CS450/550
Operating Systems

Lecture 2A  POSIX Threads Programming

Dr. Xiaobo Zhou
Department of Computer Science

POSIX Thread Programming Tutorial

A.1 Pthreads Overview
A.2 Pthreads API
A.3 Compiling and Debugging
A.4 Thread Management
A.5 Mutex Variables
A.6 Condition Variable
A.7 References
Review: The Thread Model

- Process: for resource grouping and execution
- Thread: a finer-granularity entity for execution and parallelism
  - Lightweight processes, multi-threading

(a) Three processes each with one thread, but **different address spaces**
(b) One process with three threads, **sharing the address space**

The Thread Model (2)

- Because threads within the same process share resources
  - Changes made by one thread to shared system resources (closing a file) will be seen by all other threads
  - Two pointers having the same value point to the same data
  - Reading and writing to the same memory location is possible, and therefore requires explicit synchronization by the programmer!

<table>
<thead>
<tr>
<th>Per process items</th>
<th>Per thread items</th>
</tr>
</thead>
<tbody>
<tr>
<td>Address space</td>
<td>Program counter</td>
</tr>
<tr>
<td>Global variables</td>
<td>Registers</td>
</tr>
<tr>
<td>Open files</td>
<td>Stack</td>
</tr>
<tr>
<td>Child processes</td>
<td>State</td>
</tr>
<tr>
<td>Pending alarms</td>
<td>Scheduling properties</td>
</tr>
<tr>
<td>Signals and signal handlers</td>
<td></td>
</tr>
<tr>
<td>Accounting information</td>
<td></td>
</tr>
</tbody>
</table>

Items shared by all threads in a process | Items private to each thread
Pthreads Overview

- What are Pthreads?
  - An IEEE standardized thread programming interface
  - POSIX threads
  - Defined as a set of C language programming types and procedure calls, implemented with a pthread.h header/include file and a thread library

Why Pthreads

- Performance!
  - Lightweight
  - Overlapping CPU work with I/O
  - Priority/real-time scheduling
  - Asynchronous event handling

<table>
<thead>
<tr>
<th>Platform</th>
<th>fork()</th>
<th>pthread_create()</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>real</td>
<td>user</td>
</tr>
<tr>
<td>IBM 3.75 MHz POWER3</td>
<td>61.94</td>
<td>3.49</td>
</tr>
<tr>
<td>IBM 1.5 GHz POWER4</td>
<td>44.08</td>
<td>2.21</td>
</tr>
<tr>
<td>IBM 1.9 GHz POWER5 p5-575</td>
<td>50.66</td>
<td>3.32</td>
</tr>
<tr>
<td>INTEL 2.4 GHz Xeon</td>
<td>23.81</td>
<td>3.12</td>
</tr>
<tr>
<td>INTEL 1.4 GHz Itanium 2</td>
<td>23.61</td>
<td>0.12</td>
</tr>
</tbody>
</table>

Time results on SMP systems
### Thread Usage – A Multi-threaded Web Server

- A dispatcher thread and worker threads
  - Shared Web page cache
  - Asynchronous event handling: working on previous requests and managing the arrival of new requests

![Diagram of a multi-threaded Web server](image)

### Design Threaded Programs

- A program must be able to be organized into discrete, independent tasks which can execute concurrently
  - E.g.: r1 and r2 can be interchanged, interleaved, and/or overlapped in real time
  - Thread-safeness: race conditions
The Pthreads API

- The API is defined in the ANSI/IEEE POSIX 1003.1 – 1995
  - Naming conventions: all identifiers in the library begins with `pthread_`
  - Three major classes of subroutines
    - Thread management, mutexes, condition variables

<table>
<thead>
<tr>
<th>Routine Prefix</th>
<th>Functional Group</th>
</tr>
</thead>
<tbody>
<tr>
<td>pthread_</td>
<td>Threads themselves and miscellaneous subroutines</td>
</tr>
<tr>
<td>pthread_attr_</td>
<td>Thread attributes objects</td>
</tr>
<tr>
<td>pthread_mutex_</td>
<td>Mutexes</td>
</tr>
<tr>
<td>pthread_mutexattr_</td>
<td>Mutex attributes objects.</td>
</tr>
<tr>
<td>pthread_cond_</td>
<td>Condition variables</td>
</tr>
<tr>
<td>pthread_condattr_</td>
<td>Condition attributes objects</td>
</tr>
<tr>
<td>pthread_key_</td>
<td>Thread-specific data keys</td>
</tr>
</tbody>
</table>

