Operating System Principles: Processes, Execution, and State

CS 111

Operating Systems

Peter Reiher

Outline

- What are processes?
- How does an operating system handle processes?
- How do we manage the state of processes?

What Is a Process?

- A type of interpreter
- An executing instance of a program
- A virtual private computer
- A process is an *object*
  - Characterized by its properties (*state*)
  - Characterized by its *operations*
  - Of course, not all OS objects are processes
  - But processes are a central and vital OS object type

What is “State”?

- One dictionary definition of “state” is
  - “A mode or condition of being”
  - An object may 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
  - We can save/restore the aggregate/total state
  - We can talk of a subset (e.g., 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

What Are Operations?

- Activities performed by an object
- Often resulting in changes to its state
- Sometimes (especially in OSes) resulting in changes to hardware devices
- Sometimes resulting in changes to other objects' states
  - Usually indirectly, since typically one object cannot directly change another's state

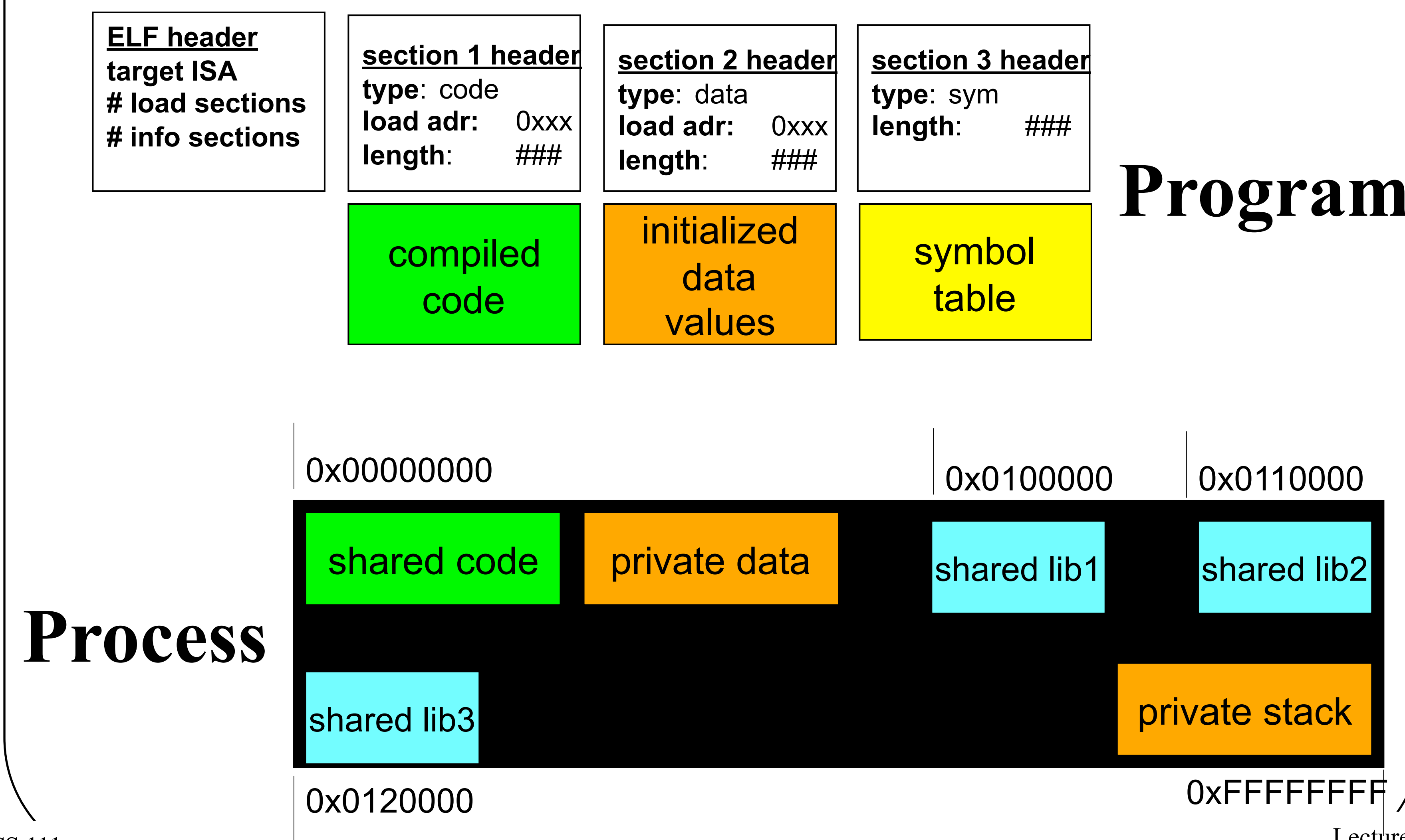
Examples of OS Object Operations

- Create a process
- Deallocate a page of memory
- Open a file
- Write to a display
- Not directly performable by user processes
  - They must ask the OS to perform the operation
- Almost always resulting in OS object state changes

Process Address Spaces

- Each process has some memory addresses reserved for its private use
- That set of addresses is called its *address space*
- A process' address space is made up of all memory locations that the process can address
- Modern OSes provide the illusion that the process has all of memory in its address space
  - But that's not true, under the covers

Program vs. Process Address Space



Process Address Space Layout

- All required memory elements for a process must be put somewhere in its address space
- Different types of memory elements have different requirements
  - E.g., code is not writable but must be executable
  - And stacks are readable and writable but not executable
- Each operating system has some strategy for where to put these process memory segments

Layout of Unix Processes in Memory

- In Unix systems[1],
  - Code segments are statically sized
  - Data segment grows up
  - Stack segment grows down
- They aren't allowed to meet

[1] Linux is one type of Unix system

Address Space: Code Segments

- We start with a load module
  - The output of a linkage editor
  - All external references have been resolved
  - All modules combined into a few segments
