COMP 206 Final Review

CSUS Helpdesk - Zoé McLennan & Daniel Busuttil
13/12/2018

Outline

- Questions/Concerns
- Review
- Problems
- Questions again!

Before we start...

Questions?

Bash scripting

- You can create and manipulate shell variables: “my_var=someval”
- Remember to start ALL scripts with “#!/bin/bash”
- Bash does not operate on a high level so there exist two ways of indicating things: double quotes (“ “) and back-tick characters (` `). The back-tick indicates to bash to execute that what they contain is a command and to execute it (similar to pre-processing)

Bash wildcards

- There are many special characters in BASH:
  - * -> Matches *everything* (i.e. 0-n many strings)
  - ? -> Matches a single character
  - [abc] -> Matches for the enclosed characters
  - [a-z] -> Matches for the enclosed *range* of characters (using the ASCII table)
  - [!a-z] -> Matches for everything that isn't in the enclosed *range* of characters
- You can do some crazy things with wildcards!
  - https://www.tecmint.com/use-wildcards-to-match-filenames-in-linux/

Bash variables

- Bash has some shell variables that it creates on startup and they’re incredible useful:

PWD	*current working directory*
PATH	*list of places to look for commands*
HOME	*home directory of user*
MAIL	*where your email is stored*
TERM	*what kind of terminal you have*
HISTFILE	*where your command history is saved*
PS1	*the string to be used as the command prompt*

- You can use these in your scripts to great effect!

Bash maths (adv)

- Every shell variable in Bash is a string, if you want to do mathematical operations you need to use “expr”, i.e. “ echo ` expr 3 + 4` “
- Alternatively, you can use $((*computation*))
- If you want to do floating-point arithmetic things can get funky - see https://mathblog.com/floating-point-arithmetic-in-the-bourne-again-shell-bash/
    - Simple example:

```
$ bc <<< 'scale=2; 100/3'

33.33
```

I/O redirection

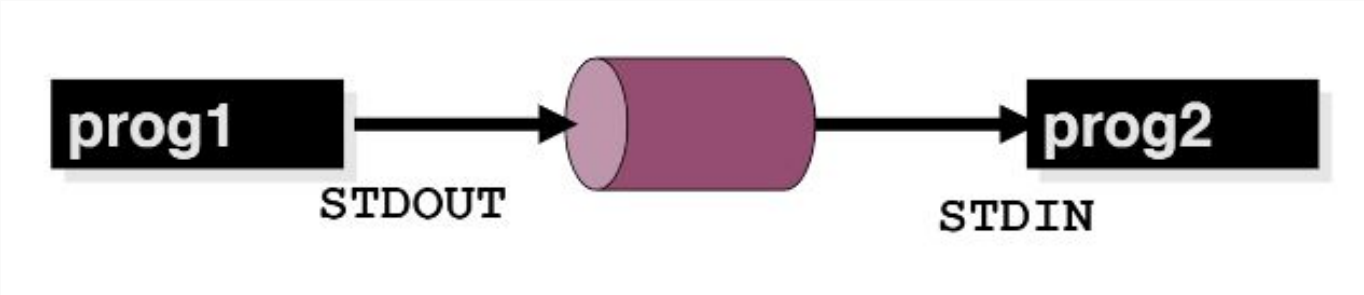
Many things in your shell are *streams* and we can redirect them to different outputs than just your screen (i.e. stdout):

- Output redirection “ > ” operator
  - E.g. $ ls > file_info.txt
- Input redirection “ < ” operator
  - E.g. $ sort < nums.txt

Even do it together: $ sort < nums.txt > sortednums.txt

I/O redirection

- Pipes allow you to ‘connect’ programs by treating the output of one as the input of another:



I/O redirection (advanced)

By default the behaviour of “ > ” is to redirect stdout - but we can choose specific streams:

- “ 1> ”
- “ 2> ”
- “ &> ”

This allows you to splice streams: $ command 2> error.txt 1> output.txt

Grep & command-line arguments

grep [options] string files
- search for occurrences of the string.

`grep` is used to search for the occurrence of a regular expression in files.

Regular expressions, are best specified in apostrophes (or single quotes) when used with grep.

Some common options include:
- -i : ignore case
- -c : report only a count of the number of lines containing matches
- -v : invert the search, displaying only lines that do not match
- -n : display the line number along with the line on which a match was found
- -l : list filenames, but not lines, in which matches were found

$# : number of arguments on the command line

$- : options supplied to the shell

$? : exit value of the last command executed

$$: process number of the current process

$! : process number of the last command done in background

$n : argument on the command line, where n is from 1 through 9, reading left to right
- $0 : the name of the current shell or program
- $* : all arguments on the command line ("$1 $2 ... $9") ~ string
- $@ : all arguments on the command line, each separately quoted ("$1" "$2" . . . "$9") ~ array

Grep example:

- Consider the following text file :

```
Alex
Marc
Micheal
Ting
Juan
Jeremy
Jessica
Yannick
Nicolas
Jean-Sebastien
Nadeem
```

- Grep for specific characters . . .

```
[jvybihal][~/cs206] grep -i '^.e' demo.txt
Jeremy
Jessica
Jean-Sebastien

[jvybihal][~/cs206] grep -i '^.e|a.$' demo.txt
Micheal
Juan
Jeremy
Jessica
Nicolas
Jean-Sebastien
```

'^.[a-e]|a.$'

If statements pt. 1

Bash, like all languages, has its own control flow commands. The most important of these if the “if” statement and the syntax is as follows:

```
if _condition_
then
    _code_
elif _condition_
then
    _code_
else
    _code_
fi
```

In Bash, if-statements will check if ‘_condition_’s evaluate to zero (i.e. their return values) and execute the corresponding code if that is the case. There are many switches and commands that you can use to your advantage, and we’ll look at those next

If statements pt. 2

You can use these commands with the ‘test’ & ‘if’ commands to test for certain conditions, i.e.:

“ if [[$x -lt 4]] “ == “ test $x -lt 4 “

n1 -eq n2 : true if integers n1 and n2 are equal

n1 -ne n2 : true if integers n1 and n2 are not equal

n1 -gt n2 : true if integer n1 is greater than integer n2

n1 -ge n2 : true if int n1 is greater than or equal to int n2

n1 -lt n2 : true if integer n1 is less than integer n2

n1 -le n2 : true if int n1 is less than or equal to int n2

-z string : true if the string length is zero

-n string : true if the string length is non-zero

string1 = string2 : true if strings are identical

string1 != string2 : true if strings not identical

string : true if string is not NULL

-r file: true if it exists and is readable

-w file: true if it exists and is writeable

-x file: true if it exists and is executable

-f file: true if it exists and is a regular file

-d name: true if it exists and is a directory

If statements pt. 2 (adv.)

