Software Construction Laboratory CS35L – Lab 1

Course Webpage: https://web.cs.ucla.edu/classes/fall18/cs35L/index.html
TA: Zhiyi Zhang
Email: zhiyi@cs.ucla.edu
Webpage: https://zhiyi-zhang.com

Session 10-2

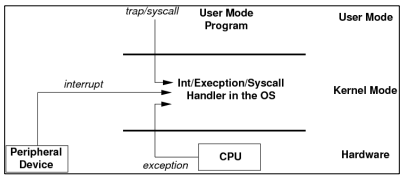
- Week 5: CPU Mode, System Call
- Week 6: Multithreading
- Week 7: Linking
- Week 8: Security, SSH
- Week 9: Git

Week 5

Processor/Operating/CPU Modes

Operating systems place restrictions on the type and scope of operations that can be performed by certain processes

- only highly trusted kernel code is allowed to execute in the unrestricted mode



User Mode v.s. Kernel Mode

User Mode (mode bit = 1)

- restricted access to system resources
- CPU restricted to unprivileged instructions and a specified area of memory

Kernel Mode (mode bit = 0)

- unrestricted access
- CPU is **unrestricted**, can use all instructions, access all areas of memory and take over the CPU anytime

Why Dual-mode Operation?

System resources are shared among processes and the OS must ensure

- **Protection:** an incorrect/malicious program cannot cause damage to other processes or the system as a whole
  - Prevent processes from accessing illegal memory and modifying kernel code and data structures
  - Prevent processes from performing illegal I/O operations
- **Fairness:** Make sure processes have a fair use of devices and the CPU
  - Prevent a process from using the CPU for too long

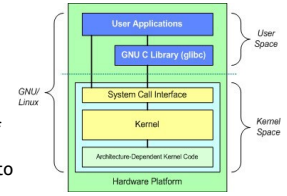
Only Kernel Code is Trusted

Core of OS software executing in kernel/supervisor state

Trusted software:

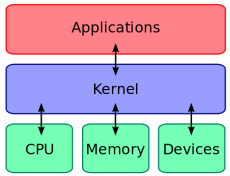
- Software who manages hardware resources (CPU, Memory and I/O)
- Software who implements protection mechanisms that could not be changed through actions of untrusted software in user space

System call interface is a safe way to expose privileged functionality and services of the processor



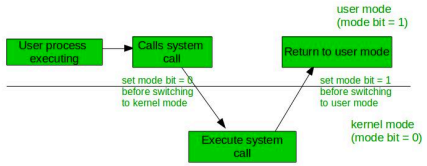
How User Mode Processes Use Kernel Space?

- The kernel executes privileged operations on behalf of untrusted user processes
- The way is using **system calls**



When System Calls Are Made

- When a system call is made, the program being executed is interrupted and control is passed to the kernel
- If operation is valid, the kernel performs it



System Call Overhead

System calls are expensive and can hurt performance

The system must do many things

- Process is interrupted & computer saves its state
- OS takes control of CPU & verifies validity of operations
- OS performs requested action
- OS restores saved context, switches to user mode
- OS gives control of the CPU back to user process

Examples

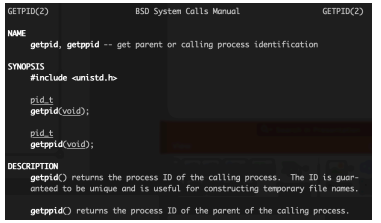
Process Control	fork(), exit(), wait()
File Manipulation	open(), read(), write(), close()
Device Manipulation	ioctl(), read(), write()
Information Maintenance	getpid(), alarm(), sleep()
Communication	pipe(), shmget(), mmap()
Protection	chmod(), umask(), chown()

How to Use System Calls?

Use man to check the system calls

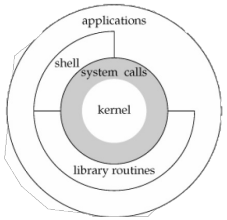
Examples:

- getpid(), getppid()
- read(), write(),
- open(), close()
- fstat(),
- dup()



System Call Through Library Functions

- Functions that are a part of standard C library
- To reduce system call overhead, use equivalent library functions
  - getchar, putchar vs. read, write (for standard I/O)
  - fopen, fclose vs. open, close (for file I/O), etc.
- How do these functions perform privileged operations?
  - They make system calls
- So what's the difference?

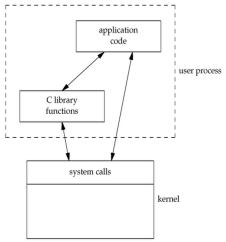


Why Bother?

