CSCI 3411 – OS Lab 3

POINTERS AND PTHREADS

Overview

- Shells and System Commands Review
- Pointers Review
- Void and Function Pointers
- Pthreads

Review From Last Lab

SHELLS AND SYSTEM COMMANDS

Review from Last Time

- Terminal vs. Shell
  - A terminal is the *window* that runs a shell.
    - Features: background color, font, transparency, etc.
  - A shell is a program that executes other programs.
    - Features: history, autocomplete, aliases, etc.

- Unix Commands vs. Shell Commands
  - Unix Commands: written in C, compiled, and shipped with OS.
    - `ls`, `mkdir`, `touch`, `rm`, `vim`, `nano`, etc.
  - Shell Commands: these are commands built into the shell (i.e. you can't find them on your hard drive – they're in the shell's source code).
    - `history`, `alias`, `!!`, `!n`, etc.

What a Shell Does

1. Print the prompt.
2. Read user input.
3. Attempt to execute the inputted command.
4. `GOTO 1`

System Commands Review

- To find out headers, type man function
- To implement "`cd`" – look into the `chdir(…)` function (unistd.h)
- `fork()`
  - Child process spawned and begins execution from the fork.
- `exec (`*`command, null-terminated arguments array`*`)`
  - Executes a program, replacing the current process.
- `wait(`*`NULL`*`)`
  - Parent waits until a child finishes execution.

Steps for Your Shell

1. Print the prompt.
   - Example “osh>”, “capurso@SEAS: ~/Documents$”, “11:36PM ~/$”
2. Read user input.
   - Parse user input into a string array for exec.
3. Attempt to execute the inputted command.
   - Fork & exec.
     - If exec returns -1 => command not found.
   - Parent calls wait if “&” was in the command.
4. `GOTO 1`
   - Steps 1 – 3 should be in a while loop until the user types “`exit`”.

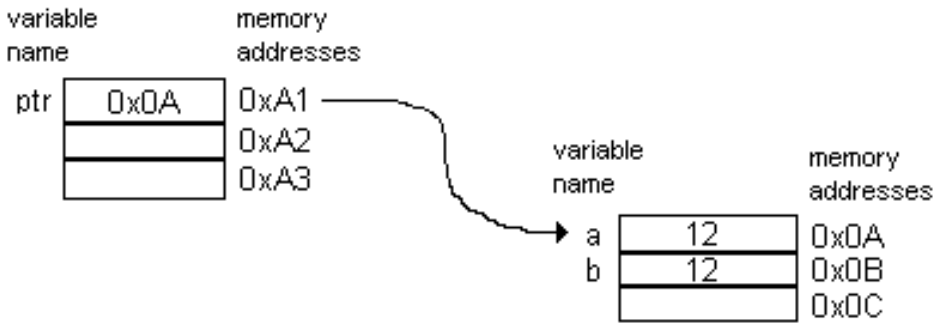
Assignment 2 Questions?

DUE SATURDAY AT MIDNIGHT

C Pointers

STANDARD, VOID, AND FUNCTION POINTERS

Pointers in C




```
int main {
    int *ptr;     // pointer to an integer variable
    int a, b;     // integer variables

    a = 12;       // assign value to a
    ptr = &a;     // assign address of a (0x0A in the example) to ptr
    b = *ptr;     // assign value at the address contained in ptr to b
}
```

The result is that a and b contain the same value (12).
And ptr still contains the address of a.

C Pointers

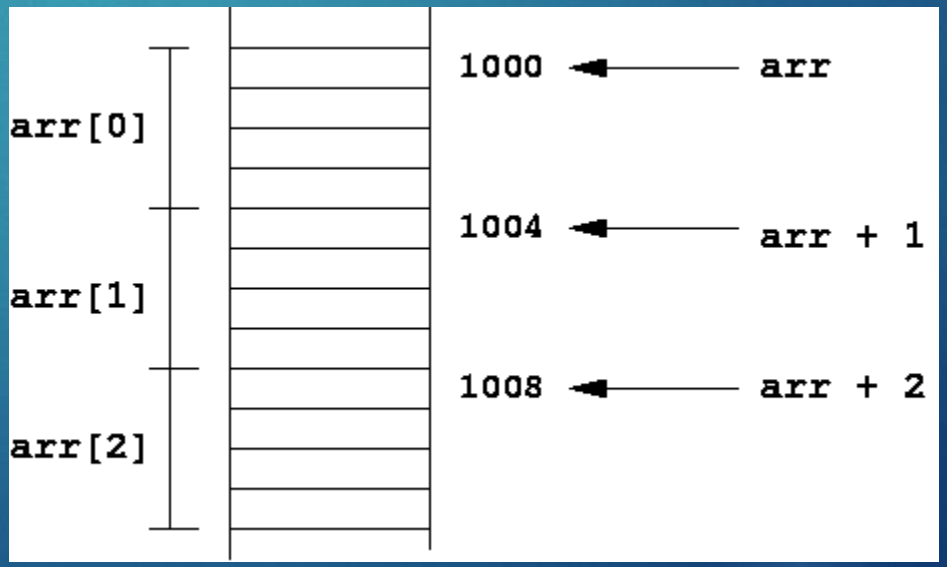
- An array's name is the same as a pointer to the first element.

