

# Linux Kernel Exploration

Development Environment & Debugging Tips

March 9, 2021

SNU Operating Systems

# Table of Contents

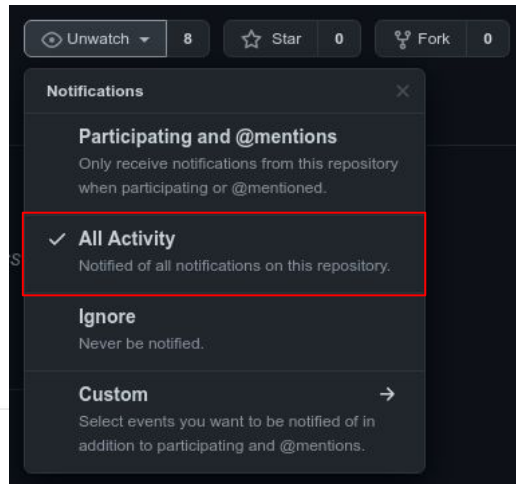
- Notices
- Host OS & Project hardware
  - Use Ubuntu 18.04 or 20.04.
- Building the kernel
  - Setting up the environment might not be trivial.
- Coding environment
  - Setting up jump-to-definition, completion, etc with LSP.
- Debugging Setup
  - Running GDB on QEMU.
- Kernel Programming
  - Kernel programming is not like ordinary programming.

# Learning Platforms

1. eTL: Lecture slides and Zoom links
2. Class Github repository: Q&A, docs, and notices
3. Github Classroom: Assignment distribution

# Notification Settings

- <https://github.com/swsnu/osspr2021>
- <https://github.com/settings/emails>
- <https://github.com/settings/notifications>



## Email notification preferences

Default notification email

 Save

Choose which email updates you receive on conversations you're participating in or watching

- ☒ Comments on Issues and Pull Requests
- ☒ Pull Request reviews
- ☒ Pull Request pushes
- ☐ Include your own updates

## Custom routing

You can send notifications to different **verified** email addresses depending on the organization that owns the repository.

 swsnu	hsgkim@snu.ac.kr (default)	Edit
---	----------------------------	------

# Issue Board

- <https://github.com/swsnu/osspr2021/issues>
- All questions and announcements go to the issue board
- Active participation is highly encouraged
- Do NOT close your issue!

# Host OS

- **Use Ubuntu 18.04 or 20.04.**
  - Arch Linux works, too
  - You must compile QEMU from source on Ubuntu 18.04
- Virtual machines should work, but they might be slow and difficult to configure
- We heard that WSL has dependency problems
- It is up to the students to setup their own project machine

# Project Hardware

- We won't use Raspberry Pi 3 this year
- We'll use the QEMU full-system emulator throughout the course
- Just in case you want to run your kernel on a real device
  - Tizen OS makes it easy for us to flash devices and run the kernel
  - (Ideally) develop using QEMU and test on RPi3 (much faster since it's real hardware)
  - <https://docs.tizen.org/platform/developing/flashing-rpi/>

# Building and Booting the Kernel

- Build: `./build-rpi3.sh`
- Setup image files: `./setup-images.sh`
- Boot up QEMU: `./qemu.sh`
- Make sure you read and understand the scripts that we provide



# Configuring the Coding Environment

- Dealing with kernel code only with vanilla vim is *extremely* painful
- Quality-of-life features for kernel development
  - jump to definition/declaration
  - find and list references
  - code completion as you type
- You can find guides for ctags or cscope everywhere
  - not much merit
- Language Servers - What we'll learn today!

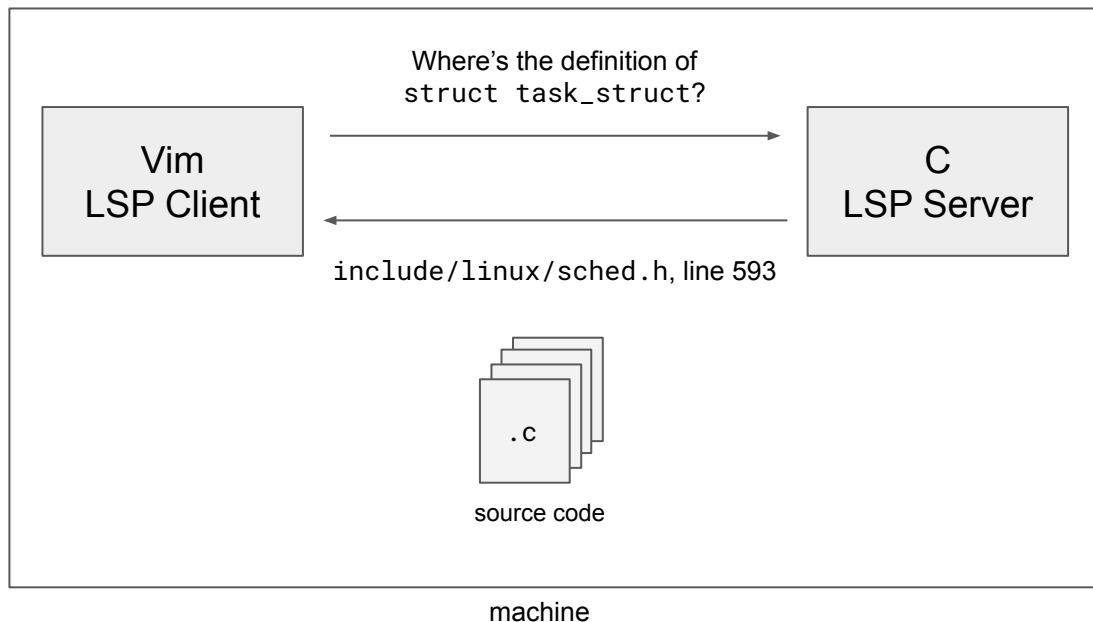
# Language Server Protocol (LSP)<sup>1</sup>

