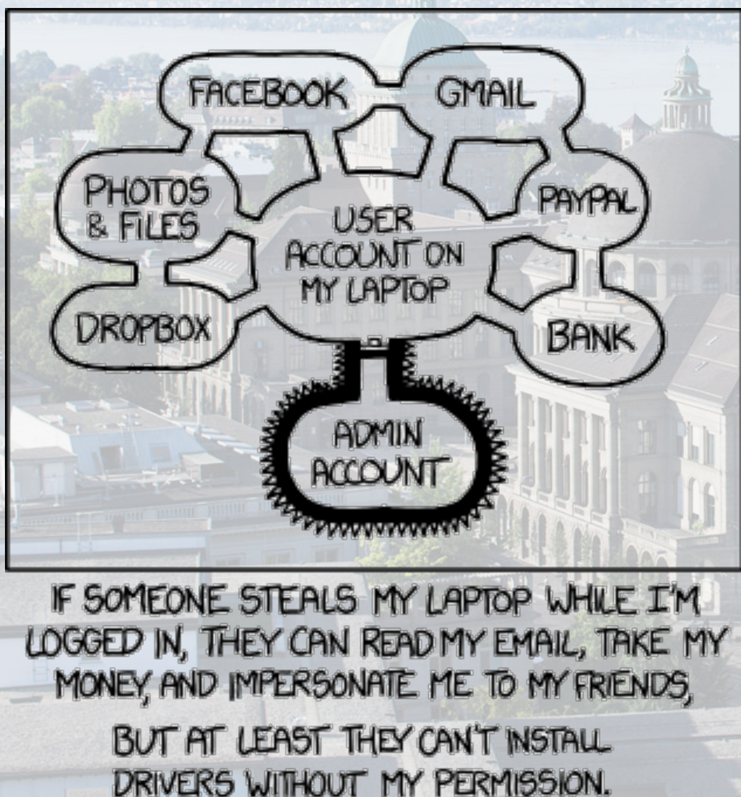
ADRIAN PERRIG & TORSTEN HOEFLER

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



© source: xkcd.com

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/

Last time: introduction

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

Löschsystem hätte intaktes Triebwerk "gelöscht"

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

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

Entering and exiting the kernel

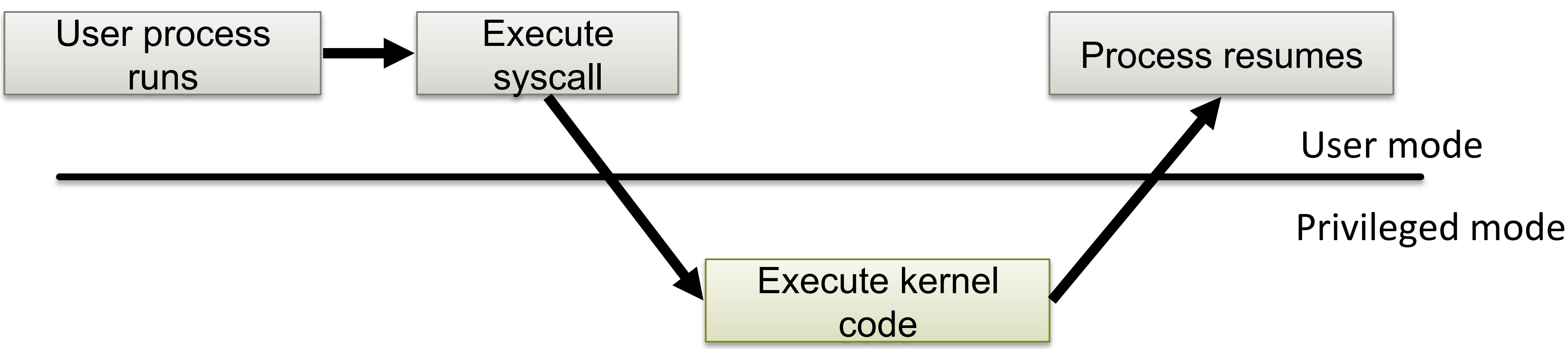
When is the kernel entered?

- **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

Recall: System Calls

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



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**

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***

Processes

Process concept

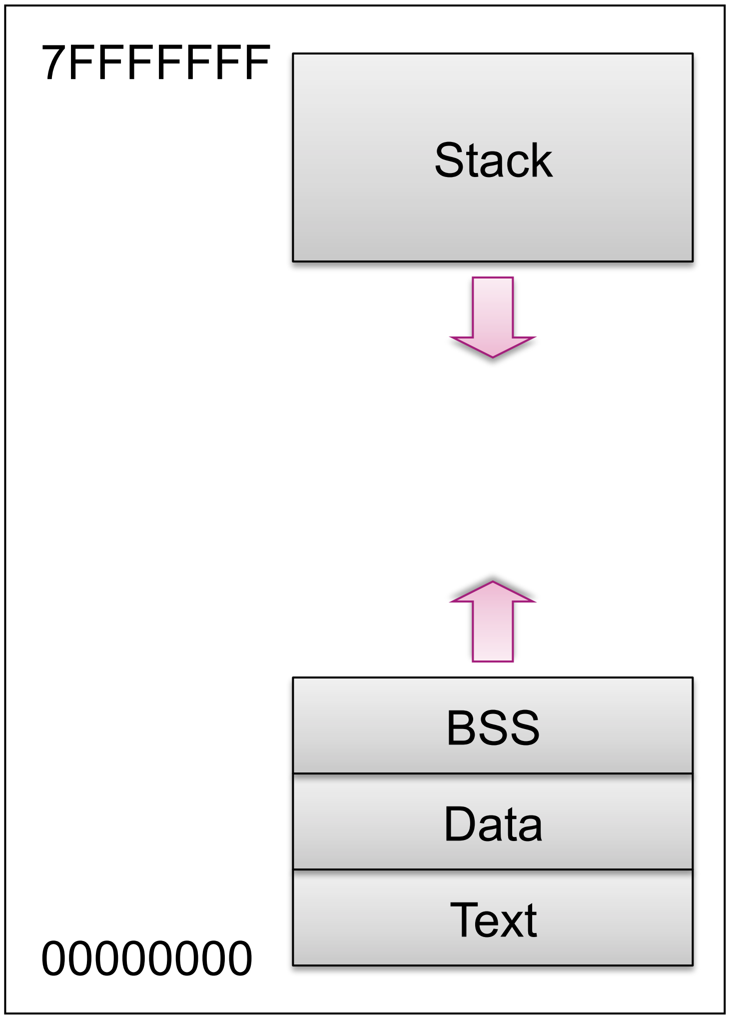
"The execution of a program with restricted rights"

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

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.

