Threads

Consider here a program which must perform some expensive operation (here mocked up by simply calling `sleep` to wait for one second) several times:

sequential.c

```
#include <stdlib.h>
#include <stdio.h>
#include <unistd.h>

#define COUNT 5

void expensive_operation(void)
{
        printf("Beginning expensive operation.\n");
        sleep(1);
        printf("Done with expensive operation.\n");
}

int main(int argc, char **argv)
{
        int i;

        for (i = 0; i < COUNT; ++i)
                expensive_operation();

        return EXIT_SUCCESS;
}
```

This program takes five seconds, and prints:

```
$ ./sequential
Beginning expensive operation.
Done with expensive operation.
Beginning expensive operation.
Done with expensive operation.
Beginning expensive operation.
Done with expensive operation.
Beginning expensive operation.
Done with expensive operation.
Beginning expensive operation.
Done with expensive operation.
```

If it is possible to perform the expensive operation in parallel (because, for instance, the machine has multiple cores, or, as in this case, if the "work" is actually just waiting for something like a timer or network peer), then there is significant advantage to using threads.

On Linux, threading is provided by the `pthread` library. The functions and types are to be found in `pthread.h`, and when compiling threaded programs you must pass the `-pthread` option to `gcc`.

This example does the exact same thing as the sequential version, but runs each invocation of `expensive_operation` in a separate thread with `pthread_create`, then just waits for them to finish with `pthread_join`.

A thread begins with a function call, always to a function that takes a `void *` and returns a `void *` as well; this allows arbitrary data to be passed into and out of threads without the thread library caring what it is. Note that we're seeing function pointers again!

threaded.c

```
1  #include <stdlib.h>
2  #include <stdio.h>
3  #include <unistd.h>
4  #include <pthread.h>

6  #define COUNT 5

8  void *expensive_operation(void *arg)
9  {
10         printf("Beginning expensive operation.\n");
11         sleep(1);
12         printf("Done with expensive operation.\n");
13         return NULL;
14 }

16 int main(int argc, char **argv)
17 {
18         pthread_t tid[COUNT];
19         int i;

21         for (i = 0; i < COUNT; ++i)
22                 pthread_create(&tid[i], NULL, expensive_operation, NULL);
23         for (i = 0; i < COUNT; ++i)
24                 pthread_join(tid[i], NULL);

26         return EXIT_SUCCESS;
27 }
```

This program does a much different thing from the sequential version.

```
$ gcc -Wall -g -pthread -o threaded threaded.c
$ ./threaded
Beginning expensive operation.
Beginning expensive operation.
Beginning expensive operation.
Beginning expensive operation.
```

```
Beginning expensive operation.
Done with expensive operation.
Done with expensive operation.
Done with expensive operation.
Done with expensive operation.
Done with expensive operation.
```

Note that all of the operations begin at about the same time and all end at about the same time, and the program only runs for about one second rather than five.

Threading is not all fun and games, however. Consider this sequential program that updates a variable, again using `sleep` to simulate doing a lot of work.

increment.c

```
1  #include <stdlib.h>
2  #include <stdio.h>
3  #include <unistd.h>

5  #define COUNT 5

7  int *increment(int *x)
8  {
9          int y;

11         printf("Beginning increment.\n");
12         y = *x;
13         sleep(1);
14         *x = y + 1;

16         printf("Done with increment.\n");
17         return x;
18 }

20 int main(int argc, char **argv)
21 {
22         int x = 0, i;

24         printf("x = %d\n", x);

26         for (i = 0; i < COUNT; ++i)
27                 increment(&x);

29         printf("x = %d\n", x);

31         return EXIT_SUCCESS;
32 }
```

The result, unsurprisingly, is that `x` is incremented five times over five seconds.

```
$ ./increment
x = 0
Beginning increment.
Done with increment.
Beginning increment.
Done with increment.
Beginning increment.
Done with increment.
Beginning increment.
Done with increment.
Beginning increment.
Done with increment.
x = 5
```

Let us do the same thing with threads. The only new quirk is showing how to pass arguments to threads, and some casting at the call to avoid casting within the function.

race.c

```
1  #include <stdlib.h>
2  #include <stdio.h>
3  #include <unistd.h>
4  #include <pthread.h>

6  #define COUNT 5

8  int *increment(int *x)
9  {
10         int y;

12         printf("Beginning increment.\n");
13         y = *x;
14         sleep(1);
15         *x = y + 1;

17         printf("Done with increment.\n");
18         return x;
19 }

21 int main(int argc, char **argv)
22 {
23         pthread_t tid[COUNT];
24         int x = 0, i;

26         printf("x = %d\n", x);

28         for (i = 0; i < COUNT; ++i)
29                 pthread_create(&tid[i], NULL, (void *(*)(void *))increment, &x);
30         for (i = 0; i < COUNT; ++i)
```

```
31                  pthread_join(tid[i], NULL);

33          printf("x = %d\n", x);

35          return EXIT_SUCCESS;
36  }
```

However, this one does not behave quite as expected. It runs in only one second, but finds the wrong answer!

```
$ ./race
x = 0
Beginning increment.
Beginning increment.
Beginning increment.
Beginning increment.
Beginning increment.
Done with increment.
Done with increment.
Done with increment.
Done with increment.
Done with increment.
x = 1
```

This weird behavior is shown because this program has what is called a "race condition", in which the way the threads are scheduled can change the result. The problem is that all of the threads get the same initial value to work from, and thus all update `x` to the same thing, rather than taking turns.

The only solution is to reintroduce sequential behavior in this "critical section". Threads can explicitly synchronize certain parts of code using mutexes. A mutex (short for "mutual exlusion") is a sort of a lock, which can be held by only one thread at a time. By locking the mutex at the beginning of a critical section and unlocking it at the end, all of the threads guarantee that no two are in the critical section at once:

mutex.c

```
1   #include <stdlib.h>
2   #include <stdio.h>
3   #include <unistd.h>
4   #include <pthread.h>

6   #define COUNT 5

8   pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;

10  int *increment(int *x)
11  {
12          int y;
```

```
14          printf("Beginning increment.\n");
15          pthread_mutex_lock(&mutex);
16          y = *x;
17          sleep(1);
18          *x = y + 1;
19          pthread_mutex_unlock(&mutex);

21          printf("Done with increment.\n");
22          return x;
23  }

25  int main(int argc, char **argv)
26  {
27          pthread_t tid[COUNT];
28          int x = 0, i;

30          printf("x = %d\n", x);

32          for (i = 0; i < COUNT; ++i)
33                  pthread_create(&tid[i], NULL, (void *(*)(void *))increment, &x);
34          for (i = 0; i < COUNT; ++i)
35                  pthread_join(tid[i], NULL);

37          printf("x = %d\n", x);

39          return EXIT_SUCCESS;
40  }
```

Now we're back to five seconds but the right answer.

```
$ ./mutex
x = 0
Beginning increment.
Beginning increment.
Beginning increment.
Beginning increment.
Beginning increment.
Done with increment.
Done with increment.
Done with increment.
Done with increment.
Done with increment.
x = 5
```

Note that this is not quite the same as the sequential program, because they're still threaded. Only the critical section, which is explicitly synchronized, is back to running sequentially. Try adding another `sleep` before the critical section in both the sequential and threaded versions to see the difference!

All of this only scratches the barest surface of threading and synchronization, but you can go a long way with just thread creation, joining, and mutexes.