Operating Systems Principles
Resources, Services, and Interfaces

Mark Kampe
(markk@cs.ucla.edu)

Resources, Services, and Interfaces

2A. Operating Systems Services
2B. System Service Layers and Mechanisms
2C. Service Interfaces and Standards
2D. Service and Interface Abstractions

Services: Hardware Abstractions

- CPU/Memory abstractions
  - processes, threads, virtual machines
  - virtual address spaces, shared segments
  - signals (as execution exceptions)
- Persistent Storage abstractions
  - files and file systems, virtual LUNs
  - databases, key/value stores, object stores
- other I/O abstractions
  - virtual terminal sessions, windows
  - sockets, pipes, VPNs, signals (as interrupts)

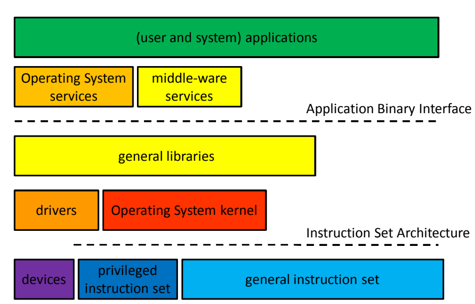
Services: Higher Level Abstractions

- cooperating parallel processes
  - locks, condition variables
  - distributed transactions, leases
- security
  - user authentication
  - secure sessions, at-rest encryption
- user interface
  - GUI widgetry, desktop and window management
  - multi-media

Services: under the covers

- enclosure management
  - hot-plug, power, fans, fault handling
- software updates and configuration registry
- dynamic resource allocation and scheduling
  - CPU, memory, bus resources, disk, network
- networks, protocols and domain services
  - USB, BlueTooth
  - TCP/IP, DHCP, LDAP, SNMP
  - iSCSI, CIFS, NFS

Software Layering



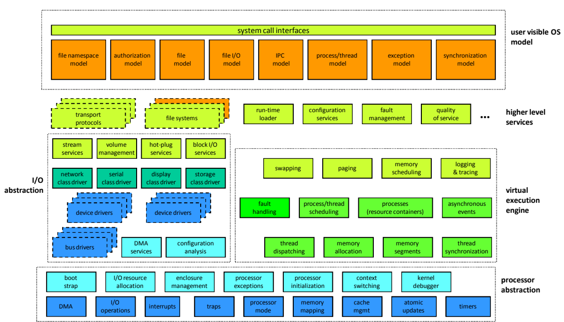
Service delivery via subroutines

- access services via direct subroutine calls
  - push parameters, jump to subroutine, return values in registers on on the stack
- advantages
  - extremely fast (nano-seconds)
  - DLLs enable run-time implementation binding
- disadvantages
  - all services implemented in same address space
  - limited ability to combine different languages

Layers: libraries

- convenient functions we use all the time
  - reusable code makes programming easier
  - a single well written/maintained copy
  - encapsulates complexity … better building blocks
- multiple bind-time options
  - static … include in load module at link time
  - shared … map into address space at exec time
  - dynamic … choose and load at run-time
- it is only code … it has no special privileges

Kernel Structure (artists conception)



Service delivery via system calls

- force an entry into the operating system
  - parameters/returns similar to subroutine
  - implementation is in shared/trusted kernel
- advantages
  - able to allocate/use new/privileged resources
  - able to share/communicate with other processes
- disadvantages
  - all implemented on the local node
  - 100x-1000x slower than subroutine calls

Layers: the kernel

- primarily functions that require privilege
  - privileged instructions (e.g. interrupts, I/O)
  - allocation of physical resources (e.g. memory)
  - ensuring process privacy and containment
  - ensuring the integrity of critical resources
- some operations may be out-sourced
  - system daemons, server processes
- some plug-ins may be less-trusted
  - device drivers, file systems, network protocols

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 <u>mediated</u> access
  - often involves <u>under-the-covers</u> multiplexing

Layers: system services

- not all trusted code must be in the kernel
  - it may not need to access kernel data structures
  - it may not need to execute privileged instructions
- some are actually privileged processes
  - login can create/set user credentials
  - some can directly execute I/O operations
- some are merely trusted
  - sendmail is trusted to properly label messages
  - NFS server is trusted to honor access control data

