Syscalls, exceptions, and interrupts, …oh my!

CS 3410
Computer System Organization & Programming

[D. Altinbuken, K. Bala, A. Bracy, E. Sirer, and H. Weatherspoon]
Which of the following is **not** a viable solution to protect against a buffer overflow attack? (There are *multiple* right & wrong answers. Pick 1 right one.)

(A) Prohibit the execution of anything stored on the Stack.
(B) Randomize the starting location of the Stack.
(C) Use only library code that requires a buffer length to make sure it doesn’t overflow.
(D) Write only to buffers on the OS Stack where they will be protected.
(E) Compile the executable with the highest level of optimization flags.
November 1988: Internet Worm
Internet Worm attacks thousands of Internet hosts

Best Wikipedia quotes:

“According to its creator, the Morris worm was not written to cause damage, but to gauge the size of the Internet. The worm was released from MIT to disguise the fact that the worm originally came from Cornell.”

“The worm …determined whether to invade a new computer by asking whether there was already a copy running. But just doing this would have made it trivially easy to kill: everyone could run a process that would always answer "yes". To compensate for this possibility, Morris directed the worm to copy itself even if the response is "yes" 1 out of 7 times. This level of replication proved excessive, and the worm spread rapidly, infecting some computers multiple times. Morris remarked, when he heard of the mistake, that he "should have tried it on a simulator first".”

Computer Virus TV News Report 1988
Operating System

- Manages all of the software and hardware on the computer
- Many processes running at the same time, requiring resources
  - CPU, Memory, Storage, etc.

OS multiplexes these resources amongst different processes, and isolates and protects processes from one another!
Operating System

Operating System (OS) is a trusted mediator:

- *Safe control transfer between processes*
- *Isolation (memory, registers) of processes*

![Diagram showing trusted and untrusted processes, with components such as OS, MMU, CPU, disk, network card, VM, filesystem, net, driver, and P1, P2, P3, P4.]
One Brain, Many Personalities

You are what you execute.

Personalities:
- hailstone_recurisve
- Microsoft Word
- Minecraft
- Linux ← yes, this is just software like every other program that runs on the CPU

Are they all equal?
Trusted vs. Untrusted

• Only **trusted** processes should access & change important things
  • Editing TLB, Page Tables, OS code, OS $sp, OS $fp...

• If an **untrusted** process could change the OS’ $sp/$fp/$gp/etc., OS would crash!
Privileged Mode

**CPU Mode Bit in Process Status Register**

- Many bits about the current process (Mode bit is just one of them)

0 = user mode = untrusted
   - “Privileged” instructions and registers are disabled by CPU

1 = kernel mode = trusted
   - All instructions and registers are enabled
# MIPS Privileged Instructions

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>CACHE</td>
<td>Perform Cache Operation</td>
</tr>
<tr>
<td>ERET</td>
<td>Exception Return</td>
</tr>
<tr>
<td>MFC0</td>
<td>Move from Coprocessor 0</td>
</tr>
<tr>
<td>MTC0</td>
<td>Move to Coprocessor 0</td>
</tr>
<tr>
<td>TLBP</td>
<td>Probe TLB for Matching Entry</td>
</tr>
<tr>
<td>TLBR</td>
<td>Read Indexed TLB Entry</td>
</tr>
<tr>
<td>TLBWI</td>
<td>Write Indexed TLB Entry</td>
</tr>
<tr>
<td>TLBWR</td>
<td>Write Random TLB Entry</td>
</tr>
<tr>
<td>WAIT</td>
<td>Enter Standby Mode</td>
</tr>
</tbody>
</table>
Privileged Mode at Startup

1. **Boot sequence**
   - load first sector of disk (containing OS code) to predetermined address in memory
   - Mode ← 1; PC ← predetermined address

2. **OS takes over**
   - initializes devices, MMU, timers, etc.
   - loads programs from disk, sets up page tables, etc.
   - Mode ← 0; PC ← program entry point
     - User programs regularly yield control back to OS
Users need access to resources

If an untrusted process does not have privileges to use system resources, how can it
  • Use the screen to print?
  • Send message on the network?
  • Allocate pages?
  • Schedule processes?

