Linux, Pointers and pthreads

Edward Zhang

SOFTENG 370 T1

Hello!

I'm in Part IV, and you probably remember me from SOFTENG 251, SOFTENG 206, and SOFTENG 254

- Ask questions on Piazza instead of emailing me so your classmates can see the answers (also such that Robert can answer questions that I can't, such as specifics regarding what you can and can't do in the assignment)
- If you want to meet, email me first at ezha210@aucklanduni.ac.nz
- These slides will be on Canvas, and any source code demonstrated along with TeX source code for these slides can be found on github.com/encryptededdy

You need a UNIX system

Some ways to get a UNIX system to do this assignment

- Dual Boot Linux
- Run Linux in a Virtual Machine
- Run natively on macOS
  - Probably won't work for Assignment 2 (no FUSE)
- Run within Windows Subsystem for Linux (WSL)
  - Probably won't work for Assignment 2 (no FUSE)
- Run within Windows Subsystem for Linux 2 (WSL2)
  - Unreleased, unless you want to run Insider Fast Ring (not recommended)

On Virtual Machines

You can use any distro you want, but you'll probably be able to get more help when googling if you use one of the more popular desktop ones.

- Ubuntu (probably 18.04 LTS)
- Fedora Workstation (my personal preference)
- Debian
- Arch (great wiki, and u use arch btw), Manjaro if you actually want an installer

Hypervisors

Oracle's VirtualBox is the usual free go-to. I personally prefer VMWare Player, feel free to give it a try. Parallels is a good option on macOS, but it's $$$.

Also try Hyper-V on Windows if you have Pro and already have it enabled, as it lets you keep other Windows features on (like Windows Sandbox or Core Isolation). It also supports one-click install of Ubuntu.

Note on Dual Booting

Beware you may be unable to dual-boot on some hardware, such as Surface Devices (drivers are a bit of a pain, especially on the book; check r/surfacelinux for more resources), or the 2019 MacBook Pro (can't even install, T2 chip NVMe storage support broken).

VSCode Remote

You can develop in a Linux environment with a Linux toolchain, while running VSCode from within Windows. This supports WSL. See: https://code.visualstudio.com/docs/remote/wsl

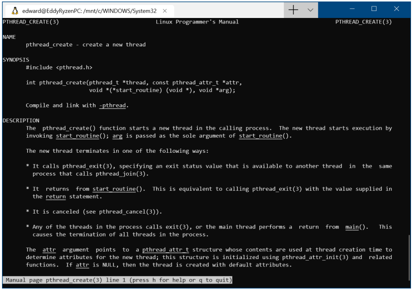
Software to use

- Install gcc (if not part of your distro) using apt/dnf/pacman
- Visual Studio Code is a fine text editor with IntelliSense
- You could also use CLion (JetBrains) if you prefer IntelliJ-like shortcuts and autocomplete, however you will need to create your own CMake file for building. There's no free version, but you can sign up for a JetBrains educational account

Using man to find documentation

Man is a built in documentation tool. In this case, we can check the documentation for pthread_create using...

`$ man pthread_create`



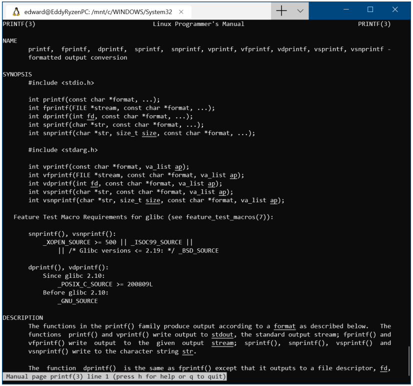

```
PTHREAD_CREATE(3)                    Linux Programmer's Manual                    PTHREAD_CREATE(3)

NAME
       pthread_create - create a new thread

SYNOPSIS
       #include <pthread.h>

       int pthread_create(pthread_t *thread, const pthread_attr_t *attr,
                          void *(*start_routine) (void *), void *arg);

       Compile and link with -pthread.

DESCRIPTION
       The  pthread_create() function starts a new thread in the calling process.  The new thread starts execution by
       invoking start_routine(); arg is passed as the sole argument of start_routine().

       The new thread terminates in one of the following ways:

       * It calls pthread_exit(3), specifying an exit status value that is available to another thread  in  the  same
         process that calls pthread_join(3).

       * It  returns  from start_routine().  This is equivalent to calling pthread_exit(3) with the value supplied in
         the return statement.

       * It is canceled (see pthread_cancel(3)).

       * Any of the threads in the process calls exit(3), or the main thread performs a  return  from  main().   This
         causes the termination of all threads in the process.

       The  attr  argument  points  to  a pthread_attr_t structure whose contents are used at thread creation time to
       determine attributes for the new thread; this structure is initialized using pthread_attr_init(3) and  related
       functions.  If attr is NULL, then the thread is created with default attributes.
 Manual page pthread_create(3) line 1 (press h for help or q to quit)
```

Finding the correct manpage

What if there are multiple versions of a given function?

