

# Surgical Recovery from Kernel-Level Rootkit Installations

**Linux Based Systems**

Julian Grizzard

July 2005

# Latest Slides and Tools

PLEASE DOWNLOAD THE LATEST SLIDES AND TOOLS

[ Latest slides available ]

[http://www.ece.gatech.edu/research/labs/nsa/presentations/dc13\\_grizzard.pdf](http://www.ece.gatech.edu/research/labs/nsa/presentations/dc13_grizzard.pdf)

[ Latest system call table tools ]

[http://www.ece.gatech.edu/research/labs/nsa/sct\\_tools.shtml](http://www.ece.gatech.edu/research/labs/nsa/sct_tools.shtml)

[ Latest spine architecture work ]

<http://www.ece.gatech.edu/research/labs/nsa/spine.shtml>

# Problem

## What does a rootkit do?

- **Retain Access**

- Trojan sshd client with hard coded user/pass for root access
- Initiate remote entry by specially crafted packet stream

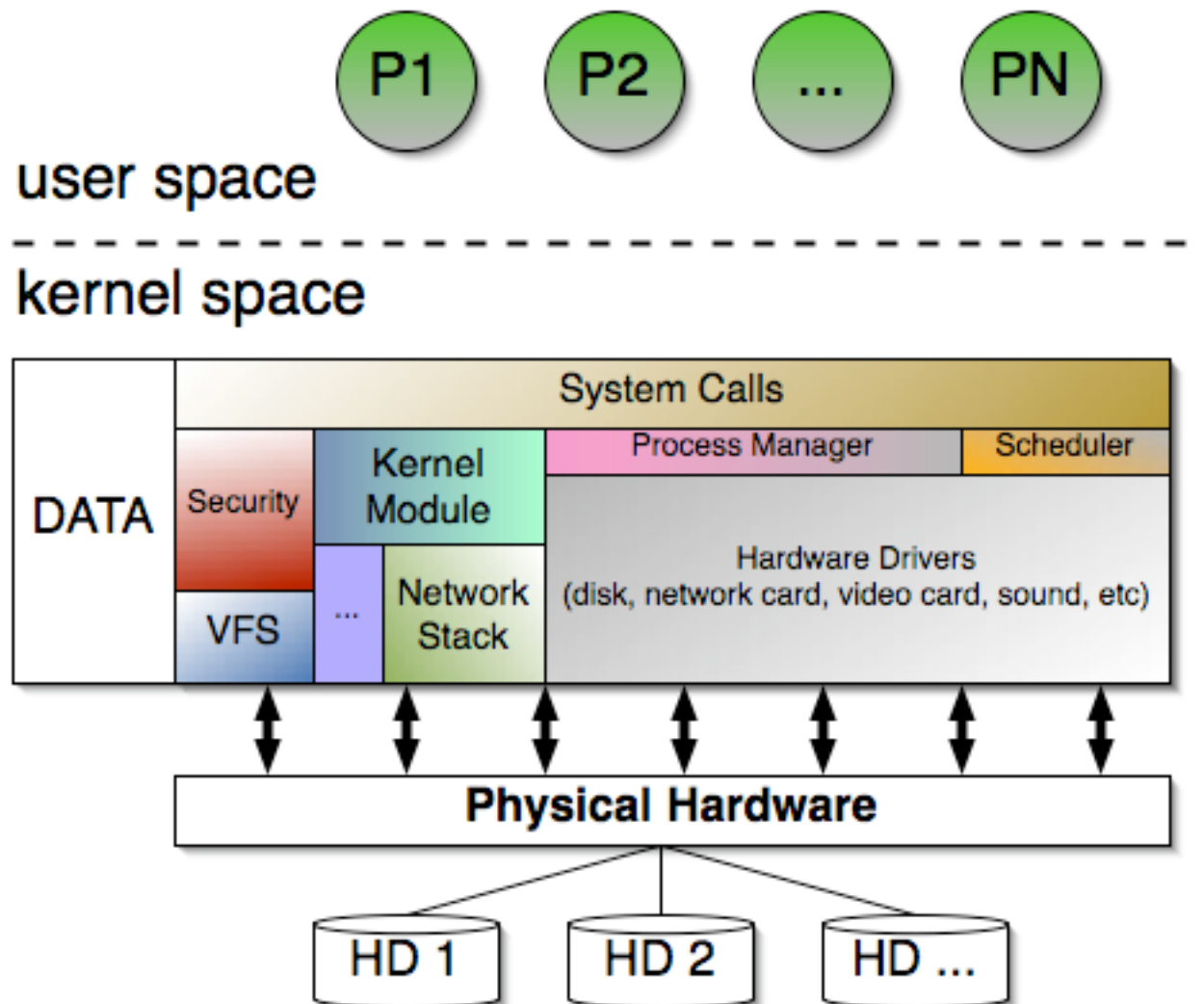
- **Hide Activity**

- Hide a process including resource usage of process
- Hide malicious rootkit kernel modules from lsmod