Service delivery via messages

- exchange messages with a server (via syscalls)
  - parameters in request, returns in response
- advantages:
  - server can be anywhere on earth
  - service can be highly scalable and available
  - service can be implemented in user-mode code
- disadvantages:
  - 1,000x-100,000x slower than subroutine
  - limited ability to operate on process resources

Layers: middle-ware

- Software that is a key part of the application or service platform, but not part of the OS
  - database, pub/sub messaging system
  - Apache, Nginx
  - Hadoop, Zookeeper, Beowulf, OpenStack
  - Cassandra, RAMCloud, Ceph, Gluster
- Kernel code is very expensive and dangerous
  - user-mode code is easier to build, test and debug
  - user-mode code is much more portable
  - user-mode code can crash and be restarted

Application Programming Interfaces

- a source level interface, specifying
  - include files
  - data types, data structures, constants
  - macros, routines, parameters, return values
- a basis for software portability
  - recompile program for the desired ISA
  - linkage edit with OS-specific libraries
  - resulting binary runs on that ISA and OS

Application Binary Interfaces

- a binary interface, specifying
  - load module, object module, library formats
  - data formats (types, sizes, alignment, byte order)
  - calling sequences, linkage conventions
- a basis for binary compatibility
  - one binary will run on any ABI compliant system
    - e.g. all x86 Linux/BSD/OSx/Solaris/…
    - may even run on windows platforms

Other interoperability interfaces

- Data formats and information encodings
  - multi-media content (e.g. MP3, JPG)
  - archival (e.g. tar, gzip)
  - file systems (e.g. DOS/FAT, ISO 9660)
- Protocols
  - networking (e.g. ethernet, WLAN, TCP/IP)
  - domain services (e.g. IMAP, LPD)
  - system management (e.g. DHCP, SNMP, LDAP)
  - remote data access (e.g. FTP, HTTP, CIFS, S3)

Interoperability requires compliance

- Complete interoperability testing impossible
  - cannot test all applications on all platforms
  - cannot test interoperability of all implementations
  - new apps and platforms are added continuously
- Rather, we focus on the interfaces
  - interfaces are completely and rigorously specified
  - standards bodies manage the interface definitions
  - compliance suites validate the implementations
- and hope that sampled testing will suffice

Interoperability requires stability

- no program is an island
  - programs use system calls
  - programs call library routines
  - programs operate on external files
  - programs exchange messages with other software
- API requirements are frozen at compile time
  - execution platform must support those interfaces
  - all partners/services must support those protocols
  - all future upgrades must support older interfaces

Compatibility Taxonomy

- upwards compatible (with …)
  - new version still supports previous interfaces
- backwards compatible (with …)
  - will correctly interact with old protocol versions
- versioned interface, version negotiation
  - parties negotiate a mutually acceptable version
- compatibility layer
  - a cross-version translator
- non-disruptive upgrade

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

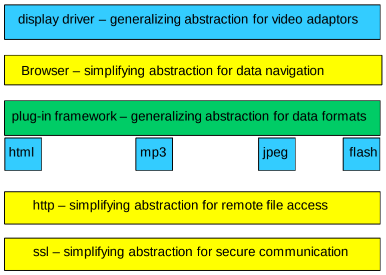
Simplifying Abstractions

- 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

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

assignments

- reading for the next lecture
  - Arpaci ch 3 … introduction
  - Arpaci ch 4 … Processes
  - Arpaci ch 5 … Process API
  - Arpaci ch 6 … Direct Execution
  - manual sections: kill(2), signal(2)

Quiz 3 is due <u>before</u> the lecture!

Try to complete project 0 before lab session

Supplementary Slides

Instruction Set Architectures

- the set of instructions supported by a computer
  - what bit patterns correspond to what operations
- there are many different ISAs (all incompatible)
  - different word/bus widths (8, 16, 32, 64 bit)
  - different design philosophies (RISC vs CISC)
  - competitive reasons (68000, x86, PowerPC)
- they usually come in families
  - newer models add features (e.g. Pentium vs 386)
  - try to remain upwards-compatible with older models
  - occasional discontinuities are inevitable (e.g. IA64)

Portability to multiple ISAs

- start with API compliance
- data type dependencies
  - word length (e.g. of "int")
  - byte order (e.g. in messages or bit processing)
  - alignment (of fields in data structures)
- code dependencies
  - use of vendor specific libraries or functions
  - use of in-line assembler

Standards in the Dark Ages (1965)