Compiling

<table>
<thead>
<tr>
<th>Platform</th>
<th>Compiler Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>IBM AIX</td>
<td>xlC_r / cc_r</td>
<td>C (ANSI / non-ANSI)</td>
</tr>
<tr>
<td></td>
<td>xlC_r</td>
<td>C++</td>
</tr>
<tr>
<td></td>
<td>xlf_r-qnosave</td>
<td>Fortran - using IBM's Pthreads API (non-portable)</td>
</tr>
<tr>
<td></td>
<td>xlf90_r-qnosave</td>
<td></td>
</tr>
<tr>
<td>INTEL LINUX</td>
<td>icc -pthread</td>
<td>C</td>
</tr>
<tr>
<td></td>
<td>icpc -pthread</td>
<td>C++</td>
</tr>
<tr>
<td>COMPAQ Tru64</td>
<td>cc -pthread</td>
<td>C</td>
</tr>
<tr>
<td></td>
<td>cox -pthread</td>
<td>C++</td>
</tr>
<tr>
<td>All Above Platforms</td>
<td>gcc -pthread</td>
<td>GNU C</td>
</tr>
<tr>
<td></td>
<td>g++ -pthread</td>
<td>GNU C++</td>
</tr>
<tr>
<td></td>
<td>guidec -pthread</td>
<td>KAI C (if installed)</td>
</tr>
<tr>
<td></td>
<td>KCC -pthread</td>
<td>KAI C++ (if installed)</td>
</tr>
</tbody>
</table>
Debugging

- **GDB:**
  - GDB: stopping and starting multi-thread programs
  - GDB/MI: Threads commands

- **DDD:**
  - Examining threads

Implementing Threads in User Space

- User-space threads: the kernel knows nothing about them
  - No clock interrupts, no preemption *(scheduling is programmer’s job)*

A user-level threads package
Thread Management – Creation and Termination

### pthread_create (threadid, attr, start_routine, arg)
* creates a thread and makes it executable; arg must be passed by reference as a pointer cast of type void

### pthread_exit (status)
- If main() finishes before the threads it has created, and exists with the pthread-exit(), the other threads will continue to execute. Otherwise, they will be automatically terminated when main() finishes

### pthread_attr_init (attr)
- Initialize the thread attribute object (other routines can then query/set attributes)

### pthread_attr_destroy (attr)
- destroy the thread attribute object

Initially, your main() program comprises a single, default thread.

---

Example: Pthread Creation and Termination

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

don't understand the code
```
Pthread Argument Passing – single parameter

Example 1 - Thread Argument Passing

This code fragment demonstrates how to pass a simple integer to each thread. The calling thread uses a unique data structure for each thread, ensuring that each thread's argument remains intact throughout the program.

```c
int *taskids[NUM_THREADS];

for(t=0; t<NUM_THREADS; t++)
{
    taskids[t] = (int *) malloc(sizeof(int));
    taskids[t] = t;
    printf('Creating thread %d\n', t);
    rc = pthread_create(&threads[t], NULL, PrintHello,
                       (void *) taskids[t]);
    ...
}
```

Pthread Argument Passing – multiple parameters

Example 2 - Thread Argument Passing

This example shows how to setup pass multiple arguments via a structure. Each thread receives a unique instance of the structure.

```c
struct thread_data{
    int thread_id;
    int sum;
    char *message;
};

struct thread_data thread_data_array[NUM_THREADS];

void *PrintHello(void *threadarg)
{
    struct thread_data *my_data;
    ...
    my_data = (struct thread_data *) threadarg;
    taskid = my_data->thread_id;
    sum = my_data->sum;
    hello_msg = my_data->message;
    ...
}

int main (int argc, char *argv[])
{
    ...
    thread_data_array[t].thread_id = t;
    thread_data_array[t].sum = sum;
    thread_data_array[t].message = messages[t];
    rc = pthread_create(&threads[t], NULL, PrintHello,
                        (void *) & thread_data_array[t]);
    ...
}
**Thread Management – Joining and Detaching**

- `pthread_join` (threadid, status)
- `pthread_detach` (threadid, status)
- `pthread_attr_setdetachstate` (attr, detachstate)
- `pthread_attr_getdetachstate` (attr, detachstate)

Joining is one way to accomplish synchronization between threads: The `pthread_join()` subroutine blocks the calling thread until the specified threadid thread terminates.

---

**Example 3 - Thread Argument Passing (Incorrect)**

This example performs argument passing incorrectly. The loop which creates threads modifies the contents of the address passed as an argument, possibly before the created threads can access it.

```c
int rc, t;

for(t=0; t<NUM_THREADS; t++)
{
    printf("Creating thread %d\n", t);
    rc = pthread_create(&threads[t], NULL, PrintHello,
        (void *) &t);
    ... 
}
```

---

**Joining or not?**

Master Thread

Worker Thread

Worker Thread

DO WORK

Join or not?
Thread Management – Joining and Detaching

```c
int main (int argc, char *argv[]) {
    pthread_t thread[1000];
    pthread_attr_t attr;
    int rc, t, status;

    /* Initialize and set thread detached attribute */
    pthread_attr_init(&attr);
    pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_DETACHED);
    for (t=0; t<NUM_THREADS; t++) {
        printf("Creating thread \n");
        rc = pthread_create(&thread[t], &attr, DelayWork, NULL);
        if (rc)
            printf("ERROR: return code from pthread_create() is \n", rc);
        exit(1);
    }

    /* Free attribute and wait for the other threads */
    for (t=0; t<NUM_THREADS; t++) {
        rc = pthread_join(thread[t], (void *)&status);
        if (rc)
            printf("ERROR: return code from pthread_join() is \n", rc);
        exit(1);
    }
    pthread_exit(NULL);
}
```

Thread Management – Stack Management

- `pthread_attr_getstacksize` (attr, stacksize)
- `pthread_attr_setstacksize` (attr, stacksize)
- `pthread_attr_getstackaddr` (attr, stackaddr)
- `pthread_attr_setstackaddr` (attr, stackaddr)

The POSIX standard does not indicate the size of a thread’s stack, which is system implementation dependent.