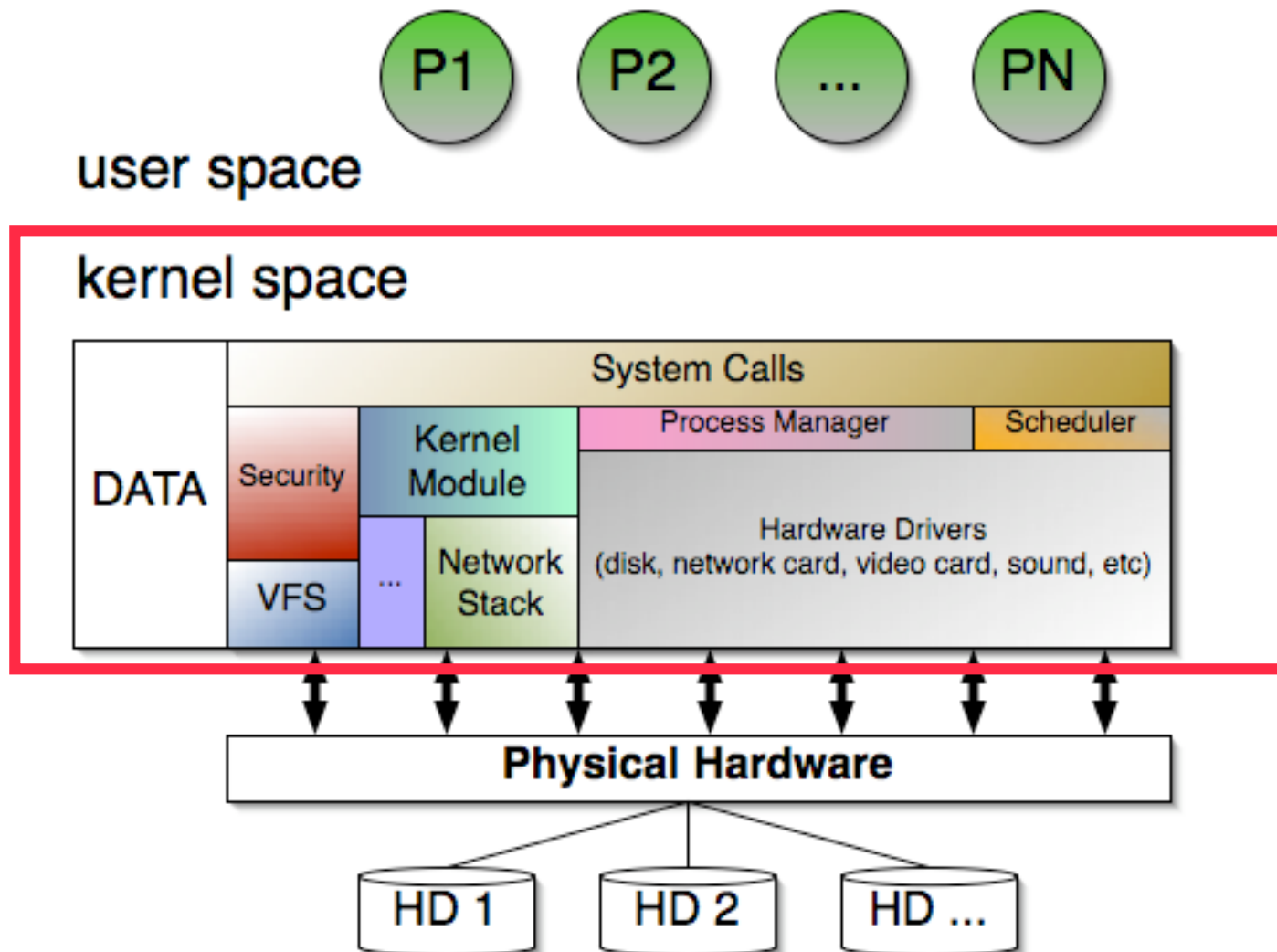
# Most Widely Accepted Solution

## Format and Reinstall

# Monolithic Operating System



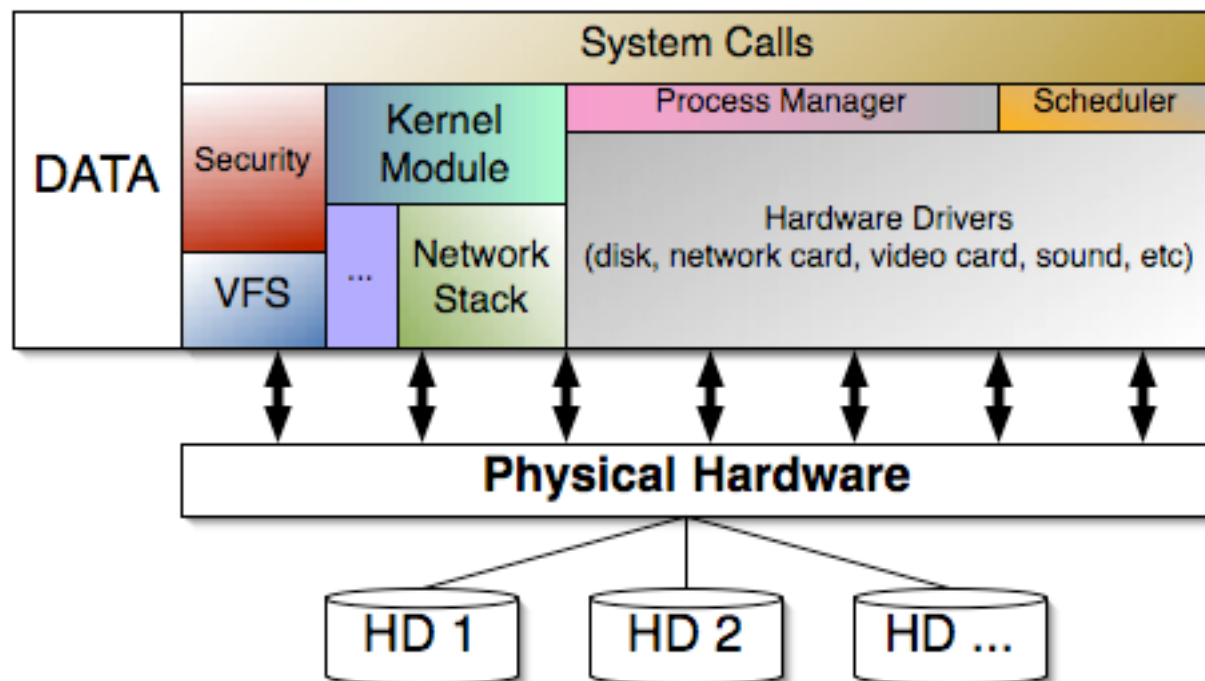
# Kernel Space



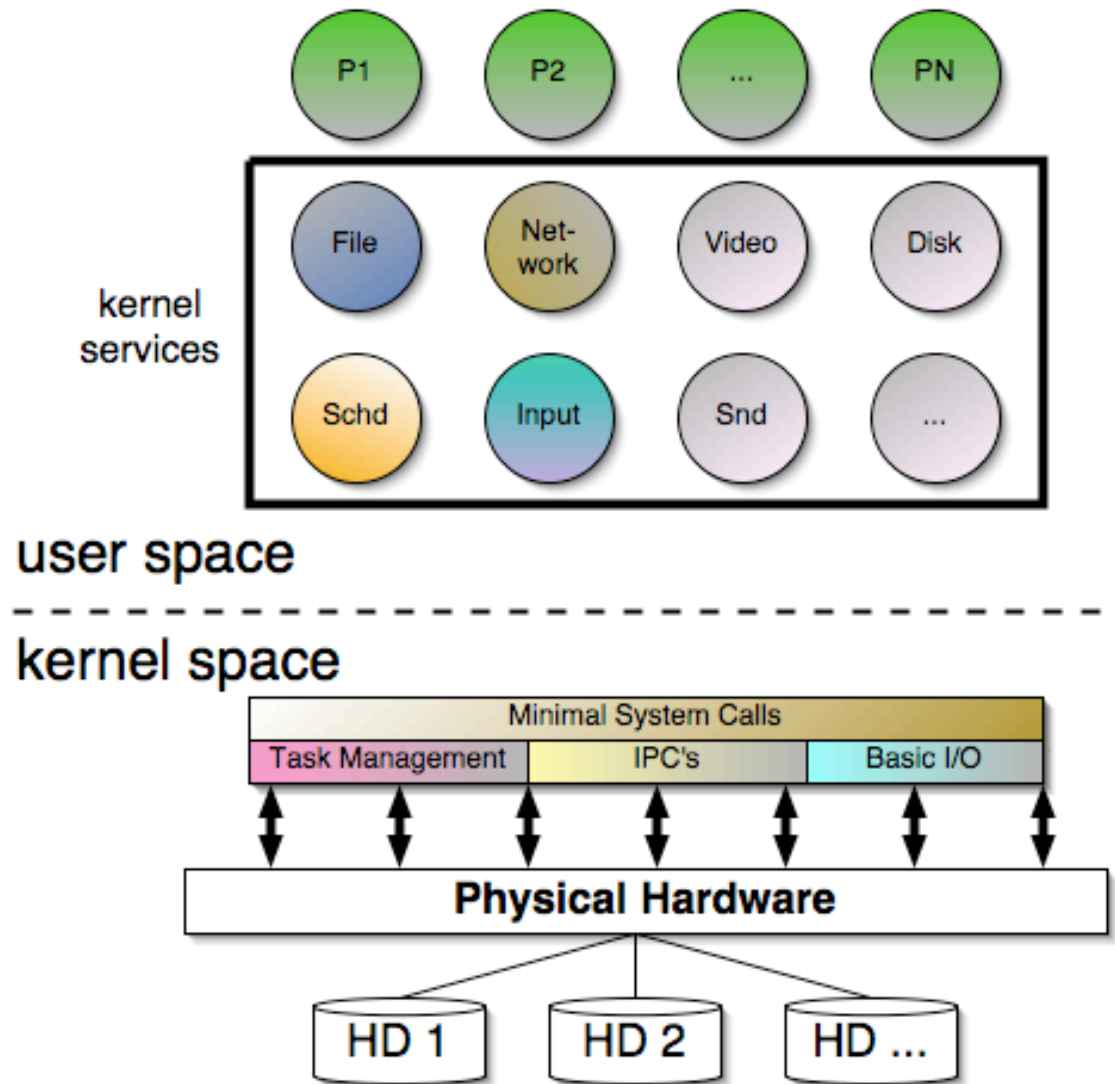
# User Space



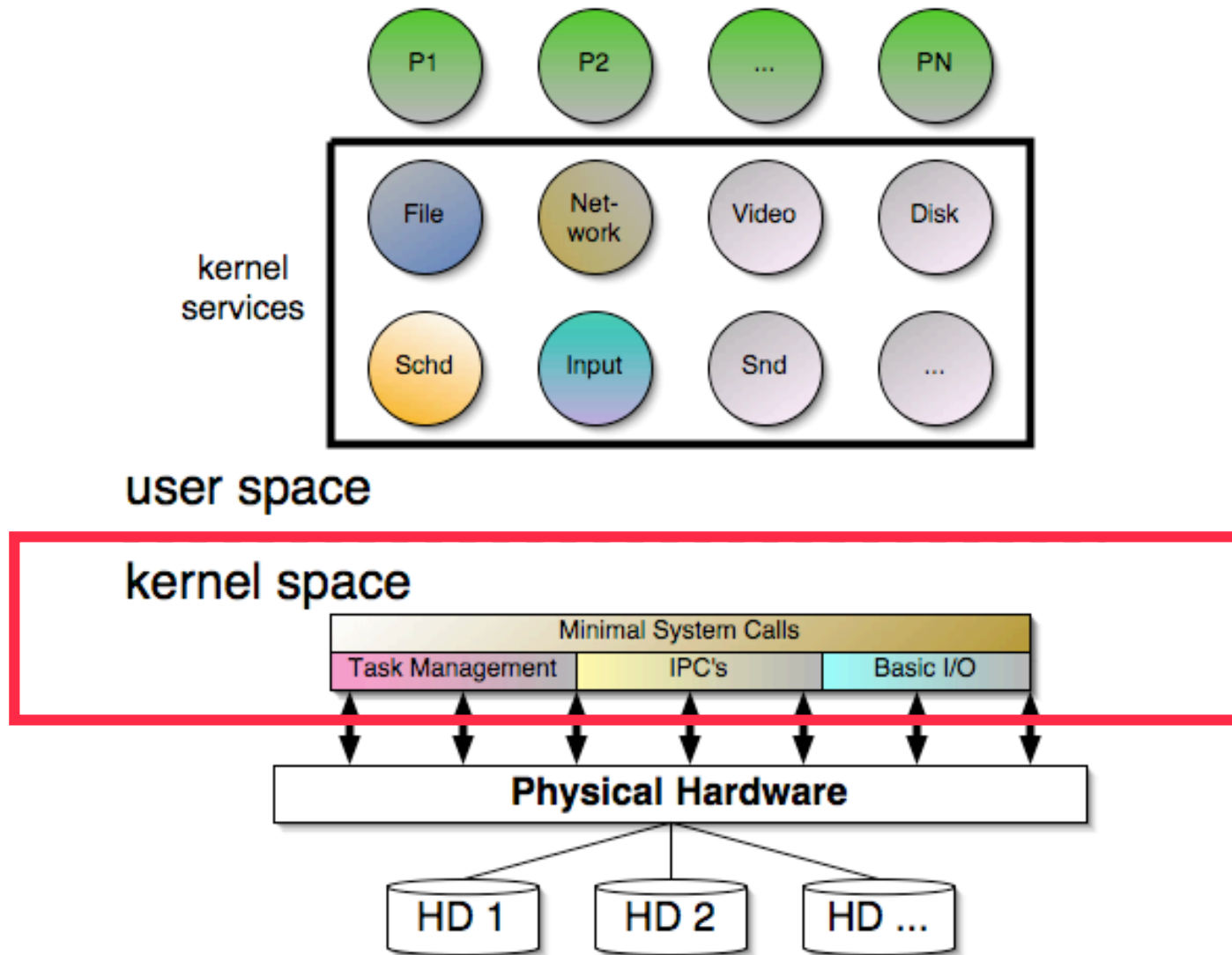
kernel space



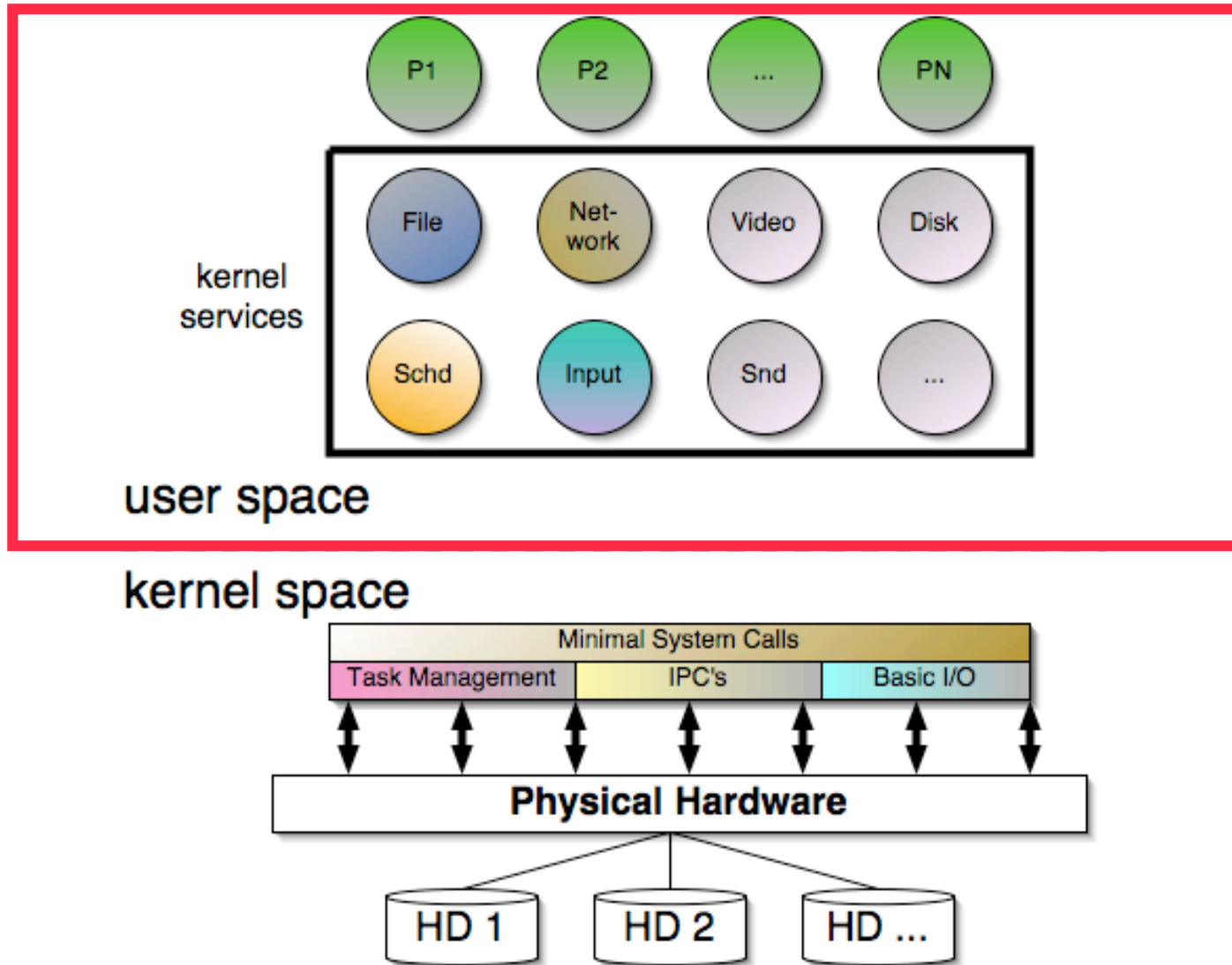
# Microkernel Operating System



# Microkernel Operating System



# Microkernel Operating System



# Intel Descriptor Privilege Level

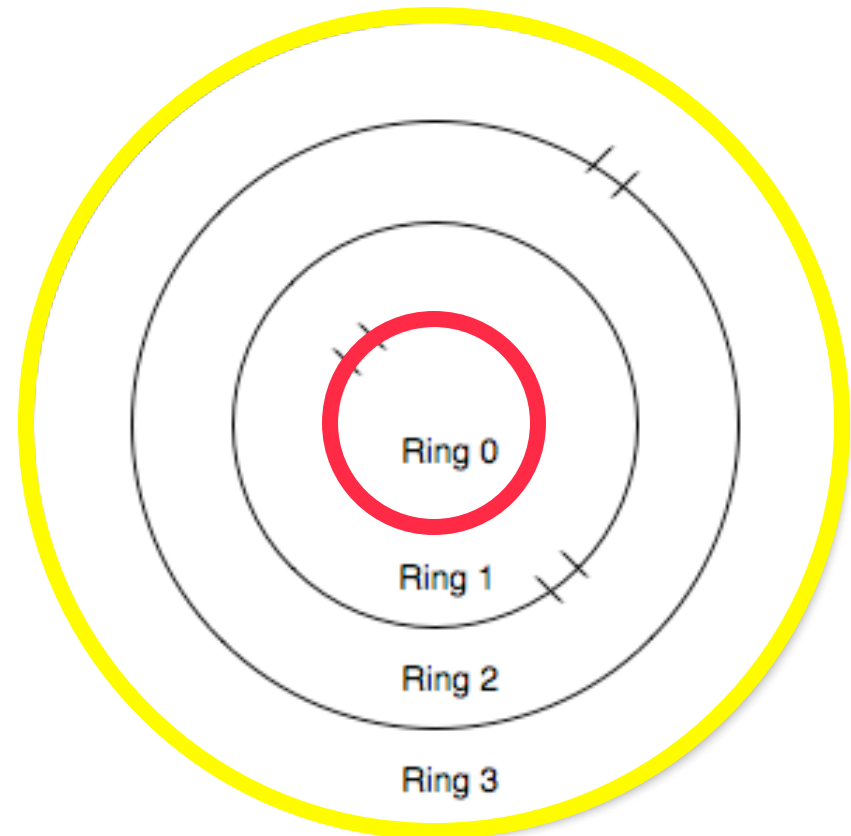
- Level 3
  - Minimal hardware access
  - User space processes run at level 3

## User Space

- Limited hardware access
- N/A in Linux
- Level 1
  - Limited hardware access
  - N/A in Linux

- Level 0
  - Unlimited hardware access
  - Kernel space threads run at level 0

## Kernel Space

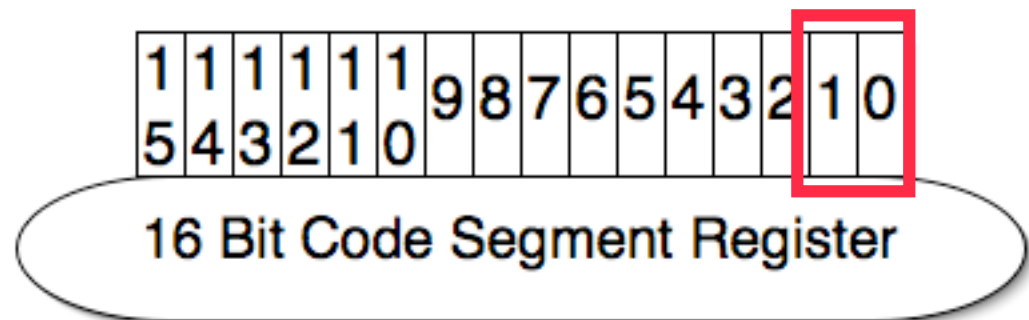


# Testing Privilege Level - User (R3)

```
#include <stdio.h>
#include <stdint.h>
```

```
int main()
{
    uint16_t cs_reg;
    asm("mov %%cs,%0" : "=m" (cs_reg));
    cs_reg = cs_reg & 0x0003;
    printf("ring: %d\n", cs_reg);

    return 0;
}
```



# Testing Privilege Level - User (R3)

```
Terminal
#include <stdio.h>
#include <stdint.h>

int main()
{
    uint16_t cs_reg;
    asm("mov %%cs,%0" : "=m" (cs_reg));
    cs_reg = cs_reg & 0x0003;
    printf("ring: %d\n", cs_reg);

    return 0;
}
$ ./get_cpl
ring: 3
$
```

# Testing CPL - Kernel (R0)

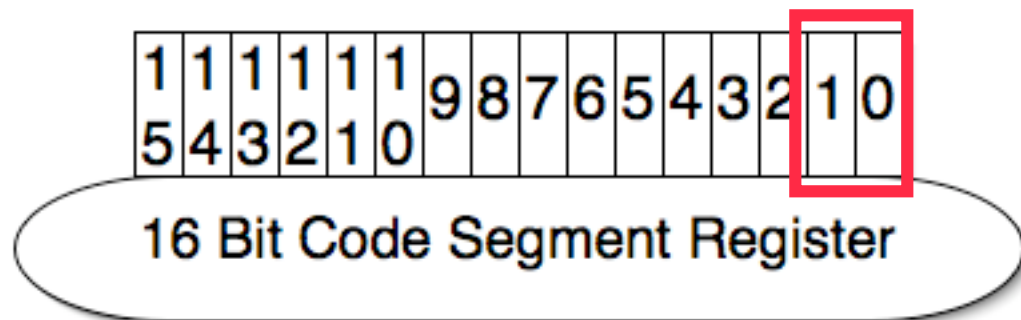
```
#include <linux/module.h>
#include <linux/kernel.h>
#include <linux/init.h>

static int __init get_cpl_init(void)
{
    uint16_t cs_reg;
    asm("mov %%cs,%0" : "=m" (cs_reg));
    cs_reg = cs_reg & 0x0003;
    printk(KERN_ALERT "ring: %d\n", cs_reg);

    return 0;
}

static void __exit get_cpl_exit(void)
{
}

module_init(get_cpl_init);
module_exit(get_cpl_exit);
```



# Testing CPL - Kernel (R0)

```
Terminal
    printk(KERN_ALERT "ring: %d\n", cs_reg);

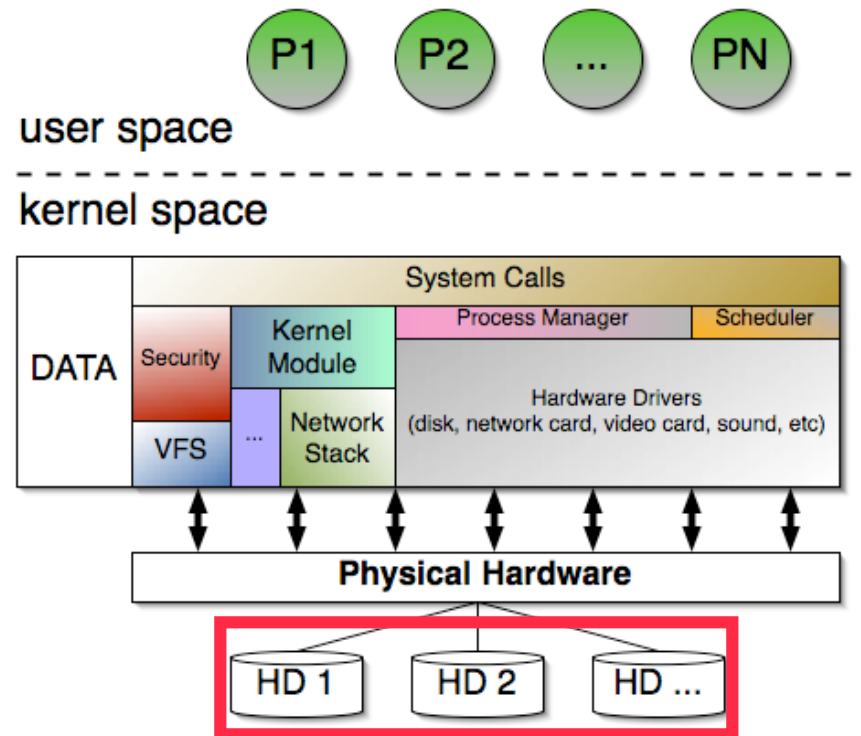
    return 0;
}

static void __exit get_cpl_exit(void)
{
}

module_init(get_cpl_init);
module_exit(get_cpl_exit);
# insmod get_cpl.ko
# tail -n 1 /var/log/syslog
Jul 12 02:54:09 localhost kernel: ring: 0
#
```

