

Make your Program Better: Build a Multi-Threader!

- ◆ Comparison of threads and processes
- ◆ Pointers to Functions
- ◆ Creating new threads
- ◆ Exiting threads
- ◆ Waiting for threads
 - ◆ Automatic detachment.
- ◆ Thread semaphores

Multi-threading and Multi-processing

Multi-threading borrows much from the multi-processing paradigm. Most of your favorite multi-processing concepts have a multi-threaded counterpart...

`fork()` ←————→ `pthread_create(...)`

`exit(...)` ←————→ `pthread_exit(...)`

`wait(...)` ←————→ `pthread_join(...)`

Multi-threading and Multi-processing

... however, one will also find several differences between the two paradigms. In particular:

- Because threads all run through a single process, all of your memory is shared by default.
- Certain C functions can no longer be called (at least safely):
 - If a function returns a pointer to a static memory location, there is a chance that two threads call the function in succession and corrupt each other's function call.
 - Some system level functions will have unexpected behaviors. For example, calling `sleep()` will put all of your threads to sleep (not just the calling thread)
- Many functions will now appear to require several “useless” parameters. These extra parameters are actually required to circumvent the static return value problem.
- Threaded functions no longer set the `errno` value when an error occurs. In other words, you can no longer print error messages with `perror()`.

Pointers to Functions

One can create pointers to functions in C (and C++!). In general, these are useful for creating flexible interfaces that hide most of the details from the programmers. The designers of the `pthread`s library used pointers to functions to help control program flow.

```
typedef void* (*P_FUNK)(void *)
```



This is the return value for the function pointer.



These are the parameters of the function pointer.

Creating Threads

`pthread_create` is like the `fork()` system call on steroids:

```
int pthread_create(  
    pthread_t *thread, ←
```

```
    pthread_attr_t *attr, ←
```

```
    P_FUNC func, ←
```

```
    void *arg ←
```

```
)
```

thread will contain the unique id of the new thread (much like how `fork()` returned the pid).

Controls the scheduling for the thread. We probably won't deal with this, so set this to `NULL`.


The new thread will jump immediately to this function. This helps you control program flow.

These are the arguments for `func`. This way, you can pass the new threads only the variables they need.

Exiting Threads

`pthread_exit` is similar to the `exit()` system call, except it now expects a pointer to a void to store the exit status of the thread:

```
void pthread_exit(void *status)
```



Contains the exit status for the thread. You can return NULL to signify a “don't care” value.

Joining Threads

`pthread_join` is the thread equivalent of `wait()` (not the semaphore `wait` – the other one used to wait for child processes)

```
int pthread_join(pthread_t id, void **status)
```

This is the id of the thread that you want to wait on [ie, the `*thread` from the call to `pthread_create(...)`]

Returns 0 on success, or non-zero for any error (check the return value to find the error message)

This will contain the return status of the thread that you wait on. I'm not sure why it needs to be a pointer to a pointer, but I'm sure that together we can call get through this.

Detaching Threads

Usually when a thread exits, some of its resources are not removed from the system until another thread calls `pthread_join()`. Sometimes, however, you might not care about joining the process and would rather just have it destroy all of its resources automatically on exit. This can be accomplished with the following call:

```
int pthread_detach(pthread_t id)
```

Returns 0 on Success,
or a non-zero error
value on Failure.

This is the id of the thread
that you want to clean up
automatically.

Threaded Semaphores!

That's right: semaphores are available for threads. More importantly,

**The semaphore
functions actually
make sense!**

Creating Semaphores

Since threads can be passed specific arguments, there is no longer a need to create semaphores in groups. Semaphores can be created as follows:

```
int sem_init(sem_t *sem, int pshared, unsigned int value)
```

↑
-1 on
Failure,
and sets
errno

↑
A pointer to the new
semaphore that you
want to create.

↑
Setting this to non-
zero signifies that
your semaphores
will also be shared
between processes.
Just set this to 0.

↑
Set this to
the initial
value that
you want
for the
semaphore.

Example:

```
sem_t *TicketBooth = (sem_t *) malloc(sizeof(sem_t));  
sem_init(TicketBooth, 0, 8);
```

Waiting on Semaphores

```
int sem_wait(sem_t *sem)
```

-1 on
Failure,
and sets
errno

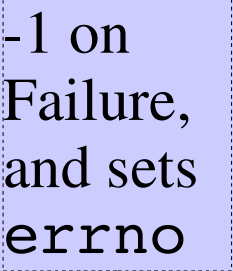
A pointer to the
semaphore you
want to wait on.

This is empty space. Reading the Athenian has taught me that empty space causes cancer or something, so try not to look at this corner.

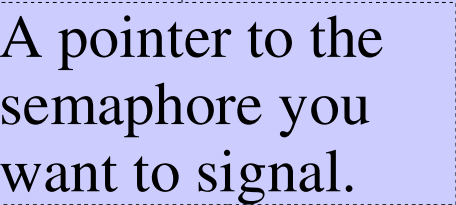
Signaling on Semaphores

```
int sem_post(sem_t *sem)
```

-1 on
Failure,
and sets
errno



A pointer to the
semaphore you
want to signal.



You were warned. Now
you have the cancer.

A few final notes:

- ◆ `#include <pthread.h>` for threads
- ◆ `#include <semaphores.h>` for thread semaphores
- ◆ You will need to link the pthread library into your program explicitly at compile time:

```
gcc -lpthread [... other options ...]
```