

ETH zürich spezial@ethz.ch @spci\_eth

**ADRIAN PERRIG & TORSTEN HOEFLER**  
**Networks and Operating Systems (252-0062-00)**  
**Chapter 2: Processes**

**RSA Key Extraction via Low-Bandwidth Acoustic Cryptanalysis**  
 Genkin, Shamir, Tromer, Dec. 2013

*"Here, we describe a new acoustic cryptanalysis key extraction attack, applicable to GnuPG's current implementation of RSA. The attack can extract full 4096-bit RSA decryption keys from laptop computers (of various models), within an hour, using the sound generated by the computer during the decryption of some chosen ciphertexts."*

<http://tau.ac.il/~tromer/acoustic/>

IF SOMEONE STEALS MY LAPTOP WHILE I'M LOGGED IN, THEY CAN READ MY EMAIL, TAKE MY MONEY AND IMPERSONATE ME TO MY FRIENDS, BUT AT LEAST THEY CAN'T INSTALL DRIVERS WITHOUT MY PERMISSION.  
 © source: xkcd.com

ETH zürich spezial@ethz.ch @spci\_eth

**Last time: introduction**

- **Introduction: Why?** → Löschesystem hätte intaktes Triebwerk "gelöscht"
- **Roles of the OS**
  - Referee
  - Illusionist
  - Glue
- **Structure of an OS**

Unglaublicher Fehler: Bei drei Dreamlinern waren Löschesysteme falsch verkabelt. Im Falle eines Brandes wäre nicht das in Flammen stehende, sondern das noch intakte Triebwerk gelöscht worden. Von Gerhard Hegmann

2

ETH zürich spezial@ethz.ch @spci\_eth

**This time**

- Entering and exiting the kernel
- Process concepts and lifecycle
- Context switching
- Process creation
- Kernel threads
- Kernel architecture
- System calls in more detail
- User-space threads

3

ETH zürich spezial@ethz.ch @spci\_eth

**Entering and exiting the kernel**

4

ETH zürich spezial@ethz.ch @spci\_eth

**When is the kernel entered?**

- System Startup
- Exception (aka. trap): caused by user program
- Interrupt: caused by "something else"
- System calls

▪ **Exception vs. Interrupt vs. System call** (analog technology quiz, raise hand)

- Division by zero
- Fork
- Incoming network packet
- Segmentation violation
- Read
- Keyboard input

5

ETH zürich spezial@ethz.ch @spci\_eth

**Recall: System Calls**

- RPC to the kernel
- Kernel is a series of syscall event handlers
- Mechanism is hardware-dependent

System calls 6

**ETH zürich** spoc.inf.ethz.ch @spoc\_eth

## System call arguments

Syscalls are *the* way a program requests services from the kernel.

Implementation varies:

- Passed in processor registers
- Stored in memory (address (pointer) in register)
- Pushed on the stack

- System library (libc) wraps as a C function
- Kernel code wraps handler as C call

7

**ETH zürich** spoc.inf.ethz.ch @spoc\_eth

## When is the kernel exited?

- **Creating a new process**
  - Including startup
- **Resuming a process after a trap**
  - Exception, interrupt or system call
- **User-level upcall**
  - Much like an interrupt, but to user-level
- **Switching to another process**

8

**ETH zürich** spoc.inf.ethz.ch @spoc\_eth

## Processes

9

**ETH zürich** spoc.inf.ethz.ch @spoc\_eth

## Process concept

“The execution of a program with restricted rights”

- **Virtual machine, of sorts**
- **On older systems:**
  - Single dedicated processor
  - Single address space
  - System calls for OS functions
- **In software:**  
computer system = (kernel + processes)

10

**ETH zürich** spoc.inf.ethz.ch @spoc\_eth

## Process ingredients

- **Virtual processor**
  - Address space
  - Registers
  - Instruction Pointer / Program Counter
- **Program text (object code)**
- **Program data (static, heap, stack)**
- **OS “stuff”:**
  - Open files, sockets, CPU share,
  - Security rights, etc.

11

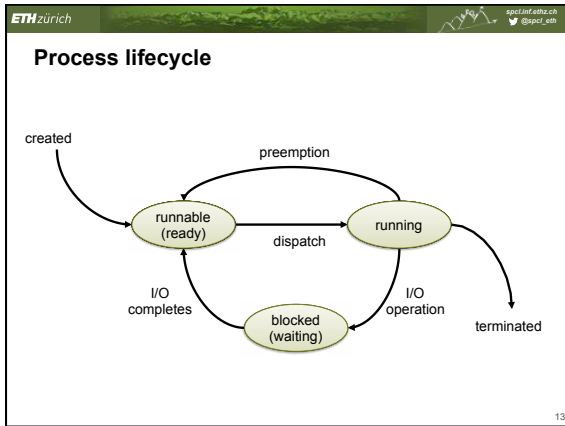
**ETH zürich** spoc.inf.ethz.ch @spoc\_eth

## Process address space

Should look familiar ...