- no software industry as we now know it
- all the money was made on hardware
  - but hardware is useless without software
  - all software built by hardware suppliers
  - platforms were distinguished by software
- software portability was an anti-goal
  - keep customers captive to your hardware
  - portability means they could go elsewhere
- standards were few and weak

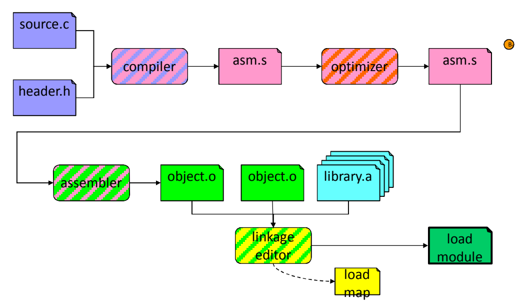
The Software Reformation (1985)

- the advent of the "killer application"
  - desk-top publishing, spreadsheets, ...
  - the rise of the Independent Software Vendor
- fundamental changes to platform industry
  - the “applications, demand, volume” cycle
  - application capture became strategic
- applications portability became strategic
  - standards are the key to portability
  - standards compliance became strategic

The Role of Standards Today

- there are many software standards
  - subroutines, protocols and data formats, …
  - both portability and interoperability
  - some are general (e.g. POSIX 1003, TCP/IP)
  - some are very domain specific (e.g. MPEG2)
- key standards are widely required
  - non-compliance reduces application capture
  - non-compliance raises price to customers
  - proprietary extensions are usually ignored

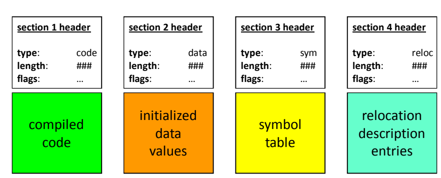
the compilation process



(Compilation/Assembly)

- compiler
  - reads source code and header files
  - parses and understands "meaning" of source code
  - optimizer decides how to produce best possible code
  - code generation typically produces assembler code
- assembler
  - translates assembler directives into machine language
  - produces relocatable object modules
    - code, data, symbol tables, relocation information

Typical Object Module Format

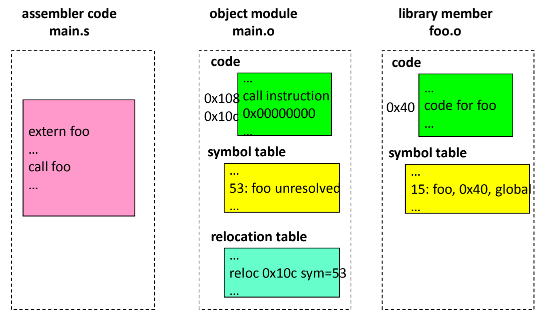


each code/data section is a block of information that should be kept together, as a unit, in the final program

(Relocatable Object Modules)

- code segments
  - relocatable machine language instructions
- data segments
  - non-executable initialized data, also relocatable
- symbol table
  - list of symbols defined and referenced by this module
- relocation information
  - pointers to all relocatable code and data items

object modules, symbols, & relocation



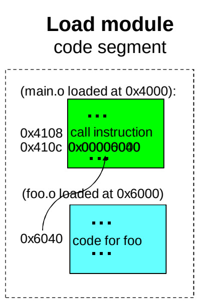
Libraries

- programmers need not write all code for programs
  - standard utility functions can be found in libraries
- a library is a collection of object modules
  - a single file that contains many files (like a zip or jar)
  - these modules can be used directly, w/o recompilation
- most systems come with many standard libraries
  - system services, encryption, statistics, etc.
  - additional libraries may come with add-on products
- programmers can build their own libraries
  - functions commonly needed by parts of a product

Linkage Editing

- obtain additional modules from libraries
  - search libraries to satisfy unresolved external references
- combine all specified object modules
  - resolve cross-module references
  - copy all required modules into a single address space
  - relocate all references to point to the chosen locations
- result should be complete load module
  - no unresolved external addresses
  - all data items assigned to specific virtual addresses
  - all code references relocated to assigned addresses

Linkage editing: resolution & relocation



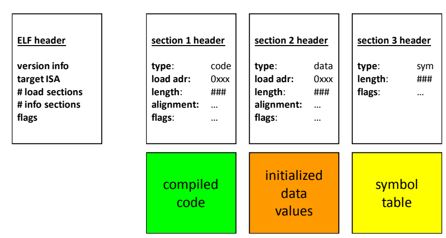
Resolution

search all specified libraries to find modules that can satisfy all unresolved external references.

