Lecture 14 — UNIX Processes and POSIX Threads (pthread)

Jeff Zarnett
jzarnett@uwaterloo.ca

Department of Electrical and Computer Engineering
University of Waterloo

June 5, 2024

UNIX Workflow

Parent spawns the child process with the `fork` system call.

If waiting for the child process to finish, `wait`.
    Alternatively, carry on.

When the child process is finished, it returns a value with `exit`

The parent gets this as the return value of `wait` and may proceed.

About `fork`

Note: `fork` creates a new process as a copy of itself.

Both parent and child continue after that statement.

The call `fork` can return a value:
  A negative value means the fork failed.
  A zero value means this process is the child.
  A positive value: this is the parent; the value is the child `pid`.

After the `fork`, the `exec`

After the `fork`, one of the processes may use the `exec` system call.

This will replace its memory space with a new program.

There's no rule that says this must happen!
  a child can continue to be a clone of its parent if it wishes.

After the `fork`, the `exec`

The `exec` invocation loads a binary file into memory & starts execution.

At this point, the programs can go their separate ways.

Or the parent might want to wait for the child to finish.

Putting it Together

```c
int main( int argc, char** argv ) {
  pid_t pid;
  int childStatus;

  /* fork a child process */
  pid = fork();

  if (pid < 0) {
    /* error occurred */
    fprintf(stderr, "Fork␣Failed");
    return 1;

 } else if (pid == 0) {
    /* child process */
    execlp("/bin/ls","ls",NULL);

  } else {
    /* parent process */
    /* parent will wait for the child to complete */
    wait(&childStatus);
    printf("Child␣Complete␣with␣status:␣%i␣\n", childStatus);

  }

  return 0;
}
```

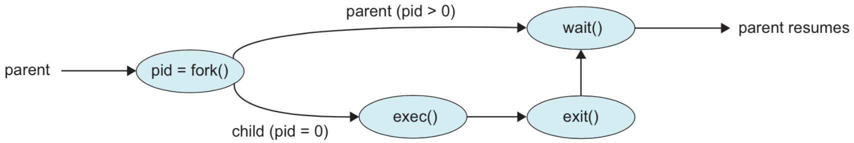
Code Output

Thus, the output is:

```
jz@Freyja:~/fork$ ./fork
fork   fork.c
Child Complete with status: 0
jz@Freyja:~/fork$
```

Fork Visually

Or, to represent this visually:



Termination?

What about termination?

On the assumption that the process is terminating normally and not being killed, the system call for that is `exit`.

If the program itself has no explicit call to `exit`, the `return` statement at the end of `main` will have the same effect.

Termination?

The `wait` function also returns the process ID of the child.

This is so that the parent can identify which of its children has terminated, though it is not used in this example.

After the `fork`

Afterwards, the system will need to choose which process is going to run:

1. The parent process. The child is in the ready to run state.
2. The child process. The parent is in the ready to run state.
3. Another process. Both parent and child are in the ready to run state.

Fork Design Problem

There is a task that can be split into parts ’A’ and ’B’.

Use `fork()` to create a child process.

The child process should call function `execute_B()` and return the result to the parent.

The parent process should call `execute_A()` and collect its result.

The parent should then collect the result of the child using `wait()` and then produce the console output.

Fork Design Problem

If no errors occurred, `main` should return `0`; otherwise it should return -1.

If an error occurs, it should be reported to the console including the error number (e.g., “Error 7 Occurred.”).

If more than one error occurs, report both errors.

If both functions return zero, it means all is well and the program should print “Completed.” to the console.

Fork Design Solution

```
int main( int argc, char** argv ) {
  int child_result;
  int parent_result;

  pid_t pid = fork();
  if ( pid < 0 ) { /* Fork Failed */
    return -1;
  } else if ( pid == 0 ) { /* Child */
    return execute_B();
  } else { /* Parent */
    parent_result = execute_A();
    wait( &child_result );
  }

  if ( child_result == 0 && parent_result == 0 ) {
    printf( "Completed.\n" );
    return 0;
  }

  if ( child_result != 0 ) {
    printf( "Error␣%d␣Occurred.\n", child_result);
  }
  if ( parent_result != 0 ) {
    printf( "Error␣%d␣Occurred.\n", parent_result);
  }
  return -1;
}
```

Après `fork`, le déluge

A short digression on a denial of service attack: the “fork bomb”.

