Object Modules, Linkage Editing, Libraries

Introduction

- Process is often defined as an executing instance of a program
- We often think of a program as one or more files (e.g. C, Java Python)
    - These are not executable programs, but rather source files that can be translated into machine language -> then creates executable program
- From the OS perspective, a program is a file full of ones and zeros, that when loaded into memory becomes machine language instructions that are combined with other machine language modules to create an executable program.

The Software Generation Tool Chain

- Limiting discussion to compiled languages
- source modules:
    - editable text in some language (C, assembly, Java), that can be translated into machine language using a compile/assembler
- relocatable object modules
    - sets of compiled or assembled instructions created from individual source modules, but which are not yet completed programs
- libraries
    - collections of object modules, from which we can fetch functions that are required by source/object modules
- load modules
    - complete programs (usually created by combining numerous object modules) that are ready to be loaded into memory and executed by the CPU

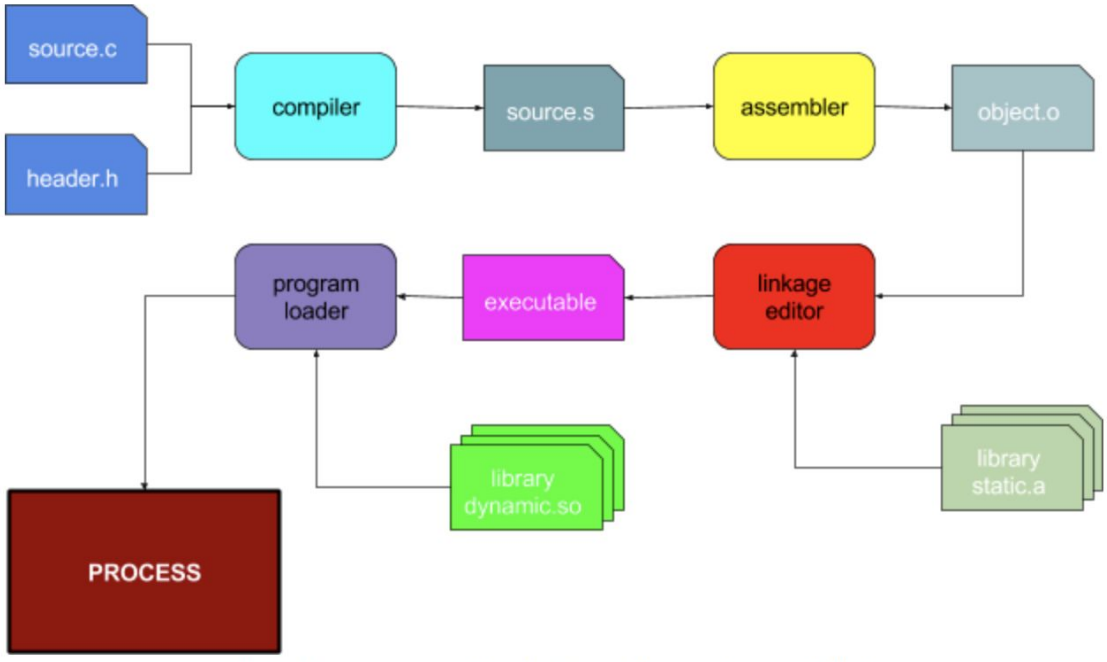


Fig 1. Components of the Software Generation Tool Chain

-

- rounded boxes represent software tools that are run and the rectangles with one corner cut off representing files used/created during process. Process is the final result
- Compiler
    - Reads source modules and included header files, parses the input language (e.g. C or Java) , and inferes the intended computations -> generate lower level code
        - The generated code will be produced in assembly language
        - Languages such as Java or Python's compilers directly produce a pseudo-machine language that will be executed by a virtual machine or interpreter
- Assembler
    - Assembly language is much lower level, with each line of code often translating directly to a single machine language instruction or data item.
    - In user-mode code, modules written in assembler often include
        - performance critical string and data structure manipulations
        - routines to implement calls into operating system
    - In the operating system, modules written in assembler often include
        - CPU initialization
        - first level trap/interrupt handlers
        - synchronization operations
    - **The output of the assembler is an object module** (sets of compiled/assembled instructions) containing mostly machine language code. Output only corresponds to a single input module
        - some functions (or data items) may not be present yet and therefore their addresses may not have been filled in
        - even locally defined symbols may not have yet been assigned hard in-memory addresses, and so may be expressed as offsets relative to some TBD starting point.
- Linkage Editor
    - The **linkage editor** reads a specified set of object modules (compiled/assembled sets of instructions), **placing them consecutively into a virtual address space**
    - Notes unresolved external references
    - **searches a specified set of libraries** to find object modules that can satisfy those references, and places them in the evolving virtual space
    - After locating and placing all of the required object modules, i**t finds every reference to a relocatable/external symbol and updates it to reflect the address where the desired code/data was actually placed**
    - Resulting bits represent a program that is ready to be loaded into memory and executed -> written to a file called a **executable load module**
    - static libraries are placed into the virtual address space using the linkage editor
- Program loader
    - The program loader is usually part of the OS

    - Examines the information in a load module, creates an appropriate virtual address space, and reads the instructions and initialized data values from the load module
    - Program loader also finds reference to dynamically loaded/shared libraries and maps them into the appropriate places in the virtual address space -> then the program can be executed by the CPU
- Summary: Compiler parses the inputted language into a lower level language (usually assembly, unless its an interpreted language like Python or Java). Then the assembler reads the assembly code where each line of code represents a machine language instruction, creating an object module. The assembler handles things like traps and interrupts, signals, CPU initialization, and synchronization in the OS. However, since it represents a single input file, it is incomplete and may have some data that has not yet been loaded. Then, the linkage editor loads the instructions into a virtual address space for the program. It also looks for dependencies from libraries (static) to find object modules that specify references. The linkage editor outputs a load module (executable) that is ready for execution. Then, the program loader reads the load module and executes the program. It also finds references to dynamically loaded/shared libraries and maps them into the proper virtual address space.