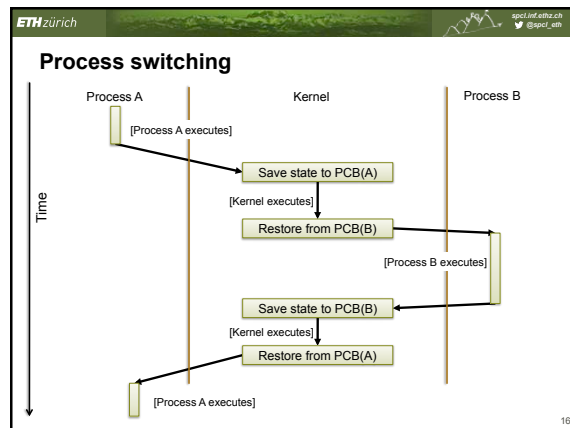
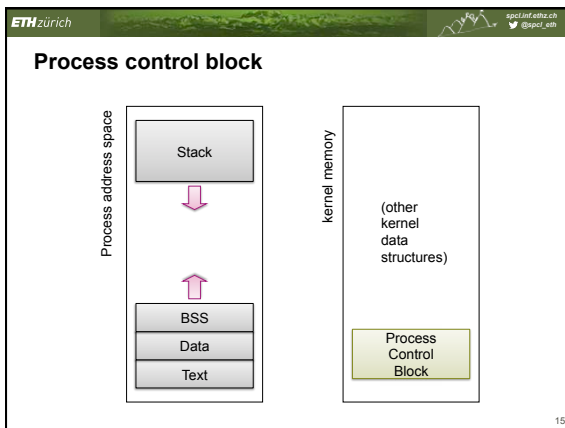
(addresses are examples: some machines used the top address bit to indicate kernel mode)

12



### Multiplexing

- OS time-division multiplexes processes
  - Or space-division on multiprocessors
- Each process has a **Process Control Block (PCB)**
  - In-kernel data structure
  - Holds all virtual processor state
  - Identifier and/or name
  - Registers
  - Memory used, pointer to page table
  - Files and sockets open, etc.



### Process Creation

### Process Creation

- Bootstrapping problem. Need:**
  - Code to run
  - Memory to run it in
  - Basic I/O set up (so you can talk to it)
  - Way to refer to the process
- Typically, "spawn" system call takes enough arguments to construct, from scratch, a new process.

### Process creation on Windows

Did it work?

```

BOOL CreateProcess(
    in_opt LPCTSTR ApplicationName,
    inout_opt LPCTSTR CommandLine,
    in_opt LPSECURITY_ATTRIBUTES ProcessAttributes,
    in_opt LPSECURITY_ATTRIBUTES ThreadAttributes,
    in BOOL InheritHandles,
    in DWORD CreationFlags,
    in_opt LPVOID Environment,
    in_opt LPCTSTR CurrentDirectory,
    in LPSTARTUPINFO StartupInfo,
    out LPPROCESS_INFORMATION ProcessInformation
);
    
```

Annotations:

- ApplicationName, CommandLine: What to run?
- ProcessAttributes, ThreadAttributes: What rights will it have?
- CurrentDirectory: What will it see when it starts up?
- ProcessInformation: The result

Moral: the parameter space is large!

### Unix fork () and exec ()

Dramatically simplifies creating processes:

1. fork () : creates "child" copy of calling process
2. exec () : replaces text of calling process with a new program
3. There is no "CreateProcess (...)"

Unix is entirely constructed as a family tree of such processes.

### Unix as a process tree

Exercise: work out how to do this on your favorite Unix or Linux machine...

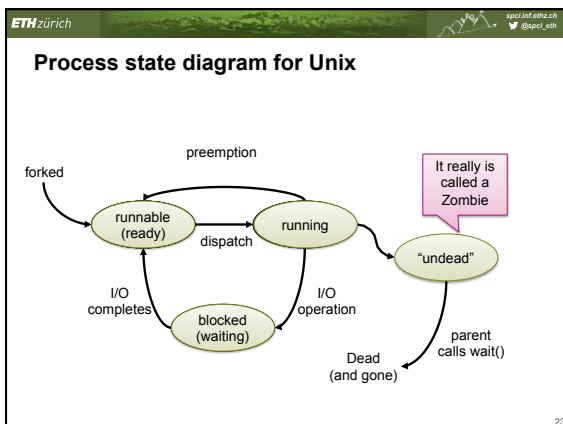
### Fork in action

```

pid_t p = fork();
if ( p < 0 ) {
    // Error...
    exit(-1);
} else if ( p == 0 ) {
    // We're in the child
    execlp("/bin/lis", "lis", NULL);
} else {
    // We're a parent.
    // p is the pid of the child
    wait(NULL);
    exit(0);
}
    
```

Annotations:

- Return code from fork() tells you whether you're in the parent or child (cf. setjmp())
- Child process can't actually be cleaned up until parent "waits" for it.