- `int x[3] = {1, 2, 3};`
  - `x == &(x[0])`
  - `*x == x[0]`
  - `*(x+2) == x[2]`

- Thus, [x] is just "syntactic sugar" of offsetting a pointer by x.

C Pointers

- Pointer Arithmetic
  - Advancing a pointer by 1 advances it logically according to its size.
  - Example: adding 1 to an int pointer advances it by 4 bytes.
- `arr[i] == * (arr + i)`



C Pointers

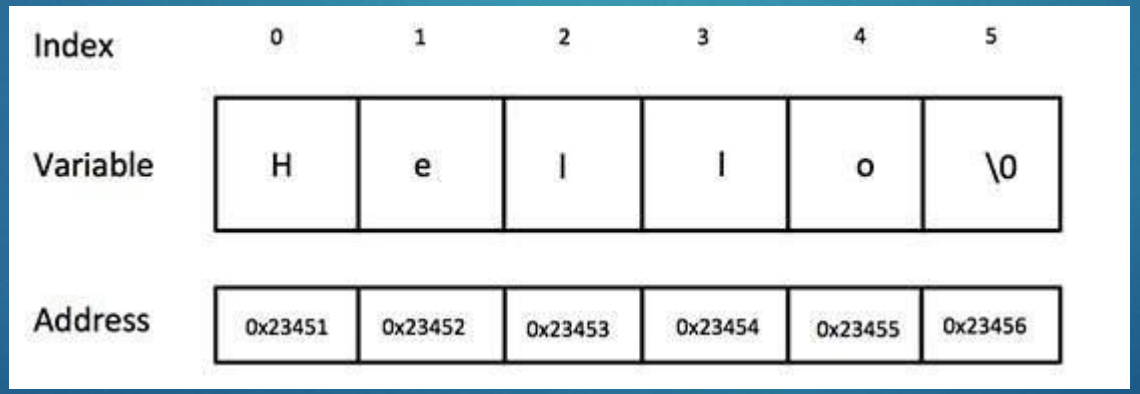
```
int x[5] = {1, 2, 3, 4, 5};
int * myPtr = x + 2;

printf("*myPtr = %d\n, *myPtr);          // Line A

myPtr++:
printf("*myPtr = %d\n, *myPtr);          // Line B

printf("myPtr[1] = %d\n, myPtr[1]);      // Line C
```

C Pointers

```
int x = 5;
int * ptr = &x;

printf(“*ptr = %d\n”, *ptr);      // Line A

printf(“ptr[0] = %d\n”, ptr[0]);  // Line B

printf(“ptr[1] = %d\n”, ptr[1]);  // Line C
```

C Pointers

- Strings in C are char arrays (ending in the null character `'\0'`)
  - `char* x` is then a string (and so is `char x[]`)
  - Examples:
    - `char hi[] = {'h', 'i', '\0'};`
    - `char * hi = "hi";` `//internally = {'h', 'i', '\0'}`

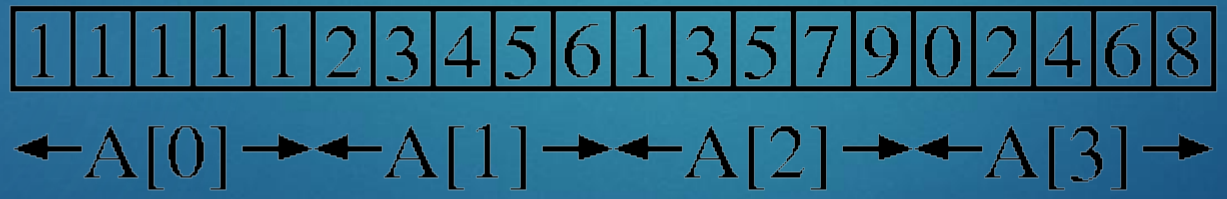


- An array of "strings" in C should therefore be a two dimensional array.

C Pointers - Two Dimensional Arrays

```
int A[4][5]      // A is a statically allocated array
```

1	1	1	1	1
2	3	4	5	6
1	3	5	7	9
0	2	4	6	8




```
A[X][Y] == A + ( (X-1) * Number of Columns) + Y)
```

C Pointers - malloc

- `malloc` simply allocates some space in memory for you to use. In returns a pointer to the free memory.
  - The pointer has to be casted to whatever pointer type you're assigning.

- `malloc(1)` – allocates one byte.
- `malloc(sizeof(int))` – allocates enough bytes for one int.
- `malloc(sizeof(int) * 4)` – allocates space for four ints (like an array)

- `int *ptr = (int *) malloc(sizeof(int) * 4)`
  - Allocates space for four ints and sets ptr to point to the first byte.

