Threads 2: Creation & Termination

1. Introduction
2. **Creation and Termination**
   - **thread creation**
   - **Pthreads example**
   - **passing data to a Pthread**
   - **race condition example**
   - **thread termination methods**
   - **returning data from a Pthread**
   - **joinable/detached threads**
   - **thread IDs (TIDs)**
   - **Pthread attributes**
3. Synchronization (part a)
4. Synchronization (part b)
5. Miscellaneous

Thread Creation

`pthread_create` is the system call to create a new pthread:

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

- `thread` is the new thread's ID (returned via pointer)
- `attr` is an optional thread attributes structure; use `NULL` to create a thread with *default* attribute values
- `start_routine` is the name of the *function* that starts running in the new thread
- `arg` is the argument passed to `start_routine` (a `void*`)
- returns 0 on success, else a (positive) thread error number

Thread Creation (contd.)

A key aspect of thread creation is that a thread must begin running a particular *function*, and such functions must match a *specific prototype*.

The notation `void *(*start_routine) (void *)` means that the function supplied to `pthread_create()` must:

- take a single argument of type `void*` (i.e., any pointer type)
- return a value of type `void*` (i.e., any pointer type)

Note that `(*start_routine)` is C notation for a **function pointer**, but this is what results when you use a function *name* in a call.

Thread Creation (contd.)

The `start_routine` prototype requirements have the following implications for passing arguments to `start_routine`:

- if no arguments are required, use `NULL` as `arg`
- if a single argument is required, a pointer must be used and the desired argument obtained by *dereferencing* this pointer
- an array argument may be passed directly (since arrays are represented as pointers)
- if multiple arguments are required, define an appropriate `struct` and pass a pointer to an instance of the `struct`

The `start_routine` prototype requirements have the following implications for `start_routine`'s return value:

- the return pointer *must not reference stack addresses*
- if a return value is not required, return `NULL`

Pthreads Example

An example using two threads to simultaneously:

1. get commands from the terminal and send them to a server
2. receive output from the server and print it on the terminal

```
int main(...)
{
  ...
  int sockfd;
  ...connect to server, sockfd is FD...

  //Create second thread to read server data and print it out:
  pthread_t threadid;
  pthread_create(&threadid,NULL,handle_output,&sockfd);

  //Handle commands from the terminal in original thread:
  handle_commands(sockfd);
  ...
}
```

Pthreads Example (contd.)

```
void handle_commands(int sockfd)
{
  ...
  //Continue reading a command and sending to server, until read "bye":
  while(1) {
    fgets(buff,size,stdin);
    if (strcmp(buff,"bye\n") == 0) break;
    write(sockfd,buff,strlen(buff));
  }
  //Normal termination of entire process (all threads):
  exit(EXIT_SUCCESS);
}

void *handle_output (void *sockfd_ptr)
{
  //Recover sockfd, which was passed as pointer:
  int sockfd = *(int*)sockfd_ptr;  //Recover passed sockfd.
  ...
  while((nread = read(sockfd,buff,n)) > 0)
    write(1,buff,nread);
  ...
}
```

Passing Data to a Pthread

Starting a new Pthread is quite similar to calling a function.

However, with a function call you can pass data to the function via its *parameters*.

With Pthreads, the only mechanism to pass data/parameters to the new thread is via `pthread_create()`'s single `void* arg` parameter.

I.e., data must be passed to `start_routine` by passing a single *pointer* to a data object, which can be of *any type*.

In the example above, we saw how a file descriptor `int` was passed: by passing the address of the FD variable.

The FD had to then be recovered from the pointer by using the *dereference operator* (and compilers may require *casting* too).

Passing Data (contd.)

There is another, more subtle difference between passing data to a Pthread vs. to a function.

Because you are passing a *pointer* (i.e., memory address), that memory address must *remain valid and its contents unchanged* until the new thread is definitely done accessing the data.

Remember, however, that the OS controls when and for how long the thread calling `pthread_create()` and the newly created thread get run.

This means that a program cannot make any assumptions about when the new thread will have completed accessing the passed data during its execution!

Failing to consider this issue can lead to **race conditions**, which are very difficult to detect and diagnose.

Passed Data Race Condition Example

Server handling each client via a separate Pthread:

```
...
int listenfd = socket(...);
bind(listenfd,...);
listen(listenfd,....);

int connfd;
while (1) {
  //Accept new client connection:
  connfd = accept(listenfd,...);
  //Create a new thread to handle the new client:
  pthread_create(&threadid,NULL,handle_client,&connfd);
}
...
```

Race Condition Example (contd.)

Function run in each client thread:

```
void *handle_client (void *connfd_ptr)
{
  //Recover connfd, which was passed as pointer:
  int connfd = *(int*)connfd_ptr;

  ...use connfd to communicate with client...

  return NULL;
}
```

Race Condition Example (contd.)

The problem with this code is that `connfd`'s value can be changed before the new thread has retrieved it!

This may not be obvious, because the new Pthread will definitely be *created* before the server loops and runs `accept()` again.

However, even though the new Pthread may have been created before another `accept()`, it may not get run to the point of recovering `connfd`.

If a second client is `accept()`'ed *immediately* after `pthread_create()` returns, the value of `connfd` could get modified before the first client's thread recovers that client's FD.

