CS 35L

Week 10

TA: Tomer Weiss
March-07-2016

goo.gl/xIdDFA

Slides

Announcements

- Student presentations today:
  - Judge confirms what many suspected: Feds hired CMU to break Tor
  - Quantum Computing
  - Detecting Hidden Malicious Ads
  - ?
  - For reference on presentation, grading, please refer to this rubric.
  - Assignment 10 is due this Friday, 11:55pm

- Final next week!
  - Final review today

Announcements - Please Fill teaching evaluation survey

Students have until 8:00 AM Saturday, March 12 to log into MyUCLA to complete evaluations for your courses listed below:

COM SCI 35L section 6

You have the option to use class time for your students to complete their evaluations with smartphones, iPads, and tablets. Please note that not all classrooms are WI-FI enabled.

For more details about this process, please contact your departmental Evaluation Coordinator or visit our website at:

http://www.oid.ucla.edu/assessment/eip

Thank you!
Evaluation of Instruction Program
eip@oid.ucla.edu

Final Review

Week 10

Notes about these review slides:

- Not comprehensive
  - Meant to give you a review of some of the concepts we covered in the course
- Conceptual understanding of concepts is more important than memorization
  - The final is open book/note, if you need something specific you'll have it in front of you!
    - no electronic devices though
- Questions are posed throughout the review slides
  - Strive to be able to confidently answer all these questions

Final Information

- Tuesday, March 15, 2016, 11:30am-2:30pm
- Boelter 3760 (our regular room)
- Open book and open note
  - No calculators, smartphones, smartwatches, etc.
- 50% of course grade (from syllabus)

Sample Final Review

Week 10

Week 5

Debugger

- A program that is used to run and debug other (target) programs
- Advantages:

  Programmer can:

  – step through source code line by line
    - each line is executed on demand
  – interact with and inspect program at run-time
  – If program crashes, the debugger outputs where and why it crashed
- Why have a debugger?
- When do I use a debugger?

Using GDB

1. **Compile Program**
   - Normally: `$ gcc [flags] <source files> -o <output file>`
   - Debugging: `$ gcc [other flags] -g <source files> -o <output file>`
     - enables built-in debugging support

2. **Specify Program to Debug**
   - `$ gdb <executable>`

     or
   - `$ gdb`
   - `(gdb) file <executable>`

Using GDB

3. **Run Program**
   - `(gdb) run` or
   - `(gdb) run [arguments]`
4. **In GDB Interactive Shell**
   - Tab to Autocomplete, up-down arrows to recall history
   - `help [command]` to get more info about a command
5. **Exit the gdb Debugger**
   - `(gdb) quit`

Run-Time Errors

- Segmentation fault
  - Program received signal SIGSEGV, Segmentation fault. 0x0000000000400524 in *function* (arr=0x7fffc902a270, r1=2, c1=5, r2=4, c2=6) at *file.c*:12
    - Line number where it crashed and parameters to the function that caused the error
- Logic Error
  - Program will run and exit successfully
- How do we find bugs?
- What if we can't reproduce the error?

Setting Breakpoints

- Breakpoints
  - used to stop the running program at a specific point
  - If the program reaches that location when running, it will pause and prompt you for another command
- Example:
  - `(gdb) break file1.c:6`
    - Program will pause when it reaches line 6 of file1.c
  - `(gdb) break my_function`
    - Program will pause at the first line of `my_function` every time it is called
  - `(gdb) break [position] if expression`
    - Program will pause at specified position only when the expression evaluates to true

- How do we know where to set breakpoints?
- What do we do once we've stopped at a breakpoint?

Deleting, Disabling and Ignoring BPs

- `(gdb) delete [bp_number | range]`
  - Deletes the specified breakpoint or range of breakpoints
- `(gdb) disable [` *`bp_number | range`*`]`
  - Temporarily deactivates a breakpoint or a range of breakpoints
- `(gdb) enable [` *`bp_number | range`*`]`
  - Restores disabled breakpoints
- If no arguments are provided to the above commands, all breakpoints are affected!!
- `(gdb) ignore` *`bp_number iterations`*
  - Instructs GDB to pass over a breakpoint without stopping a certain number of times.
    - bp_number: the number of a breakpoint
    - Iterations: the number of times you want it to be passed over

Displaying Data

- Why would we want to interrupt execution?
  - to see data of interest at run-time:
  - `(gdb) print [/format] expression`
    - Prints the value of the specified expression in the specified format
  - Formats:
    - d: Decimal notation (default format for integers)
    - x: Hexadecimal notation
    - o: Octal notation
    - t: Binary notation
- What's the point of displaying data?
- What sort of data might we want to display?
- How can we use displayed data?

Resuming Execution After a Break

- When a program stops at a breakpoint
  - 4 possible kinds of gdb operations:
    - **c or continue**: debugger will continue executing until next breakpoint
    - **s or step**: debugger will continue to next source line
    - **n or next**: debugger will continue to next source line in the current (innermost) stack frame
    - **f or finish**: debugger will resume execution until the current function returns. Execution stops immediately after the program flow returns to the function's caller
      - the function's return value and the line containing the next statement are displayed
- What is the difference between ‘s’ and ‘n’?
- When would we use each one of the above?

Process Memory Layout

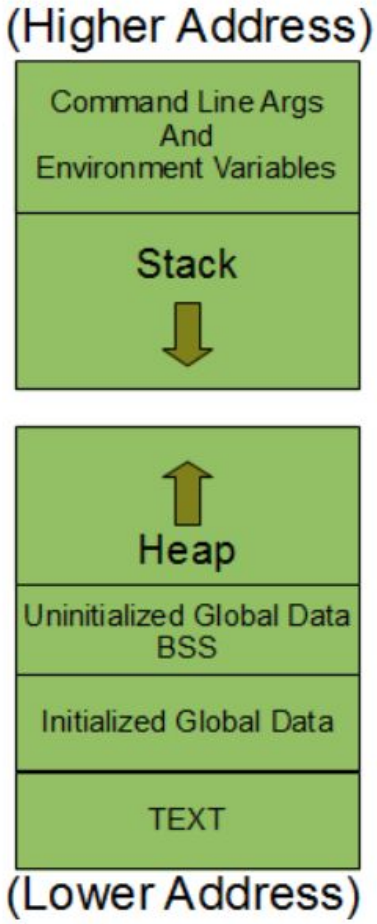


Image source : thegeekstuff.com

- TEXT segment
  - Contains machine instructions to be executed
- Global Variables
  - Initialized
  - Uninitialized
- Heap segment
  - Dynamic memory allocation
  - malloc, free
- Stack segment
  - Push frame: Function invoked
  - Pop frame: Function returned
  - Stores
    - Local variables
    - Return address, registers, etc
- Command Line arguments and Environment Variables

Stack Info

- A program is made up of one or more functions which interact by calling each other
- Every time a function is called, an area of memory is set aside for it. This area of memory is called a **stack frame** and holds the following crucial info:
  - storage space for all the local variables
  - the memory address to return to when the called function returns
  - the arguments, or parameters, of the called function
- Each function call gets its own stack frame. Collectively, all the stack frames make up the **call stack**
- Why does the stack exist at all? How is the stack different than the heap?