Object Modules

- When we write software, we do not put all of the code that will be executed into a single file
    - A single file containing everything would be massive, difficult to understand, and cumbersome to update
- Most programs are created combining multiple modules together. These program fragments are called relocatable object modules
    - They make references to code or data items that may be supplied in other libraries/modules
    - Has not determined where/how much address space is necessary, and so even references to code or items within the same module can be approximations relative to the address
- **The code (machine language instructions) within an object module created by the assembler are ISA specific.**
    - However, many contemporary object module formats are common across many ISA
        - **ELF (Executable and Linkable Format) in UNIX/Linux systems**
            - Header section: describes the types, sizes, and locations of other sections
            - Code and Data sections, each containing bytes that are to be loaded into memory
            - Symbol table: lists external symbols defined or needed by the module
            - A collection of relocation entries:

    - the location of a field (in a code or data section that requires relocation)
    - the width/type of the field to be relocated (32/64 bit addresses)
    - the symbol table entry, whose address should be used to perform that relocation
- *.o files represent object modules created by the assembler

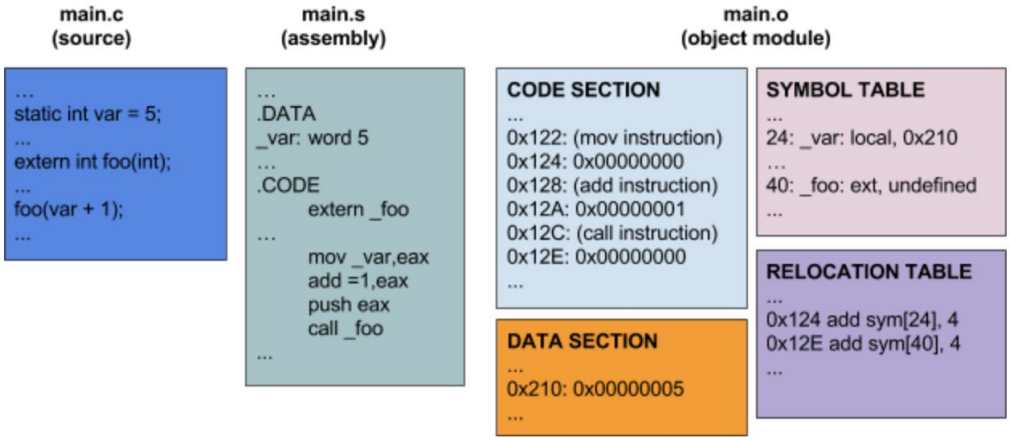


Fig 2. A Program in Various Stages of Preparation for Execution

- ■

Libraries

- In addition to its own modules, an interesting program can easily use hundreds, or even thousands of standard/reusable functions
    - A library is a simple collection of (usually related) object modules: also can be thought of as a collection of several subroutines within object modules
    - One library may contain standard system calls, while another might contain commonly used mathematical functions
    - The Linux command for creating, updating, and examining libraries is ar(1)
- Building a program usually starts by combining a group of enumerated object modules. However, there will most likely still be unresolved external references for sure
- Libraries are not always orthogonal and independent
    - It is common to implement higher level libraries (e.g. image file decoding) using functionality from lower level libraries (e.g. mathematical functions and file I/O)
    - It is not uncommon to use alternative implementations for some library functionality or to intercept calls to standard functions to collect usage data
- The order of which library is searched first can be very important
    - Ex: if we want to override the standard malloc(3) with valgrind's more powerful diagnostic version, we need to search the valgrind library before we search the standard C library.

Linkage Editing

- At least three things must be done to turn a relocatable object module into an executable program
    - Resolution: Search the specified libraries to find object modules that can satisfy all unresolved external references
    - Loading: Lay the text and data segments from all of those object modules down in a single virtual address space, and note where each symbol was placed
    - Relocation: Go through all of the relocation entries in all of the loaded object modules, each reference to correctly reflect the chosen addresses
- These operations are called linkage editing because we are filling in all of the linkages between the load modules. The UNIX command for performing linkage editing is ld(1)

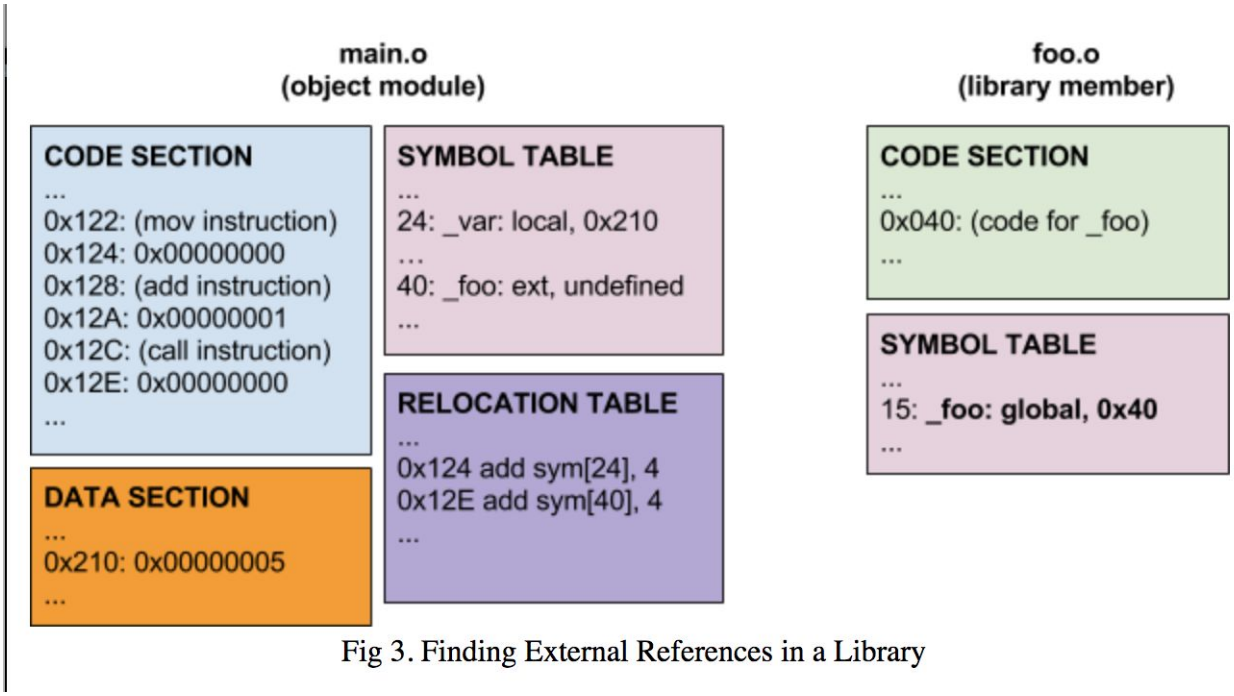


Fig 3. Finding External References in a Library

- Finding such a module, the linkage editor would then add it to the virtual space it was accumulating

