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
  - CPU state, consisting of
    - The program counter (PC), indicating the next instruction
    - The stack pointer register
    - 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
- 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
- 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 caller's state (registers, other control state) so it can use the CPU
  - OS function code runs
    - OS must verify caller'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)

user mode

kernel mode

trap handler

sys_read() kernel routine

Verify args

Initiate read

Choose next process to run

Setup return values

Restore app state

PC = saved PC

Enter user mode

User mode

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

user mode

kernel mode

trap handler

PC = saved PC

Enter user mode

sys_read() kernel routine

Verify args

Initiate read

Choose next process to run

Setup return values

Restore app state

PC = saved PC

Enter user mode

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.)

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 design space

(old) Process address space

Thread
In the beginning …

- Fork a process
  - Creates an address space that’s a clone of the parent, with one thread
  - There’s a PCB that describes the address space and the OS resources
  - There’s a TCB that holds the CPU state and is the unit of scheduling
  - The TCB and the PCB are linked – e.g., so the OS knows which set of page tables to use when scheduling a particular thread
- First thread can create additional threads
  - OS creates a new TCB, initializes CPU state (an entry point must be provided in the “create” syscall)

Kernel threads

- OS now manages threads and address spaces
  - All thread operations are implemented in the kernel
  - OS schedules all of the threads in a system
    - 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
  - Must maintain kernel state for each thread

User-level threads

- OS kernel
- CPU
- Mach, NT, Linux, …

What happens when a thread wants to:
- Create another thread?
- Terminate itself?
- Wait for some condition?
- Signal some condition?
- Do I/O?

Kernel threads

- OS now manages threads and address spaces
  - All thread operations are implemented in the kernel
  - OS schedules all of the threads in a system
    - 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
  - Must maintain kernel state for each thread

User-level threads

- OS kernel
- CPU
- Mach, NT, Linux, …

What happens when a thread wants to:
- Create another thread?
- Terminate itself?
- Wait for some condition?
- Signal some condition?
- Do I/O?