Lecture 5 — Creating Processes & Threads

Patrick Lam & Jeff Zarnett
p.lam@ece.uwaterloo.ca jzarnett@uwaterloo.ca

Department of Electrical and Computer Engineering
University of Waterloo

December 22, 2015

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.

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.

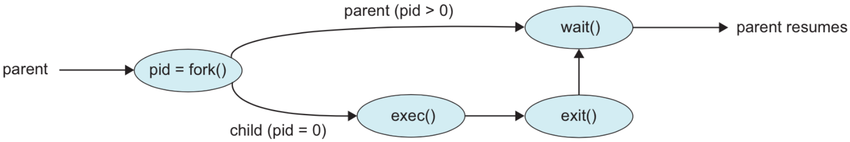
```
int main()
{
  pid_t pid;
  int childStatus;

  /* 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;
}
```

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:



Assumptions

First, we'll see how to use threads on "embarrassingly parallel problems".

- mostly-independent sub-problems (little synchronization); and
- strong locality (little communication).

Later, we'll see:

- which problems can be parallelized (dependencies)
- alternative parallelization patterns
  (right now, just use one thread per sub-problem)

POSIX Threads

- Available on most systems
- Windows has Pthreads Win32, but I wouldn't use it; use Linux for this course
- API available by `#include <pthread.h>`
- Compile with pthread flag (`gcc -pthread prog.c -o prog`)

C++ 11 Threads

- Now part of the C++ standard (library)
- API available with `#include <thread>`
- Compile with flags:
  (`g++ -std=c++11 -pthread prog.c -o prog`)

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

thread: creates a handle to a thread at pointer location

attr: thread attributes (NULL for defaults, more details later)

start_routine: function to start execution

arg: value to pass to start_routine

returns 0 on success, error number otherwise
(contents of *thread are undefined)

Creating Threads—Pthreads Example

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

void* run(void*) {
  printf("In run\n");
}

int main() {
  pthread_t thread;
  pthread_create(&thread, NULL, &run, NULL);
  printf("In main\n");
}
```

Simply creates a thread and terminates
(usage isn't really right, as we'll see.)

Creating Threads—C++11 Example

```
#include <thread>
#include <iostream>

void run() {
  std::cout << "In run\n";
}

int main() {
  std::thread t1(run);
  std::cout << "In main\n";
  t1.join(); // hang in there...
}
```

Waiting for Threads

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

thread: wait for this thread to terminate (thread must be joinable).

retval: stores exit status of thread (set by `pthread_exit`) to the location pointed by *retval. If cancelled, returns `PTHREAD_CANCELED`. `NULL` is ignored.

returns 0 on success, error number otherwise.

Only call this one time per thread! Multiple calls on the same thread leads to undefined behaviour.

Waiting for Threads—Pthreads example

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

void* run(void*) {
  printf("In run\n");
}

int main() {
  pthread_t thread;
  pthread_create(&thread, NULL, &run, NULL);
  printf("In main\n");
  pthread_join(thread, NULL);
}
```

This now waits for the newly created thread to terminate.

Creating Threads—C++11 Example

```
#include <thread>
#include <iostream>

void run() {
  std::cout << "In run\n";
}

int main() {
  std::thread t1(run);
  std::cout << "In main\n";
  t1.join(); // aha!
}
```

Passing Data to Pthreads threads…Wrongly

Consider this snippet:

```
int i;
for (i = 0; i < 10; ++i)
  pthread_create(&thread[i], NULL, &run, (void*)&i);
```

This is a terrible idea. Why?

Passing Data to Pthreads threads…Wrongly

Consider this snippet:

```
int i;
for (i = 0; i < 10; ++i)
  pthread_create(&thread[i], NULL, &run, (void*)&i);
```

This is a terrible idea. Why?

1. The value of `i` will probably change before the thread executes
2. The memory for `i` may be out of scope, and therefore invalid by the time the thread executes

What about:

```
int i;
for (i = 0; i < 10; ++i)
  pthread_create(&thread[i], NULL, &run, (void*)i);

...

void* run(void* arg) {
  int id = (int)arg;
```

This is suggested in the book, but should carry a warning:

What about:

```
int i;
for (i = 0; i < 10; ++i)
  pthread_create(&thread[i], NULL, &run, (void*)i);

...

void* run(void* arg) {
  int id = (int)arg;
```

This is suggested in the book, but should carry a warning:

- Beware size mismatches between arguments: no guarantee that a pointer is the same size as an int, so your data may overflow.
- Sizes of data types change between systems. For maximum portability, just use pointers you got `from malloc`.

Passing Data to C++11 threads

It’s easier to get data to threads in C++11:

```
#include <thread>
#include <iostream>

void run(int i) {
  std::cout << "In run " << i << "\n";
}

int main() {
  for (int i = 0; i < 10; ++i) {
    std::thread t1(run, i);
    t1.detach(); // see the next slide...
  }
}
```

…but it's harder to get data back.
Use `async` and `future` abstractions:

```
#include <thread>
#include <iostream>
#include <future>

int run() {
  return 42;
}

int main() {
  std::future<int> t1_retval =
                  std::async(std::launch::async, run);
  std::cout << t1_retval.get();
}
```

Joinable threads (the default) wait for someone to call `pthread_join` before they release their resources.

Detached threads release their resources when they terminate, without being joined.

```
int pthread_detach(pthread_t thread);
```

thread: marks the thread as detached

returns 0 on success, error number otherwise.

Calling `pthread_detach` on an already detached thread results in undefined behaviour.

Thread Termination

```
void pthread_exit(void *retval);
```

retval: return value passed to function that calls `pthread_join`

start_routine returning is equivalent to calling `pthread_exit` with that return value;

`pthread_exit` is called implicitly when the `start_routine` of a thread returns.

There is no C++11 equivalent.

Attributes

By default, threads are *joinable* on Linux, but a more portable way to know what you’re getting is to set thread attributes. You can change:

- Detached or joinable state
- Scheduling inheritance
- Scheduling policy
- Scheduling parameters
- Scheduling contention scope
- Stack size
- Stack address
- Stack guard (overflow) size

Attributes—Example

```
size_t stacksize;
pthread_attr_t attributes;
pthread_attr_init(&attributes);
pthread_attr_getstacksize(&attributes, &stacksize);
printf("Stack size = %i\n", stacksize);
pthread_attr_destroy(&attributes);
```

Running this on a laptop produces:

```
jon@riker examples master % ./stack_size
Stack size = 8388608
```

Setting a thread state to joinable:

```
pthread_attr_setdetachstate(&attributes,
                            PTHREAD_CREATE_JOINABLE);
```

Detached Threads: Warning!

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

void* run(void*) {
  printf("In run\n");
}

int main() {
  pthread_t thread;
  pthread_create(&thread, NULL, &run, NULL);
  pthread_detach(thread);
  printf("In main\n");
}
```

When I run it, it just prints “In main”, why?

Detached Threads: Solution to Problem

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

void* run(void*) {
  printf("In run\n");
}

int main() {
  pthread_t thread;
  pthread_create(&thread, NULL, &run, NULL);
  pthread_detach(thread);
  printf("In main\n");
  pthread_exit(NULL); // This waits for all detached
                      // threads to terminate
}
```

Make the final call `pthread_exit` if you have any detached threads. (There is no C++11 equivalent.)

Threading Challenges

- Be aware of scheduling (you can also set affinity with pthreads on Linux).
- Make sure the libraries you use are **thread-safe**:
  - Means that the library protects its shared data.
- glibc reentrant functions are also safe: a program can have more than one thread calling these functions concurrently.
- **Example:** `rand_r` versus `rand`.