Final Review
Call Chain of a System Call

7 December 2018
Aftab Hussain
University of California, Irvine
a visual summary of what we’ve done thus far
xv6 is a program
a legend of the diagrams to follow
FRONT END

user → shell [system call] → API → kernel

- read()
- write()
- fork()
- exec()
The diagram illustrates the interaction between the user, shell, API, kernel, and I/O devices. The user interacts with the shell, which then calls the API. The API communicates with the kernel, and the kernel uses system calls to manage the user interface, file descriptors, and I/O devices such as printer, monitor, keyboard, disk, and file.
I/O Device

user -> shell -> system call -> API -> kernel

I/O Device

printer
monitor
keyboard
disk

file descriptors

read()
write()
fork()
exec()

creating processes
I/O Device

printer
monitor
keyboard
disk

file descriptors

user
shell
API
ekernel

system call

read()
write()
fork()
exec()

creating processes

FRONT END

I/O Device

printer
monitor
keyboard
disk

file descriptors

user
shell
API
ekernel

system call

read()
write()
fork()
exec()

creating processes

FRONT END
Any running program -- i.e., a process -- must be in memory, so the CPU can run it.
I/O Device
- printer
- monitor
- keyboard
- disk
- file descriptors

FRONT END
- user
- shell
- API
- kernel
- system call

BACK END
- creating processes
- memory (physical address space)

FILE DESCRIPTORS
- read()
- write()
- fork()
- exec()
I/O Device

FRONT END

user
shell
API
kernel

BACK END

memory (physical address space)

file descriptors

creating processes

printer
monitor
keyboard
disk

read()
write()
fork()
exec()

I/O Device

system call

T
I/O Device

FRONT END

user
shell
system call
API
kernel

BACK END

memory (physical address space)

printer
monitor
keyboard
disk

file descriptors

read()
write()
fork()
exec()

creating processes

memory (physical address space)
user
shell
system
API
kernel

I/O Device
printer
monitor
keyboard
disk

file descriptors
read()
write()
fork()
exec()

memory (physical address space)

isolation
- b/w user & kernel space
- b/w processes

memory virtualization

creating processes

FRONT END
BACK END
I/O Device

printer
monitor
keyboard
disk

file descriptors

create processes

user
shell
API
kernel

system call

memory (physical address space)

memory (virtual address space)

segmentation
paging

virtual address space

isolation
- b/w user & kernel space
- b/w processes

memory virtualization

fork()
exec()
read()
write()
I/O Device
- printer
- monitor
- keyboard
- disk
- file descriptors

FRONT END
- user
- shell
- system call
- API
- kernel

BACK END
- memory (physical address space)
- isolation
  - b/w user & kernel space
  - b/w processes
- memory virtualization
- virtual address space
- segmentation
- paging

Creating processes

read()
write()
fork()
exec()
So what exactly must be in memory?
I/O Device

printer
monitor
keyboard
disk

file descriptors

read()
write()
fork()
exec()

creating processes

memory virtualization

memory (physical address space)
The diagram illustrates the relationships between various components of a computing system, including:

- **Front End**:
  - User
  - Shell
  - System call
  - API
  - Kernel
  - I/O Device
    - Printer
    - Monitor
    - Keyboard
    - Disk
    - File descriptors
  - Creating processes

- **Back End**:
  - Linker
  - Loader
  - ELF format
  - Memory virtualization
  - Memory (physical address space)
  - Process data:
    - Code
    - Stack
    - Heap

- **System Calls**:
  - `read()`
  - `write()`
  - `fork()`
  - `exec()`
Need to handle dynamic info.
FRONT END

user

shell

system call

API

kernel

BACK END

printer monitor keyboard disk

file descriptors

read() write() fork() exec()

creating processes

memory (physical address space)

Linker

Loader

ELF format

memory virtualization

process data:

code

stack

heap

stack management

memory (physical address space)

I/O Device

printer

monitor

keyboard

disk

descriptors
FRONT END

user
shell
API
kernel

BACK END

Linker
Loader

memory (physical address space)

I/O Device
printer
monitor
keyboard
disk

file descriptors

read()
write()
fork()
exec()

creating processes

virtualization

process data:
code
stack
heap

ELF format

memory virtualization

memory

stack management

calling conventions
I/O Device

user

shell

system call

API

kernel

file descriptors

read()

write()

fork()

exec()

creating processes

Linker

Loader

ELF format

memory (physical address space)

registers

memory virtualization

process data:

code

stack

heap

stack management

calling conventions
### FRONT END

- User
- Shell
- System call

### BACK END

- Memory (physical address space)
- Process data:
  - Code
  - Stack
  - Heap
- ELF format
- Memory virtualization
- Stack management
- Calling conventions
- Registers