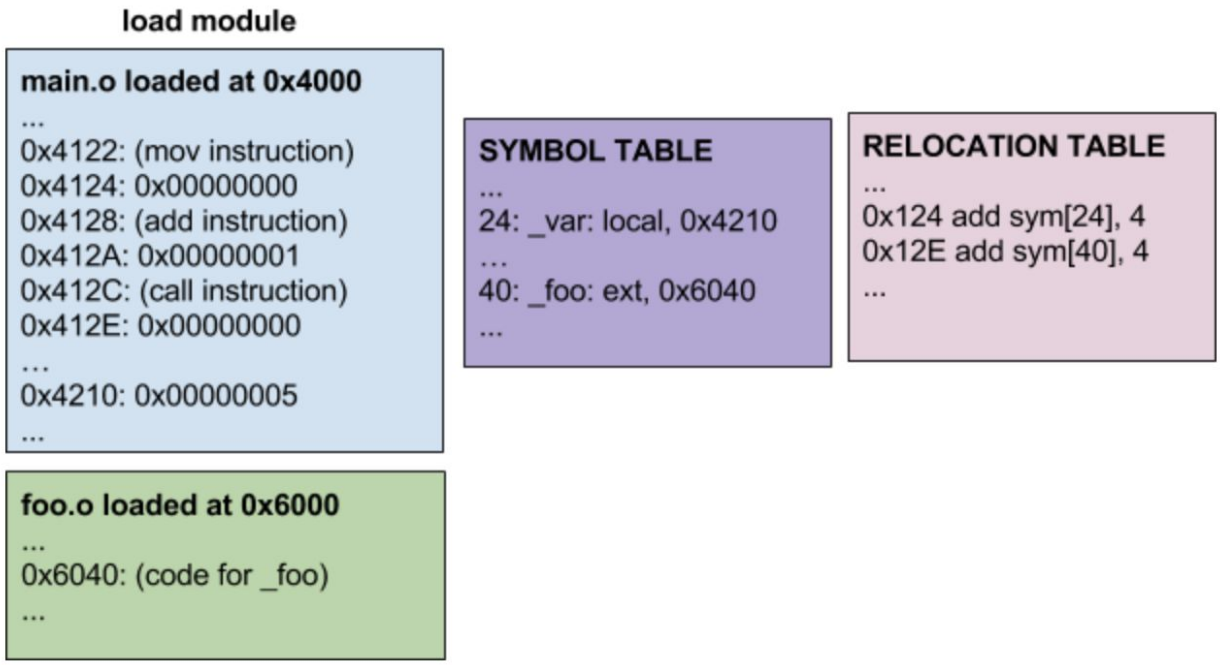


Fig 4. Updating a Process' Virtual Address Space

-
- Then, with all the unresolved external references satisfied, and all relocatable addresses fixed, the linkage editor would go back and perform all of the relocations call out in the object modules

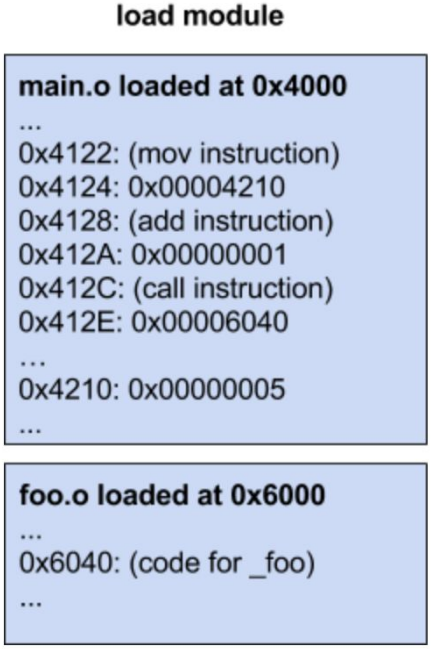


Fig 5. Performing a Relocation in an Load Module

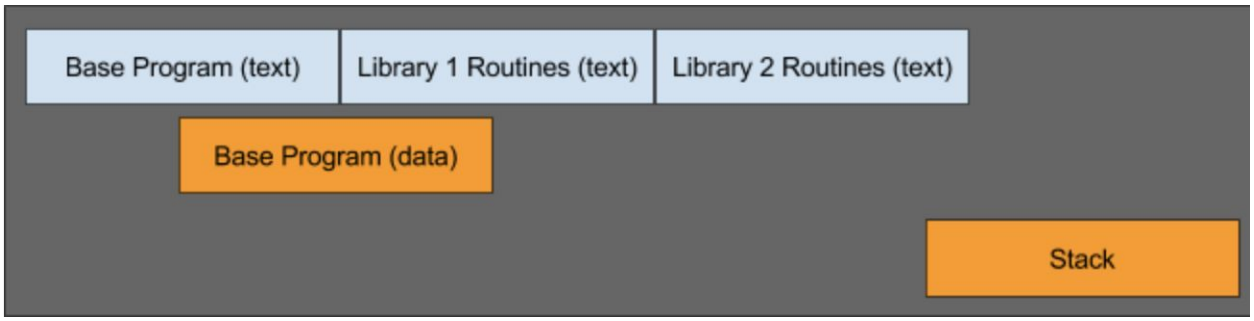
-

- At this point, all relocatable addresses have been adjusted to reflect the locations at which they were loaded, and all unresolved external references have been filled in. -> executable program is now complete (load module)

Load Modules

- A load module is in a similar format to an object module
- Unlike object module, the load module is complete and requires no relocation
    - Each section specifies the address (in the process' virtual address space) at which it should be loaded.
- When the OS is instructed to load a new program (ex: with the exec() system call)
    - It consults the load module to determine the required text and data segments sizes and locations
    - Allocates the appropriate segments within the virtual address space
    - Reads the contents of the text and data segments from the load module into memory
    - Creates a stack segment, and initializes the stack pointer to point to it
- **Why do we need a symbol table if there is no relocation required in a load module**
    - When there is an exception thrown/received, the symbol table would enable us to determine the error occured
    - Many load modules have no symbol tables while some have very extensive symbol tables
    - Symbol table: lists external symbols defined/needed by the module. Contains a symbol table entry whose address is used to perform relocation

Static vs. Shared Libraries



- Library modules were directly (and permanently) incorporated into the load module
- Because of this permanence, this process is referred to as static linking.
    - Disadvantages
        - Many libraries are used by almost every program on the system. Thousands of identical copies of the same code increase the required download time, disk space, start-up time, and memory. Would be more efficient to allow all programs that use a popular library to share a single copy

        - Popular libraries change over time with enhancements and optimizations. With static linking, each version of the library is frozen, so you are unable to update to new versions of the library
    - These issues are addressed with run-time loadable shared libraries
    - *STEPS TO IMPLEMENT SHARED LIBRARIES
        - Reserve an address for each shared library. (Possible for both 32-bit and 64-bit architectures)
        - Linkage edit each shared library into a read-only code segment, loaded at the address reserved for that library.
        - Assign a number to each routine, and put a redirection table at the beginning of that shared segment, containing the address (to be filled by the linkage editor) of each routine in the shared library
        - Create a stub library, that defines symbols for each entry point in the shared library, but implements each as a branch through the appropriate entry in the redirection table.
            - The linker leaves some stubs, or unresolved symbols, to be filled at application time
        - Linkage edit the client program with the stub library
        - When the OS loads the program into memory, it will notice that the program requires shared libraries and will open the associated (shareable, read-only) code segments and map them into the new program's address space at the appropriate location: ld.so(8)