The idea is to call `fork` repeatedly.

Keep doing this until the system crashes (or no work can get done).

Exponential growth (2^n) processes after n calls.

Fork Bomb

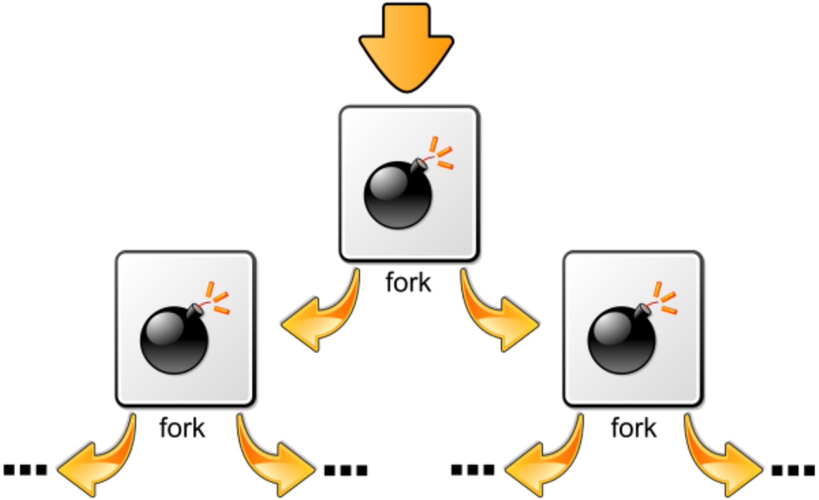


Image Credit: Wikipedia user Dake

Après `fork`, le déluge

A system can be configured to defend against this.

1. Limit total number of processes per user.

2. Limit rate of process spawning.

Note: do not attempt this on University computers!

The POSIX Thread

The term `pthread` refers to the POSIX standard (also known as the IEEE 1003.1c standard) that defines thread behaviour in UNIX.

Wait, why are we talking about this?!

This is an embedded systems type course…

The POSIX Thread

The pthread model is very common and used all across industry.

Knowing about this might seriously help you get a co-op job.

It will also help you if you want to take ECE 459 in the future...
  Statistically, many of you do!

The POSIX Thread

- `pthread_create`
- `pthread_exit`
- `pthread_join`
- `pthread_detach`
- `pthread_yield`
- `pthread_attr_init`
- `pthread_attr_destroy`
- `pthread_cancel`
- `pthread_testcancel`

Let’s Make a New Thread

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

`thread`: a pointer to a `pthread` identifier and will be assigned a value when the thread is created.

`attr`: attributes; may be `NULL` for defaults.

`start_routine`: the function the new thread is to run.

`arg`: The argument passed to the routine we want to start.

Start Routine

The type of `start_routine` above is a function signature.

Thus, the `pthread_create` function has to be called with the name of a function matching that signature, such as:

```
void* do_something( void* start_params )
```

After creating a new thread, the process has two threads in it.

Scheduling of the threads is up to the operating system.

There Can Be Only One

C: it is normal to have a single return value from a function, but usually we can have multiple input parameters.

But here we get only one of each?

Define a `struct` for the argument and return type!

```
void* function( void * void_arg ) {
  parameters_t *arguments = (parameters_t*) args;
  /* continue after this */
}
```

We have to cast it inside the thread anyway...

The caller of the `pthread_create` function has to know what kind of argument is expected in the function being called.

Attributes

Attributes can be used to set whether a thread is detached or joinable, scheduling policy, etc.

By default, new threads are usually joinable (that is to say, that some other thread can call `pthread_join` on them).

To prevent a thread from ever being joined, it can be created in the detached state (or use `pthread_detach`)

For virtually all scenarios that we will consider in this course the default values will be fine.

Threadception

There is no mandatory hierarchy of threads.