**I/O Device**
- Printer
- Monitor
- Keyboard
- Disk
- File descriptors

**Creating processes**
- read()
- write()
- fork()
- exec()

**Helps various other features of xv6 (e.g. memory virt., interrupts, by pointing to GDT, IDT, etc.)**
I/O Device

FRONT END

user
shell
system call
API
kernel

BACK END

Linker
Loader

ELF format

memory (physical address space)

registers

process data:
- code
- stack
- heap

stack management

calling conventions

memory virtualization

process descriptors

creating processes

read()
write()
fork()
exec()

I/O Device

printer
monitor
keyboard
disk

file descriptors

Linker
Loader

ELF format

memory (physical address space)

registers

process data:
- code
- stack
- heap

stack management

calling conventions

memory virtualization

process descriptors

creating processes

read()
write()
fork()
exec()
Did we forget to load the OS -- or kernel -- itself?
I/O Device
printer
monitor
keyboard
disk

user
shell
system
API
kernel

frontend
I/O Device

memory (physical address space)
process data:
- code
- data
- stack
- heap

_registers

booting
- loading bootloader
- preparing the memory
- loading the kernel
- running the first process

ELF format

Stack

Linker
Loader

calling conventions

Virtualization

reading
writing
forking
executing
multi-tasking
I/O Device
- printer
- monitor
- keyboard
- disk
- file descriptors

Creating processes

Memory (physical address space)
- registers
- process data:
  - code
  - stack
  - heap

Virtualization

Linker
ELF format

Booting

Multi-tasking

Calling conventions

Front End
- user
- shell
- system call
- API
- kernel

Back End
- Linker
- Loader
user
shell
system call
API
kernel
I/O Device
printer
monitor
keyboard
disk
file descriptors
read()
write()
fork()
exec()
creating processes
memory (physical address space)
process data:
code
stack
heap
stack management
calling conventions
multi-tasking
booting
ELF format
Linker
Loader
memory virtualization
virtualization
context switching
registers
process
loading
memory
virtualization
system
call
creating
processes
I/O Device
printer
monitor
keyboard
disk
file descriptors
read()
write()
fork()
exec()
FRONT END

- user
- shell
- system call
- API
- kernel

I/O Device

- printer
- monitor
- keyboard
- disk
- file descriptors

BACK END

- Linker
- Loader
- ELF format
- Linker
- registers
- memory (physical address space)
- process data:
  - code
  - stack
  - heap
- memory virtualization
- context switching
- synchronization
- calling conventions
- multi-tasking
- creating processes

booting

calling conventions

ELF format
I/O Device
- printer
- monitor
- keyboard
- disk
- file descriptors

Creating processes

Front End
- user
- shell
- system call
- API
- kernel

Back End
- Linker
- Loader
- ELF format
- memory virtualization
- memory (physical address space)
- stack management
- calling conventions
- registers
- context switching
- synchronization
- multi-tasking

Memory (physical address space)
- process data:
  - code
  - stack
  - heap

Booting
user  shell  system  call  API  kernel  I/O Device  printer  monitor  keyboard  disk  file descriptors

creating processes

reading  writing  fork  exec

memory (physical address space)

memory virtualization

process data: code  stack  heap

stack management

multi-tasking

context switching

synchronization

calling conventions

registers

booting

Linker  Loader

ELF format

process: code  stack  heap

virtualization

virtual memory

desktop

I/O Device

print

monitor

keyboard

disk

file descriptors

creating processes

reading  writing  fork  exec

memory (physical address space)

memory virtualization

process data: code  stack  heap

stack management

multi-tasking

context switching

synchronization

calling conventions

registers

booting

Linker  Loader

ELF format

process: code  stack  heap

virtualization

virtual memory

desktop

I/O Device

print

monitor

keyboard

disk

file descriptors

creating processes

reading  writing  fork  exec

memory (physical address space)

memory virtualization

process data: code  stack  heap

stack management

multi-tasking

context switching

synchronization

calling conventions

registers

booting

Linker  Loader

ELF format

process: code  stack  heap

virtualization

virtual memory

desktop

I/O Device

print

monitor

keyboard

disk

file descriptors

creating processes

reading  writing  fork  exec

memory (physical address space)