- “Using the [[...]] test construct, rather than [...] can prevent many logic errors in scripts…”
- https://www.tldp.org/LDP/abs/html/testconstructs.html#DBLBRACKETS

For & While loops

```
for var in list
do
    actions
done
```

```
while condition
do
    actions
    [continue]
    [break]
done
```

Job Control

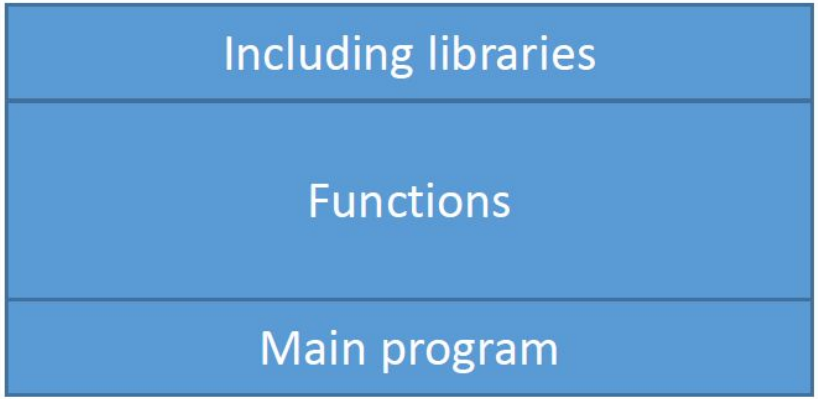
- A shell has the capacity to manage ‘jobs’ which are processes that you run simultaneously; if you finish a command with ‘&’ the shell will run your process and, while it is not finished, it’ll run it concurrently with other processes you have running
- The shell will, appropriately, assign an ID number to each job it runs in the background. You can view these with the command “jobs” and suspend or kill any running jobs with ‘ctrl-Z’ and ‘ctrl-C’ respectively

```
> jobs
[1] Stopped                 ls -lR > saved_ls &
> fg %1
ls -lR > saved_ls
```

C Basics

```c
#include <stdio.h>

int main()
{
        printf( "Hello, world!\n" );
}
```

1. vim program.c
2. gcc program.c
3. ./a.out



Recommended layout

C Basics

Remember that the normal way to define your main method:

```
int main( int argc, char *argv[] ) {

    // code

}
```

- argc : represents the number of command-line variables passed
- argv : character array that contains the whitespace delimited arguments

C Basics: Datatypes

Built-in types:

- int → 16/32 bit integer
- float → 32 bit floating point number
- double → 64 bit floating point number
- char → 8 bit character

Modifiers:

- short/long: 16 vs 32 bit int
- signed/unsigned: positive vs negative
- Pointers: int *, char *

C Basics: Datatypes

Description	Type	Bits	Range
Integer	short int long	16 32 64	+/-32 thousand +/-2.1 billion +- 9.2×10^{18}
Floating point	float double long double	32 64 128	+/- 10^{38} +/- 10^{308} +/- 10^{4932}
Character	char unsigned char	8	-127 to 128 0 to 255
Pointer	char* int* (etc)	64	0 to 1.8×10^{19}

C Basics: control flow

```
if ( COND ) STATEMENT;

if ( COND )
{STATEMENTS;}

if (COND1) {CODE}
else if (COND2) {CODE2}
else {CODE3}
```

```
switch (VAR) {
        case VAL1:   CODE
                     break; // not optional
        case VAL2:   CODE
                     break; // not optional
…
        default:     CODE
}
```

```
While ( COND ) STATEMENT;

while( COND ) { STATEMENTS; }

do STATEMENT; while (COND);

do {STATEMENTS;} while (COND);

for ( START; COND; EXPRESSION ) STATEMENT;

for ( START; COND; EXPRESSION )
{STATEMENTS;}
```

C Basics: Pointers pt. 1

- Pointers are variables which allow you to access (typed) areas of memory
- Incredibly powerful mechanism for us programmers (but also dangerous)
- &VAR -> “get the address of VAR”
- *PTR -> “get data at the address of PTR”
- Allows us to pass parameters either by “pass-by-reference” or “pass-by-value”

C Basics: Pointers pt. 2

- Arrays are - in C - a series of contiguous variables of the array's type with the array variable being a pointer to the first variable, i.e.
  - array[0] == *array
  - array[i] == *(array+ i)
  - &(array[j]) == array + j
- Strings are simply char arrays with a 'terminating' null character: '\0'

The null character is REALLY important!!

C Basics: Pointers pt. 2 (adv.)

- Difference between:
  - char x[];
  - char *x;
  - char *x = “Hello World”;

To learn a bit more about the problem:

- https://www.tutorialspoint.com/cprogramming/c_constants.htm
- https://gsamaras.wordpress.com/code/string-literal-vs-string-in-array-c/

<string.h> - Important functions

size_t strlen(const char *str)
Computes the length of the string str up to but not including the terminating null character.

void *memset(void *str, int c, size_t n)
Copies the character c (an unsigned char) to the first n characters of the string pointed to, by the argument *str*.

int strcmp(const char *str1, const char *str2)
Compares the string pointed to, by *str1* to the string pointed to by *str2*.

char *strcat(char *dest, const char *src)
Appends the string pointed to, by *src* to the end of the string pointed to by *dest*.

char *strcpy(char *dest, const char *src)
Copies the string pointed to, by *src* to *dest*.

char *strstr(const char *haystack, const char *needle)
Finds the first occurrence of the entire string *needle* (not including the terminating null character) which appears in the string *haystack*.

<stdio.h>

- 3 types of I/O
  - Console
    - Keyboard, screen
    - stdin, stdout, stderr
  - Stream
    - Constant stream of data from logical/physical device
  - File
    - Reading or writing to file
- <stdio.h provides built in functions to deal work with I/O

<stdio.h> - Important functions

FILE *fopen(const char *filename, const char *mode)
Opens the filename pointed to by filename using the given mode.

int fclose(FILE *stream)
Closes the stream. All buffers are flushed.

size_t fread(void *ptr, size_t size, size_t nmemb, FILE *stream)
Reads data from the given stream into the array pointed to by ptr.

size_t fwrite(const void *ptr, size_t size, size_t nmemb, FILE *stream)
Writes data from the array pointed to by ptr to the given stream.

long int ftell(FILE *stream)
Returns the current file position of the given stream.

int fseek(FILE *stream, long int offset, int whence)
Sets the file position of the stream to the given offset. The argument *offset* signifies the number of bytes to seek from the given *whence* position.

<stdio.h> - Important functions

int printf(const char *format, ...)
Sends formatted output to stdout.

int fprintf(FILE *stream, const char *format, ...)
Sends formatted output to a stream.

int fputs(const char *str, FILE *stream)
Writes a string to the specified stream up to but not including the null character.

char *fgets(char *str, int n, FILE *stream)
Reads a line from the specified stream and stores it into the string pointed to by str. It stops when either (n-1) characters are read, the newline character is read, or the end-of-file is reached, whichever comes first.

int fputc(int char, FILE *stream)
Writes a character (an unsigned char) specified by the argument char to the specified stream and advances the position indicator for the stream.

int fgetc(FILE *stream)
Gets the next character (an unsigned char) from the specified stream and advances the position indicator for the stream.

fopen()

- FILE *fopen(const char *filename, const char *mode)
- FILE → built in pointer type
- Always check if NULL pointer after opening
- Modes:
    - r → read
    - w → write
    - a → append

fseek()

```c
#include <stdio.h>

int main(){

  FILE* fp;
  fp = fopen( "myfile.txt", "r" );
  fseek( fp, 0L, SEEK_END );
  int sz = ftell(fp);
  rewind(fp);

  char file_data_array[sz+1];
  fread( file_data_array, 1, sz+1, fp );
  printf( "File contents:\n%s\n", file_data_array );

  for( int pos=0; pos<sz+1; pos++ ){
    printf( "String character %d has AASCI value %d.\n", pos, file_data_array[pos] );
  }

  return 0;
}
```

Example

```
#include <stdio.h>
#include <stdlib.h>
void copyFile (FILE *source, FILE *destination) {
    char c;
    while(!feof(source)) {
        c = fgetc(source);
        fputc(c, destination);
    }
}
void main() {
    FILE *s = fopen("letter.txt","rt"), *d = fopen("copy.txt","wt");
    if (s == NULL || d == NULL) exit(1); // terminate with an error code
    copyFile(s, d);
    fclose(s); fclose(d);
}
```

Stack vs. Heap Memory

Stack:

- Static memory allocation
- Good for when you know exactly how much memory you need to allocate before compile time.

