Processes and Threads
CS 111
Operating System Principles

Peter Reiher

Outline

- The concepts of processes and threads
- Going from conceptual to real systems
- How does the OS handle processes and threads?
- Creating and destroying processes

Processes and Threads

- Threads are a simple concept
- They are used in real operating systems
- But they aren't the actual key interpreter abstraction of real operating systems
- Systems like Linux and Windows use another abstraction
  - The *process*

What Is a Process?

- Essentially, a virtual machine for running a program
- So it contains state
- And resources required to do its work
  - Like threads, virtual memory, communications primitives
- Most machines run multiple processes
  - Serially and simultaneously

Processes and Programs

- A program is a static representation of work to be done
- A process is the dynamic, running instantiation of a program
- Most programs are run many different times
  - On the same or different machines
- Each individual run is represented by a unique process
  - Which has a discrete start and (usually) end

How Does a Process Differ From a Thread?

- Processes are a higher level abstraction
- They can contain multiple threads
  - Implying that there can be simultaneous actions within one program
  - Which is not possible in a thread
- They typically encapsulate an entire running program
- They are heavier weight

The OS and Processes

- The OS must multiplex virtual processes onto physical processors
  - Start and end processes
  - Set them up to run properly
  - Isolate them from other processes
  - Ensure that all processes get a chance to do their work
  - Share the physical resources properly
- One important aspect of this task is properly handling process state

Process State

- Similar to thread state
- Need information on:
  - What instruction to run next
  - Where the process' memory is located
  - What are the contents of important registers
  - What other resources (physical or virtual) are available to the process
  - Perhaps security-related information (like owner)
- Major components are register state (e.g., the PC) and memory state

Process State and Memory

- Processes have several different types of memory segments
  - The memory holding their code
  - The memory holding their stack
  - The memory holding their data
- Each is somewhat different in its purpose and use

Process Code Memory

- The instructions to be executed to run the process
- Typically static
  - Loaded when the process starts
  - Then they never change
- Of known, fixed size
- Often, a lot of the program code will never be executed by a given process running it

Implications for the OS

- Obviously, memory object holding the code must allow execution
  - Need not be writeable
    - Self-modifying code is a bad idea, usually
  - Should it be readable?
- Can use a fixed size domain
  - Which can be determined before the process executes
- Possibility of loading the code on demand

Process Stack Memory

- Memory holding the run-time state of the process
- Modern languages and operating systems are stack oriented
  - Routines call other routines
  - Expecting to regain control when the called routine exits
  - Arbitrarily deep layers of calling
- The stack encodes that

Stack Frames

- Each routine that is called keeps its relevant data in a stack frame
  - Its own piece of state
- Stack frames contain:
  - Storage for procedure local (as opposed to global) variables
  - Storage for invocation parameters
  - Space to save and restore registers
    - Popped off stack when call returns

Characteristics of Stack Memory

- Of unknown and changing size
  - Grows when functions are called
  - Shrinks when they return
- Contents created dynamically
  - Not the same from run to run
  - Often data-dependent
- Not inherently executable
  - Contains pointers to code, not code itself
- A compact encoding of the dynamic state of the process

Implications for the OS

- The memory domain for the stack must be readable and writeable
  - But need not be executable
- OS must worry about stack overrunning the memory area it's in
  - What to do if it does?
    - Extend the domain?
    - Kill the process?

Process Data Memory

- All the data the process is operating on
- Of highly varying size
  - During a process run
  - From run to run of a process
- Read/write access required
  - Usually not execute access
  - Few modern systems allow processes to create new code

Implications for the OS

- Must be prepared to give processes new domains for dynamic data
  - Since you can’t generally predict ahead of time how much memory a process will need
  - Need strategy if process asks for more memory than you can give it
- Should give read/write permission to these domains
  - Usually not execute

Layout of Process in Memory

- In Unix systems, data segment grows up
- Stack segment grows down
- They aren't allowed to meet

Loading Programs Into Processes

- The program represents a piece of code that could be executed
- The process is the actual dynamic executing version of the program
- To get from the code to the running version, you need to perform the *loading* step
  - Initializing the various memory domains we just mentioned

Loading Programs

- The load module
  - All external references have been resolved
  - All modules combined into a few segments
  - Includes multiple segments (code, data, symbol table)
- 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

Shareable Executables

- Often multiple programs share some code
  - E.g., widely used libraries
- Do we need to load a different copy for each process?
  - Not if all they're doing is executing the code
- OS can load one copy and make it available to all processes that need it
  - Obviously not in a writeable domain

Some Caveats

- Code must be relocated to specific addresses
  - All processes must use shared code at the same address
- Only the code segments are sharable
  - Each process requires its own copy of writable data
    - Which may be associated with the shared code
  - Data must be loaded into each process at start time