# User-Level Rootkit Attacks

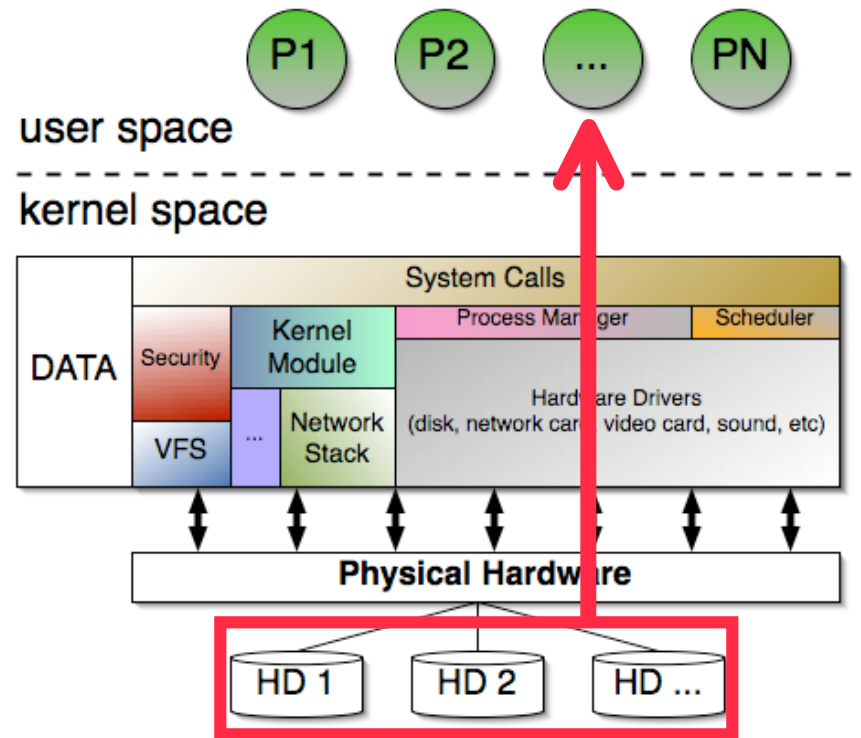
Modify/replace system binaries



e.g. *ps*, *netstat*, *ls*, *top*, *passwd*

# User-Level Rootkit Attacks

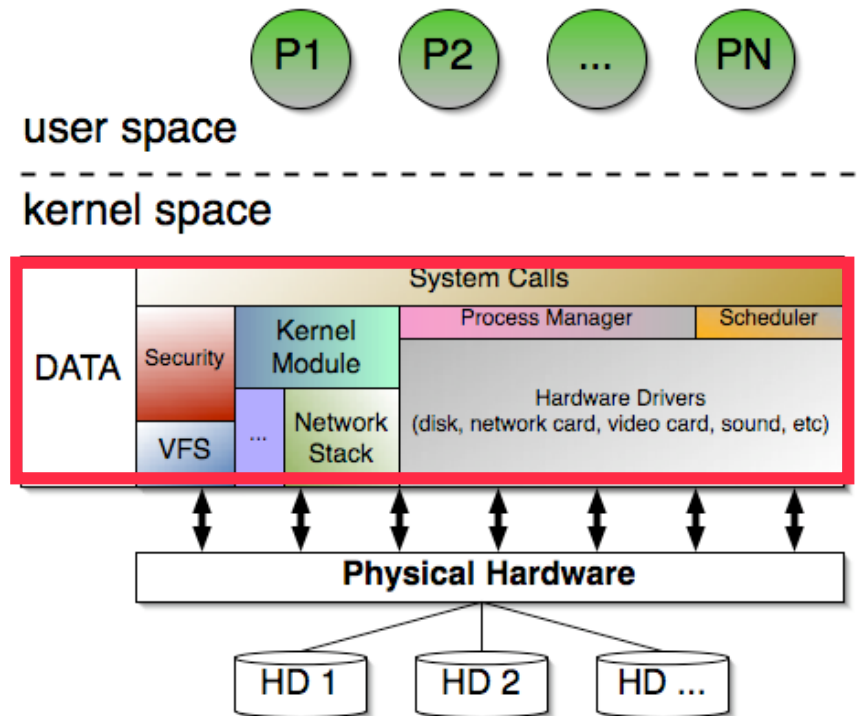
Modify/replace system binaries



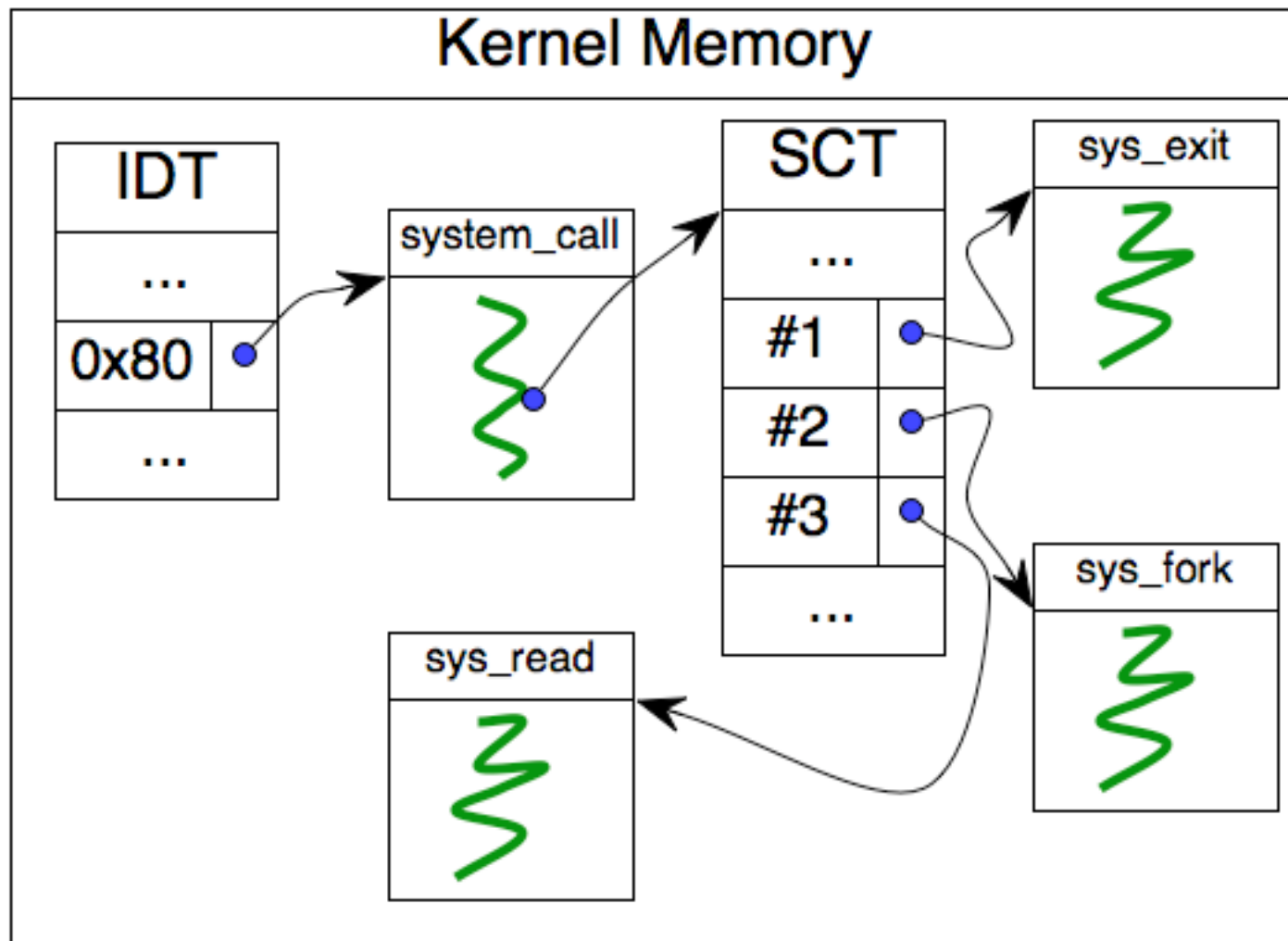
e.g. *ps*, *netstat*, *ls*, *top*, *passwd*

# Kernel-Level Rootkit Attacks

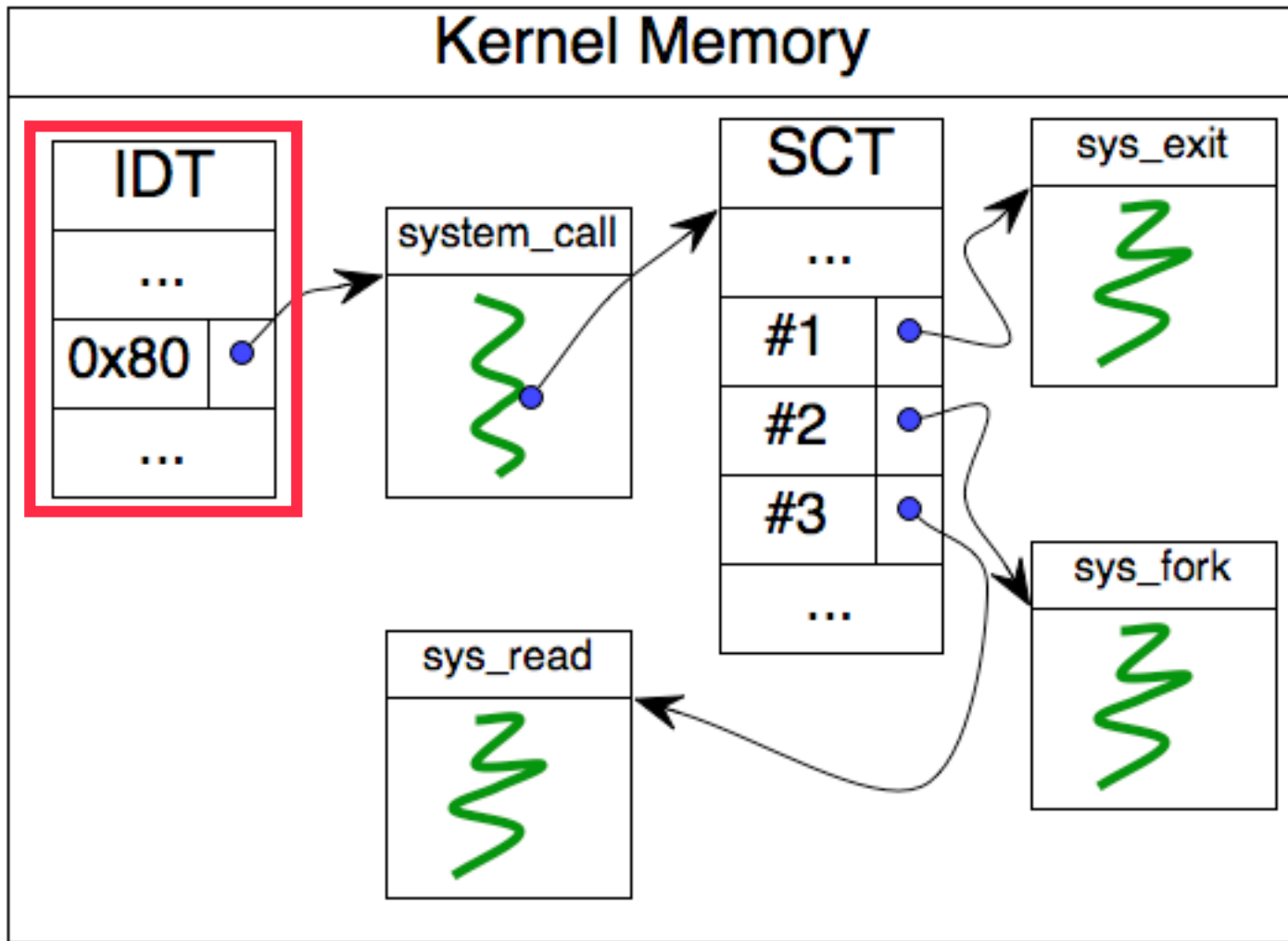
Modify running kernel code and data structures



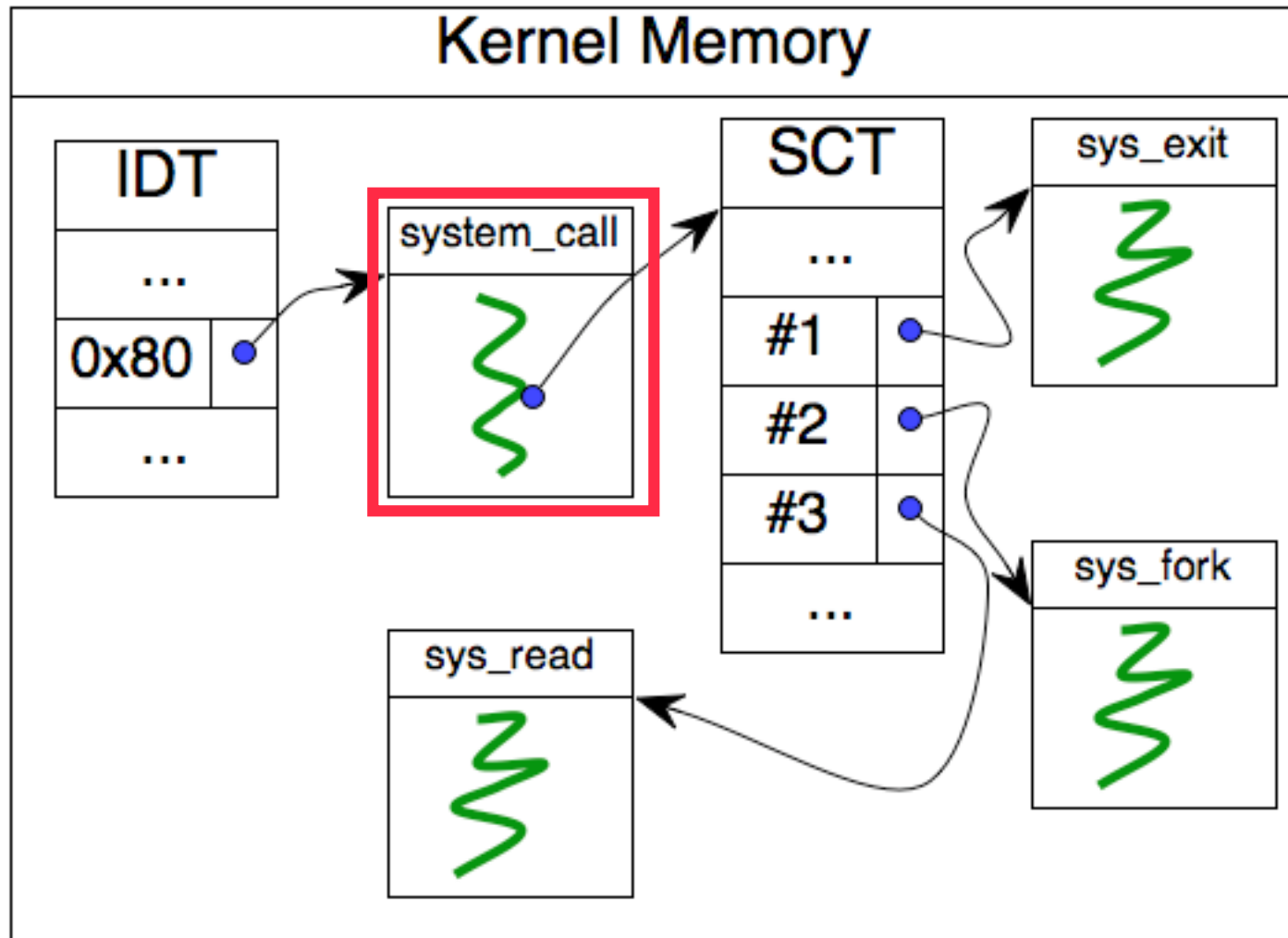
# Example 1 (System Call Table)