Libraries wrap the system call and optimize the performance

- Usually less system calls
- Non-frequent switches from user mode to kernel mode

Therefore less overhead and better developing experiences.



Library Function Call Examples

- getchar, putchar vs. read, write (for standard I/O)
- fopen, fclose vs. open, close (for file I/O)
- malloc vs. brk sbrk (for memory allocation)
- …

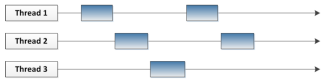
Case Study: Why fopen Is Better Than open

- fopen provides you with **buffered I/O** while open is **unbuffered I/O**
  - Every byte is read/written by the kernel through a system call in unbuffered I/O while buffered I/O collect as many bytes as possible (in a buffer) and read more than a single byte (into buffer) at a time and use one system call for a block of bytes
- fopen returns FILE struct which can also be used by other stdio functions like fscanf()
- fopen can work across platforms (fopen can work on platforms that does not use open, e.g., some old versions of C)

Week 6

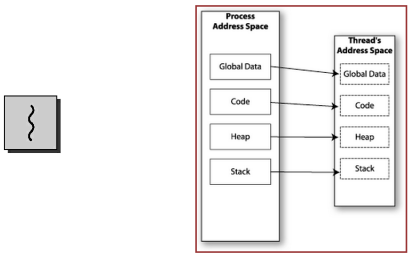
Multithreading with single/multiple CPUs

Multiple threads sharing a single CPU

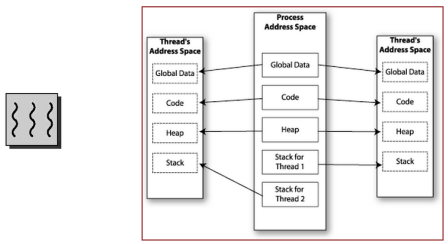


Multiple threads on multiple CPUs

Memory Layout: Single-Threaded Program

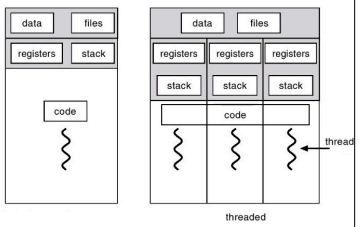


Memory Layout: Multithreaded Program



thread

- A thread is a single sequence stream within in a process.
- A thread share the code section, data section, and heap space with other threads.
- A thread has its own stack space, register set.



Multithreading

Threads share all of the process's memory except for their stacks. Therefore, data sharing requires no extra work (no system calls, pipes, etc.)

Shared memory makes multithreaded programming

- **Powerful**: can easily access data and share it among threads
- **More efficient**: No need for system calls when sharing data and thread creation and destruction are less expensive than process creation and destruction
- **Non-trivial**: Have to prevent several threads from accessing and changing the same shared data at the same time (synchronization)

Compile your c program with pthread

- Link your program with pthread library: -lpthread
- gcc [source_files] -lpthread -o [output_name]
- Demo with a very simple example.

```
#include <stdio.h>
#include <pthread.h>
#include <unistd.h>
void* threadfun(void* a)
{
  sleep(1);
  printf("Printing from Thread.\n");
  return NULL;
}

int main(void)
{
  pthread_t thread_id;
  printf("Before Thread\n");
  pthread_create(&thread_id, NULL, threadfun,
NULL);
  pthread_join(thread_id, NULL);
  printf("After Thread\n");
  return 0;
```

Multithreaded Programming in C: pthread

There are 5 basic pthread functions:

- **pthread_create**: creates a new thread within a process
- **pthread_join**: waits for another thread to terminate
- **pthread_equal**: compares thread ids to see if they refer to the same thread
- **pthread_self**: returns the id of the calling thread
- **pthread_exit**: terminates the currently running thread

pthread_create

- Creates a new thread and makes it executable
- Can be called any number of times from anywhere within your code
- Return value:
  - Success: zero
  - Failure: error number

Reference

int pthread_create(pthread_t ***thread***, const pthread_attr_t ***attr***,
void *(***start_routine***) (void *), void ***arg***);

Parameters of pthread_create

- **thread**: unique identifier for newly created thread
- **attr**: object that holds thread attributes (priority, stack size, etc.)
  - Pass in NULL for default attributes
- ***start_routine*** : function that thread will execute once it is created
- **arg**: a single argument that may be passed to ***start_routine***
  - Pass in NULL if no arguments

pthread_join

- Makes main thread wait for the completion of all its spawned threads' tasks
- Without join, the originating thread would exit as soon as it completes its job
  - A spawned thread can get aborted even if it is in the middle of its processing
- Return value:
  - Success: zero
  - Failure: error number

Reference

int pthread_join(pthread_t **tid**, void ****status**);

Parameters of pthread_join

- **tid**: thread ID of thread to wait on
- **status**: the exit status of the target thread is stored in the location pointed to by *status
  - Pass in NULL if no status is needed

pthread_self

- The **pthread_self**() function returns the ID of the calling thread.
- This is the same value that is returned in *thread in the **pthread_create** call that created this thread.