Heap:

- Dynamic memory allocation
- Used when you don’t know how much memory you will need before compile time

Heap Memory

- What to do when you don’t know how much memory you’ll need ahead of time?
  - Use dynamically allocated memory
  - Heap memory can allocate and deallocate memory dynamically many times

Request for N bytes of heap memory (not initialized):

```
void *malloc(int numberOfBytes);
```

Request for an array of `N` elements each with `size` bytes, and initializes the values all to 0:

```
void *calloc(int N, int size);
```

Heap Memory

realloc asks for additional memory (size is the new total size, not added to the old request)

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

- Might not have enough space at original address
- Beware, data may be moved to new location
- After using heap memory:
  - free(void *ptr)
  - ptr= NULL;

```
int main(void) {
    int *array;
    int n;

    scanf("%d", &n);                    // notice we define size of array at run-time
    array = (int *) calloc(n, sizeof(int));   // int is 4 bytes, can replace sizeof with 4
    if (array == NULL) exit(1);

    *(array+2) = 5;                     // notice how we access data in array
    printf("%d", *(array+2));
    free(array);

    return 0;
}
```

Heap Memory

- Common Errors:

Mismatch between sizes:

- int *pi = (int*)malloc(10*sizeof(char));

Not casting to pointer:

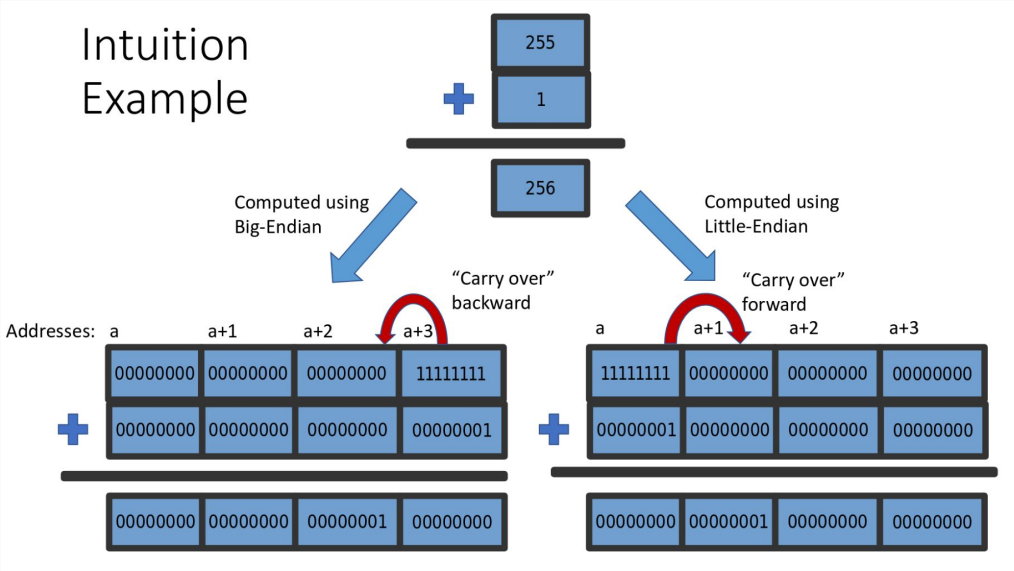
- int i = (int)malloc(sizeof(int));

Forgetting sizeof the datatype:

- int *my_array = (int*)malloc(10);

Endianess

- Order that bytes within an integer are stored in memory
- Little:
  - The least significant comes first
  - Most systems use this
- Big:
  - The most significant comes first
  - Humans use this



Bit-wise Operations

Shifts:

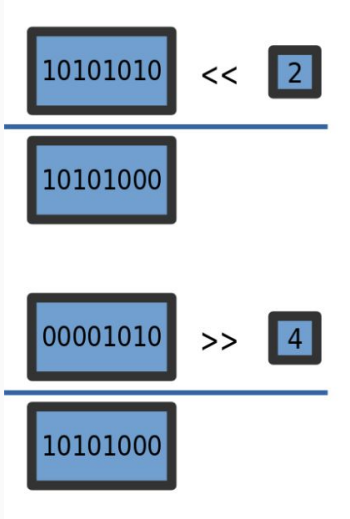
- bit_arg<<shift_arg
  - Shifts bits to of bit_arg shift_arg places to the left -- equivalent to multiplication by 2^shift_arg
- bit_arg>>shift_arg
  - Shifts bits to of bit_arg shift_arg places to the right -- equivalent to integer division by 2^shift_arg

Bit-wise logic:

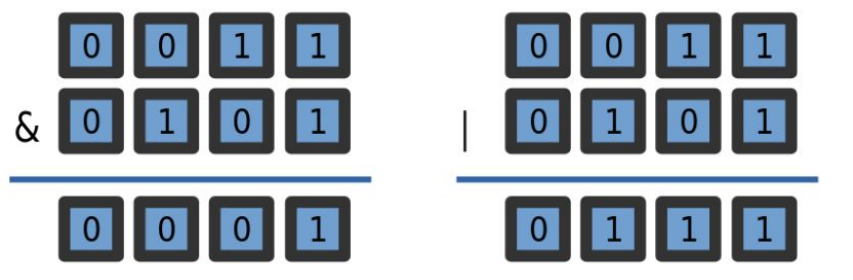
- left_arg & right_arg
  - Takes the bitwise AND of left_arg and right_arg
- left_arg | right_arg
  - Takes the bitwise OR of left_arg and right_arg
- left_arg ^ right_arg
  - Takes the bitwise XOR of left_arg and right_arg (one or the other but not both)
- ~arg
  - Takes the bitwise complement of arg

Bit-wise operations

Shift



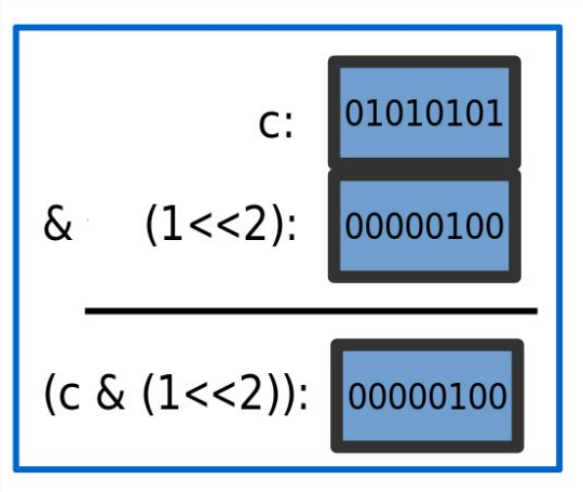
Logical



Bit-wise operations

Check the value of bit 3:

```
char c = 85;
if( (c & (1<<2)) > 0 )
    printf( “bit 3 of c is a 1!\n” );
else
    printf( “bit 3 of c is a 0!\n” );
```



Debugging: Method 1

- printf everything!
  - printf(STRING, OPTIONAL_ARGUMENTS)
    - %d → integer
    - %f → float
    - %c → character
    - %s → string
- Ex: **printf(“comp %d review session”, 206);**
- Pros: know exactly what’s going on at all times
- Cons: tedious and does not always work
  - Effects timing and program behaviour

Debugging: Method 2

- Use the pre-processor!
  - Before gcc compiles your code, the pre-processor will process it and make a .i file
  - Interacts with “#” directives, each having a special meaning
    - #include → import libraries or other .c files to our file
    - #define → allows you to define terms not included in C language
      - #define SYMBOL VALUE
      - Ex: #define TRUE 1
      - #undef removes SYMBOL from pre-processor list
    - #ifdef/#ifndef
      - #ifdef TERM → if term was already defined true, else false
      - #ifndef TERM → if term was not already defined true, else false
    - #else → used wit #ifdef and #ifndef
    - #endif → ends conditional block

Debugging: Method 2

- Example:

```
#ifdef DEBUG
	#define debug(x,y) printf(x,y)
#else
	#define debug(x,y) /*x,y*/
#endif
```

- In code → debug(“The value of a is %d.\n”, a);
- Use flag -D<SYMBOL> to interact with pre-compiler
  - gcc -DDEBUG ...