  - Includes multiple segments (text, data, BSS)
- Code must be loaded into memory
  - A virtual code segment must be created
  - Code must be read in from the load module
  - Map segment into virtual address space
- Code segments are read/execute only and sharable
  - Many processes can use the same code segments

Address Space: Data Segments

- Data too must be initialized in address space
  - Process data segment must be created
  - Initial contents must be copied from load module
  - BSS[1]: segments to be initialized to all zeroes
  - Map segment into virtual address space
- Data segments
  - Are read/write, and process private
  - Program can grow or shrink it (using the `sbrk` system call)

[1]Block Started by Symbol – a legacy term of no importance

Processes and Stack Frames

- Modern programming languages are stack-based
  - Greatly simplified procedure storage management
- Each procedure call allocates a new stack frame
  - Storage for procedure local (vs. global) variables
  - Storage for invocation parameters
  - Save and restore registers
    - Popped off stack when call returns
- Most modern CPUs also have stack support
  - Stack too must be preserved as part of process state

Address Space: Stack Segment

- Size of stack depends on program activities
  - E.g., by amount of local storage used by each routine
  - Grows larger as calls nest more deeply
  - After calls return, their stack frames can be recycled
- OS manages the process' stack segment
  - Stack segment created at same time as data segment
  - Some OSes allocate fixed sized stack at program load time
  - Some dynamically extend stack as program needs it
- Stack segments are read/write and process private
  - Usually not executable

Address Space: Libraries

- Static libraries are added to load module
  - Each load module has its own copy of each library
  - Program must be re-linked to get new version
- Shared libraries use less space
  - One in-memory copy, shared by all processes
  - Keep the library separate from the load modules
  - Operating system loads library along with program
- Reduced memory use, faster program loads
- Easier and better library upgrades

Other Process State

- Registers
  - General registers
  - Program counter, processor status
  - Stack pointer, frame pointer
- Process' own OS resources
  - Open files, current working directory, locks
- But also OS-related state information

OS State For a Process

- The state of process' virtual computer
- Registers
  - Program counter, processor status word
  - Stack pointer, general registers
- Address space
  - Text, data, and stack segments
  - Sizes, locations, and contents
- The OS needs some data structure to keep track of a process' state

