CSCI 6411: Operating Systems

Acknowledgements: Some slide material derived from Silberschatz, et al.
“High-level”

Cars

Computers
Cars

Computers

/**
* So far all flags should be taken in the context of the
* actual invoking thread (they affect the thread switching
* _from_, rather than the thread to switch _to_) in which case
* we would want to use the sched_page flags.
*/
flags = rflags;
switch_thread_update_flags(da, &flags);

if (unlikely(flags)) {
    thd = switch_thread_slowpath(curr, flags, curr_spd, rthd_id, da, &ret_c
&curr_sched_flags, &thd sched_flags);
    /* If we should return immediately back to this
    * thread, and its registers have been changed,
    * return without setting the return value */
    if (ret_code == COG_SCHED_RET_SUCCESS && thd == curr) goto ret;
    if (thd == curr) goto_err(ret_err, "sloppown");
} else {
    next_thd = switch_thread_parse_data_area(da, &ret_code);
    if (unlikely(0 == next_thd)) goto_err(ret_err, "data area\n");
    thd = switch_thread_get_target(next_thd, curr, curr_spd, &ret_code);
    if (unlikely(NULL == thd)) goto_err(ret_err, "get target");
}

/**
* If a thread is involved in a scheduling decision, we should
* assume that any preemption chains that existed aren’t valid
* anymore. */
break_preemption_chain(curr);
...“low-level”...
What is an Operating System!??
What is an OS: Where is it?

- Applications
  (excel, word, browser, ...)

- Operating Systems

- Hardware
  (CPU, memory, hard drive)
  "things you can kick"
What is an OS: Where is it?

- Applications
  (excel, word, browser, ...)
- Operating Systems
- Hardware
  (CPU, memory, hard drive)
  “things you can kick”
What is an OS: Analogy

- You!
- Customer₁
- Customer₂
- Customerₙ
What is an OS: Analogy

You!
Customer
Customer
Customer
Customer

You! → Customer
Customer → Customer
Customer → Customer
Customer → Customer

Customer → You!
What is an OS: Analogy

Hardware

Operating System

Applications

You!

Customer$_1$

Customer$_2$

Customer$_n$
Operating System as Abstraction

- "The effective exploitation of his powers of abstraction must be regarded as one of the most vital activities of a competent programmer." - Edsger W. Dijkstra

- Provides abstractions for resources (memory, CPU, disk) and controls application execution
- Provide environment for application execution
  - Each application can pretend like it is using the entire computer!
- Allow users to translate intentions into actions
- Aside: Edsger Dijkstra - Discipline in Thought
OS as Abstraction: System Layers
AN x64 PROCESSOR IS SCREAMING ALONG AT BILLIONS OF CYCLES PER SECOND TO RUN THE XNU KERNEL, WHICH IS FRANTICALLY WORKING THROUGH ALL THE POSIX-SPECIFIED ABSTRACTION TO CREATE THE DARWIN SYSTEM UNDERLYING OS X, WHICH IN TURN IS STRAINING ITSELF TO RUN FIREFOX AND ITS GECKO RENDERER, WHICH CREATES A FLASH OBJECT WHICH RENDERS DOZENS OF VIDEO FRAMES EVERY SECOND

BECAUSE I WANTED TO SEE A CAT JUMP INTO A BOX AND FALL OVER.

I AM A GOD.
Computers as Distributed Systems

“Hardware: The parts of a computer system that can be kicked.”
- Jeff Pesis
OS as Hardware Manager

- Control a diverse set of hardware
  - Processors
  - Memory
  - Disks
  - Networking cards
  - Video cards
- Coordinates these hardware resources amongst user programs
- OS as a resource manager/multiplexer
OS Services

Applications
(excel, word, browser, ...)

