Client-Facing Interfaces

- 2C Standards and stability
- 3Z (stacks and linkage conventions)
- 3X linkage editing
- 3Y sharable/dynamically loaded libraries
- 3E (asynchronous exceptions and events)
- 3U (user-mode exceptions)

Many Kinds of Interfaces

- Application Programming/Binary 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, WiFi, 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 Completeness

- Interface specification must be complete
  - all input parameters (w/correctness assertions)
  - all output values (w/correctness assertions)
  - all input/output behaviors
  - all dependencies, statefullness, side-effects
- All specifications should be explicit
  - no undocumented features
  - not defined by an implementation's behavior

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 …)
  - runs on version X and higher
- backwards compatible (with …)
  - can safely interact with (older) version X
- versioned interface, version negotiation
  - parties negotiate a mutually acceptable version
- compatibility layer
  - a cross-version translator
- non-disruptive upgrade
  - update components one-at-a-time w/no down-time (e.g. application upgrades on your phone)

Simple (IA-64) linkage conventions

calling routine

```
// Dianne's silk dress costs $89
mov p1, %rdi     // parm 1
mov p2, %rsi     // parm 2
…
mov p6, %r9      // parm 6
pushq p8         // parm 8 on stack
pushq p7         // parm 7 on stack
call foo         // save pc, call routine

add $16,%rsp     // on-stack parms
…
```

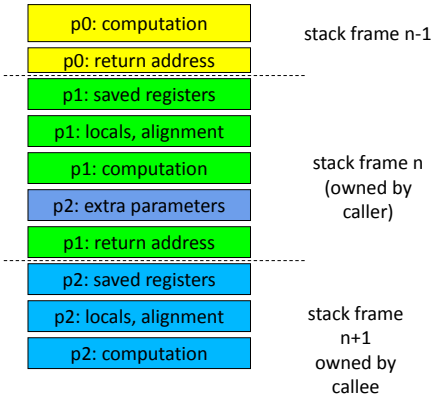
called routine

```
// callee-saved: %r12-15, %rbx, %rbp
foo:  // needs %r12,rbp, 16 bytes of locals
    pushq  %r12        // register saving
    pushq %rbp         // register saving
    subq $24, %rsp     // alignment, locals
      …

    movq rslt,%rax     // return value

    addq $24.%rsp      // free locals
    popq %rbp          // restore saved
    popq %r12          // restore saved
    ret
```

Sample (IA-64) stack frames



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.

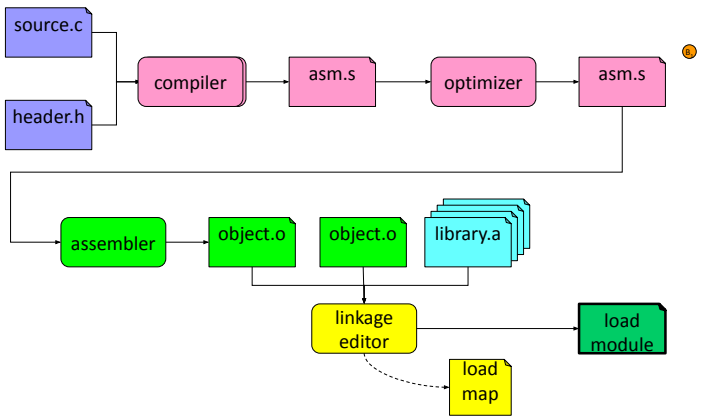
Data segment grows up; Stack segment grows down
Both grow towards the hole in the middle. They are not allowed to meet.

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 reclaimed when thread exits
- procedure linkage conventions remain same