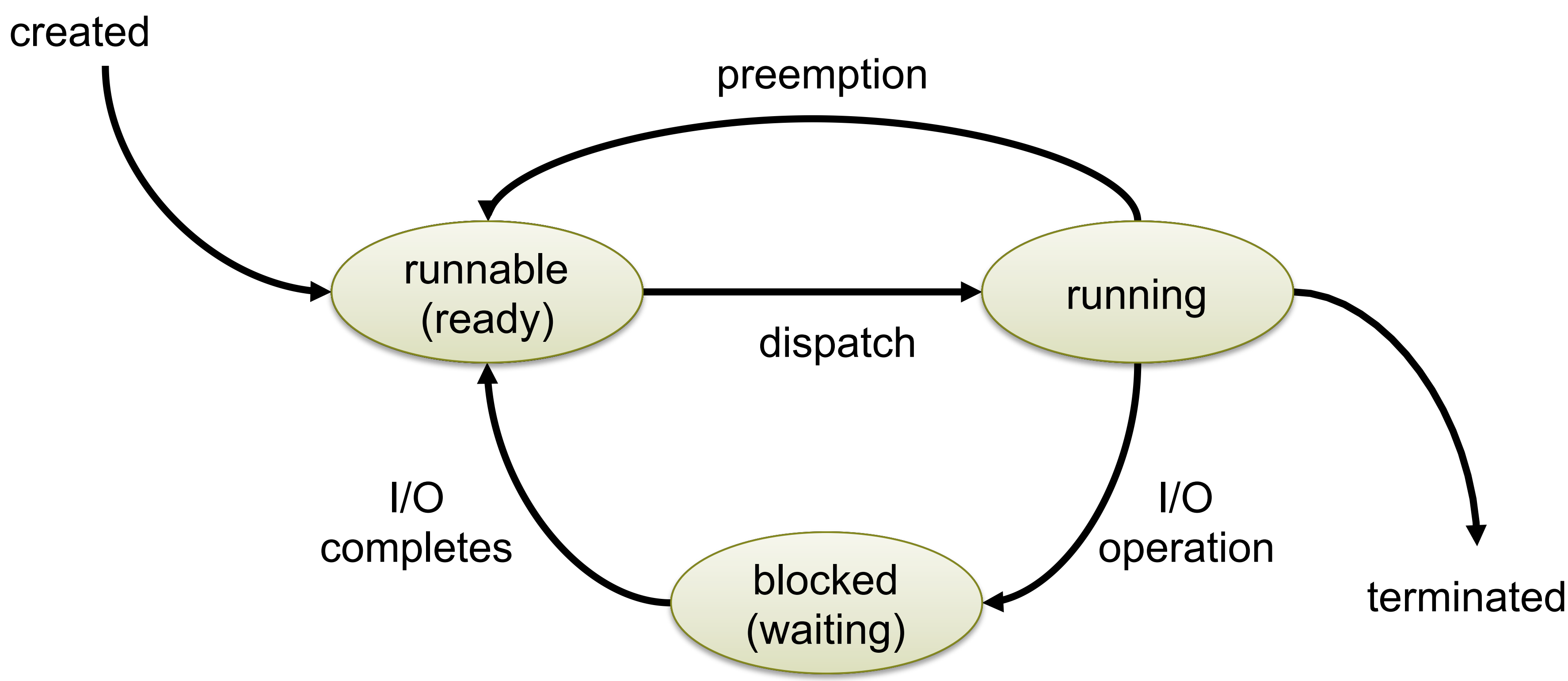
Process address space



Should look familiar …

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

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

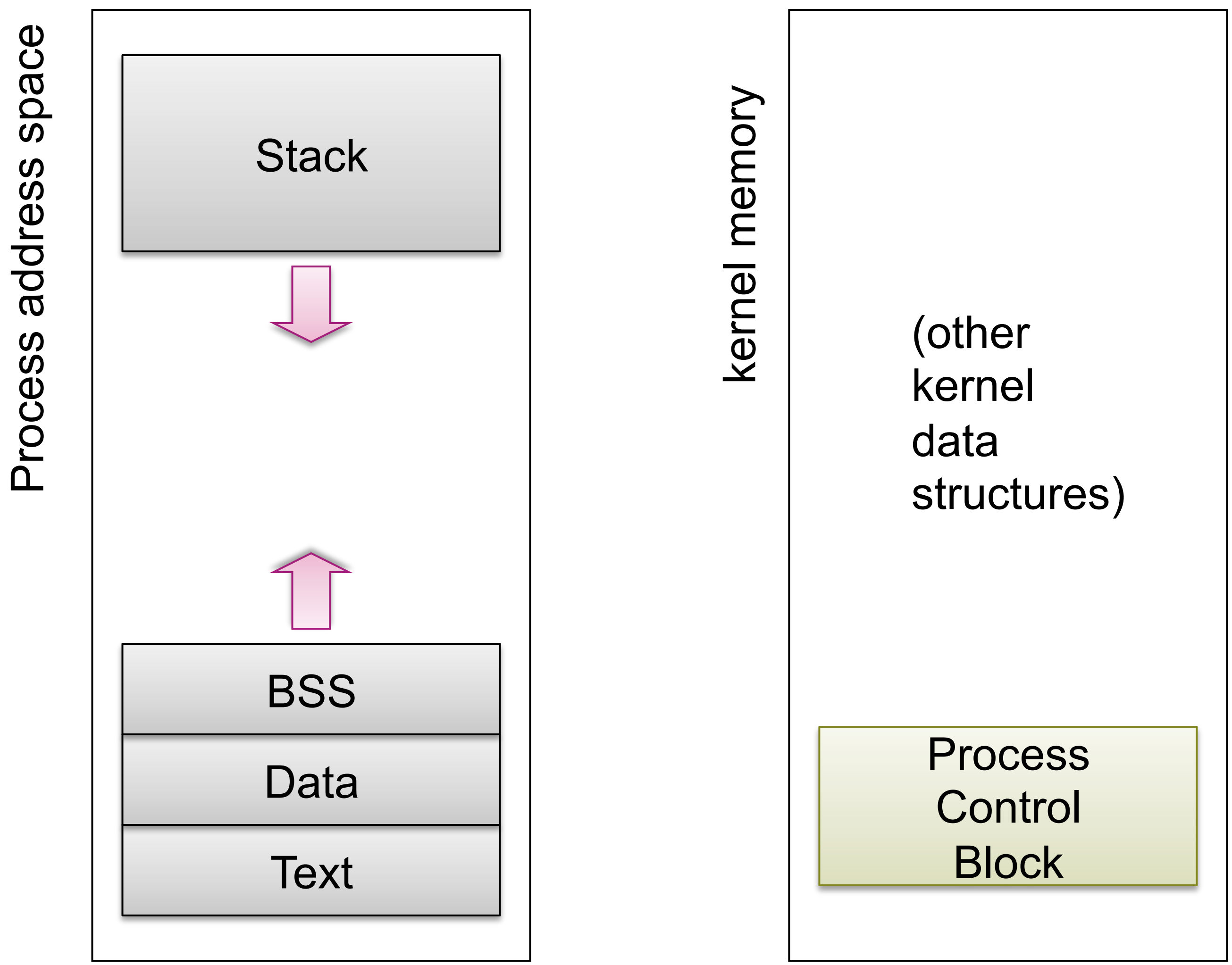
    Identifier and/or name

    Registers

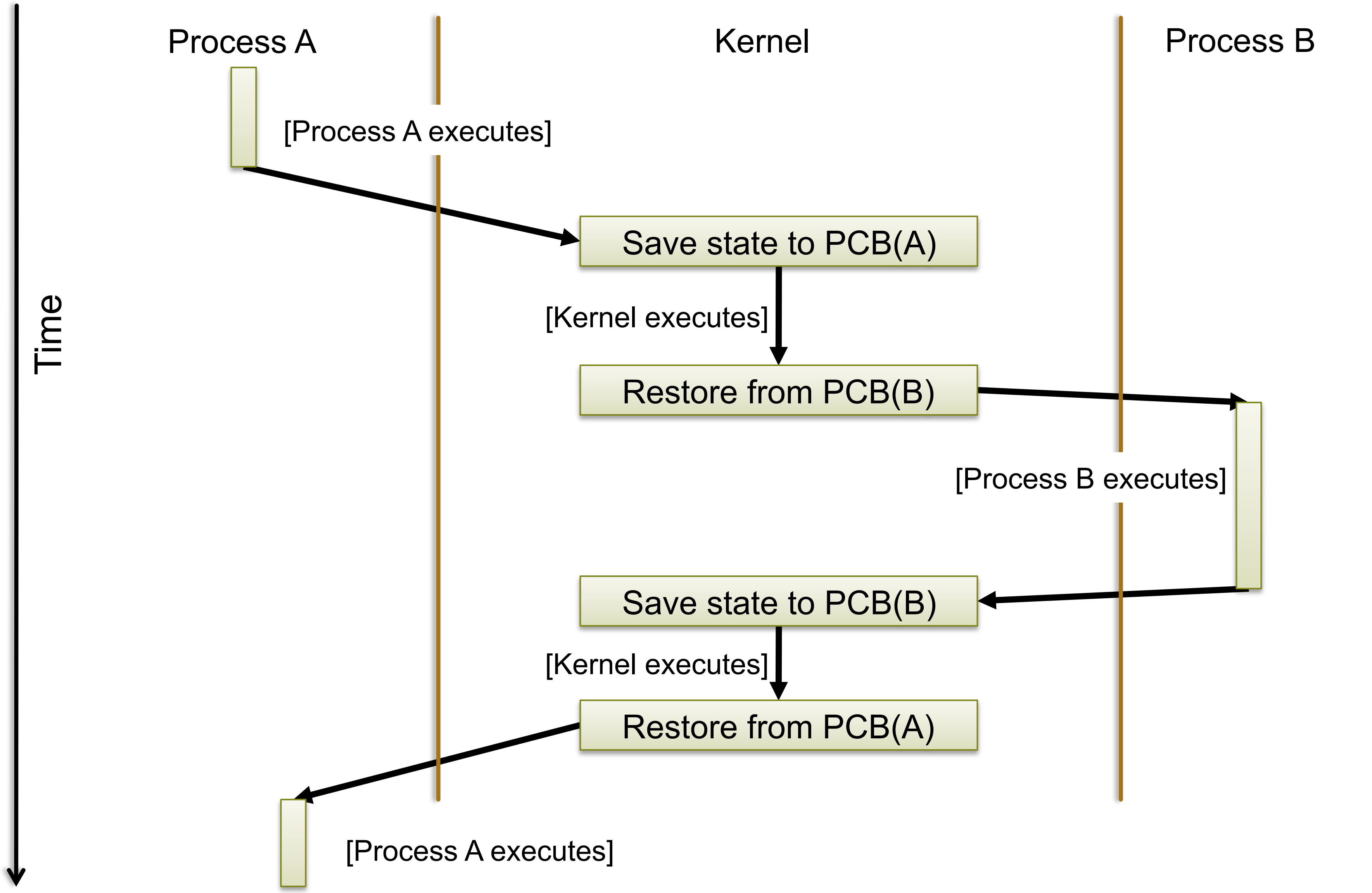
    Memory used, pointer to page table

    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, ThreadAttributes, InheritHandles, 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

```
PPID   PID  PGID   SID TTY      TPGID STAT   UID   TIME COMMAND
   0     1     1     1 ?           -1 Ss       0   0:01 /sbin/init
   1   437   436   436 ?           -1 S        0   0:00 upstart-udev-bridge --daemon
   1   439   439   439 ?           -1 S<s      0   0:00 udevd --daemon
 439  2095   439   439 ?           -1 S<       0   0:00  \_ udevd --daemon
 439  2096   439   439 ?           -1 S<       0   0:00  \_ udevd --daemon
   1   657   657   657 ?           -1 Ss       0   0:00 dd bs=1 if=/proc/kmsg of=/var/run/rsyslog/k
   1   664   659   659 ?           -1 Sl     101   0:00 rsyslogd -c4
   1   675   675   675 ?           -1 Ss     108   0:03 dbus-daemon --system --fork
 729   745   745   745 ?           -1 Ss     110   0:00  \_ avahi-daemon: chroot helper
   1   731   731   731 ?           -1 Ss     111   0:02 hald --daemon=yes
 731   853   731   731 ?           -1 S        0   0:00  \_ hald-runner
 853  1044   731   731 ?           -1 S        0   0:00      \_ /usr/lib/hal/hald-addon-rfkill-kill
 853  1045   731   731 ?           -1 S        0   0:00      \_ /usr/lib/hal/hald-addon-leds
 853  1060   731   731 ?           -1 S        0   0:00      \_ /usr/lib/hal/hald-addon-generic-bac
 853  1074   731   731 ?           -1 D        0   0:01      \_ hald-addon-storage: polling /dev/sd
 853  1085   731   731 ?           -1 S        0   0:00      \_ hald-addon-input: Listening on /dev
 853  1100   731   731 ?           -1 S        0   0:00      \_ /usr/lib/hal/hald-addon-cpufreq