Analyzing the Stack in GDB

- `(gdb) backtrace|bt`
  - Shows the call trace (the call stack)
  - Without function calls:
    - #0 main () at program.c:10
    - one frame on the stack, numbered 0, and it belongs to main()
  - After call to function display()
    - #0 display (z=5, zptr=0xbffffb34) at program.c:15

      #1 0x08048455 in main () at program.c:10
    - Two stack frames: frame 1 belonging to main() and frame 0 belonging to display().
    - Each frame listing gives
      - the arguments to that function
      - the line number that's currently being executed within that frame

C Programming

- You should be able to develop a basic C program that incorporates the following:
  - Basic data types
  - Control/flow (if, while, etc)
  - Pointers
  - Structs
  - Dynamic memory
  - Basic I/O

Dynamic Memory

- Memory that is allocated at runtime
  - Why?
- Allocated on the heap
  - Why not the stack?

void *malloc (size_t size);

- Allocates *size* bytes and returns a pointer to the allocated memory

void *realloc (void *ptr, size_t size);

- Changes the size of the memory block pointed to by *ptr* to *size* bytes

void free (void *ptr);

- Frees the block of memory pointed to by *ptr*

- What happens if I never call free?
- What happens if I try to put data into dynamic memory but I haven't yet called malloc?

Week 6

Communication Over the Internet

- What type of guarantees do we want?
  - **Confidentiality**
    - Message secrecy
  - **Data integrity**
    - Message consistency
  - **Authentication**
    - Identity confirmation
  - **Authorization**
    - Specifying access rights to resources
- Why do we want these guarantees?

Cryptography

- **Plaintext** – Actual message
- **Ciphertext** – Encrypted message (unreadable gibberish)
- **Encryption** – Going from plaintext to ciphertext
- **Decryption** – Going from ciphertext to plaintext
- **Secret key**
  - Part of the mathematical function used to encrypt/decrypt.
  - Good key makes it hard to get back plaintext from ciphertext

Be familiar with all of these terms and what they mean/represent.

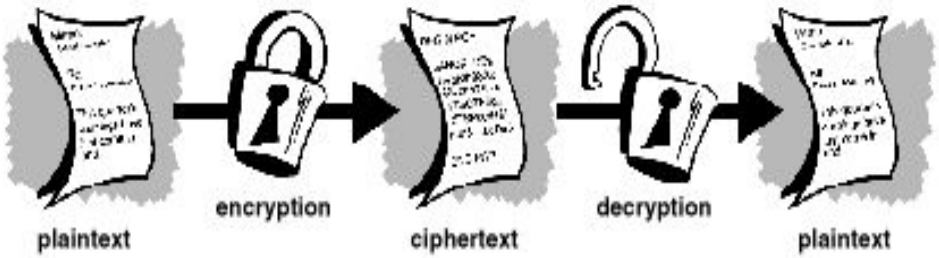


Image Source: gpgtools.org

Symmetric-key Encrption

- Same secret key used for encryption and decryption
- **Example** : Data Encryption Standard (**DES**)
- **Caesar's cipher**
  - Map the alphabet to a shifted version
    - ABCDEFGHIJKLMNOPQRSTUVWXYZ
    - DEFGHIJKLMNOPQRSTUVWXYZABC
  - Plaintext – SECRET. Ciphertext – VHFUHW
  - Key is 3 (number of shifts of the alphabet)

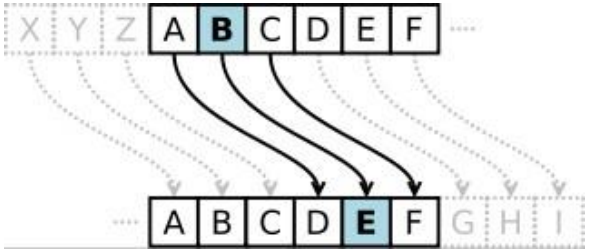


Image Source: wikipedia

- **Key distribution** is a problem
  - The secret key has to be delivered in a safe way to the recipient
  - Chance of key being compromised