Solution: System Calls
System Call Examples

`putc()`: print character to screen
  • Need to multiplex screen between competing processes

`send()`: send a packet on the network
  • Need to manipulate the internals of a device

`sbrk()`: allocate a page
  • Needs to update page tables & MMU

`sleeep()`: put current program to sleep, wake another
  • Need to update page table base register
System Calls

System call: **not** just a function call

- Don’t let process jump just anywhere in OS code
- OS can’t trust process’ registers (sp, fp, gp, etc.)

**SYSCALL insn**: safe control transfer to OS

MIPS system call convention:

- Exception handler saves temp regs, saves ra, …
- $v0 = system call number, which specifies the operation the application is requesting
Libraries and Wrappers

Compilers do not emit SYSCALL instructions
  • Compiler doesn’t know OS interface
Libraries implement standard API from system API
libc (standard C library):
  • gets() → getc()
  • getc() → syscall
  • sbrk() → syscall
  • printf() → write()
  • write() → syscall
  • malloc() → sbrk()
  • …
Invoking System Calls

```c
char *gets(char *buf) {
    while (...) {
        buf[i] = getc();
    }
}

int getc() {
    asm("addiu $v0, $0, 4");
    asm("syscall");
}
```
Anatomy of a Process, v1

- 0xfffffffffc: system reserved
- 0x80000000: stack
- 0x7fffffff: dynamic data (heap)
- 0x10000000: static data
- 0x00400000: code (text)
- 0x00000000: system reserved

??

getc
[library] getc
[user] gets
Clicker Questions

Where are the following program components located?

A. System Reserved  
B. Stack  
C. Heap  
D. Data  
E. Text

1) P1  
2) the address that p1 points to  
3) malloc()  
4) main()  
5) beyond  
6) big_array

```c
char big_array[1<<24];
char *p1, *p2;

int main()
{
    int beyond;

    p1 = malloc(1 << 28);
    p2 = malloc(1 << 8);
}
```
Where does the OS live?

In its own address space?

- Syscall has to switch to a different address space
- Hard to support syscall arguments passed as pointers

... So, NOPE

In the same address space as the user process?

- Protection bits prevent user code from writing kernel
- Higher part of virtual memory
- Lower part of physical memory

... Yes, *this is how we do it.*
Full System Layout

All kernel text & most data:
• At same virtual address in every address space

OS is omnipresent, available to help user-level applications
• Typically in high memory
Full System Layout

Virtual Memory

0xffffffffc
OS Stack
OS Heap
OS Data
OS Text

0x80000000
stack

0x7fffffff
static data
dynamic data (heap)

0x10000000
code (text)

0x00400000
system reserved

0x00000000

0x00...00

Physical Memory

OS Stack
OS Heap
OS Data
OS Text
Anatomy of a Process, v2

- Code (text) at 0x10000000
- Static data at 0x00400000
- Dynamic data (heap) at 0x7fffffff
- Stack at 0xffffffffc
- System reserved at 0x00000000
- Implementation of `getc()` syscall

The diagram illustrates the memory layout of a process, showing the allocation of various regions such as code, static data, dynamic data, stack, and system reserved areas.
Clicker Question

Which statement is FALSE?

A) OS manages the CPU, Memory, Devices, and Storage.
B) OS provides a consistent API to be used by other processes.
C) The OS kernel is always present on Disk.
D) The OS kernel is always present in Memory.
E) Any process can fetch and execute OS code in user mode.
Clicker Question

Which one of the following statements is true?

A. Multiple copies of OS code reside in physical memory because every process keeps a copy of the kernel in its reserved address space.
B. A programmer can invoke the operating system by using an instruction that will trigger an interrupt.
C. The OS uses its own stack when executing a system call on behalf of user code.
D. The OS can interrupt user code via a system call.
E. The OS is always actively running on the CPU.
Inside the SYSCALL instruction

SYSCALL instruction does an atomic jump to a controlled location (i.e., MIPS 0x8000 0180)

- Saves the old (user) SP value
- Switches the SP to the kernel stack
- Saves the old (user) PC value (= return addr)
- Saves the old privilege mode
- Sets the new privilege mode to 1
- Sets the new PC to the kernel syscall handler
Inside the SYSCALL implementation

Kernel system call handler carries out the desired system call

- Saves callee-save registers
- Examines the syscall number
- Checks arguments for sanity
- Performs operation
- Stores result in v0
- Restores callee-save registers
- Performs a "return from syscall" (ERET) instruction, which restores the privilege mode, SP and PC
Exceptional Control Flow

Anything that *isn’t* a user program executing its own user-level instructions.