`$ man 3 printf`

Use 3 to access section 3, which contains the C function version of printf. Without 3 you get the linux command.



```
PRINTF(3)                    Linux Programmer's Manual                    PRINTF(3)

NAME
       printf,  fprintf,  dprintf,  sprintf,  snprintf, vprintf, vfprintf, vdprintf, vsprintf, vsnprintf -
       formatted output conversion

SYNOPSIS
       #include <stdio.h>

       int printf(const char *format, ...);
       int fprintf(FILE *stream, const char *format, ...);
       int dprintf(int fd, const char *format, ...);
       int sprintf(char *str, const char *format, ...);
       int snprintf(char *str, size_t size, const char *format, ...);

       #include <stdarg.h>

       int vprintf(const char *format, va_list ap);
       int vfprintf(FILE *stream, const char *format, va_list ap);
       int vdprintf(int fd, const char *format, va_list ap);
       int vsprintf(char *str, const char *format, va_list ap);
       int vsnprintf(char *str, size_t size, const char *format, va_list ap);

   Feature Test Macro Requirements for glibc (see feature_test_macros(7)):

       snprintf(), vsnprintf():
           _XOPEN_SOURCE >= 500 || _ISOC99_SOURCE ||
               || /* Glibc versions <= 2.19: */ _BSD_SOURCE

       dprintf(), vdprintf():
           Since glibc 2.10:
               _POSIX_C_SOURCE >= 200809L
           Before glibc 2.10:
               _GNU_SOURCE

DESCRIPTION
       The functions in the printf() family produce output according to a format as described below.   The
       functions  printf() and vprintf() write output to stdout, the standard output stream; fprintf() and
       vfprintf() write  output  to  the  given  output  stream;  sprintf(),  snprintf(),  vsprintf()  and
       vsnprintf() write to the character string str.

       The  function  dprintf()  is the same as fprintf() except that it outputs to a file descriptor, fd,
 Manual page printf(3) line 1 (press h for help or q to quit)
```

Defining Pointers

Consider a variable foo. Say we define it as `int foo;`

- `&foo` gives us the address of foo.
- `int *fooPointer` stores a pointer to something of type int. Thus, we could do something like `int *fooPointer = &foo;`

Assignment / Dereferencing

Ok, now we have a pointer to foo that we defined with `int *fooPointer = &foo;`. How can we write to what it's pointing too (foo)?

- You cannot just go `fooPointer = 12`
- We can instead dereference using an asterisk and perform a store, such as `*fooPointer = 12`
- We can load the value such as `int bar = *fooPointer;`
- Note that once we load it into bar, updating bar won't change foo.

Example

```c
#include <stdio.h>

int main( int argc, const char* argv[] )
{
    int foo;
    int *fooPointer = &foo;
    *fooPointer = 420;

    printf("%d\n", fooPointer); // Compiler warning
    printf("%d\n", *fooPointer);
    printf("%d\n", foo);

    int bar = *fooPointer;
    bar = 840;

    printf("%d\n", bar);
    printf("%d\n", foo);
}
```

Indirection

You can do this by the way..

```
int     a =  100;
int    *b = &a;
int   **c = &b;
int  ***d = &c;
```

And to dereference these, use the appropriate number of asterisks

```
***d == **c == *b == a == 100;
```

Note that **d would return a type int * (b), and *d would return a type int ** (c).

Function Pointers

There are cases where we have to pass around functions, and for that we can use function pointers! Consider this function...

```
void meme(int a) {
  printf("Nobody: 0, C: %d", a);
}
```

To define a variable that stores a function that returns void and takes an int, then assign it with the meme function, we can do this...

```
void (*funcPtr)(int);
funcPtr = &meme;
```

In order to call funcPtr, we simply dereference it and give it the input we want.

```
(*funcPtr)(370); // prints Nobody: 0, C: 370
```

Function Pointers cont.