Debugging: Method 2

- Variadic Macro Function
  - Allows you to have printf statements of variable length
    - Before:

      #define debug(x,y) printf(x,y)

      ...

      debug(“The value of a is %d.\n”, a);
    - After:

      #define debug(...) printf(__VA_ARGS__)

      ...

      debug(“a,b,c,d = %d, %d, %d, %d. \n”, a,b,c,d);

Debugging: Method 3

- GDB Debugger
  - GNU debugger, free and available as part of gcc
  - Has key commands that allow more precision and control than other methods
  - Starting gdb:

    gcc -g -o helloworld helloworld.c

    gdb helloworld
  - Common commands:
    - run <args>, **break <line# or funct>**, list, backtrace, **print <var>**, disp <var>, **next**, nexti, finish
    - help command for more

Multiple-file Projects and Libraries

- Managing a program composed of multiple files
- Use header files!
  - Include the things you want to share among files
  - .h file usually has the same name as corresponding .c file
  - Has names and types but no implementation → signature
  - #include it in any files that want to use that info
  - Only list .c files when compiling with gcc, and there will no longer be a warning

File: swap.h

```
void swap( int *a, int *b);
```

File: swap.c

```
void swap( int *a, int *b){
  int temp = *a;
  *a = *b;
  *b = temp;
}
```

File: main.c

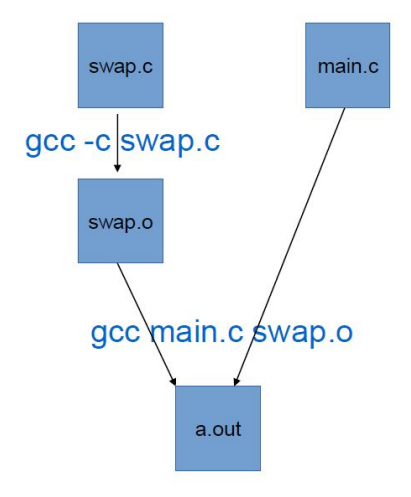
```
#include “swap.h”
void main(){
  int a = 5;
  int b = 6;
  swap( &a, &b);
}
```

Compiling vs Linking

- It is a waste of time to recompile libraries repeatedly (especially if they re complex)
- Instead of re-compiling → use linker

  gcc -c swap.c → swap.o
  gcc main.c swap.o

- If you make changes to .c file, must recompile



Libraries

- Can be imported into your code using #include (linker)
- Common libraries include:
  - Char (ctype.h)
  - File I/O (stdio.h)
  - Math (math.h)
  - Dynamic memory (stdlib.h)
  - String (string.h)
  - Time (time.h)
- Two types to focus on:
  - Static → .a (copied by linker into executable)
  - Shared → .so (loaded dynamically as needed, remain in separate files)

Libraries

- Create/Use Static Library
  - gcc -c swap.c
  - ar rcs libswap.a swap.o
  - gcc main.c libswap.a
- Create/Use Shared Library
  - gcc -shared -fpic -o libswap.so swap.c
  - gcc main.c -lswap -L
  - Set library path:
    - Export LD_LIBRARY_PATH=${LD_LIBRARY_PATH}
    - Environment variable
- Tools:
  - ldd (print shared object dependencies), nm (list symbols from object files)

STRUCT

- Allows you to compose primitive elements into a single complex structure

```
struct OPTIONAL_NAME {
        FIELDS;
}OPTIONAL_VAR_NAME;
```

- OPTIONAL_NAME → user defined type name
- OPTIONAL_VAR_NAME → variable containing structure
- FIELDS → semicolon separated variable declarations
    - TYPE1 VAR1; TYPE2 VAR2; ….

STRUCT

- Dot operator:
  - Used to access the fields within the struct

```
struct PERSON a;

scanf("%s", a.name);
scanf("%d", &(a.age));
a.salary = 50.25;

printf("%s %d %f", a.name, a.age, a.salary);
```

```
#include <stdio.h>
struct PERSON { char name[30]; int age; float salary; } a, b, c; // 3 variables
int main() {
    printf("Enter the name for person a: ");
    scanf("%s", a.name);
    printf("Enter the age: ");
    scanf("%d", &(a.age));
    printf("Enter the salary: ");
    scanf("%f", &(a.salary));
    // repeat the above for b and c
    printf("Enter the name for person b: ");
    scanf("%s", b.name);
}
```

Array of Struct

```
#include <stdio.h>
struct PERSON { char name[30]; int age; float salary; } ;
int main() {
    struct PERSON people[100];
    for(int x=0; x<100; x++) {
            scanf("%s", people[x].name);
            scanf("%d", &(people[x].age));
            scanf("%f", &(people[x].salary));

            if (people[x].age == 20) printf("Hey you are 20!!\n");
    }
}
```

Vybihal

Struct - Malloc

- Notice new syntax to access fields when dealing with pointers!
- Only applicable to structs

```
struct STUDENT {
    int age;
    float GPA;
};

struct STUDENT *x;

x = (struct STUDENT *) malloc(sizeof(struct STUDENT));
if (x == NULL) exit(1);

// two ways to access the contents
(*x).age = 5;
x->age = 5; // this is more common
```

Struct - Malloc

- fscanf:

```
struct PERSON x; // variable

FILE *p = fopen(“something.txt”,”rt”);

fscanf(p,”%d”, &x.age);
```

- fread

```
struct PERSON x; // variable

FILE *p = fopen(.....);

fread(x,1,sizeof(struct PERSON), p);
```

Struct - typedef

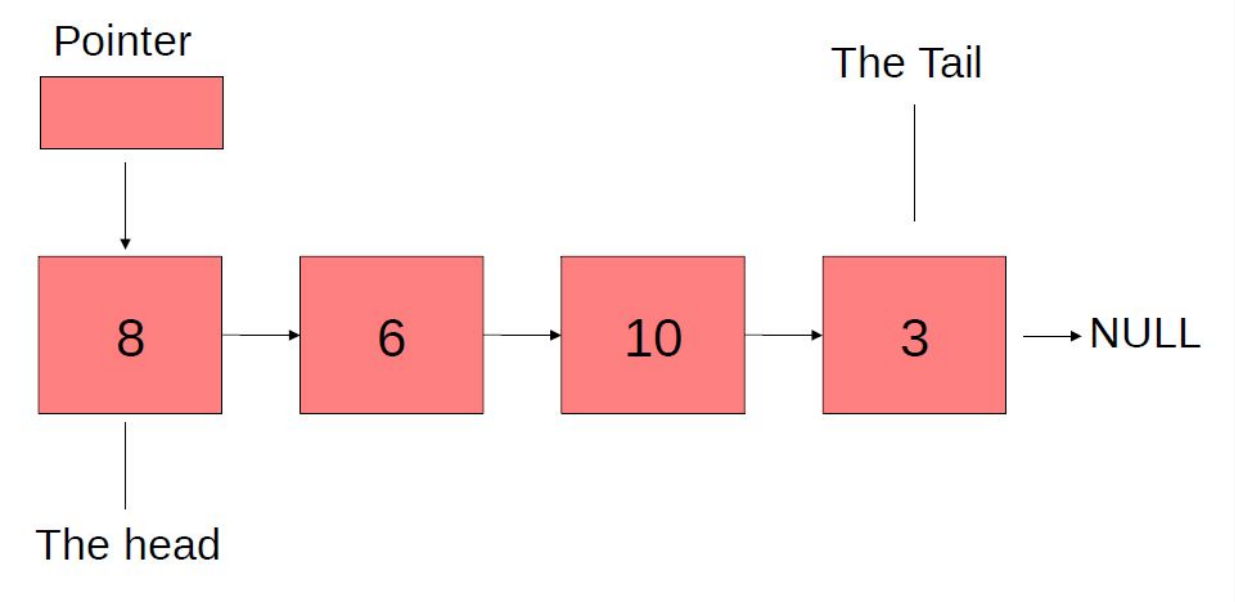
- typedef → renames types
- This is just a trick to not have to retype “struct TYPE” everytime you declare a new TYPE
- MYCOURSE cs206 instead of struct COURSE cs206

```
typedef struct COURSE {
    int number_of_students;
    char name_professor[100];
    char location_building[100];
    int location_room;
} MYCOURSE;
```