Relocation

update all pointers to externally resolved symbols to correctly refer to the locations where the corresponding modules were actually loaded.

Load Modules (ELF)



program loading – executable code

- 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)
    - each to be loaded, contiguously, at a specified address
- 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
    - in ancient times this involved an additional relocation step

program loading – data segments

- code segments are read-only & fixed size
- programs include data as well as code
- data too must be initialized in address space
  - memory must be allocated for each data segment
  - initial contents must be copied from load module
  - BSS: segments to be initialized to all zeroes
- data segments read/write & variable size
  - execution can change contents of data segments
  - program can extend data segment to get more memory

Processes – the User View

- 4C – sharable and dynamically loadable code
- Shared Executables
  - advantages and use
- Shared Libraries
  - advantages
  - implementation
- Dynamically Loadable Libraries
  - advantages
  - implementation

Sharable executables

- code segments are usually read-only
  - one copy could be shared by multiple processes
  - allow more process to run in less memory
- code has been relocated to specific addresses
  - all procs must use shared code at the same address
- only the code segments are sharable
  - each process requires its own copy of writable data
  - data must be loaded into each process at start time

address space – shared executable

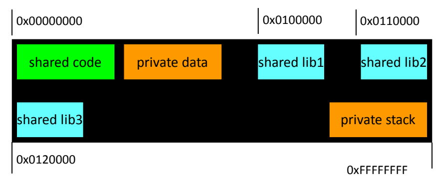
Shared Libraries

- library modules are usually added to load module
  - each load module has its own copy of each library
    - this dramatically increases the size of each process
  - program must be re-linked to incorporate new library
    - existing load modules don't benefit from bug fixes
- 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

Advantages of Shared Libraries

- reduced memory consumption
  - one copy can be shared by multiple processes/programs
- faster program start-ups
  - if it is already in memory, it need not be loaded again
- simplified updates
  - library modules are not included program load modules
  - library can be updated (e.g. new version w/ bug fixes)
  - programs automatically get new version when restarted

address space – shared libraries



Implementing Shared Libraries

- multiple code segments in a single address space
  - one for the main program, one for each shared library
  - each sharable, and mapped in at a well-known address
- deferred binding of references to shared libs
  - applications are linkage edited against a stub library
    - stub module has addresses for each entry point, but no code
    - linkage editor resolves all refs to standard map-in locations
  - loader must find a copy of each referenced library
    - and map it in at the address where it is expected to be

Stub modules vs real shared libraries

stub module: libfoo.a

symbol table:

0: libfoo.so, shared library
1: foosub1, global, absolute, 0x1020000
2: foosub2, global, absolute, 0x1020008
3: foosub3, global, absolute, 0x1020010
4: foosub4, global, absolute, 0x1020018
…

Program is linkage edited against the stub module, and so believes each of the contained routines to be at a fixed address.

The real shared object is mapped into the process' address space at that fixed address. It begins with a jump table, that effectively seems to give each entry point a fixed address.

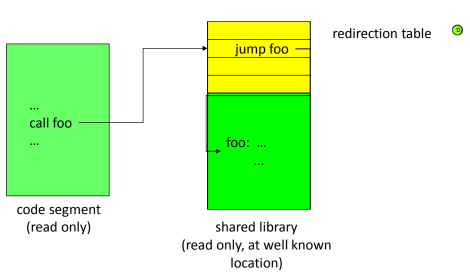
shared library: libfoo.so …

(to be mapped in at 0x1020000)

0x1020000 jmp foosub1
0x1020008 jmp foosub2
0x1020010 jmp foosub3
0x1020018 jmp foosub4
….

foosub1: …
foosub2: …

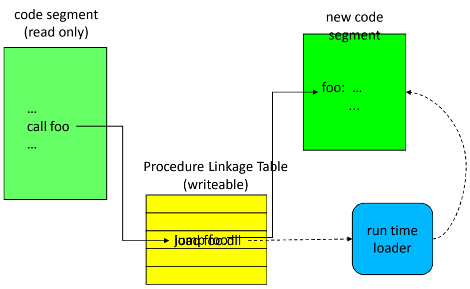
Indirect binding to shared libraries



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 the fetching of the code is delayed 'til run-time
  - symbols known at compile time, bound at link time
- Dynamically Loadable Libraries are more general
  - they eliminate all of these limitations ... at a price