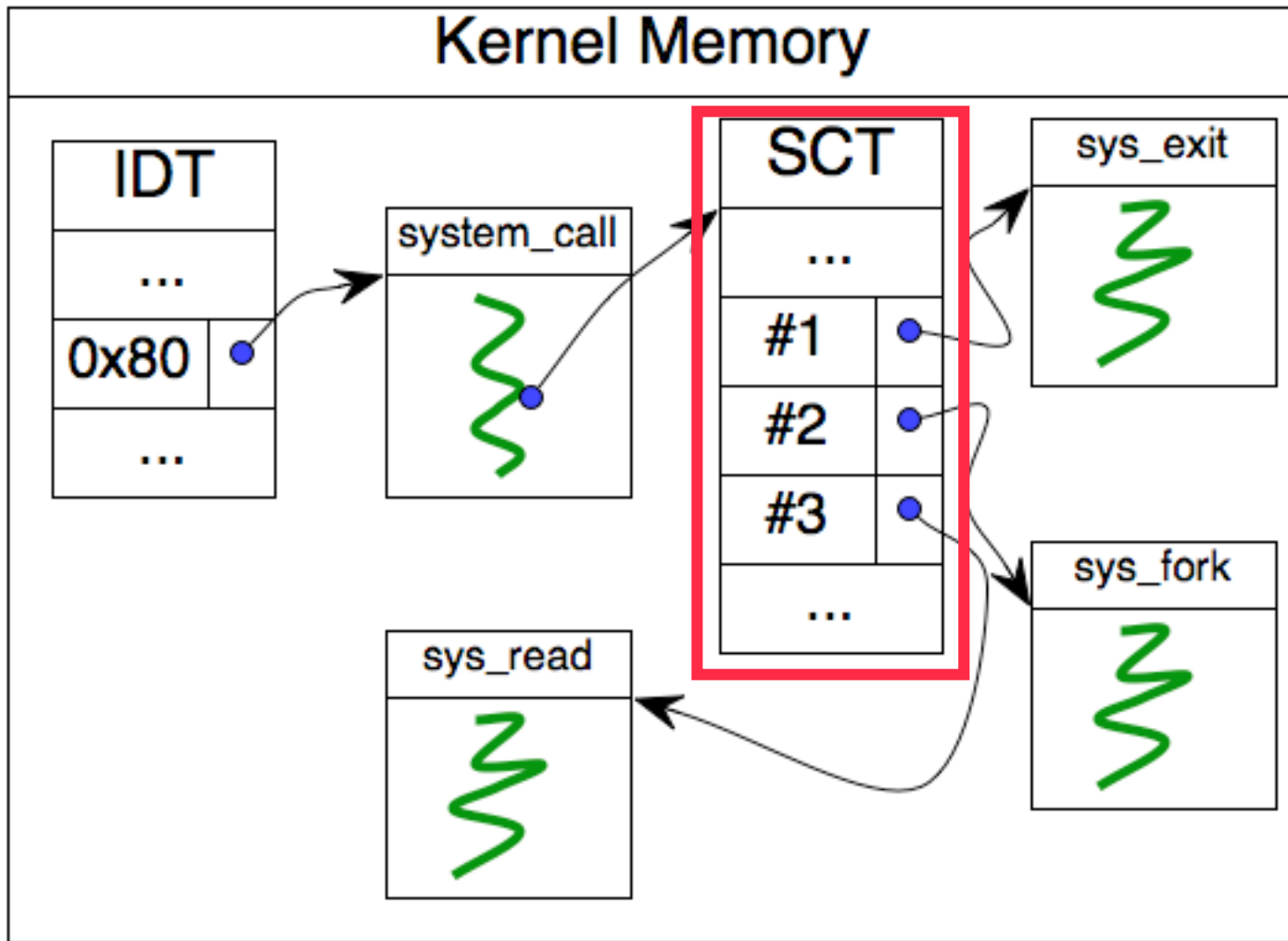
# 0x80ith IDT Entry Lookup



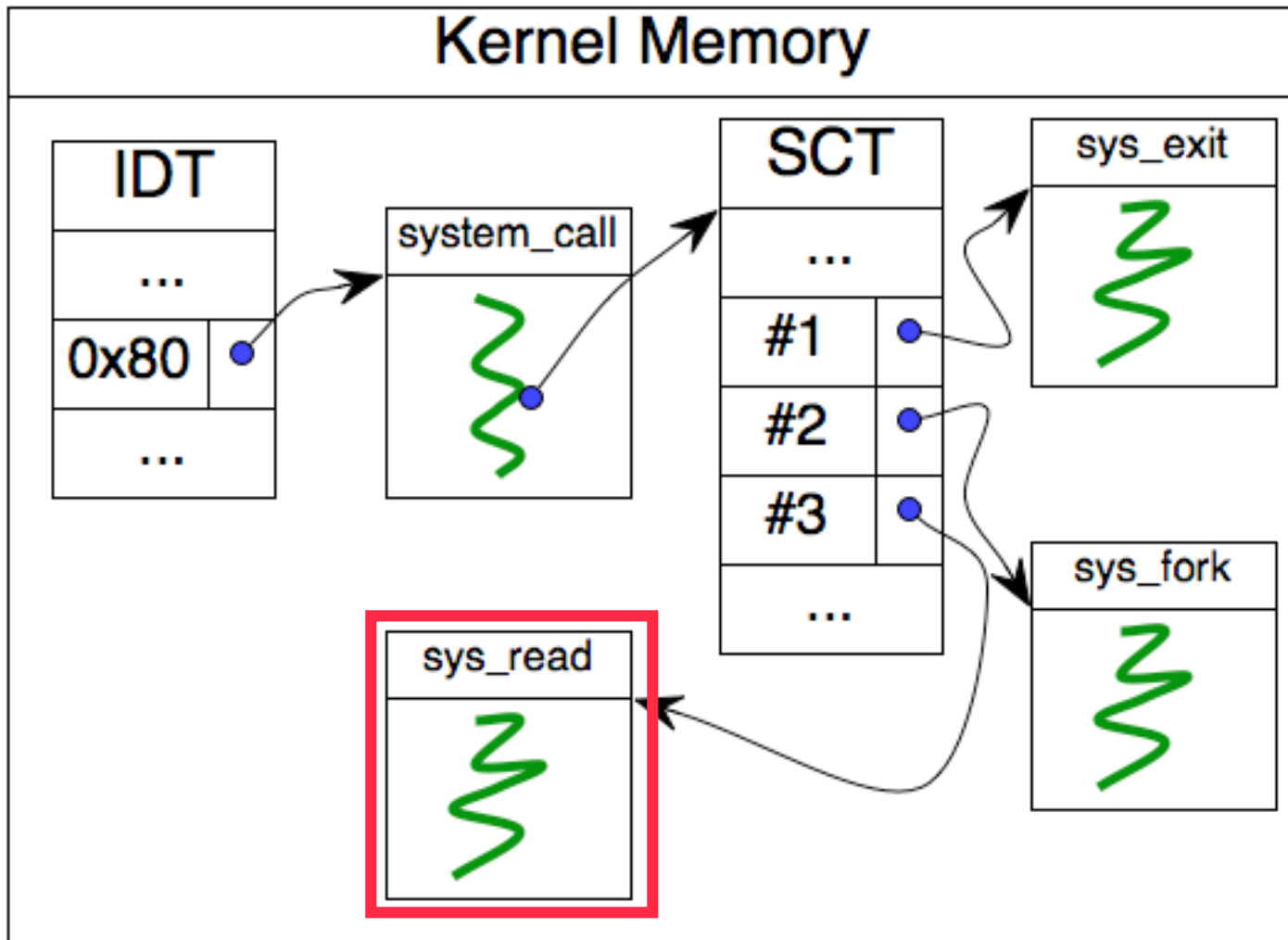
# System Call Handler



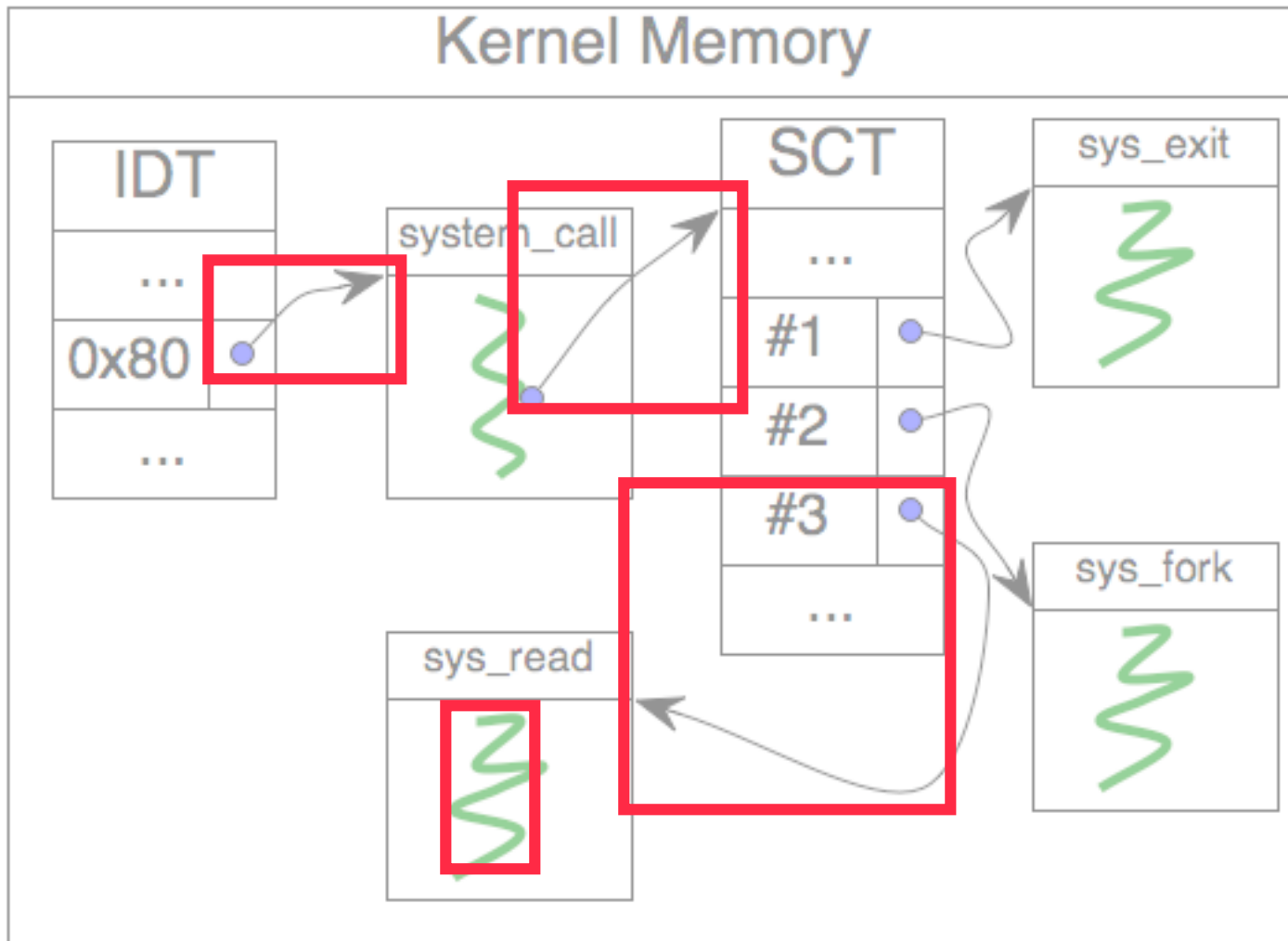
# System Call Lookup



# System Call Executes



# Attack Points



# Manual Recovery Algorithm

- 1) Copy clean system calls to kernel memory (get from kernel image with modified gdb)
- 2) Create new system call table
- 3) Copy system call handler to kmem (set new SCT)
- 4) Query the *idtr* register (interrupt table)
- 5) Set *0x80*th entry to new handler

# Copying Kernel Functions

- Some trickery involved with algorithm
- x86 code has *call* instructions with a relative offset parameter
- Could recompile the code
- Chose to recompute relative offset and modify the machine code

# /dev/kmem Details from SuckKIT

- SuckKIT accesses kernel memory from user space
- Redirects entire system call table
- How does suckKIT find the system call table?
- How does suckKIT allocate kernel memory?

# Find System Call Handler

```
struct idtr idtr;
struct idt idt80;
ulong old80;

/* Pop IDTR register from CPU */
asm("sidt %0" : "=m" (idtr));

/* Read kernel memory through /dev/kmem */
rkm(fd, &idt80, sizeof(idt80), idtr.base +
0x80 * sizeof(idt80));

/* Compute absolute offset of
 * system call handler for kmem
 */
old80 = idt80.off1 | (idt80.off2 << 16);
```

# Find System Call Handler

```
struct idtr idtr;
struct idt idt80;
ulong old80;

/* Pop IDTR register from CPU */
asm("sidt %0" : "=m" (idtr));

/* Read kernel memory through /dev/kmem */
rkm(fd, &idt80, sizeof(idt80), idtr.base +
0x80 * sizeof(idt80));

/* Compute absolute offset of
 * system call handler for kmem
 */
old80 = idt80.off1 | (idt80.off2 << 16);
```

# Find System Call Handler

```
struct idtr idtr;
struct idt idt80;
ulong old80;

/* Pop IDTR register from CPU */
asm("sidt %0" : "=m" (idtr));

/* Read kernel memory through /dev/kmem */
rkm(fd, &idt80, sizeof(idt80), idtr.base +
0x80 * sizeof(idt80));

/* Compute absolute offset of
 * system call handler for kmem
 */
old80 = idt80.off1 | (idt80.off2 << 16);
```

# Find System Call Handler

```
struct idtr idtr;
struct idt idt80;
ulong old80;

/* Pop IDTR register from CPU */
asm("sidt %0" : "=m" (idtr));

/* Read kernel memory through /dev/kmem */
rkm(fd, &idt80, sizeof(idt80), idtr.base +
0x80 * sizeof(idt80));

/* Compute absolute offset of
 * system call handler for kmem
 */
old80 = idt80.off1 | (idt80.off2 << 16);
```

# Kmalloc as a System Call (sucKIT)

```
#define rr(n, x) ,n ((ulong) x)
#define __NR_oldolduname 59
#define OURSYS __NR_oldolduname
#define syscall2(__type, __name, __t1, __t2) \
    __type __name(__t1 __a1, __t2 __a2) \
{ \
    ulong __res; \
    __asm__ volatile \
    ("int $0x80" \
     : "=a" (__res) \
     : "0" (__NR_##__name) \
     rr("b", __a1) \
     rr("c", __a2)); \
    return (__type) __res; \
}
#define __NR_KMALLOC OURSYS
static inline syscall2(ulong, KMALLOC, ulong, ulong);
```

# System Call Table Tools Demonstration

# System Calls Used for Recovery

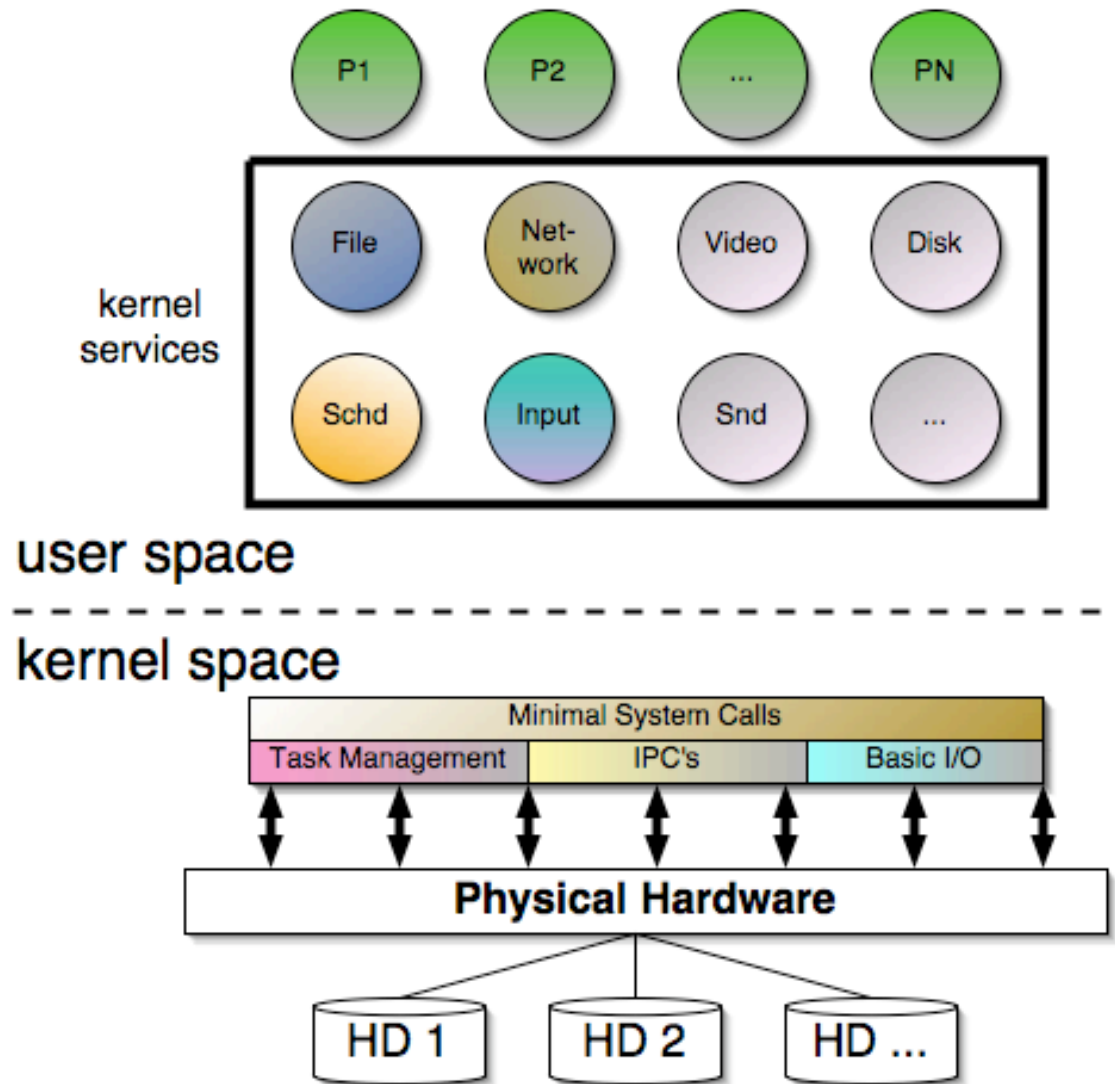
- Using /dev/kmem
  - `sys_open`, `sys_read`, `sys_write`
- Using kernel module
  - `sys_create_module`, `sys_init_module`

Problem: system call table has been redirected by a rootkit

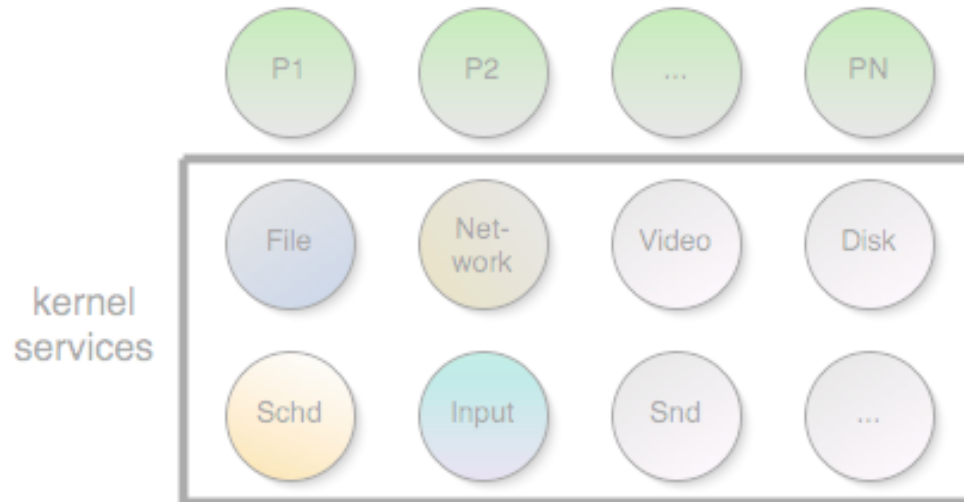
# Solution

- **Intrusion Recovery System (IRS)**
  - Use **spine** architecture to minimize chance of rootkit attack
  - IRS capable of verifying **integrity** of system
  - Contains copy of known good state for entire system (including kernel) in isolated area called **statehold**

# Spine - Based on Microkernel

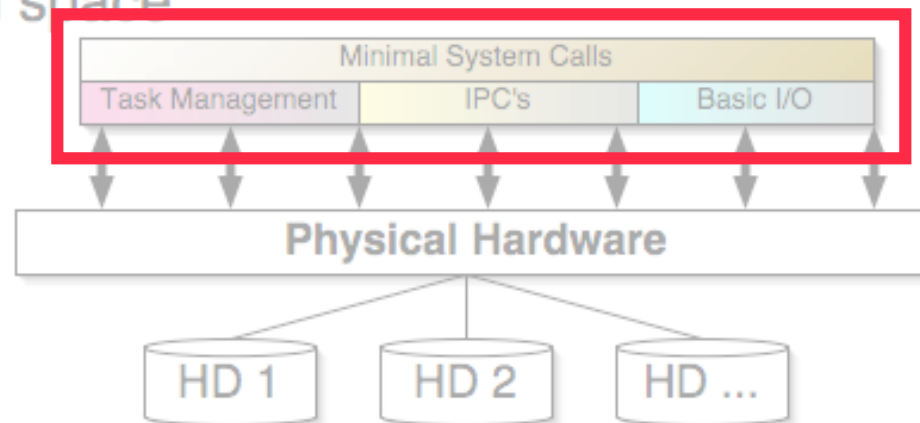


# Spine - Based on Microkernel



20,000 lines of code - small

user space  
kernel space



# L4 System Calls (Fiasco)

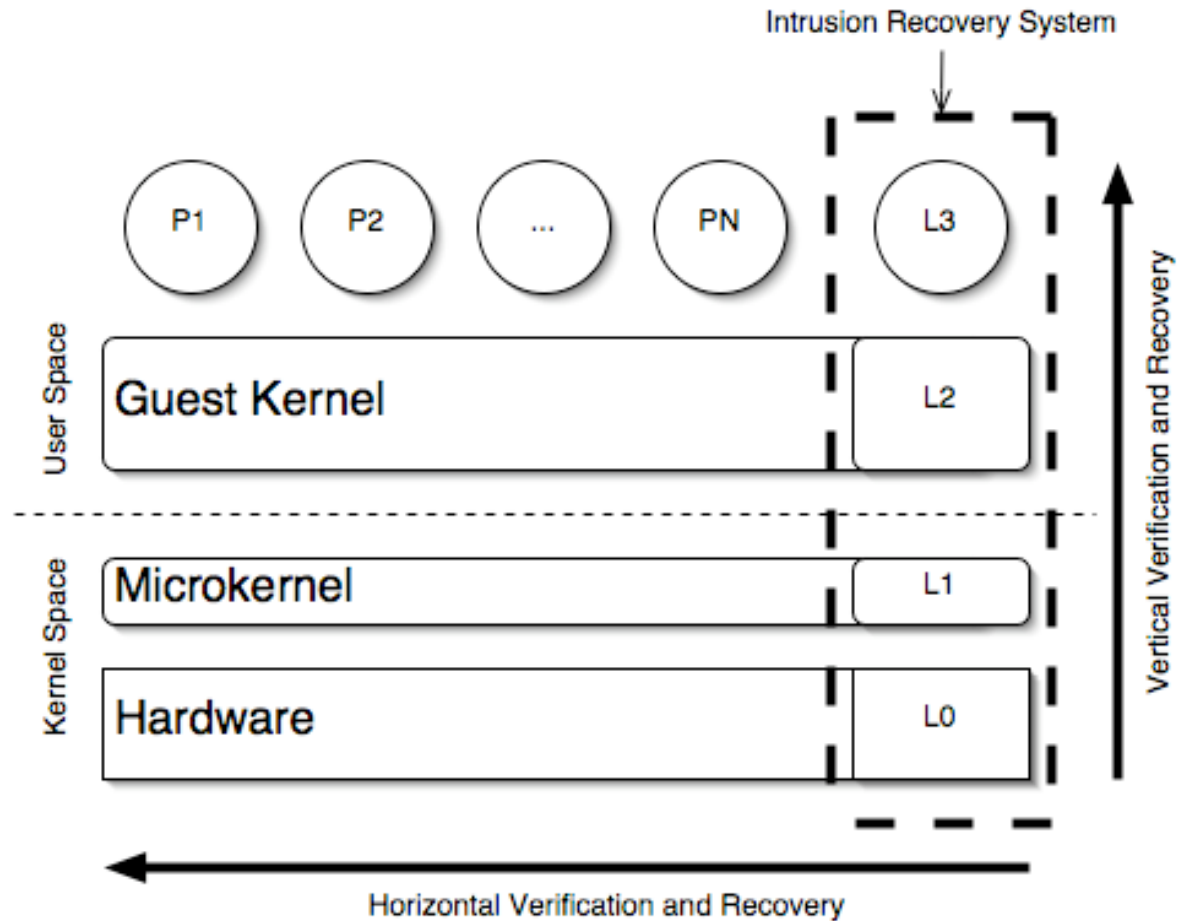
- **9 IPC Calls**

- l4\_ipc\_call, l4\_ipc\_receive
- l4\_ipc\_reply\_and\_wait
- l4\_ipc\_send\_deceit, l4\_ipc\_reply\_deceit\_and\_wait
- l4\_ipc\_send, l4\_ipc\_wait
- l4\_nchief
- l4\_fpage\_unmap