Reference

pthread_t pthread_self(void);

pthread_equal

- The pthread_equal() function compares two thread identifiers.
- If the two thread IDs are equal, pthread_equal() returns a nonzero value; otherwise, it returns 0.

Reference

int pthread_equal(pthread_t **t1**, pthread_t **t2**);

pthread_exit

- The pthread_exit() function terminates the calling thread and returns a value that is available to another thread that calls pthread_join()

Reference

void pthread_exit(void *retval);

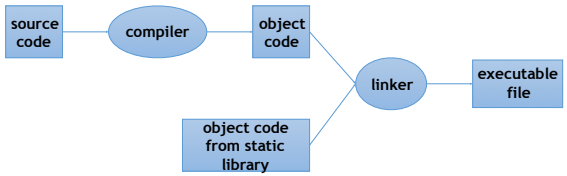
Week 7

Static Linking

- Carried out only once to produce an executable file
- If static libraries are called, the linker will copy all the modules referenced by the program to the executable

Static libraries are typically denoted by the **.a** file extension

Static Linking



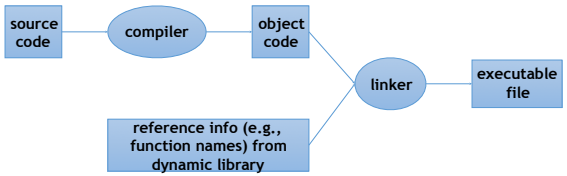
A previously compiled collection of standard program functions

Dynamic Linking

- Allows a process to add, remove, replace or relocate object modules during its execution
- If shared libraries are called:
  - Only copy a little reference information when the executable file is created
  - Complete the linking during loading time or running time

Dynamic libraries are typically denoted by the **.so** file extension (.dll on Windows)

Dynamic Linking



Doesn't require the code to be copied

Summary: Static and Dynamic Linking

Static Linking

Cons:

- Executable file is large in size

Pros:

- Complete and self-contained.

Dynamic Linking

Cons:

- Not self-sufficient
- Broken when library becomes incompatible

Pros:

- The size of executable file is smaller.
- No need to re-compile executable file if we update library functions
- More efficient mem use when several programs share the lib functions

Commands for Creating Static Library

- Create a C file or C files that contains functions in your library
- Create a header file or headers for the library
- Compile library files (-c to get object files)
  - gcc -c [your source files] -o [object file(s)]
- Create static library. This step is to bundle multiple object files in one static library.
  - ar rcs [your lib name] [object file(s)]

Using Static Library

- Create a c file containing a main function
- Compile the source file with
  - -L[search path] to add the searching path for you static library
  - -l[library name] to add the static library
  - gcc -o [executable file] [your source file] -L[path] -l[lib name]

GCC Flags For Creating Dynamic Library

- **-fPIC:**

This flag stands for "Position Independent Code" generation, a requirement for shared libraries. Because it's impossible to know where the shared library code will be, this flag allows the code to be located at any virtual address at runtime.

- **-shared:**

This flag creates the shared library (shared libraries have the prefix *lib* and suffix *.so*

Commands for Creating Shared Library

- Create a C file or C files that contains functions in your library
- Create a header file or headers for the library
- Compiling with Position Independent Code
  - gcc -c -fPIC [your source files] -o [object file(s)]
- Create dynamic library.
  - gcc -shared -o [your lib name] [object file(s)]

Using Shared Library

- Create a c file containing a main function
- Compile the source file with
  - -L[search path] to add the searching path for you static library
  - -l[library name] to add the dynamic library
  - gcc -o [executable file] [your source file] -L[path] -l[lib name]
- This will cause problem when you run the executable file
  - Where to load the shared library?
  - Use **-Wl,-rpath=[path]** flag: -Wl passes options to linker. -rpath at runtime finds .so from this path.
  - gcc -o [executable file] [your source file] -L[path] -l[lib name] -Wl,-rpath=[path]

Dynamic Loading

Dynamic loading is a mechanism by which a computer program can, at run time, load a library (or other binary) into memory

How to use?