Struct - Linked Lists

- Every node has two fields:
  - Value (data)
  - Pointer to next node

```
Struct NODE {
        int value;
        struct NODE *next;
};
```

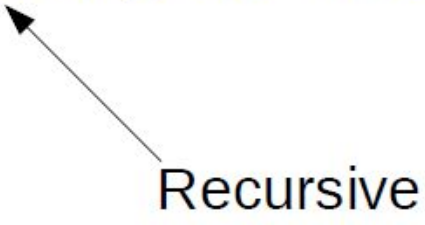


Struct - Linked Lists

```
void addToLinkedList(aNode** list, int value) {

    aNode *newNode = (aNode *)malloc(sizeof(aNode));
    newNode->value = value;
    newNode->next  = NULL;

    aNode* freeSpot = *list;

    if( freeSpot == NULL ) *list = newNode;
    else {
        while(freeSpot->next != NULL) {
            freeSpot = freeSpot->next;
        }
        freeSpot->next = newNode;
    }
}
```

Struct - Linked Lists

```
void printNodes(aNode* my_node) {

        printf(" %i ", my_node->value);
        if (my_node->next != NULL) {
                printNodes(my_node->next);
        }
}
```



Pointers to functions

- int *fn() ; → function returns int pointer
- int (*fn) (); → pointer to a function that returns an integer

The second creates a function pointer, which can be used to "point to" any existing function that matches it's return and argument types, like this:

```
int return5() { return 5; }
int (*fn)() = return5;
```

Afterwards, the word fn is a valid way to execute the code in the return5 function (until fn might be pointed somewhere else later, which is OK):

```
int x = (*fn)();    // x will equal 5
int y = return5(); // y will return 5
int z = fn();       // this syntax is fine too.
```

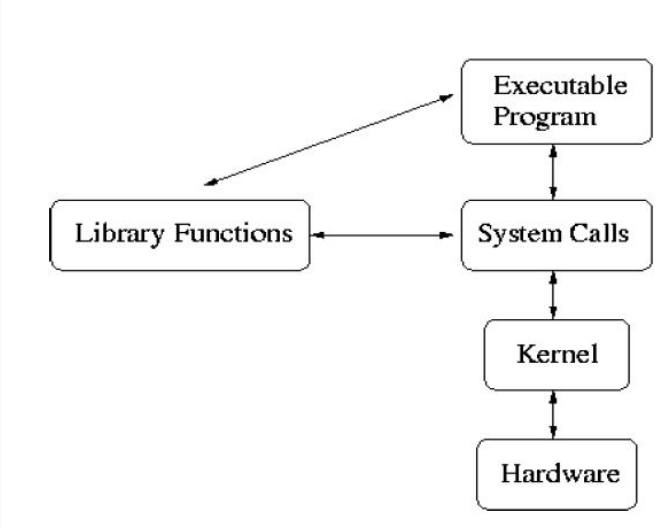
Pointers to functions

- Allows you to use the function in different contexts
- Function pointers as a field in a struct → like a **class** in oop (Java)

```
struct classlike{
   int a;
   int b;
   int (*compare)( void*, void* );
}

struct classlike my_fake_class;
my_fake_class.compare=compare_strings;
my_fake_class.compare( "hello", "world" );
```

System Calls

- Everything you've seen so far use the library functions that live in the C standard library or imported from other libraries (more convenient)
- These functions make system calls for you
- However, direct system calls exist
  - Library → System
    - malloc → sbrk
    - fopen → unlink (delete files), access (permissions)
    - fread/fwrite → read and write



Make

- Allows you to compile only the files to which you have made changes
- Becomes very useful when working as part of a team on a large scale project
- Need to create a **Makefile** →

```
program: file1.o file2.o
  gcc –o program file1.o file2.o
file1.o: file1.c
  gcc –c file1.c
file2.o: file2.c
  gcc –c file2.c
backup:
  cp *.c /home/project/backup
```

This means:
- gcc –c file1.c file2.c
- gcc –o program file1.o file2.o

It can also mean:
- cp *.c /home/project/backup

Some power exists in selectively executing portions of the makefile.

Make

- MACROS:
  - Allows you to save time and not have to write everything out
  - Think of macros as variables
  - Can use them in gcc line or in targets

    name=text_string

    ...

    ${name}

Make

```
SRCS=foo.c bar.c
CFLAGS=-Wall -ansi
LDFLAGS=-lm -lmylib
INCDIR=-I/home/ericb/include
LIBDIR=-L/home/ericb/lib
```

Example command in `make`:

```
gcc ${CFLAGS} ${INCDIR} -o foo ${SRCS}\
  ${LIBDIR} ${LDFLAGS}
```

Make

- String substitution

```
SRCS = defs.c redraw.c calc.c
OBJS = ${SRCS:.c=.o}
```

Same as:

```
OBJS = defs.o redraw.o calc.o
```

Make

- Suffix Rules
  - file.o: file.c → .c.o:
  - Compiles all .c files to .o files
  - Saves a lot of time

```
.c.o:
	${CC} ${CFLAGS} ${INCDIR} -c $<
```

```
SRCS=foo.c bar.c
OBJS=foo.o bar.o barbar.o
CFLAGS=-Wall -ansi
LDFLAGS=-lm -lmylib
INCDIR=-I/home/ericb/include
LIBDIR=-L/home/ericb/lib

all: ${OBJS}
  ${CC} -o foo ${OBJS} ${LIBDIR} ${LDFLAGS}
foo: ${OBJS}
  ${CC} -o foo ${OBJS} ${LIBDIR} ${LDFLAGS}
clean:
   /bin/rm -f ${OBJS}
install:  foo
   /bin/cp -f foo /usr/local/bin

.c.o:
    ${CC} ${CFLAGS} ${INCDIR} -c $<
```

Multiple Processes with C

Use fork to launch multiple processes

Use each process' PID to distinguish during control flow

```
#include <stdlib.h> /* needed to define exit() */
#include <unistd.h>/* needed for fork() and getpid() */
#include <stdio.h>  /* needed for printf() */

int  main(int argc, char *argv[]) {
        int pid;    /* process ID */

        switch (pid = fork()) {
        case 0:                 /* fork returns 0 to the child */
                printf("I am the child process: pid=%d\n", getpid());
                break;

        default:   /* fork returns the child's pid to the parent */
                printf("I am the parent process: pid=%d, child pid=%d\n", getpid(), pid);
                break;

        case -1:   /* something went wrong */
                perror("fork"); // send string to STDERR
                exit(1);
        }
        exit(0);
}
```

Programs can use fork() as often as needed.
A process invoked by fork() can also use fork().

