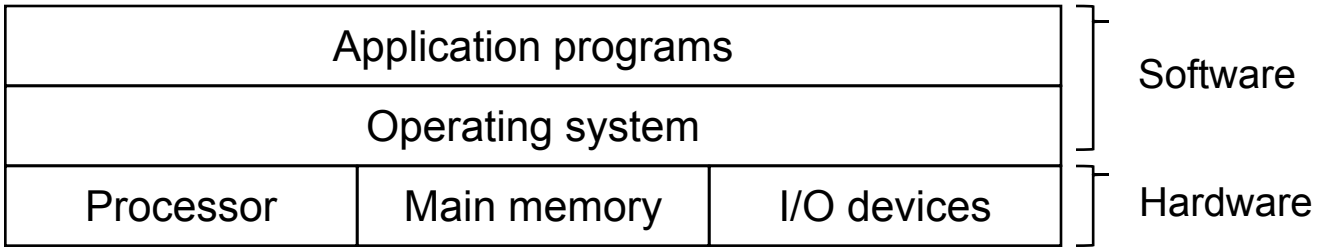
System Calls & Signals

CS449 Spring 2016

Operating system

- OS – a layer of software interposed between the application program and the hardware

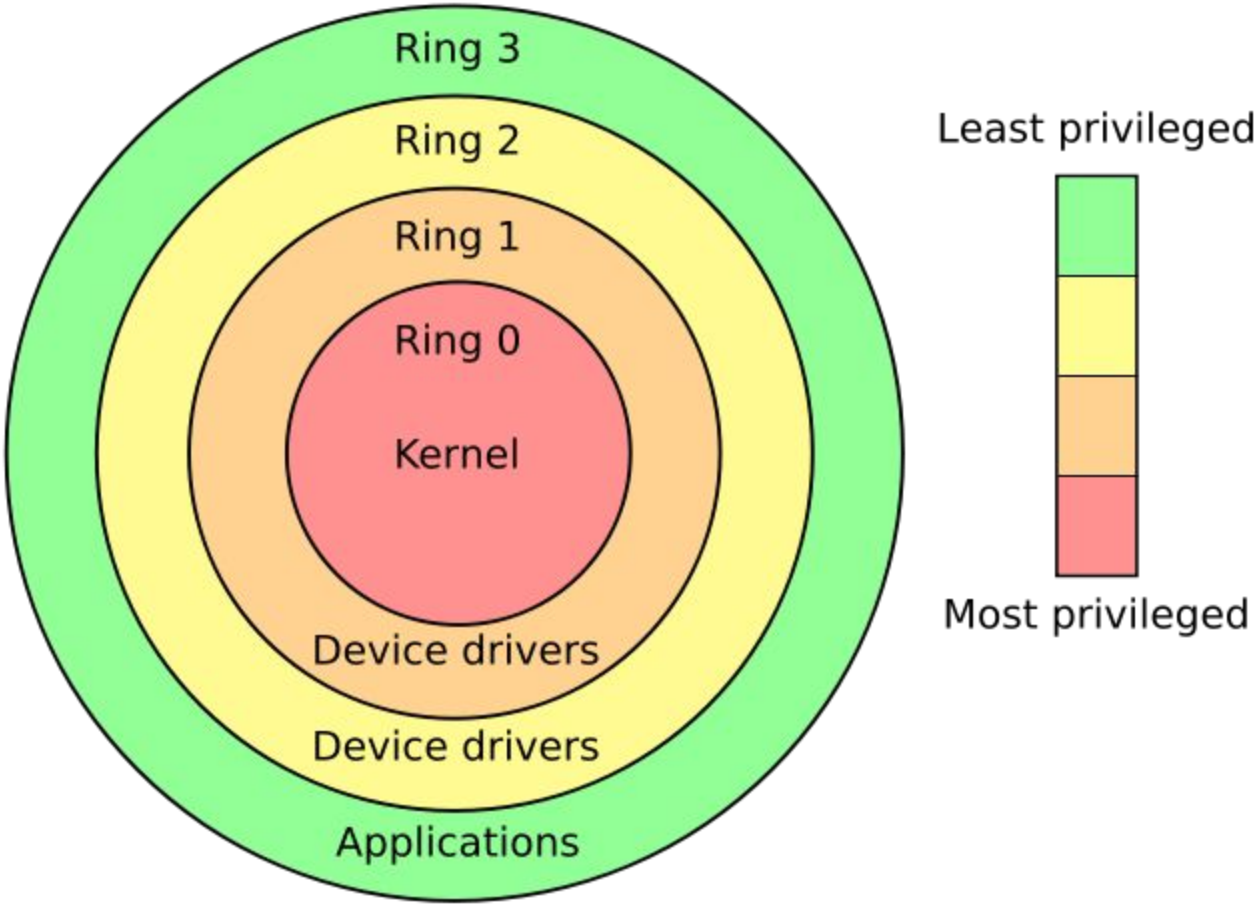


- Two primary goal
  - Manage hardware resources
  - Abstract details about hardware
    - I/O devices abstracted by files
    - Main memory abstracted by virtual memory
    - Processors abstracted by processes

What is an OS

- Pedantically it is just a Kernel
  - A program that implements the core OS functions (process scheduling, memory/HW management)
- More commonly it refers to the basic environment
  - Libraries, user interface, foundational programs
- A Kernel is a program, but it is not an application
  - Doesn't actually execute on its own (except at boot time)
  - Only reacts to events and requests
    - System calls, exceptions, and interrupts

x86 Privilege Levels



OS Interactions

- System Calls
  - Allow an application to request the OS to do something
    - Read a file, print to console, etc…
- Signals
  - OS notifies an application of something
    - SEGFAULT, Killed, etc…
- Exceptions
  - Application does something it wasn't supposed to
    - Access invalid memory address
- Interrupts
  - Processor receives a hardware signal on an interrupt pin
    - Mouse move or keyboard type

Signals

- Notifications sent to a program by OS
  - Indicate special events
- Allows for asynchronous notification rather than polling
- Polling – to explicitly ask if something occurred, usually repeatedly