Process Descriptors

- Basic OS data structure for dealing with processes
- 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
- Used for scheduling, security decisions, allocation issues

Linux Process Control Block

- The data structure Linux (and other Unix systems) use to handle processes
  - AKA *PCB*
- An example of a process descriptor
- Keeps track of:
  - Unique process ID
  - State of the process (e.g., running)
  - Parent process ID
  - Address space information
  - And various other things

Other Process State

- Not all process state is stored directly in the process descriptor
- Other process state is in multiple other places
  - Application execution state is on the stack and in registers
  - Linux processes also have a supervisor-mode stack
    - To retain the state of in-progress system calls
    - To save the state of an interrupt preempted process
- A lot of process state is stored in the other memory areas

Handling Processes

- Creating processes
- Destroying processes
- Running processes

Where Do Processes Come From?

- Created by the operating system
  - Using some method to initialize their state
  - In particular, to set up a particular program to run
- At the request of other processes
  - Which specify the program to run
  - And other aspects of their initial state
- Parent processes
  - The process that created your process
- Child processes
  - The processes your process created

Creating a Process Descriptor

- The process descriptor is the OS' basic per-process data structure
- So a new process needs a new descriptor
- What does the OS do with the descriptor?
- Typically puts it into a *process table*
  - The data structure the OS uses to organize all currently active processes
  - Process table contains one entry for each process in the system

What Else Does a New Process Need?

- An address space
- To hold all of the segments it will need
- So the OS needs to create one
  - And allocate memory for code, data and stack
- OS then loads program code and data into new segments
- Initializes a stack segment
- Sets up initial registers (PC, PS, SP)

Choices for Process Creation

1. Start with a "blank" process
   – No initial state or resources
   – Have some way of filling in the vital stuff
     • Code
     • Program counter, etc.
   – This is the basic Windows approach
2. Use the calling process as a template
   – Give new process the same stuff as the old one
   – Including code, PC, etc.
   – This is the basic Unix/Linux approach

Starting With a Blank Process

- Basically, create a brand new process
- The system call that creates it obviously needs to provide some information
  - Everything needed to set up the process properly
  - At the minimum, what code is to be run
  - Generally a lot more than that
- Other than bootstrapping, the new process is created by command of an existing process

Windows Process Creation

- The `CreateProcess()` system call
- A very flexible way to create a new process
  - Many parameters with many possible values
- Generally, the system call includes the name of the program to run
  - In one of a couple of parameter locations
- Different parameters fill out other critical information for the new process
  - Environment information, priorities, etc.

Process Forking

- The way Unix/Linux creates processes
- Essentially clones the existing parent process
- On assumption that the new child process is a lot like the old one
  - Most likely to be true for some kinds of parallel programming
  - Not so likely for more typical user computing

Why Did Unix Use Forking?

- Avoids costs of copying a lot of code
  - *If* it's the same code as the parent's . . .
- Historical reasons
  - Parallel processing literature used a cloning fork
  - Fork allowed parallelism before threads invented
- Practical reasons
  - Easy to manage shared resources
    - Like stdin, stdout, stderr
  - Easy to set up process pipe-lines (e.g. `ls | more`)
  - Eases design of command shells

What Happens After a Fork?

- There are now two processes
  - With different IDs
  - But otherwise mostly exactly the same
- How do I profitably use that?
- Program executes a fork
- Now there are two programs
  - With the same code and program counter
- Write code to figure out which is which
  - Usually, parent goes “one way” and child goes “the other”

Forking and the Data Segments

- Forked child shares the parent's code
- But not its stack
  - It has its own stack, initialized to match the parent's
  - Just as if a second process running the same program had reached the same point in its run
- Child should have its own data segment, though
  - Forked processes do not share their data segments

Forking and Copy on Write

- If the parent had a big data area, setting up a separate copy for the chi
  - And fork was suppose
- If neither parent nor c data area, though, no c
- So set it up as copy-on-write
- If one of them writes it, then make a copy and let the process write the copy
  - The other process keeps the original

Potential serious problems if OS doesn't get this right. E.g., Linux DirtyCOW bug.