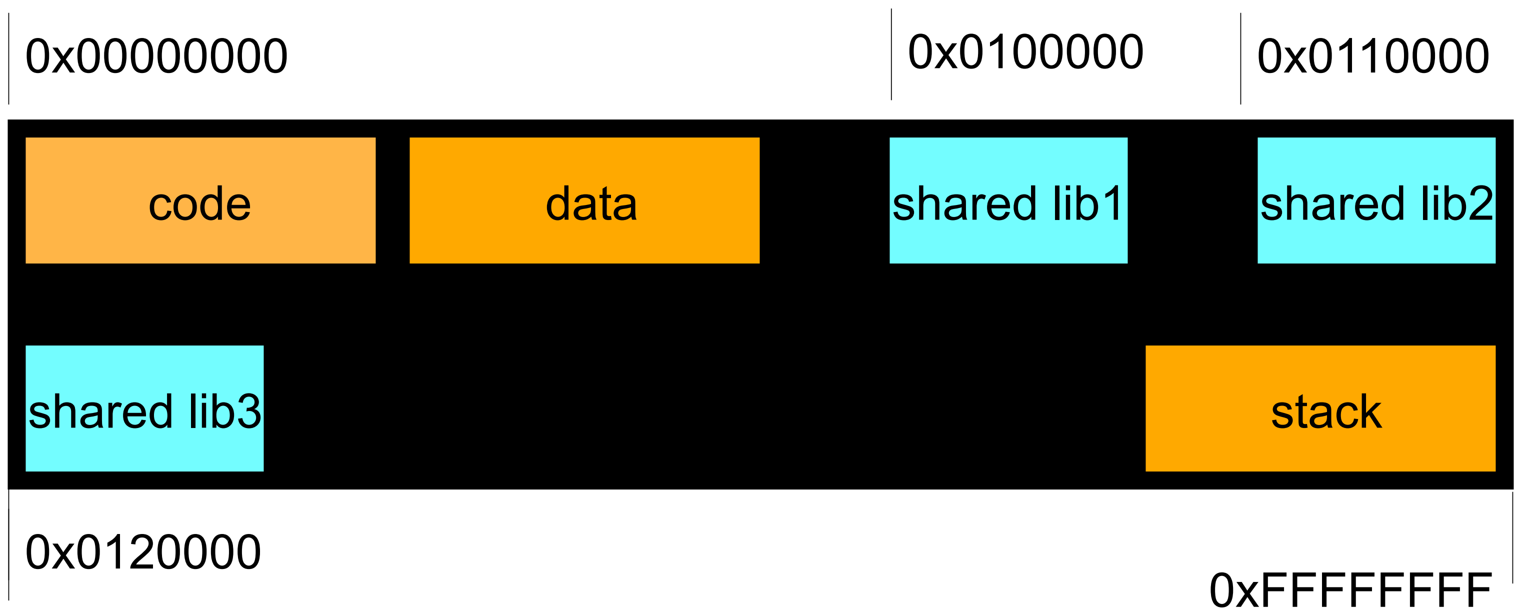
Shared Libraries

- Commonly used pieces of code
  - Like I/O routines or arithmetic functions
- Some obvious advantages:
  - Reduced memory consumption
  - Faster program start-ups, since library is often already in memory
  - Simplified updates
    - All programs using it updated by just updating the library

Limitations of Shared Libraries

- Not all modules will work in a shared library
  - They cannot define/include static data storage
- They are read into program memory
  - Whether they are actually needed or not
- Called routines must be known at compile-time
  - Only fetching the code is delayed until run-time
- Dynamically loaded libraries solve some of these problems

Layout With Shared Libraries



Dynamically Loadable Libraries

- DLLs
- Libraries that are not loaded when a process starts
- Only made available to process if it uses them
  - No space/load time expended if not used
- So action must be taken if a process does request a DLL routine
- Essentially, need to make it look like the library was there all along

Making DLLs Work

- The program load module includes a Procedure Linkage Table
  - Addresses for routines in DLL resolve to entries in PLT
  - Each PLT entry contains a system call to a run-time loader
- First time a routine is called, we call run-time loader
  - Which finds, loads, and initializes the desired routine
  - Changes the PLT entry to be a jump to loaded routine
  - Then jumps to the newly loaded routine
- Subsequent calls through that PLT entry go directly

Shared Libraries Vs. DLLs

- Both allow code sharing and run-time binding
- Shared libraries:
  - Simple method of linking into programs
  - Shared objects obtained at program load time
- Dynamically Loadable Libraries:
  - Require more complex linking and loading
  - Modules are not loaded until they are needed
  - Complex, per-routine, initialization possible
    - E.g., allocating private data area for persistent local variables

How Do Threads Fit In?