kill -l

(returns a complete list of signals for your system)

SIGHUP	**SIGINT**	**SIGQUIT**	**SIGILL**	**SIGTRAP**
SIGABRT	**SIGBUS**	**SIGFPE**	**SIGKILL**	**SIGUSR1**
SIGSEGV	**SIGUSR2**	**SIGPIPE**	**SIGALRM**	**SIGTERM**
SIGCHLD	**SIGCONT**	**SIGSTOP**	**SIGTSTP**	**SIGTTIN**
SIGTTOU	**SIGURG**	**SIGXCPU**	**SIGXFSZ**	**SIGVTALRM**
SIGPROF	**SIGWINCH**	**SIGIO**	**SIGPWR**	**SIGSYS**
SIGRTMIN	SIGRTMIN+1	SIGRTMIN+2	SIGRTMIN+3	SIGRTMIN+4
SIGRTMIN+5	SIGRTMIN+6	SIGRTMIN+7	SIGRTMIN+8	SIGRTMIN+9
SIGRTMIN+10	SIGRTMIN+11	SIGRTMIN+12	SIGRTMIN+13	SIGRTMIN+14
SIGRTMIN+15	SIGRTMAX-14	SIGRTMAX-13	SIGRTMAX-12	SIGRTMAX-11
SIGRTMAX-10	SIGRTMAX-9	SIGRTMAX-8	SIGRTMAX-7	SIGRTMAX-6
SIGRTMAX-5	SIGRTMAX-4	SIGRTMAX-3	SIGRTMAX-2	SIGRTMAX-1
SIGRTMAX				

Common Error Signals

- **SIGILL** – Illegal Instruction
- **SIGBUS** – Bus Error, usually caused by bad data alignment or a bad address
- **SIGFPE** – Floating Point Exception
- **SIGSEGV** – Segmentation violation, i.e., a bad address

Termination Signals

- **SIGINT** – Interrupt, or what happens when you hit CTRL + C
- **SIGTERM** – Ask nicely for a program to end (can be caught)
- **SIGKILL** – Ask meanly for a program to end (cannot be caught)
- **SIGABRT**, **SIGQUIT** – End a program with a core dump

kill

- From the shell in UNIX you can send signals to a program.
- Use ps to get a process ID

```
(1) thot $ ps -af
UID        PID  PPID  C STIME TTY          TIME CMD
jrmst106 27500 27470  0 07:13 ???      00:00:00 crashed_program
jrmst106 27507 27474  0 07:13 pts/5    00:00:00 ps -af
```

- kill it!

