# Process, Execution, and State

Process

- A type of interpreter

- An executing instance of a program

- Similar to a virtual private computer
  - Abstractions make it appear that the process has total access and control over the computer
  - Seems like a process has complete access to memory, isolated from other processes

- A process is an object (an entity)
  - Characterized by its properties (state, or bits)
  - Characterized by its operations
  - Of course, not all OS objects are processes

State

- Condition of being
  - Objects can have a wide range of possible states

- All persistent objects have “state”
  - Distinguishing them from other objects
  - Characterizing object’s current condition

- Contents of state depends on object
  - Complex operations often mean complex state
  - State is represented as bits, meaning it can be copied and moved
  - We can save/restore the bits of the total state
  - We can communicate state subset (scheduling state)

Examples of OS Object State

- Scheduling priority of a process
- Current pointer into a file
- Completion condition of an I/O operation
- List of memory pages allocated to a process
- OS objects’ state is mostly managed by the OS itself
  - Not directly by user code
  - It must ask the OS to access or alter state of OS objects

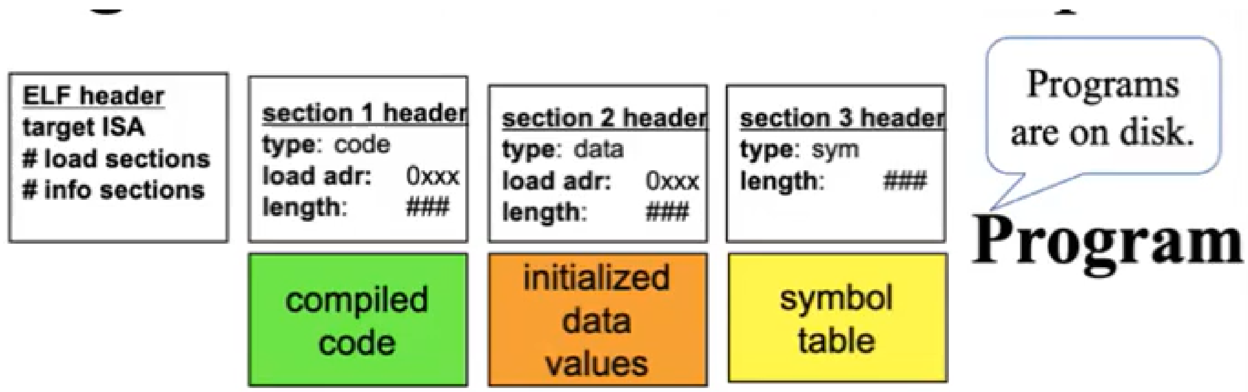
Process Address Spaces

- Each process has some memory addresses reserved for its private use
- Address space: made of memory locations that the process can address
  - If an address isn't in its address space, the process cant request access to it

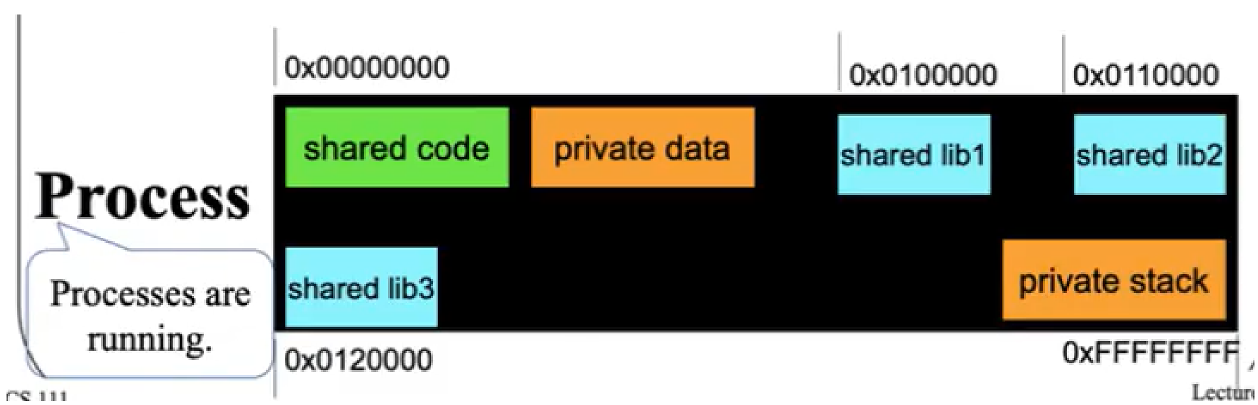
- Modern OSes pretend that every processes' address space can include all of memory
  - But this isn't true in the implementation