Fig 7. Virtual Address Space with Shared Libraries

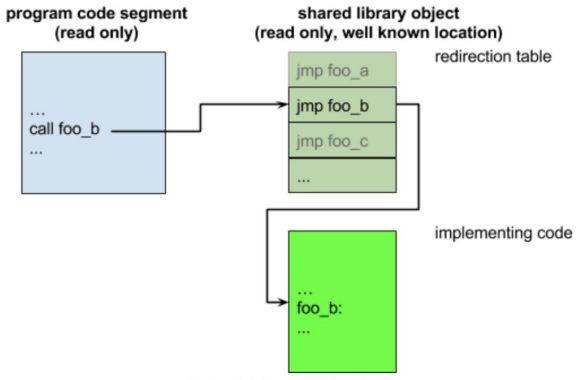


Fig 8. Linking Shared Libraries

    -
    - A single copy of a shared library implementation can be shared among all of the programs that use that library
    - The version of the shared segment that gets mapped into the process address space is chosen, not during linkage editing, but at program load time

    - Because all calls between the client program and the shared library are vectored through a table, client programs are not affected by changes in the sizes of library routines
    - With correct coding of the stub modules, it is possible for one shared library to make calls into another
        - Shared segment contains only read-only code.
        - The shared segment will not be linkage edited against the client program, and cannot make any static calls or reference global variables to/in the client program
    - Routines to be included in shared libraries must be designed with these limitations.
- SUMMARY: Reserve an address for each shared library and linkage edit it so that it produces a read-only code segment, loaded at the address reserved for the library. Create a stub library with the routines needed from each library, defining the symbols for every entry point in the shared library. Linkage edit the stub library with the client program. When the program loader sees that the client needs a shared library, it will load the routines/ code segments and map them into the new program's address space so that they can be used.
- Advantage over static linking: We will always get the newest version of the library that we want to use. We also save lots of space and time through only needing one shared copy.
- However, we still need to know the name of the library during the linkage editing process, meaning that the library is always going to be loaded into memory -- the routines that it calls from the library are the only thing that gets delayed, not the actual loading of the library (fixed through DLLs)

Dynamically Loaded Libraries

- Shared libraries are very powerful and convenient but they have proved to be too limiting for many applications
    - There may be large/expensive libraries that aren't used too much that may take up address space at program load time.
    - While loading is delayed until program load time, the name of the library to be loaded must be known at linkage editing time.
- Dynamically Loadable Libraries (DLLs): Not loaded (or even chosen) until they are actually needed
    - The application chooses a library to be loaded (based on some run-time information like the MIME type in a message)
    - The application asks the OS to load the library into its address space
    - The OS returns addresses of a few standard entry points (e.g. initialization and address space)
    - The application calls the supplied initialization entry point, and the application and DLL bind to each other

        - creates session data structures, exchanging vectors of service entry points
    - The application requests services from the DLL by making calls through the dynamically established vector of service entry points. When the application does not need the DLL anymore, it calls the shut-down method and tells the OS to unload the module
- Linux support for DLL is described in dlopen(3), which is used in user mode as well as inside the OS
    - makes it possible for pre-compiled applications to exploit plugins to support functionality that was no implemented
- Calls from the client application into a shared or DLL are generally handled through a table of vector of entry points
    - An application can register a call-back routine by passing its address to an appropriate library registration method
    - The same approach can be generalized to a large number of functions by having the application call the library register to a vector of entry points
    - **Dynamically loaded kernel modules (like device drivers and file systems)** are allowed to access a myriad of functions and data structures within the kernel into which they have been loaded.
        - This is often enabled by a run-time loader that effectively linkage edits the newly loaded module against the OS
            - fills the addresses of all unresolved external references from the dynamically loaded module into the OS
        - Linux kernel run-time loader: ld.so(8)

Implicitly Loaded Dynamically Loadable Libraries

- There are DLL implementations that are more similar to shared libraries
    - Applications are linkage edited against a set of stubs (unresolved symbols), which create Program Linkage Table (PLT) entries in the client load module
    - The PLT entries are intialized to point to calls to a Run-Time loader
    - The first time one of these entry points is called, the stub calls the Run-Time Loader to open and load the appropriate DLL
    - After the required library has been loaded, the PLT entry is changed to point to appropriate routine in the newly loaded library
- These implicitly loaded DLLs are almost indistinguishable from statically loaded libraries
- The greater functionality and performance benefits of DLLs are only available when the client applications become aware of them
- **Shared Libraries vs Dynamically Loaded Libraries**
    - Shared libraries allow delayed binding to some version of a library that was chosen at linkage-edit time.
    - DLLs allow the library to be loaded to be chosen at run time
    - Shared libraries consume memory for the entire time that the process is running
    - DLL can be unloaded when they are no longer needed

    - Shared libraries impose numerous constraints and simple interfaces on the code they contain
    - DLLs are capable of performing complex initialization and support bi-directional calls between the client and library
    - Shared libraries are almost (from client's perspective) indistinguishable from static libraries to use
    - Dynamically loaded libraries are more difficult to use
- Shared libraries are a more efficient mechanism for binding libraries to client applications.
- DLLs are a mechanism to dynamically extend the functionality of a client application based on resources and information that may not be available until the moment they are needed

DLL vs Shared (Reiterated)

- Shared Libraries must know the name of the library at linkage editing time, and only the routines that are called are delayed till load time. However, with DLL's the library itself is loaded only when it is needed. Additionally, shared libraries cannot hold static data and is read into memory whether or not you like it. Symbols known at compile time, bound at link time
- Called routines in a shared library must be known at compile time
- Shared libraries do not require a special linkage editor and also the shared objects are known at program load time
- DLL requires special linkage editor, and PLT (Procedure Linkage Table) that contains a system call to a run time loader. This run-time loader then chooses the library you need and the routines you need. Then it changes the PLT entry to be a jump to a loaded routine

Stack Frames and Linkage Conventions

The Stack Model of Programming Languages

- Modern programming languages support procedure-local variables
    - they are automatically allocated whenever the procedure is entered
    - they are only visible to code within that block
    - Each distinct invocation of the procedure has its own distinct set of local variables
    - They are automatically deallocated when the procedure exits/returns
- The local variables are usually stored ina last-in-first-out stack, with new call frames being pushed onto the stack whenever a procedure is called and old frames being popped off the stack whenever a procedure returns
- ex: recursive function factorial
    - if you call factorial 4 times before it is returned, then you would have 4 instances of the factorial function on the stack

Subroutine Linkage Conventions

- The details of linkage conventions are highly ISA specific, sometimes language specific

- x86 architecture example
- Basic elements of subroutine linkage
    - parameter passing: marshaling the information that will be passed as parameters to the called routine
    - subroutine call: save the return address on the stack, and transfer control to the entry point
    - register saving: saving the contents of registers that the linkage conventions declare to be non-volatile, so that they can be restored when the called routine returns
    - allocate space for the local variables in the called routine
- Process of returning when the routine is finished
    - return value: placing the return value in the place where the calling routine expects it
    - popping the local storage off the stack
    - register storing: restore the non-volatile registers to the values they had when the call routine was entered
    - subroutine return: transfer control to the return address that the calling routine saved at the beginning of the call
- Quick summary: Basic Elements of subroutine linkage: get the information that will be passed in as parameters -> save the return address on the stack -> save the non-volatile contents onto registers so that they can be returned when the routine is finished -> allocate enough space for the local variables
- Quick Summary: Return when the routine finishes: place the return value in the place where the calling routine expects to find it -> pop off the local storage from the stack -> restore register values of non-volatile contents -> transfer control to the return address of the calling routine saved at the beginning

The corresponding x86 code (for the above `factorial` example illustrates the register conventions and respective responsibilities of the caller and callee. The code in green is executed by *callees*. The code in red is executed by *callers*.