**System Calls:**
- just one type of exceptional control flow
- Process requesting a service from the OS
- Intentional – *it’s in the executable!*
Software Exceptions

Trap
Intentional
Examples:
System call
(OS performs service)
Breakpoint traps
Privileged instructions

Fault
Unintentional but
Possibly recoverable
Examples:
Division by zero
Page fault

Abort
Unintentional
Not recoverable
Examples:
Parity error

One of many ontology / terminology trees
Hardware support for exceptions

Exception program counter (EPC)
- 32-bit register, holds addr of affected instruction
- Syscall case: Address of SYSCALL

Cause register
- Register to hold the cause of the exception
- Syscall case: 8, Sys

Special instructions to load TLB
- Only do-able by kernel
Exceptional Control Flow

Hardware interrupts
- Asynchronous
  - caused by events external to CPU

Software exceptions
- Synchronous
  - caused by CPU executing an instruction

Maskable
- Can be turned off by CPU
  - Example: alert from network device that a packet just arrived, clock notifying CPU of clock tick

Unmaskable
- Cannot be ignored
  - Example: alert from the power supply that electricity is about to go out
Clicker Q
Which sequence best describes a:

1) System Call
2) Page Fault
3) Interrupt

Sequence A:
- current instruction (at PC) triggers handler
- control passes to handler
- execution returns to user process
- current instruction (at PC) executes once more

Sequence B:
- current instruction (at PC) triggers handler
- control passes to handler
- execution returns to user process
- next instruction (at PC+4) executes

Sequence C:
- current instruction (at PC) completes
- control passes to handler
- execution returns to user process
- next instruction (at PC+4) executes

Sequence D:
- current instruction (at PC) triggers handler
- control passes to handler
- execution never returns to the user process
Interrupts & Unanticipated Exceptions

No **SYSCALL** instruction. **Hardware** steps in:

- Saves PC of exception instruction (EPC)
- Saves cause of the interrupt/privilege (Cause register)
- Switches the sp to the kernel stack
- Saves the old (user) SP value
- Saves the old (user) PC value
- Saves the old privilege mode
- Sets the new privilege mode to 1
- Sets the new PC to the kernel **syscall hander** interrupt/exception handler
Inside Interrupts & Unanticipated Exceptions

**Kernel system call handler** carries out system call all

- Saves callee-save registers
- Examines the syscall number cause
- Checks arguments for sanity
- Performs operation
- Stores result in v0
- Restores callee-save registers
- Performs a ERET instruction (restores the privilege mode, SP and PC)
Clicker Question

What other task requires both Hardware and Software?

A) Virtual to Physical Address Translation
B) Branching and Jumping
C) Clearing the contents of a register
D) Pipelining instructions in the CPU
E) What are we even talking about?
Address Translation: HW/SW Division of Labor

Virtual → physical address translation!

**Hardware**
- has a concept of operating in physical or virtual mode
- helps manage the TLB
- raises page faults
- keeps Page Table Base Register (PTBR) and ProcessID

**Software/OS**
- manages Page Table storage
- handles Page Faults
- updates Dirty and Reference bits in the Page Tables
- keeps TLB valid on context switch:
  - Flush TLB when new process runs (x86)
  - Store process id (MIPS)
Demand Paging on MIPS

1. TLB miss
2. Trap to kernel
3. Walk Page Table
4. Find page is invalid
5. Convert virtual address to file + offset
6. Allocate page frame
   • Evict page if needed
7. Initiate disk block read into page frame
8. Disk interrupt when DMA complete
9. Mark page as valid
10. Load TLB entry
11. Resume process at faulting instruction
12. Execute instruction