Exam 4 Notes

Linking

Linking is a fundamental process in software development that combines various pieces of compiled code into a single executable program. In UNIX-like systems, linking can be either static or dynamic:

- **Dynamic Linking**: Libraries are linked at runtime, which allows multiple programs to share the code of a single library file on disk, reducing overall memory usage.
- **Static Linking**: The linker combines all the necessary library routines and modules into a single executable file at compile time.

Symbol Resolution In Linking

During the linking process, the linker must resolve references to symbols (variables, functions, etc.) that are defined in one module but used in another. This resolution can become complex when multiple definitions of the same symbol are found across different modules. Here, the concepts of strong and weak symbols are crucial.

Strong Symbols

- **Definition**: Strong symbols are typically defined as functions and initialized global variables. The linker uses strong symbols to resolve ambiguities when the same symbol appears multiple times.
- **Characteristics**:
  - Have higher precedence during symbol resolution.
  - If there are multiple strong symbols of the same name across different modules, it leads to a linking error, as each symbol is assumed to represent different entities.

Weak Symbols

- **Definition**: Weak symbols are generally used for uninitialized global variables and provide a way for the linker to resolve symbols when multiple definitions occur without causing errors.
- **Characteristics**:
  - If a weak symbol and a strong symbol of the same name exist, the strong symbol is preferred, and no error is raised.
  - If multiple weak symbols of the same name exist, any of them might be chosen by the linker, and the choice does not result in a linking error.

Rules For Symbol Resolution

Given these definitions, UNIX linkers typically follow a set of rules to handle duplicate symbols:

- **Any Weak If Only Weak Present**: If multiple weak symbols with the same name exist and no strong symbol is present, the linker arbitrarily chooses one. This flexibility is useful for linking against large libraries where weak symbols can act as placeholders or defaults.
- **No Multiple Strong Symbols**: If more than one definition of a strong symbol is found, it results in a linker error. This ensures that each strong symbol is unique.
- **Strong Over Weak**: If both strong and weak symbols with the same name exist, the strong symbol takes precedence. This rule allows for optional features in libraries where a weak symbol provides a default implementation, and a strong symbol in the user's code provides a custom implementation.

Symbol Tables

Symbol tables are a crucial aspect of the compilation process in programming. They serve as a repository where information about the identifiers (symbols) used in a program is stored. These identifiers can be variable names, function names, constants, and data types. The symbol table is used by the compiler and linker to ensure that all symbols are correctly identified and accessible.

Functions Of Symbol Tables

- **Scope Management**: They help in managing the scope of variables. Local variables in different functions can use the same name without conflict because the symbol table will treat them as separate entries.
- **Storage Of Symbol Information**: Symbol tables store details such as the name of the symbol, its type, scope (local or global), and sometimes its memory location.
- **Type Checking**: Symbol tables are used during semantic analysis by compilers to check for type inconsistencies in operations.

Components Of Symbol Tables

- **Symbol**: The variable (symbol) that is present in one of source files.
- **Entry**: A boolean value that indicates if a symbol is present in the table.
- **Type**: The type of symbol that is / isn't present in the table: `static`, `extern`, `global`, `local`.
- **Location (Module)**: Where the symbol is defined in reference to the source files.
- **Section**: Location of where the symbol is stored in compilation.

Types

Symbols are categorized into main types:

- `extern`: The `extern` keyword is used to declare a variable or function and indicates that its definition is in another file or translation unit.
  - **Usage**: `extern` is primarily used when you need to access a variable or function defined in another source file or to declare the variable in a header file that multiple source files include.
  - **Characteristics**:
    - Does not allocate memory by itself.
    - Requires an external definition with a matching type.
    - Useful in managing global variables across different files.
- `global`: Global variables and functions are those defined outside any function (usually at the top of the source file) and can be accessed from any function in the program.
  - **Usage**: Global symbols are accessible throughout the program from any translation unit that includes a declaration of them. This is default for functions in C and C++.
  - **Characteristics**:
    - Stored usually in the global data segment.
    - Persistent for the lifetime of the application.
    - Can cause issues like name clashes and are generally discouraged in modern programming due to their impact on code maintainability and testing.
- `local`: Local variables are declared inside a function or block and can only be accessed within that function or block (scope-limited).