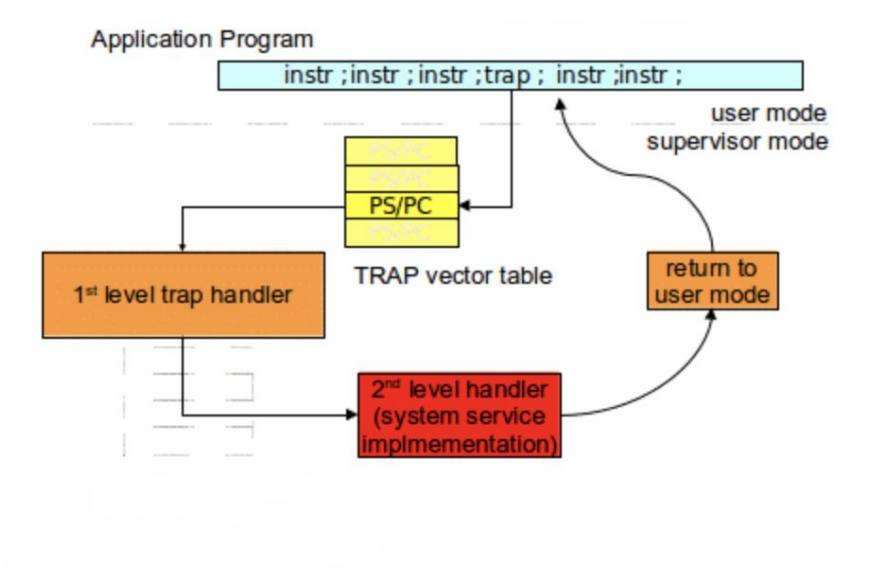
```
_factorial:

        pushl   %ebp            // save previous frame pointer
        movl    %esp, %ebp      // top of stack becomes new frame pointer
        pushl   %ebx            // save non-volatile EBX register

        movl    8(%ebp), %ebx   // copy parameter off stack into EBX
        movl    $1, %eax        // get a constant 1
        cmpl    $1, %ebx        // compare value with 1
        jle     L2              // if less than or equal ...
        leal    -1(%ebx), %eax  // EAX = EBX - 1

        subl    $12, %esp       // extend stack for recursive call
        pushl   %eax            // push parameter onto stack
        call    _factorial      // call factorial
        addl    $12, %esp       // clean off the call frame

        imull   %ebx, %eax      // multiply value by return value
    L2:                         // return value is in eax

        movl    -4(%ebp), %ebx  // restore saved EBX register
        movl    %ebp, %esp      // restore top of stack
        popl    %ebp            // restore saved frame pointer
        ret                     // return to caller
```

- caller: saves temporary local variables (volatile)
- callee: saves non-volatile long lived values that should be persisted
- X86 Register conventions
    - %esp is the hardware-defined stack pointer
    - the x86 stack grows downwards. A push operation causes the top of the stack to be the next lower address. Pop makes the address higher
    - %ebp is the frame pointer: it points at the start of the current stack frame
    - %eax is the volatile register that is expected to contain the return value before the caller's return address
    - parameters are pushed onto the stack immediately before the caller's return address
    - the call instruction pushes the address of the next instruction onto the stack, and then transfers control to the next location
    - the ret instruction pops the return address of the top of the stack
- Summary: Volatile variables -> local variables that are stored temporarily, non-volatile -> important variables that need to be persistent. Frame pointer points to the start of the current stack frame, while stack pointer points to the top of the stack.
- Register saving is the responsibility of the called routine.
- Cleaning parameters off of the stack is the responsibility of the calling routines. Only the calling routine knows how many parameters were actually passed
- **Another summary just to remember**!: Parameters are stored onto the stack by the caller. Then the return address of the previous state is stored and control is given to the caller. Non-volatile information is stored in registers so that when the routine finishes, the values can be saved and returned. Create enough space for the local variables on the stack. Place the return value where the calling function expects to find it. Restore values on the saved registers. Pop the local storage off the stack and give control back to where it was before the calling routine

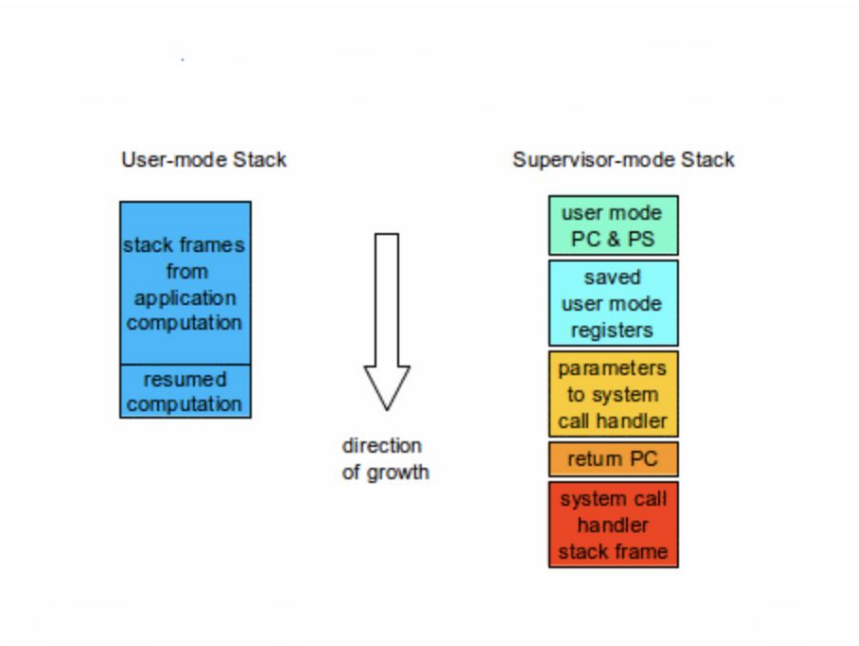
Traps and Interrupts

- Most ISA includes supports for **interrupts (to inform the software than an external event has happened) and traps (to inform the software of an execution fault)**
- **Similarities between procedure call and an interrupt**
    - we want to transfer control to an interrupt or trap handler
    - we need to save the state of the running computation before doing so
    - after the event has been handled, we want to restore the saved state and resume the interrupted computation
- **Differences between procedure call and an interrupt/trap**
    - A procedure call is requested by the running software, and the calling software expects that, upon return, some function will have been performed, and an appropriate value returned
    - because a procedure call is initiated by software, all of the linkage conventions are under software control. However, trap/interrupt handling is done by the hardware