Problems

- Race conditions
  - Two threads trying to execute on the same resource at the same time
  - OS controls scheduling of threads, so developer does not know the order in which the threads will execute
  - Threads are competing against each other to execute
- Deadlocks
  - Two threads, each one depends on the actions of the other, and both are waiting for the other to complete an action
  - Execution is stalled

Producer/Consumer Problem (Scheduling)

The producer

- A program that generate data saving that data in a data structure or in a file.

The consumer

- A program that reads data from a data structure or a file performing some kind of computation.

Concurrency

- For some reason the producer and consumer need to run independently but cooperatively.
  - Maybe the computation is slow but the data comes in quickly. The intermediate data structure or file is used as a temporary cache.

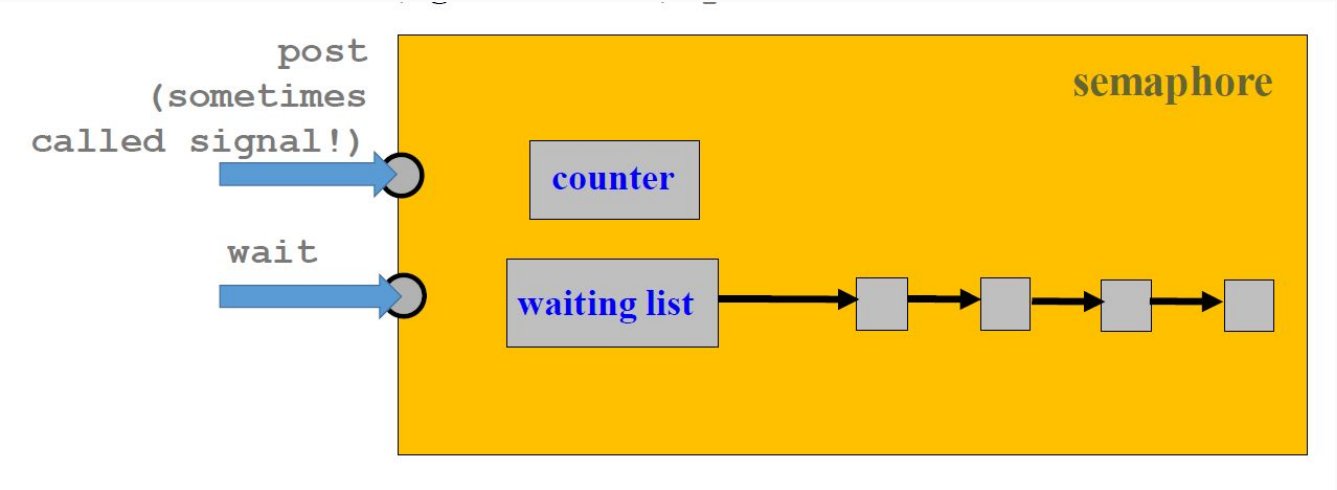
Semaphores - Solution

- Central agent keeps track of who asked to execute first and only lets one process run at a time
- First to ask is first to execute
- Processes are executed one at a time
- Process must tell agent when it is done executing
- Library example

Semaphores

Data structure equipped with:

- Counter
- Waiting list
- 2 functions:
  - post()
  - wait()



Semaphores

```
void wait(sem S)
{
    S.count--;
    if (S.count < 0) {
        add the caller to the waiting list ;
        block();
    }
}
```

```
void post(sem S)
{
    S.count++;
    if (S.count <= 0) {
        remove a process P from the waiting list;
        resume( P);
    }
}
```

Semaphores

- **MutEx** → locking mechanism, locks access to a resource so that once process at a time can use it.
- **Count-down counter** → limits the number of processes that are waiting until a process calls post()
- **Notification** → Notifies other processes when they are done so they can take their turn to execute

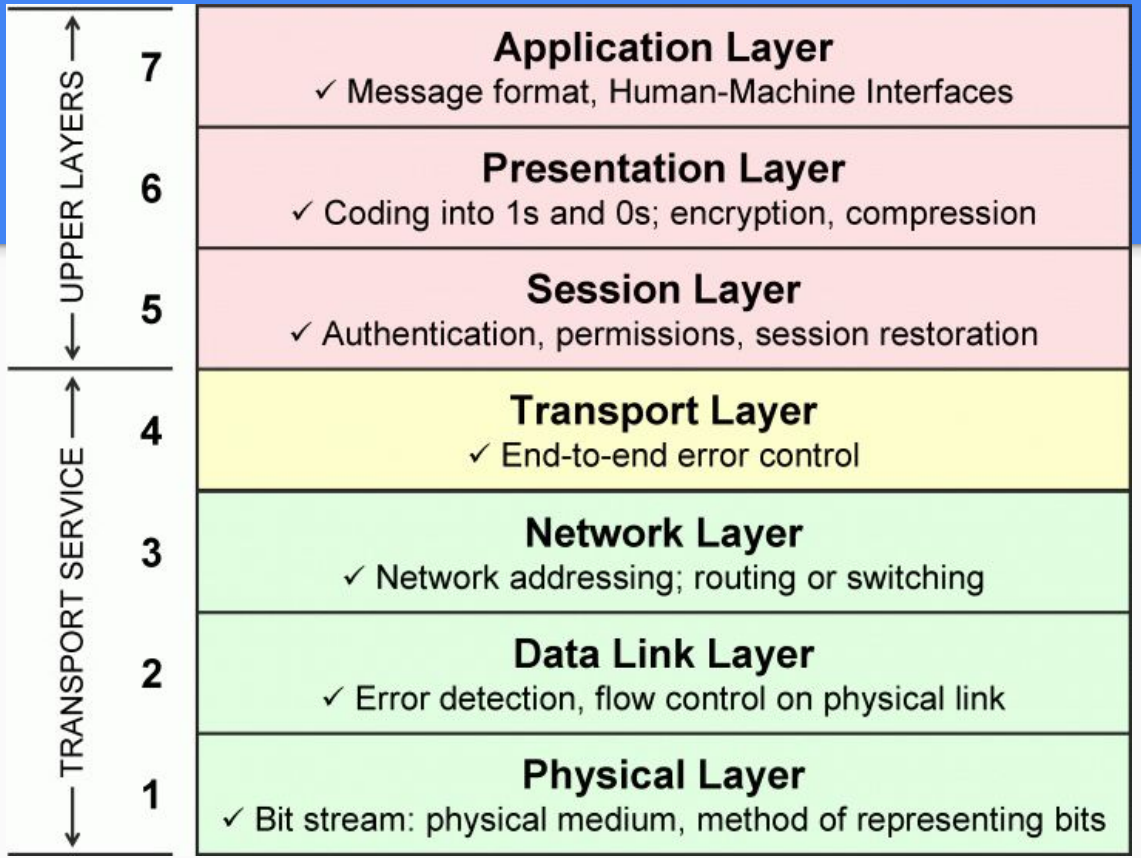
Socket Communication

What is a **Socket**?