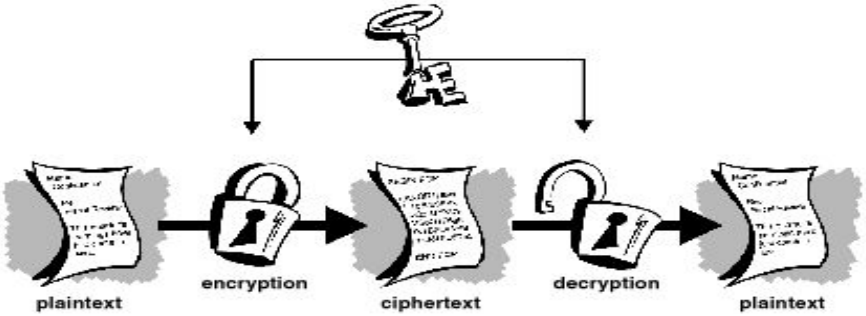
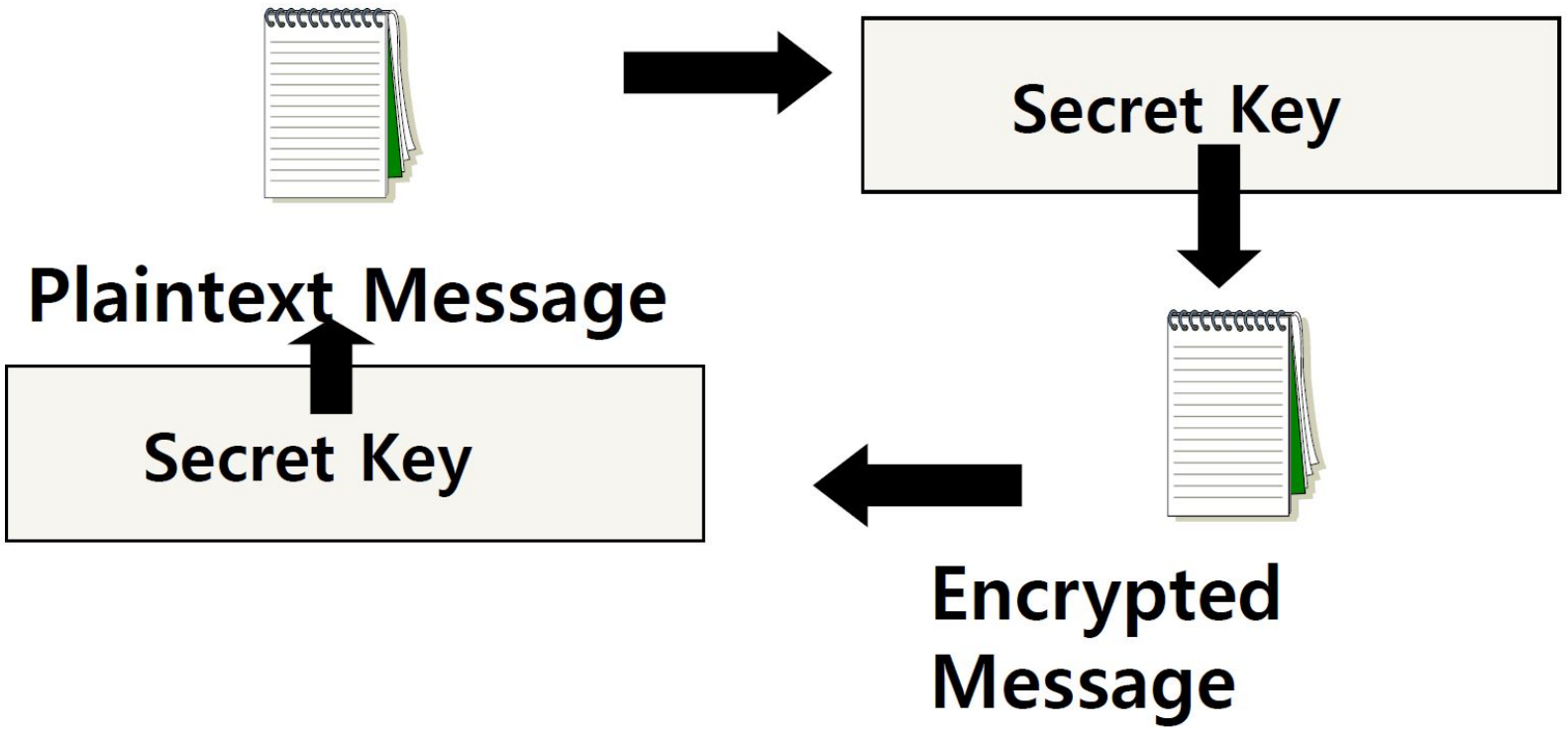


Image Source: gpgtools.org

Secret Key (symmetric) Cryptography

- A single key is used to both encrypt and decrypt a message



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 message, but they cannot decrypt the ciphertext
- **Decryption**
  - Use private key to decrypt messages
- **Example** : **RSA** – Rivest, Shamir & Adleman
  - Property used - **Difficulty of factoring** large integers to prime numbers
  - N = p * q (3233 = 61 * 53)
  - N is a large integer and p, q are prime numbers N is
  - part of the public key
  - http://en.wikipedia.org/wiki/RSA_Factoring_Challenge

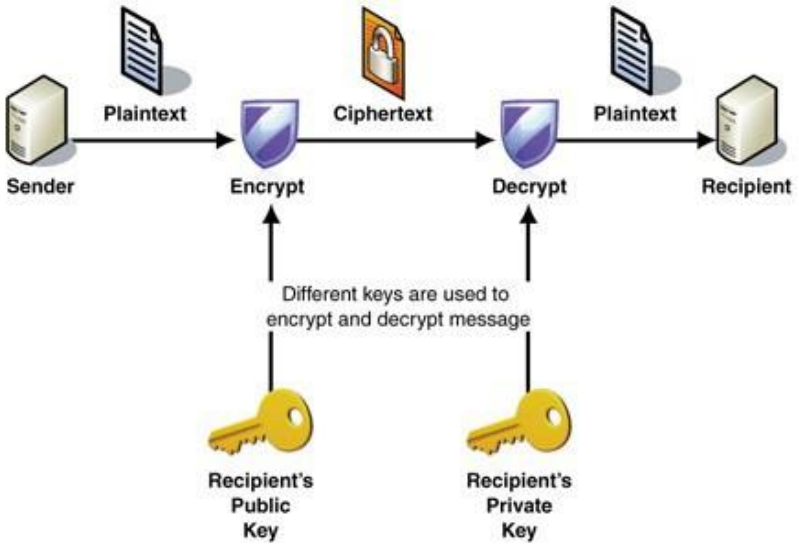
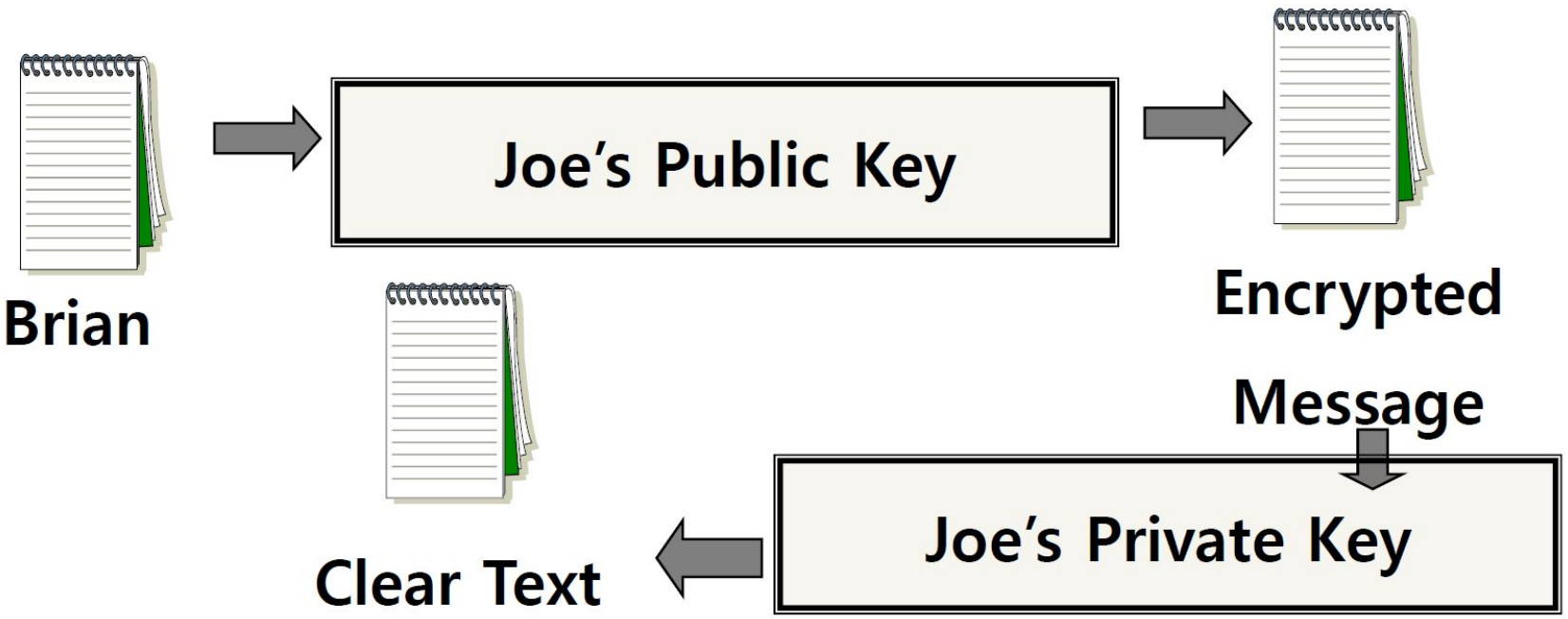