 853  1101   731   731 ?           -1 S      111   0:00      \_ hald-addon-acpi: listening on acpid
   1   740   740   740 ?           -1 Ssl      0   0:02 NetworkManager
 740  1463  1463   740 ?           -1 S        0   0:00  \_ /sbin/dhclient -d -sf /usr/lib/NetworkM
   1   751   751   751 ?           -1 Ss       0   0:00 gdm-binary
 751   985   751   751 ?           -1 S        0   0:00  \_ /usr/lib/gdm/gdm-simple-slave --display
 985  1102  1102  1102 tty7      1102 Rs+      0   3:42      \_ /usr/bin/X :0 -br -verbose -auth /v
 985  1346   751   751 ?           -1 S        0   0:00      \_ /usr/lib/gdm/gdm-session-worker
1346  1361  1361  1361 ?           -1 Ssl   1000   0:00          \_ gnome-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/seahorse-agent --e
1361  1789  1361  1361 ?           -1 S     1000   0:00              \_ /bin/sh /usr/bin/compiz
1789  1904  1361  1361 ?           -1 R     1000   0:48              |   \_ /usr/bin/compiz.real --
1904  1984  1984  1984 ?           -1 Ss    1000   0:00              |       \_ /bin/sh -c /usr/bin
1984  1985  1984  1984 ?           -1 S     1000   0:11              |           \_ /usr/bin/gtk-wi
1361  1905  1361  1361 ?           -1 S     1000   0:16              \_ gnome-panel
1361  1907  1361  1361 ?           -1 S     1000   0:04              \_ nautilus
1361  1912  1361  1361 ?           -1 S     1000   0:01              \_ gnome-power-manager
1361  1913  1361  1361 ?           -1 Sl    1000   0:00              \_ /usr/lib/evolution/2.28/evo
1361  1916  1361  1361 ?           -1 S     1000   0:00              \_ /usr/lib/policykit-1-gnome/
1361  1917  1361  1361 ?           -1 S     1000   0:00              \_ bluetooth-applet
1361  1918  1361  1361 ?           -1 S     1000   0:01              \_ update-notifier --startup-d
1361  1921  1361  1361 ?           -1 S     1000   0:00              \_ python /usr/share/system-co
1361  1931  1361  1361 ?           -1 S     1000   0:00              \_ /usr/lib/gnome-disk-utility
helene: ..ce-2.6.31/arch/x86/ia32>
```