This is useful if we want a function that takes a function as a parameter...

```
void caller(void (*func)(int))
{
    func(100);
}
```

...prehaps by a library that helps you run your function on a seperate thread :thinking:

What's this?

Consider this section of code from the assignment. What's happening here?

```
struct block right_block;
struct block left_block;
left_block.size = my_data->size / 2;
left_block.first = my_data->first;
right_block.size = left_block.size + (my_data->size % 2);
right_block.first = my_data->first + left_block.size;
merge_sort(&left_block);
merge_sort(&right_block);
merge(&left_block, &right_block);
```

Recall that the block struct has `size` as an int, and `first` as an *int

Pointer Addition

If the first element is at memory location 0, then the second is at 4, then 8 and so on. (ints are usually 4 bytes).

```
right_block.first = my_data->first + left_block.size;
```

When we add `size` to `first`, we essentially shift the pointer forward by `size` elements (therefore selecting the second half). Note that we aren't adding `size` bytes. Since `first` is an int pointer, (size of int $\times$ `size`) bytes are added.

Structure Basics

We can define a struct that holds multiple variables like this...

```
struct Stuff
{
  int a;
  int b;
}
```

Structure Basics

We can define a struct that holds multiple variables like this. . .

```
struct Stuff
{
  int a;
  int b;
}
```

And declare and assign to it. . .

```
struct Stuff foo;
foo.a = 0;
foo.b = 1;
// or
struct Stuff foo = {0, 1};
```

Pointing to structs

```
struct Stuff foo = {0, 1};
struct Stuff *fooPtr = &foo;
```

Now we have a pointer to a struct. But how do we access a and b inside it using the pointer?

Pointing to structs

```
struct Stuff foo = {0, 1};
struct Stuff *fooPtr = &foo;
```

Now we have a pointer to a struct. But how do we access a and b inside it using the pointer?
Well, we could dereference it. . .

```
(*fooPtr).a
(*fooPtr).b
```

But that's ugly. So instead we can use an arrow ("pointer to member"). . .

```
fooPtr->a
fooPtr->b
```

Lifetime of a stack variable

If we just initialize a variable like we do with “local” below, it is simply allocated on the stack. Recall that the stack is freed once a function returns.

```
int* func()
{
  int local = 7;
  return &local;
}
```

What’s wrong with this code?

Lifetime of a stack variable

If we just initialize a variable like we do with “local” below, it is simply allocated on the stack. Recall that the stack is freed once a function returns.

```
int* func()
{
  int local = 7;
  return &local;
}
```

What’s wrong with this code?
A: After we return this function, local will be removed from the stack. Therefore, when whatever calls func tries to dereference the pointer that was returned, it may not point to what we want it to.