- It connects two computers over a network
- It is composed of:
  - A data structure
  - A network
  - A communication algorithm
- Example: Typing www.google.com in your browser causes your browser to create a socket between you and Google so it can send and receive data

OSI model

- 1 → 7 is source to destination
- 206 handles layers 3-5



Network Layer

- Bundles data into packets with specific formats
  - IP addresses of source and destination
    - 32-bit for IPv4 and 128-bit for IPv6
  - Message length
  - Time-to-live
  - Header checksum

Transport Layer: TCP

- Transmission control protocol
- Provides a reliable, ordered and error-checker delivery of a stream of bytes between applications communicating through an IP network.
- Important tools:
  - Sequenced numbers
  - Acknowledgements
  - Re-transmission
  - Congestion control
- Port to Port communication

High-Level Layers

Protocols we use everyday:

- HTTP → HyperText Transfer Protocol (Port 80)
- SSH → Secure shell
- FTP → File Transfer Protocol
- SMTP → Email
- DHCP → Obtaining address on the network
- DNS → Looking up computers with specific names

Sockets in C

- Most of the time, you can treat a socket the same way you treat a file
- Functions:

socket() : Create a socket, specify it's protocol type (AF_INET)

bind() : Specify the coordinates: address and port number

listen() : Wait for someone to respond, the OS does all the checking

accept() : If someone is there, acknowledge and start the comms

read() : Just like a file, put data into the socket

write() : Just like a file, get data out of the socket

Socket Data Structure

```
// IPv4 AF_INET sockets:
struct sockaddr_in {
    short            sin_family;   // e.g. AF_INET, AF_INET6
    unsigned short   sin_port;     // e.g. htons(3490)
    struct in_addr   sin_addr;     // see struct in_addr, below
    char             sin_zero[8];  // zero this if you want to
};

struct in_addr {
    unsigned long s_addr;          // load with inet_pton()
};

struct sockaddr {
    unsigned short    sa_family;    // address family, AF_xxx
    char              sa_data[14];  // 14 bytes of protocol address
};
```

```c
/*
 * socket demonstrations:
 * This is the server side of an "internet domain" socket connection, for
 * communicating over the network.
 */

#include <stdio.h>
#include <unistd.h>
#include <string.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>

int main()
{
    int fd, clientfd;
    int len;
    socklen_t size;
    struct sockaddr_in r, q;
    char buf[80];

    /* "AF_INET" says we'll use the internet */
    if ((fd = socket(AF_INET, SOCK_STREAM, 0)) < 0) {
        perror("socket");
        return(1);
    }

    /* Specify port number. Note, we do not give our address here because
       the server's job is to listen. The whole computer has an address, so
       this is the only one we can listen on. */
    memset(&r, '\0', sizeof r);
    r.sin_family = AF_INET;
    r.sin_addr.s_addr = INADDR_ANY;
    r.sin_port = htons(1234);

    /* Bind connects the socket, tells the OS we're ready to use it. */
    if (bind(fd, (struct sockaddr *)&r, sizeof r) < 0) {
        perror("bind");
        return(1);
    }

    /* Listen blocks until a client tries to connect */
    if (listen(fd, 5)) {
      perror("listen");
      return(1);
    }

    /* Accept says, OK, let's talk! */
    size = sizeof q;
    if ((clientfd = accept(fd, (struct sockaddr *)&q, &size)) < 0) {
        perror("accept");
        return(1);
    }

    /* Read is now much like a file, we can keep grabbing data that's sent. */
    if ((len = read(clientfd, buf, sizeof buf - 1)) < 0) {
        perror("read");
        return(1);
    }
    buf[len] = '\0';
    /*
     * Here we should be converting from the network newline convention to the
     * unix newline convention, if the string can contain newlines. Ignoring for
     * now to keep it simple.
     */

    printf("The other side said: %s\n", buf);

    /* We're done listening to this client. */
    close(clientfd);

    /*
     * We didn't really have to do that since we're exiting.
     * But usually you'd be looping around and accepting more connections.
     */

    return(0);
}

```

DNS: Domain Name System

- Name to IP address mapping
- Allows you to type in http://google.com instead of http://172.217.14.174
- DNS Dataflow:
  - Client's query is sent
  - DNS server uses a tree structure to determine mapping
  - Then asks host server for permission to access
  - Receives response
  - Transmits response to client

WWW

World Wide Web is one of the most pervasive application-level software systems on the internet!

Some important protocols:

- HTTP → HyperText Transmission Protocol (request and interpret data)
- HTML → HyperText Markup Language (building blocks of web pages)
- CGI → Common Gateway Interface (execution of console applications)

HTTP

- Used to structure requests and responses so that both sides can understand
  - Browser → HTTP client
  - Web server → HTTP server
  - Transmit *resources*
- Use TCP/IP and socket to transmit formatted plain-text payloads
  - Client establishes connection over predefined port (80 for http)
  - Server parses request and responds

HTTP

Request

- Method (**GET,** POST, PUT, DELETE)
- URI (Part of URL after address and port)
- Protocol version
- Header

Response

- Response code, response text, header and data
- Common response codes →

200: OK
401: Unauthorized
403: Forbidden
404: Not Found
500: Internal Server Error

HTTP

Method, URI, Protocol Version →

Protocol Version, Response Code, Response text →

Rest is part of the headers

```
Client request:

  GET /hello.txt HTTP/1.1
  User-Agent: curl/7.16.3 libcurl/7.16.3 OpenSSL/0.9.7l zlib/1.2.3
  Host: www.example.com
  Accept-Language: en, mi


Server response:

  HTTP/1.1 200 OK
  Date: Mon, 27 Jul 2009 12:28:53 GMT
  Server: Apache
  Last-Modified: Wed, 22 Jul 2009 19:15:56 GMT
  ETag: "34aa387-d-1568eb00"
  Accept-Ranges: bytes
  Content-Length: 51
  Vary: Accept-Encoding
  Content-Type: text/plain

  Hello World! My payload includes a trailing CRLF.
```

HTML

- Formatted Text
- Made up of tags with content and arguments

<html></html> encloses the full document
<body></body> encloses the main content of the page
<a> encloses information about a link

- All elements are loaded through HTTP requests
- A computer is running persistent server code and waiting for browsers to connect
- This server holds the HTML data

Web Browser in C

1. Initiate socket → socket()
    - Provide address and port number
2. Connect to server → connect()
3. HTTP request → “GET /HTTP1.1”
4. Write to socket → write()
5. Wait (Loop)
6. Get response from socket → read()

A simple server

```
int main(int argc, char *argv[]) {
   int listenfd = 0, connfd = 0;
   struct sockaddr_in serv_addr;

   char sendBuff[1025];
   time_t ticks;

   listenfd = socket(AF_INET, SOCK_STREAM, 0); // AF_INET=IPv4, SOCKET_STREAM=TCP, 0=auto protocol
   memset(&serv_addr, '0', sizeof(serv_addr));
   memset(sendBuff, '0', sizeof(sendBuff));

   serv_addr.sin_family = AF_INET;                          // Accept IPv4 addresses
   serv_addr.sin_addr.s_addr = htonl(INADDR_ANY);           // Accept any address
   serv_addr.sin_port = htons(5000);                        // communication through port 5000

   bind(listenfd, (struct sockaddr*)&serv_addr, sizeof(serv_addr)); // assign values to socket

   listen(listenfd, 10); // permit maximum of 10 users on this socket

   while(1) {                                               // servers never stop working…
      connfd = accept(listenfd, (struct sockaddr*)NULL, NULL);  // server sleeps until a connection made
                                                            // connfd= is client socket ID
      ticks = time(NULL); // this server program simply return the time to the client socket (get time)
      snprintf(sendBuff, sizeof(sendBuff), "%.24s\r\n", ctime(&ticks)); // (format time as string)
      write(connfd, sendBuff, strlen(sendBuff));                // (send string to client)

      close(connfd);     // close connection with client… work is done.
      sleep(1);          // give a chance for other programs on server to run…
   }
}
```

Review Questions!