Image Source: MSDN

Public Key (asymmetric) Cryptography

- Two keys are used: a public and a private key. If a message is encrypted with one key, it has to be decrypted with the other.



Encryption questions

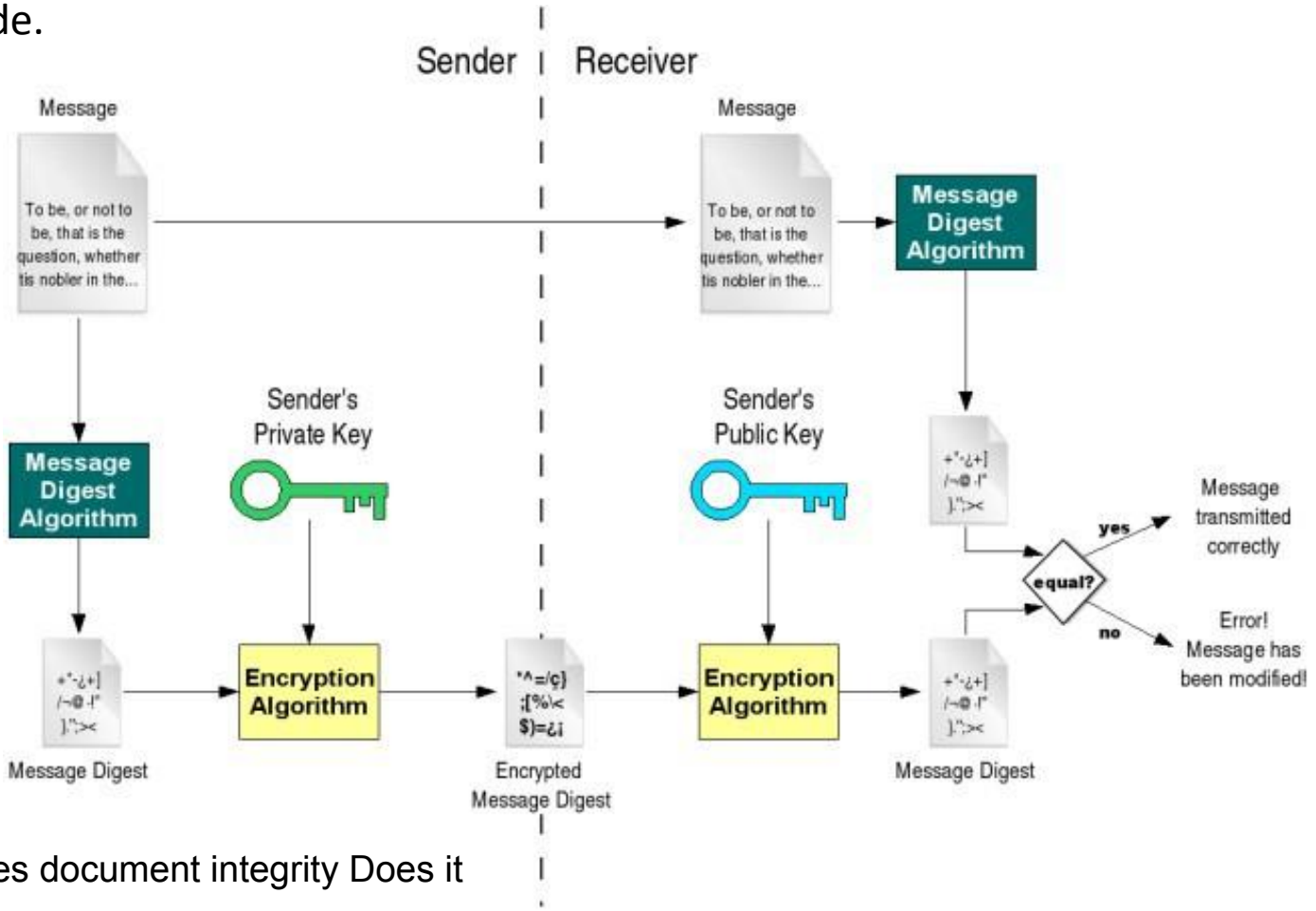
- Why have encryption?
- What are the differences between symmetric and asymmetric encryption? When would I use one or the other?
- Why is the RSA important? What is meant by ‘difficulty factoring’ or ‘intractable’?
- What is used on the Internet? What is a certificate authority?
- How can I trust a message came from someone?

Digital Signature

- An electronic stamp or seal
  - almost exactly like a written signature, except more guarantees!
- Is appended to a document
  - Or sent separately (detached signature)
- Ensures data integrity
  - document was not changed during transmission
- How are signatures different than encryption?

Digital Signature

You should understand this entire slide.



- Verifies document integrity Does it
- prove origin?
- Who is Certificate Authority (CA) ?