- **Usage**: Local variables are used to store temporary state or intermediate results within a function.
- **Characteristics**:
  - Stored on the stack (typically).
  - Automatically allocated and deallocated when the function is called and returns, respectively.
  - Not visible outside the function or block where they are declared.

- `static`: The `static` keyword can modify both local and global variables. It alters the storage duration of the variable it qualifies.
- **Usage**:
  - **Global Static**: When used outside any function, it restricts the scope of the variable to the file in which it is declared, making it a private global.
  - **Local Static**: When used within a function, the variable retains its value between function calls.
- **Characteristics**:
  - Local static variables are initialized only once, and they exist until the end of the program.
  - Global static variables are only accessible within the same translation unit (source file), protecting against namespace pollution.

Sections

For each symbol, there is a location where the symbol is stored upon compilation:

- **`.bss` (Block Started by Symbol)**: The section where uninitialized `static` variables, and `global` or `static` variables that are initialized to zero are stored. This section is used for declaring variables that are not initialized by the programmer. By default, the system initializes them to zero.
- `COMMON`: The section where uninitialized `global` variables are stored. This is a special section used in the context of weak linkage and tentative definitions. If a global variable is declared but not initialized (a common practice in C for external variables), it is typically placed in the `COMMON` section. This allows the linker to handle multiple tentative definitions.
- `.data`: The section where initialized `global` and `static` variables are stored. Both `global` and `static` variables in this context retain their values through execution.
- **`.rodata` (Read-Only Data)**: This section stores constant values and string literals that should not be modified, making them read-only at runtime.
- `.text`: This section contains the executable code of the program. It is where the machine instructions reside.

Exceptional Flow Control

Exceptional Control Flow (ECF) involves mechanisms that alter the normal sequential execution order of programs. ECF mechanisms include signals, process context switching, and system calls like `fork()`. These are essential for operating systems to perform efficient multitasking, handle asynchronous events, and manage multiple processes.

`fork()` System Call

One example of ECF is the `fork()` system call. This system call produces children from a parent process (and sometimes a child from an already existing child). The core tenants of the `fork()` system call are:

- **Control Flow Implications**: After a `fork()`, the child process may execute the same or different code based on the return value. This leads to complex flow control scenarios, especially when multiple `fork()`

calls are nested or combined with other control flow mechanisms like loops and conditionals.

- **Functionality**: The `fork()` system call is used to create a new process by duplicating the calling process. The new process is referred to as the child process, while the original process is the parent.
- **Memory Handling**: Initially, both processes share the same physical memory pages, but typically a copy-on-write mechanism is used where pages are duplicated only if either process attempts to modify them.
- **Memory Sharing**: Initially, the child process shares the same memory segments (code, data, and stack) with the parent process. Modern operating systems typically use a copy-on-write mechanism where the actual copying of the memory pages is deferred until one of the processes attempts to modify a page.
- **Process Tree**: When `fork()` is executed, it returns twice: once in the parent process (returning the child's PID) and once in the child process (returning 0). This dual return allows both processes to execute the same subsequent code but often follow different branches depending on the return value.
- **Return Value**: `fork()` returns twice, once in the parent process and once in the child process. In the parent, it returns the PID of the newly created child, while in the child, it returns 0. If `fork()` fails, it returns -1 in the parent.

Signal Handling

In UNIX and UNIX-like systems, signals are one of the primary methods for exceptional control flow, providing a way for processes to interrupt or notify each other asynchronously. Signals can indicate events like division by zero, termination requests, or user-defined conditions. They are used for a variety of purposes including process control, inter-process communication, and handling asynchronous events.