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

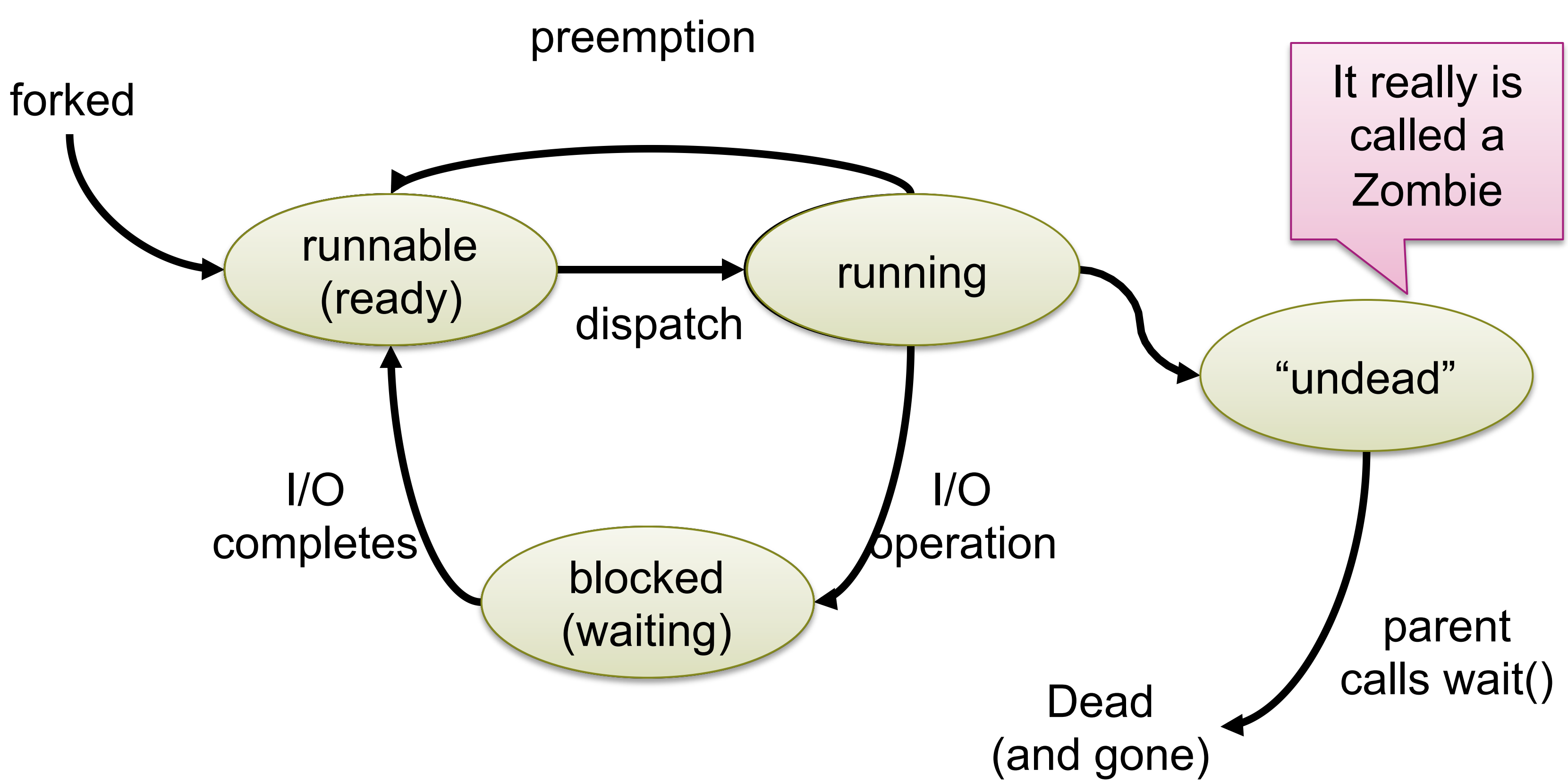
Fork in action

```
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?

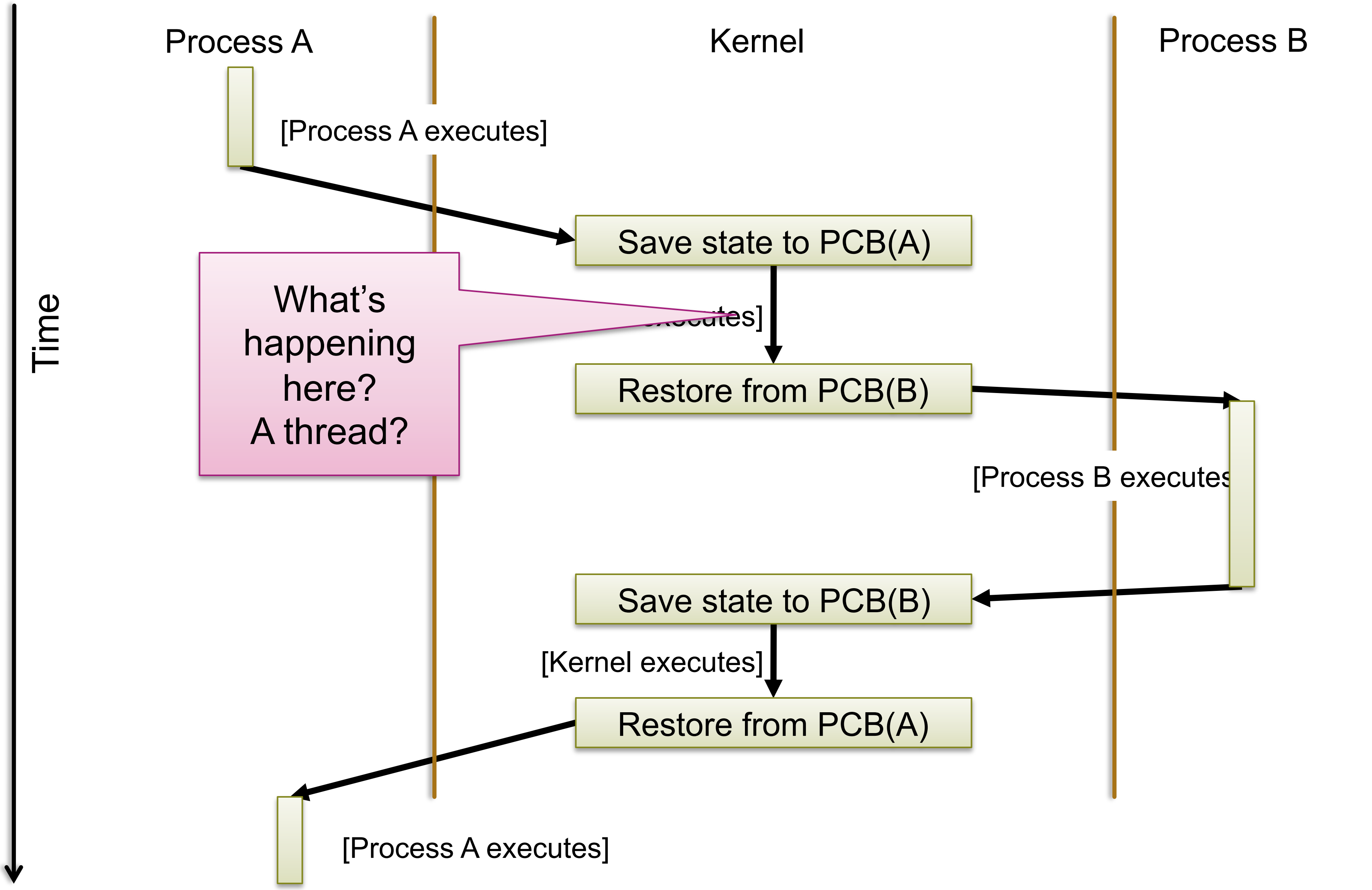
- **It depends…**

- **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



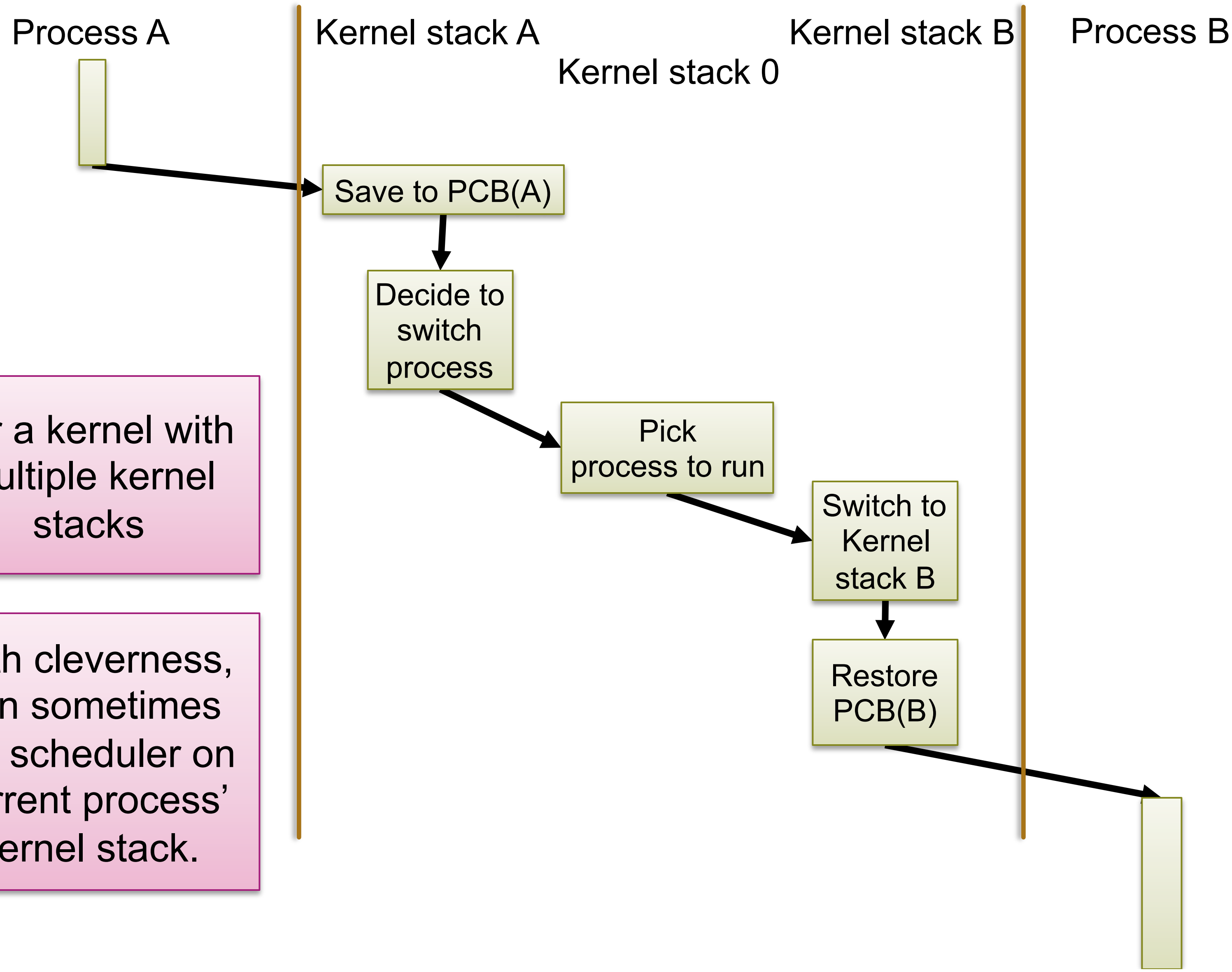
Kernel architecture