`kill 27500` – Sends SIGTERM

`kill -9 27500` – Sends SIGKILL

kill

- kill() is the system call that can send a process a signal (any signal, not just SIGKILL)

```
#include <unistd.h>
#include <sys/types.h>
#include <signal.h>

int main() {
    pid_t my_pid = getpid();
    kill(my_pid, SIGSTOP);
    return 0;
}
```

Catching Signals

- Most signals can be caught
  - Like exceptions in Java
- Do some cleanup, then exit
- Generally bad to try to continue, the machine might be in a corrupt state
- Some signals can be caught safely though

SIGALRM

```c
#include <unistd.h>
#include <signal.h>

int timer = 10;

void catch_alarm(int sig_num) {
        printf("%d\n",timer--);
        alarm(1);
}

int main() {
        signal(SIGALRM, catch_alarm);

        alarm(1);
        while(timer > 0) ;
        alarm(0);
        return 0;
}
```

```
>> ./a.out
10
9
8
7
6
5
4
3
2
1
```

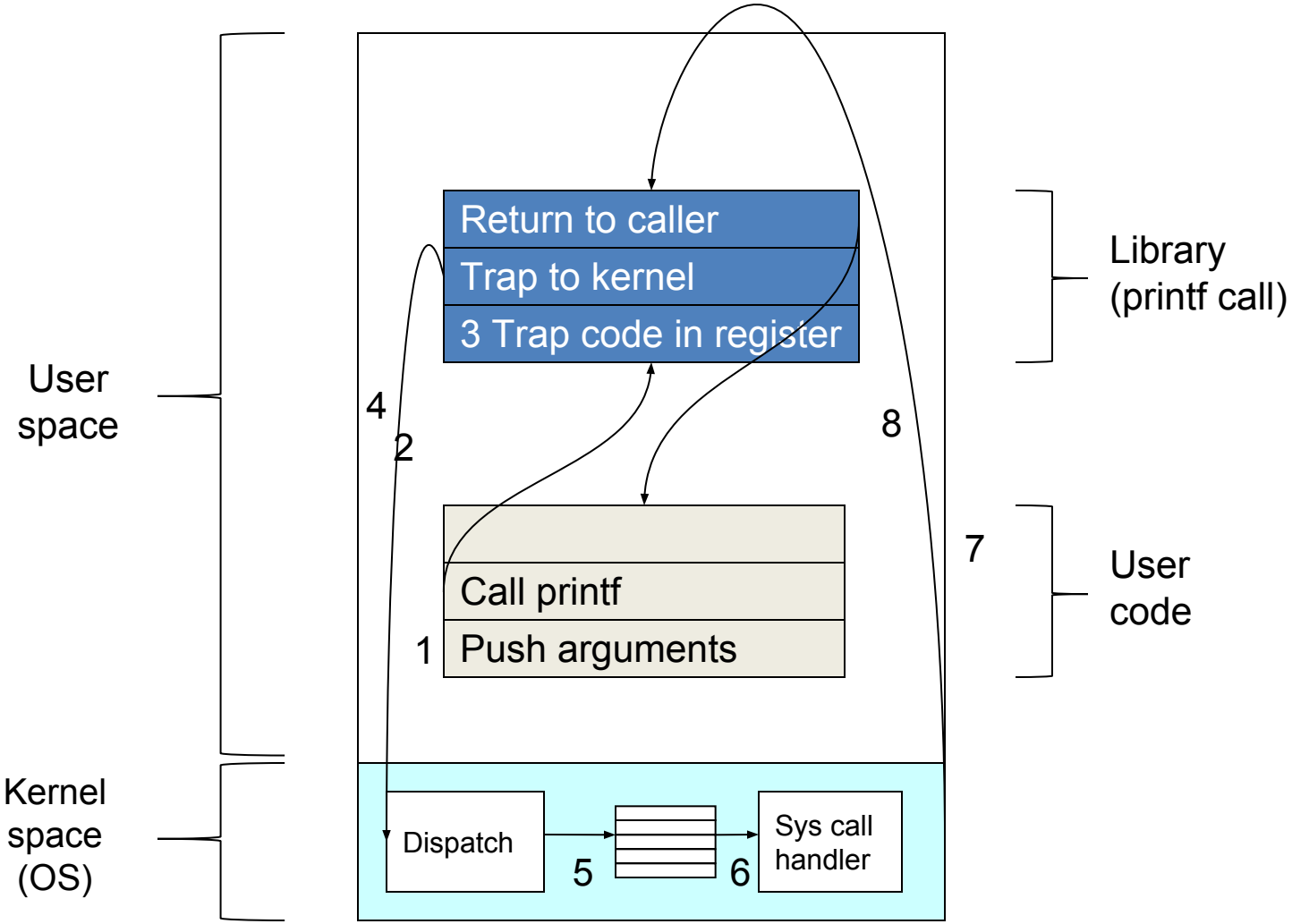
Exceptions

- You've seen these in Java
  - Exceptions are generated by unexpected errors
- Hardware exceptions exist for the same purpose
  - Hardware has to tell OS that a program tried to do something illegal
    - Executed an illegal instruction, violated memory protection
  - OS then has to respond to the violation
    - Handle the exception let program continue to run (e.g. page fault)
      - Page Fault: exception that happens when accessing page in disk
    - Notify the program that something happened (via signal)
    - Kill the program

Exception Handling

- Each exception has an entry in a dispatch table
  - An array of function pointers to specific handlers
    - Interrupt Vector Table
  - Appropriate privilege given on jump to handler
  - Registers must be saved to/restored from kernel stack
- Hardware interrupts handled by same dispatch table
- System calls often implemented using exceptions
  - x86 versions uses a trap instruction: int 0x80
    - Trap: an intentional exception to invoke a specific handler
    - Jumps to entry 0x80 in dispatch table

System Call


User space
Kernel space (OS)
Return to caller
Trap to kernel
3 Trap code in register
Call printf
Push arguments
Dispatch
Sys call handler
1
2
4
5
6
7
8
Library (printf call)
User code

strace ./hello

```
fstat(1, {st_mode=S_IFCHR|0600, st_rdev=makedev
