POSIX threads

CS 241

February 17, 2012

Copyright © University of Illinois CS 241 Staff
Recall: Why threads over processes?

Creating a new process can be expensive

- **Time**
  - A call into the operating system is needed
  - Context-switching involves the operating system
- **Memory**
  - The entire process must be replicated
- **The cost of inter-process communication and synchronization of shared data**
  - May involve calls into the operation system kernel

Threads can be created without replicating an entire process

- Creating a thread is done in user space rather than kernel

Shared virtual address space
POSIX threads

Early on

• Each OS had it’s own thread library/API
• Difficult to write multithreaded programs
  ▪ Learn a new API with each new OS
  ▪ Modify code with each port to a new OS

So later...

• POSIX (IEEE 1003.1c-1995) provided a standard known as pthreads
The pthreads API

Thread management
- Creating, detaching, joining, etc. Set/query thread attributes

Mutexes
- Synchronization

Condition variables
- Communications between threads that share a mutex
Creating a Thread

```c
int pthread_create (pthread_t* tid,
                   pthread_attr_t* attr,
                   void* (start_routine),
                   void* arg);
```

- Unique thread identifier returned from call
- Attributes structure (NULL for defaults)
- Argument passed to child thread's `start_routine`
- Main routine for child thread

Zero for success, else error number
Creating a Thread

`pthread_create()` takes a pointer to a function as one of its arguments

- `start_routine` is called with the argument specified by `arg`
- `start_routine` can only have one parameter of type `void *`
- Complex parameters can be passed by creating a structure and passing the address of the structure
- The structure shouldn’t be a local variable
Example: pthread_create()

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

void *snow(void *data) {
    printf("Let it snow ... %s\n", data);
    pthread_exit(NULL);
}

int main(int argc, char *argv[]) {
    pthread_t mythread;
    int result;
    char *data = "Let it snow."
    result = pthread_create(&mythread, NULL, snow, data);
    printf("pthread_create() returned %d\n", result);
    if(result)
        exit (1);
    pthread_exit(NULL);
}
```
Thread vs. Process Creation

fork()

- Two separate processes with independent destinies
- Start from same position as parent (clone)
- Independent memory space for each process

pthread_create()

- Two separate threads with independent destinies
- Start from a function
- Share memory
fork()
pthread_create()
Possible output?

Shared code

```c
int x = 1;
void* func(void* p){
    x = x + 1;
    printf("x is %d\n", x);
    return NULL;
}
```

fork version

```c
main(...) {
    fork();
    func(NULL);
}
```

threads version

```c
main(...) {
    pthread_t tid;
    pthread_create(&tid,NULL,
                   func,NULL);
    func(NULL);
}
```
int x = 1;

void* func(void* p){
    x = x + 1;
    printf("x is %d\n", x);
    return NULL;
}

Output:

x is 2
x is 3
int x = 1;

void* func(void* p) {
    x = x + 1;
    printf("x is %d\n", x);
    return NULL;
}

Output:

x is 3
x is 2
Possible output: threads version, #3

int x = 1;

void* func(void* p){
    x = x + 1;
    printf("x is %d\n", x);
    return NULL;
}

Output:
  x is 3
  x is 3
int x = 1;

void* func(void* p){
    x + 1
    x = ___;
    printf("x is %d\n", x);
    return NULL;
}

Output:
  x is 2
  x is 2
Summary: Creating Threads

Initially, `main()` has a single thread
- All other threads must be explicitly created

`pthread_create()` ➔ new executable thread
- Can be called any number of times from anywhere

Maximum number of threads is implementation dependent

Question:
- After a thread has been created, how do you know when it will be scheduled to run by the operating system?
- Answer: It is up to the operating system
- Correct coding should not require knowledge of scheduling
  - Later: How to accomplish that
pthreads Attributes

Attributes

- Data structure `pthread_attr_t`
- Set of choices for a thread
- Passed in thread creation routine

Choices

- Scheduling options (more later on scheduling)
- Detached state
  - Detached
    - Main thread does not wait for the child threads to terminate
  - Joinable
    - Main thread waits for the child thread to terminate
    - Useful if child thread returns a value
pthreads Attributes

Initialize an attributes structure to the default values

• \texttt{int pthread_attr_init (pthread_attr_t* attr);} \\

Set the detached state value in an attributes structure

• \texttt{int pthread_attr_setdetachstate (pthread_attr_t* attr, int value);} \\
  • value is one of
    • PTHREAD_CREATE_DETACHED ("zombie antidote")
    • PTHREAD_CREATE_JOINABLE

Can change your mind later

• joinable to detached via \texttt{pthread_detach ()}
• but, nothing to go from detached to joinable
Detached Threads

Master Thread

Worker Thread

Worker Thread

Worker Thread

...
Joined Threads
Waiting for Threads: `pthread_join()`

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

Suspends calling thread until target thread terminates

Returns
- 0 on success
- Error code on failure

Parameters
- **thread**: Target thread identifier
- **retval**: Value passed to `pthread_exit()` by the terminating thread is made available in the location referenced by `retval`
Waiting for Threads: pthread_join()

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

Note

- You cannot call `pthread_join()` on a detached thread
- Detaching means you are **not** interested in knowing about the thread’s exit and return value

Set `pthread_attr` to joinable before creating thread

- `pthread_attr_init(&attr);`
- `pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_JOINABLE);`
Returning data via pthread_join()

