CIS 5480 Recitation 0

Thursday, January 29 2025

Agenda

- wait versus waitpid
- strtok
- GDB
- Valgrind
- Penn Shredder tips

wait() versus waitpid()

Syntax:

```
 -   pid_t wait(int* wstatus);
 -   pid_t waitpid(pid_t pid, int* wstatus, int options);
```

What is the difference between wait() and waitpid()?

1.
2.

What arguments would you provide to waitpid() to make it behave identically to wait()?

- Hint: use the man pages!

wait() versus waitpid()

Syntax:

```
-   pid_t wait(int* wstatus);
-   pid_t waitpid(pid_t pid, int* wstatus, int options);
```

What is the difference between wait() and waitpid()?

1. waitpid() allows you to specify the process you want to wait on, while wait() will unblock at the first instance of a child terminating
2. waitpid() allows the process to check on its child's status without blocking using option WNOHANG. Because of this, waitpid can return 0 to indicate no change.

What arguments would you provide to waitpid() to make it behave identically to wait()?

- Hint: use the man pages!

  Answer: `waitpid(-1, &wstatus, 0);`

wait() versus waitpid() in penn-shredder

Given that waitpid has more options (the ability to specify the process it waits on, or avoid blocking with WNOHANG), why is just using **wait()** sufficient for programming penn-shredder?

wait() versus waitpid() in penn-shredder

Given that waitpid has more options (the ability to specify the process it waits on, or avoid blocking with WNOHANG), why is just using **wait()** sufficient for programming penn-shredder?

Answer:

- The shell is not supposed to reprompt until either exec finishes or the time limit is reached. Therefore, using waitpid() with WNOHANG doesn't make sense (unless we like busy-waiting). Using wait() will automatically block the parent process until the child has terminated.
- The shell is only going to fork one child per prompt because only one extra process is needed to run exec. If there are no other processes, the parent only needs to wait on that child, and there is no need to specify *which* child we are waiting for.

Strtok()

- Let’s go to the man page for this one :)

GDB

- GDB is a useful debugger that allows you to inspect a program
- Compile a c program with flag -g allows you to use gdb on it
    - Your provided makefiles will have this option already
- gdb [program name] to run gdb on the program
- List of gdb commands next slide

GDB Commands

- **file [program]** - mounts the program onto gdb
- **run** - runs the program
- **backtrace/bt** - show the stack of this program
- **print [variable]** - shows variable’s value at this instant of program execution
- **break/br [line]** - set a breakpoint at line number
- **continue/c** - continues execution
- **list** - prints the source code around the current line.
- **step** - runs the next line of code, and stops again. If the current line is a function call, it steps into the function call.
- **next** - is similar to step, but steps *over* any function calls.
- **watch [var]**- stops each time the variable var changes.

Valgrind

- IMO, much easier to understand than GDB.
- Valgrind catches memory errors of all sorts
  - Important because your code could be passing test cases while also performing illegal memory operations without your knowledge
- On terminal: `valgrind [executable name] [optional executable arguments]`

What memory errors could you run into?

- Conditional move or jump (on uninitialized values)
- Invalid read/write: accessing memory you should not
- Invalid free's
- Memory leaks!!

More info here: https://valgrind.org/docs/manual/mc-manual.html#mc-manual.errormsgs

Penn Shredder Tips

- Anytime you print-debug, make sure you include \n at the end of the print statement, otherwise the buffer might not flush properly
    - Check out this blog post for more info
- Your implemented vector can be useful! Not just for shredder – it can also be used in future projects
- strlen() does not include the null terminator. read() does not include the null terminator.
- When in doubt, add your signal handlers early
- If you’re not sure whether a function accounts for x y or z (for example, does strcpy include null terminator?), look it up on the man pages!

OLD SLIDES BELOW

Contents

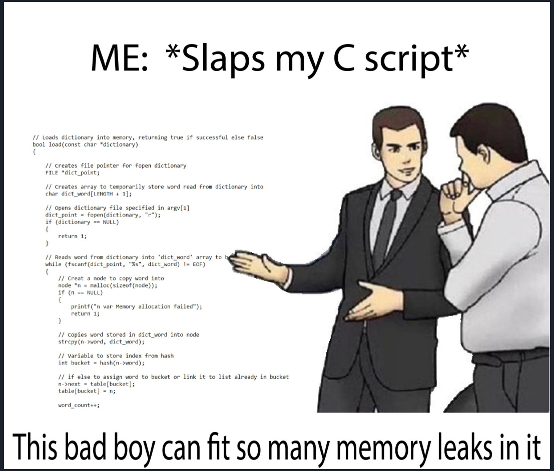
Quick C History

- C came out of Bell Laboratories along with Unix around the same time (1972ish)
- Created by Dennis Ritchie and Ken Thompson who also created Unix
- Linux and MacOS are both based upon Unix
- Linux (used in this course) is written completely in C (but possibly soon Rust as well)
- Despite its old age and simplicity, C is still an extremely popular language
- https://github.com/torvalds/linux

Tux the Penguin!