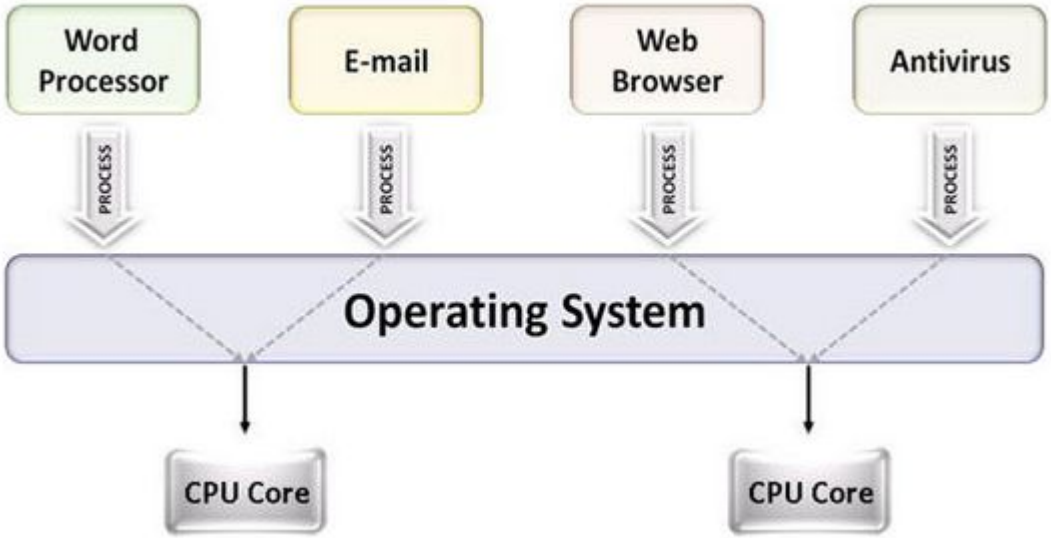
Image Source : gdp.globus.org

Week 7

Parallelism

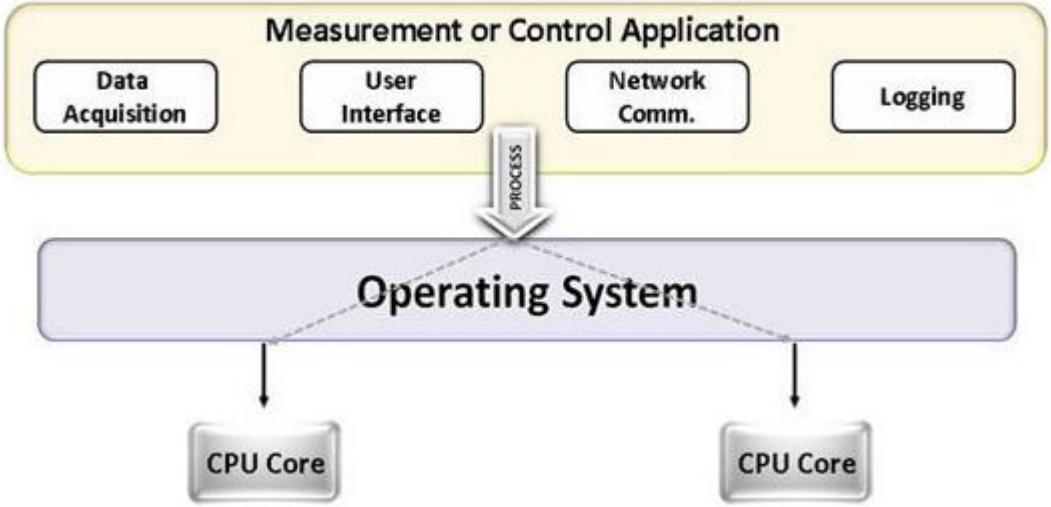
- Executing several computations simultaneously to gain performance
- Different forms of parallelism
  - **Multitasking**
    - Several processes are scheduled alternately or possibly simultaneously on a multiprocessing system
  - **Multithreading**
    - Same job is broken logically into pieces (threads) which may be executed simultaneously on a multiprocessing system
- What's the point of parallelism? Isn't it just too complicated?
- How can you decide whether your application should use multiple processes or multiple threads? Or both?

Multitasking vs. Multithreading



Multitasking

Multithreading



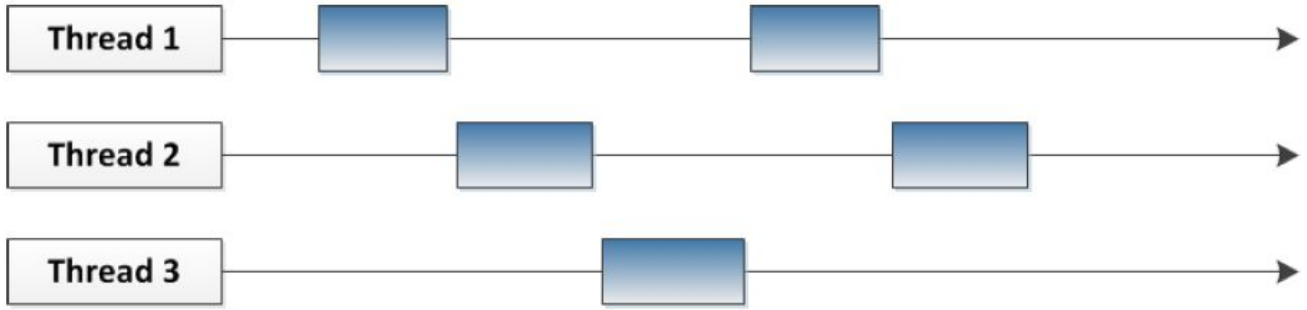
Multithreading & Multitasking: Comparison

- **Multithreading**
  - Threads share the same address space
    - Light-weight creation/destruction
    - Easy inter-thread communication
    - An error in one thread can bring down all threads in process
- **Multitasking**
  - Processes are insulated from each other
    - Expensive creation/destruction
    - Expensive IPC
    - An error in one process cannot bring down another process

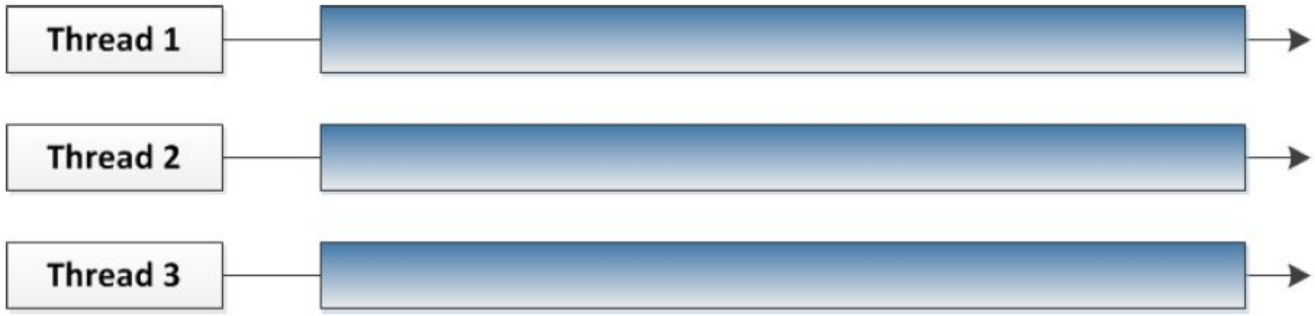
What is a thread?