Thread Stack Allocation

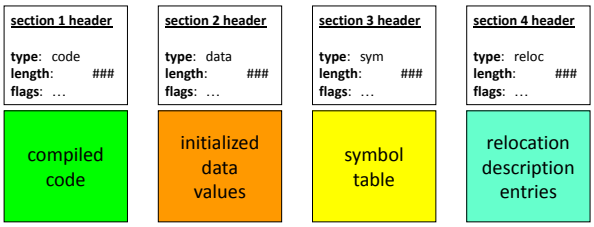
the compilation process



(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

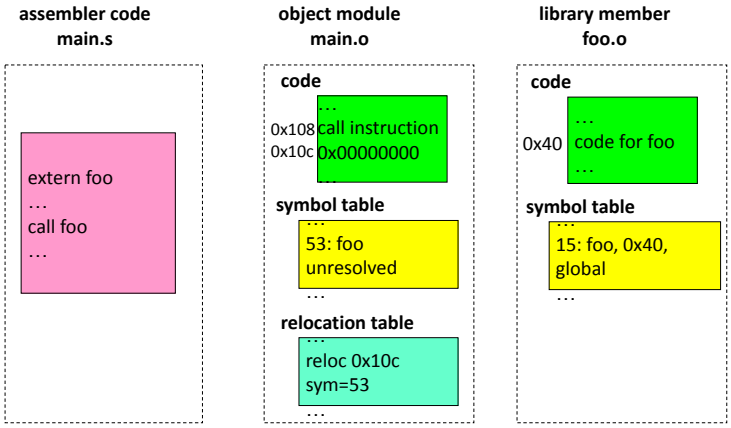


each code/data section is a block of information that should be kept together, as a unit, in the final program

(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



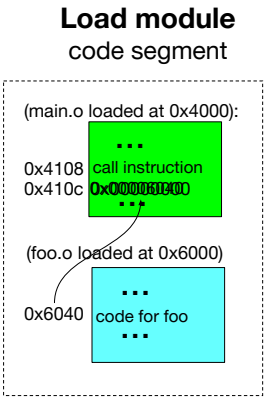
Libraries

- programmers need not write all required code
  - 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



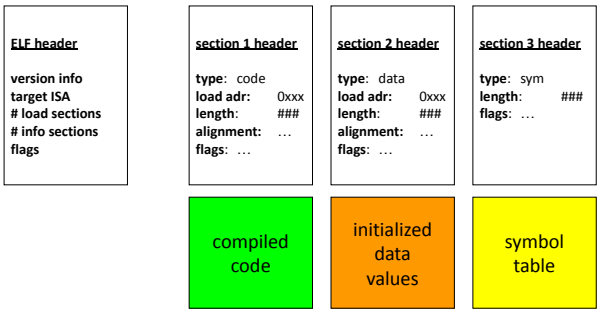
Resolution

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)



Types of Libraries

- Multiple bind-time options
  - *static* … add to load module at linkage edit time
  - *shared* … map into address space at exec time
    - reduce size of load module and in-memory copy
    - able to benefit from updated copies of library
    - client usage very similar to statically linked libraries
  - *dynamic* … choose and load at run-time
    - a very different and more powerful model
- all are merely collections of code
  - no special privileges

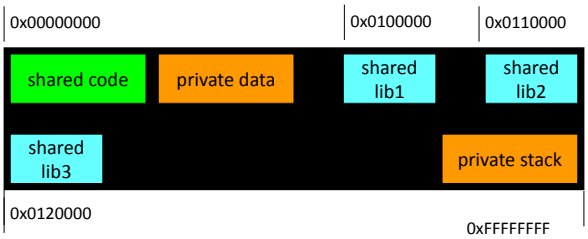
Shared Libraries

- 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



Dynamically Loaded Libraries

- DLLs are not merely "better" shared libraries
  - shared libs are loaded to satisfy static external references
  - DLLs are designed for dynamic binding
    - target DLL may not be known/exist at compile-, load-, or exec-time
- Typical DLL usage scenario
  - identify needed module (e.g. new device or media type)
  - call `dlopen(module, binding-options)`
    - run-time loader will bring new module into our address space
    - DLL can use symbols in calling executable (or other libraries)
    - returns a handle for subsequent calls into that DLL
  - use `handle->dlsym()` to find desired entry points
    - returns a pointer to desired routine
    - multiple DLLs can export the same entry points
  - use `handle->dlclose()` to unload module

