Pthreads, Part 2: Usage in Practice

Thomas Liu edited this page on Jan 25 · 1 revision

How do I create a pthread?

See Pthreads Part 1 which introduces `pthread_create` and `pthread_join`

If I call `pthread_create` twice how many stacks does my process have?

Your process will contain three stacks - one for each thread. The first thread is created when the process starts and you created two more. Actually there can be more stacks than this but let's ignore that complication for now. The important idea is that each thread requires a stack because the stack contains automatic variables and the old CPU PC register so that it can back to executing the calling function after the function is finished.

What is the difference between a process and a thread?

In addition, unlike processes, threads within the same process can share the same global memory (data and heap segments).

What does `pthread_cancel` do?

Stops a thread. Note the thread may not actually be stopped immediately. For example it can be terminated when the thread makes an operating system call (e.g. `write`).

In practice `pthread_cancel` is rarely used because it does not give a thread an opportunity to clean up after itself (for example, it may have opened some files). An alternative implementation is to use a boolean (int) variable whose value is used to inform other threads that they should finish and clean up.

What is the difference between `exit` and `pthread_exit`?

`exit(42)` exits the entire process and sets the processes exit value. This is equivalent to `return 42` in the main method. All threads inside the process are stopped.

`pthread_exit(void *)` only stops the calling thread i.e. the thread never returns after calling `pthread_exit`. The pthread library will automatically finish the process if there are no other threads running. `pthread_exit(...)` is equivalent to returning from the thread's function; both finish the thread and also set the return value (void *pointer) for the thread.

Calling `pthread_exit` in the the `main` thread is a common way for simple programs to ensure that all threads finish. For example, in the following program, the `myfunc` threads will probably not have time to get started.

```
int main() {
  pthread_t tid1, tid2;
  pthread_create(&tid1, NULL, myfunc, "Jabberwocky");
  pthread_create(&tid2, NULL, myfunc, "Vorpel");
  exit(42); //or return 42;

  // No code is run after exit
}
```

The next two programs will wait for the new threads to finish-

```
int main() {
  pthread_t tid1, tid2;
  pthread_create(&tid1, NULL, myfunc, "Jabberwocky");
  pthread_create(&tid2, NULL, myfunc, "Vorpel");
  pthread_exit(NULL);

  // No code is run after pthread_exit
  // However process will continue to exist until both threads have finished
}
```

Alternatively, we join on each thread (i.e. wait for it to finish) before we return from main (or call exit).

```
int main() {
  pthread_t tid1, tid2;
  pthread_create(&tid1, NULL, myfunc, "Jabberwocky");
  pthread_create(&tid2, NULL, myfunc, "Vorpel");
  // wait for both threads to finish :
  void* result;
  pthread_join(tid1, &result);
  pthread_join(tid2, &result);
  return 42;
}
```

Note the pthread_exit version creates thread zombies, however this is not a long-running processes, so we don't care.

How can a thread can be terminated?

- Returning from the thread function
- Calling `pthread_exit`
- Cancelling the thread with `pthread_cancel`
- Terminating the process (e.g. SIGTERM); exit(); returning from `main`

What is the purpose of pthread_join?

- Wait for a thread to finish
- Clean up thread resources.

What happens if you don't call `pthread_join`?

Finished threads will continue to consume resources. Eventually, if enough threads are created, `pthread_create` will fail. In practice, this is only an issue for long-runnning processes but is not an issue for simple, short-lived processes as all thread resources are automatically freed when the process exits.

Should I use `pthread_exit` or `pthread_join`?

Both `pthread_exit` and `pthread_join` will let the other threads finish on their own (even if called in the main thread). However, only `pthread_join` will return to you when the specified thread finishes. `pthread_exit` does not wait and will immediately end your thread and give you no chance to continue executing.

Can you pass pointers to stack variables from one thread to another?

Yes. However you need to be very careful about the lifetime of stack variables.

```
pthread start_threads() {
  int start = 42;
  pthread_t tid;
  pthread_create(&tid, 0, myfunc, &start); // ERROR!
  return tid;
}
```

The above code is invalid because the function `start_threads` will likely return before `myfunc` even starts. The function passes the address-of `start`, however by the time `myfunc` is executes, `start` is no longer in scope and its address will re-used for another variable.

The following code is valid because the lifetime of the stack variable is longer than the background thread.

```
void start_threads() {
  int start = 42;
  void *result;
  pthread_t tid;
  pthread_create(&tid, 0, myfunc, &start); // OK - start will be valid!
  pthread_join(tid, &result);
}
```

How can I create ten threads with different starting values.