Programs vs Process Memory

- Program:
  - inactive set of bits which sits on persistent storage device, such as the disk
  - ELF header contains the ISA, load, and info sessions
  - Section 1 header contains the compiled code
  - Section 2 header: contains the Initialized data variables
  - Section 3 code: symbol table (mainly for informational purposes and debugging, since section 1 contains the compiled code already)

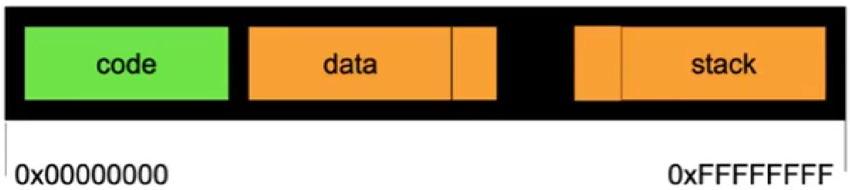



- Process
  - Programs actively running in RAM
  - Has all required memory elements for a process, must be placed somewhere in its address space
  - Different types of memory elements have different requirement
    - Code isn't writable but must be executable
    - And stacks are readable and writable but not executable
  - Each operating system has some strategy on where to put process memory segments
  - Stacks are typically not executable



Linux Processes in Memory

- Code segments are statically sized, never change once you allocate that memory
- Data segment grows up
- Stack segment grows down (recursive calls increase stack size)
- Can't overlap




Address Space: Code Segments

- Start with a load module (bits sitting statically on storage)
  - The output of a linkage editor
  - All external references have been resolved
  - All modules combined into a few segments (text, data, BSS)

- Code must be loaded into memory
  - Instructions can only be run from RAM
  - A code segment must be read in from load module
  - Map segment into process' address space

- Code segments are read/execute only and shareable
  - Many processes can reuse the same code segment

Address Space: Data Segments

- Data must be initialized in address space
  - Process data segment must be created and mapped into process' address space
  - Initial continents must be copied form load module
  - BSS' segments must be initialized to all zeros

- Data segments:
  - Read, write, and process private
  - Programs can grow or shrink it using the sbrk syscall

Processes and Stack Frames

- Modern Programming languages are stack-based

- Each procedure call allocates a new stack frame
  - Storage for procedure local vs global variables
  - Stack must be preserved as part of process state

Address Space: Stack Segment

- Size of stack depends on program activities
    - Very dynamic, hard to determine beforehand
    - Amount of local storage used by each routine
    - Grows larger as calls nest more deeply
    - After calls return, stack frames recycled

- OS manages the process' stack segment

- Stack segments are read/write and process private
    - Usually not executable

Address Space: Libraries

- Static libraries
    - Each load module has its own copy of each library
    - Program must be relinked to get new version

- Shared libraries
    - Uses less space since one in memory copy shared by all processes
    - Keep the library separate from the load modules
    - Operating system loads library along with the program

Process Descriptors

- OS data structure for dealing with a process
- Stores all information relevant to the process
    - State to restore when process is dispatched
    - References to allocated resources
    - Information to support process operations

- Managed by the OS and used for scheduling, security decisions

Linux Process Control Block

- Data structure Linux uses to handle processes
- An example of process descriptor
- Keeps track of
    - PID
    - State of process
    - Address space info
    - Other info

Other Process State

- OS itself isn't a process, but it's stack oriented meaning it uses the stack to keep track of info

- Not all process state stored in the descriptor

- Registers
    - Application execution state on the stack and registers
    - General registers
    - Program counter, stack pointer, frame pointer, processor status

- Process' own OS resources
    - Open files (pointers to open files), current working directory, locked resources

- OS state information
    - Ex. time spent executing a process, priority of the process

- Other info is stored in other memory areas

Process Creation

- Created by the OS using some method to initialize the state (preparing a program to run)
- Other processes can start a process
- Parent process: process that created your process
- Child process: the process your process created