- Without LSP: each editor implements custom support for each language

	C/C++	Python	Rust
Vim	ronakg/quickcr-cscope	davidhalter/jedi-vim	rust-lang/rust.vim
VSCode	C/C++ plugin from marketplace	Python plugin from marketplace	Rust plugin from marketplace
Sublime Text	...	...	...

# Language Server Protocol (LSP)

- LSP: The editor asks the language server about the source code



# Language Server Protocol (LSP)

- With LSP: Editor and language decoupled via the server-client architecture!

	LSP Client Impl.
Vim	<code>prabirshrestha/vim-lsp</code>
Neovim	<code>built-in</code>
VSCode	<code>built-in</code>

	LSP Server Impl.
C/C++	<code>ccls, clangd</code>
Python	<code>pyls, pyls_ms</code>
Rust	<code>rls, rust-analyzer</code>

# Setting up LSP for Kernel Development

- We'll use the ccls language server for C/C++/ObjC
  - <https://github.com/MaskRay/ccls>
- We'll use Neovim since it has an LSP client built-in
  - <https://github.com/neovim/neovim>
  - Version  $\geq 0.5$  (nightly) required for built-in LSP client

# Setting up ccls for Linux Kernel Development

- Installation

- ```
# ccls and neovim
apt-get update
apt-get install curl rsync git ccls -y # 18.04: snap install ccls --classic
mkdir -p ~/.local
curl -LO
https://github.com/neovim/neovim/releases/download/nightly/nvim-linux64.tar.gz
tar xzf nvim-linux64.tar.gz
rsync -a nvim-linux64/* ~/.local
export PATH="$HOME/.local/bin:$PATH"
```
- ```
# vim-plug for neovim plugin management
curl -fLo ~/.local/share/nvim/site/autoload/plug.vim --create-dirs \
https://raw.githubusercontent.com/junegunn/vim-plug/master/plug.vim
```

# Setting up ccls for Linux Kernel Development

- Setup Neovim (~/.config/nvim/init.vim)

- ```
call plug#begin()
Plug 'neovim/nvim-lspconfig'
Plug 'nvim-lua/completion-nvim'
Plug 'nvim-lua/popup.nvim'
Plug 'nvim-lua/plenary.nvim'
Plug 'nvim-telescope/telescope.nvim'
call plug#end()

set omnifunc=v:lua.vim.lsp.omnifunc
set completeopt=menuone,noinsert,noselect
set shortmess+=c
set signcolumn=yes

nnoremap gd <cmd>lua vim.lsp.buf.definition()<CR>
nnoremap gr <cmd>Telescope lsp_references<CR>
nnoremap gw <cmd>Telescope lsp_workspace_symbols<CR>
imap <Tab> <Plug>(completion_smart_tab)
imap <S-Tab> <Plug>(completion_smart_s_tab)
```

# Setting up ccls for Linux Kernel Development

- Setup Neovim (~/.config/nvim/init.vim) - continued

- ```
lua << END
local lspconfig = require'lspconfig'
lspconfig.ccls.setup{
  on_attach = require'completion'.on_attach,
  init_options = {
    client = { snippetSupport = false }
  }
}
END
```

- Generate compile\_commands.json (I already did this for you)

- ```
pip install compiledb
compiledb make ARCH=arm64 CROSS_COMPILE=aarch64-linux-gnu- -j$(nproc)
```

