживать программы. Так что пора познакомиться с ним поближе.

3.1. Команды для изучения

`run, r`

Запуск программы с точки входа. Поддерживает задание аргументов и простое перенаправление ввода/вывода.

`quit, q`

Выход из `gdb`.

`kill`

Остановка выполняемой программы.

`break, break … if <условие>, clear`

Задание точки останова при входе в функцию (например `break strcpy`) или на конкретной строчке кода (например `break file.c:80`).

`step, s`

Если текущая строка кода содержит вызов функции, то `gdb` шагнёт внутрь тела этой функции. Если вызова функции нет, то будет выполнена текущая строка и выполнение остановится на следующей.

`next, n`

Выполнение текущей строки кода и остановка на следующей. В вызовы функций не заходит.

`continue, c`

Продолжение выполнения (до следующей точки останова или до конца программы).

`finish`

Выполнение текущей фукции до конца.

`print, p`

Вывод значения переменной.

`call`

Выполнение произвольного кода и вывод результатов.

`watch, rwatch, awatch`

Останов программы при достижении заданного условия (например `x > 5`).

`backtrace, bt, bt full`

Вывод порядка вызовов в стеке в текущем состоянии программы.

`disassemble`

Вывод кода текущей функции на ассемблере.

`set follow-fork-mode <режим>` **(MacOS это не поддерживает)**

`gdb` может одновременно отлаживать только один процесс. Если происходит `fork` (процесс клонирует себя), `gdb` может продолжить отладку в родителе (оригинале) или потомке (клоне). `Режим` может принимать значения `parent` и `child` соответственно.

Команды `print` и `call` могут применяться для выпонения произвольных строк кода во время выполнения программы! Можно менять значения переменных и вызывать функции. Например, `call close(0)` или `print i = 4`. (На самом деле в большинстве случаев `print` и `call` взаимозаменяемы). Это одна из наиболее мощных возможностей `gdb`.

4. Отладка программ в Linux

Отладка программ может осуществляться не только с помощью дебаггеров и вывода сообщений о выполнении программы (трейсинг). Образ ВМ для этого курса содержит набор инструментов для получения информации о работающих программах. Вот некоторые, наиболее полезные, из них.

4.1. strace и ltrace

Все программы используют функции стандартной библиотеки, типа `printf`, подавляющее большинство которых приводят к **системным вызовам**, взаимодействующих с ядром ОС. Это справедливо для любой программы написаной на любом языке (C, Python, Java и так далее). Программы `strace` и `ltrace` позволяют отобразить все системные вызоывы (`strace`) или все вызовы функций стандартной библиотеки (`ltrace`), использующиеся в отлаживаемой программе.

```
# Вывод всех системных вызовов при выполнении ‘ls files/’
strace ls files/
# Вывод всех вызовов стандартной библиотеки при выполнении ‘ls files/’
ltrace ls files/
# Вывод системных вызовов запущенного процесса с идентификатором PID = 1234
strace -p 1234
# Вывод вызовов стандартной библиотеки запущенной процессом PID = 1234
ltrace -p 1234
```

Программы `strace` и `ltrace` можно использовать для определения причин зависания процесса, поиска причин низкой скорости выполнения кода, наконец из-за желания полюбопытствовать о работе чьей-то чужой программы. Обе программы готовы принять флаг `-f`, который распространит действие `strace` или `ltrace` на дочерние процессы.

4.2. catchsegv

Программа `catchsegv` помогает получить полезную информацию при наступлении ошибки сегментации (segmentation fault).

```
catchsegv ./твоя_глючная_программка
```

Вывод `catchsegv` состоит из трёх частей. Первая часть содержит состояния всех регистров CPU при наступлении ошибки. Вторая часть выводит содержимое стека на момент ошибки (это сработает только при указанном флаге `-g` при компиляции). Третья часть выводит содержимое памяти. Программа `catchsegv` также может сообщать о других ошибках и крэшах.

4.3. top

Программа `top` отображает в текстовом интерактивном режиме наиболее активные запущенные процессы. Для выхода достаточно нажать `q`. Для вывода справки по всем остальным командам — нажать `c`. Клавиши ←, → для изменения сортируемого столбца. Вот что показывают в столбцах:

PID

Идентификатор процесса.

PR

Значение приоритета уровня ядра (PR).