- A flow of instructions, path of execution within a process
- The smallest unit of processing scheduled by OS
- A process consists of at least one thread
- Multiple threads can be run on:
  - **A uniprocessor (time-sharing)**
    - Processor switches between different threads
    - Parallelism is an illusion
  - **A multiprocessor**
    - Multiple processors or cores run the threads at the same time
    - True parallelism

Multiple threads sharing a single CPU

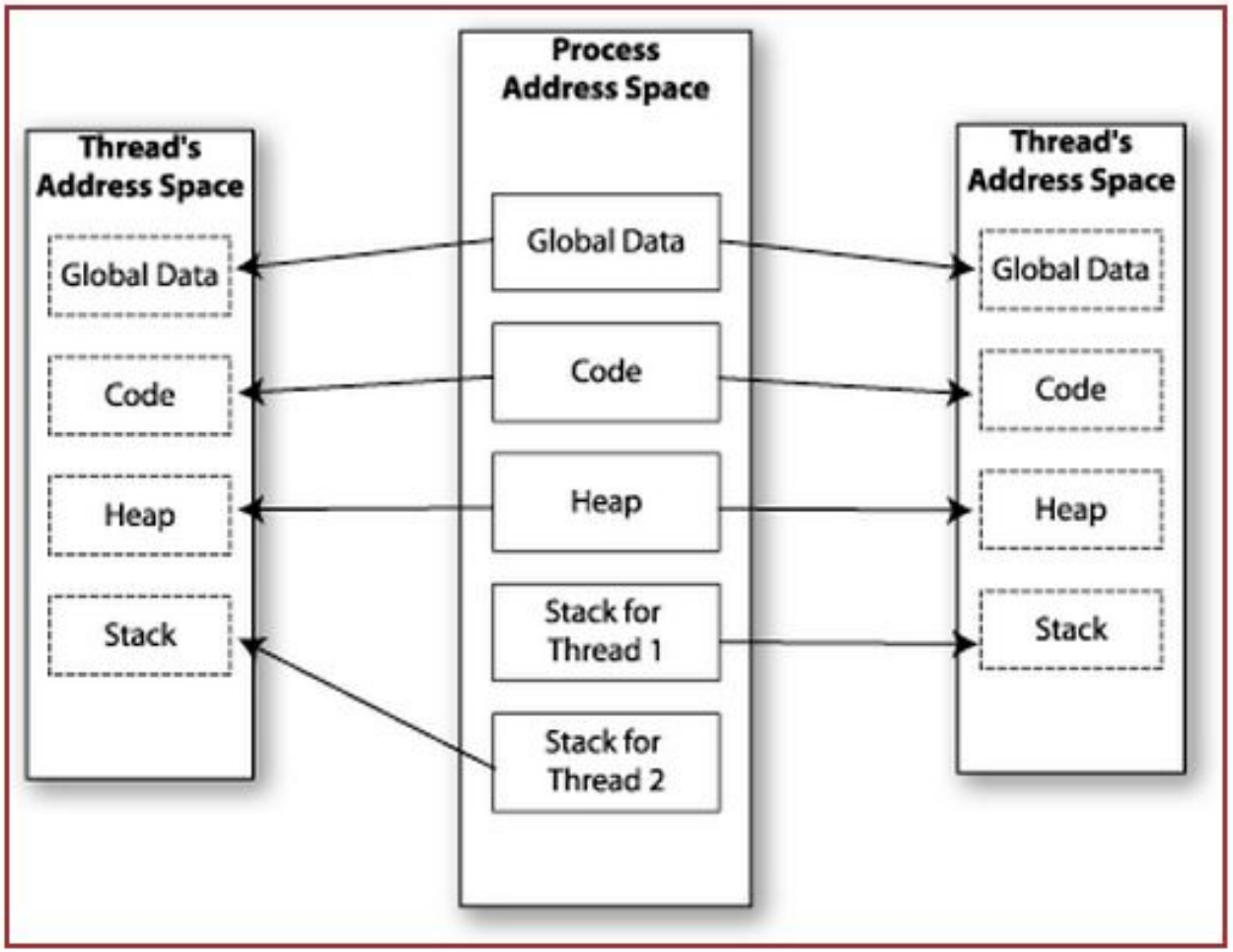


Multiple threads on multiple CPUs



Memory Layout: Multithreaded Program

Why share the same code?
Why share the same heap?



