

ETH zürich spoc.inf.ethz.ch @spoc_eth

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

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 spoc.inf.ethz.ch @spoc_eth

Last time: introduction

- **Introduction: Why?**
- **Roles of the OS**
 - Referee
 - Illusionist
 - Glue
- **Structure of an OS**

2

ETH zürich spoc.inf.ethz.ch @spoc_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 spoc.inf.ethz.ch @spoc_eth

Entering and exiting the kernel

4

ETH zürich spoc.inf.ethz.ch @spoc_eth

When is the kernel entered?

- **System Startup**
- **Exception: 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 spoc.inf.ethz.ch @spoc_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

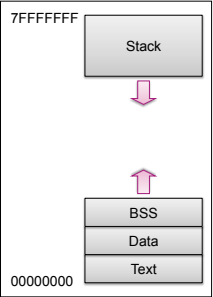
Process ingredients

- 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

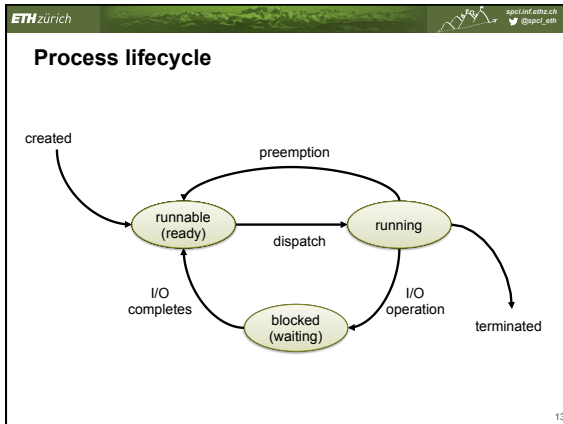
Process address space



Should look familiar ...

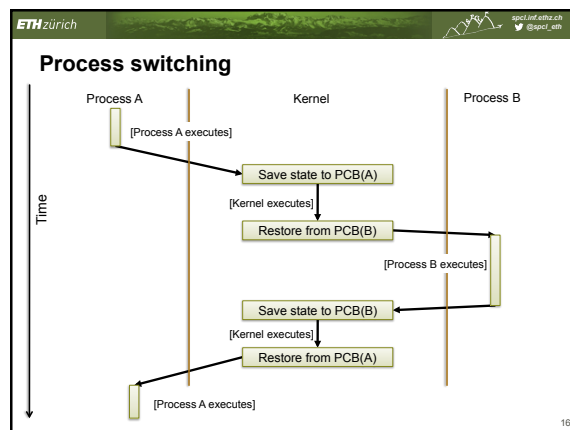
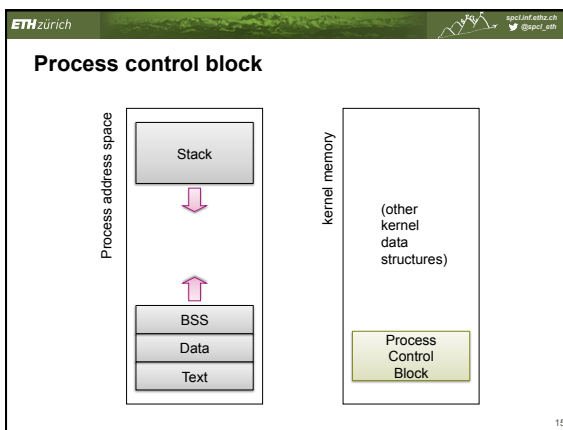
(addresses are examples: some machines use the top address bit to indicate kernel mode)

12



Multiplexing

- OS time-division multiplexes processes
 - Or space-division on multiprocessors
- Each process has a **Process Control Block**
 - 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
);
    
