Processes

- 2D Services and abstract resources
- 3A What is a Process
- 3B Process Address Space
- 3C Process Operations
- 3D Implementing Processes

Services: an object-oriented view

- my execution platform implements objects
  - they may be bytes, longs and strings
  - they may be processes, files, and sessions
- an object is defined by
  - its properties, methods, and their semantics
- what makes a particular set of objects good
  - they are powerful enough to do what I need
  - they don't force me to do a lot of extra work
  - they are simple enough for me to understand

Better Objects and Operations

- easier to use than the original resources
  - disk I/O without DMA, interrupts and errors
- compartmentalize/encapsulate complexity
  - a securely authenticated/encrypted channel
- eliminate behavior that is irrelevant to user
  - hide the complex erase cycle of flash memory
- create more convenient behavior
  - highly reliable/available storage from anywhere

Simplifying Abstractions

- hardware is fast, but complex and limited
  - using it correctly is extremely complex
  - it may not support the desired functionality
  - it is not a solution, but merely a building block
- encapsulate implementation details
  - error handling, performance optimization
  - eliminate behavior that is irrelevant to the user
- more convenient or powerful behavior
  - operations better suited to user needs

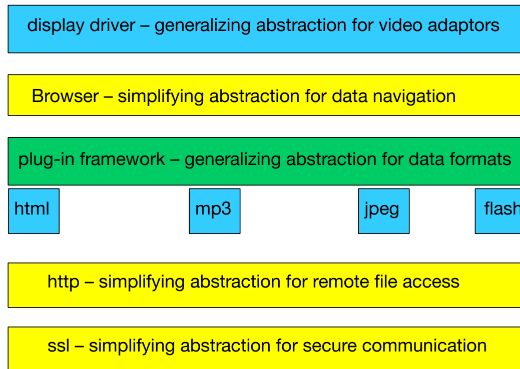
Generalizing Abstractions

- make many different things appear the same
  - applications can all deal with a single class
  - often Lowest Common Denominator + sub-classes
- requires a common/unifying model
  - *portable document format* for printed output
  - SCSI/SATA/SAS standard for disks, CDs, SSDs
- usually involves a federation framework
  - device-specific drivers
  - browser plug-ins to handle multi-media data

Federation Frameworks

- (pre-O-O) alternative to *sub-classing*
  - *"Adapter"* implementation integration framework
  - defines an *interface* that all must meet
  - implementations can be registered (plugged in)
  - framework directs calls to correct implementation
- allows use of many different implementations
  - device drivers
  - file systems
  - sound/video types

Layers of Abstraction: a browser



Building Blocks and World Views

- An OS is a general purpose platform
  - it must support a wide range of applications
  - including those to be designed in the future
- OS services are software building blocks
  - not solutions, but pieces for building solutions
- OS abstractions represent a world view
  - concepts that encompass all possible s/w
  - interaction rules to govern their combinations
  - frame (guide/constrain) all future discussions

Virtualizing Physical Resources

- serially reusable (temporal multiplexing)
  - used by multiple clients, one at a time
  - requires access control to ensure exclusive access
- partitionable resources (spatial multiplexing)
  - different clients use different parts at same time
  - requires access control for containment/privacy
- sharable (no apparent partitioning or turns)
  - often involves mediated access
  - often involves under-the-covers multiplexing

Serially Reusable Resources

- Used by multiple clients ... one at a time
  - temporal multiplexing
- Require access control to ensure exclusivity
  - while A has resource, nobody else can use it
- Require graceful transitions between users
  - B will not start until A finishes
  - B receives resource in like-new condition
- Examples: printers, bathroom stalls

Partitionable Resources

- Divided into disjoint pieces for multiple clients
  - Spatial multiplexing
- Needs access control to ensure:
  - Containment: *you cannot access resources outside of your partition*
  - Privacy: *nobody else can access resources in your partition*
- Examples: RAM, hotel rooms

Shareable Resources

- Usable by multiple concurrent clients
  - clients do not have to “wait” for access to resource
  - clients don’t “own” a particular subset of resource
- May involve (effectively) limitless resources
  - air in a room, shared by occupants
  - copy of the operating system, shared by processes
- May involve under-the-covers multiplexing
  - shared network interface (time multiplexed)
  - cell-phone channels (CDMA spread-spectrum)

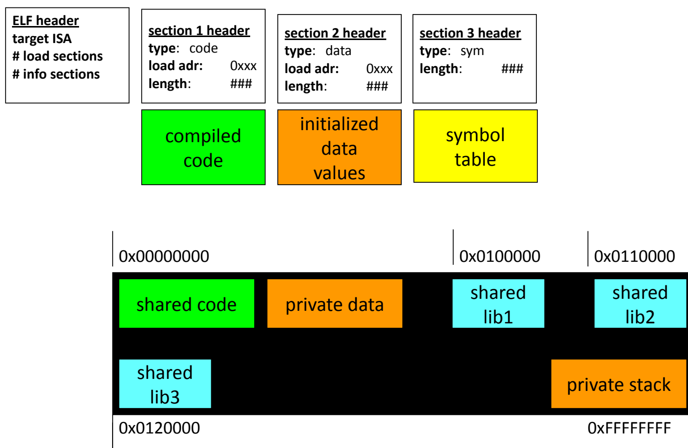
What is a Process?