    - The computer state should be restored as if the trap/interrupt never occured
- Typical interrupt or trap mechanism
    - there is a number associated with every possible external interrupt or execution exception
    - there is a table, initialized by the OS software that associates a Program Counter and Processor Status word with each possible external interrupt/etrap
    - When an event happens that would trigger an interrupt/trap
        - the CPU uses the associated interrupt/trap number to index into the appropriate interrupt/trap vector table
        - the CPU loads a new program counter and processor status word from the vector table
        - the CPU pushes the program counter and associated processor word onto the CPU stack
        - execution continues at the address specified by the new program counter
        - the selected code (usually written in the assembly, is called a first level handler)
            - saves all of the general registers on the stack
            - gathers information from the hardware on the cause of the interrupt/trap
            - chooses the appropriate second level handler
            - makes a normal procedure call to the second level handler, which actually deals with the interrupt/exception
        - after the second level handler has dealt with the event, it returns to the first level handler
            - the first level handler restores all of the saved registers
            - the first level handler executes a privileged return from interrupt or return from trap instruction
            - the CPU re-loads the program counter and processor status word from the values saved at the time of the interrupt/trap
            - execution resumes at the point of interruption



Flow-of-control associated with a (system call) trap



Stacking and un-stacking of a (system call) trap

-
- This interrupt/trap mechanism
  - does a more complete job of saving and restoring the state of the interrupted computation
  - translates the much simpler hardware driven call to the first level handler into a normal higher-level language procedure call to the chosen 2nd level handler
- traps and interrupts are highly expensive because a new processor status word is loaded into the CPU, in processor mode, running in a new address space, and surely involve a complete loss of CPU caches

Summary: Procedure and trap/interrupt linkage instructions provide us with

- a preliminary low level definition of the state of a computation and what it means to save and restore that state
- an intro to the basic CPU supported mechanisms for synchronous and asynchronous transfers of control
- an example of how it might be possible to interrupt an on-going computation, do other things, and then return to the interrupt computation as if it had never been interrupted

Readings: Virtualization
Chapter 3

- Peach example -- imagine that you have a peach. However, there are many people who want to eat this peach. In order to make people happy, we create multiple virtual peaches, where it looks to the eater as if they're eating the actual peach but they're not.
- CPU Example: Assume that there is one physical CPU in the system. What virtualization does is take that single CPU and make it look like many virtual CPUs to the application running the system

Chapter 4: The Abstraction: The Process

- One of the most fundamental abstractions that the OS provides to user is the process
- Informal definition: A running program -- the program itself is a lifeless thing, it just sits there on the disk, a bunch of instructions, waiting to spring into action
    - The operating system is the one that takes these bytes and makes them run properly
- How to provide the illusion of many CPU's?
    - The OS creates an illusion of many CPU's by virtualizing the CPU.
    - By running one process, then stopping it and running another, and so forth, the OS can promote the illusion that many programs are running at the same time
        - Creates illusion that there are several virtual CPUs even though there is only one physical CPU
        - This technique is known as **time sharing**: allows users to run as many concurrent processes as they would like
- Implementing Virtualization of the CPU
    - The OS needs to know low-level machinery as well as high-level intelligence
    - low-level machinery known as mechanisms
        - low-level methods or protocols that implement a needed functionality
        - example: Implement a context switch, a mechanism that gives the OS the ability to stop running one program and start running another.
    - high-level intelligence known as policies
        - policies are algorithms for making some kind of decision in the OS
        - example: Given a number of programs to run on a CPU, which program should the OS run?

- Tip: Use Time Sharing (and space sharing)
    - Allow the resource to be used for a little while by one entity, and then a little while by another.
    - Space sharing: Resource is divided among those who wish to use it
        - example: Disk space is naturally a space-shared resource
- 4.1 The Abstraction: A Process
    - The abstraction provided by the OS of a running program is something we call the process
        - A process is simply a running program
        - machine state: what a program can read or update when it is running. What parts of the machine are important to the execution of the program?
        - One component of the machine state is its memory
            - Instructions lie in memory; the data that the running program reads and writes sits in memory as well. **Thus the address space is part of the process**
        - Another component of the machine state are registers
            - many instructions explicitly read or update registers
            - Program counter or instruction pointer (IP) tells us which instruction of the program is currently being executed
            - stack pointer and associated frame pointer are used to manage the stack for function parameters, local variables, and return addresses
        - Programs also access persistent storage devices such as file systems
    - Tip: Separate Policy and Mechanism
        - Think of the mechanism as providing the answer to a how question about a system.
            - ex: How does an operating system perform a context switch?
        - The policy provides the answer to a which question
            - ex: Which process should the operating system run right now.
        - Separating the two allows one easily to change policies without having to rethink the mechanism and is thus a form of **modularity**
- 4.2: Process API
    - These are examples of what may be included in a real process API
    - Create
    - Destroy
    - Wait
    - Miscellaneous Control: e.g., providing a method to suspend a process
    - Status

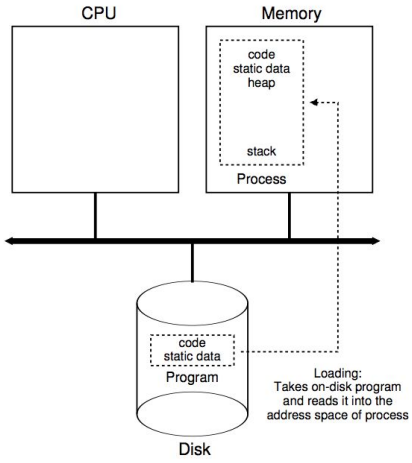


Figure 4.1: **Loading: From Program To Process**

- 4.3 Process Creation: A Little More Detail
    - How are programs transformed into process? How does the OS get a program up and running?
    - 1) The OS must load its code and any static data (e.g., initialized variables) into memory, **into the address space of the process**.
        - Programs initially reside on disk (or in some modern systems, flash-based SSDs) in some kind of executable format
        - The process of loading a program and static data into memory requires the OS to read those bytes from disk and place them in memory
        - Early OS did the loading process eagerly, reading all at once before running the program
        - Modern OS load perform the process lazily; loading pieces of code or data only as they are needed during program execution
    - 2) Some memory must be allocated for the program's run-time stack
        - C-programs use the stack for local variables, function parameters, and return addresses
        - The OS will also likely initialize the stack with arguments. Specifically the arguments of argc and argv in the main function
    - 3) OS may allocate some memory for the heap
        - In C programs, the heap is used for explicitly requested dynamically allocated data
        - Request space in the heap using malloc() and free it using free()
        - The heap is needed for data structures such as linked lists, hash tables, trees, etc.
    - 4) OS will do other initialization tasks as related to I/O
        - For example, in UNIX systems, each process by default has three open file descriptors.
        - These descriptors let programs easily read input from the terminal as well as print output to screen