- Adding header file: #include <dlfcn.h>
- When compiling your program: add gcc flag: -ldl
- In you program using dl APIs

Dynamic Loading APIs

Table 1. The Dl API

Function	Description
dlopen	Makes an object file accessible to a program
dlsym	Obtains the address of a symbol within a `dlopen`ed object file
dlerror	Returns a string error of the last error that occurred
dlclose	Closes an object file

Using LD APIs

- Opening Dynamic Library

```
void* sdl_library = dlopen("libSDL.so", RTLD_LAZY);
if (sdl_library == NULL) { // report error ... }
else { // use the result in a call to dlsym }
```

- Extracting Content from Dynamic Library

```
void* initializer = dlsym(sdl_library,"SDL_Init");
if (initializer == NULL) { // report error ... }
else { // cast initializer to its proper type and use }
```

- Unloading Dynamic Library

```
dlclose(sdl_library);
```

Week 8

Encryption Types

Symmetric Key Encryption

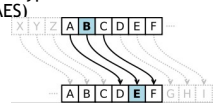
- a.k.a shared secret/key
- Key used to encrypt is the same as key used to decrypt

Asymmetric Key Encryption: Public/Private

- 2 different (but related) keys: public and private
  - Only creator knows the relation. Private key cannot be derived from public key
- Data encrypted with public key can only be decrypted by private key and vice versa
- Public key can be seen by anyone but never publish private key!!!

Symmetric-key Encrption

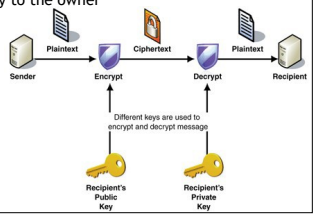
- Same secret key used for encryption and decryption
- Typical example: Caesar's cipher, Data Encryption Standard (DES), Advanced Encryption Standard (AES)
- Caesar's cipher
  - Map the alphabet to a shifted version
  - Key is 3 (number of shifts of the alphabet)



- **Key distribution** is a problem
  - The secret key has to be delivered in a safe way to the recipient
  - Diffie Hellman

Public-key Encryption (Asymmetric)

- Uses a pair of keys for encryption
  - **Public key** – Published and known to everyone
  - **Private key** – Secret key known only to the owner
- Encryption
  - Use public key to encrypt messages
  - Anyone can encrypt
- Decryption
  - Use private key to decrypt message
- Example: RSA, ABE



Digital Signature

- An electronic stamp or seal
  - Almost exactly like a written signature, except more guarantees!
- Is appended to a message
  - Or sent separately (detached signature)
- Ensures data integrity and authenticity
  - Data was not changed during transmission
  - Data was truly signed by the private key

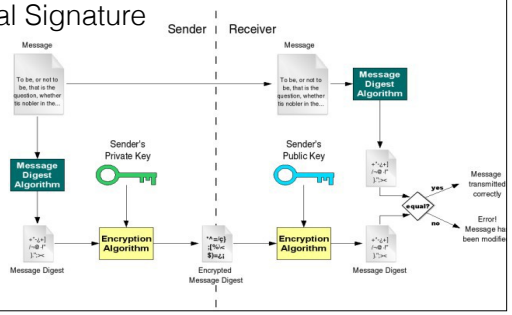
Steps for Generating a Digital Signature

1) Generate a Message Digest
   - The message digest is generated using a set of hashing algorithms
   - A message digest is a 'summary' of the message we are going to transmit
2) Create a Digital Signature
   - The message digest is encrypted using the sender's *private* key. The resulting encrypted message digest is the *digital signature*
3) Attach digital signature to message and send to receiver

Steps for Verifying a Digital Signature

1) Recover the Message Digest
   - Decrypt the digital signature using the sender's public key to obtain the message digest generated by the sender
2) Generate the Message Digest by the receiver itself
   - Use the same message digest algorithm used by the sender to generate a message digest of the received message
3) Compare digests (the one sent by the sender as a digital signature, and the one generated by the receiver)
   - If they are not exactly the same => the message has been tampered with by a third party

Digital Signature

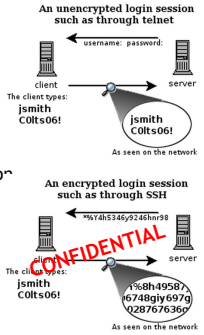


Message Authentication Code

- MAC confirms that the message came from the stated sender (its authenticity) and has not been changed (its integrity)
- A strawman solution
  - Hash(Message)
- keyed-hash message authentication code (HMAC)
  - A specific type of message authentication code (MAC) involving a cryptographic hash function and a secret cryptographic key
  - Two parties use the shared key to generate HMAC
  - HMAC is generated over the message and the key

What is SSH?

