What’s “in” a process?

- A process consists of (at least):
  - An address space, containing:
    - The code (instructions) for the running program
    - The data for the running program (static data, heap data, stack)
  - CPU state, consisting of:
    - The program counter (PC), indicating the next instruction
    - The stack pointer
    - Other general purpose register values
  - A set of OS resources
    - Open files, network connections, sound channels, ...
- In other words, it’s all the stuff you need to run the program
  - or to re-start it, if it’s interrupted at some point

The OS gets control because of …

- Trap: Program executes a syscall
- Exception: Program does something unexpected (e.g., page fault)
- Interrupt: A hardware device requests service

PCBs and CPU state

- When a process is running, its CPU state is inside the CPU
  - PC, SP, registers
  - CPU contains current values
- When the OS gets control (trap, exception, interrupt), the OS saves the CPU state of the running process in that process’s PCB
  - When the OS returns the process to the running state, it loads the hardware registers with values from that process’s PCB – general purpose registers, stack pointer, instruction pointer
- This is called a context switch

The syscall

- How do user programs do something privileged?
  - e.g., how can you write to a disk if you can’t execute an I/O instructions?
- User programs must call an OS procedure – that is, get the OS to do it for them
  - OS defines a set of system calls
  - User-mode program executes system call instruction with a parameter indicating the specific function desired
- Syscall instruction
  - Like a protected procedure call

- The syscall instruction atomically:
  - Saves the current PC
  - Sets the execution mode to privileged
  - Sets the PC to a handler address
- With that, it’s a lot like a local procedure call
  - Caller puts arguments in a place callee expects (registers or stack)
  - One of the args is a syscall number, indicating which OS function to invoke
  - Callee (OS) saves callee’s state (registers, other control state) so it can use the CPU
  - OS function code runs
    - OS must verify callee’s arguments (e.g., pointers)
    - OS returns using a special instruction
      - Automatically sets PC to return address and sets execution mode to user
A kernel crossing illustrated

Firefox: read(int fileDescriptor, void *buffer, int numBytes)

Save user PC
PC = trap handler address
Enter kernel mode

Save app state
Verify syscall number
Find sys_read() handler in vector table

sys_read() kernel routine
Verify args
Initiate read
Choose next process to run
Setup return values
Restore app state

ERET instruction
PC = saved PC
Enter user mode

Interrupts and exceptions work the same way as traps

- Transition to kernel mode
- Save state of running process in PCB
- Handler routine deals with whatever occurred
- Choose a next process to run
- Restore that process’s CPU state from its PCB
- Execute an instruction that returns you to user mode at the appropriate instruction

The OS kernel is not a process

- It’s just a block of code!
- (In a microkernel OS, many things that you normally think of as the operating system execute as user-mode processes. But the OS kernel is just a block of code.)

The design space

<table>
<thead>
<tr>
<th>Key</th>
<th>address space</th>
<th>thread</th>
</tr>
</thead>
<tbody>
<tr>
<td>MS/DO</td>
<td>one thread per process</td>
<td>one thread per process</td>
</tr>
<tr>
<td>Java</td>
<td>many threads per process</td>
<td>many threads per process</td>
</tr>
</tbody>
</table>

(old) Process address space

| 0xFFFFF | stack (dynamic allocated mem) |
| 1 | heap (dynamic allocated mem) |
| 0 | static data (data segment) |
| 0 | code (text segment) |

Threads

- Key idea:
  - separate the concept of a process (address space, OS resources)
  - … from that of a minimal “thread of control” (execution state: stack, stack pointer, program counter, registers)
- This execution state is usually called a thread, or sometimes, a lightweight process
(new) Address space with threads

Kernel threads

- OS now manages threads and processes / address spaces
  - all thread operations are implemented in the kernel
  - if one thread in a process blocks (e.g., on I/O), the OS knows about it, and can run other threads from that process
  - possible to overlap I/O and computation inside a process

- Kernel threads are cheaper than processes
  - less state to allocate and initialize
- But, they're still pretty expensive for fine-grained use
  - orders of magnitude more expensive than a procedure call
  - thread operations are all system calls
  - context switch
  - argument checks

User-level threads

- There is an alternative to kernel threads
- Threads can also be managed at the user level (that is, entirely from within the process)
  - a library linked into the program manages the threads
  - because threads share the same address space, the thread manager doesn’t need to manipulate address spaces (which only the kernel can do)
  - threads differ (roughly) only in hardware contexts (PC, SP, registers), which can be manipulated by user-level code
  - the thread package multiplexes user-level threads on top of kernel thread(s)
  - each kernel thread is treated as a “virtual processor”
  - we call these user-level threads

Getting started ...

- Fork a process (one kernel thread, one or more user-level threads)
  - creates an address space that’s a clone of the parent
  - in the kernel, there’s a new PCB that describes the child’s address space and OS resources
  - a kernel thread is created – there’s a new TCB that describes the child’s address space
  - the address space is cloned, the child has as many user-level threads as the parent did
  - in both parent and child, next instruction is the one after the fork
  - child can exec
  - child or parent can create additional threads for itself, etc.
Getting started …

- Fork a process (multiple kernel threads)
  - The child gets only one kernel thread - the one that issued the fork
  - So in the child, the next instruction to be executed is the one after the fork

Summary

- You really want multiple threads per address space
- Kernel threads are much more efficient than processes, but they’re still not cheap
  - all operations require a kernel call and parameter validation
- User-level threads are:
  - really fast/cheap
  - great for common-case operations
  - creation, synchronization, destruction
  - can suffer in uncommon cases due to kernel obliviousness
    - I/O
    - preemption of a lock-holder
- Scheduler activations are an answer
  - return control to the user-level scheduler upon blockage
- “Optimize the common case” is a key design principle