### Kernel Threads

**ETH zürich** spoc.inf.ethz.ch @spoc\_eth

### How do threads fit in?

- It depends...
- Types of threads:**
  - Kernel threads
  - One-to-one user-space threads
  - Many-to-one
  - Many-to-many
- Do NOT confuse this with hardware threads/SMT/Hyperthreading**
  - In these, the CPU offers more physical resources for threads!

25

**ETH zürich** spoc.inf.ethz.ch @spoc\_eth

### Kernel threads

- Kernels can (and some do) implement threads
- Multiple execution contexts inside the kernel**
  - Much as in a JVM
- Says nothing about user space**
  - Context switch still required to/from user process
- First, how many stacks are there in the kernel?**

26

**ETH zürich** spoc.inf.ethz.ch @spoc\_eth

### Process switching

Time ↓

Process A | Kernel | Process B

[Process A executes] → Save state to PCB(A) → Restore from PCB(B) → [Process B executes] → Save state to PCB(B) → Restore from PCB(A) → [Process A executes]

Kernel executes

What's happening here? A thread?

27

**ETH zürich** spoc.inf.ethz.ch @spoc\_eth

### Kernel architecture

- Basic Question: How many kernel stacks?**
- Unix 6<sup>th</sup> edition has a kernel stack per process**
  - Arguably complicates design
  - Q. On which stack does the thread scheduler run?
  - A. On the first thread (#1)
    - ⇒ Every context switch is actually *two*!
  - Linux et al. replicate this, and try to optimize it.
- Others (e.g., Barrelfish) have only one kernel stack per CPU**
  - Kernel must be purely event driven: no long-running kernel tasks
  - More efficient, less code, harder to program (some say).

28

**ETH zürich** spoc.inf.ethz.ch @spoc\_eth

### Process switching revisited

Process A | Kernel stack A | Kernel stack 0 | Kernel stack B | Process B

Save to PCB(A) → Decide to switch process → Pick process to run → Switch to Kernel stack B → Restore PCB(B)

For a kernel with multiple kernel stacks

With cleverness, can sometimes run scheduler on current process' kernel stack.


29

**ETH zürich** spoc.inf.ethz.ch @spoc\_eth

### System Calls in more detail

- We can now say in more detail what happens during a system call**
- Precise details are very dependent on OS and hardware**
  - Linux has 3 different ways to do this for 32-bit x86 *alone*!
- Linux:**
  - Good old int 0x80 or 0x2e (software interrupt, syscall number in EAX)
    - Set up registers and call handler
  - Fast system calls (sysenter/sysexit, >Pentium II)
    - CPU sets up registers automatically

30


ETH zürich  [spoc.inf.ethz.ch](http://spoc.inf.ethz.ch)  
@spoc\_eth

## Performing a system call

**In user space:**

1. Marshal the arguments somewhere safe
2. Saves registers
3. Loads system call number
4. Executes SYSCALL instruction (or SYSENTER, or INT 0x80, or..)
5. And?


31

ETH zürich  [spoc.inf.ethz.ch](http://spoc.inf.ethz.ch)  
@spoc\_eth

## System calls in the kernel

- **Kernel entered at fixed address**
  - Privileged mode is set
- **Need to call the right function and return, so:**
  1. Save user stack pointer and return address
    - In the Process Control Block
  2. Load SP for this process' kernel stack
  3. Create a C stack frame on the kernel stack
  4. Look up the syscall number in a jump table
  5. Call the function (e.g. `read()`, `getpid()`, `open()`, etc.)


32

ETH zürich  [spoc.inf.ethz.ch](http://spoc.inf.ethz.ch)  
@spoc\_eth

## Returning in the kernel


- **When function returns:**
  1. Load the user space stack pointer
  2. Adjust the return address to point to:
    - Return path in user space back from the call, OR*
    - Loop to retry system call if necessary*
  3. Execute "syscall return" instruction
- **Result is execution back in user space, on user stack.**
- **Alternatively, can do this to a different process...**

33

ETH zürich  [spoc.inf.ethz.ch](http://spoc.inf.ethz.ch)  
@spoc\_eth

## User-space threads


34

ETH zürich  [spoc.inf.ethz.ch](http://spoc.inf.ethz.ch)  
@spoc\_eth

## From now on assume:

- **Previous example was Unix 6<sup>th</sup> Edition:**
  - Which had *no* threads *per se*, only processes
  - i.e. Process ↔ Kernel stack
- **From now on, we'll assume:**
  - Multiple kernel threads per CPU
  - Efficient kernel context switching
- **How do we implement user-visible threads?**

35

ETH zürich  [spoc.inf.ethz.ch](http://spoc.inf.ethz.ch)  
@spoc\_eth

## What are the options?

1. Implement threads within a process
2. Multiple kernel threads in a process
3. Some combination of the above

- and other more unusual cases we won't talk about...