Creating Process Descriptors

- The process descriptor is the OS' basic per process data structure
- New process needs a new descriptor

- Process table
    - Data structure the OS uses to organize all currently active processes
    - Process tables contains one entry for each process in the system

- Address space to hold all segments
    - OS creates a data structure called an address space and allocates memory for code, data, and stack

Choices for Process Creation

1. Start with a blank process
    - No initial state or resources

2. Calling process is used as a template
    - Child process is identical to the parent process

Windows process creation

- CreateProcess() syscall
- Flexible way to create a new process with many parameters with many possible values
- Different parameters dill out other critical information

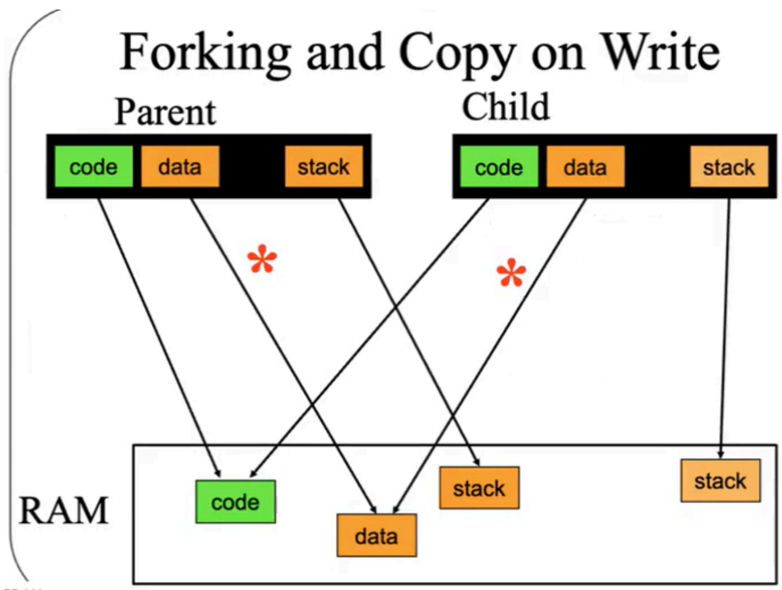
Process Forking

- Unix/Linux process creation
- Essentially clones an existing process
- On assumption that the child looks like the new one

After the fork
- 2 processes with different PIDs, but otherwise the same
- One process knows that is the parent, the other knows it's the child
- Parent and child go in separate ways

Forking Memory

- Data of the child needs to to separate from the child
- Copying the data from the parent for the child could be expensive
- Code can be shared, and stack must be separate




- Copy on write: Child data is set to point to the parent’s data
- Only when a write is made to the data will a copy be made

Exec()

- Forking is expensive, sometimes I want a process that does something entirely different from the parent
- Exec() is a unix syscall to “remake” a process
- Changes the code associated with a process and resets most of the state as well

Exec call
- After fork, exec calls allow the parent process to continue while making a blank child process

- Replace a process' exiting program with new code, different resources, different stack
- Doesn't need to be called after a fork, can be used to change any process

Handling exec

- Removes the child's old code and loads new set of code for that process
- Intalizes stack, data of parent processed referenced with copy on write
- Initializes child's stack, PC, and other relevant control structure

Destroying Processes

- When processes terminate, they finish their work and terminate
- When a process terminates, OS needs to clean up and free all resources

1. Reclaim resource (memory, locals, access to hardware)
2. Inform other processes (parent or child processes or processes waiting for interprocess comms)
3. Remove process descriptor form process table (reclaim its memory)

Running Processes

- Must execute code to complete a task, meaning OS gives them access to a processor core
- Usually more processes than cores, so processes must share
- All executions can't be run at once
- A process that isn't running on a core needs to be placed onto a core

Loading a Process

- To run a process on a core, core's hardware must be initialized
    - Either to an initial state or the state of the previous run
- Must load core's registers and initialize stack and set the stack pointer
- Set up memory control structures
- Set program counter

Running a Process

- Limited direct execution: Most instructions executed directly by they process on the core without OS intervention
- Trap: Privileged instructions instead cause the OS to handle to instruction

Limited Direct Execution

- CPU executes most application code
    - Occasional traps and timer interrupts

- Maximizing direct execution is always the goal
    - For Linux and Windows user mode processes
    - For OS emulation (Windows on Linux)