```c
void *thread(void *vargp) {
    pthread_exit((void *)42);
}

int main() {
    int i;
    pthread_t tid;

    pthread_create(&tid, NULL, thread, NULL);
    pthread_join(tid, (void **)&i);
    printf("%d\n", i);
}
```
Returning data via pthread_join()

```c
void *thread(void *vargp) {
    pthread_exit((void *)42);
}

int main() {
    int i;
    pthread_t tid;

    /* Initialize and set thread detached attribute */
    pthread_attr_t attr;
    pthread_attr_init(&attr);
    pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_JOINABLE);

    pthread_create(&tid, &attr, thread, NULL);
    pthread_join(tid, (void **)&i);
    printf("%d\n", i);
}
```

Terminating Threads: pthread_exit()

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

Terminate the calling thread

Makes the value `retval` available to any successful join with the terminating thread

Returns

- `pthread_exit()` cannot return to its caller

Parameters

- `retval`: Pointer to data returned to joining thread

Note

- If `main()` exits before its threads via `pthread_exit()`, the other threads continue. Otherwise, they will be terminated when `main()` ends.
Termination example

```c
#include <pthread.h>
#define NUM_THREADS 5

void *PrintHello(void *threadid) {
    printf("%d: Hello World!\n", threadid);
    pthread_exit(NULL);
}
```
int main (int argc, char *argv[]) {
    pthread_t threads[NUM_THREADS];
    int rc, t;

    for(t=0; t < NUM_THREADS; t++) {
        printf("Creating thread %d\n", t);
        rc = pthread_create(&threads[t], NULL, f, (void *)t);
        if (rc) {
            printf("ERROR; pthread_create() return code is %d\n", rc);
            exit(-1);
        }
    }
}

pthread_exit(NULL);
Termination example

```c
int main (int argc, char *argv[]) {
    pthread_t threads[NUM_THREADS];
    int rc, t;

    for(t=0;t < NUM_THREADS;t++) {
        printf("Creating thread %d\n", t);
        rc = pthread_create(&threads[t], NULL, f, (void *)t);
        if (rc) {
            printf("ERROR; pthread_create() return code is %d\n", rc);
            exit(-1);
        }
    }
    pthread_exit(NULL); for(t=0;t < NUM_THREADS;t++) {
        pthread_join(thread[t], NULL);
        printf("Joined thread %d\n",t);
    }
}
```

Will all threads get a chance to execute before the parent exits?
pthreads functions do not follow the usual Unix conventions

- **Similarity**
  - Returns 0 on success

- **Differences**
  - Returns error code on failure
  - Does not set errno

- **What about errno?**
  - Each thread has its own
  - Define _REENTRANT (-D_REENTRANT switch to compiler) when using pthreads
Thread Lifetime

A thread exists until...

- It returns from the function or calls pthread_exit()
- The whole process terminates
- The machine catches fire
So, your process terminates when...

Any thread calls exit();

The main thread returns
  • main() {
      pthread_create();
      return 0;
  }

Segmentation fault
  • *(char*)0 = 0;

There are no more threads left to run
Main points

A thread is the lightest unit of work that can be scheduled to run on the processor

To create a thread you
- Indicate which function the thread should execute
- Indicate the detach state of the thread

When a new thread is created
- It runs concurrently with the creating thread
- It shares common data space
Reference slides
## Threads vs. Processes

<table>
<thead>
<tr>
<th>Property</th>
<th>Processes created with fork</th>
<th>Threads of a process</th>
<th>Ordinary function calls</th>
</tr>
</thead>
<tbody>
<tr>
<td>variables</td>
<td>Get copies of all variables</td>
<td>Share global variables</td>
<td>Share global variables</td>
</tr>
<tr>
<td>IDs</td>
<td>Get new process IDs</td>
<td>Share the same process ID but have unique thread ID</td>
<td>Share the same process ID (and thread ID)</td>
</tr>
<tr>
<td>Data/control</td>
<td>Must communicate explicitly, e.g., use pipes or small integer return value</td>
<td>May communicate with return value or carefully shared variables</td>
<td>May communicate with return value or shared variables</td>
</tr>
<tr>
<td>Parallelism (one CPU)</td>
<td>Concurrent</td>
<td>Concurrent</td>
<td>Sequential</td>
</tr>
<tr>
<td>Parallelism (multiple CPUs)</td>
<td>May be executed simultaneously</td>
<td>Kernel threads may be executed simultaneously</td>
<td>Sequential</td>
</tr>
</tbody>
</table>
Getting the current thread ID

You can retrieve the current thread ID

- `pthread_t pthread_self(void);`
- Returns currently executing thread’s ID