36

**ETH zürich**

### Many-to-one threads

- **Early "thread libraries"**
  - Green threads (original Java VM)
  - GNU Portable Threads
  - Standard student exercise: implement them!
- **Sometimes called "pure user-level threads"**
  - No kernel support required
  - Also (confusingly) "Lightweight Processes"

37

**ETH zürich**

### Many-to-one threads

The diagram shows a horizontal line separating the 'User' space from the 'Kernel' space. Above the line, there are three groups of three wavy boxes representing user threads. Below the line, there are three single wavy boxes representing kernel threads. Vertical lines separate the three groups. Arrows point from each group of three user threads down to its corresponding kernel thread. Below the kernel threads, two ovals labeled 'CPU 0' and 'CPU 1' are shown, with the kernel threads positioned between them.

38

**ETH zürich**

### Address space layout for user level threads

The diagram shows two memory layouts. On the left, a process layout with a 'Stack' at the top, followed by 'BSS', 'Data', and 'Text' at the bottom. A pink arrow points from the 'Stack' to the 'BSS' section. On the right, a multi-threaded layout where the 'Text', 'Data', and 'BSS' sections are shared at the bottom, and each thread has its own 'Thread 1 stack', 'Thread 2 stack', and 'Thread 3 stack' above them. A pink arrow points from the 'BSS' section up to the 'Thread 3 stack'. A pink callout box with an arrow pointing to the 'Thread 3 stack' contains the text 'Just allocate on the heap'.

39

**ETH zürich**

### One-to-one user threads

- **Every user thread is/has a kernel thread.**
- **Equivalent to:**
  - multiple processes sharing an address space
  - Except that "process" now refers to a group of threads
- **Most modern OS threads packages:**
  - Linux, Solaris, Windows XP, MacOSX, etc.

40

**ETH zürich**

### One-to-one user threads

The diagram shows a horizontal line separating the 'User' space from the 'Kernel' space. Above the line, there are six wavy boxes representing user threads. Below the line, there are six wavy boxes representing kernel threads. Vertical lines separate the three groups of two threads. Arrows point from each user thread down to its corresponding kernel thread. Below the kernel threads, two ovals labeled 'CPU 0' and 'CPU 1' are shown, with the kernel threads positioned between them.

41

**ETH zürich**

### One-to-one user threads

The diagram shows two memory layouts. On the left, a process layout with a 'Stack' at the top, followed by 'BSS', 'Data', and 'Text' at the bottom. A pink arrow points from the 'Stack' to the 'BSS' section. On the right, a multi-threaded layout where the 'Text', 'Data', and 'BSS' sections are shared at the bottom, and each thread has its own 'Thread 1 stack', 'Thread 2 stack', and 'Thread 3 stack' above them. A pink arrow points from the 'BSS' section up to the 'Thread 3 stack'.

42

ETH zürich spoc.inf.ethz.ch @spoc\_eth

### Comparison

|  |   |
|--|---|
| <p><b>User-level threads</b></p> <ul style="list-style-type: none"> <li>▪ Cheap to create and destroy</li> <li>▪ Fast to context switch</li> <li>▪ Can block entire process</li> <li>▪ Not just on system calls</li> </ul> | <p><b>One-to-one threads</b></p> <ul style="list-style-type: none"> <li>▪ Memory usage (kernel stack)</li> <li>▪ Slow to switch</li> <li>▪ Easier to schedule</li> <li>▪ Nicely handles blocking</li> </ul> |
|--|---|

43

ETH zürich spoc.inf.ethz.ch @spoc\_eth

### Many-to-many threads

- Multiplex user-level threads over several kernel-level threads
- Only way to go for a multiprocessor
  - I.e., pretty much everything these days
- Can “pin” user thread to kernel thread for performance/predictability
- Thread migration costs are “interesting”...

44

ETH zürich spoc.inf.ethz.ch @spoc\_eth

### Many-to-many threads

45

ETH zürich spoc.inf.ethz.ch @spoc\_eth

### Next week

- **Synchronisation:**
  - How to implement those useful primitives
- **Interprocess communication**
  - How processes communicate
- **Scheduling:**
  - Now we can pick a new process/thread to run, how do we decide which one?

46