- How do multiple threads in the same process affect layout?
- Each thread has its own registers, PS, PC
- Each thread must have its own stack area
- Maximum size specified at thread creation
  - A process can contain many threads
  - They cannot all grow towards a single hole
  - Thread creator must know max required stack size
  - Stack space must be reclaimed when thread exits

Thread Stack Allocation

Problems With Fixed Size Thread Stacks

- Requires knowing exactly how deep a thread stack can get
  - Before we start running the thread
- Problematic if we do recursion
- How can developers handle this limitation?
  - The use of threads is actually relatively rare
  - Generally used to perform well understood tasks
  - Important to keep this limitation in mind when writing multi-threaded algorithms

How Does the OS Handle Processes?

- The system expects to handle multiple processes
  - Each with its own set of resources
  - Each to be protected from the others
- Memory management handles stomping on each other's memory
  - E.g., use of domain registers
- How does the OS handle the other issues?

Basic OS Process Handling

- The OS will assign processes (or their threads) to cores
  – If more processes than cores, multiplexing them as needed
- When new process assigned to a core, that core must be initialized
  – To give the process illusion that it was always running there
  – Without interruption

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
- Kept in an OS data structure
- Used for scheduling, security decisions, allocation issues

The Process Control Block

- The data structure Linux (and other Unix systems) use to handle processes
- 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
  - Accounting information
  - And various other things

OS State For a Process

- The state of process's virtual computer
- Registers
  - Program counter, processor status word
  - Stack pointer, general registers
- Virtual address space
  - Text, data, and stack segments
  - Sizes, locations, and contents
- All restored when the process is dispatched
  - Creating the illusion of continuous execution

Process Resource References

- OS needs to keep track of what system resources the process has available
- Extremely important to get this right
  - Process expects them to be available when it runs next
  - If OS gives something it shouldn't, major problem
- OS maintains unforgeable handles for allocated resources
  - Encoding identity and resource state
  - Also helpful for reclamation when process ends

Why Unforgeable Handles?

- Process can ask for any resource
- But it shouldn't always get it
- Process must not be able to create its own OS-level handle to access a resource
  - OS must control which ones the process gets
  - OS data structures not accessible from user-mode
  - Only altered by trusted OS code
    - So if it's there, the OS put it there
    - And it has not been modified by anyone else

Process Creation

- Processes get created (and destroyed) all the time in a typical computer
- Some by explicit user command
- Some by invocation from other running processes
- Some at the behest of the operating system
- How do we create a new process?

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

What Else Does a New Process Need?

- A virtual 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 process
- On assumption that the new 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 parents' . . .
- 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)
  - Share exclusive-access resources (e.g. tape drives)

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 child is expensive
  - And fork was supposed to be cheap
- If neither parent nor child write the parent's data area, though, no copy necessary
- 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

Sample Use of Fork

```
if (fork() ) {
     /* I'm the parent!   */
      execute parent code
} else {
     /* I'm the child! */
      execute the child code
}
```

- Parent and child code could be very different
- In fact, often you want the child to be a totally different program
  - And maybe not share the parent's resources

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

Using exec

```
if (fork() ) {
    /* I'm the parent!   */
     continue with what I was doing before
} else {
    /* I'm the child! */
     exec("new program", <program arguments>);
}
```

- The parent goes on to whatever is next
- The child replaces its code with "new program"

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

New Processes and Threads

- All processes have at least one thread
  - In some older OSes, never more than one
    - In which case, the thread is not explicitly represented
  - In newer OSes, processes typically start with one thread
- As process executes, it can create new threads
- New thread stacks allocated as needed

A Thread Implementation Choice

- Threads can be implemented in one of two ways

1. The kernel implements them
2. User code implements them

- These alternatives have fundamental differences

User Threads

- The kernel doesn't know about multiple threads per process
- But the process itself knows
- So the process must schedule its threads
- Since the kernel doesn't know the process has multiple threads,
  - The process can't run threads on more than one core
- Switching threads doesn't require OS involvement, though
  - Which can be cheaper

Typical Use of User Threads

- A server process that expects to have multiple simultaneous clients
- Server process can spawn a new user thread for each client
- And can then use its own scheduling methods to determine which thread to run when
- OS need not get involved in running threads
  - No context switch costs to change from one client to another

Kernel Threads

- The OS is aware that processes can contain more than one thread
- Creating threads is an OS operation
- Scheduling of threads handled by OS
  - Which can schedule several process threads on different cores simultaneously
- Saves the program complexity of handling threads
- But somewhat more heavyweight

Typical Use of Kernel Threads

- A program that can do significant parallel processing on its data
- Each parallel operation is run as a kernel thread
  - All sharing the same data space and code
  - But each with its own stack
- If multiple cores available, OS can achieve true parallelism for the program

Process Termination

- 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