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 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)

- 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.)

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

The design space

(key)

Address space

Thread

MS/DOS

older UNIXes

Java

Mach, NT, Linux, ...

(old) Process address space

PC

stack (dynamic allocated mem)

heap (dynamic allocated mem)

static data (data segment)

code (text segment)
(new) Address space with threads

- thread 1 stack
- thread 2 stack
- thread 3 stack
- heap (dynamic allocated mem)
- static data (data segment)
- code (text segment)

© 2013 Gribble, Lazowska, Levy, Zahorjan

Kernel threads

- OS now manages threads and processes/address spaces
  - all thread operations 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

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)

User-level threads

- 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
  - can suffer in uncommon cases due to kernel obliviousness
- I/O
- preemption of a lock-holder
- Scheduler activations are an answer
  - pretty subtle though