    - After the OS has loaded the code and static data, initialized the stack, allocated memory for the heap/did initializations related to I/O, the program becomes ready for execution.
    - Program begins its execution through looking for the main() routine, and then the OS transfers control of the CPU to the newly-created process
- 4.3 Process States
    - Running: In the running state, a process is running on a processor. This means that is is already executing instructions
    - Ready: The process is ready to run but for some reason the OS has chosen not to run it
    - Blocked: A process has performed some kind of operation that makes it not ready to run until some other event takes place.
        - Example: When a process initiates an I/O request to a disk, it becomes blocked and thus some other process can use the processor

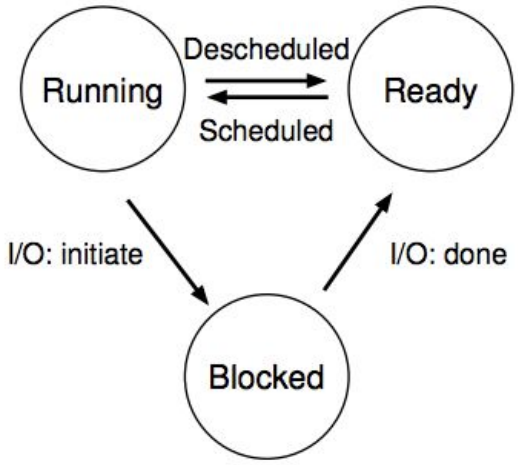


Figure 4.2: **Process: State Transitions**

    -
    - Being moved from ready to running means that the process has been **scheduled**
    - Being moved from running to ready means that the process has been **descheduled**
    - Once a process has become blocked, the OS will keep it as such until another event occurs and the process then returns to the ready state
    - The OS has to make critical decisions on when to run which programs at what times
- 4.5 Data Structures
    - Key Idea: The OS is a program, and like any other program, it would probably still like to keep a set of data structures to hold information

- ex: the OS will likely keep some sort of process list for all processes that are ready, as well as some additional information to track which process is currently running

```
// the registers xv6 will save and restore
// to stop and subsequently restart a process
struct context {
  int eip;
  int esp;
  int ebx;
  int ecx;
  int edx;
  int esi;
  int edi;
  int ebp;
};

// the different states a process can be in
enum proc_state { UNUSED, EMBRYO, SLEEPING,
                  RUNNABLE, RUNNING, ZOMBIE };

// the information xv6 tracks about each process
// including its register context and state
struct proc {
  char *mem;                  // Start of process memory
  uint sz;                    // Size of process memory
  char *kstack;               // Bottom of kernel stack
                              // for this process
  enum proc_state state;      // Process state
  int pid;                    // Process ID
  struct proc *parent;        // Parent process
  void *chan;                 // If non-zero, sleeping on chan
  int killed;                 // If non-zero, have been killed
  struct file *ofile[NOFILE]; // Open files
  struct inode *cwd;          // Current directory
  struct context context;     // Switch here to run process
  struct trapframe *tf;       // Trap frame for the
                              // current interrupt
};
```

Figure 4.5: **The xv6 Proc Structure**

-
- The register context will hold, for a stopped process, the contents of its registers. When a process is stopped, its registers will be saved to this memory location;
  - The OS can resume running the process afterwards -> context switches
- Sometimes, the system will have an initial state that the process is in when it is being created
  - Also, a process could be placed in a final state where it has exited but has not yet cleaned up. In UNIX systems, this is known as a Zombie state

- Summary: Remember the lower-level mechanisms needed to implement processes, and the higher-level policies to schedule them in an intelligent way
- *Mechanisms: ask the question how? How does an operating system perform a context switch? (e.g. CPU, hardware, memory, disks)
- Policies: ask the question which? Which process should the OS use right now
- *Remember mechanism/policy separation from lecture 1: The mechanisms/implementations should not overly constrain the policies/applications of an operating system.

Chapter 5: Interlude: Process API

- UNIX systems use a pair of system calls to create processes, fork() and exec(). A third routine wait() can be used by a process wishing to wait for a process it has created to complete
- 5.1: The fork() system call
    - One of the strangest routines you will ever call
    - In UNIX systems, the process identifier, or the PID, is used to name the process if one wants to do something with the process, such as stop it from running
    - The process that is created using the fork() function is almost an exact copy of the calling process
        - ex: This means that, to the OS, it looks like there are two copies of the same program running, and both are about to return from the fork() system call.
        - This newly created process is called the child, but doesn't start running at main() like you might think.
        - Rather it comes into life as if it had called fork() itself
        - The child get its own copy of the address space (i.e. its own private memory), its own registers, its own PC, etc. The value it returns to the caller fork() is different.
        - Specifically, while the parent receives the PID of the newly-created child, the child receives a return code of zero.
        - Now a program may have two processes, a parent and a child. For a single CPU computer, either the parent or the child may run at that point
    - The CPU scheduler determines which process runs at which time
- 5.2: The wait() System call
    - Sometimes, it is useful for the parent to wait for the child process to finish
    - If the parent process calls wait() to delay execution until the child finishes executing, the output becomes deterministic
        - the child process will always finish first, even if the parent process is running on a single CPU system
- 5.3: The exec() System call
    - This routine is useful for when you want to run a program that is different from the calling program
    - What the exec() system call does
        - Given the name of an executable and some arguments, it loads code from that executable and overwrites its current code segment with it. The heap, the stack, etc are all re-initialized
        - Thus, it does not create a new process. Rather, it transforms the currently running process into a new one
- 5.4: Why? Motivating the API
    - Lets the shell run code after the call to fork() and before the call to exec()
    - The shell is just a user program that shows you a prompt and then waits for you to type something into it

        - The shell firsts finds where the executable resides, then creates a new child process by calling fork() then calls some variant of exec() to run the command, and then waits for the command to complete by calling wait(). When the child completes, the shell returns from wait() and prints out a prompt again.
        - This separation of fork() and exec(0 let the shell do a bunch of useful things
            - example: prompt > wc p3.c > newfile.txt (redirection)
            - when the child is created (fork()), and before calling exec, the shell closes std out and opens the file newfile.txt. By doing this, all output is sent to the file instead of the screen
        - Unix pipes are implemented in a similar way, but with the pipe() system call instead
            - the output of one process is connected to an in-kernel pipe (i.e. queue), and the input of another process is connected to that same pipe. Thus, the output of the process is seamlessly used as the input of the next
- 5.5: Other Parts of the API
    - Beyond fork(), exec(), and wait()
        - kill(): system call used to send signals to a process, including directions to go to sleep, die, and other useful imperatives.

Chapter 6: Mechanism: Limited Direct Execution (Achieving Virtualization)

- How can we implement virtualization without making it too expensive?
- How can we run processes efficiently while retaining control over the CPU?

