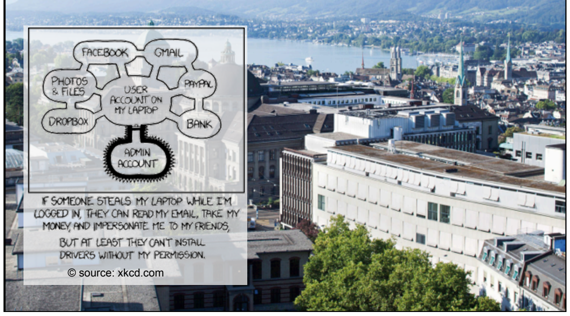
ADRIAN PERRIG & TORSTEN HOEFLER

Networks and Operating Systems (252-0062-00)
Chapter 2: Processes



© source: xkcd.com

Last time: introduction

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

2

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

Entering and exiting the kernel

4

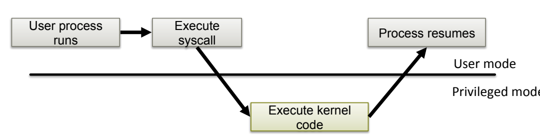
When is the kernel entered?

- 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

Recall: System Calls

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



System calls 6

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

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

Processes

9

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)

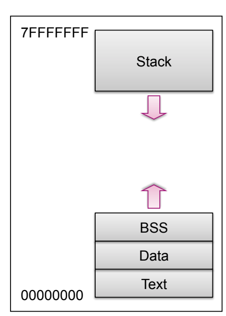
10

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

Process address space

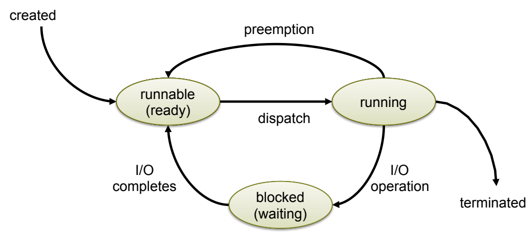


Should look familiar …

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

12

Process lifecycle



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

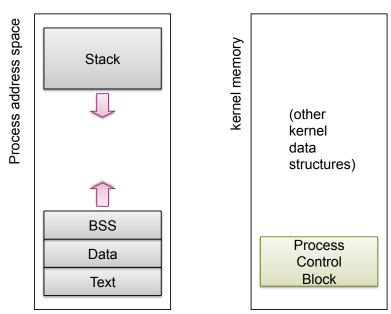
    Identifier and/or name

    Registers

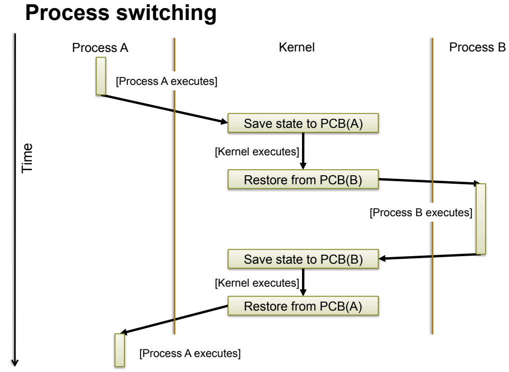
    Memory used, pointer to page table

    Files and sockets open, etc.

Process control block



Process switching



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   LPTSTR          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 … CreationFlags)
- What will it see when it starts up? (Environment, CurrentDirectory, StartupInfo)
- The result (ProcessInformation)

Moral: the parameter space is large!