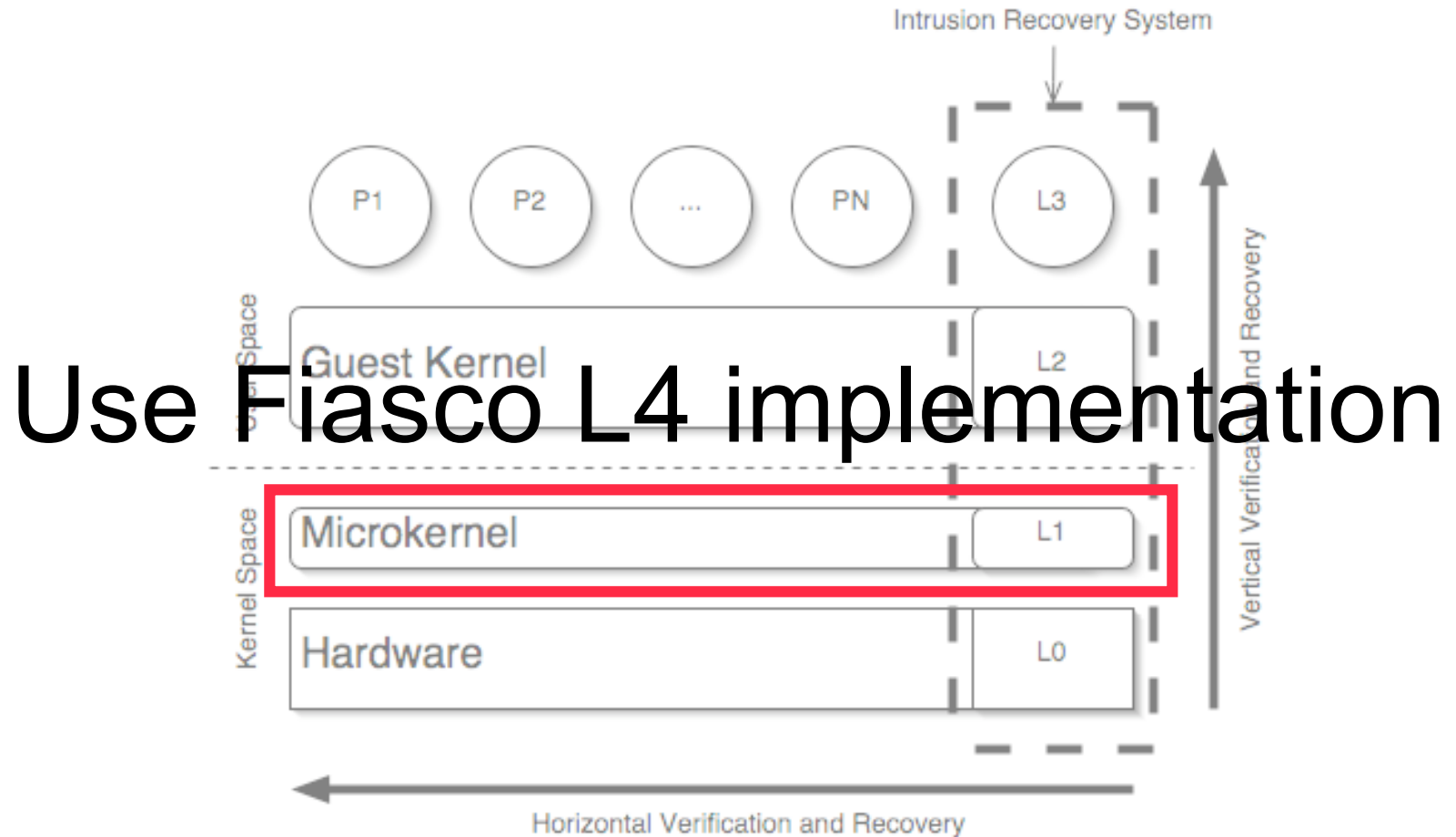
- **5 Thread calls**

- l4\_myself
- l4\_task\_new
- l4\_thread\_ex\_regs
- l4\_thread\_schedule
- l4\_thread\_switch