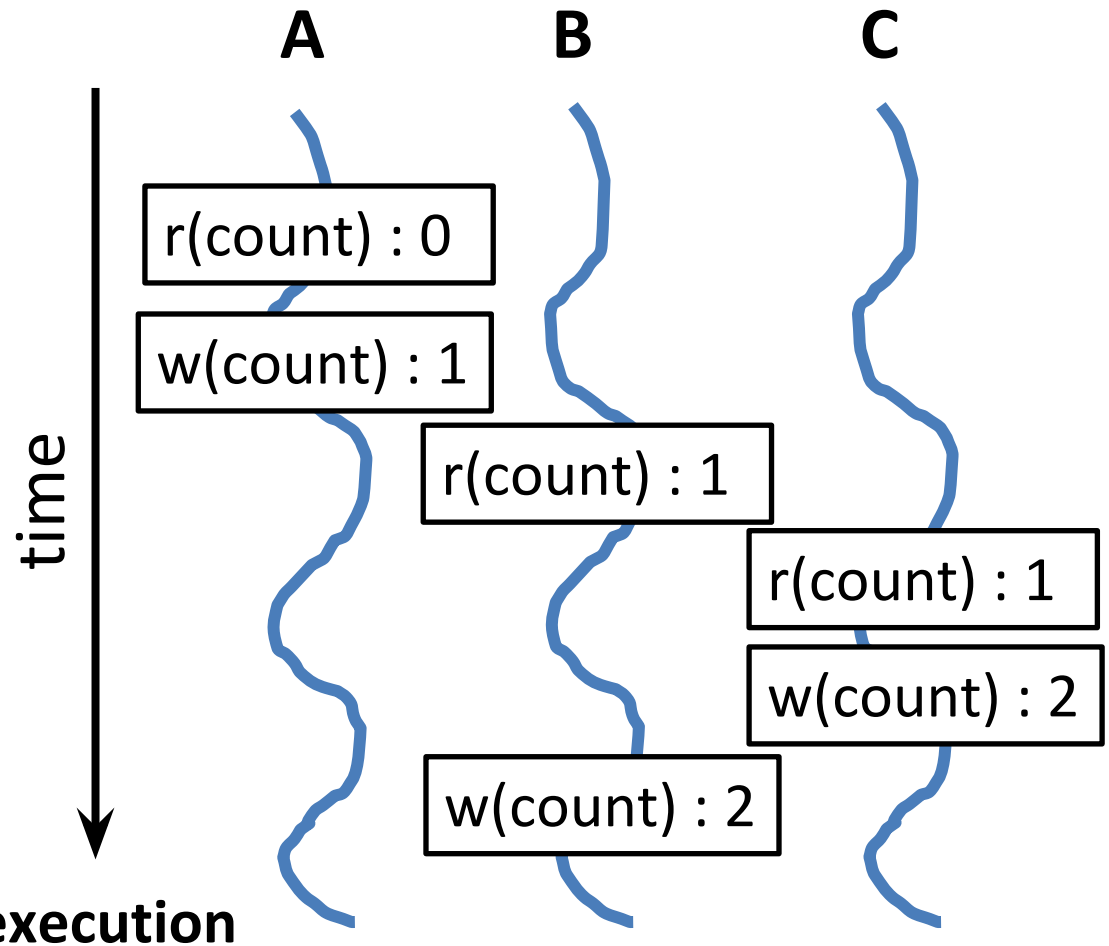
Shared Memory

- Makes multithreaded programming
  - **Powerful**
    - can easily access data and share it among threads
  - **More efficient**
    - No need for system calls when sharing data
    - Thread creation and destruction 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)

What happens when we have race conditions?

Race Condition

```
int count = 0;
void increment()
{
  count = count + 1;
}
```



Result depends on order of execution

=> Synchronization needed

pthread_create

- **Function:** creates a new thread and makes it executable
- Can be called any number of times from anywhere within code
- Return value:
  - Success: zero
  - Failure: error number
- How do we keep track of threads within a program's execution? How many can we have?
- How do we pass data to threads we create? How do we tell them what to work on?
- What happens if our application isn't "embarassingly parallel"?

Parameters

```
int pthread_create( pthread_t *tid, const pthread_attr_t *attr,
                    void *(my_function)(void *), void *arg );
```

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

pthread_join

- **Function:** makes originating 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 chore
- Return value:
  - ⇒ Success: zero
  - ⇒ Failure: error number
- Why join at all? What does a join guarantee?

Arguments

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

- **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

Week 8

Processor Modes

- Operating modes that place restrictions on the type of operations that can be performed by running processes
  - User mode: restricted access to system resources
  - Kernel/Supervisor mode: unrestricted access
- System resources?
  - Memory
  - I/O Devices
  - CPU
- Why have different modes? How do we switch modes?
- Can’t we just set standards for how to use I/O and let everyone have at it?

User Mode vs. Kernel Mode

- Hardware contains a mode-bit, e.g. 0 means kernel mode, 1 means user mode
- User mode
  - CPU **restricted** to unprivileged instructions and a specified area of memory
- Supervisor/kernel mode
  - CPU is **unrestricted**, can use all instructions, access all areas of memory and take over the CPU anytime
- What happens if user code is given unrestricted access to CPU?

How to Achieve Protection and Fairness

- Goals:
  - **I/O Protection**
    - Prevent processes from performing illegal I/O operations
  - **Memory Protection**
    - Prevent processes from accessing illegal memory and modifying kernel code and data structures
  - **CPU Protection**
    - Prevent a process from using the CPU for too long

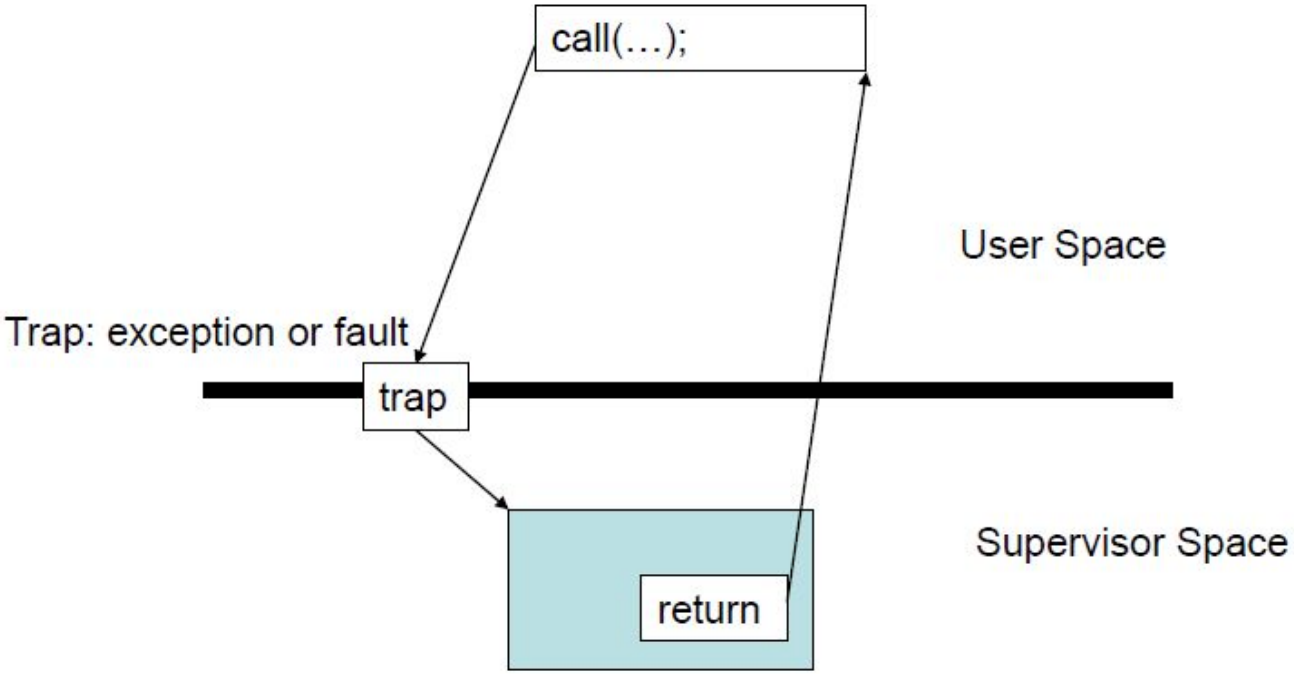
=> instructions that might affect goals are privileged and can only be executed by *trusted code*

System Calls

- Special type of function that:
  - Used by user-level processes to request a service from the kernel
  - Changes the CPU's mode from user mode to kernel mode to enable more capabilities
  - Is part of the kernel of the OS
  - Verifies that the user should be allowed to do the requested action and then does the action (kernel performs the operation on behalf of the user)
  - Is the ***only way*** a user program can perform privileged operations
- When do I need to use system calls?