NI

Значение приоритета уровня пользователя (NI).

VIRT

Объем виртуальной памяти используемой процессом (KiB).

RES

Объем резидентной памяти (расположенной в физической памяти) используемой процессом (KiB).

SHR

Объем разделяемой памяти (общие библиотеки, файлы и так далее) (KiB).

S

Состояние процесса (S — сон, R — исполнение, T - приостановленный, Z — зомби).

%CPU, %MEM

Проценты использования процессом ресурса CPU и физической памяти.

TIME+

Количество «времени процессора» использованного процессом. Измеряется в секундах.

COMMAND

Название исполнимого файла.

4.4. ps

Программа `ps` выводит список выполняющихся процессов. По-умолчанию `ps` выведет только процессы относящиеся к текущей консольной сессии, которая скорее всего включает только сам `ps` и `bash`. Для получения более полной информации можно вызвать `ps -eLf` или `ps auxww`.

Информация предоставляемая `ps` почти полностью дублирует то, что можно получить из `top`. Однако вывод `ps` проще отфильтроватm и найти необходимое. Например так `ps aux | grep bash`.

Столбец `PPID` присутствующий в выводе `ps -eLf` сообщает идентификатор PID родительского процесса. Столбец `NLWP` выводит информацию о количестве легких процессов, нитей (thread) в процессе.

4.5. pidof и pgrep

Программа `pidof` выводит PID по имени процесса. Например, `pidof vim` выведет PID процесса vim. Если процессов с одинаковым именем несколько, то будут выведены все PID'ы. Это может быть удобно при передаче в качестве аргумента в другую прогамму. Например, прикрепление `gdb` к процессу `httpserver` можно достигнуть следующей командой `gdb -p $(pidof httpserver)`.

Так же работает команда `pgrep`, разница в том, что она не требует полного названия процесса, а производит поиск на основании частичного совпадения. То есть `pgrep a` выведет PID'ы всех процессов в названии которых присутствует «a».

4.6. proc

Директорий `/proc` в Linux'е — настоящий кладезь информации. Можно выполнить `ls /proc/1` для получения информации о процессе с PID=1. Многие рассмотреные выше инструменты используют `/proc` в качестве источника информации. Если надо получить полные «сырые» данные, заглядывай туда! Директорий `/proc/self` — это способ получения информации о текущем процессе. Команда `ls /proc/self` выведет информацию о процессе `ls`, который выводит эту информацию.

Некоторые полезные части `/proc`:

`cmdline`
: Команда использованая для запуска процесса.

`cwd`
: Текущий рабочий директорий процесса.

`environ`
: Переменные окружения процесса.

`exe`
: Выполняемая в процессе программа.

`fd/`
: Список дескрипторов файлов.

`fdinfo/`
: Более подробная информация о дескрипторах файлов.

`maps`
: Карта памяти процесса.

`net`

Информация о сетевой конфигурации.

`status`

Состояние процесса.

4.7. lsof

Программа `lsof` предоставляет информацию о файлах, процессах и сетевых соединениях.

Полезные примеры:

`lsof /lib/x86_64-linux-gnu/libc.so.6`

Список процессов, использующих стандартную библиотеку C.

`lsof -p 1234`

Список файлов, открытых процессом с PID=1234.

`lsof -i tcp:80`

Список «использований» TCP-порта 80 (http).

4.8. netstat и ip

Программы `netstat` и `ip` помогают разобраться с сетевыми соединениями. Вот несколько полезных примеров использования:

- Запусти `ip addr` или `ifconfig` для получения списка всех сетевых интерфейсов и ip-адресов.
- Запусти `ip route` или `netstat -nr` для вывода таблицы маршрутизации.
- Запусти `netstat -nt` для получения списка текущих TCP-соединений.
- Запусти `netstat -ntl` для получения списка открытых TCP-портов.

4.9. kill, pkill и killall

Для уничтожения процесса можно вызвать `kill <PID>`. К команде также можно присоединить сигнал, например `kill -TERM <PID>` или `kill -QUIT <PID>`.

Программа `pkill` убивает все процессы, соответствующие заданному шаблону. Например, `pkill v` уничтожит все процессы, в имени которых есть «v».

Программа `killall` делает то же, что и `pkill`, но требует полного совпадения.