6.1: Basic Technique: Limited Direct Execution

- The “direct execution”
    - run the program directly on the CPU. Thus, when the OS wishes to start a program running, it creates a process entry for it in the process list, allocates some memory for it, loads the program code into memory, and starts running the code
    - How can the OS make sure that we don’t do something that we don’t want to do?
    - How does the OS switch from one process to another, thus implementing the time sharing necessary to do virtualization

6.2: Problem #1: Restricted Operations

- Direct execution has the obvious advantage of being fast, but what if the program wants to run some sort of privileged I/O operation or gain access to CPU or memory?
- Aside: Why system calls look like procedure calls?
    - A system call is a procedure call, except there is a trap instruction that is hidden
    - Ex: When you call open(), you are executing a procedure call in the C library. Therein, whether for open() or any of the other system calls provided, the library uses an agreed-upon calling convention with the kernel (privileged functions) to

put the argument in well-known locations such as the stack or registers, puts the system call number in a well-known location, and then executes the trap instruction
    - After the trap instruction, the code in the library unpacks return values and returns control to the program that issued the system call.
- Introduce a new processor mode! User mode
    - code that runs in user mode is restricted in what it can do
    - a process running in user mode cannot issue I/O requests or access memory
- Kernel Mode
    - the OS/kernel runs in this mode. Can run privileged instructions such as I/O requests
- System calls
    - enables user code to call kernel/privileged instructions to make I/O requests, etc.
    - system calls allow the kernel to carefully expose certain key pieces of functionality to users, such as accessing the file system, creating and destroying processes, etc.

MECHANISM: LIMITED DIRECT EXECUTION

OS @ boot (kernel mode)	Hardware	
initialize trap table		
	remember address of... syscall handler	

OS @ run (kernel mode)	Hardware	Program (user mode)
Create entry for process list		
Allocate memory for program		
Load program into memory		
Setup user stack with argv		
Fill kernel stack with reg/PC		
return-from-trap		
	restore regs from kernel stack	
	move to user mode	
	jump to main	
		Run main()
		...
		Call system call
		trap into OS
	save regs to kernel stack	
	move to kernel mode	
	jump to trap handler	
Handle trap		
Do work of syscall		
return-from-trap		
	restore regs from kernel stack	
	move to user mode	
	jump to PC after trap	
		...
		return from main
		trap (via `exit()`)
Free memory of process		
Remove from process list		

Figure 6.2: **Limited Direct Execution Protocol**

- when you make system calls, traps are executed. It saves the register values from kernel stack, moves to kernel mode, and jumps to the trap handler.

- Two phases in the Limited Direct Execution
    - First, at boot time, the kernel initializes the trap table, and the CPU remembers its location for subsequent use. Kernel does this through privileged instructions
    - Second, the kernel sets up a few things such as allocating a node on the process list or allocating memory, before using a return-from-trap instruction to start the execution of the process
        - this switches CPU to user mode and begins running the process
        - At boot, the system is in kernel mode -> initializes elements such as memory and process list, then returns from trap in order to go back to user mode

6.3: Problem #2: Switching between Processes

- **If a process is running on the CPU, the OS is not running!**
- How can operating systems regain control of the CPU so that it can switch between processors?
- A Cooperative Approach: Wait for system calls (Macintosh OS M11)
    - the OS trusts the processes of the system to behave reasonably. Processes that run for too long are assumed to periodically give up the CPU so that the OS can decide to run some other task
    - Most systems have explicit yield() calls that transfer control to the OS so it can decide what process to run next
    - If an application tries to do something illegal such as divide by zero or try to access private memory, it will generate a trap to the OS. The OS will then have control of the CPU again -> likely terminate offending process
    - Passive approach: waits for process to finish/system call or for an illegal operation to occur
- A non-cooperative approach
    - Without some additional help from the hardware, the OS can't do much at all when a process refuses to make system calls (or mistakes)
    - If your process gets stuck in an infinite loop, your only course of action would be do to reboot the machine
    - The solutions do these problems was a mechanism called a **timer interrupt** to regain control
        - **A timer device can be programmed to raise an interrupt every so many milliseconds;** when the interrupt is raised, the currently running process is halted and a preconfigured interrupt handler in the OS runs.
        - The OS must inform the hardware of which code to run when the timer interrupt occurs
            - The OS does that at boot time
            - The OS must also start the timer during boot time, which is a privileged operation of course

        - The hardware called by the interrupt must the responsibilities of saving and storing the data somewhere, so that when it returns, it returns the correct information
- Saving and Restoring Data
    - Now that the OS has regained control, whether cooperatively via a system call, or more forcefully via a timer interrupt, a decision has to be made
        - continue running the currently-running process, or switch to another
        - This decision is made by a **scheduler**
    - If the decision is made to switch, the OS executes a low-level piece of code called the context switch
        - saves a few register values for the currently-executing process (onto its kernel stack)
    - To save the context of the currently running process, the OS will execute some low-level assembly code to save the general purpose registers, Program Counter, as well as the kernel stack pointer of the currently-running process, and then restore said registers, PC, and switch to the kernel stack of the executing process.
    - By switching stacks, the kernel enters the call to the stack as if it were one process and returns in the context of another

OS @ boot (kernel mode)	Hardware	
initialize trap table		
	remember addresses of...	
	syscall handler	
	timer handler	
start interrupt timer		
	start timer	
	interrupt CPU in X ms	

OS @ run (kernel mode)	Hardware	Program (user mode)
		Process A
		...
	timer interrupt	
	save regs(A) to k-stack(A)	
	move to kernel mode	
	jump to trap handler	
Handle the trap		
Call `switch()` routine		
save regs(A) to proc-struct(A)		
restore regs(B) from proc-struct(B)		
switch to k-stack(B)		
return-from-trap (into B)		
	restore regs(B) from k-stack(B)	
	move to user mode	
	jump to B's PC	
		Process B
		...

Figure 6.3: **Limited Direct Execution Protocol (Timer Interrupt)**

-
- Switching from process A to B:

    - When the OS decides to switch from process A to B, it carefully saves the register values in A, restores the registers of B (from its process structure entry) and then switches contexts

6.4: Worried About Concurrency?

- What happens when, during a system call, a timer interrupt occurs?
- What happens when you're handling one interrupt and another one happens?
- Does that get hard to handle in the kernel?
    - OS may disable interrupts during interrupt processing; doing so ensures that when one interrupt is being handled another is not being called to the OS

What is a state?

- Dictionary Definition: "A mode or condition of being"
- 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 state
    - we can talk of a subset such as scheduling state

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!

Stub Modules vs Real Shared Libraries

- Stub modules
    - program is linkage edited against a stub module, and so believes each of the contained routines to be at a fixed address
    - really is just a table of addresses that you should call (table of entry points)
- Real Shared Libraries
    - shared libraries have hard-reserved addresses
    - 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
    - Make the vector bigger than it needs to be

    - vector redirection table for the shared libraries makes a bunch of jump call to redirect to a certain address

Asynchronous Events

- 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 of call-backs
    - This is a common programming paradigm
    - computers support interrupts
    - commonly associated with I/O devices and timers

Asynchronous Exceptions

- Some errors are routine
    - end of file, arithmetic overflow, conversion error
    - we should check for these after each operation
- some errors occur unpredictably
    - segmentation fault (e.g. dereferencing NULL)
    - user short, hang up, power failure
- these must raise async. exceptions
    - some languages support try/catch operations
    - computer support traps
    - OS also uses 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
    - 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