- **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



For a kernel with multiple kernel stacks

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

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

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?

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.)

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…**

User-space threads

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?**

What are the options?

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

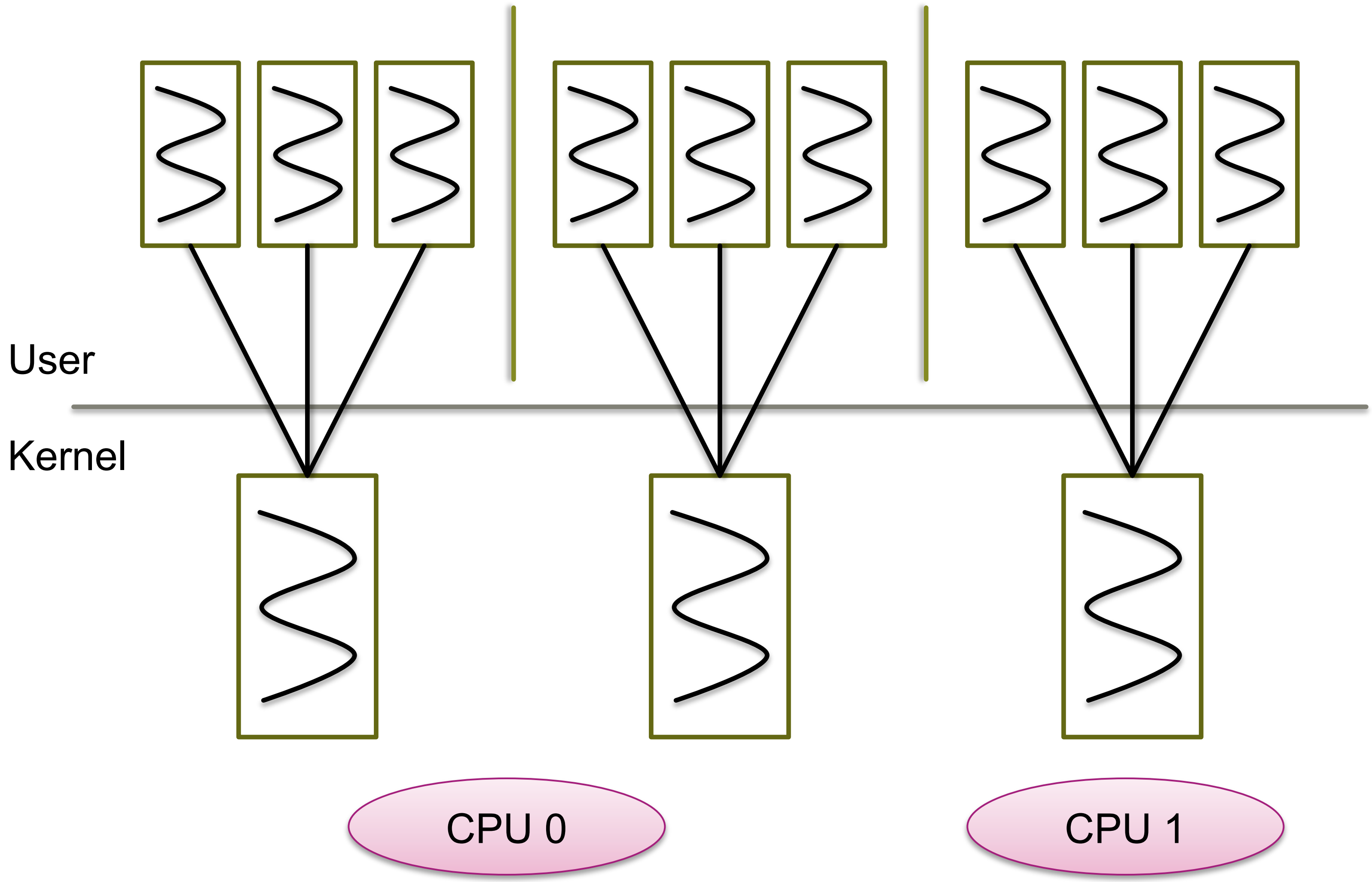
- **and other more unusual cases we won't talk about…**

Many-to-one threads

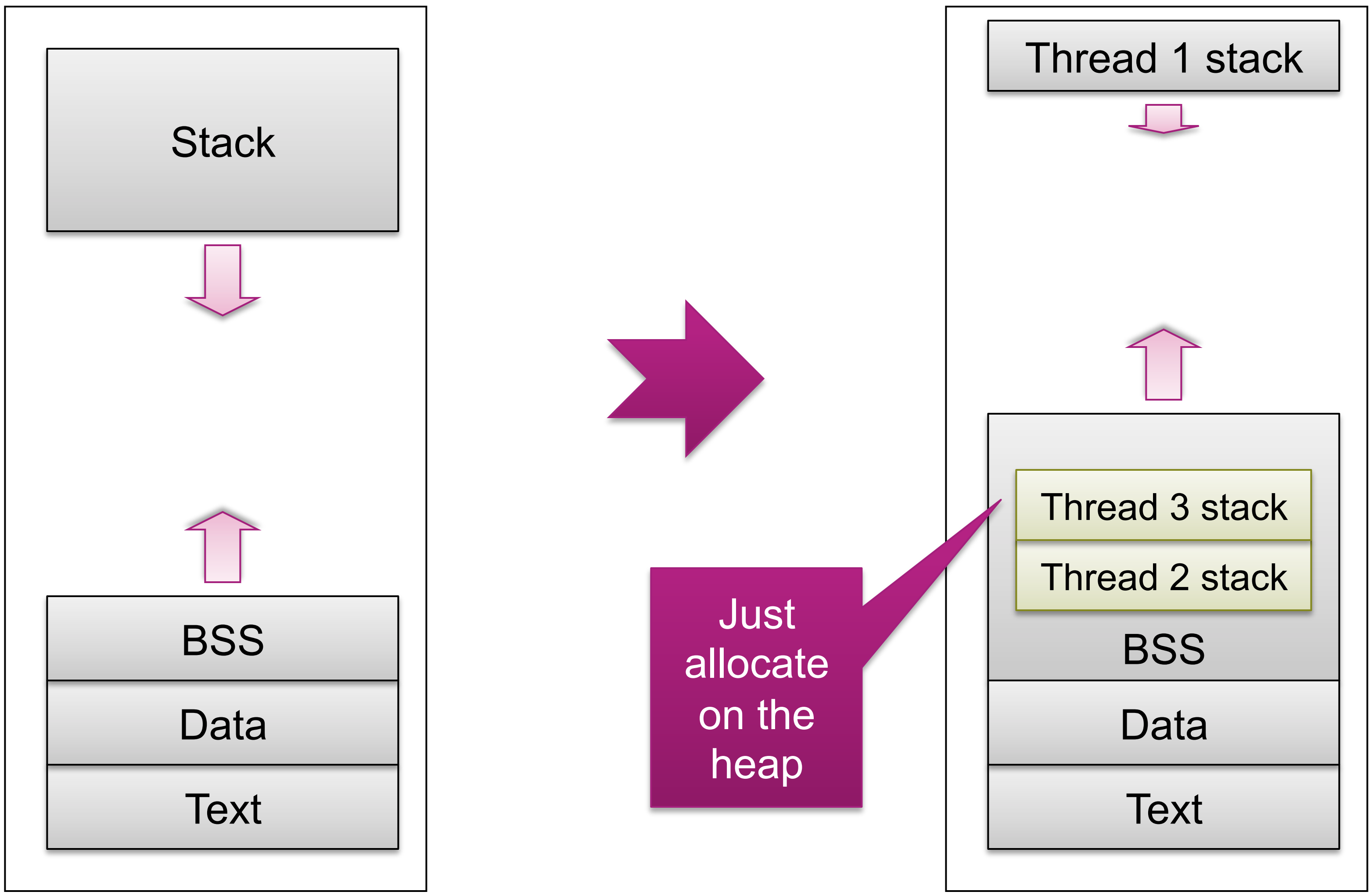
- **Early "thread libraries"**