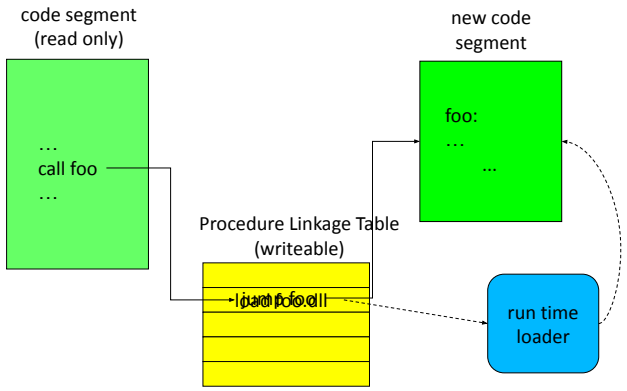
simple DLL usage example

```
#include <stdlib.h>
#include <dlfcn.h>
…
    char *image_lib = argv[2];   // DLL to be used
    char *image_file = argv[3];  // image file to be sized
    …
    void *handle = dlopen(image_lib, RTLD_LAZY);
    if (handle) {
        int (*pixels)(char *) = dlsym(handle, "image_size");
        if (pixels)
            int tot_size = pixels(image_file);
        …
        dlclose(handle);
    }
```


Run-Time Loader Binding Options

- DLL can make calls into other modules
  - to the hosting executable
    - if linked with -rdynamic or –export-dynamic option
  - to other DLLs
    - if they were loaded with RDLD_GLOBAL option
- Run-Time Loader resolves these references
  - RTLD_LAZY option
    - resolve undefined symbols as they are called
  - RTLD_NOW
    - resolve all undefined symbols at dlopen time

Delayed Binding to DLLs



(delayed 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

Asynchronous Completions

- some things are worth waiting for
  - when I read(), I want to wait for the data
- sometimes 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

Asynchronous Exceptions/Events

- some errors are routine
  - end of file, arithmetic overflow, conversion error
  - we should check for these after each operation
- some errors cannot be "checked for"
  - segmentation fault (illegal address)
  - user abort (^C), hang-up, power-failure
- these must raise *asynchronous exceptions*
  - some languages support try/catch operations
  - computers support traps
    operating systems use these for system calls

Hardware: Traps and Interrupts

- Used to get immediate attention from S/W
  - Traps: exceptions recognized by CPU
  - Interrupts: events generated by external devices
- The basic processes are very similar
  - program execution is preempted immediately
  - each trap/interrupt has a numeric code (0-n)
  - that is used to index into a table of PC/PS vectors
  - new PS is loaded from the selected vector
  - previous PS/PC are pushed on to the (new) stack
  - new PC is loaded from the selected vector

Traps vs. Interrupts

- Traps originate within the CPU
  - illegal instruction, invalid address, loss of power, …
  - each represents a critical one-time event
- Interrupts come from devices on I/O bus
  - I/O completion, device added/removed, …
  - they can be (temporarily) blocked/disabled
  - they may represent a continuing condition
  - may block acceptance of other similar interrupts
  - may not clear until they are acknowledged

Review (User vs. Supervisor mode)

- the OS executes in supervisor mode
  - able to perform I/O operations
  - able to execute privileged instructions
    - e.g. enable, disable and return from interrupts
  - able update memory management registers
    - to create and modify process address spaces
  - access data structures within the OS
- application programs execute in user mode
  - they can only execute normal instructions
  - they are restricted to the process's address space

Direct Execution

- Most operations have no security implications
  - arithmetic, logic, local flow control & data movement
- Most user-mode programs execute directly
  - CPU fetches, pipelines, and executes each instruction
  - this is very fast, and involves zero overhead
- A few operations do have security implications
  - h/w refuses to perform these in user-mode
  - these must be performed by the operating system
  - program must request service from the kernel