malloc

Allocates a give number of bytes (not on the stack!), and return a pointer to said memory. We can then store stuff at this memory location that won't be lost when our function returns.

```
MALLOC(3)                        Linux Programmer's Manual                        MALLOC(3)

NAME
       malloc, free, calloc, realloc - allocate and free dynamic memory

SYNOPSIS
       #include <stdlib.h>

       void *malloc(size_t size);
       void free(void *ptr);
       void *calloc(size_t nmemb, size_t size);
       void *realloc(void *ptr, size_t size);
       void *reallocarray(void *ptr, size_t nmemb, size_t size);

   Feature Test Macro Requirements for glibc (see feature_test_macros(7)):

       reallocarray():
           _GNU_SOURCE

DESCRIPTION
       The  malloc()  function allocates size bytes and returns a pointer to the allocated memory.  The memory is not
       initialized.  If size is 0, then malloc() returns either NULL, or a unique pointer value  that  can  later  be
       successfully passed to free().

       The free() function frees the memory space pointed to by ptr, which must have been returned by a previous call
       to malloc(), calloc(), or realloc().  Otherwise, or if free(ptr) has already  been  called  before,  undefined
       behavior occurs.  If ptr is NULL, no operation is performed.

       The calloc() function allocates memory for an array of nmemb elements of size bytes each and returns a pointer
       to the allocated memory.  The memory is set to zero.  If nmemb or size is  0,  then  calloc()  returns  either
       NULL, or a unique pointer value that can later be successfully passed to free().

       The  realloc()  function  changes  the size of the memory block pointed to by ptr to size bytes.  The contents
       will be unchanged in the range from the start of the region up to the minimum of the old and  new  sizes.   If
       the  new size is larger than the old size, the added memory will not be initialized.  If ptr is NULL, then the
       call is equivalent to malloc(size), for all values of size; if size is equal to zero, and  ptr  is  not  NULL,
       then  the  call is equivalent to free(ptr).  Unless ptr is NULL, it must have been returned by an earlier call
       to malloc(), calloc(), or realloc().  If the area pointed to was moved, a free(ptr) is done.

       The reallocarray() function changes the size of the memory block pointed to by ptr to be large enough  for  an
       array of nmemb elements, each of which is size bytes.  It is equivalent to the call

               realloc(ptr, nmemb * size);

       However,  unlike  that  realloc() call, reallocarray() fails safely in the case where the multiplication would
       overflow.  If such an overflow occurs, reallocarray() returns NULL, sets errno to ENOMEM, and leaves the orig
       inal block of memory unchanged.

RETURN VALUE
       The  malloc()  and  calloc() functions return a pointer to the allocated memory, which is suitably aligned for
       any built-in type.  On error, these functions return NULL.  NULL may also be returned by a successful call  to
       malloc() with a size of zero, or by a successful call to calloc() with nmemb or size equal to zero.

       The free() function returns no value.

       The  realloc()  function  returns  a  pointer to the newly allocated memory, which is suitably aligned for any
       built-in type and may be different from ptr, or NULL if the request fails.  If size was  equal  to  0,  either
       NULL or a pointer suitable to be passed to free() is returned.  If realloc() fails, the original block is left
       untouched; it is not freed or moved.

       On success, the reallocarray() function returns a pointer to the  newly  allocated  memory.   On  failure,  it
       returns NULL and the original block of memory is left untouched.

ERRORS
       calloc(), malloc(), realloc(), and reallocarray() can fail with the following error:

       ENOMEM Out  of  memory.   Possibly,  the application hit the RLIMIT_AS or RLIMIT_DATA limit described in getr
              limit(2).
 Manual page malloc(3) line 1 (press h for help or q to quit)
```

Using malloc

Let's update our simple code from before to use malloc, such that we can safely return the poiner.

```
int* func()
{
  int *pointer;
  pointer = (int *)malloc(sizeof(int));
  if (pointer == 0)
  {
    // Couldn't malloc, probably out of memory
    return 0;
  }
  *pointer = 7
  return pointer;
}
```

More memory management

There way more to memory management than just using malloc, however you should look into this yourself.

- Use free(pointer) to free memory after you're doing using item
- Using malloc to create dynamically sized arrays (not strictly needed after C99) or other data structures
- Malloc does not initialize the memory to 0. Use `calloc` for that (slower)
- `realloc` to change the size of already malloc-ed memory

Use `man` to find out more!

pthread_create

```
int pthread_create(
  pthread_t *thread,
  const pthread_attr_t *attr,
  void *(*start_routine) (void *),
  void *arg
);
```

`*thread`: Pointer to a `pthread_t` struct at which a data about the thread will be stored.

pthread_create

```
int pthread_create(
  pthread_t *thread,
  const pthread_attr_t *attr,
  void *(*start_routine) (void *),
  void *arg
);
```

`*thread`: Pointer to a `pthread_t` struct at which a data about the thread will be stored.

`*attr`: A pointer to a `pthread_attr_t` struct with parameters for the thread. If you have no parameters to pass, you can set this to NULL.

pthread_create

```
int pthread_create(
  pthread_t *thread,
  const pthread_attr_t *attr,
  void *(*start_routine) (void *),
  void *arg
);
```

`*thread`: Pointer to a `pthread_t` struct at which a data about the thread will be stored.

`*attr`: A pointer to a `pthread_attr_t` struct with parameters for the thread. If you have no parameters to pass, you can set this to NULL.

`(*start_routine)`: A function pointer to a function that takes one arg of type void* and has a return value of void*.

pthread_create

```
int pthread_create(
  pthread_t *thread,
  const pthread_attr_t *attr,
  void *(*start_routine) (void *),
  void *arg
);
```

`*thread`: Pointer to a `pthread_t` struct at which a data about the thread will be stored.
`*attr`: A pointer to a `pthread_attr_t` struct with parameters for the thread. If you have no parameters to pass, you can set this to NULL.
`(*start_routine)`: A function pointer to a function that takes one arg of type void* and has a return value of void*.
`*arg`: Pointer to the argument for the above function.

`void*` pointer?

A pointer to void is basically a “generic” pointer.