

## Exceptions and Dispatching

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

## 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, 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)

Resources, Services, and Interfaces

2

## 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, statefulness, side-effects
- All specifications should be explicit
  - no undocumented features
  - not defined by an implementation's behavior

Resources, Services, and Interfaces

3

## 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

Resources, Services, and Interfaces

4

## 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

Resources, Services, and Interfaces

5

## 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)

Resources, Services, and Interfaces

6

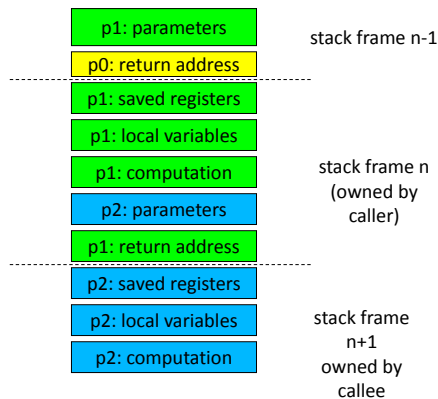
## Processes – stack frames

- 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	called routine
<pre> push p1 ; push first parameter push p2 ; push second parameter call foo ; save pc, call routine                     </pre>	<pre> 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                     </pre>
<pre> add =8,sp ; pop parameters ...                     </pre>	

## 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



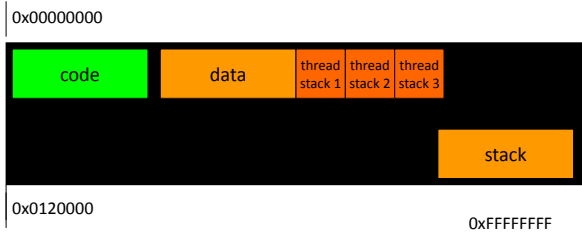
Data segment starts at page boundary after code segment  
 Stack segment starts at high end of address space  
 Unix extends stack automatically as program needs more.

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 reclaimed when thread exits
- procedure linkage conventions remain same

# Thread Stack Allocation



# Characteristics of Libraries

- Many advantages
  - 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

# 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



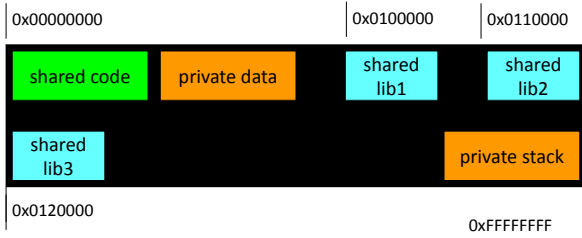
# Shared Libraries

- 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:

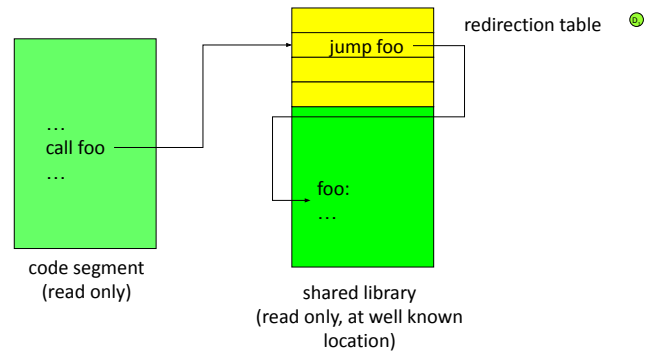
- 0: libfoo.so, shared library
- 1: foosub1, global, absolute, 0x1020000
- 2: foosub2, global, absolute, 0x1020008
- 3: foosub3, global, absolute, 0x1020010
- 4: foosub4, global, absolute, 0x1020018
- ...