Fixed Race Condition Example

Note use of *dynamic/heap memory* to fix race condition:

```
...
int listenfd = socket(...);
bind(listenfd,...);
listen(listenfd,....);

int connfd;
while (1) {
  connfd = accept(listenfd,...);
  int *connfd_ptr = malloc(sizeof(int));
  *connfd_ptr = connfd;
  pthread_create(&threadid,NULL,handle_client,connfd_ptr);
}
```

Fixed Race Condition Example (contd.)

Allocating fresh dynamic/heap memory to hold a copy of `connfd` fixes the problem, because each thread will have its own copy of `connfd`.

(`handle_client()` must `free` that memory when done with it.)

Passing a pointer to anything other than dynamic/heap memory in `pthread_create()` can be dangerous:

- *local variable* – value may be changed or (*stack*) memory deallocated before thread uses pointer
- *global/static variable* – value may be changed before thread uses pointer

Pthread Termination Methods

A Pthread's execution can be terminated in several ways:

- the `start_routine` function in `pthread_create()` calls `return`
- the thread calls `pthread_exit()`: `void pthread_exit(void *retval)`
- the thread is **cancelled** by another thread in the process calling `pthread_cancel()`
- any of the containing process' threads calls `exit()`
- the main/initial thread of the containing process calls `return` from its `main`

The first two methods are equivalent.

The first three methods terminate only a single thread.

The last two methods terminate the entire *process*—i.e., *all threads* in the process.

Returning Data from a Pthread

A thread is able to return data when it terminates, using a mechanism similar to that used to pass it data: a single `void*` value (a pointer to any oject type).

As noted above, this can be done by the thread's `start_routine` calling `return` or by the thread calling `pthread_exit()`.

This return pointer value can be obtained only by another thread that calls `pthread_join()`.

`pthread_join()` is the Pthreads equivalent of `wait()`.

(`pthread_join()` is discussed in a later lecture.)

Joinable/Detached Pthreads

Every Pthread will either be **joinable** or **detached**.

The joinable/detached status of a thread determines what happens to its return data (if any) when it terminates.

By default, a Pthread is *joinable* when created.

A *joinable* thread can have its return data obtained by another thread calling `pthread_join()`.

Only when a *terminated joinable thread* has been joined are all of its resources released.

This means that if a joinable thread terminates, and none of its peer threads "joins with it," we will have the Pthreads equivalent of a **zombie process**: resources will get wasted holding the return data.

Joinable/Detached Pthreads (contd.)

By contrast, a *detached* thread cannot be joined with.

As a side-effect of this, when a detached thread terminates, its *resources are automatically released.*

If the return data from a Pthread's `start_routine` is not going to be required, it is *advisable to set the thread to be detached*.

This can be done by having the thread call `pthread_detach()`:

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

It can also be done by passing an appropriate `attr` argument to `pthread_create()` when creating the thread.

For more info, see the man page for: `pthread_attr_setdetachstate()`

Thread IDs

Each process is identified by a unique ID, the **process ID** (**PID**).

Each Pthread is identified by an ID, the **Thread ID** (**TID**).

Before returning, `pthread_create()` stores the TID of the new thread in the `pthread_t` variable pointed to by its `thread` argument.

It is this TID that is used in other `pthread_*` functions.

A thread may obtain its own TID by calling `pthread_self()`:

```
pthread_t pthread_self(void)
```

While `pthread_t` TIDs are similar to PIDs in some ways, there are critical differences!

Thread IDs (contd.)

The `pthread_self()` man page says:

- POSIX.1 allows an implementation wide freedom in choosing the type used to represent a thread ID
- E.g., either an arithmetic type or a structure is permitted
- Therefore, variables of type `pthread_t` can't portably be compared using the C equality operator (`==`); use `pthread_equal()`
- Thread identifiers should be considered opaque: any attempt to use a thread ID other than in pthreads calls is nonportable and can lead to unspecified results.
- Thread IDs are guaranteed to be unique only within a process.
- A thread ID may be reused after a terminated thread has been joined, or a detached thread has terminated.
- The thread ID returned by `pthread_self()` is not the same thing as the **kernel thread ID** returned by a call to `gettid()`.

Thread IDs (contd.)

In Linux, each thread actually has *two different* thread IDs:

1. POSIX TID:
   - the TID returned from `pthread_create()` and `pthread_self()`
   - of type `pthread_t` (need not be scalar/numeric)
   - in Linux defined as: `typedef unsigned long int pthread_t`
   - not necessarily unique system-wide (may be so in Linux)
   - TID required for other `pthread_*` functions
2. kernel TID:
   - the TID returned from `gettid()`
   - of type `pid_t` (an integer—)
   - *unique* system-wide (among all threads)
   - since unique and in *same space as PIDs*, can be used in same way to identify a thread
   - *main thread* in a process has its TID equal to its PID

Thread IDs (contd.)

Note that `gettid()` is Linux-specific.

`gettid()` has no C wrapper, so must be called as:

```
syscall(SYS_gettid)
```

(Be sure to include `syscall.h` header file.)

On Linux, can determine if thread is *main* (initial) thread:

```
syscall(SYS_gettid) == getpid()
```

Pthread Attributes

The are a set of Pthread-specific *attributes* that can be set for each Pthread when it is created.

These attributes deal with joinable/detached state, CPU scheduling and affinity, and thread stack size and position.

Attributes are set at creation time, by passing an object of type `pthread_attr_t` to `pthread_create()`.

A `pthread_attr_t` object must be initialized by calling:

`pthread_attr_init()`

Attributes can then be set in the object using a variety of calls named `pthread_attr_*`.