But Fork Isn't What I Usually Want!

- Indeed, you usually don't want another copy of the same process
- You want a process to do something entirely different
- Handled with `exec()`
  - A Unix system call to "remake" a process
  - Changes the code associated with a process
  - Resets much of the rest of its state, too
    - Like open files

The `exec` Call

- A Linux/Unix system call to handle the common case
- Replaces a process' existing program with a different one
  - New code
  - Different set of other resources
  - Different PC and stack
- Essentially, called after you do a fork

How Does the OS Handle Exec?

- Must get rid of the child's old code
  - And its stack and data areas
  - Latter is easy if you are using copy-on-write
- Must load a brand new set of code for that process
- Must initialize child's stack, PC, and other relevant control structure
  - To start a fresh program run for the child process

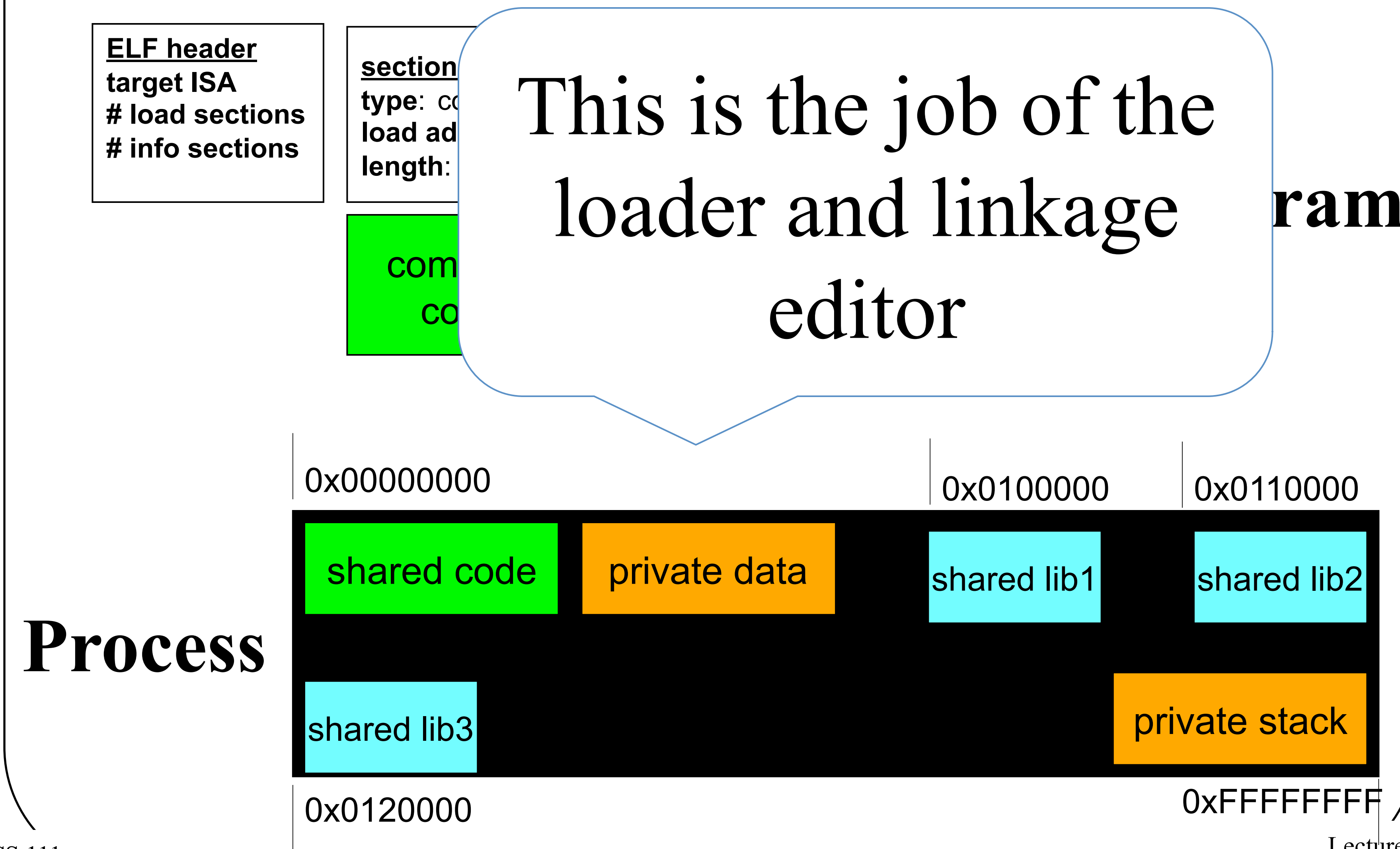
Loading Programs Into Processes

- Whether you did a Windows `CreateProcess()` or a Unix `exec()`
  - You need to go from loadable program to runnable process
- To get from the code to the running version, you need to perform the *loading* step
  - Initializing the various memory domains we discussed earlier
    - Code, stack, data segment, etc.

Loading Programs

- You have a load module
  - The output of linkage editor
  - All external references have been resolved
  - All modules combined into a few segments
  - Includes multiple segments (code, data, etc.)
- A computer cannot “execute” a load module
  - Computers execute instructions in memory
  - Memory must be allocated for each segment
  - Code must be copied from load module to memory

Program to Process Transition



Destroying Processes

- Most processes terminate
  - All do, of course, when the machine goes down
  - But most do some work and then exit before that
  - Others are killed by the OS or another process
- When a process terminates, the OS needs to clean it up
  - Essentially, getting rid of all of its resources
  - In a way that allows simple reclamation

What Must the OS Do to Terminate a Process?

- Reclaim any resources it may be holding