  - Green threads (original Java VM)
  - GNU Portable Threads
  - Standard student exercise: implement them!

- **Sometimes called "pure user-level threads"**
  - No kernel support required
  - Also (confusingly) "Lightweight Processes"

Many-to-one threads



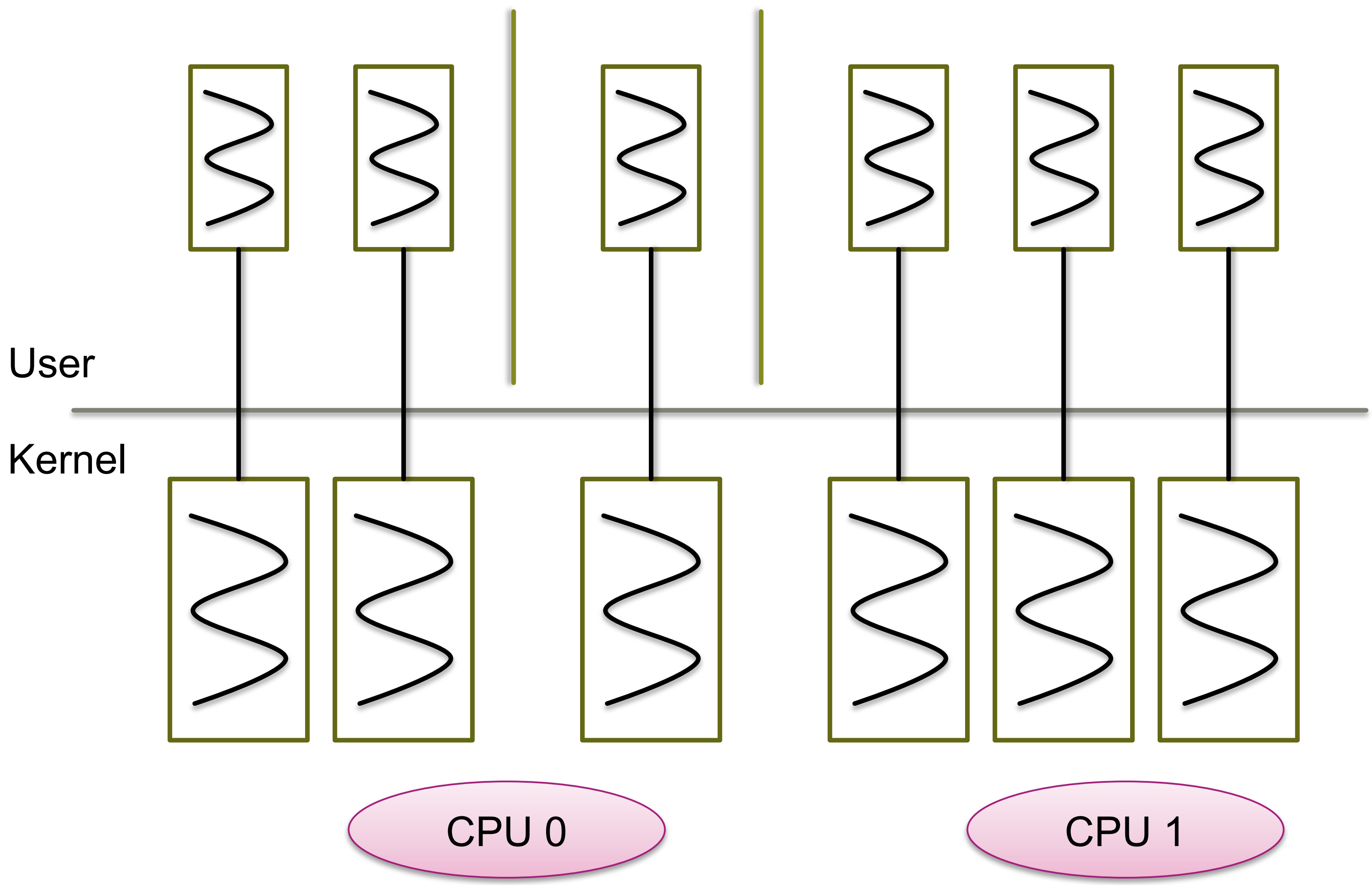
Address space layout for user level threads