```

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

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

PPID	PID	PPID	SID	TTY	TRGID	STAT	UID	TIME	COMMAND
0	1	437	436	436 ?	-1	Ss	0	0:01	/sbin/init
437	439	439	439 ?		-1	Ss	0	0:00	upstart-udev-bridge --daemon
439	2096	439	439 ?		-1	Ss	0	0:00	udevd --daemon
439	2098	439	439 ?		-1	Ss	0	0:00	udevd --daemon
1	657	657	657 ?		-1	Ss	0	0:00	dbus-daemon --fork --print-debug
1	654	659	659 ?		-1	Ss	101	0:00	rsyslogd -c
1	678	678	678 ?		-1	Ss	108	0:00	dbus-daemon --system --fork
729	745	745	745 ?		-1	Ss	110	0:00	awkd-daemon: chroot helper
1	731	731	731 ?		-1	Ss	111	0:00	haldaemon
731	763	731	731 ?		-1	Ss	0	0:00	haldaemon
851	1044	731	731 ?		-1	Ss	0	0:00	/usr/lib/hal/hal-daemon-rl-1-1-11
851	1045	731	731 ?		-1	Ss	0	0:00	/usr/lib/hal/hal-daemon-libs
851	1050	731	731 ?		-1	Ss	0	0:00	/usr/lib/hal/hal-daemon-gnome-c-bac
851	1074	731	731 ?		-1	Ss	0	0:00	hal-daemon-storage: polling /dev/sd
851	1086	731	731 ?		-1	Ss	0	0:00	hal-daemon-pipe: listening on dbus
851	1100	731	731 ?		-1	Ss	0	0:00	/usr/lib/hal/hal-daemon-udevreq
851	1101	731	731 ?		-1	Ss	0	0:00	hal-daemon-usb: listening on aspid
741	1463	1463	740 ?		-1	SaI	0	0:02	NetworkManager
751	869	751	751 ?		-1	Ss	0	0:00	dbus-daemon --system --fork
889	1102	1102	1102 tty?		1102	Ra	0	0:00	gdm-session
889	1346	751	751 ?		-1	Ss	0	0:00	/usr/lib/gdm/gdm-session-window
1346	1361	1361	1361 ?		-1	SaI	1000	0:00	gdm-session
1361	1413	1413	1413 ?		-1	Ss	1000	0:00	/usr/bin/ssh-agent /usr/bin
1361	1446	1446	1446 ?		-1	Ss	1000	0:00	/usr/bin/ssh-agent --n
1361	1289	1361	1361 ?		-1	Ss	1000	0:00	/bin/sh /usr/bin/cowsay
1789	1904	1361	1361 ?		-1	R	1000	0:48	/usr/bin/cowsay: read --
1361	1994	1994	1994 ?		-1	Ss	1000	0:00	/usr/bin/cowsay
1361	1998	1994	1994 ?		-1	Ss	1000	0:11	/usr/bin/cowsay
1361	1995	1361	1361 ?		-1	Ss	1000	0:00	gdm-session
1361	1997	1361	1361 ?		-1	Ss	1000	0:04	mail
1361	1312	1361	1361 ?		-1	Ss	1000	0:00	gdm-session-manager
1361	1313	1361	1361 ?		-1	Ss	1000	0:00	/usr/lib/evolution/2.32/ewo
1361	1316	1361	1361 ?		-1	Ss	1000	0:00	/usr/lib/evolution/2.32/gnome
1361	1317	1361	1361 ?		-1	Ss	1000	0:00	bluezobex-aptel
1361	1318	1361	1361 ?		-1	Ss	1000	0:00	update-notifier --startur=
1361	1321	1361	1361 ?		-1	Ss	1000	0:00	python /usr/share/system-co
1361	1323	1361	1361 ?		-1	Ss	1000	0:00	/usr/lib/gnome-disk-util/111

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

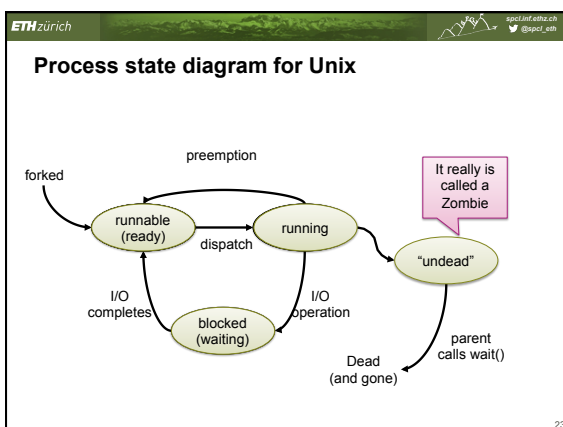
Fork in action

```

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

Return code from fork() tells you whether you're in the parent or child (c.f. 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

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

27

ETH zürich spoc.inf.ethz.ch @spoc_eth

Kernel architecture

- Basic Question: How many kernel stacks?**
- Unix 6th edition has a kernel stack per process**
 - Arguably complicates design
 - Q. On which thread 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

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*!

30


ETH zürich  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
@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
@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
@spoc_eth

User-space threads


34

ETH zürich  spoc.inf.ethz.ch
@spoc_eth

From now on assume:

- **Previous example was Unix 6th 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
@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 spoc.inf.ethz.ch @spoc_eth

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 spoc.inf.ethz.ch @spoc_eth

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 for each CPU connected to it.

38

ETH zürich spoc.inf.ethz.ch @spoc_eth

Address space layout for user level threads

The diagram shows two memory layouts. On the left, a single process layout with 'Stack' at the top, 'BSS', 'Data', and 'Text' at the bottom. A pink arrow points from the 'Stack' to the 'BSS' area. On the right, a multi-threaded layout where the 'Text', 'Data', and 'BSS' are shared at the bottom, and each thread has its own 'Thread X stack' above. A pink arrow points from the 'BSS' area up to the 'Thread 3 stack'. A pink callout box says 'Just allocate on the heap' with an arrow pointing to the space between the shared BSS/Data and the individual thread stacks.

39

ETH zürich spoc.inf.ethz.ch @spoc_eth

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 spoc.inf.ethz.ch @spoc_eth

One-to-one 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 for each CPU connected to it.


41

ETH zürich spoc.inf.ethz.ch @spoc_eth

One-to-one user threads

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


42

ETH zürich  spoc.inf.ethz.ch
@spoc_eth

Comparison

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

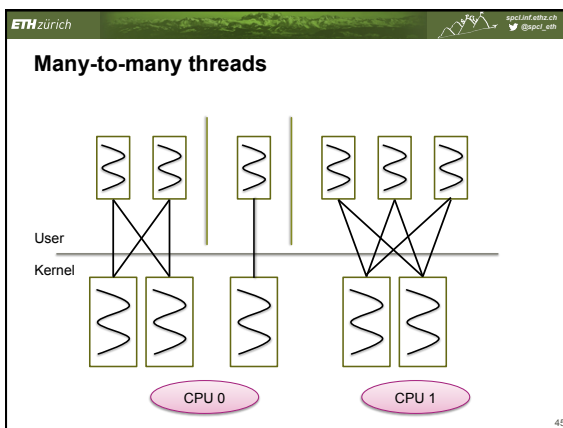
43


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

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