  - Memory
  - Locks
  - Access to hardware devices
- Inform any other process that needs to know
  - Those waiting for interprocess communications
  - Parent (and maybe child) processes
- Remove process descriptor from the process table

Running Processes

- Processes must execute code to do their job
- Which means the OS must give them access to a processor core
- But usually more processes than cores
  - Easily 200-300 on a typical modern machine
- So processes will need to share the cores
  - And they can't all execute instructions at once
- Sooner or later, a process not running on a core needs to be put onto one

Loading a Process

- To run a process on a core, the core's hardware must be initialized
  - Either to initial state or whatever state the process was in the last time it ran
- Must load the core's registers
- Must initialize the stack and set the stack pointer
- Must set up any memory control structures
- Must set the program counter
- Then what?

How a Process Runs on an OS

- It uses an execution model called *limited direct execution*
- Most instructions are executed directly by the process on the core
  - Without any OS intervention
- Some instructions instead cause a trap to the operating system
  - Privileged instructions that can only execute in supervisor mode
  - The OS takes care of things from there

Limited Direct Execution

- CPU directly executes most application code
  - Punctuated by occasi al traps (for system calls)
  - With occasional tir rrupts (for time sharing)
- Maximizing di s always the goal
  - For Linux use
  - For OS emu on Linux)
  - For virtual machines
- Enter the OS as seldom as possible
  - Get back to the application as quickly as possible

The key to good system performance !

Exceptions

- The technical term for what happens when the process can't (or shouldn't) run an instruction
- Some exceptions are routine
  - End-of-file, arithmetic overflow, conversion error
  - We should check for these after each operation
- Some exceptions occur unpredictably
  - Segmentation fault (e.g., dereferencing NULL)
  - User abort (^C), hang-up, power-failure
  - These are asynchronous exceptions