# Spine Architecture

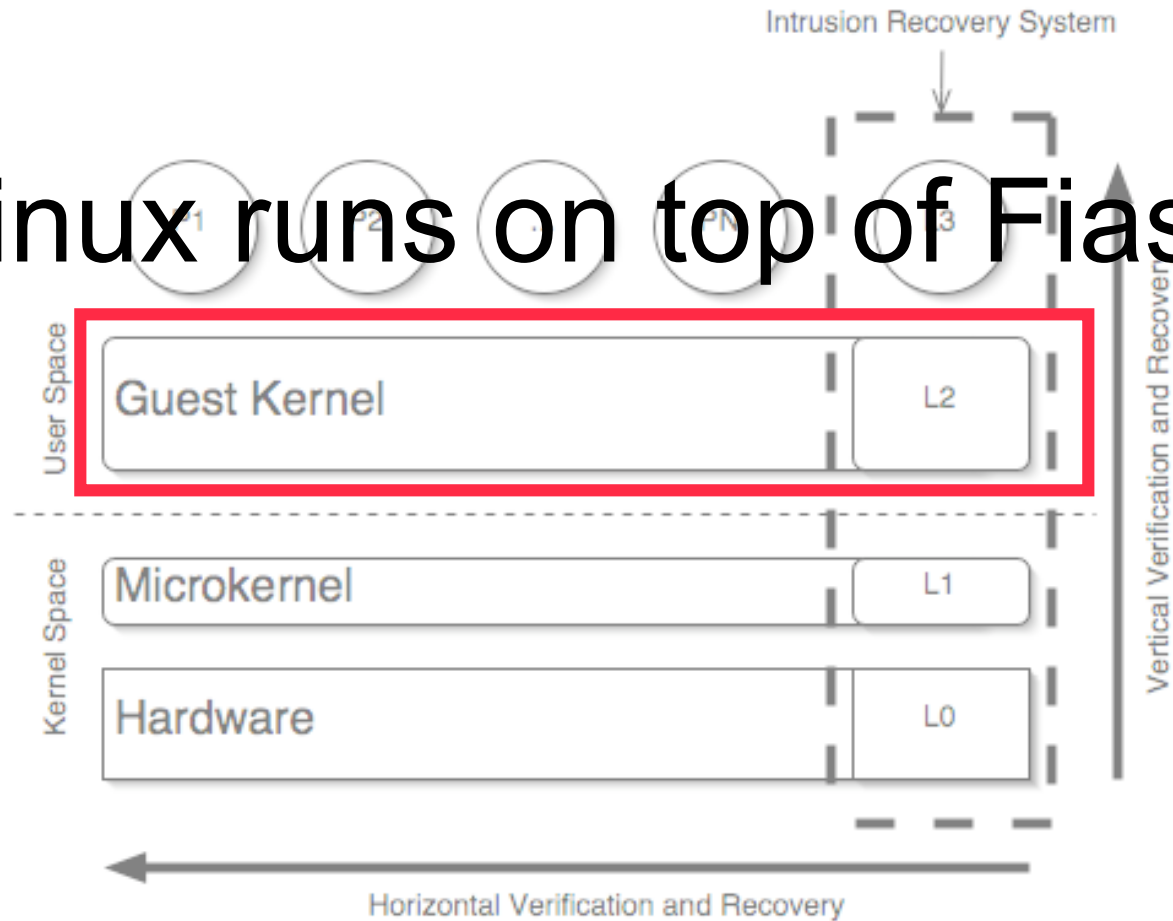


# Spine Architecture - Microkernel



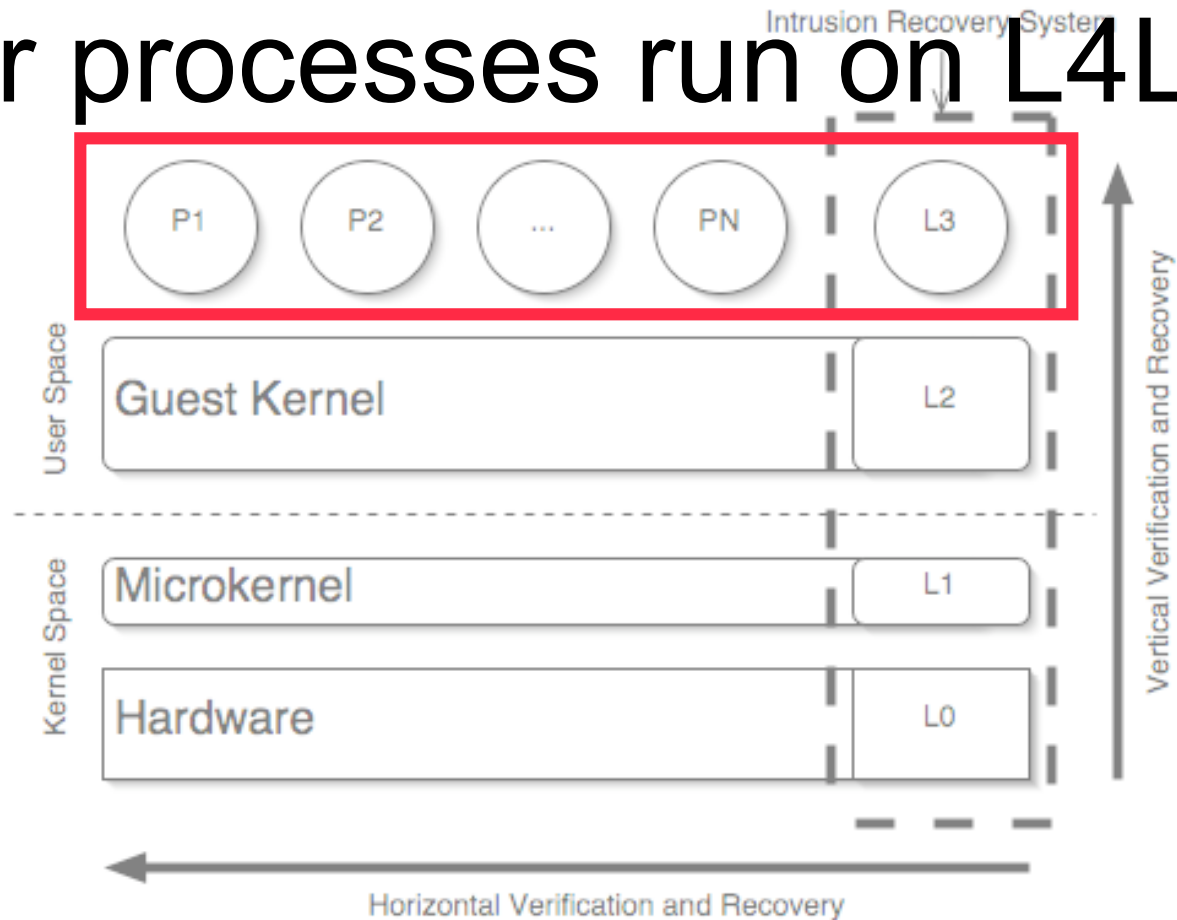
# Spine Architecture - Guest

L4Linux runs on top of Fiasco

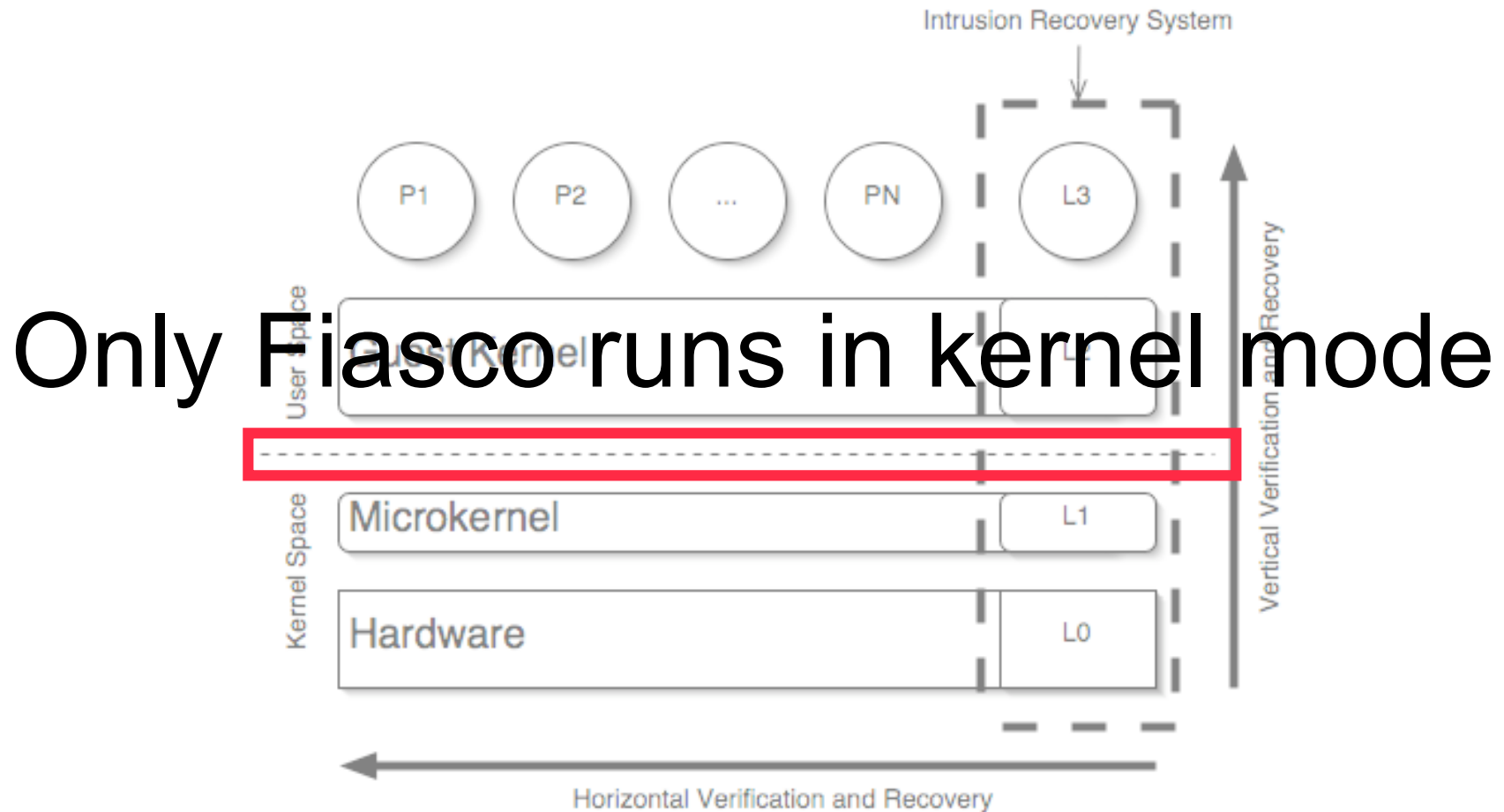


# Spine Architecture - Processes

User processes run on L4Linux

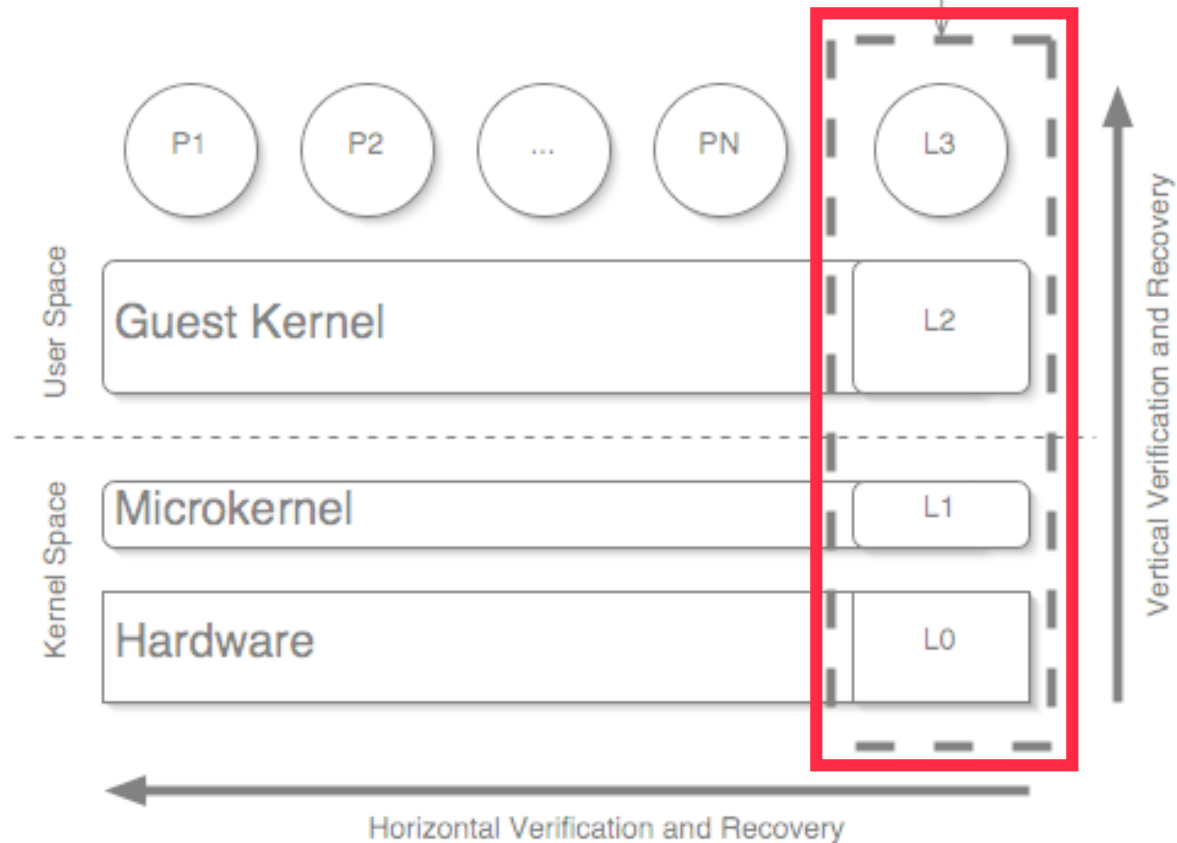


# Spine Architecture - Separation

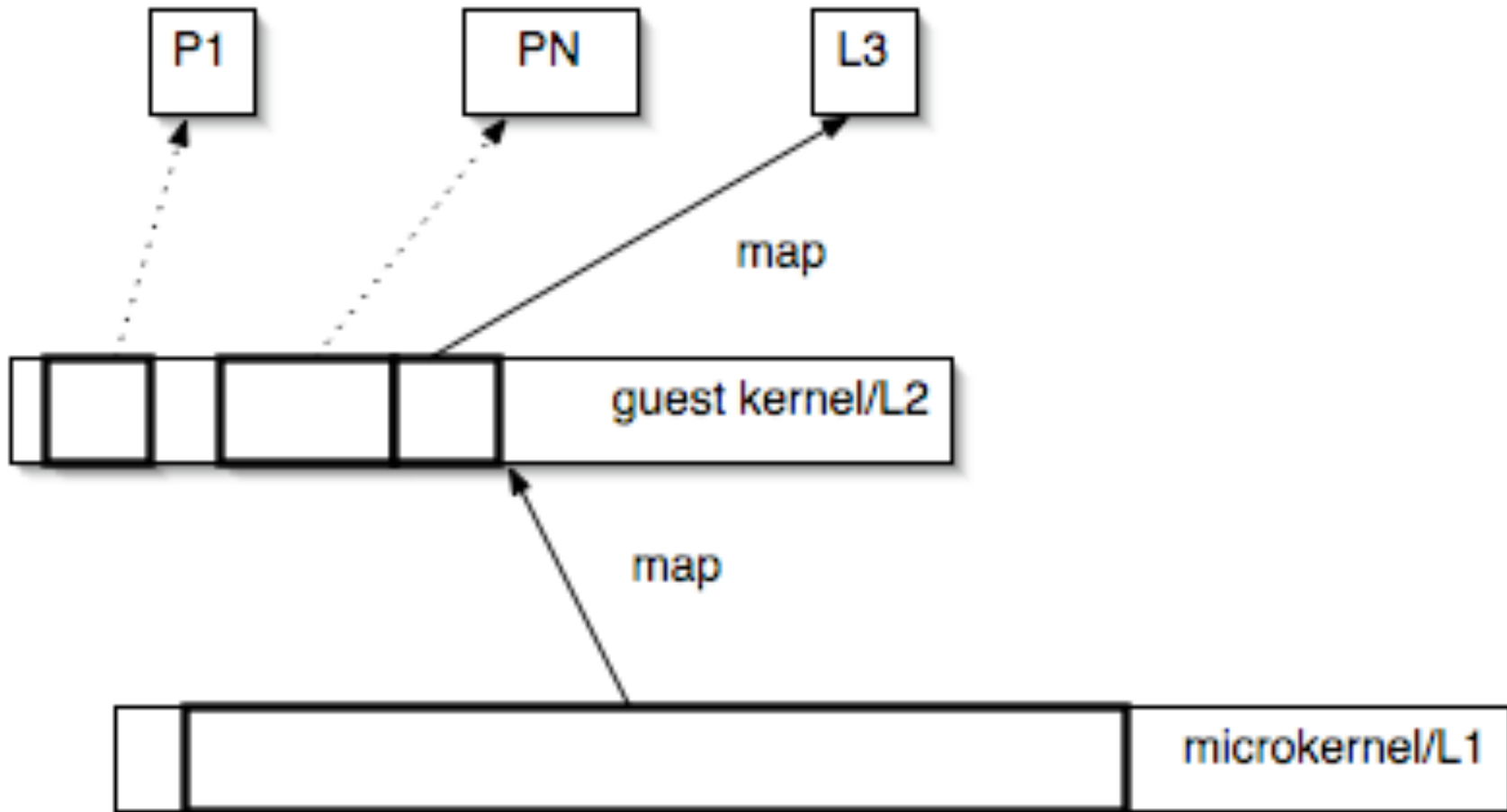


# Spine Architecture - IRS

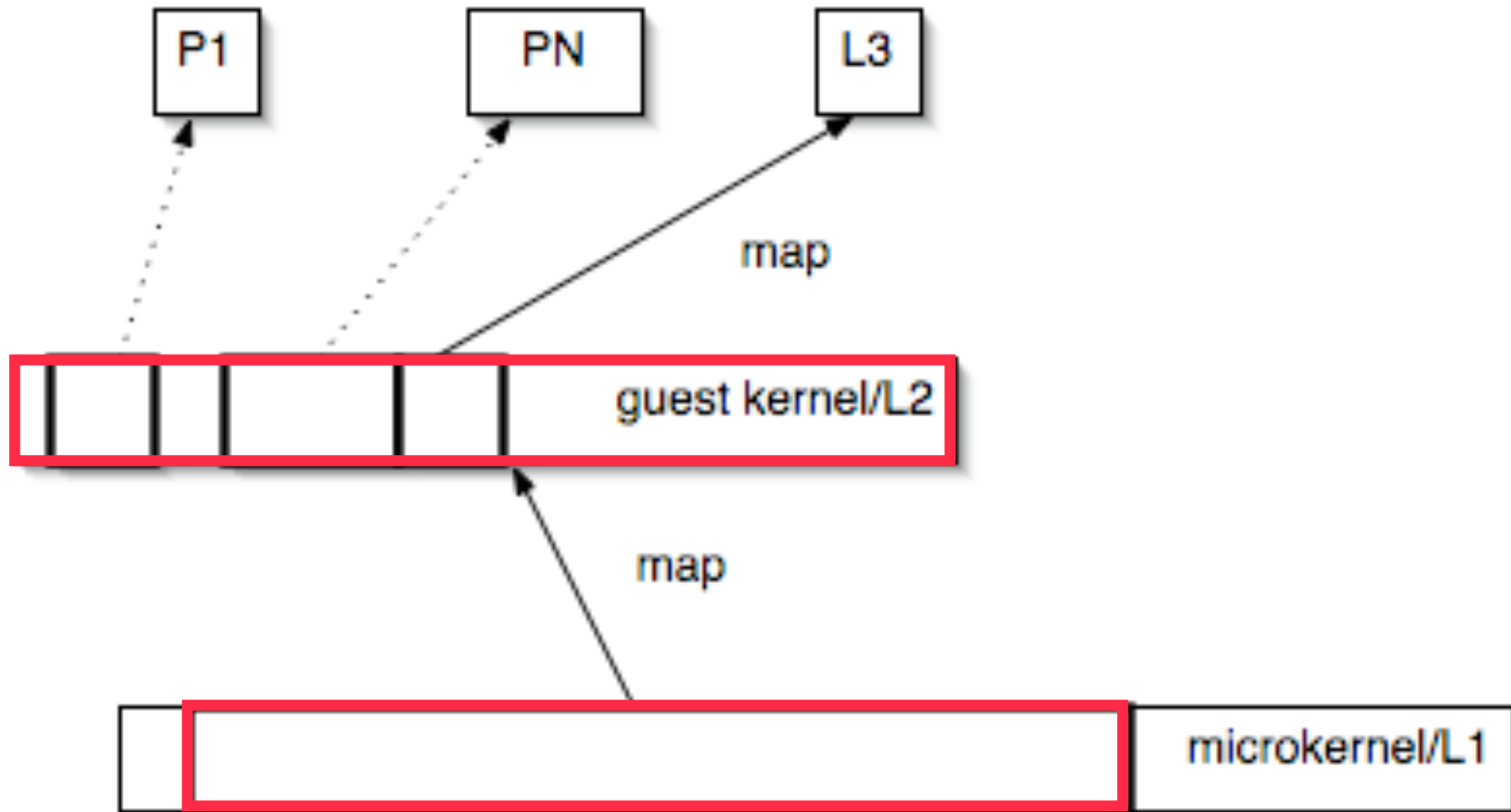
## Component of IRS at each level



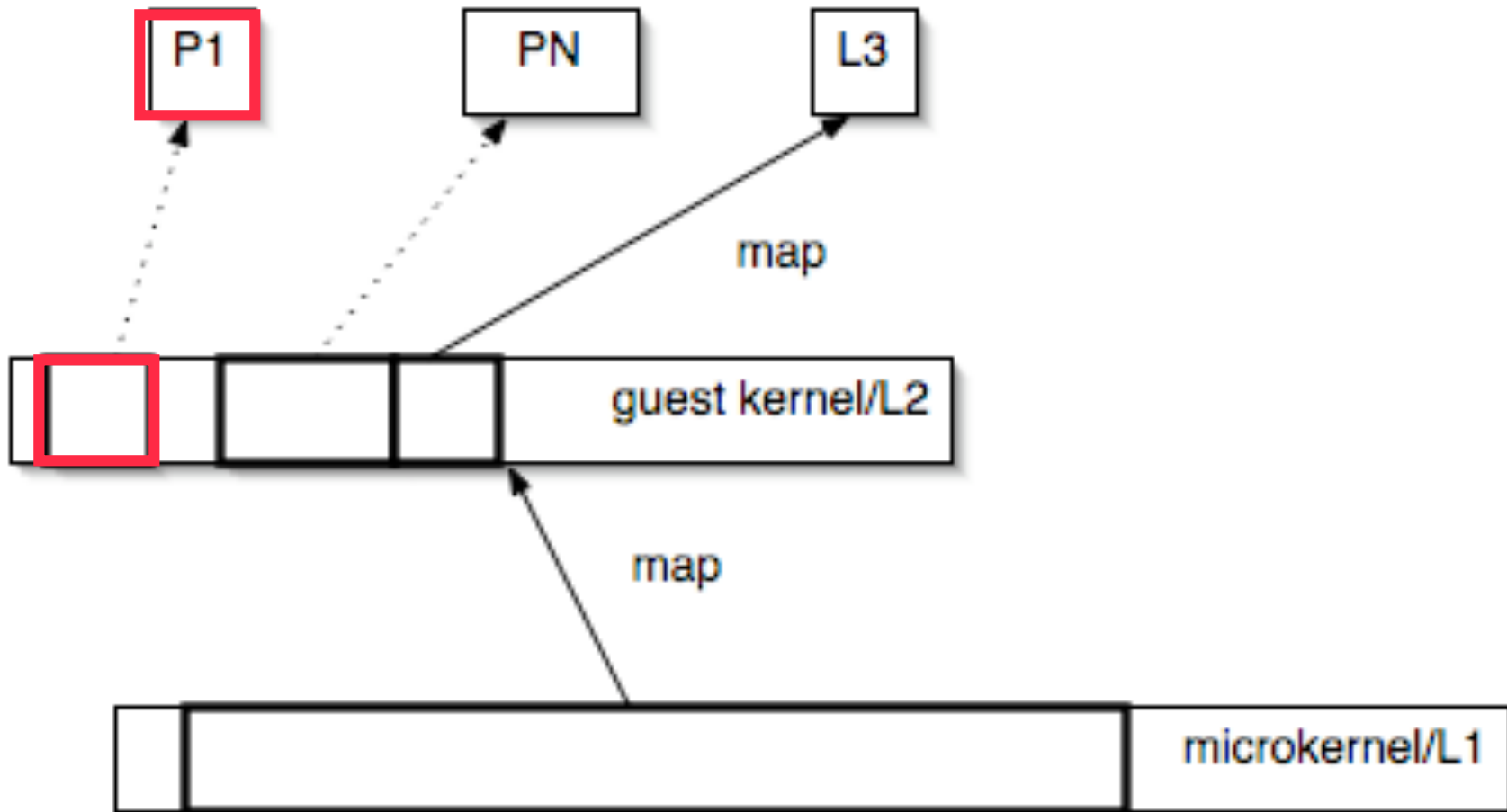
# Memory Hierarchy Detail



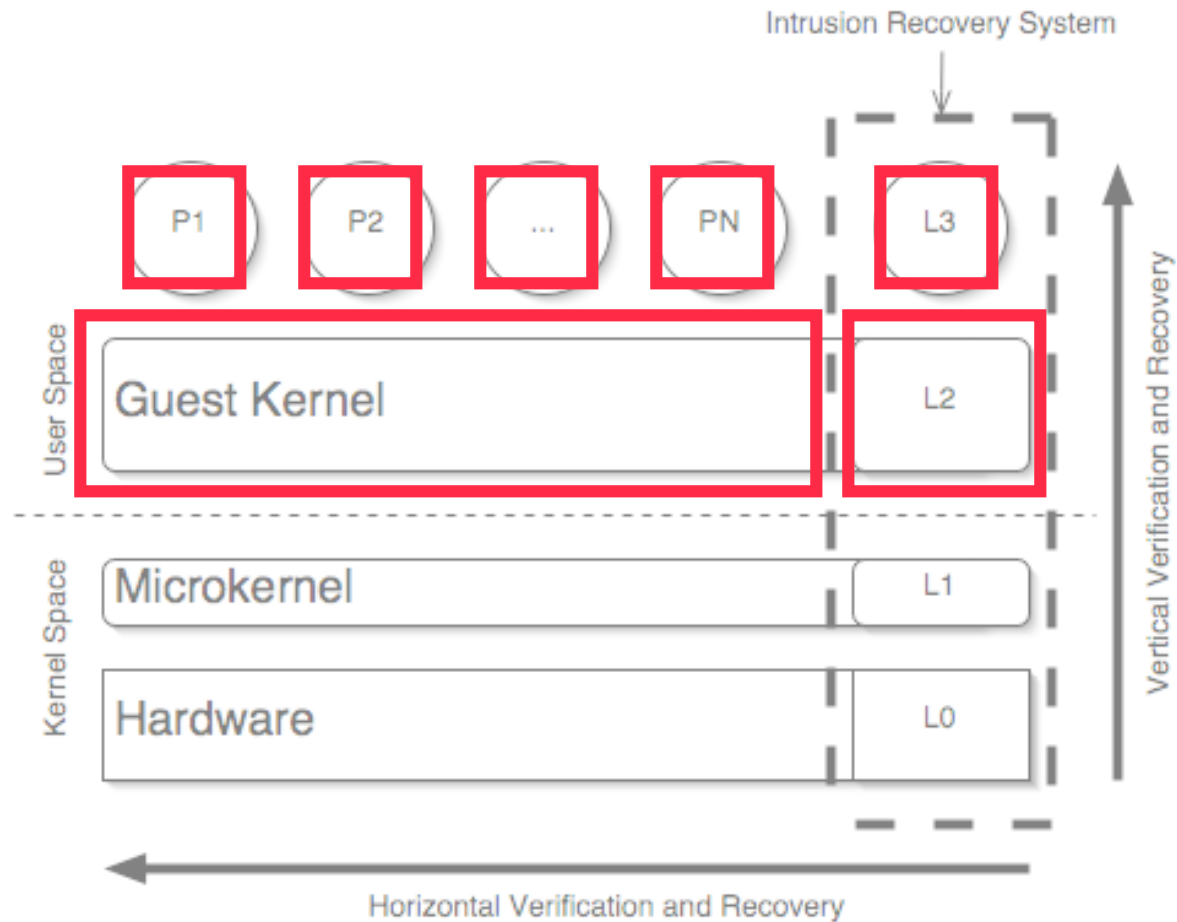
# Memory Hierarchy Detail



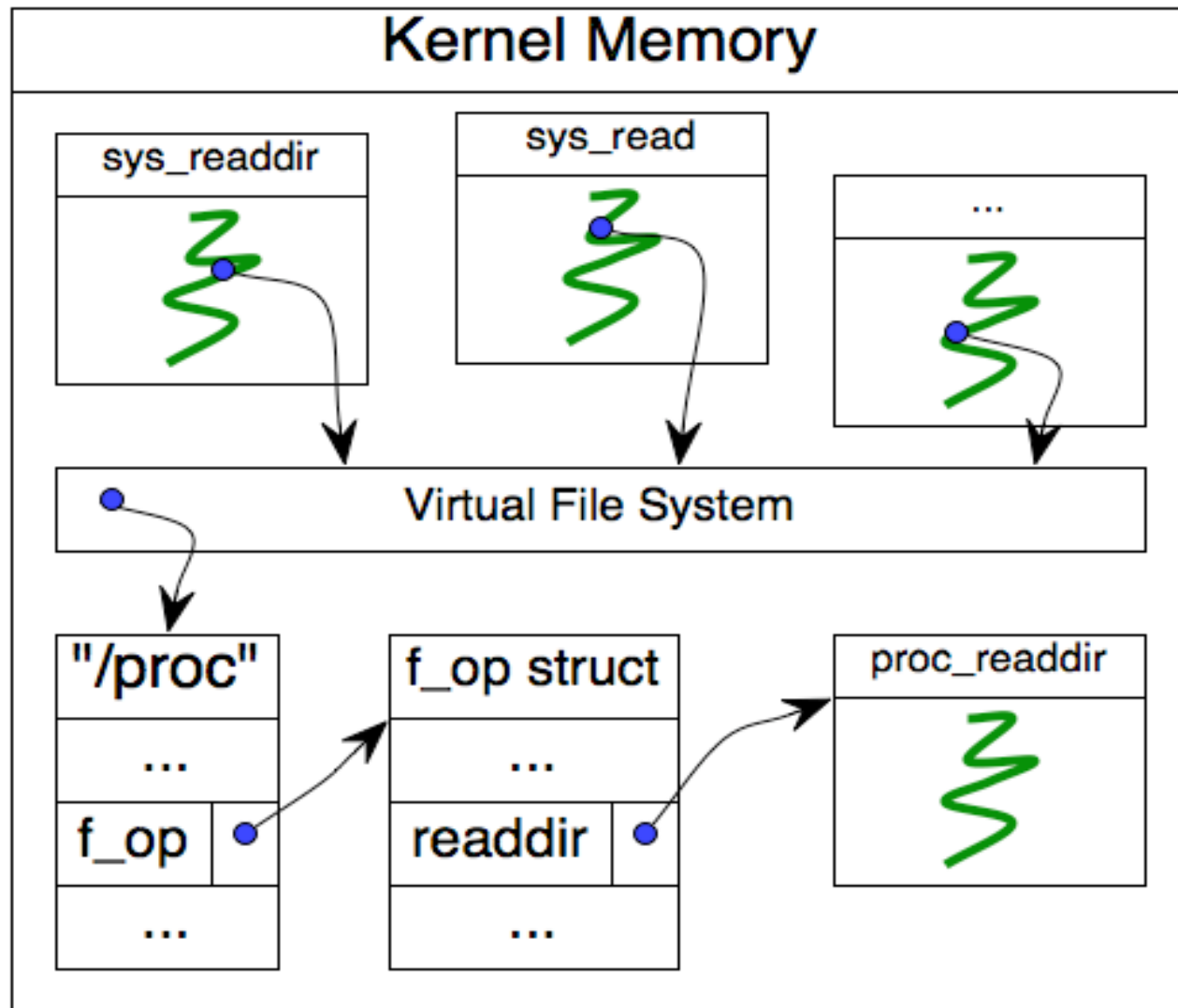
# Memory Hierarchy Detail



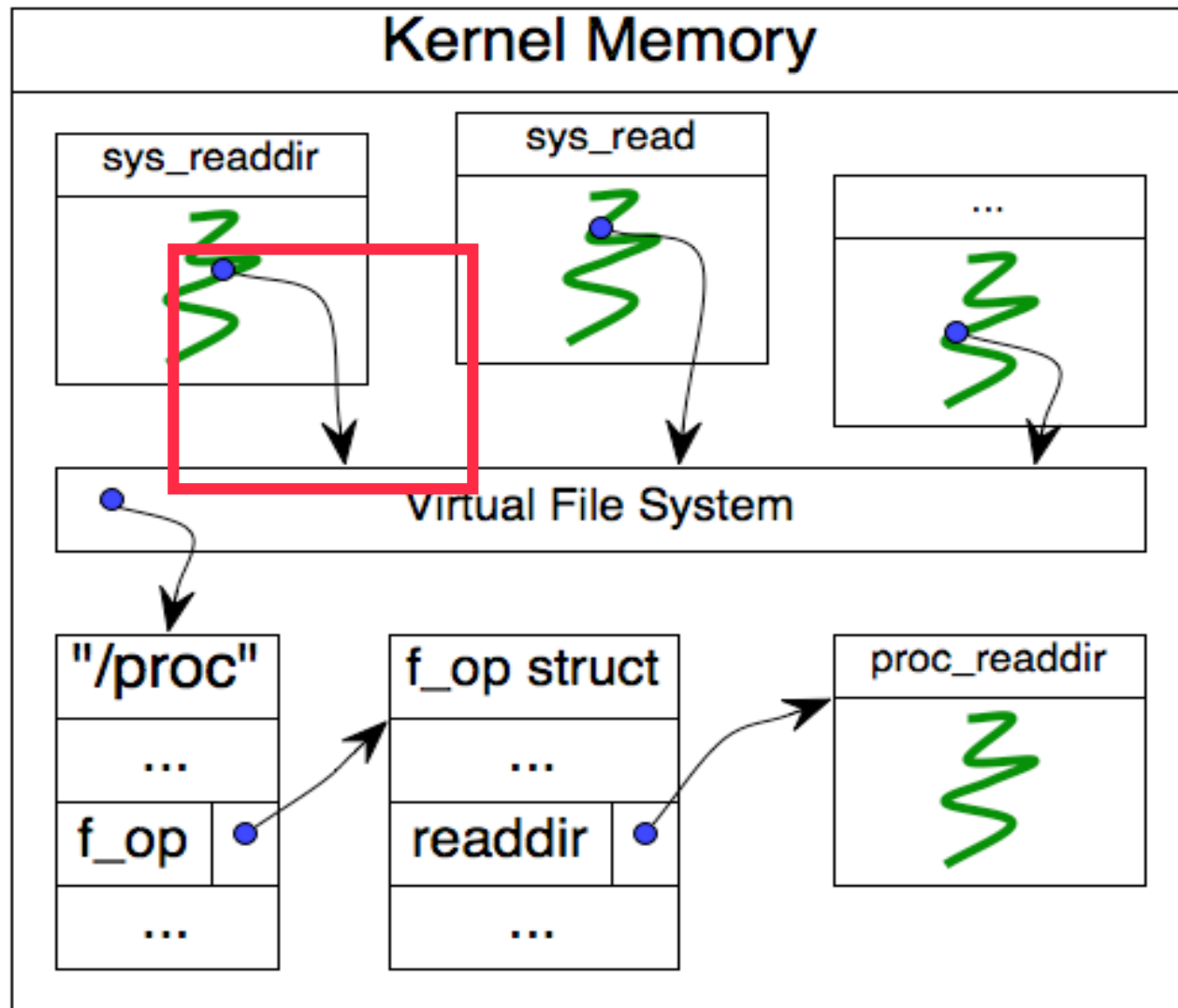
# Spine Architecture - Attacking



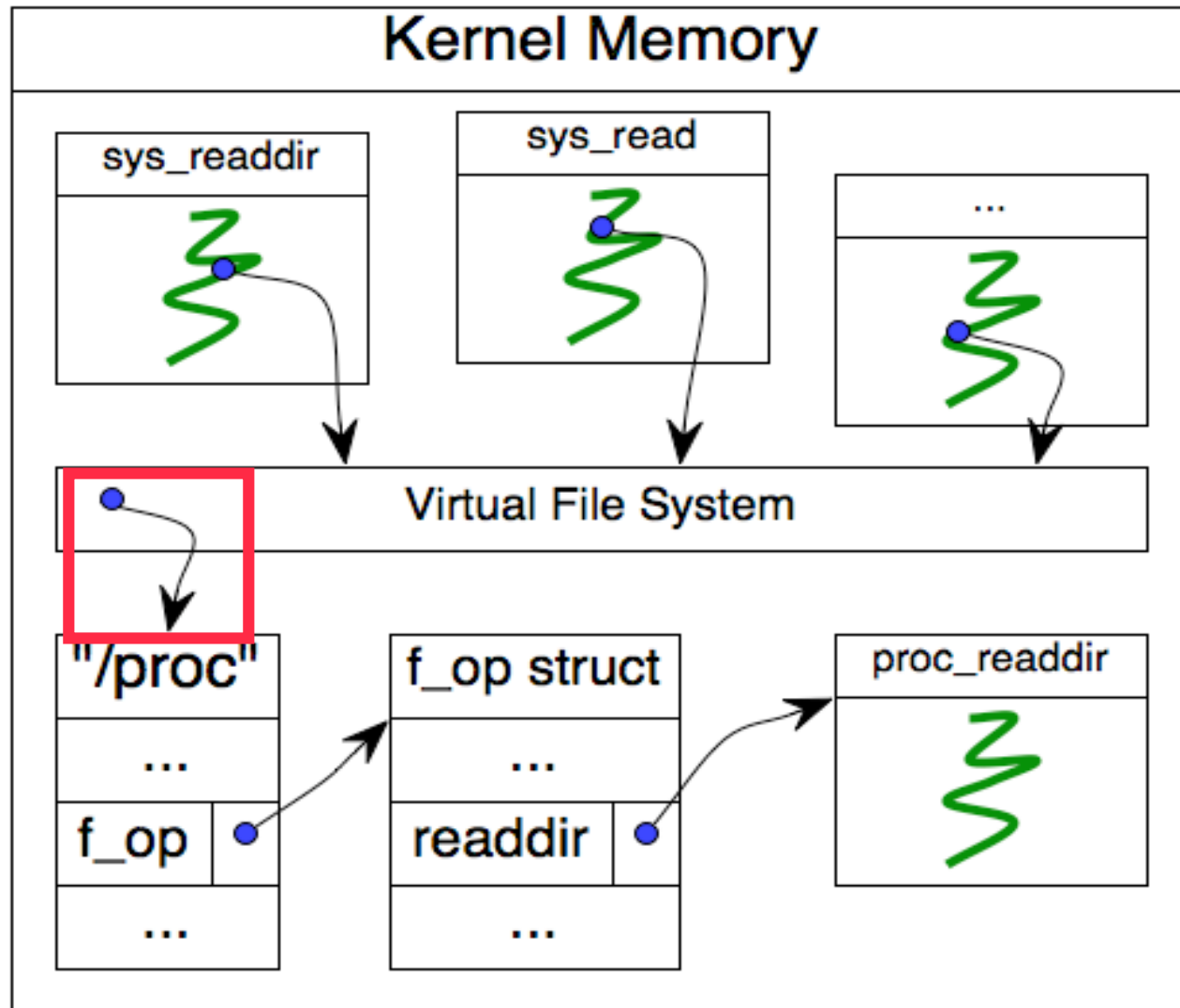
