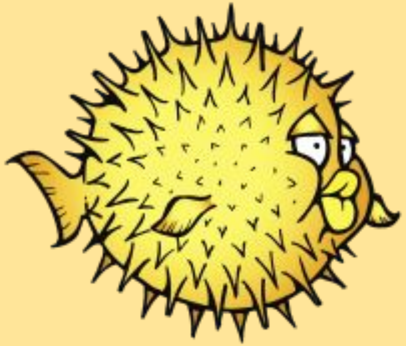


Introduction to OpenBSD Kernel Development and System Calls

COMP3301 - Week 2 Contact

COMP3301 - 2025

The University of Queensland
Australia

Overview

In this contact, we will:

- Briefly Discuss User space vs Kernel Space
- OpenBSD source tree
- Introduction to Syscalls
  - How they work
- Demo adding a syscall to OpenBSD

Basic OS Overview (More in lectures/text book)

Userland

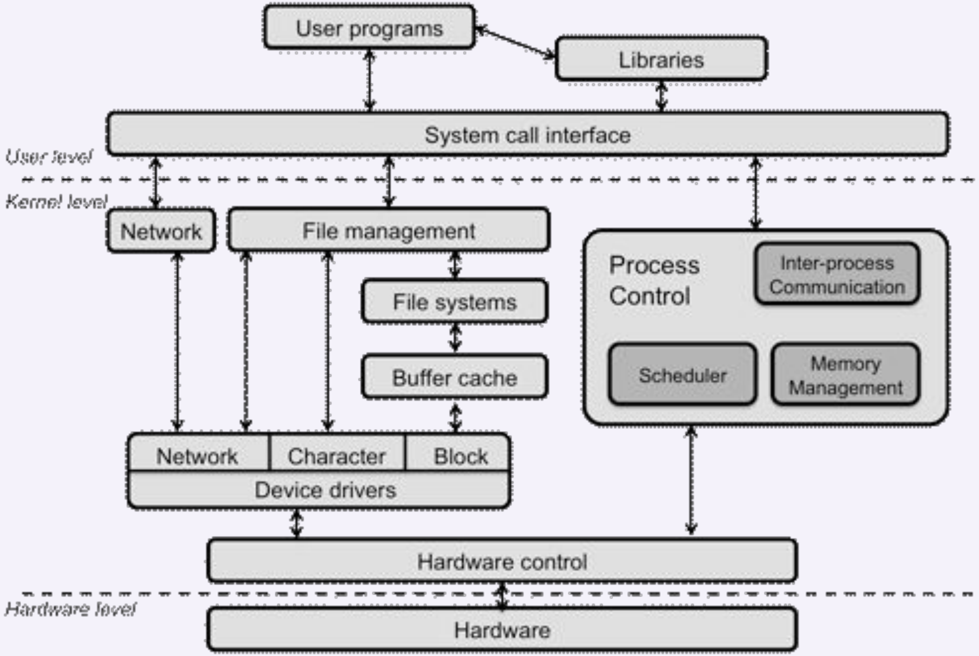
- User processes
- Each with their own context, memory etc.
- Everything is rainbows and butterflies here, the system just “works” (most of the time)

Kernel

- Provides interfaces to hardware, controls user processes, files, file systems, devices etc.
- Layers upon layers of abstraction and subsystems are used to abstract away to complexity of the OS
- It needs to make things just “work”

Hardware

- What we’re trying to abstract away
- Can be nasty, Often broken and can be unreliable
- Can influence our decisions significantly



OpenBSD Source Directories

OpenBSD source code exists in /usr/src

- Not there by default – you should have cloned it by now

Source directories (relative to /usr/src)

- /sys - Where the kernel source code exists (system)
- /bin - core system programs
- /sbin - core sudo programs
- /usr.bin - system programs
- /usr.sbin - sudo programs
- /lib - libraries
- /libexec - library executable (ld.so)
- /include - includes

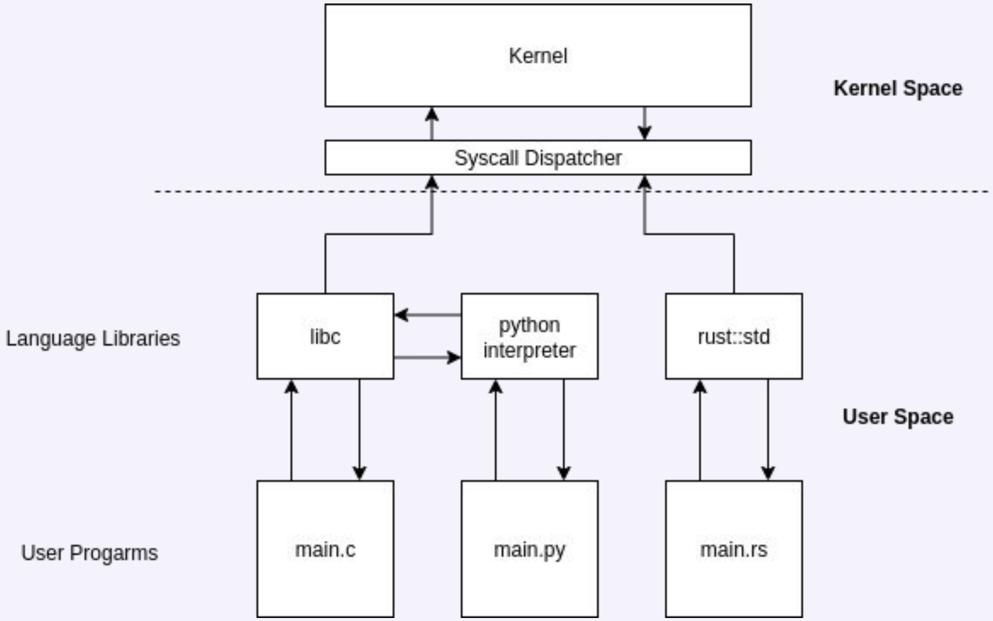
System Calls (syscalls)