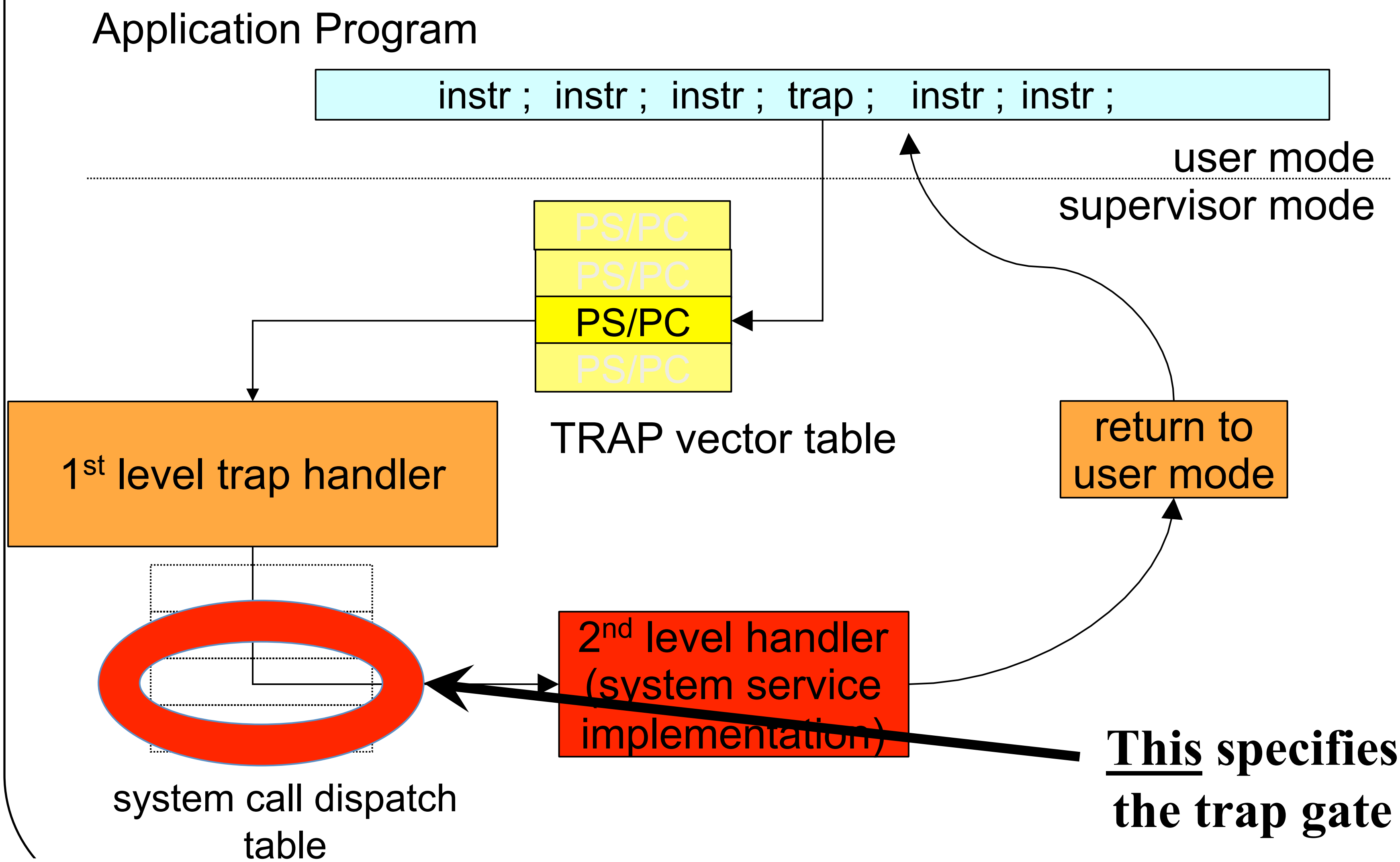
Asynchronous Exceptions

- Inherently unpredictable
- Programs can't check for them, since no way of knowing when and if they happen
- Some languages support try/catch operations
- Hardware and OS support traps
  - Which catch these exceptions and transfer control to the OS
- Operating systems also use these for *system calls*
  - Requests from a program for OS services

Using Traps for System Calls

- Made possible at processor design time, not OS design time
- Reserve one privileged instruction for system calls
  - Most computers specifically define such instructions
- Define system call linkage conventions
  - Call: r0 = system call number, r1 points to arguments
  - Return: r0 = return code, condition code indicates success/failure
- Prepare arguments for the desired system call
- Execute the designated system call instruction
- Which causes an exception that traps to the OS
- OS recognizes & performs requested operation
  - Entering the OS through a point called a *gate*
- Returns to instruction after the system call