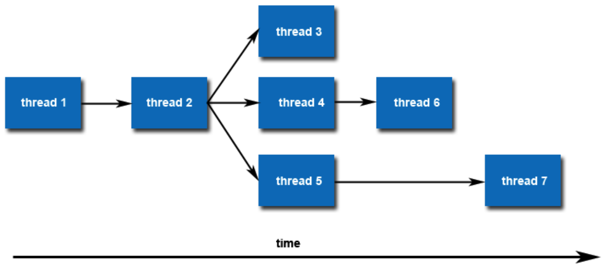


Image Credit: Blaise Barney

New threads can create other threads.

Off We Go

The thread executes its function, until of course it gets to the end.

Usually, it will terminate with `pthread_exit`.

The use of `pthread_exit` is not the only way that a thread may be terminated.

Sometimes we want the thread to persist (hang around), but if we want to get a return value from the thread, then we need it to exit.

Nobody’s Listening

If a thread has no return values, it can just `return NULL;`

This will send `NULL` back to the thread that has joined it.

If the function that is called as a task returns normally rather than calling the exit routine, the thread will still be terminated.

Oh... Guess You Didn’t Need This After All

Another way a thread might terminate is if the `pthread_cancel` function.
  We’ll come back to this topic in more detail soon.

A thread may also be terminated indirectly: if the entire process is terminated or if `main` finishes first (without calling `pthread_exit` itself).

End `main` with `pthread_exit` to automatically wait for all spawned threads.

Report, Number One!

Report, Number One!

Like the `wait` system call, the `pthread_join` is how we get a value out of the spawned thread:

```
pthread_join( pthread_t thread, void** retval );
```

`thread`: the thread you wish to join.

`retval`: wait... two stars?

Gotta Play the Level Again, Only Got 2 Stars

What we are looking for is a pointer to a void pointer.

That is, we are going to supply a pointer that the join function will update to be pointing to the value returned by that function.

Typically we supply the address of a pointer.

Maybe the example makes it clearer.

Collecting Return Value

```c
#include <stdlib.h>
#include <stdio.h>
#include <pthread.h>

void * run( void * argument ) {
  char* a = (char*) argument;
  printf("Provided␣argument␣is␣%s!\n", a);
  int * return_val = malloc( sizeof( int ));
  *return_val = 99;
  pthread_exit( return_val );
}

int main( int argc, char** argv ) {
  if (argc != 2) {
      printf("Invalid␣args.\n");
      return -1;
  }
  pthread_t t;
  void* vr;

  pthread_create( &t, NULL, run, argv[1] );
  pthread_join( t, &vr );
  int* r = (int*) vr;
  printf("The␣other␣thread␣returned␣%d.\n", *r);
  free( vr );
  pthread_exit( 0 );
}
```

Ah! Ah! Ah!

Next, we'll do an example where we don't use the return value of a thread, but do use attributes.

For the sake of simplicity: we are just going to count!

Slightly Larger Example

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

int sum; /* Shared Data */

void *runner(void *param);

int main( int argc, char **argv ) {
  pthread_t tid; /* the thread identifier */
  pthread_attr_t attr; /* set of thread attributes */

  if ( argc != 2 ) {
    fprintf(stderr,"usage: %s <integer value>\n", argv[0]);
    return -1;
  }
  if ( atoi( argv[1] ) < 0 ) {
    fprintf(stderr, "%d must be >= 0\n", atoi(argv[1]));
    return -1;
  }

  pthread_attr_init( &attr ); /* set the default attributes */
  pthread_create( &tid, &attr, runner, argv[1] ); /* create the thread */
  pthread_join( tid, NULL );
  printf( "sum = %d\n", sum );
  pthread_attr_destroy( &attr );
  pthread_exit( NULL );
}
```

Slightly Larger Example

```
void *runner( void *param ) {
  int upper = atoi( param );
  sum = 0;
  for ( int i = 1; i <= upper; i++ ) {
    sum += i;
  }
  pthread_exit( 0 );
}
```

In this example, both threads are sharing the global variable `sum`.

Do we have coordination?

Coordination, Count to 10

Yes! The parent thread will join the newly-spawned thread (i.e., wait until it is finished) before it tries to print out the value.

If it did not, the parent would print the sum early.

Let’s do a different take on that program.

When you watch Sesame Street and can now count to 20.
My powers have doubled since the last time we met, Count.

Slightly Larger Example 2

