Week 4 - Monday

COMP 3400

Last time

- What did we talk about last time?
- File metadata
- Signals
- Overriding signal handlers

Questions?

Project 1

Events and Signals

Common signals

- When using the kill command, the flag can either be the name of the signal (**`-KILL`**) or its number (**`-9`**)
- Here are some common signals:

Name	Number	Description
`SIGINT`	2	Interrupts the process, generally killing it. Sent with **`Ctrl-C`**.
`SIGKILL`	9	Kills the process. Cannot be ignored or overwritten.
`SIGSEGV`	11	Sent to a process when it has a segmentation fault.
`SIGCHLD`	18	Sent to a parent when a child process finishes. Used by **`wait()`**.
`SIGSTOP`	23	Suspends the process. Cannot be ignored or overwritten.
`SIGTSTP`	24	Suspends the process. Sent with **`Ctrl-Z`**.
`SIGCONT`	25	Resumes a suspended process.

Sending signals in a program

- Just as you can use the kill command from the command line, you can also call the **`kill()`** function to send a signal to another process
- The function takes two parameters:
  - PID of the process to kill
  - **`int`** value giving the signal, usually a named constant

```
kill (pid, SIGSTOP); // Suspends process with pid
```

- You can usually only kill processes that you own
  - Unless you're a superuser (like root)

Pointers

Pointers

- Before we go into IPC, some students have mentioned that they're a little rusty at C
- One of the biggest differences between C and Java is the use of pointers
- For that reason, I'm including this short review of pointers

Pointers

- A **pointer** is a variable that holds an address
- Often this address is to another variable
- Sometimes it's to a piece of memory that is mapped to file I/O or something else
- Important operations:
  - Reference (**&**) gets the address of something
  - Dereference (*****) gets the contents of a pointer

Declaration of a pointer

- We typically want a pointer that points to a certain kind of thing
- To declare a pointer to a particular type

```
type * name;
```

- Example of a pointer with type **`int`**:

```
int * pointer;
```

Reference operator

- A fundamental operation is to find the address of a variable
- This is done with the reference operator (**&**)

```
int value = 5;
int *pointer;
pointer = &value; // pointer has value's address
```

- We usually can't predict what the address of something will be

Dereference operator

- The reference operator doesn't let you do much
- You can get an address, but so what?
- Using the dereference operator, you can read and write the contents of the address

```
int value = 5;
int* pointer;
pointer = &value;
printf("%d", *pointer); // prints 5
*pointer = 900; // value just changed!
```

Aliasing

- Java doesn't have pointers
  - But it does have references
  - Which are basically pointers that you can't do arithmetic on
- Like Java, pointers allow us to do aliasing
  - Multiple names for the same thing

```
int wombat = 10;
int* pointer1;
int* pointer2;
pointer1 = &wombat;
pointer2 = pointer1;
*pointer1 = 7;
printf("%d %d %d", wombat, *pointer1, *pointer2);
```

Pointer arithmetic

- One of the most powerful (and most dangerous) qualities of pointers in C is that you can take arbitrary offsets in memory
- When you add to (or subtract from) a pointer, it jumps the number of bytes in memory of the size of the type it points to

```
int a = 10;
int b = 20;
int c = 30;
int* value = &b;
value++;
printf("%d", *value); // What does it print?
```

Arrays are pointers too

- An array **is** a pointer
  - It is pre-allocated a fixed amount of memory to point to
  - You can't make it point at something else
- For this reason, you can assign an array directly to a pointer

```
int numbers[] = {3, 5, 7, 11, 13};
int* value;

value = numbers;
value = &numbers[0]; // Exactly equivalent

value = &numbers; // What about this?
```