- **Sh**ell
- Used to remotely access shell
- Successor of telnet
- Encrypted and better authenticated session



How does SSH work?

- Based on TCP protocol (IP address + port number)
  - Enroll CS118 if you are not familiar with TCP/IP network
- Public Key Signature to provide host authentication
- Symmetric Key Encryption to ensure message confidentiality
- Message Authentication Code (MAC) to ensure integrity
- Password or Public Key Signature to provide user authentication

SSH Command

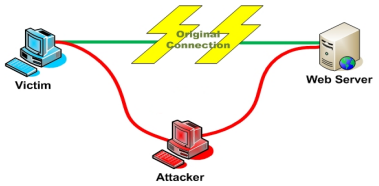
- Client ssh's to remote server
  - ssh connects and logs into the specified destination
  - Format: ssh [options] [username@]hostname
- Examples
  - ssh 192.168.1.2
  - ssh root@192.168.1.2
  - ssh zhiyi@lnxsrv09.seas.ucla.edu
  - ssh -p 8022 192.168.1.2
    - Use the port 8022 instead of the default 22

First-Time SSH and Host Validation

- If ssh for the first time, the **host validation** process will be triggered
  - TOFU: Trust on First Time Use.
  - SSH shows hostname, IP address and fingerprint of the server's public key, so you can be sure you're talking to the correct host
    - The authenticity of host 'somehost (1.2.3.4)' can't be established. RSA key fingerprint is blablabla. Are you sure you want to continue connecting (yes/no)?
- After accepting, the host public key is saved in ~/.ssh/known_hosts

Host Validation

- Next time client connects to server
  - Check host's signature using the public saved
- Man In The Middle (MITM) Attack?



Session Encryption and Data Integrity

- Client and server agree on a **symmetric encryption key** (session key)
  - Diffie-Hellman
- All messages sent between client and server
  - MAC (message authentication code) calculated
  - encrypted at the sender with session key
  - decrypted at the receiver with session key
  - MAC (message authentication code) verified
- Anybody who doesn't know the session key doesn't know any of the contents of those messages

User Authentication

Password-based authentication

- Prompt for password on remote server
- If username specified exists and remote password for it is correct then the system lets you in

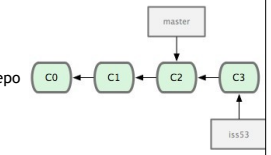
Key-based authentication

- The Client side generates a digital signature using the client's private key
- The Server side verifies the signature using client's public key
- Client's key must be known to the Server in advance

Week 9

Terminologies in Git

- Head
  - Refers to a commit object
  - There can be more than one heads in a repo
- HEAD
  - Refers to the current active head
- Branch
  - Refers to a head and its entire set of ancestor commits
- Master Branch
  - Default branch
- Detached HEAD
  - If a commit is not pointed to by a branch



Initialize a Git Repo and Add a Commit

git init

- Creates an empty git repo
- Creates a .git sub directory in the current dir and contains all the information of the this repo

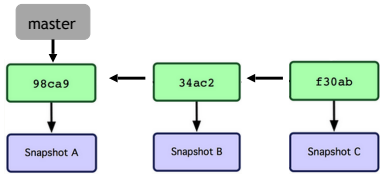
git add .

- Stage your changes
- A must before a commit

git commit

- Commit your staged changes to the repo and create a git commit
- Commit message to let yourself/others to know the purpose and other information of your change

When Commit on a Branch



Get Repo From an Existing Repo

Obtain a repo from a web-based hosting service, e.g., GitHub.

git clone

- Create a copy of an existing repository

Be Aware of Your Local Changes

git status
- Show your modified files
- Whether your changes have been staged or not

git diff
- Show changes we made compared to the current commit and staged files

git diff HEAD
- Show changes we made compared to the HEAD commit

Controlling Version with Branch and Tags

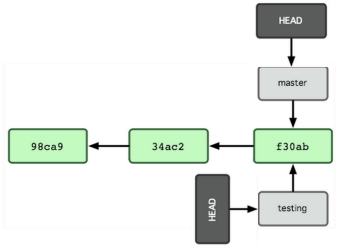
git checkout
- Jump to the specified commit or branch or tag
- Can also be used to reset the changes

git branch, git checkout -b
- Create a new branch on the current commit or specified commit

git tag
- Create a tag to the current commit

When Checkout to another Branch

git checkout testing



Integrating Changes

Needed when you want to integrate the changes from multiple branches

Two main ways
- **Merge**
  - simple and straightforward
  - merge two branches into one
- **Rebase** (recommended)
  - much cleaner
  - apply one branch's changes onto the other branch

Some other ways, e.g., cherry-pick