Why C?

- C is fast and very portable
- It does not include quality of life aspects other programming languages have
- No garbage collection, bounds checks and objects
- Can create powerful programs but also monstrous errors (memory safety)
- Since Unix/Linux are written in C, C's use of system calls matches what the OS uses
- Soft bound paper to learn more about bound checking https://acg.cis.upenn.edu/papers/pldi09_softbound.pdf



```
// Loads dictionary into memory, returning true if successful else false
bool load(const char *dictionary)
{

    // Creates file pointer for fopen dictionary
    FILE *dict_point;

    // Creates array to temporarily store word read from dictionary into
    char dict_word[LENGTH + 1];

    // Opens dictionary file specified in argv[1]
    dict_point = fopen(dictionary, "r");
    if (dictionary == NULL)
    {
        return 1;
    }

    // Reads word from dictionary into 'dict_word' array to b
    while (fscanf(dict_point, "%s", dict_word) != EOF)
    {
        // Creat a node to copy word into
        node *n = malloc(sizeof(node));
        if (n == NULL)
        {
            printf("n var Memory allocation failed");
            return 1;
        }

        // Copies word stored in dict_word into node
        strcpy(n->word, dict_word);

        // Variable to store index from hash
        int bucket = hash(n->word);

        // if else to assign word to bucket or link it to list already in bucket
        n->next = table[bucket];
        table[bucket] = n;

        word_count++;
```

Pointers

- Literally point to a place in memory
- A variable which contain a memory location
- Declared with a type and an * ex (int *ptr;)
- * is used to dereference a pointer or operand
- Dereference is to “go to” the memory location in the pointer
- & is used to get the memory location of the passed operand
- & is used in conjunction with pointers ex (int *ptr = &integer)
- Can dereference multiple times
- pointers.c

Arrays

- Contiguous memory of the same type
- Arrays are referenced by using a pointer to the first element
- Very easy to go past the bounds of an array and cause memory errors
- arrays.c

Strings

- No “String” type in C
- Strings are just arrays of characters
- All strings in C must end in ‘\0’, the null character
- Functions often continue reading until they find a null character
- This is referred to as a null terminated string
- String literals are super funky
- Very important for project 0!
- strings.c

Structs

- Objects do not exist in C, but structs do!
- Custom data types which contain inner custom fields
- Structs are allocated as contiguous memory
- Very similar to arrays, but filled with possibly different data types
- structs.c and structs.h

Memory Management

- Stack
    - Static storage / local scope
    - Automatically allocated/deallocated
    - Small upper bound in size (stack overflow)
    - char str[6] = “hello”;
- Heap
    - Dynamic storage / program scope
    - Allocated with system calls and freed with free(3)
    - Large size
    - char* str = malloc(6 * sizeof(char));

Memory Management

- Proj 0 can be done solely on the stack, but proj1 and proj2 will be tough to only allocate stack memory
- System calls to allocate memory
    - **malloc(3),** calloc(3), etc.
- System call to free allocated memory
    - free(3)
- What to allocate?
    - ANY pointers / arrays unless we tell you otherwise
    - Strings (char*)
- Make sure to FREE all memory before exit(2)!
- Example code memory.c

C coding style

See c_style.c, c_style.h

Valgrind

- Memory error checking program
- Very useful for finding memory leaks and memory errors
- Valgrind runs around the program running
- Common valgrind errors are memory leaks, invalid reads/writes, and uninitialised bytes
- Simply run `valgrind ./program <program arguments>`
- Useful valgrind arguments:
  - `--trace-children=<yes|no> [default: no]`
  - `--track-origins=<yes|no> [default: no]`
  - `--leak-check=<no|summary|yes|full> [default: summary]`

GDB

- GDB is a useful debugger that allows you to inspect a program
- Compile a c program with flag -g allows you to use gdb on it
  - Your provided makefiles will have this option already
- gdb [program name] to run gdb on the program
- List of gdb commands next slide
- Example debug code gdb.c

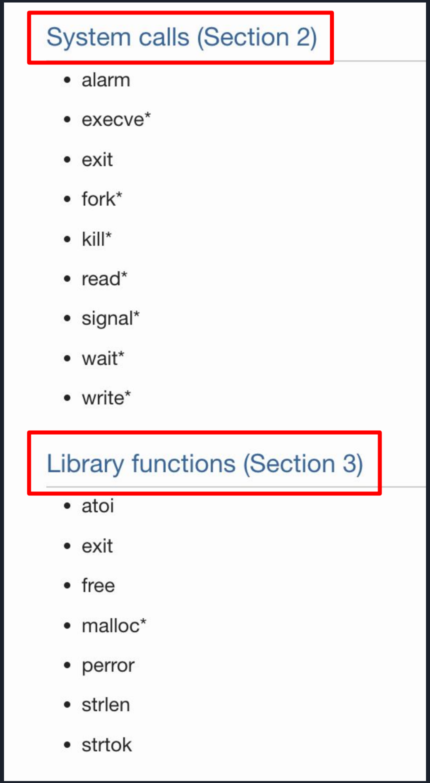
GDB Commands

- **file [program]** - mounts the program onto gdb
- **run** - runs the program
- **backtrace/bt** - show the stack of this program
- **print [variable]** - shows variable’s value at this instant of program execution
- **break/br [line]** - set a breakpoint at line number
- **continue/c** - continues execution
- **list** - prints the source code around the current line.
- **step** - runs the next line of code, and stops again. If the current line is a function call, it steps into the function call.
- **next** - is similar to step, but steps *over` any function calls.
- **watch [variable]**- stops each time the variable var changes.