Entry point into the kernel

- A request to the kernel – usually to make a request a user process cannot do on it’s own (e.g. for hardware or system resource access)
- Gets called through a **library** wrapper function
- Examples:
  - read(), dup(), open(), fork()

System Call Interface Diagram

- Kernel and User space are separate entities
- System calls are made to kernel through syscall dispatcher
- Syscalls are made via wrappers that exist in the standard library of the language you are using.



libc System Call Wrapper (amd64):
– yes this means these are architecture specific

- Moves Parameters into registers (as it would for a function call)
- Performs a special context switch into the kernel
  "**syscall"** instruction
  - Blocks the user process
  - Hardware privilege escalation occurs on cpu
  - The kernel takes control of the cpu and serves the request from its own code. Hardware privilege is restored to previous
  - Resumes the user process
- Context switch back into the user process
- User process now has the result of the syscall and now continues to treat the call like any other function call

Disassembly of the library wrapper from Prac 2:

```
<add2+0>:    repz (bad)
<add2+3>:    cli
<add2+4>:    mov    396757(%rip),%r11        #
<add2+11>:   xor    (%rsp),%r11
<add2+15>:   push   %r11
<add2+17>:   mov    $0x14b,%eax
<add2+22>:   mov    %rcx,%r10
<add2+25>:   syscall
<add2+27>:   jae    0x904ac <add2+44>
<add2+29>:   mov    %eax,%fs:0x20
<add2+37>:   mov    $0xffffffffffffffff,%rax
<add2+44>:   pop    %r11
<add2+46>:   xor    (%rsp),%r11
<add2+50>:   cmp    396711(%rip),%r11        #
<add2+57>:   je     0x904cc <add2+76>
<add2+59>:   int3
```

System Calls in OpenBSD (Live DEMO!)

- Write a syscall to increment a positive integer by 1
    - `int incnum(int num);`
    - Takes an `int` and returns an `int`
    - Fails (returns `-1`) if the input is negative, and sets `errno` to `EINVAL` (`22`)
    - Else returns `num + 1` and sets `errno` to `0`
    - Write a sysc
- All details are covered in prac 2 for a different syscall

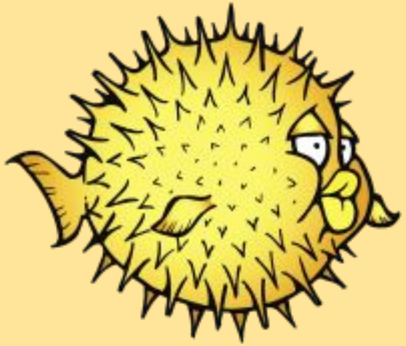
System Calls – trap.c
(https://github.com/openbsd/src/blob/master/sys/arch/amd64/amd64/trap.c)

```c
/*
 * syscall(frame):
 *	System call request from POSIX system call gate interface to kernel.
 */
void
syscall(struct trapframe *frame)
{
	const struct sysent *callp;
	struct proc *p;
	int error = ENOSYS;
	register_t code, *args, rval[2];

	verify_smap(__func__);
	uvmexp.syscalls++;
	p = curproc;

	if (verify_pkru(p)) {
		userret(p);
		return;
	}

	code = frame->tf_rax;
	args = (register_t *)&frame->tf_rdi;

	if (code <= 0 || code >= SYS_MAXSYSCALL)
		goto bad;
	callp = sysent + code;

	rval[0] = 0;
	rval[1] = 0;

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

```c
	switch (error) {
	case 0:
		frame->tf_rax = rval[0];
		frame->tf_rflags &= ~PSL_C;	/* carry bit */
		break;
	case ERESTART:
		/* Back up over the syscall instruction (2 bytes) */
		frame->tf_rip -= 2;
		break;
	case EJUSTRETURN:
		/* nothing to do */
		break;
	default:
	bad:
		frame->tf_rax = error;
		frame->tf_rflags |= PSL_C;	/* carry bit */
		break;
	}

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

Notes:

- last slide shows where the kernel process starts
- If you’re feeling curious, try following the path into your add2 syscall once you finish implementing it in the prac
- Prac 2 is very important – it will be relevant to assignment 1



HAPPY DEVVING!!!

Thanks for Coming

COMP3301 - 2025