System Call Trap Gates



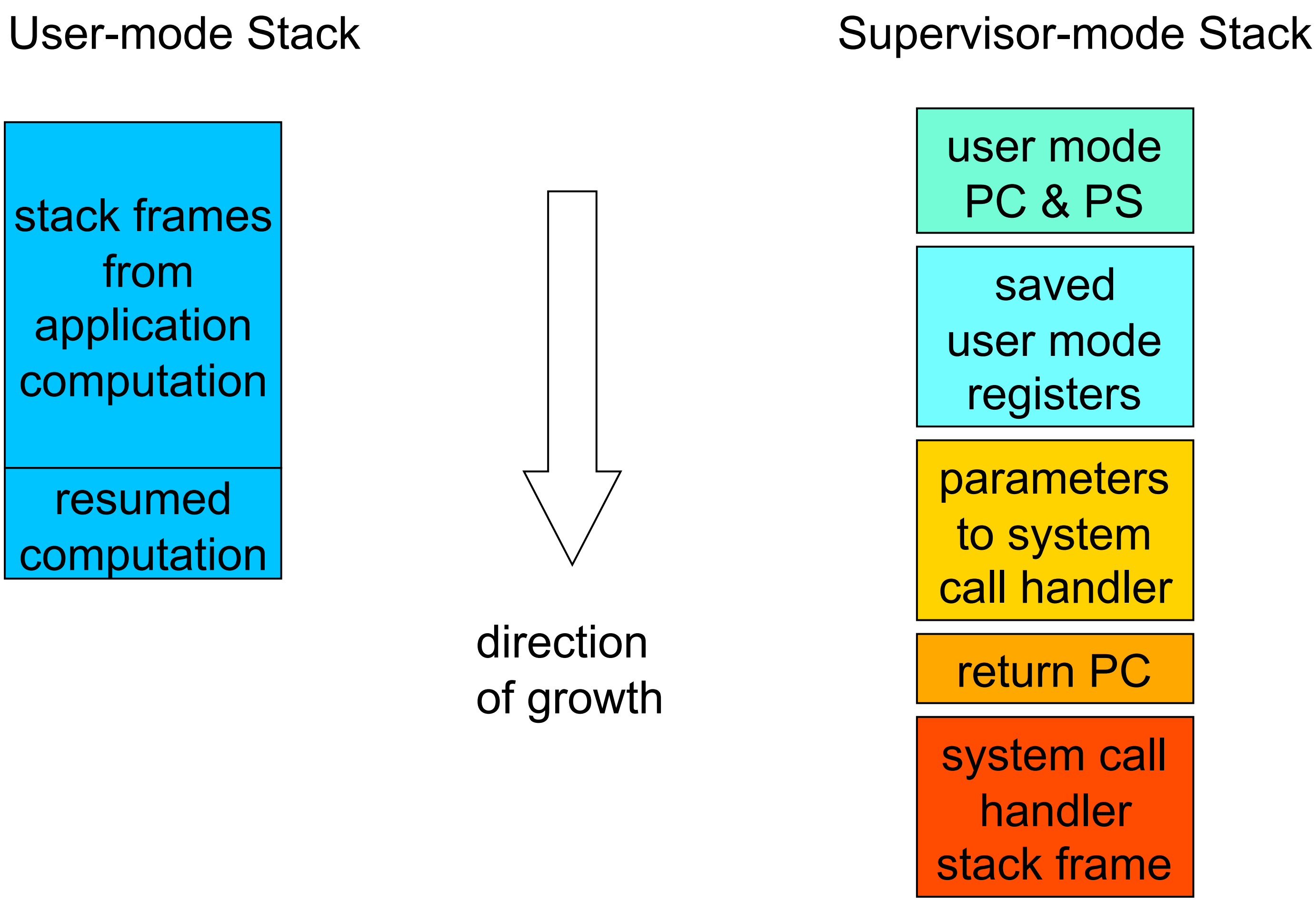
Trap Handling

- Partially hardware, partially software
- Hardware portion of trap handling
  - Trap cause an index into trap vector table for PC/PS
  - Load new processor status word, switch to supervisor mode
  - Push PC/PS of program that caused trap onto stack
  - Load PC (with address of 1st level handler)
- Software portion of trap handling
  - 1st level handler pushes all other registers
  - 1st level handler gathers info, selects 2nd level handler
  - 2nd level handler actually deals with the problem
    - Handle the event, kill the process, return ...