  (136, 7), ...}) = 0

mmap(NULL, 4096, PROT_READ|PROT_WRITE,
  MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) =
  0x2a95557000

write(1, "Hello world!\n", 13Hello world!) = 13

exit_group(0)
```

Linux Syscalls

- 325 syscall slots reserved in system call table (2.6.23.1 kernel) -- Not all are used

exit	Causes a process to terminate
fork	Creates a new process, identical to the current one
read	Reads data from a file or device
write	Writes data to a file or device
open	Opens a file
close	Closes a file
creat	Creates a file

Using Syscalls

```c
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>

int main()
{
        int fd;
        char buffer[100];
        strcpy(buffer, "Hello, World!\n");

        fd = open("hello.txt", O_WRONLY | O_CREAT);
        write(fd, buffer, strlen(buffer));
        close(fd);
        exit(0);

        return 0;
}
```

OR-ing Flags

- Define constants as powers of 2
- Bitwise OR to combine
- Bitwise AND to test

```
#define O_RDONLY        0
#define O_WRONLY        1
#define O_RDWR          2
#define O_CREAT        16
```

File Descriptors

- Integer identifying a unique open file
  - Similar to FILE *
  - FILE struct has a file descriptor member
- File descriptors index into a file descriptor table
- OS maintains additional information about the file to do things such as clean up on process termination
- Three standard file descriptors opened automatically:

  0 – stdin
  1 – stdout
  2 – stderr

Dumping the File Descriptor Table

- ls -l /proc/self/fd (listing default file descriptors)

```
(84) thot $ ls -l /proc/self/fd
total 0
lrwx------ 1 wahn UNKNOWN1 64 Sep 11 13:37 0 -> /dev/pts/0
lrwx------ 1 wahn UNKNOWN1 64 Sep 11 13:37 1 -> /dev/pts/0
lrwx------ 1 wahn UNKNOWN1 64 Sep 11 13:37 2 -> /dev/pts/0
lr-x------ 1 wahn UNKNOWN1 64 Sep 11 13:37 3 -> /proc/16970/fd
```

- ls -l /proc/self/fd > /tmp/ls.out (listing with redirect)

```
(85) thot $ ls -l /proc/self/fd > /tmp/ls.out && cat /tmp/ls.out
total 0
lrwx------ 1 wahn UNKNOWN1 64 Sep 11 13:41 0 -> /dev/pts/0
l-wx------ 1 wahn UNKNOWN1 64 Sep 11 13:41 1 -> /tmp/ls.out
lrwx------ 1 wahn UNKNOWN1 64 Sep 11 13:41 2 -> /dev/pts/0
lr-x------ 1 wahn UNKNOWN1 64 Sep 11 13:41 3 -> /proc/17025/fd
```