- an executing instance of a program
  - how is this different from a program?
- a virtual private computer
  - what does a virtual computer look like?
  - how is a process different from a virtual machine?
- a process is an *object*
  - characterized by its properties (state)
  - characterized by its operations

What is "state"?

- the primary 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 it 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)

Program vs Process Address Space



Address Space: Code Segments

- load module (output of 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/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: 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 (with sbrk syscall)

Address Space: Stack Segment

- size of stack depends on program activities
  - grows larger as calls nest more deeply
  - amount of local storage allocated by each procedure
  - after calls return, their stack frames can be recycled
- OS manages the process's stack segment
  - stack segment created at same time as data segment
  - some allocate fixed sized stack at program load time
  - some dynamically extend stack as program needs it
- Stack segments are read/write and process private

Address Space: Shared 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
- make each library a sharable code segment
  - 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, (perhaps) frame pointer
- processes own OS resources
  - open files, current working directory, locks
- processes have OS-related state
  - Process ID, User ID, Group ID, scheduling priority
  - registered signal handlers, queued events, …

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

Variations on Process Creation

- tabula rasa – a blank slate
  - a new process with minimal resources
  - a few resources may be passed from parent
  - most resources opened, from scratch, by child
- cloning fork (from classic parallelism theory)
  - much data and resources to be copied
  - convenient for setting up pipelines
  - allows (limited) sharing of data and resources
- run – fork + exec
  - create new process to run a specified command

Process Forking

- The way Unix/Linux creates processes
  - child is a clone of the parent
  - the classical Computer Science *fork* operation
- <u>Occasionally</u> a clone is what you wanted
  - likely for some kinds of parallel programming
  - more likely wanted threads (in same address space)
- Program in child process can adjust resources
  - change input/output file descriptors
  - change working directory
  - change environment variables
  - choose which program to run

Process Operations: fork

- parent and child are identical:
  - data and stack segments are copied
  - all the same files are open
- code sample:

```
int rc = fork();
if (rc < 0) {
   fprintf(stderr, “Fork failed\n”);
} else if (rc == 0) {
   fprintf(stderr, “Child\n”);
} else
   fprintf(stderr, “Fork succeeded, child pid = %d\n”, rc);
```

What Happens After a Fork?

- There are now two processes
  - with different process ID numbers
  - but otherwise nearly (modulo PID #) identical
- How do I profitably use that?
  - two processes w/same code & program counter
  - figure out which is which
  - parent process goes one way
  - child process goes another
    - perhaps adjust process resources
    - perhaps load (*exec*) a new program into the process
    - this code takes the place of (*CreateProcess*) parameters

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
- Practical reasons
  - easy to manage shared resources
    - like stdin, stdout, stderr
  - easy to set up process pipe-lines (e.g. ls | more)
  - simplifies design of command shells

Fork isn't what I usually want!

- two identical processes
  - running the same program on the same data
- we usually want new program in new process
  - to do a different thing in a different process
- *fork(2)* and *exec(2)* are orthogonal operations
  - *fork(2)* creates a new process
  - *exec(2)* loads a new program into a process

Process Operations: exec

- load new program, pass parameters
  - address space is completely recreated
  - open files remain open, disabled signals disabled
  - available in many polymorphisms
- code sample:

```
char *myargs[3];
myargs[0] = "/usr/bin/wc";
myargs[1] = "myfile";
myargs[2] = NULL;
int rc = execvp(myargs[0], myargs);
```


Windows Process Creation

- The `CreateProcess()` system call
- A very flexible way to create a new process
- Numerous parameters to shape the child
  - name of program to run
  - security attributes (new or inherited)
  - open file handles to pass/inherit
  - environment variables
  - initial working directory

How Processes Terminate

- Perhaps voluntarily
  - by calling the *exit(2)* system call
- Perhaps involuntarily
  - as a result of an unhandled signal/exception
  - a few signals (e.g. SIGKILL) cannot be caught
- Perhaps at the hand of another
  - a parent sends a termination signal (e.g. TERM)
  - a system management process (e.g. INT, HUP)

Process Operations: wait

- await termination of a child process
  - collect exit status
- code sample:

```
pid_t pid  = waitpid(P_ALL, &status, WEXITED);
if (pid > 0) {
  fprintf(stderr, “process %d exited rc=%d\n”, pid, status);
}
```