# Example 2 (VFS - /proc)



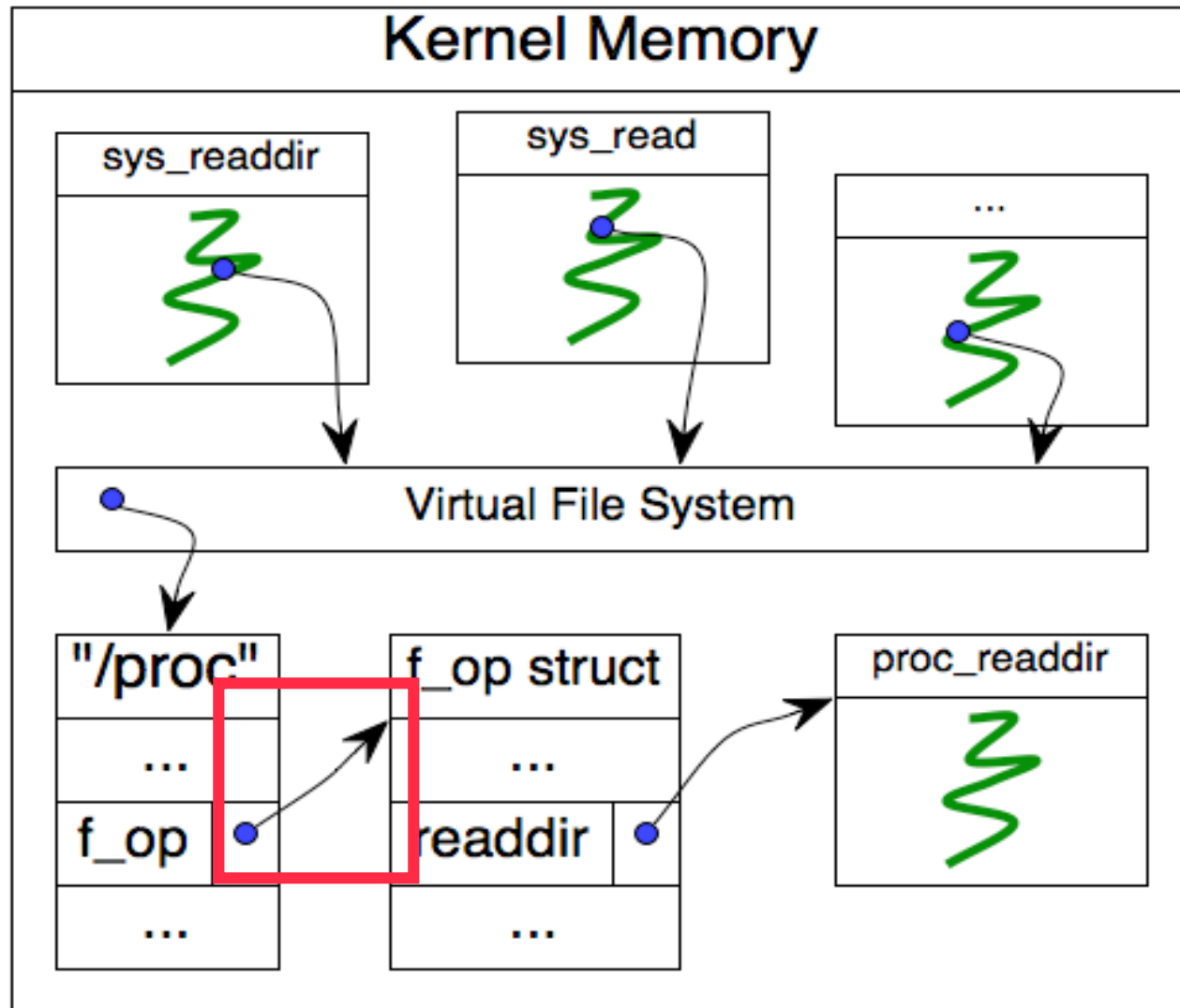
# Example 2 - Sys Call Uses VFS



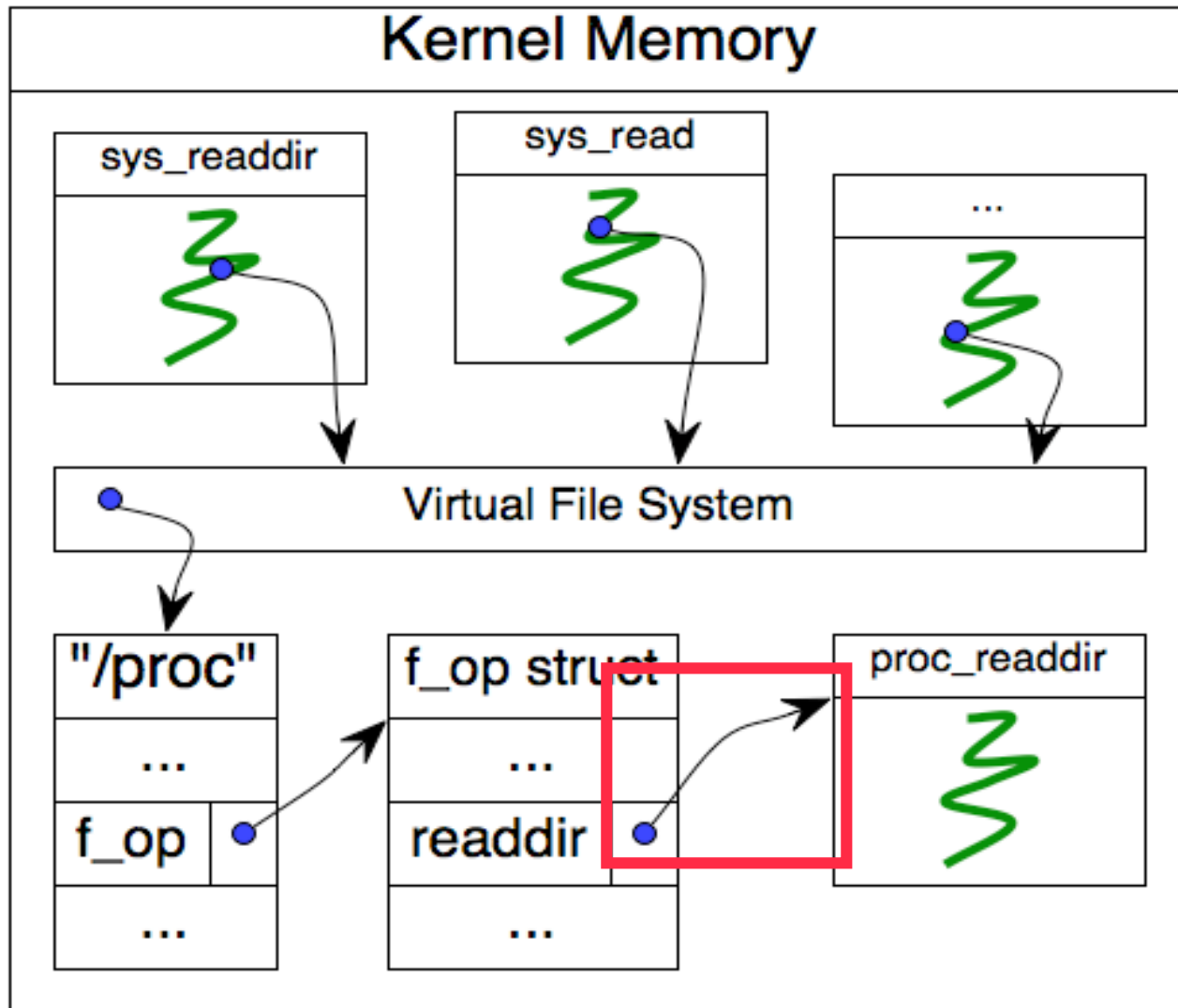
# Example 2 - /proc Filesystem



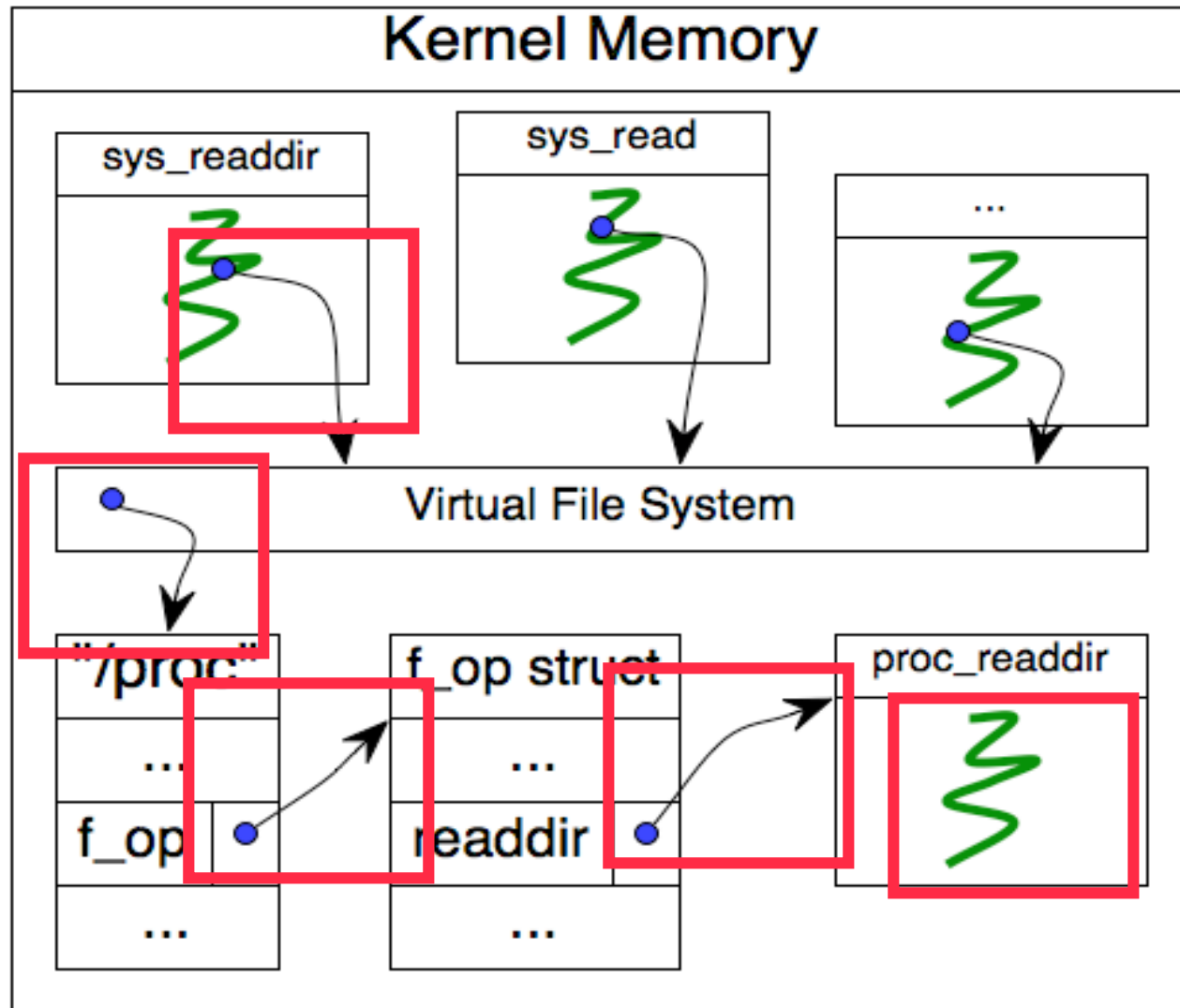
# Example 2 - File Operations



# Example 2 - Read Directory



# Example 2 - Attacking



# Recovery Methods

- Manual methods similar to SCT work
- Generally, consistency checking on function pointers and hashing of execution code works
- Must maintain a good copy of the known good state in order to repair
- IRS can do it automatically

# Intrusion Recovery System Demonstration

# Limitations and Conclusions

- Can an attacker install a microkernel-level rootkit?
- What if attacker has physical access?
- There is no be all end all solution!  
However, an IRS can make systems more reliable.