The following code is supposed to start ten threads with values 0,1,2,3,...9 However, when run prints out `1 7 8 8 8 8 8 8 8 10`! Can you see why?

```
#include <pthread.h>
void* myfunc(void* ptr) {
    int i = *((int *) ptr);
    printf("%d ", i);
    return NULL;
}

int main() {
    // Each thread gets a different value of i to process
    int i;
    pthread_t tid;
    for(i =0; i < 10; i++) {
        pthread_create(&tid, NULL, myfunc, &i); // ERROR
    }
    pthread_exit(NULL);
}
```

The above code suffers from a `race condition` - the value of i is changing. The new threads start later (in the example output the last thread starts after the loop has finished).

To overcome this race-condition, we will give each thread a pointer to it's own data area. For example, for each thread we may want to store the id, a starting value and an output value:

```
struct T {
  thread_t id;
  int start;
  char result[100];
}
```

These can be stored in an array -

```
struct T *info = calloc(10 , sizeof(struct T)); // reserve enough bytes for ten T
```

And each array element passed to each thread -

```
pthread_create(&info[i].id, NULL, func, &info[i]);
```

Why are some functions e.g. asctime,getenv, strtok, strerror not thread-safe?

To answer this, let's look at a simple function that is also not 'thread-safe'

```
char *to_message(int num) {
    char static result [256];
    if (num < 10) sprintf(result, "%d : blah blah" , num);
    else strcpy(result, "Unknown");
    return result;
}
```

In the above code the result buffer is stored in global memory. This is good - we wouldn't want to return a pointer to an invalid address on the stack, but there's only one result buffer in the entire memory. If two threads were to use it at the same time then one would corrupt the other:

Time	Thread 1	Thread 2	Comments
1	to_m(5)		
2		to_m(99)	Now both threads will see "Unknown" stored in the result buffer

What are condition variables, semaphores, mutexes?

These are synchronization locks that are used to prevent race conditions and ensure proper synchronization between threads running in the same program. In addition these locks are conceptually identical to the primitives used inside the kernel.

Are there any advantages of using threads over forking processes?

Yes! Sharing information between threads is easy because threads (of the same process) live inside the same virtual memory space. Also, creating a thread is significantly faster than creating(forking) a process.

Are there any dis-advantages of using threads over forking processes?

Yes! No- isolation! As threads live inside the same process, one thread has access to the same virtual memory as the other threads. A single thread can terminate the entire process (e.g. by trying to read address zero).

Can you fork a process with multiple threads?

Yes! However the child process only has a single thread (which is a clone of the thread that called `fork`. We can see this as a simple example, where the background threads never print out a second message in the child process.

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

static pid_t child = -2;

void *sleepnprint(void *arg) {
  printf("%d:%s starting up...\n", getpid(), (char *) arg);

  while (child == -2) {sleep(1);} /* Later we will use condition variables */
  printf("%d:%s finishing...\n",getpid(), (char*)arg);

  return NULL;
}
int main() {
  pthread_t tid1, tid2;
  pthread_create(&tid1,NULL, sleepnprint, "New Thread One");
  pthread_create(&tid2,NULL, sleepnprint, "New Thread Two");

  child = fork();
  printf("%d:%s\n",getpid(), "fork()ing complete");
  sleep(3);

  printf("%d:%s\n",getpid(), "Main thread finished");

  pthread_exit(NULL);
  return 0; /* Never executes */
}
```

```
8970:New Thread One starting up...
8970:fork()ing complete
8973:fork()ing complete
8970:New Thread Two starting up...
8970:New Thread Two finishing...
8970:New Thread One finishing...
8970:Main thread finished
8973:Main thread finished
```

In practice creating threads before forking can lead to unexpected errors because (as demonstrated above) the other threads are immediately terminated when forking. Another thread might have just lock a mutex (e.g. by calling malloc) and never unlock it again. Advanced users may find `pthread_atfork` useful however we suggest you usually try to avoid creating threads before forking unless you fully understand the limitations and difficulties of this approach.

Are there other reasons where `fork` might be preferable to creating a thread.

Creating separate processes is useful

- When more security is desired (for example, Chrome browser uses different processes for different tabs)
- When running an existing and complete program then a new process is required (e.g. starting 'gcc')

How can I find out more?

See the complete example in the man page And the pthread reference guide ALSO: Concise third party sample code explaining create, join and exit