    - For virtual machines

- Use the OS as seldom as possible
    - Goes directly to the hardware instead of through the OS (less overhead)
    - Get back to the application quickly (fasted execution)
    - Good system performance

- Usual flow: Runs in limited direct execution, execute a trap to run a privileged instruction, then return to limited direct execution

Exceptions

- Technical term for what happens when the process can’t (or shouldn’t) run an instruction

- Routine Exceptions
    - Error: arithmetic overflow, conversion error
    - Caused by the process
    - May not be an error, such as end of file

- Unpredictable Exceptions
    - Segmentation fault (accessed invalid memory)
    - User abort, power failure
    - Async exceptions

Asynchronous Exceptions

- Inherently unpredictable since the response could be an error

- Program can't check for them since its unknown what happens

- Some languages support try/catch operations

- Hardware and OS can support traps
    - Trap: intentional exception which requests the OS to perform a restricted operation
    - Catches exceptions and transfers control to the OS

- OS also use system calls
    - Requests from a program for OS services

Using Traps for Syscalls

- Reserved privileged instructions for syscalls

1. Define syscall linkage conventions

- Information about the system call is stored in registers
- Call: r0 = system call number, r1 points to arguments

2. Prepare arguments for the syscall

3. Execute the designated system call instruction (traps to OS)

4. OS recognizes and performs the requested operation
    - Entering the OS through a point called a gate

5. Returns to instruction after the sys call
    - If there is a return, stores it into register r0

Trap and System Call Steps

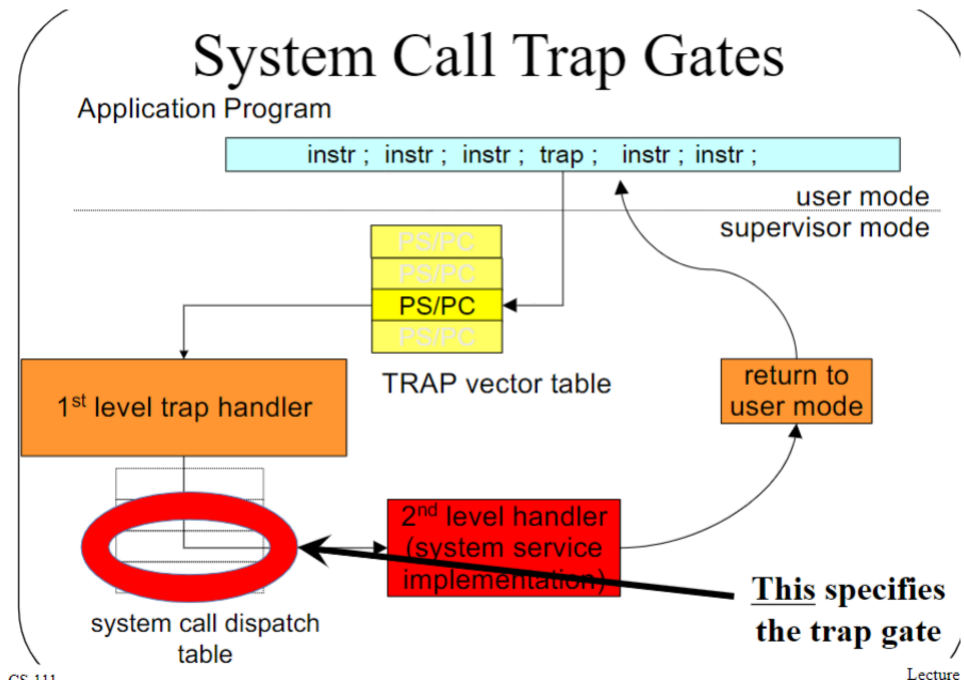
1. After encountering a trap, we enter a trap vector table in the OS
    - Determines the type of syscall that is requested

2. 1st Level Trap Handler
    - Sets up the resources for the syscall (setting up stack and memory for parameters, accesses memory)

3. System call dispatch table
    - Maps system call numbers to the kernel functions
    - Contains the addresses for the code of the syscall

4. 2nd Level Trap Handler
    - Detmeins how to handle the trap and what syscall should be used