Hardware
(CPU, memory, hard drive)
“things you can kick”
Interrupts, exceptions, and traps – OH MY

- Interrupts thus far: Device ↔ kernel
- Software-triggered events
  - Application state saved (as for interrupt) and can be resumed
- Exceptions
  - Program faults (divide by zero, general protection fault, segmentation fault)
  - Not requested by executing application
- Traps/Software Interrupts
  - Requested by application by executing specific instruction: `sysenter` or `int %d` on x86
MSDOS: No Structure/Protection

Diagram:
- Application program
- Resident system program
- MS-DOS device drivers
- ROM BIOS device drivers
System Calls

- Wait, hardware support for calling the kernel?
- Why can't I just call it directly (function call)?
Timesharing systems: 1) protection applications from each other, and 2) kernel from applications (why the latter?)

- Mode bit == 0
  - Access kernel memory segments
  - Protected instructions
    - Access I/O: instructions to read/write to device control registers (in/out on x86)
  - Sensitive instructions
  - *What happens to the registers, and stack?*
Syscall Mechanics

printf(“print me!”)

➔ write(1, “print me!”)

➔ put syscall number for write (4), file descriptor (1), and pointer to “print me!” into registers

➔ sysenter: mode bit = 0
  ➔ Change to kernel stack

➔ Call address in syscall tbl at index 4

➔ Execute write system call

➔ sysexit: mode bit = 1
  ➔ Restore application registers

```c
#include <stdio.h>
int main ( )
{
    ...
    printf ("Greetings");
    ...
    return 0;
}
```
Abstraction for syscalls: APIs

- Application Programmer Interfaces (APIs)
  - Hide the details of how a syscall is carried out
  - POSIX (UNIX, Linux)
  - Win32 (Windows)
  - .Net (Windows XP and later)
  - Cocoa (OS X)
System Structure

- **System Structure** – How different parts of software
  1) Are separated from each other (*Why?*)
  2) Communicate
- How does a system use
  - dual mode
  - *virtual address spaces*
- Implications on
  - Security/Reliability
  - Programming style/Maintainability
Monolithic System Structure

- Includes Unix/Windows/OSX

```
Word
Excel
Browser
```

```
Operating System
(File System, Networking,
Memory Mgmt, Scheduling, ...)
```

```
Kernel
- most trusted
- must work
```

```
Hardware (CPU, Memory, Peripherals
– hard drive, NIC, GPU)
```

```
System Call
open(...)```

"User-Level"
Monolithic System Structure

- Includes Unix/Windows/OSX

When's the last time you tried to get **50 MLOC** to work???

- hard drive, NIC, GPU

ʻvel’

dusted work
Microkernel System Structure

- Moves functionality from the kernel to “user” space
- Communication takes place between user servers using inter-process communication (IPC)
- Benefits:
  - Easier to add functionality
  - More reliable (less code is running in kernel mode)
  - More secure
- Detriments: performance! (why?)
Processes

- An executable program (seen in *ls*)
- passive collection of code and data; kept in file
- UNIX Process: active entity that includes (seen in *ps*)
  - Registers (instruction counter, stack pointer, etc..)
  - Execution stack
  - Heap
  - Data and text (code) segments
char *s = malloc(10);
printf("stk: %p\n", &s);
printf("malloc: %p\n", s);
printf("global: %p\n", &global_variable);
printf("fn: %p\n", main);

stk: 0xBC87B240
malloc: 0x1070B4F0
global: 0x004C72A0
fn: 0x000BC240
OS Support for Process Memory

- OS uses HW to provide virtual address space (VAS)
  - Each process thinks it has all memory
    - OS abstraction!!!
  - Provides protection between processes
  - Only subset of that address space is populated by actual memory
OS Support for Process Memory II

- Kernel must manage virtual address spaces
  - Create mapping between virtual and actual memory
  - Switch between apps == switch between VAS
  - Only mode 0 can switch VAS!
Process Control Block (PCB)

- Kernel, per-process, data-structure includes:
  - CPU registers (including instruction counter)
  - Scheduling state (priority)
  - Memory management information (amount of memory allocated, virtual address space mapping, stack location)
  - CPU accounting info (exec time at user/kernel level)
  - File info (open files)
  - Process state