Surprisingly, pointers are arrays too

- Well, no, they aren't
- But you can still use array subscript notation (**[ ]**) to read and write the contents of offsets from an initial pointer

```
int numbers[] = {3, 5, 7, 11, 13};
int* value = numbers;

printf("%d", value[3] );      // prints 11
printf("%d", *(value + 3) ); // prints 11
value[4] = 19; // changes 13 to 19
```

void pointers

- What if you don't know what you're going to point at?
- You can use a **void***, which is an address to…something!
- You have to cast it to another kind of pointer to use it
- You can't do pointer arithmetic on it
- It's not useful very often

```
char s[] = "Hello World!";
void* address = s;
int* thingy = (int*)address; // Uh-oh
printf("%d\n", *thingy);
```

Functions that can change arguments

- In general, data is passed **by value**
- This means that a variable cannot be changed for the function that calls it
- Usually, that's good, since we don't have to worry about functions screwing up our data
- It's annoying if we need a function to return more than one thing, though
- Passing a pointer is equivalent to passing the original data **by reference**

Example

- Let's imagine a function that can change the values of its arguments

```
void swapIfOutOfOrder (int *a, int *b)
{
  if (*a > *b)
     {
        int temp = *a;
        *a = *b;
        *b = temp;
     }
}
```

How do you call such a function?

- You have to pass the addresses (pointers) of the variables directly

```
int x = 5;
int y = 3;
swapIfOutOfOrder(&x, &y); // Will swap x and y
```

- With normal parameters, you can pass a variable or a literal
- However, you **cannot** pass a reference to a literal

```
swapIfOutOfOrder(&5, &3); // Impossible
```

malloc()

- Memory can be allocated dynamically using a function called **malloc()**
  - Similar to using **new** in Java or C++
  - **#include <stdlib.h>** to use **malloc()**
- Dynamically allocated memory is on the heap
  - It doesn't disappear when a function returns
- To allocate memory, call **malloc()** with the number of bytes you want
- It returns a pointer to that memory, which you cast to the appropriate type

```
int* data = (int*)malloc(sizeof(int));
```

Allocating arrays

- It's common to allocate an array of values dynamically
- The syntax is exactly the same, but you multiply the size of the type by the number of elements you want

```
int i = 0;
int *array = (int*)malloc (sizeof(int)*100);
for (i = 0; i < 100; ++i) // Initialize for fun
  array[i] = i + 1;
```

Pointers to structs

- We can define a pointer to a struct variable
  - We can point it at an existing struct
  - We can dynamically allocate a struct to point it at

```
struct student bob;
struct student *studentPointer;
strcpy(bob.name, "Bob Blobberwob");
bob.GPA = 3.7;
bob.ID = 100008;
studentPointer = &bob;
(*studentPointer).GPA = 2.8;
studentPointer = (struct student*)malloc(sizeof(struct
student));
```

Arrow notation

- As we saw on the previous slide, we have to dereference a struct pointer and then use the dot to access a member

```
struct student* studentPointer = (struct student*)
    malloc(sizeof(struct student));
(*studentPointer).ID = 3030;
```

- This is cumbersome and requires parentheses
- Because this is a frequent operation, dereference + dot can be written as an arrow (->)

```
studentPointer->ID = 3030;
```

Passing structs to functions

- If you pass a struct directly to a function, you are passing it by value
  - A copy of its contents is made
- It is common to pass a struct by pointer to avoid copying and so that its members can be changed

```
void flip (struct point *value)
{
  double temp = value->x;
  value->x = value->y;
  value->y = temp;
}
```

calloc()

- One problem with **malloc()** is that the memory it allocates is filled with garbage
- Like **malloc()**, **calloc()** allocates memory, but it also zeroes all of it out
- Many programmers think it's safer to use **calloc()** in *all* situations where you would use **malloc()**
- There's a slight syntax difference:
  - **calloc()** takes two arguments: number of elements and size of each one

```
// malloc() version
int *array1 = (int*)malloc (sizeof(int)*100);
// equivalent calloc() version
int *array2 = (int*)calloc (100, sizeof(int));
```

realloc()