```c
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>

int sum = 0;

void* runner( void *param ) {
    int upper = atoi( param );
    for (int i = 1; i <= upper; i++ ) {
        sum += i;
    }
    pthread_exit( 0 );
}
```

Slightly Larger Example 2

```
int main( int argc, char** argv ) {

    pthread_t tid[3];

    if ( argc != 2 ) {
        printf("An␣integer␣value␣is␣required␣as␣an␣argument.\n");
        return -1;
    }
    if ( atoi( argv[1]) < 0 ) {
        printf( "%d␣must␣be␣>=␣0.\n", atoi(argv[1]) );
    }

    for ( int i = 0; i < 3; ++i ) {
        pthread_create( &tid[i], NULL, runner, argv[1] );
    }
    for ( int j = 0; j < 3; ++j ) {
        pthread_join( tid[j], NULL );
    }
    printf( "sum␣=␣%d.\n", sum );

    pthread_exit( 0 );
}
```

What happens when we run this program?

Sometimes...

For very small values of the argument, nothing goes wrong.

For a large number we get some strange and inconsistent results. Why?

There are three threads that are modifying `sum`.

Remember what we said about “at the same time”? We have to come back to it.

Thread Cancellation

Thread cancellation is exactly what it sounds like: a running thread will be terminated before it has finished its work.

The thread that we are going to cancel is called the *target*.

YOU TOLD ME MY FLIGHT WAS ON TIME
THE CANCELLED FLIGHT DETERMINED THAT WAS A LIE
memegenerator.net

Cancellation Type

1. **Asynchronous Cancellation**
2. **Deferred Cancellation**

thread can declare its own cancellation type through the use of the function:

```
pthread_setcanceltype( int type, int *oldtype )
```

`type`: `PTHREAD_CANCEL_DEFERRED` or `PTHREAD_CANCEL_ASYNCHRONOUS`

`oldtype`: previous state, if we care.

Send Cancellation

The pthread command to cancel a thread is `pthread_cancel` and it takes one parameter (the thread identifier).

To check if the current thread has been cancelled, the function call is `pthread_testcancel` which takes no parameters.

It's polite to check this, if it's a risk.

Cancellation Points

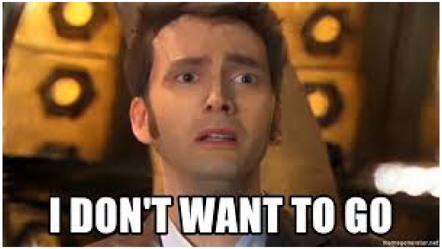
A large number of functions are cancellation points.

That is, the POSIX specification requires there is an implicit check for cancellation when calling one of those functions.

Even more are “potential cancellation points” – maybe, maybe not?

Not Now!

Sometimes a thread could die before it has cleaned up.



This can leave memory allocated, things locked...

Cleanup Handler

We can prevent this with cancellation handlers.

The functions for cleaning up are:

```
/* Register cleanup handler, with argument */
pthread_cleanup_push( void (*routine)(void*), void *argument );
/* Run if execute is non-zero */
pthread_cleanup_pop( int execute );
```

The push function always needs to be paired with the pop function at the same level in your program (where level is defined by the curly braces).

Don’t You Forget About Me

Consider the following code:

```
void* do_work( void* argument ) {
  struct job * j = malloc( sizeof( struct job ) );
  /* Do something useful with this structure */
  /* Actual work to do not shown */
  free( j );
  pthread_exit( NULL );
}
```

Don’t You Forget About me

```c
void cleanup( void* mem ) {
  free( mem );
}

void* do_work( void* argument ) {
  struct job * j = malloc( sizeof( struct job ) );
  pthread_cleanup_push( cleanup, j );
  /* Do something useful with this structure */
  /* Actual work to do not shown */
  free( j );
  pthread_cleanup_pop( 0 ); /* Don't run */
  pthread_exit( NULL );
}
```