C Pointers - Two Dimensional Arrays

- Differences between:
  1. `int A[ROWS][COLS];`
     - A statically defined 2D array of ints. Space set aside automatically.
  2. `int (*B)[COLS];`
     - A pointer to an array of COLS ints. Need to malloc the actual array.
     - The type of B is `(*)[COLS]` - (a pointer to an array of length COLS)
  3. `int *C[ROWS];`
     - An array of ROWS int pointers. Space set aside automatically.
  4. `int **D;`
     - A pointer to an int pointer.
     - This can also be a two dimension array if you malloc space and then use pointer arithmetic.

C Pointers - Two Dimensional Arrays

```
#define MAX_LINE 80
char *args[MAXLINE/2 +1]
```

- Creates an array of 41 char pointers (strings), who can be of variable length.
  - `args[0]` = the first string
  - `args[0][2]` = the third character of the first string

Questions?

C POINTERS

C Void Pointers

- In C, a general purpose pointer (i.e. can store any pointer type) is called a void pointer.

```
void *vPtr;
char c = ‘a’;
int x = 5, y = 0;

vPtr = &c;
vPtr = &x
y = *((int *) vPtr)
```

C Function Pointers

- Unlike other languages, in C you can have a pointer which points to a function.
- You can then pass this "function pointer" around and call the function being pointed to.

- Syntax:
  - Declaration: `returnType (* ptrName) (paramList);`
    - Ex. `int (* myFunPtr)(int, int);`
      - Declares a function pointer which can only point to a function that takes two int parameters and returns at int.
  - Assignment: `ptrName = &myFunction;`

C Function Pointers

```
int addInt(int n, int m){
    return n+m;
}

int main(){
    int (* myFunPtr)(int, int);
    myFunPtr = &addInt;

    int x = (* myFunPtr)(2 ,3);        // x = 5
    …
}
```

C Function Pointers

- You can pass function pointers to other functions:

```
int foo (void (* callbackFun)()){
    // Do some stuff
    …
    (* callbackFun)();      //Call the callback function when
                            //done.
}
```

C Function Pointers

```
typedef struct{
    int x, y;
} point;

typedef struct{
    point *points;
    double (* calcArea)(point *);    //Ptr to function to
                                     //calculate area
} shape;
```

Questions?

C VOID AND FUNCTION POINTERS

Pthreads

#INCLUDE <PTHREAD.H>

Pthreads

- POSIX threads
- `#include <pthread.h>`

- New types:
  - `pthread_t`
    - An ID number for a thread
  - `pthread_attr_t`
    - Holds attributes for a given thread, such as scheduling policy, priority, stack size, etc.
    - In most cases, you can use the default values (see next slide).

Pthread Functions

- `pthread_attr_int( pthread_attr_t *)`
  - Initializes the passed `pthread_attr_t` with default values for scheduling policy, priority, stack size, etc.
- `pthread_create( … )`
  - Creates and runs a new thread.
  - Four arguments:
    - A `pthread_t` in which to store the newly created thread's ID.
    - The attribute set (`pthread_attr_t`)
    - A function pointer which points to function the thread should execute.
    - A void pointer for arguments to the function.
- `pthread_join(pthread_t, NULL)`
  - Causes the parent to wait until the thread finishes execution.

```
int sum;
void *runner (void *threadParams);

int main(){
    pthread_t tid;
    pthread_attr_t attr;
    char *x = “100”;

    // Default attributes
    pthread_attr_init(&attr);

    //Create and start thread
    pthread_create(&tid, &attr, runner, &x);

    //Wait for thread to finish
    pthread_join(tid, NULL);

    // Now print sum out …
}
```

```
//Threads should run this
void *runner (void *params){
    int i;
    int upper;
    sum = 0;

    //param contains &x
    upper = atoi(param);

    //Sums up number from 1 - upper
    for(i = 1, i <= upper, i++)
        sum += i;

    pthread_exit(0);
}
```

Pthreads

- How to pass more than one data type to a thread's function?
  - i.e. an int, a char, maybe some function pointers…

Pthread Exercise

- Write a C program that sums up the numbers 1 – 10. Split the task over two threads (one adds 1 - 5, the other adds 6 – 10).
  - Just do some hardcoding so you don't have to do user input.
  - Let the parent add the two results together.

- Hints: you'll need two pthread_create and two pthread_join

- Compiling:
  - `gcc -pthread file.c`
  - `gcc -l pthread file.c`

Compiling Pthreads

- `gcc -pthread file.c`
- `gcc -l pthread file.c`

Pthreads – Fork and Exec

- Calling `fork()` in a thread?
  - Depends on OS. Some offer two versions of fork – one to duplicate all running threads and one to duplicate only the current thread.
- Calling `exec` within a thread acts as usual (entire process replaced).

Questions?

PTHREADS