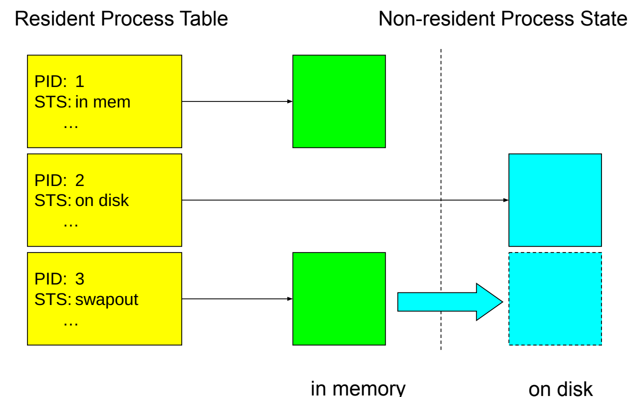

The State of a Process (review)

- Registers … same as phys/virtual machine
  - Program Counter, Processor Status Word
  - Stack Pointer, general registers
- Address space … abstraction of phys memory
  - size and location of text, data, stack segments
  - size and location of supervisor mode stack
- System Resources and Parameters
  - open files, current working directory, ...
  - owning user ID, parent PID, scheduling priority, ...

Representing a Process

- all (not just OS) objects have descriptors
  - the identity of the object
  - the current state of the object
  - references to other associated objects
- Process state is in multiple places
  - attributes and object references in a descriptor
  - app execution state is on the stack, in registers
  - each Linux process has a supervisor-mode stack
    - to retain the state of in-progress system calls
    - to save the state of an interrupt preempted process

Resident and non-Resident State



(resident process descriptor)

- state that could be needed at any time
- information needed to schedule process
  - run-state, priority, statistics
  - data needed to signal or awaken process
- identification information
  - process ID, user ID, group ID, parent ID
- communication and synchronization resources
  - semaphores, pending signals, mail-boxes
- pointer to non-resident state

(non-resident process state)

- information needed only when process runs
  - can swap out to free memory for other processes
- execution state
  - supervisor mode stack
  - including: saved register values, PC, PS
- pointers to resources used when running
  - current working directory, open file descriptors
- pointers to text, data and stack segments
  - used to reconstruct the address space

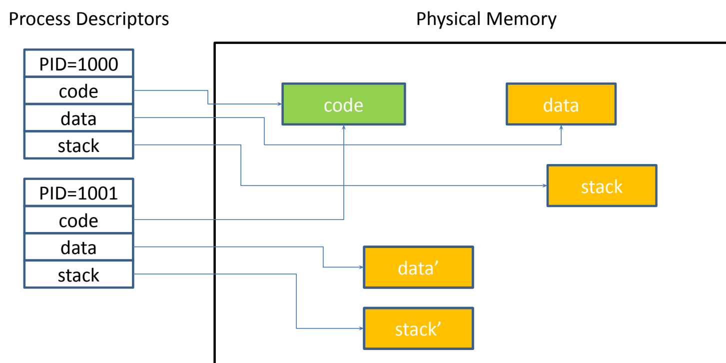
Creating a new process

- allocate/initialize resident process description
- allocate/initialize non-resident description
- duplicate parent resource references (e.g. fds)
- create a virtual address space
  - allocate memory for code, data and stack
  - load/copy program code and data
  - copy/initialize a stack segment
  - set up initial registers (PC, PS, SP)
- return from supervisor mode into new process

Forking and the Data Segments

- Forked child shares parent's code segment(s)
  - read only segment(s), referenced by both
- Stack and Data segments are private
  - each process has its own read/write copy
  - child's is initialized as a copy of parent's
  - copies diverge w/subsequent updates
- Common optimization: **Copy-on-Write**
  - start with a single shared read/only segment
  - make a copy only if parent (or child) changes it

Forking a New Process



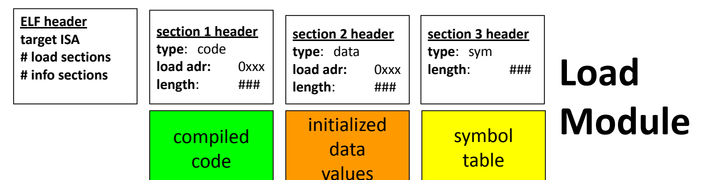
Forking and Copy on Write

- 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

Loading (exec) a Program

- We 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
  - An entirely new address space must be created
  - Memory must be allocated for each segment
  - Code must be copied from load module to memory

Loading a new program (exec)



Process Virtual Address Space

How to Terminate a Process?

- Reclaim any resources it may be holding
  - memory
  - locks
  - references to open files or network connections
- Inform any other process that needs to know
  - those waiting for interprocess communications
  - parent (and maybe child) processes
- Remove process descriptor from process table

Reading and Assignments

Reading:

- Kampe: Interface Stability
- Linking and Libraries (should be review)
- Linkage Conventions (should be review)
- signal(2), kill(2) system calls

Projects:

- start looking at project 1A (basic process operations)
- check-out (or order) your embedded system and sensors

Supplementary Slides

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