memory virtualization

process data: code  stack  heap

stack management

multi-tasking

context switching

synchronization

calling conventions

registers

booting

Linker  Loader

ELF format

process: code  stack  heap

virtualization

virtual memory

desktop

I/O Device

print

monitor

keyboard

disk

file descriptors

creating processes

reading  writing  fork  exec

memory (physical address space)

memory virtualization

process data: code  stack  heap

stack management

multi-tasking

context switching

synchronization
FRONT END

user
shell
system
API
kernel

BACK END

Linker
Loader

ELF format

booting

memory (physical address space)

memory virtualization

process data:

code
stack
heap

stack management

registers

context switching

synchronization

interrupts

traps

calling conventions

multi-tasking

I/O Device

printer
monitor
keyboard
disk

descriptors

creating processes

read()
write()
fork()
exec()
Now ... System Call Demo
if (write(1, buf, n) != n)
if (write(1, buf, n) != n)

defined here

SYSCALL(write)
if (write(1, buf, n) != n)
define SYSCALL(name) \  .globl name; \  name: \  movl $SYS_## name, %eax; \  int $T_SYSCALL; \  ret
expanded to
SYSCALL(write)
- cat.c
  - if (write(1, buf, n) != n)

- syscall.h
  - #define SYS_write 16

- usys.S
  - SYSCALL(write)
    - .globl name;
    - name:
      - movl $SYS_##name, %eax;
      - int $T_SYSCALL;
      - ret
if (write(1, buf, n) != n)

#define SYSCALL(name) \
.globl name; \
name: \
    movl $SYS_## name, %eax; \
    int $T_SYSCALL; \
ret

SYSCALL(write)
The diagram illustrates the process of a user program calling a system call (write) in a Unix-like system. The user program (cat.c) contains the function:

```c
if (write(1, buf, n) != n)
```

When called, this function will result in a system call to the kernel. The system call is defined in the header file (`sySCALL.h`) as:

```c
#define SYSCALL(name) \
.globl name; \
name: \
    movl $SYS_##name, %eax; \
    int $T_SYSCALL; \
    ret
```

This macro expands to actual assembly code (usys.S) for the system call `write`.

```
SYSCALL(write)
```

The generated assembly code in `usys.o` for the `write` system call is:

```
00000028 <write>: 
  28:b8 10 00 00 00    mov    $0x10,%eax
  2d:cd 40             int    $0x40
  2f:c3                ret
```
if (write(1, buf, n) != n)

#define SYSCALL(name) \
.globl name; \
name: \
movl $SYS_##name, %eax; \
int $T_SYSCALL; \
ret

SYSCALL(write)

定义在这里

实际生成的汇编代码在这里

```
if (write(1, buf, n) != n)

#define SYSCALL(name) \
.globl name; \
name: \ 
movl $SYS_##name, %eax; \ 
int $T_SYSCALL; \ 
ret

SYSCALL(write)
```

```
00000028 <write>: 
  28:b8 10 00 00 00 mov $0x10,%eax 
  2d:cd 40 int $0x40 
  2f:c3 ret
```
If (write(1, buf, n) != n)

Defined here

Expanded to

Actual assembly code generated here

00000028 <write>:
  28:b8 10 00 00 00 mov $0x10,%eax
  2d:cd 40 int $0x40
  2f:c3 ret

Vectors.S:
  jmp alltraps
if (write(1, buf, n) != n)

#define SYSCALL(name) \
.globl name; \
name: \
    movl $SYS_##name, %eax; \
    int $T_SYSCALL; \
    ret

usys.S
SYSCALL(write)

usys.o
00000028 <write>:
    b8 10 00 00 00    mov    $0x10,%eax
    cd 40             int    $0x40
    c3                ret

trapasm.S
jmpl alltraps

vectors.S defined in

SYSCALL defined here

actual assembly code generated here

expanded to

defined here

actual assembly code generated here
cat.c

```c
if (write(1, buf, n) != n)
```

defined here

usys.S

```
SYSCALL(write)
```

expanded to

actual assembly
code generated here

usys.o

```
00000028 <write>:
  28:b8 10 00 00 00 movl $0x10,%eax
  2d:cd 40 int $0x40
  2f:c3 ret
```

defined in

calls trap()
if (write(1, buf, n) != n)

`#define SYSCALL(name) \ 
.globl name; \ 
name: \ 
    movl $SYS_##name, %eax; \ 
    int $T_SYSCALL; \ 
    ret`

`SYSCALL(write)`

actual assembly code generated here

EAX

10

00000028 <write>:
  28:b8 10 00 00 00  mov $0x10,%eax
  2d:cd 40  int $0x40
  2f:c3  ret

calls syscall()
calls trap()
Thank you!