- Index the entire kernel (optional)

- ```
ccls --index .
```



# Use printk for kernel messaging

- `printf` does not work inside the linux kernel
- `printk` syntax
  - `printk(<loglevel> "message", <arguments>);`
  - No comma between log level and message!
- You can use `printk` instead for feeding kernel messages
  - Declared in `include/linux/kernel.h`
  - Mostly compatible with ANSI C `printf` function
  - Can give priority by concatenating pre-defined string constants
    - e.g. `printk(KERN_INFO "this syscall is called!\n");`
    - `KERN_INFO` is what you want for most cases
- You can read kernel messages using `dmesg`
  - `dmesg -w`: wait for new messages
  - `dmesg | tail`: only show last few messages

# Use printk for kernel messaging

- You should be aware that...
  - The internal buffer size of printk is limited to 1kB
  - Too many kernel messages will make the system slow or even hang completely
  - Make sure to add a line separator at the end of each message
- Good read: <https://www.kernel.bz/boardPost/118679/17>
  - Log levels (e.g. KERN\_INFO, KERN\_ALERT), and how to adjust how much to print in system
- Use KERN\_ALERT to print a message directly to the console
  - `printk(KERN_ALERT "hello world\n");`
- Where was this called?
  - `printk(KERN_INFO "%s: %s (%d)\n", __FILE__, __FUNCTION__, __LINE__);`

# Linux Cross Reference (LXR)

- <https://elixir.bootlin.com/linux/v4.19.49/source>
- Find functions/symbols in Linux kernel
- You won't need this if you setup `cc1s`

/

start\_kernel

Q

Defined in 6 files:

arch/alpha/boot/bootp.c, line 135 *(as a Function)*  
arch/alpha/boot/bootpz.c, line 263 *(as a Function)*  
arch/alpha/boot/main.c, line 152 *(as a Function)*  
arch/um/kernel/skas/process.c, line 16 *(as a prototype)*  
include/linux/start\_kernel.h, line 11 *(as a prototype)*  
init/main.c, line 513 *(as a Function)*

Referenced in 16 files:

arch/alpha/boot/bootp.c, line 135  
arch/alpha/boot/bootpz.c, line 263  
arch/alpha/boot/main.c, line 152  
arch/frv/kernel/debug-stub.c, line 123  
arch/metag/kernel/setup.c, line 553  
arch/mips/kernel/relocate.c  
├─ line 305  
└─ line 399

# Kernel Debugging with GDB

- We can attach GDB to the running kernel inside QEMU
  - `target remote :1234` will attach the GDB instance to the kernel
  - Made possible due to the flag `-gdb tcp:1234` given to QEMU
- Installation
  - `sudo apt-get install gdb-multiarch`
- Running GDB (after booting up QEMU)
  - `gdb-multiarch ./vmlinux`
  - Then, inside GDB run: `target remote :1234`