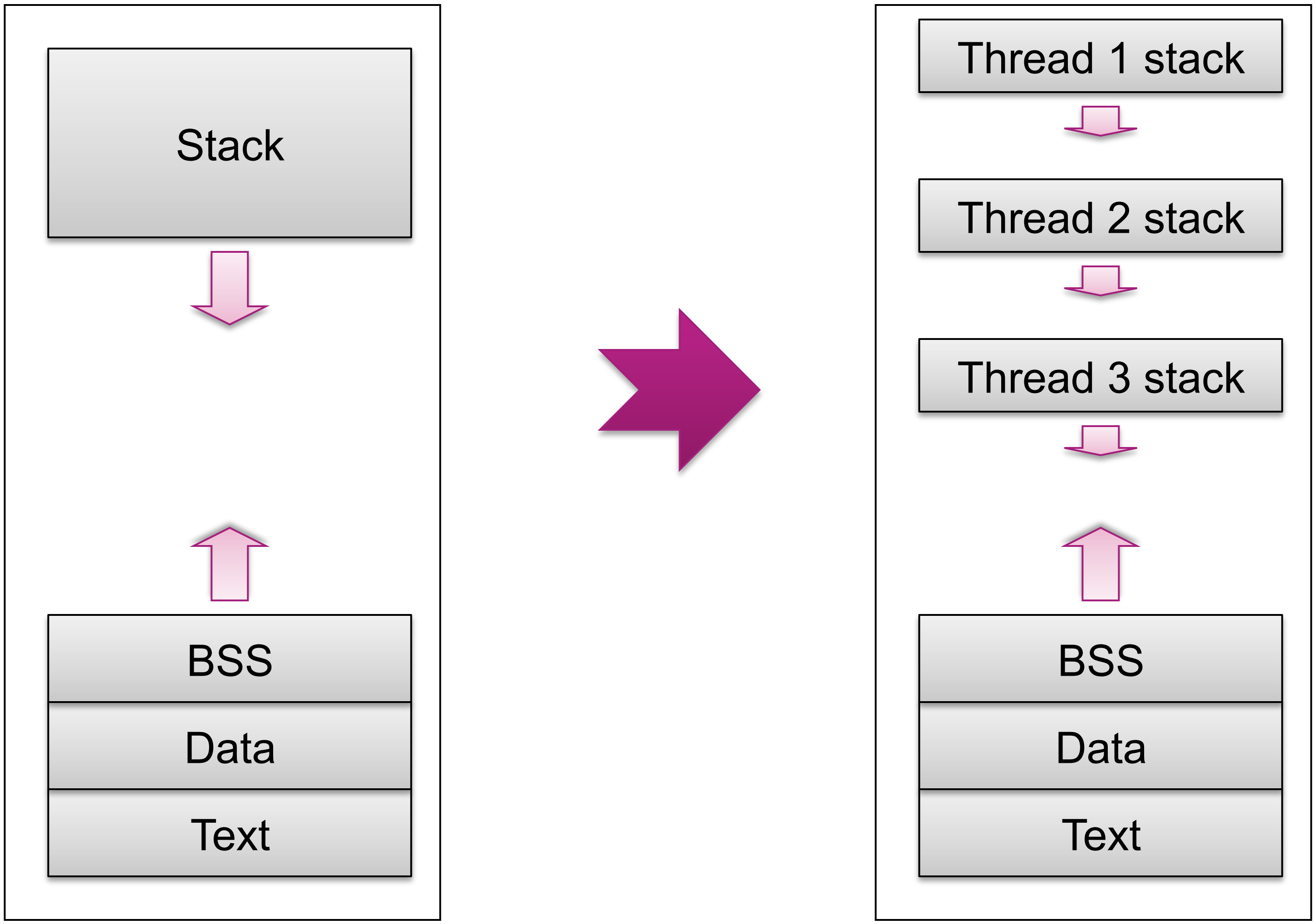
One-to-one user threads

- **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.

One-to-one user threads



One-to-one user threads



Comparison

User-level threads

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

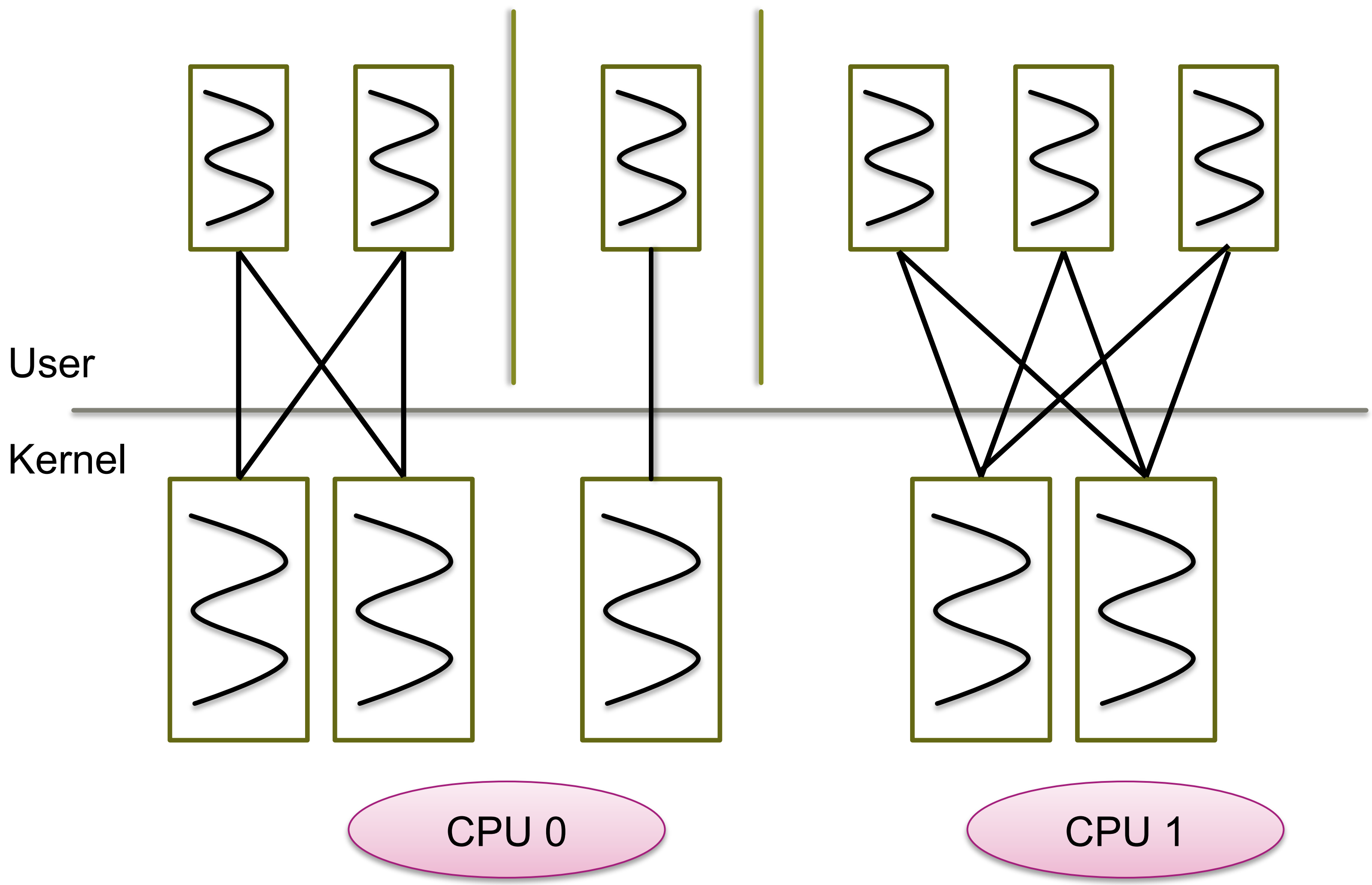
One-to-one threads

- 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?