15-213
“The course that gives CMU its Zip!”

Concurrency I: Threads
Nov 9, 2000

Topics
• Thread concept
• Posix threads (Pthreads) interface
• Linux Pthreads implementation
• Concurrent execution
• Sharing data
Traditional view of a process

Process = process context + code, data, and stack

**Process context**

- **Program context:**
  - Data registers
  - Condition codes
  - Stack pointer (SP)
  - Program counter (PC)

- **Kernel context:**
  - VM structures
  - Open files
  - Signal handlers
  - brk pointer

**Code, data, and stack**

- Stack
- Shared libraries
- Run-time heap
- Read/write data
- Read-only code/data

0
Modern view of a process

Process = thread + code, data, and kernel context

Thread (main thread)

- Stack

Thread context:
- Data registers
- Condition codes
- Stack pointer (SP)
- Program counter (PC)

Code and Data

- Shared libraries
- Run-time heap
- Read/write data
- Read-only code/data

Kernel context:
- VM structures
- Open files
- Signal handlers
- Brk pointer
A process with multiple threads

Multiple threads can be associated with a process

- Each thread has its own logical control flow (sequence of PC values)
- Each thread shares the same code, data, and kernel context
- Each thread has its own thread id (tid)

Thread 1 (main thread)
- Data registers
- Condition codes
- SP1
- PC1
- stack 1

Thread 2 (peer thread)
- Data registers
- Condition codes
- SP2
- PC2
- stack 2

Thread 1 context:

Thread 2 context:

Shared code and data

- shared libraries
- run-time heap
- read/write data
- read-only code/data

Kernel context:

- VM structures
- Open files
- Signal handlers
- brk pointer
Logical view of threads

Threads associated with a process form a pool of peers.

- Unlike processes which form a tree hierarchy
Concurrent thread execution

Two threads run concurrently (are concurrent) if their logical flows overlap in time. Otherwise, they are sequential.

Examples:
- Concurrent: A & B, A&C
- Sequential: B & C
Threads vs processes

How threads and processes are similar

• Each has its own logical control flow.
• Each can run concurrently.
• Each is context switched.

How threads and processes are different

• Threads share code and data, processes (typically) do not.
• Threads are somewhat less expensive than processes.
  – process control (creating and reaping) is twice as expensive as thread control.
  – Linux/Pentium III numbers:
    » 20K cycles to create and reap a process.
    » 10K cycles to create and reap a thread.
Threads are a unifying abstraction for exceptional control flow

**Exception handler**
- A handler can be viewed as a thread
- Waits for a "signal" from CPU
- Upon receipt, executes some code, then waits for next "signal"

**Process**
- A process is a thread + shared code, data, and kernel context.

**Signal handler**
- A signal handler can be viewed as a thread
- Waits for a signal from the kernel or another process
- Upon receipt, executes some code, then waits for next signal.
Posix threads (Pthreads) interface

Pthreads: Standard interface for ~60 functions that manipulate threads from C programs.

- Creating and reaping threads.
  - pthread_create
  - pthread_join
- Determining your thread ID
  - pthread_self
- Terminating threads
  - pthread_cancel
  - pthread_exit
  - exit() [terminates all threads], ret [terminates current thread]
- Synchronizing access to shared variables
  - pthread_mutex_init
  - pthread_mutex_[un]lock
  - pthread_cond_init
  - pthread_cond_[timed]wait
The Pthreads "hello, world" program

/*
 * hello.c - Pthreads "hello, world" program
 */
#include <ics.h>

void *thread(void *vargp);

int main() {
    pthread_t tid;

    Pthread_create(&tid, NULL, thread, NULL);
    Pthread_join(tid, NULL);
    exit(0);
}

/* thread routine */
void *thread(void *vargp) {
    printf("Hello, world!\n");
    return NULL;
}
Execution of “hello, world”

- create peer thread
- wait for peer thread to terminate
- print output
- terminate thread via `ret`
- `exit()` terminates main thread and any peer threads
Unix vs Posix error handling

Unix-style error handling (Unix syscalls)

- if error: return -1 and set errno variable to error code.
- if OK: return useful result as value $\geq 0$.

```c
if ((pid = wait(NULL)) < 0) {
    perror("wait");
    exit(0);
}
```

Posix-style error handling (newer Posix functions)

- if error: return nonzero error code, zero if OK
- useful results are passed back in an argument.

```c
if ((rc = pthread_join(tid, &retvalp)) != 0) {
    printf("pthread_join: %s\n", strerror(rc));
    exit(0);
}
```
Error checking crucial, but cluttered. Use these to simplify your error checking:

```c
/*
 * macro for unix-style error handling
 */
#define unix_error(msg) do {
    printf("%s: %s\n", msg, strerror(errno));
    exit(0);
} while (0)

عسكر

/*
 * macro for posix-style error handling
 */
#define posix_error(code, msg) do {
    printf("%s: %s\n", msg, strerror(code));
    exit(0);
} while (0)
```
Pthreads wrappers

We advocate Steven’s convention of providing wrappers for each system-level function call.

• wrapper is denoted by capitalizing first letter of function name.
• wrapper has identical interface as the original function.
• each wrapper does appropriate unix or posix style error checking.
• wrapper typically return nothing.
• declutters code without compromising safety.