Question 1:

- What prints?
  a. A really long value
  b. 10
  c. Compilation error
  d. Segmentation fault

```c
#include <stdio.h>
void foo(int*);
int main()
{
    int i = 10, *p = &i;
    foo(p++);
}
void foo(int *p)
{
    printf("%d\n", *p);
}
```

Review Questions!

Answer

→ B

```c
#include <stdio.h>
void foo(int*);
int main()
{
    int i = 10, *p = &i;
    foo(p++);
}
void foo(int *p)
{
    printf("%d\n", *p);
}
```

Review Questions!

Question 2:

- Write a makefile that for a program called “tutorial” .
- It contains the following files: f1.c f1.h f2.c f2.h
- f1.c includes f2.h
- f2.c includes f1.h

Review Questions!

Answer:

```
tutorial: f1.o f2.o
	gcc -o tutorial f1.o f2.o
f1.o: f1.c f2.h
	gcc -c f1.c
f2.o: f2.c f1.h
	gcc -c f2.c
```

Review Questions!

Question 3:

- Assume NODE *p points to a with the following fields
  - int data;
  - NODE *next;
- Write the one line C statement to print the data field to the screen using pointer p

Review Questions!

Answer:

printf(“%d”, p → data);

Review Questions!

Question 4:

Assume that in your current directory there is a file called “passwords.txt”. Write a script that takes in 1 cmd-line argument and checks if it exists (ignoring case) inside the text file.

Review Questions!

Answer:

```
if [ (echo $1 | grep -c -i) -ge 1 ]
then
        Echo “Found it!”
else
        Echo “I couldn’t find it”
fi
```

Review Questions!

Answer:

a) cd project file → git init
b) git commit → git pull → (resolve conflict) → git commit → git push
c) Git branch feature → git checkout feature → git commit → git checkout master → git merge feature → git commit → git push
d) Write useful commit messages, delete unused branches, use .gitignore, peer review before merging to master branch

Review Questions!

Question 6.1:

- Make a struct that represents a student. Each student has an age (int) and a gpa (float)

Review Questions!

Answer:

```
struct STUDENT {
    int age;
    float GPA;
};
```

Review Questions!

Question 6.2:

- Using heap memory, create an array students of unknown size **n**
  - **HINT:** calloc
- Fill the array with students, getting input from **stdin** to initialize their **age** and **gpa**
  - **HINT**: scanf

Review Questions!

Answer:

```
struct STUDENT *students, *aStudent;
int n, x;

scanf("%d", &n);
students = (struct STUDENT *) calloc(n, sizeof(struct STUDENT));
if (students == NULL) exit (1);

for(x=0; x<n, x++)  {
    aStudent = students+x;
    scanf("%d %f", &(aStudent->age), &(aStudent->GPA));
}
```

Review Questions!

Question 7.1:

- How can you avoid race conditions and deadlocks when running multiple processes?

Review Questions!

Answer:

You can synchronize your processes by using semaphores to prevent certain processes from running

Review Questions!

Question 7.2:

- Using fork and semaphores, how would you synchronize two processes to count from 1 to n?

Review Questions!

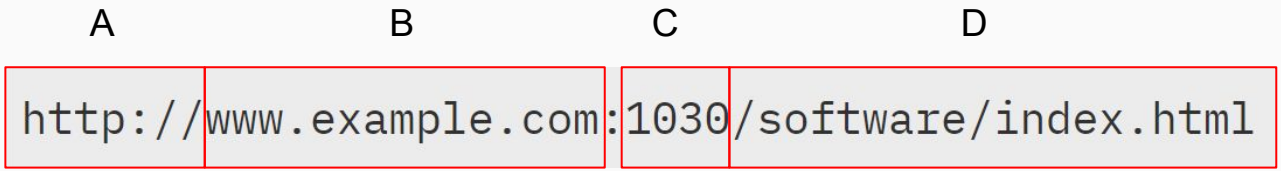
Answer:

You use sem_wait and sem_post to sync up the processes so that only one runs at a time

Review Questions!

Question 8

- Identify the parts of the URL



Review Questions!

Answer:

A) Protocol
B) Domain Name (DNS → IP)
C) Port
D) URI

Review Questions - Advanced C

WARNING!! Different prof, different material. Just an example to try out

Create a STUDENT struct with two fields: char name[100] and float GPA. Create an array of 100 students. Populate this array with data from a CSV file named students.csv. The format of the CSV file is as follows:

Student1,gpa1
Student2,gpa2

The file may have more than 100 students but we only populate that array with those students whose GPA is greater than or equal to 3. If the array becomes full before the end of the file then the program stops reading. The program ends by printing the average GPA and the name of the first student with the highest GPA to the screen. Assume student and gpa are valid. (you can use the back of the page for extra space)

```
struct STUDENT { char name[100]; float GPA; } stud[100];

int main() {
  FILE *p;
  char maxStudent[100]; float maxGPA=0.0, GPA;
  char buffer[1000], aName[100], aGPA, sumGPA=0.0;
  int pos1, pos2, countGPA=0;

  p = fopen("students.csv","rt");
  if (p == NULL) exit(1);

  fgets(buffer,999,p);
  while(!feof(p) && countGPA<100) {
    for(pos1=0; buffer[pos1]!=','; pos1++) aName[pos1]=buffer[pos1];
    aName[pos1]='\0';

    for(pos2=0,pos1++; buffer[pos1]!='\0'; pos1++,pos2++)
      aGPA[pos2]=buffer[pos1];
    aGPA[pos2]='\0';  GPA = atof(aGPA);

    if(maxGPA<GPA) {maxGPA=GPA; strcpy(maxStudent, aName);}

    if (GPA >= 3.0) {
      sumGPA+=GPA; strcpy(stud[countGPA].name,aName);
      stud[countGPA].GPA = GPA; countGPA++
    }

    fgets(buffer,999,p);
  }
  printf("Max stud %s, average gpa %.2f\n",maxStudent, sumGPA/countGPA);
  return 0;
}
```

Feedback: How did we do?

Thanks for attending! We want to know how we did. We would really appreciate it if you filled out the feedback form here: https://goo.gl/forms/Q6F107uoi3p2Dyey1

Version Control - Ignore if not covered in class

Version control is the idea of managing the different ‘conditions’/’save files’ of your code. It can be incredibly useful when experimenting with code and/or working in a team. The architecture you’ll need to be familiar with is Git.

There are three classes/types of version control: Local, Client-Server and Distributed.

All three allow you to revert changes and maintain branches.

Version Control pt. 2

When we use a client-server, the central server holds the code and the (local) client(s) work on their own local repos. You can then transmit any changes to the ‘shared’ version on the server and your partners will be able to manipulate and interact with it.

Conflicts, between local repo(s) and the central server, are then resolved (usually manually) by the users

Git Developer Cycle

Clone → Edit → Add → Commit → Push

Git paraphernalia

- Here are the most useful / important Git commands you'll be using in 206

git clone	copy a repository into local directory
git add	add files to queue for next commit (to be done on any local changes)
git commit	commit queued files
git push	push commit(s) to remote repository (default: same one you cloned from)
git pull	fetch changes from remote repository (default: same one you cloned from)
git init	start a new repository from local files
.gitignore	ignore specific files by adding them here

git status	Show uncommited changes
git rm	Remove a file from a repository
git mv	Move a file within repository
git diff	Generate a differences between
git log	View a log of commits

Even more Git

git branch new_branch	create new branch
git branch -b feature2	create new branch and switch to that branch
git commit	save some work
git checkout master	switch back
git merge feature2	work is merged in
git rebase feature2	work played on top
git branch -d feature2	delete branch

git merge branch_name	Merge changes from branch_name to the current branch
git tag -a name	Add a tag with a indicated name
git tag -l	List tags
git push --tags	Push tags to remote repository