5. Return to user mode




Trap Handling

- Partially hardware, partially software
  - Must be instruction in ISA for trap handling
  - PC: process counter (points to the instruction to execute)
  - PS: processor status (sets up the modes, user and privileged)

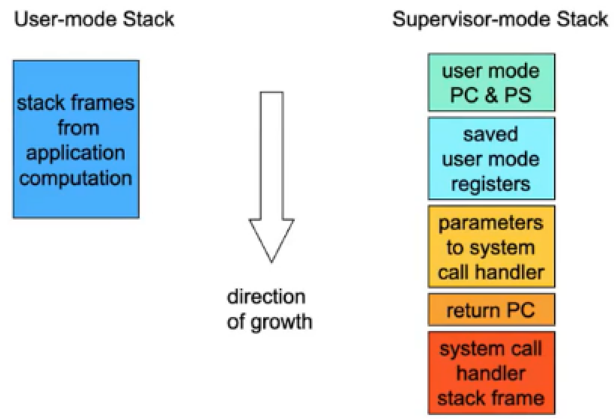
- Hardware portion of trap handling
  1. Trap cause an index index into trap vector table for PC/PS
  2. Load new processor status word, switch to supervisor mode
  3. Push PC/PS of program that caused trap onto stack
  4. Load PC (with address of 1st level handler)

- Software portion of trap handling
  - 1st level handler pushes all other registers
  - 1st level handler gather info on the syscall, then selects the second level handler
  - 2nd level handler deals with the syscall
  - Lots of OS code is run to do this

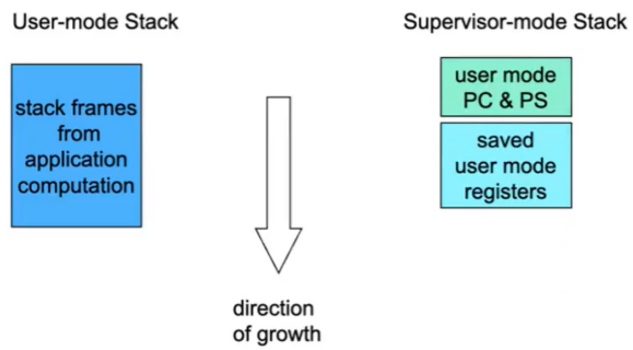
Traps and the Stack

- Code which handles the trap is code that is run in privileged mode
- Requires the stack to run since it might call multiple routines
- A second stack is used to keep track of the syscall
  - Maintained by the OS

Stacking and Unstacking a syscall




- During the syscall, the OS stack is used to carry out the syscall functions




- As the syscall returns, the OS stack is popped until nothing remains
- After that, it returns to the user mode which is the last one one the stack

Handling Asynchronous Events

- Events worth waiting for
    - read(), want to wait for the data

- Other time waiting doesn;t make sense
    - CPU should work while waiting
    - SOme vents demand prompt actions

- Event Completion Call-backs
    - Stores information about an interrupt or an event has occurred
    - Ex. Waiting for data from input, so we could do other processes while waiting. When the input is received, then we can do work on the input data
    - Computers support interrupts, which is common with I/O devices and timers

User Model Signal Handling

- OS can indicate to a process that signals have happened
- Exceptions, operator actions, communication
- Processes can control their handling
    - Ignore the signal
    - Designate a handler for this signal
- SImilar to hardware traps but implemented by the OS and delivered to user mode processes

Managing Process State

- Process can handle its own stack and the contents of its memory
    - Not an OS problem

- The OS keeps track of resources that have been allocated to the processes
    - Which memory segments
    - Open file and devices

Blocked Processes

- A process is blocked when it isn't ready to run if its not suitable to run at the moment
    - Opening a file: we can't do anything with the contents of the file without calling open()
    - I/O devices
    - Blocked process cannot run
- OS keeps track if the process is blocked
- User applications cant unblock themselves since all info on blocked process is stored in the OS

Blocking and Unblocking

- Scheduler determines whether to block or unblock
- Any part of the OS can set blocks and any part of the OS can remove them
    - Process can ask to be blocked using a system call

- Process blocked when it needs unavailable resources
- Process unblocked when these resources become available
    - Means the process is ready to run
    - Changes the scheduling of a process to “ready”, allowing the scheduler to schedule accordingly