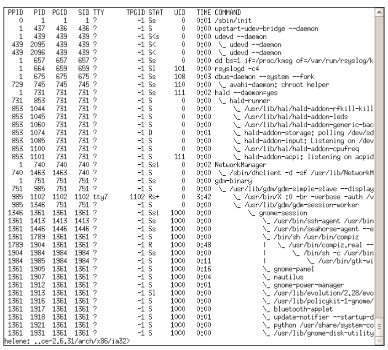
Unix `fork()` and `exec()`

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



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

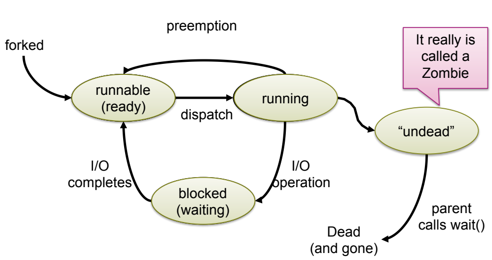
Fork in action

```
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.

Process state diagram for Unix



Kernel Threads

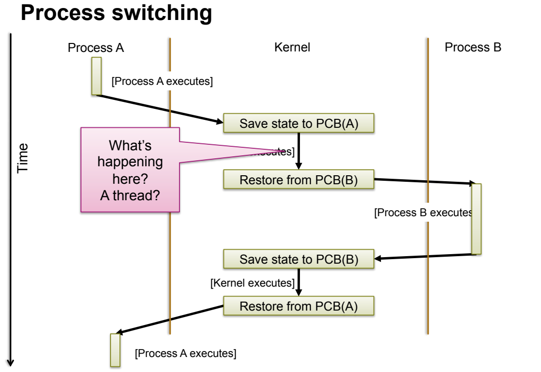
How do threads fit in?

- **It depends…**
- **Types of threads:**
  - Kernel threads
  - One-to-one user-space threads
  - Many-to-one
  - Many-to-many

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?**

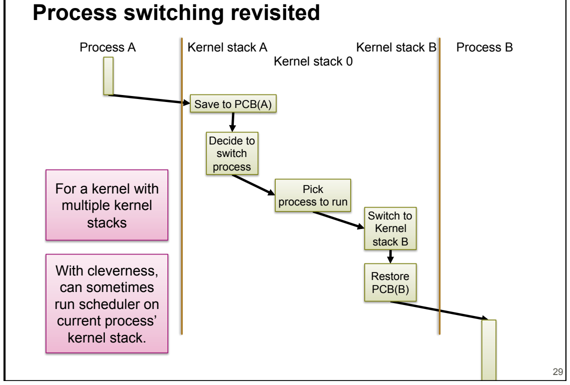
Process switching



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).

Process switching revisited



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!*

ETH zürich spcl.inf.ethz.ch @spcl_eth

Performing a system call

In user space:

1. Marshall 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 spcl.inf.ethz.ch @spcl_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 spcl.inf.ethz.ch @spcl_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 spcl.inf.ethz.ch @spcl_eth

User-space threads

34

ETH zürich spcl.inf.ethz.ch @spcl_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 spcl.inf.ethz.ch @spcl_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…**

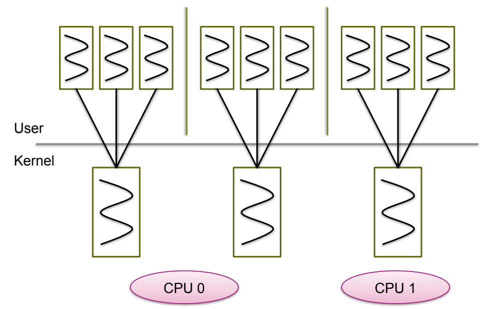
36

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"

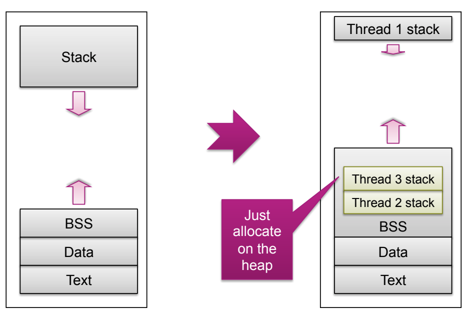
37

Many-to-one threads



38

Address space layout for user level threads



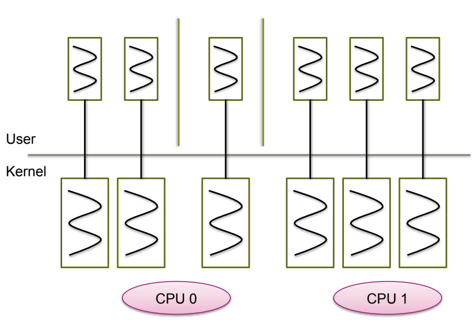
39

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.

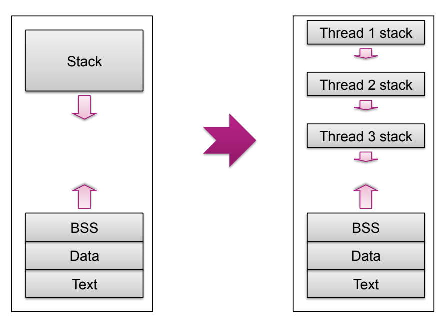
40

One-to-one threads



41

One-to-one user threads



42

Comparison

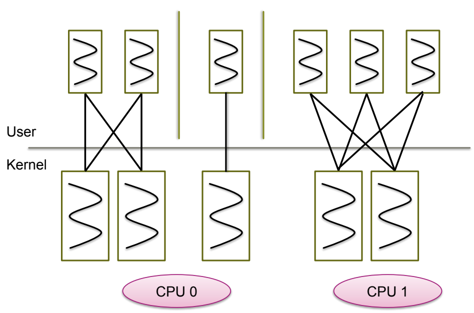
User-level threads
- **Cheap to create and destroy**
- **Fast to context switch**
- **Can block entire process**
- **Not just on system calls**

One-to-one threads
- **Memory usage (kernel stack)**
- **Slow to switch**
- **Easier to schedule**
- **Nicely handles blocking**

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"…**

Many-to-many threads



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?