System Calls

- 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 op.
  - **OS performs requested action**
  - OS restores saved context, switches to user mode
  - OS gives control of the CPU back to user process

Library Functions

- Functions that are a part of standard C library
- To avoid 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
- What are the benefits and tradeoffs of using either system calls or C library functions?

Unbuffered vs. Buffered I/O

- **Unbuffered**
  - Every byte is read/written by the kernel through a system call
- **Buffered**
  - 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
- Buffered I/O decreases the number of read/write system calls and the corresponding overhead

Which is faster in what applications? When would you use buffered or unbuffered I/O?

Week 9

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
- When would I use static linking? Why would I use it?

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
- When would I use dynamic linking? Why would I use it?

Linking and Loading

- Linker collects procedures and links them together object modules into one executable program
- Why isn't everything written as just one **big** program, saving the necessity of linking?
  - Efficiency: if just one function is changed in a 100K line program, why recompile the whole program? Just recompile the one function and relink.
  - Multiple-language programs
  - Other reasons?
- When does linking happen? When does loading happen?

How are libraries dynamically loaded?

Table 1. The DI 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

Dynamic linking

- Unix systems: Code is typically compiled as a *dynamic shared object* (DSO)
- Dynamic vs. static linking resulting size

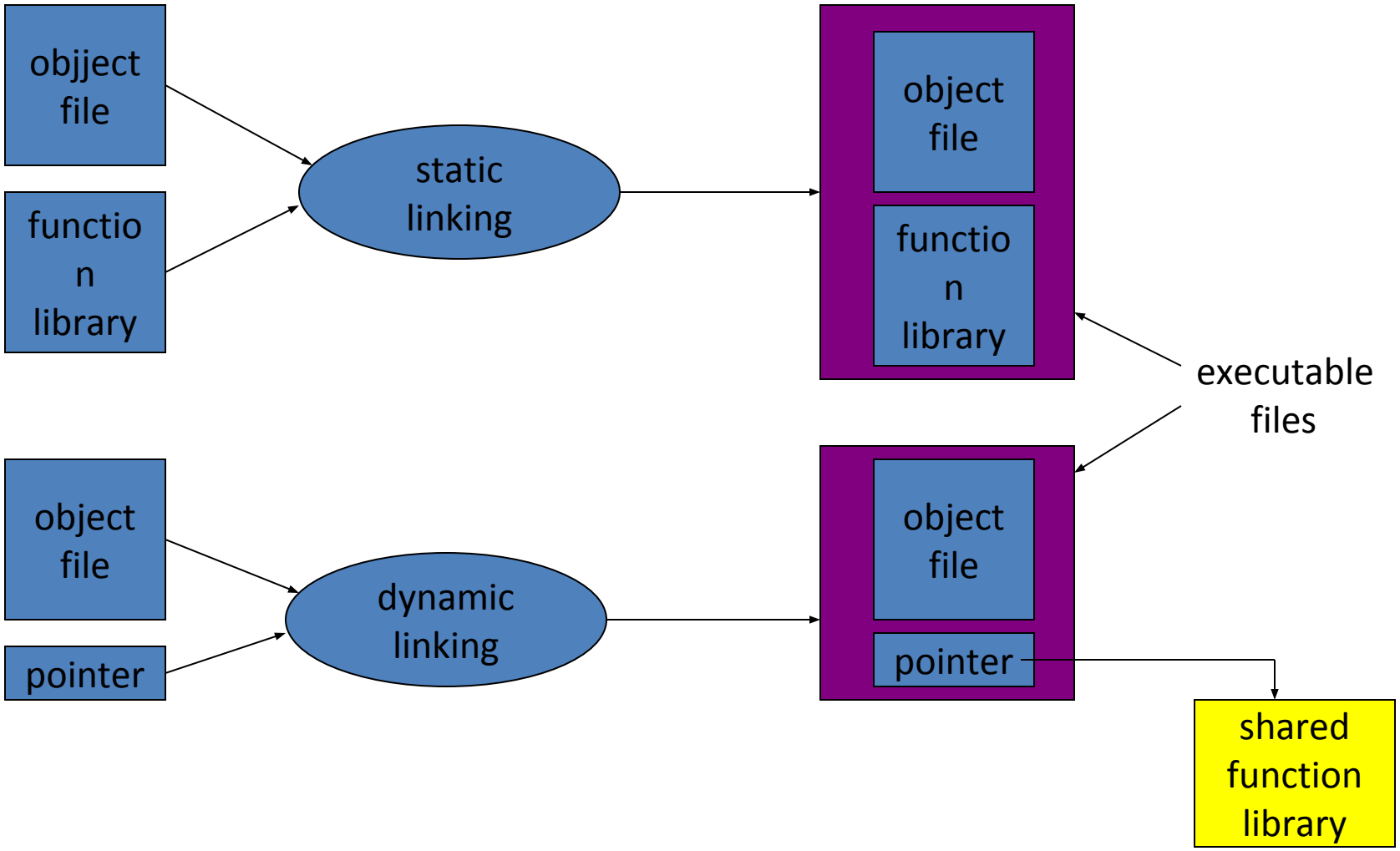
```
$ gcc -static hello.c -o hello-static
$ gcc hello.c -o hello-dynamic
$ ls -l hello
     80 hello.c
  13724 hello-dynamic
    383 hello.s
1688756 hello-static
```

- If you are the sysadmin, which do you prefer?
- What is the difference between linking and loading?

Advantages of dynamic linking

- The executable is typically smaller
- When the library is changed, the code that references it does not usually need to be recompiled
- The executable accesses the .so at run time; therefore, multiple programs can access the same .so at the same time
  - Memory footprint amortized across all programs using the same .so
- What other advantages are there of dynamic linking?

Smaller is more efficient



Disadvantages of dynamic linking

- Performance hit
  - Need to load shared objects (at least once)
  - Need to resolve addresses (once or every time)
  - Remember back to the system call assignment…
- What if the necessary dynamic library is missing?
- What if we have the library, but it is the wrong version?

Announcements - Please Fill teaching evaluation survey

Students have until 8:00 AM Saturday, March 12 to log into MyUCLA to complete evaluations for your courses listed below:

COM SCI 35L section 6

You have the option to use class time for your students to complete their evaluations with smartphones, iPads, and tablets. Please note that not all classrooms are WI-FI enabled.

For more details about this process, please contact your departmental Evaluation Coordinator or visit our website at:

http://www.oid.ucla.edu/assessment/eip

Thank you!
Evaluation of Instruction Program
eip@oid.ucla.edu

Sample Final

Available here: goo.gl/kSdbH2