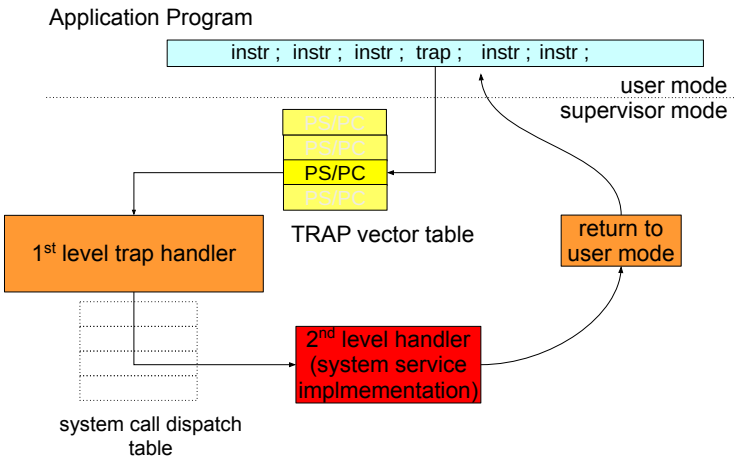
Limited Direct Execution

- CPU directly executes all application code
  - punctuated by occasional traps (for system calls)
  - with occasional timer interrupts (for time sharing)
- Maximizing direct execution is always the goal
  - for Linux user mode processes
  - for OS emulation (e.g., Windows on Linux)
  - for virtual machines
- Enter the OS as seldom as possible
  - get back to the application as quickly as possible

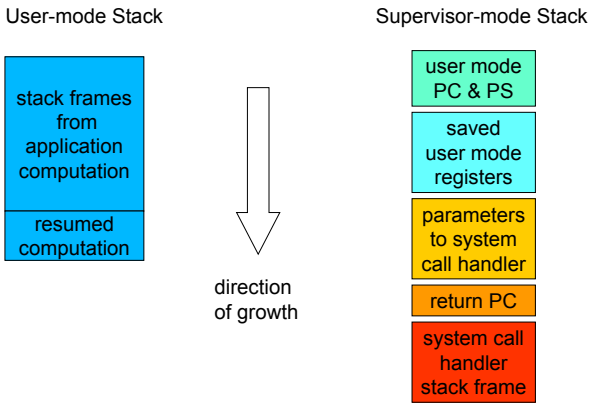
Using Traps for System Calls

- reserve one illegal 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, cc indicates success/failure
- prepare arguments for the desired system call
- execute the designated system call instruction
- OS recognizes & performs requested operation
- returns to instruction after the system call

System Call Trap Gates



Stacking and unstacking a System Call



(Trap Handling)

- hardware trap handling
  - trap cause as index into trap vector table for PC/PS
  - load new processor status word, switch to supv mode
  - push PC/PS of program that cuased trap onto stack
  - load PC (w/addr of 1st level handler)
- software 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 ...

(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

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

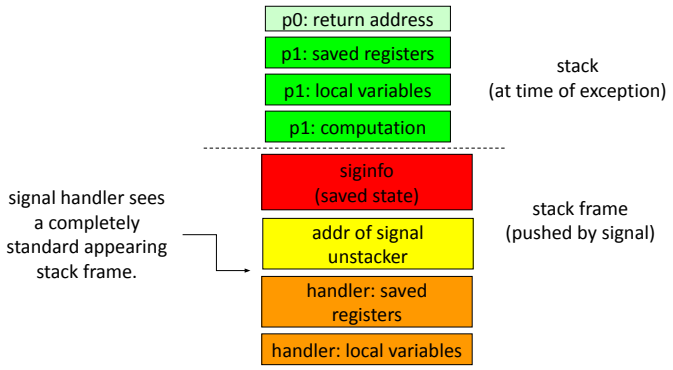
Signals and Signal Handling

- 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 whatever is necessary
  - handler can return, resume interrupted computation
- more complex than a procedure call and return
  - must also save/restore condition codes & volatile regs
  - may abort rather than return

Signals: 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;
```


Stacking a signal delivery (IA-64)



Reading and Assignments

Reading:

- Arpaci C7 … scheduling
- Arpaci C8 … adaptive scheduling
- Kampe: Real Time Scheduling

Projects:

- we will help w/project 1A problems in the lab

Supplementary Slides

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

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

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

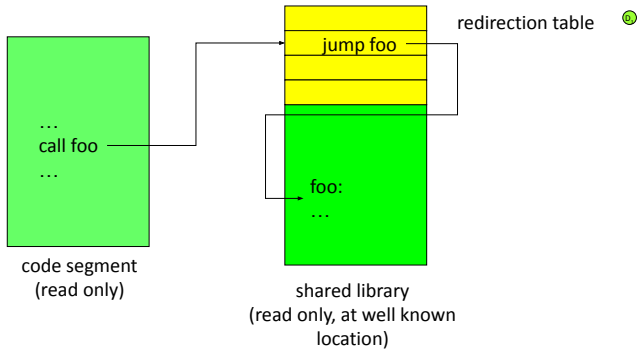
0x1020018 jmp foosub4

….

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

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