The key concepts of signal handling are:

- **Signal Delivery**: When a signal is generated, the operating system delivers it to the target process. This process then interrupts its current task to handle the signal.
- **Signal Generation**: Signals can be generated by errors, explicit requests via system calls (like kill), or hardware exceptions.
- **Signal Handlers**: Processes can register signal handlers, specific functions that are executed when signals are received. If a signal handler is not set, a default action is taken (usually terminating the process).

`kill()` System Call

The `kill()` function is a critical tool in UNIX and UNIX-like operating systems, used primarily for sending signals to processes or groups of processes. It enables a process to communicate with other processes through predefined signals, which can indicate various system events or requests.

Overview Of `kill()`

- **Function Prototype**: `int kill(pid_t pid, int sig);`
- **Parameters**:
  - `pid`: The process ID of the target process to which the signal is sent. This can also be a special value to target a group of processes.
  - `sig`: The signal number to be sent. This can be any of the standard signals defined in `<signal.h>`, like `SIGKILL`, `SIGTERM`, `SIGSTOP`, `SIGUSR1`, etc.
- **Return Value**: Returns 0 on success, and -1 on failure, setting `errno` to indicate the error.

Usage Of `pid` Parameter

- **Positive Value (`pid` > 0)**: Sends the signal to the process with the specified process ID.
- **Negative One (`pid` $== -1$)**: Sends the signal to all processes for which the calling process has permission to send signals, except for the system processes and the process sending the signal.
- **Negative Value (`pid` < 0)**: Sends the signal to all processes in the process group whose ID is the absolute value of `pid`.
- **Zero (`pid` $== 0$)**: Sends the signal to all processes in the sender's process group, which typically includes all processes started from the same terminal.

Error Handling

On failure, `kill` sets `errno` to one of the following values:

- `EINVAL`: The signal number is invalid.
- `EPERM`: The process does not have permission to send the signal to any of the target processes.
- `ESRCH`: The target process or process group does not exist.

The `kill()` function is an essential aspect of UNIX system programming, providing robust capabilities for process control and inter-process communication. Understanding its behavior and implications is crucial for effective system management and application development in UNIX environments. This function illustrates the powerful, albeit low-level, mechanisms available in UNIX systems for managing the process lifecycle and system resources.

`signal()` System Call

The `signal` system call is essential for setting up signal handling in UNIX and UNIX-like operating systems. It allows processes to manage how they respond to the various signals they might receive, which are typically used to indicate system events, errors, or external interruptions.

Overview

- **Function Prototype**: `void (*signal(int sig, void (*func)(int)))(int);`
- **Parameters**:
  - `sig`: The signal number to handle. This is an integer representing one of the system-defined signals like `SIGINT`, `SIGTERM`, `SIGCHLD`, etc.
  - `func`: A pointer to a function that will handle the signal, or a special constant like `SIG_IGN` for ignoring the signal or `SIG_DFL` for default handling.
- **Return Value**: Returns the address of the previous signal handler for the specified signal, or `SIG_ERR` in case of error.

Key Features

- **Default Behavior**: Setting the handler to `SIG_DFL` restores the default action for the signal, which often results in the process being terminated or stopped.
- **Ignoring Signals**: By passing `SIG_IGN` as the handler, the process can ignore the specified signal (except for signals that cannot be caught or ignored, like `SIGKILL` and `SIGSTOP`).

- **Signal Handlers**: A signal handler is a function designated to be called when a specific signal is received. It should have the following prototype: `void handler(int sig)`;

Signal Handling

- **Asynchronous Events**: Signals are asynchronous by nature, meaning they can interrupt the process at almost any point in its execution.

- **Atomic Operations**: Signal handlers should perform atomic operations or operate in a context where interruptions do not cause inconsistency or data corruption. They are typically used to set flags or handle simple state changes.