/*
 * wrapper function for pthread_join
 */
void Pthread_join(pthread_t tid, void **thread_return) {
    int rc = pthread_join(tid, thread_return);
    if (rc != 0)
        posix_error(rc, "Pthread_join");
}
Basic thread control: create a thread

```c
int pthread_create(pthread_t *tidp, pthread_attr_t *attrp,
                   void **(routine)(void *), void *argp);
```

Creates a new peer thread

- `tidp`: thread id
- `attrp`: thread attributes (usually NULL)
- `routine`: thread routine
- `argp`: input parameters to routine

Akin to `fork()`

- but without the confusing “call once return twice” semantics.
- peer thread has local stack variables, but shares all global variables.
Basic thread control: join

```c
int pthread_join(pthread_t tid, void **thread_return);
```

Waits for a specific peer thread to terminate, and then reaps it.
- **tid**: thread ID of thread to wait for.
- **thread_return**: object returned by peer thread via `ret` stmt

Akin to `wait` and `wait_pid` but unlike `wait` ...
- Any thread can reap any other thread (not just children)
- Must wait for a *specific* thread
  - no way to wait for *any* thread.
  - perceived by some as a flaw in the Pthreads design
Linux implementation of Pthreads

Linux implements threads in an elegant way:

- Threads are just processes that share the same kernel context.
- `fork()`: creates a child process with a new kernel context
- `clone()`: creates a child process that shares some or all of the parent’s kernel context.

```c
int __clone(int (*fn)(void *arg), void *child_stack,
            int flags, void *arg);
```

Creates a new process and executes function `fn` with argument `arg` in that process using the stack space pointed to by `child_stack`. Returns `pid` of new process.

`flags` determine the degree of kernel context sharing: e.g.,
- `CLONE_VM`: share virtual address space
- `CLONE_FS`: share file system information
- `CLONE_FILES`: share open file descriptors
hellopid.c

The following routine will show us the process hierarchy of a Linux thread pool:

```c
#include <ics.h>
void *thread(void *vargp);

int main() {
    pthread_t tid;
    printf("Hello from main thread! tid:%ld pid:%d\n",
           pthread_self(), getpid());
    pthread_create(&tid, NULL, thread, NULL);
    pthread_join(tid, NULL);
    exit(0);
}

void *thread(void *vargp) {
    printf("Hello from child thread! tid:%ld pid:%d ppid:%d\n",
            pthread_self(), getpid(), getppid());
    return NULL;
}
```
Linux process hierarchy for threads

```
bass> hellopid
Hello from main thread!  tid:1024 pid:6024
Hello from child thread! tid:1025 pid:6026 ppid:6025
```

Thread manager supports thread abstraction using signals:

- `exit()`: kills all threads, regardless where it is called from
- slow system calls such as `sleep()` or `read()` block only the calling thread.
beep.c: Performing concurrent tasks

```c
/*
 * beeps until the user hits a key
 */
#include <ics.h>
void *thread(void *vargp);

/* shared by both threads */
char shared = '\0';

int main() {
    pthread_t tid;
    Pthread_create(&tid, NULL,
                      thread, NULL);
    while (shared == '\0') {
        printf("BEEP\n");
        sleep(1);
    }
    Pthread_join(tid, NULL);
    printf("DONE\n");
    exit(0);
}

/* thread routine */
void *thread(void *vargp) {
    shared = getchar();
    return NULL;
}
```
badcnt.c: Sharing data between threads

```c
/* bad sharing */
#include <ics.h>
#define NITERS 1000
void *count(void *arg);

struct {
    int counter;
} shared;

int main() {
    pthread_t tid1, tid2;
    Pthread_create(&tid1, NULL,
        count, NULL);
    Pthread_create(&tid2, NULL,
        count, NULL);
    if (shared.counter != NITERS*2)
        printf("BOOM! counter=%d\n", shared.counter);
    else
        printf("OK counter=%d\n", shared.counter);
}

/* thread routine */
void *count(void *arg) {
    int i, val;

    for (i=0; i<NITERS; i++) {
        val = shared.counter;
        printf("%d: %d\n",
            (int)pthread_self(), val);
        shared.counter = val + 1;
    }
    return NULL;
}
```

Key point:
“struct shared” is visible to all threads.
“i” and “val” are visible only to the count thread.
### Running badcnt.c

<table>
<thead>
<tr>
<th>Output of run 1</th>
<th>Output of run 2</th>
<th>Output of run 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>1025: 0</td>
<td>1025: 0</td>
<td>1025: 0</td>
</tr>
<tr>
<td>1025: 1</td>
<td>1025: 1</td>
<td>1025: 1</td>
</tr>
<tr>
<td>1025: 2</td>
<td>1025: 2</td>
<td>1025: 2</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>1025: 997</td>
<td>1025: 997</td>
<td>1025: 997</td>
</tr>
<tr>
<td>1025: 998</td>
<td>1025: 998</td>
<td>1025: 998</td>
</tr>
<tr>
<td>1025: 999</td>
<td>1025: 999</td>
<td>1025: 999</td>
</tr>
<tr>
<td>2050: 969</td>
<td>2050: 712</td>
<td>2050: 1000</td>
</tr>
<tr>
<td>2050: 970</td>
<td>2050: 713</td>
<td>2050: 1001</td>
</tr>
<tr>
<td>2050: 971</td>
<td>2050: 714</td>
<td>2050: 1002</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>2050: 1968</td>
<td>2050: 1711</td>
<td>2050: 1999</td>
</tr>
<tr>
<td><strong>BOOM! counter=1969</strong></td>
<td><strong>BOOM! counter=1712</strong></td>
<td><strong>OK counter=2000</strong></td>
</tr>
</tbody>
</table>

**So what’s the deal?**

We must *synchronize* concurrent accesses to shared thread data
(the topic of our next lecture)