# Thanks!

- Henry Owen
- John Levine
- Sven Krasser
- Greg Conti
- Jonathan Torian
- Lawrence Phillips
- Jessica Frame
- Andrew Davenport
- Many more...

# Links

**[ Network and Security Architecture website ]**

<http://www.ece.gatech.edu/research/labs/nsa/index.shtml>

**[ Georgia Tech Information Security Center ]**

<http://www.gtisc.gatech.edu/>

**[ Fiasco project ]**

<http://os.inf.tu-dresden.de/fiasco/>

**[ Xen ]**

<http://www.cl.cam.ac.uk/Research/SRG/netos/xen/>

**[ Samhain Labs ]**

<http://la-samhna.de>

**[ Chkrootkit ]**

<http://www.chkrootkit.org>

**[ DaWheel, “So you don’t have to reinvent it!” ]**

<http://www.dawheel.org>

# Questions?

## Starter Questions:

1. How many have personally dealt with recovery from a rootkit?
2. Has anyone seen any rootkits that use direct memory access?
3. Has anyone ever cleaned a system infected with a rootkit without reinstalling?

Julian Grizzard  
grizzard AT ece.gatech.edu

# Additional Slides

Additional Slides Provided  
Beyond this Point

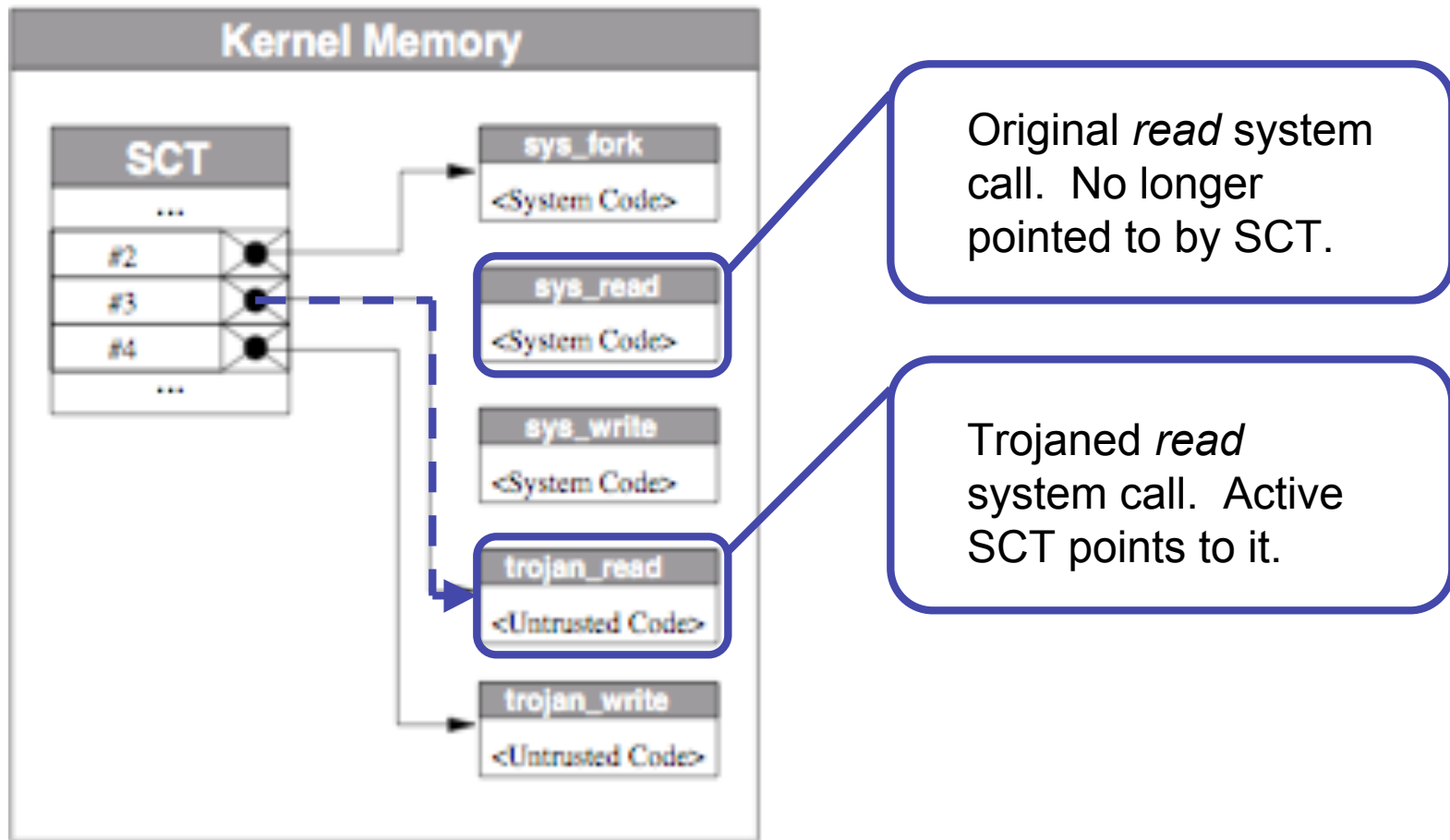
# User-Level versus Kernel-Level

- User-Level
  - Modify/replace system binaries
  - e.g. *ps*, *netstat*, *ls*, *top*, *passwd*
- Kernel-Level
  - Modify/replace kernel process
  - e.g. system call table

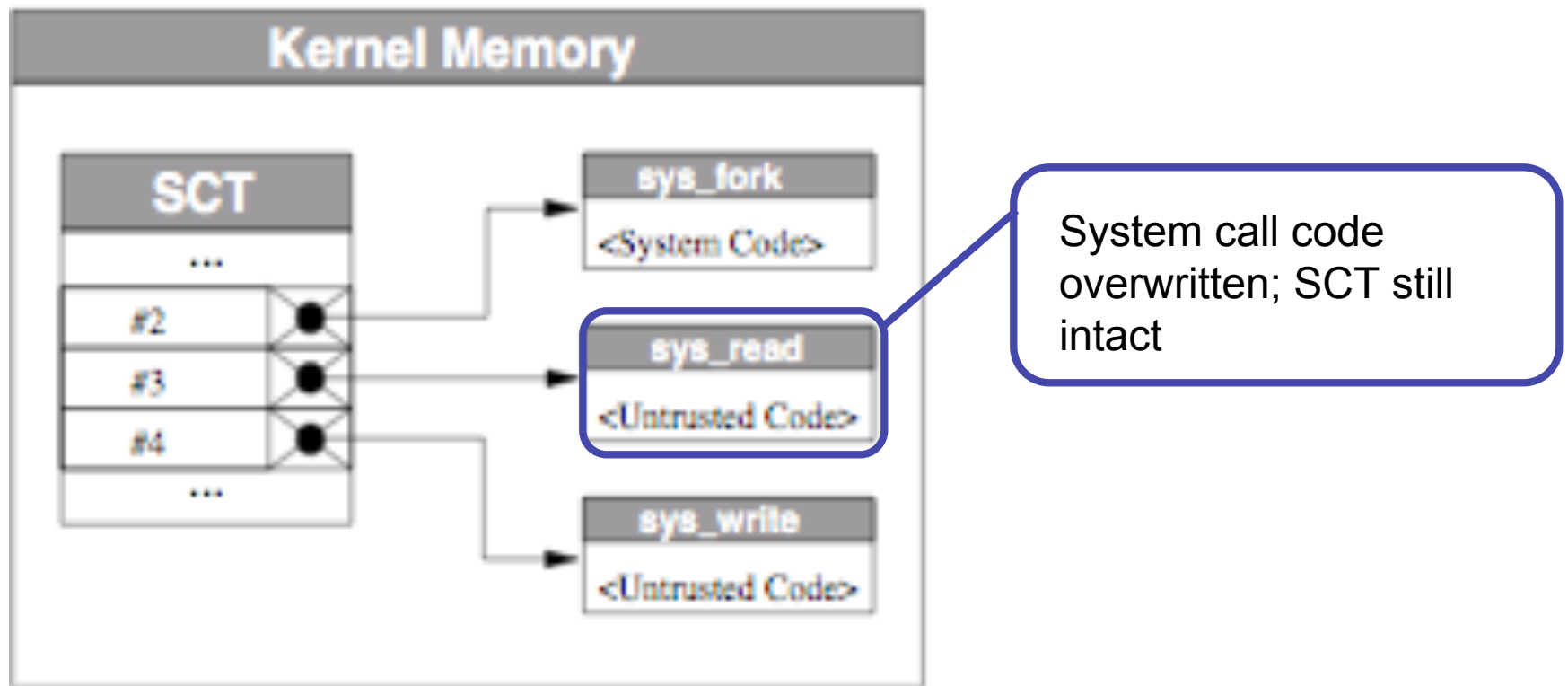
# Additional Malware Functionality

- Information harvesting
  - Credit cards
  - Bank accounts
- Resource usage
  - Spam relaying
  - Distributed denial of service

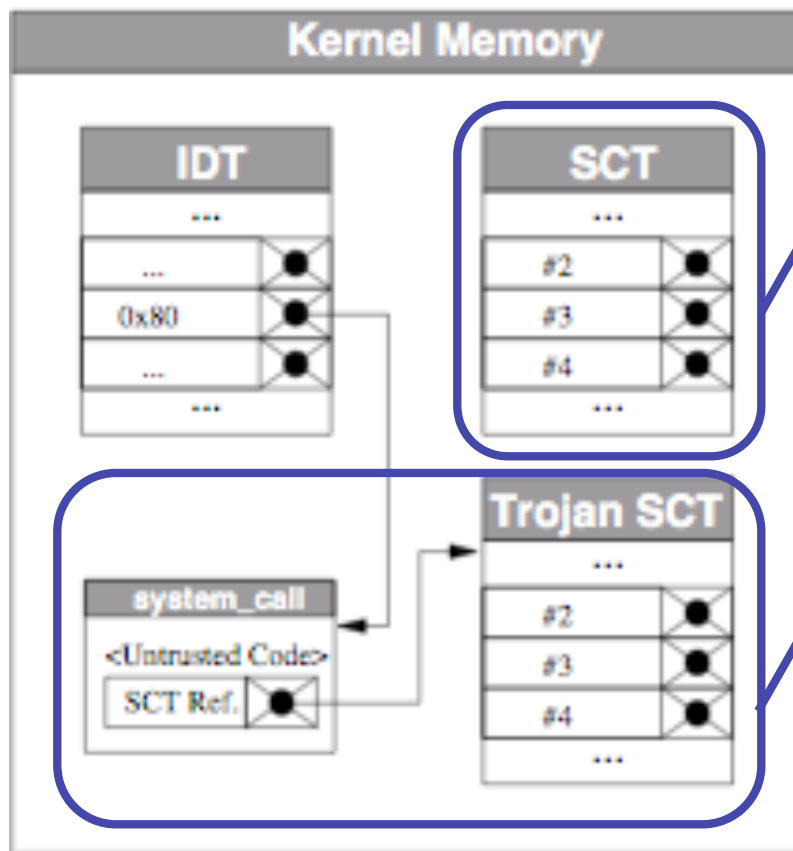
# Entry Redirection



# Entry Overwrite



# Table Redirection



- Original SCT intact

- Original system calls intact

- Handler points to Trojan table

# History of Kernel-Level Rootkits

- Heroin – October 1997
  - First public LKM
- Knark – June 1999
  - Highly popular LKM
- SuckKIT – December 2001
  - First public /dev/kmem entry
- Adore-ng 0.31 – January 2004
  - Uses VFS redirection; works on Linux 2.6.X

# Kernel-Level Rootkit Targets

- System call table
- Interrupt descriptor table
- Virtual file system layer
- Kernel data structures

# Kernel Entry

- Linux kernel module (LKM)
- /dev/kmem, /dev/mem, /dev/port
- Direct memory access (DMA)
- Modify kernel image on disk

# System Call Table Modifications

- System calls are the main gateway from user space to kernel space
- Most commonly targeted kernel structure
- Can redirect individual system calls or the entire table

# Example Kernel-Level Rootkits

<b>Rootkit</b>	<b>Kernel Entry</b>	<b>Modification</b>
<b>heroin</b>	Module	SCT Entry Redirection
<b>knark</b>	Module	SCT Entry Redirection
<b>adore</b>	Module	SCT Entry Redirection
<b>sucKIT</b>	kmem	SCT Table Redirection
<b>zk</b>	kmem	SCT Table Redirection
<b>r.tgz</b>	kmem	SCT Table Redirection
<b>adore-ng</b>	Module	VFS Redirection

# System Call Table Tools

- Developed tools that can query the state of the system call table and repair it
- Tools based on suckKIT source code and work from user space
- Algorithm to recover from rootkits is similar to algorithm used by rootkits

# Virtual Machines/Hypervisors

- VMware
- User Mode Linux
- Xen
- L4

# History of Microkernels

- Mach project started at CMU (1985)
- QNX
- Windows NT
- LynxOS
- Chorus
- Mac OS X

# Microkernel Requirements

- Tasks
- IPC
- I/O Support

That's it!

# L4 IPC's

- Fast IPCS
- Flexpages
- Clans and chiefs
- System calls, page faults are IPC's

# L4 I/O (from Fiasco lecture slides)

- Hardware interrupts: mapped to IPC
  - Special thread id for interrupts
  - IPC sender indicates interrupt source
  - Kernel provides no sharing support, one thread per interrupt
  - Malicious driver could potentially block all interrupts if given access to PIC
  - Cli/sti only allowed in kernel and trusted servers
- I/O memory and I/O ports: flexpages
- Missing kernel feature: pass interrupt association
  - Security hole
- I/O port access
- DMA - big security risk

# Rmgr (lecture slides)

- Resources --- serves page faults
  - Physical memory
  - I/O ports
  - Tasks
  - Interrupts

# Booting the System (lecture slides)

- Modified grub
- Multi-boot specification
- Rmgr, sigma0, root task (rmgr II), ...
- IDT
  - General Protection Exception #13
  - Page Fault #14
  - Divide by zero #0
  - Invalid opcode #6
  - System calls Int30 IPC
- Global Descriptor Table (GDT) vs. Local Descriptor Table (LDT)

# L4 Security Problems?

- Passing interrupt association
- Direct memory access
- Fill up page mapping database
- Kernel accessible on disk
- Cli/sti
- A few more...

# Spine Architecture Details

- Uses L4 Fiasco microkernel
- L4Linux runs on top of microkernel
- User tasks run on L4Linux
- Intrusion recovery system consists of levels 0 through 3

# L4Linux

- Port of Linux kernel to L4 architecture
- “paravirtualization” vs. pure virtualization
- Linux kernel runs in user space
- Binary compatible

# Intrusion Recovery System

- Capable of recovering from rootkit installations
- Maintain a copy of known good state to verify system integrity and repair if needed
- Must be integral part of operating system

# IRS Cont...

- Intrusion detection system is part of IRS
  - Must be able to detect that an intrusion has occurred in order to recover from it
- Most difficult part of problem is verifying system integrity
  - How to verify data structures, config files, etc.
- Another important challenge is verifying integrity of IRS itself
  - Malware has been known to disable IDS's

# Multi-Level IRS Reasoning

- Difficult to monitor state of entire system from one vantage point
- Difficulty comes in bridging the semantic gap between layers of the system
- We use a multi-level approach

# Multi-Levelled IRS Detail

- **L3** - verify file system state and repair if needed
- **L2** - kernel module to verify integrity of L4Linux and L3 and repair if needed
- **L1** - microkernel modifications to verify state of L2 and repair if needed; also provides secure storage for known good state
- **L0** - hardware support for maintaining isolation and verifying L1 (more hardware needed)