The `signal()` system call provides fundamental capabilities for handling asynchronous events in UNIX systems. It allows programs to define custom responses to various signals, which can enhance program robustness, enable graceful exits, and manage system resources effectively. Understanding and using `signal` effectively is crucial for advanced system programming and developing resilient applications in a UNIX environment.

Exceptional Control Flow (ECF) is a fundamental concept in systems programming, where the normal linear execution sequence of a program is interrupted or altered using mechanisms such as interrupts, signals, traps, system calls, and context switches. These mechanisms allow an operating system to perform more complex tasks like handling multiple processes, managing asynchronous events, and sharing resources among different programs efficiently.

Virtual Memory

Virtual Memory is a fundamental concept in modern computing systems, primarily designed to decouple the user's view of memory from the physical limitations of the available main memory. It provides an abstraction that allows each process to act as though it has its own contiguous block of addresses that it can read and write to, independently of the actual physical memory available.

Key Features Of Virtual Memory

- **Abstraction Of Physical Memory**: Virtual memory abstracts the underlying physical memory hardware, allowing an operating system to use hardware like disk storage to extend available memory.

- **Memory Management**: It simplifies memory management by allowing the system to move memory pages between physical memory and disk transparently.

- **Process Isolation**: Each process operates in its own virtual address space, which isolates it from other processes and the operating system, enhancing security and stability.

Components Of Virtual Memory

- **Physical Address Space**: The actual RAM available on the system.

- **Virtual Address Space**: This is the contiguous address space that processes use to access memory. Each process has its own virtual address space, which is mapped to the physical memory.

Address Translation

- **Page Table**: A data structure used by the operating system to store mappings from virtual pages to physical frames.

- **Virtual Addresses** are translated to **Physical Addresses** using data structures like page tables.

Pages And Page Tables

- **Page**: A fixed-size block of memory. Virtual memory is divided into pages, which are mapped into physical memory frames of the same size.
- **Page Table**: Holds the mapping of virtual pages to physical frames. Modern systems often use a multi-level page table structure to optimize memory usage and access time.

Virtual Memory Translation

Problems that consist of taking a memory addresses and translating them into their virtual and physical addresses respectively consist of some common variables:

$$\begin{aligned}
N &= 2^n && \text{(Number of virtual address bits } (n)) \\
M &= 2^m && \text{(Number of physical address bits } (m)) \\
P &= 2^p && \text{(Page size in bytes } (P)) \\
S &= 2^s && \text{(Number of TLB index bits } (s))
\end{aligned}$$

When translating an address into both the virtual address and physical address, there are specific values for both the virtual and physical addresses:

$$\begin{aligned}
\text{Physical Page Offset (PPO)} &= \text{First } p \text{ bits from, right to left of address in binary} \\
\text{Physical Page Number (PPN)} &= \text{Address found from TLB or Page Table, valid bit must be set} \\
\text{Physical Address (PA)} &= \text{PPN} + \text{PPO (Concatenated, not summed)} = m \text{ in size} \\
\text{Virtual Page Offset (VPO)} &= \text{First } p \text{ bits from, right to left of address in binary} \\
\text{Virtual Page Number (VPN)} &= \text{Bits from } p \text{ to } n-1 \text{ of address in binary} \\
\text{Virtual Address (VA)} &= \text{VPN} + \text{VPO (Concatenated, not summed)}, = n \text{ in size} \\
\text{TLB Index (TLBI)} &= (p, p+s-1) \text{ bits from } \underline{\text{Virtual Address}} \\
\text{TLB Tag (TLBT)} &= (p+s, n-1) \text{ bits from } \underline{\text{Virtual Address}}
\end{aligned}$$

The procedure for translating an address into both its virtual and physical address representation is:

1. Translate the original address from hexadecimal `0xXXXX` into binary.
    - Add padding (trailing zeros) if original address is smaller in total bits than virtual or physical address.
2. Calculate the number of offset bits p.
3. Calculate the VPO and PPO of the address in binary, convert to hexadecimal.
4. Calculate the VPN of the address in binary, convert to hexadecimal.
5. Calculate the number of TLB Index bits s.
    - Calculate the TLBI from the VA, translate to hexadecimal.
    - Calculate the TLBT from the VA, translate to hexadecimal.
6. Refer to the TLB to determine if there is a TLB hit.
    - A valid TLB hit occurs when an index (referred to as the set #), has a valid tag (V = Y) in any of the ways.