Traps and the Stack

- The code to handle a trap is just code
  - Although run in privileged mode
- It requires a stack to run
  - Since it might call many routines
- How does the OS provide it with the necessary stack?
- While not losing track of what the user process was doing?

Stacking and Unstacking a System Call



Returning to User-Mode

- Return is opposite of interrupt/trap entry
  - 2nd level handler returns to 1st level handler
  - 1st level handler restores all registers from stack
  - Use privileged return instruction to restore PC/PS
  - Resume user-mode execution at next instruction
- Saved registers can be changed before return
  - Change stacked user r0 to reflect return code
  - Change stacked user PS to reflect success/failure

Asynchronous Events

- Some things are worth waiting for
  - When I `read()`, I want to wait for the data
- Other time waiting doesn't make sense
  - I want to do something else while waiting
  - I have multiple operations outstanding
  - Some events demand very prompt attention
- We need *event completion call-backs*
  - This is a common programming paradigm
  - Computers support interrupts (similar to traps)
  - Commonly associated with I/O devices and timers

User-Mode Signal Handling

- OS defines numerous types of signals
  - Exceptions, operator actions, communication
- Processes can control their handling
  - Ignore this signal (pretend it never happened)
  - Designate a handler for this signal
  - Default action (typically kill or coredump process)
- Analogous to hardware traps/interrupts
  - But implemented by the operating system
  - Delivered to user mode processes

Managing Process State

- A shared responsibility
- The process itself takes care of its own stack
- And the contents of its memory
- The OS keeps track of resources that have been allocated to the process
  - Which memory
  - Open files and devices
  - Supervisor stack
  - And many other things

Blocked Processes

- One important process state element is whether a process is ready to run
  - No point in trying to run it if it isn't ready to run
- Why might it not be?
- Perhaps it's waiting for I/O
- Or for some resource request to be satisfied
- The OS keeps track of whether a process is blocked

Blocking and Unblocking Processes

- Why do we block processes?
  - Blocked/unblocked are merely notes to scheduler
  - So the scheduler knows not to choose them
  - And so other parts of OS know if they later need to unblock
- Any part of OS can set blocks, remove them
  - And a process can ask to be blocked itself
    - Through a system call

Better be sure someone will unblock you . . .

Who Handles Blocking?

- Usually happens in a resource manager
  - When process needs an unavailable resource
    - Change process's scheduling state to “blocked”
    - Call the scheduler and yield the CPU
  - When the required resource becomes available
    - Change process's scheduling state to “ready”
    - Notify scheduler that a change has occurred

Swapping Processes

- Processes can only run when in main memory
  - CPU can only execute instructions stored in that memory
- Sometimes we move processes out of main memory to secondary storage
  - E.g., a disk drive
  - Expecting that we'll move them back later
- Usually because of resource shortages
  - Particularly memory

Why We Swap

- To make best use of a limited amount of memory
  - A process can only execute if it is in memory
  - Max number of processes is limited by memory size
  - If it isn't READY, it doesn't need to be in memory
  - Swap it out and make room for some other process
- We don’t swap out all blocked processes
  - Swapping is expensive
  - And also expensive to bring them back
  - Typically only done when resources are tight

Basic Mechanics of Swapping

- Process' state is stored in parts of main memory
- Copy them out to secondary storage
  - If you're lucky and careful, some don't need to be copied
- Alter the process descriptor to indicate what you did
- Give the freed resources to another process

Swapping Back

- When whatever blocked the process you swapped is cleared, you can swap back
  - Assuming there's space
- Reallocate required memory and copy state back from secondary storage
  - Both stack and heap
- Unblock the process' descriptor to make it eligible for scheduling
- Ready swapped processes need not be brought back immediately
  - But they won't get any cycles till you do