7. Helpful sites for C reference

Linux man pages: https://man7.org/linux/man-pages/

- Either access online (link above) or in terminal:
  - $ man [section number] [func name]
    - i.e.: man 2 alarm, man 3 malloc
  - Press q to exit
  - "Section number":
    - Man pages are broken into sections, including commands (sec 1), system calls (sec 2), C library functions (sec 3)



System calls (Section 2)

- alarm
- execve*
- exit
- fork*
- kill*
- read*
- signal*
- wait*
- write*

Library functions (Section 3)

- atoi
- exit
- free
- malloc*
- perror
- strlen
- strtok

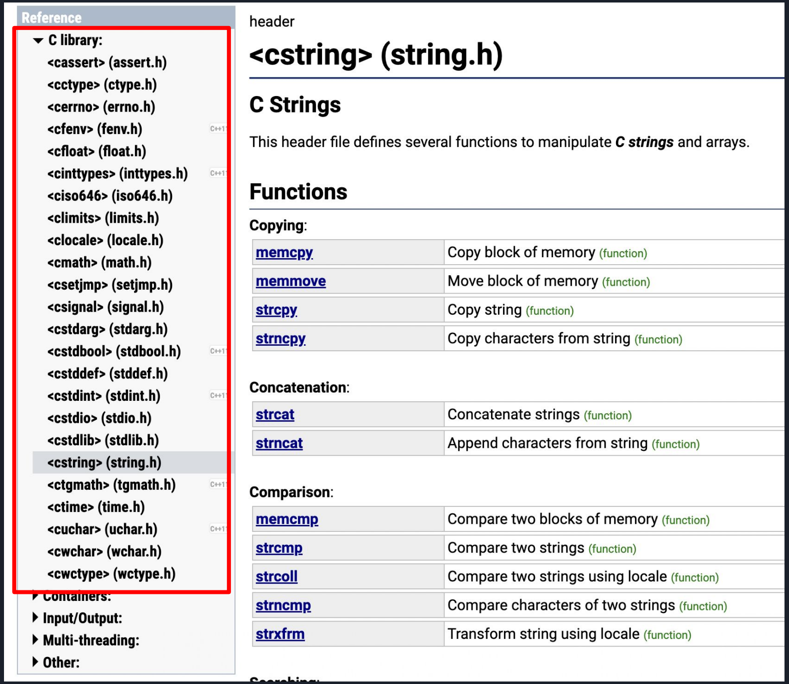
Some C++ references contain sections on C

https://cplusplus.com/reference/

- Contains really nice documentation for C string functions: https://cplusplus.com/reference/cstring/
- Be careful not to stray out of the C library

https://en.cppreference.com/w/c

- C language basics



Reference

- C library:
  - <cassert> (assert.h)
  - <cctype> (ctype.h)
  - <cerrno> (errno.h)
  - <cfenv> (fenv.h) C++11
  - <cfloat> (float.h)
  - <cinttypes> (inttypes.h) C++11
  - <ciso646> (iso646.h)
  - <climits> (limits.h)
  - <clocale> (locale.h)
  - <cmath> (math.h)
  - <csetjmp> (setjmp.h)
  - <csignal> (signal.h)
  - <cstdarg> (stdarg.h)
  - <cstdbool> (stdbool.h) C++11
  - <cstddef> (stddef.h)
  - <cstdint> (stdint.h) C++11
  - <cstdio> (stdio.h)
  - <cstdlib> (stdlib.h)
  - <cstring> (string.h)
  - <ctgmath> (tgmath.h) C++11
  - <ctime> (time.h)
  - <cuchar> (uchar.h) C++11
  - <cwchar> (wchar.h)
  - <cwctype> (wctype.h)
- Containers:
- Input/Output:
- Multi-threading:
- Other:

header

<cstring> (string.h)

C Strings

This header file defines several functions to manipulate ***C strings*** and arrays.

Functions

Copying:

memcpy	Copy block of memory (function)
memmove	Move block of memory (function)
strcpy	Copy string (function)
strncpy	Copy characters from string (function)

Concatenation:

strcat	Concatenate strings (function)
strncat	Append characters from string (function)

Comparison:

memcmp	Compare two blocks of memory (function)
strcmp	Compare two strings (function)
strcoll	Compare two strings using locale (function)
strncmp	Compare characters of two strings (function)
strxfrm	Transform string using locale (function)