- If there is a TLB hit, the PPN can be found from the TLB.
- If there is no TLB hit, we must refer to the Page Table to see if there is a valid PPN.

7. Refer to the Page Table with the aforementioned VPN to determine the PPN.
   - The valid tag must be set in this case.
8. Calculate the PA if there was indeed a PPN either from step 6 or 7.

Virtual Memory is a sophisticated system that underpins modern operating systems, allowing them to efficiently use both the physical memory (RAM) and secondary storage (like HDDs or SSDs) to manage the execution of multiple processes. This system enhances the computer's multitasking capabilities, improves security and isolation among processes, and abstracts complex memory management details from the user and application programs.

Allocation `malloc`

There are different forms memory allocation, the key concepts of memory allocation are:

- **Boundary Tag Method**:
  - This method involves using the headers and footers of blocks as "tags" to hold information about the block size and allocation status. These tags help in merging adjacent free blocks during deallocation (coalescing), effectively reducing fragmentation.
  - Both the header and footer of a block generally contain the same information so that blocks can be merged efficiently whether the allocator is moving forward or backward through the heap.
- **Implicit List Allocator**:
  - An implicit list uses the blocks themselves to keep track of memory allocations. Each block on the heap typically contains a header and possibly a footer that store metadata about the block, such as its size and whether it's allocated or free.
  - The allocator traverses the heap from the beginning to find a free block that fits the size requirement of a malloc call (first-fit, best-fit, etc.).
- **Malloc And Free Operations**:
  - **`malloc(size)`**: Allocates a block of at least **`size`** bytes and returns a pointer to the allocated memory. The allocator might also include additional bytes for alignment and metadata (headers and footers).
  - **`free(ptr)`**: Deallocates the block of memory pointed to by **`ptr`**, potentially merging it with adjacent free blocks to form larger free blocks and reduce fragmentation.

Best Fit

The Best Fit method searches the entire free list and takes the smallest block that is adequate to fulfill the request. This approach aims to find the closest matching block in size to the requested memory.

Advantages

- **Efficient Utilization Of Space**: It tends to use memory more efficiently over the long term, as it leaves larger blocks of memory available for future allocations.
- **Reduced Fragmentation**: By allocating the smallest block that meets the size requirement, Best Fit minimizes the leftover space in memory blocks, reducing external fragmentation.

Disadvantages

- **Maintenance Overhead**: Best Fit requires more complex bookkeeping to keep track of the sizes of all free blocks, which can add overhead.
- **Performance**: Scanning the entire list to find the optimal block can be slower, especially if the free list is long. This comprehensive search increases allocation time.
- **Possible Internal Fragmentation**: Although it reduces external fragmentation, it may increase internal fragmentation if the best fit block is slightly larger than needed.

First Fit

The First Fit method of memory allocation scans the heap from the beginning and allocates the first block that is large enough to satisfy the request. It stops searching as soon as it finds a block of sufficient size.

Advantages

- **Simplicity**: The algorithm is straightforward and easy to implement.
- **Speed**: First Fit generally performs faster than Best Fit for allocation because it stops searching as soon as it finds a free block that fits, rather than continuing to search for a better fit.

Disadvantages

- **Fragmentation**: It may lead to higher fragmentation compared to Best Fit because it can leave smaller unusable spaces scattered throughout the heap. This happens as it might skip smaller blocks that are a better fit in favor of the first block that fits the request.
- **Suboptimal Allocation**: Over time, it could result in larger free blocks at the end of the heap being underutilized.