Loading and Binding w/DLLs



(run-time binding to DLLs)

- 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 run-time loader (asking it to load the corresponding routine)
- 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
  - do not require a special linkage editor
  - shared objects obtained at program load time
- Dynamically Loadable Libraries
  - require smarter linkage editor, run-time loader
  - modules are not loaded until they are needed
    - automatically when needed, or manually by program
  - complex, per-routine, initialization can be performed
    - e.g. allocation of private data area for persistent local variables

Dynamic Loading

- DLLs are not merely "better" shared libraries
  - libraries are loaded to satisfy static external references
  - DLLs are designed for dynamic binding
- Typical DLL usage scenario
  - identify a needed module (e.g. device driver)
  - call RTL to load the module, get back a descriptor
  - use descriptor to call initialization entry-point
  - initialization function registers all other entry points
  - module is used as needed
  - later we can unregister, free resources, and unload

Processes – 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 computers also have stack support
  - stack too must be preserved as part of process state

Simple procedure linkage conventions

calling routine

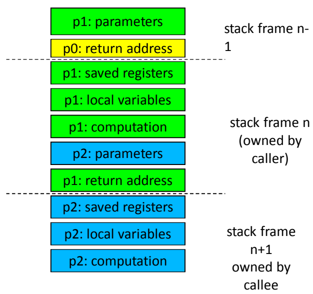
```
push p1; push first parameter
push p2; push second parameter
call foo ; save pc, call routine

add =8,sp        ; pop parameters
...
```

called routine

```
foo:   push r2-r6      ; save registers
       sub =12,sp      ; space for locals
       ...
       mov rslt,r0     ; return value
       add =12,sp      ; pop locals
       pop r2-r6       ; restore regs
       return          ; restore pc
```

Sample stack frames



Process Stacks

- size of stack depends on activity of program
  - 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

UNIX stack space management

Data segment starts at page boundary after code segment
Stack segment starts at high end of address space
Unix extends stack automatically as program needs more.

Data segment grows up; Stack segment grows down
Both grow towards the hole in the middle. They are not allowed to meet.

Thread state and thread stacks

- each thread has its own registers, PS, PC
- each thread must have its own stack area
- maximum size specified when thread is created
  - a process can contain many threads
  - they cannot all grow towards a single hole
  - thread creator must know maximum required stack size
  - stack space must be relclaimed when thread exits
- procedure linkage conventions remain the same

Thread Stack Allocation

Asynchronous Exceptions and Signals

- most program execution is synchronous
  - first execute instruction 1, then execute instruction 2
  - procedure calls are also completely synchronous
    - calling procedure stops executing until call returns
    - sub-routine return value is returned to caller upon completion
- not all events fit this synchronous model
  - program exceptions: zero-divide, illegal address, ...
  - external events: interrupt, hang-up, shut-down, ...
  - different programs may handle these in different ways
- need a way to inform program when these happen

User-mode Process Signal Handlers

- OS defines numerous types of signals
  - execution exceptions, operator actions, communication
- user-mode programs can control their handling
  - ignore this signal (pretend it never happened)
  - designate a handler for this signal
  - default action (typically kill or coredump process)
- these are analogous to hardware traps
  - but delivered by software to user-mode processes

Signal Handlers – sample code

```
int fault_expected, fault_happened;
void handler( int sig) {
        if (!fault_expected) exit(-1); /* if not expected, die */
        else fault_happened = 1;     /* if expected, note it happened */
}
signal(SIGHUP, SIGIGNORE); /* ignore hang-up signals */
signal(SIGSEGV, &handler);  /* handle segmentation faults */
...
fault_happened = 0; fault_expected = 1;
...        /* code that might cause a segmentation fault */
fault_expected = 0;
```

Signals and signal handling

- when an asynchronous exception occurs
  - the system invokes a specified exception handler
- invocation looks like a procedure call
  - save state of interrupted computation
  - exception handler can do what ever is necessary
  - handler can return and resume interrupted computation
- more complex than a procedure call and return
  - must also save/restore condition codes & volatile regs
  - may not return, rather may abort current computation

Stacking a Signal Delivery