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

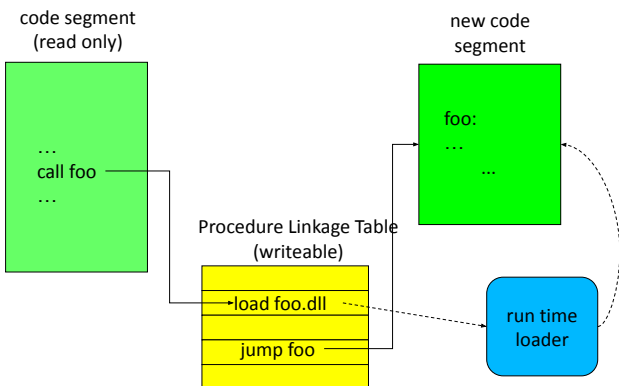
The real shared object is mapped into the process' address space at that fixed address. It begins with a jump table, that effectively seems to give each entry point a fixed address.

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



# 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

## 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

## 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

Processes, Execution, and State

26

## 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

Processes, Execution, and State

27

## 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

Processes, Execution, and State

28

## 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

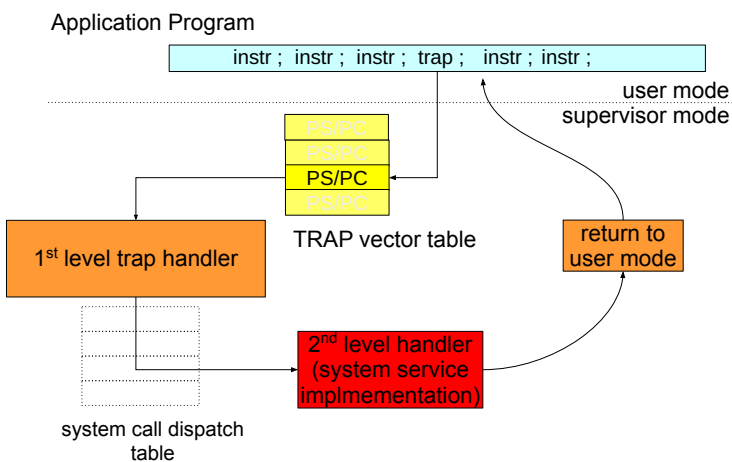
## Using Traps for System Calls

- 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

Processes, Execution, and State

32

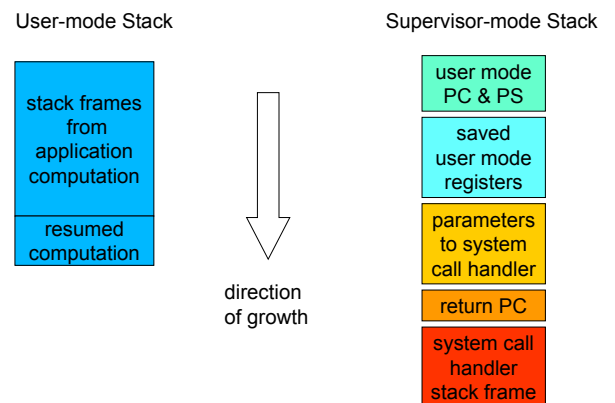
## System Call Trap Gates



Processes, Execution, and State

33

## Stacking and unstacking a System Call



Processes, Execution, and State

34

## (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 caused 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 ...

Processes, Execution, and State

35

## (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

Processes, Execution, and State

36

## 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

## 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

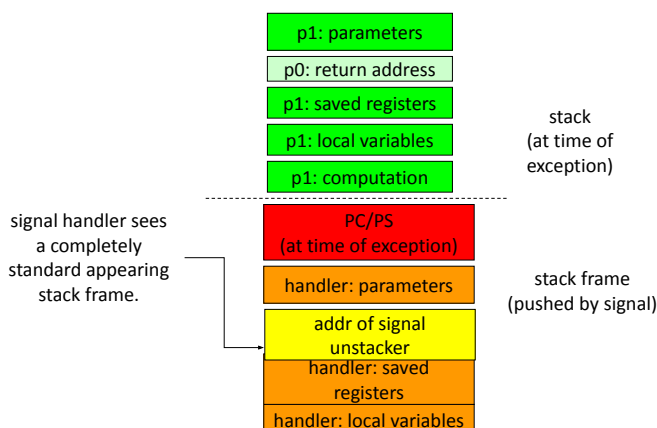
## Signals and Signal Handling

- 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



## 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

## 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