## Bonus: gdb-pwndbg

```
Tue Feb 23 08:23:53 pm CPU: 5% | Mem: 15% | Disk: 106G/884G | Vol: | Linux 5.10.10-arch1-1
```

```
+PC 0xffffffff8080a2e24 (cpu_do_idle+8) -> ret /* 0xd53bd9d2d6f8f3c0 */  
[ DISASM ]  
-> 0xffffffff8080a2e24 <cpu_do_idle+8> ret  
↓  
0xffffffff80819fa54 <trace_hardirqs_on> stp x29, x30, [sp, #-0x50]  
0xffffffff80819fa58 <trace_hardirqs_on+4> mrs x0, sp_el0  
0xffffffff80819fa5c <trace_hardirqs_on+8> mov x29, sp  
0xffffffff80819fa60 <trace_hardirqs_on+12> stp x19, x20, [sp, #0x10]  
0xffffffff80819fa64 <trace_hardirqs_on+16> stp x21, x22, [sp, #0x20]  
0xffffffff80819fa68 <trace_hardirqs_on+20> mov x21, x30  
0xffffffff80819fa6c <trace_hardirqs_on+24> ldr w1, [0, #0x10]  
0xffffffff80819fa70 <trace_hardirqs_on+28> add w1, w3, #1  
0xffffffff80819fa74 <trace_hardirqs_on+32> str w1, [0, #0x10]  
0xffffffff80819fa78 <trace_hardirqs_on+36> adrp x19, #frame_stack+1696 <0xffffffff80891e1808>  
[ SOURCE CODE ]  
In file: /home/jaywongchun/workspace/os/solution/osspr2021-project/arch/arm64/mm/proc.S  
55 Idle the processor (wait for interrupt).  
56  
57 ENTRY(cpu_do_idle)  
58 dsb sy // WFI may enter a low-power mode  
59 wfi  
-> 60 ret  
61 ENDPROC(cpu_do_idle)  
62  
63 #ifdef CONFIG_ARM64_CPU_PM  
64  
65 cpu_do_suspend - save CPU registers context  
[ STACK ]  
00:0000] x29 sp 0xffffffff8089493fc0 (init_thread_union+16044) -> 0xffffffff8089493fe0 (init_thread_union+16112) -> 0xffffffff8089493fa0 (init_thread_union+16224) -> 0xffffffff8089493fb0 (init_thread_union+16256) -> 0xffffffff8089493fd0 (init_thread_union+16288) < ...  
01:0000] 0xffffffff8089493fc0 (init_thread_union+16072) -> 0xffffffff8080dfefc4 (do_idle+596) -> bl #0xffffffff80801a0d00 /* 0x17ffffbf94038a7 */  
02:0010] 0xffffffff8089493cd0 (init_thread_union+16080) -> 0x0  
03:0010] 0xffffffff8089493cd0 (init_thread_union+16080) -> 0xffffffff8089499670 (_cpu_online_mask) -> 0xf  
04:0020] 0xffffffff8089493ce0 (init_thread_union+16096) -> 0xffffffff8067054108 (kallsyms_lookup_index+527888) -> 0x6416572 /* re-ad */  
05:0020] 0xffffffff8089493ce0 (init_thread_union+16096) -> 0xffffffff8089499808 (nf_contrack_locks+2496) -> 0x0  
06:0030] 0xffffffff8089493fe0 (init_thread_union+16112) -> 0xffffffff8089493fb0 (init_thread_union+16224) -> 0xffffffff8089493fa0 (init_thread_union+16256) -> 0xffffffff8089493fd0 (init_thread_union+16288) -> 0x0  
07:0030] 0xffffffff8089493fe0 (init_thread_union+16120) -> 0xffffffff8080ae013c (cpu_startup_entry+44) -> bl #0xffffffff80808dfc90 /* 0x17ffffff97ffed5 */  
[ BACKTRACE ]  
-> f 0 ffffffff8080a2e24 cpu_do_idle+8  
f 1 ffffffff80808b7e38 arch_cpu_idle+40  
f 2 ffffffff80808dfef4 do_idle+596  
f 3 ffffffff80808dfef4 do_idle+596  
f 4 ffffffff80808dfef4 do_idle+596  
f 5 ffffffff8080ae013c cpu_startup_entry+44  
f 6 ffffffff8080cf90bc rest_init+212  
  
pan@ubuntu:~$ sys_ptrace  
Function "sys_ptrace" not defined.  
pan@ubuntu:~$ arm64_sys_ptrace  
Breakpoint 1 at 0xffffffff80808d4980: file kernel/ptrace.c, line 91.  
pan@ubuntu:~$ c  
Continuing.
```

# Bonus: gdb-pwndbg

- *“pwndbg is a GDB plugin that makes debugging with GDB suck less”*
  - <https://github.com/pwndbg/pwndbg>
  - Another similar project: gdb-peda (<https://github.com/longld/peda>)
- Installation
  - ```
git clone https://github.com/pwndbg/pwndbg  
cd pwndbg  
./setup.sh
```

# Important Directories

- **arch**
  - Architecture dependent (i.e. x86, arm, mips, ...) parts of Linux
  - We're only interested in arch/arm64
- **kernel**
  - Common kernel code
- **net**
  - Common network related code
- **drivers**
  - Common driver code for Linux
- **fs**
  - Common file system code for Linux
- **include**
  - Common header files

# Our Architecture

- AArch64 kernel
- ARMv7 userland
- This means there are compatibility issues!
  - Example: `sizeof(long)` is different



# Kernel Configuration

- Config, then build
  - `make menuconfig`, `make xconfig`, `make gconfig`
  - Creates a file called `.config` in the kernel root
- Using distributed config files
  - Our version: `arch/arm64/configs/tizen_bcmrpi3_defconfig`
  - Applying it:
    - `make ARCH=arm64 CROSS_COMPILE=aarch64-linux-gnu- tizen_bcmrpi3_defconfig`

# Things to Keep in Mind...

- No memory protection
  - Corruption in kernel memory space can make the whole machine crash!
- No floating point or MMX operation
  - Dealing with real numbers can be challenging and painful!
    - You unfortunately have to do it for some projects :(
- Rigid stack limit
  - Use extra caution when allocating local arrays or having recursive calls
  - `kmalloc` instead for huge arrays
- Your kernel code will run in a multi-core environment
  - Use proper synchronization mechanisms to avoid race conditions
  - Beware of deadlocks

Q & A

## ... one more thing!

- We prepared a thorough setup demo for you
  - Ubuntu 18.04: <https://asciinema.org/a/jbnD7iyz34f1UoF9UaDYxc5T3>
  - Ubuntu 20.04: <https://asciinema.org/a/ruGNu9GWwvSuhaFf38vJr6CkQ>
- From nothing to running a cross-compiled binary inside QEMU in 20 minutes
- You can copy text directly from the demo and paste it in your command line
- Hope this helps :)