- For a dynamic array, it can be useful to grow an existing chunk of memory if it's too small
- You could allocate an entirely new, bigger chunk of memory, copy everything from the old memory over, and then free the old memory
  - This is what you *have* to do in Java
- C provides a slick function, **`realloc()`**, that does all of that for you
  - Arguments: memory to resize, new size
  - Return value: resized memory

```
if(size == capacity)
   {
     capacity *= 2;
     array = realloc(array, capacity*sizeof(int));
   }
array[size] = element;
++size;
```

free()

- C isn't garbage collected like Java
- If you allocate something on the stack, it disappears when the function returns
- If you allocate something on the heap, you have to deallocate it with **`free()`**
- **`free()`** does *not* set the pointer to be **`NULL`**
  - But you can (and should) afterwards

```
char *things = (char*)malloc (100*sizeof(char));
// Do stuff with things
free(things);
things = NULL;
```

Interprocess Communication

Interprocess communication

- We have talked about
  - Running processes
  - Creating new processes with **`fork()`** and **`exec()`**
  - Destroying processes
  - Sending signals to processes
- In general, these processes are separate
  - What happens in one process doesn't affect another
- However, there are times when one process needs to communicate with another
- **Interprocess communication** (**IPC**) is an umbrella term for the different ways these messages can be sent

Message passing

- There are many IPC approaches, but they can all be categorized as either **message passing** or **shared memory**
- Message passing:
  - Sender prepares a message
  - Sender makes a system call to request a data transfer
  - Kernel copies the message into a buffer
  - Receiver makes a system call to retrieve the data
  - Receiver copies the message into its own memory

Shared memory

- Shared memory IPC is completely different
- The processes decide on a chunk of virtual memory that will be used for IPC
- The processes make system calls to request that this memory is shared
- Once it's shared, processes can read and write from shared memory just like any other data in the program
- Mediation through the kernel isn't needed after the memory is shared

Pros and cons of message passing

- Message passing requires:
  - A system call to read
  - A system call to write
  - Copying the message into kernel memory
  - Copying the message into receiver memory
- Thus, sending lots of messages can cause a lot of overhead
- However, sending a small number of messages can be less expensive than setting up shared memory
- Message passing naturally handles the problem of synchronization
  - Making sure that timing doesn't corrupt memory

Pros and cons of shared memory

- It's computationally expensive to set up the shared memory
- But that's a one-time cost
- If two processes are sharing lots of messages, it can be more efficient to use a shared memory system
- Perhaps the more significant problem with shared memory is synchronization
  - Processes reading and writing the same memory can leave the memory in an inconsistent state
  - If one process executes `x += 100` while another executes `x -= 100`, the result could be the correct `x` or the incorrect `x + 100` or `x - 100`
- Tools must be used to guarantee synchronization

The IPC zoo

- Although all IPC techniques fall under the message passing or the shared memory model, there are other ways to categorize them:
  - For data exchange or purely for synchronization
  - As a stream or bytes or data with more structure
  - For local communication or for networked communication
- Note: People sometimes use the term "shared memory" to refer only to the technique using **`shm_open()`** and not memory-mapped files

IPC taxonomy

- Using the categories from the previous slide, we can list all of the IPC techniques that will be covered in this class

Technique	Model	Purpose	Granularity	Network
Pipe/FIFO	Message passing	Data exchange	Byte stream	Local
Socket	Message passing	Data exchange	Either	Either
Message queue	Message passing	Data exchange	Structured	Local
`shm()`	Shared memory	Data exchange	None	Local
Memory-mapped file	Shared memory	Data exchange	None	Local
Signal	Message passing	Synchronization	None	Local
Semaphore	Message passing	Synchronization	None	Local

- We talked about signals last week, which are a form of IPC but very limited
- We'll cover sockets when we talk about networking

Upcoming

Next time…

- More on IPC
- Pipes
- FIFOs

Reminders

- Keep working on Project 1
  - Due Friday by midnight!
- Read section 3.3