Exceeding the default stack limit: program termination and/or corrupted data

Safe and portable programs should explicitly allocate enough stack for each thread; if the stack must be placed in some particular region of memory, use the last two routines.
Thread Management – Miscellaneous Routines

- `pthread_self`()
- `pthread_equal` (thread1, thread2)
- `pthread_yield`()

The thread identifier objects are opaque, the C equivalence operator `==` should not be used to compare two thread IDs against each other, or to compare a single thread ID against another value.

---

IPC: Race Conditions

<table>
<thead>
<tr>
<th>Thread 1</th>
<th>Thread 2</th>
<th>Balance</th>
</tr>
</thead>
<tbody>
<tr>
<td>Read balance; $1000</td>
<td>Read balance; $1000</td>
<td>$1000</td>
</tr>
<tr>
<td>Deposit $200</td>
<td>Deposit $200</td>
<td>$1000</td>
</tr>
<tr>
<td>Update balance $1000 + $200</td>
<td>Update balance $1000 + $200</td>
<td>$1200</td>
</tr>
</tbody>
</table>

Two threads want to deposit an account; overwriting issue.
Mutual Exclusion and Critical Regions

- Mutual exclusion: makes sure if one process is using a shared variable or file, the other processes will be excluded from doing the same thing
  - Main challenge/issue to OS: to design appropriate primitive operations for achieving mutual exclusion
- Critical regions: the part of the program where the shared memory is accessed

**Mutexes**

- Mutex: a simplified version of the semaphores
  - a variable that can be in one of two states: unlocked or locked
  - Supports synchronization by controlling access to shared data
- A typical sequence in the use of a mutex
  - Create and initialize a mutex variable
  - Several threads attempt to lock the mutex
  - Only one succeeds and that thread owns the mutex
  - The owner thread performs some set of actions
  - The owner unlocks the mutex
  - Another thread acquires the mutex and repeats the process
  - Finally the mutex is destroyed
- Remember the priority inversion problem?
  - An unblocking call with “trylock”, instead of blocking “lock” call
  - It is programmer’s responsibility to make sure every thread that needs to use a mutex to protect shared data does so
### Mutex Management – Creating and Destroying Mutexes

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>pthread_mutex_init(mutex, attr)</code></td>
<td>Initializes a mutex variable with a mutex attribute.</td>
</tr>
<tr>
<td><code>pthread_mutex_destroy(mutex)</code></td>
<td>Destroys a mutex object.</td>
</tr>
<tr>
<td><code>pthread_mutexattr_init(attr)</code></td>
<td>Initializes a mutex attribute.</td>
</tr>
<tr>
<td><code>pthread_mutexattr_destroy(attr)</code></td>
<td>destroys a mutex attribute.</td>
</tr>
</tbody>
</table>

1. Mutex variables must be declared with type `pthread_mutex_t`, and must be initialized before can be used. The mutex is initially not locked.

   ```c
   Statically: pthread_mutex_t mymmutex = PTHREAD_MUTEX_INITIALIZER
   Dynamically, with pthread_mutex_init(mutex, attr)
   ```

2. The `attr` object must be declared with type `pthread_mutexattr_t`

3. Programmers should free a mutex object that is no longer used

### Mutex Management – Locking and Unlocking Mutexes

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>pthread_mutex_lock(mutex)</code></td>
<td>Acquires a lock on the mutex.</td>
</tr>
<tr>
<td><code>pthread_mutex_trylock(mutex)</code></td>
<td>Attempts to acquire a lock on the mutex. If the lock is already held,</td>
</tr>
<tr>
<td></td>
<td>-the call returns immediately;</td>
</tr>
<tr>
<td></td>
<td>-the call is blocked until the lock can be acquired.</td>
</tr>
<tr>
<td><code>pthread_mutexattr_unlock(mutex)</code></td>
<td>Releases the lock on the mutex.</td>
</tr>
</tbody>
</table>

1. `pthread_mutex_lock(mutex)` is a blocking call.
2. `pthread_mutex_trylock(mutex)` is a non-blocking call, useful in preventing the deadlock conditions (priority-inversion problem)
3. “gentlemen’s agreement” between participating threads
4. If you use multiple mutexes, the order is important; see textbook P115
5. Example code; see the reference link

When more than one thread is waiting for a locked mutex, which thread will be granted the lock first after it is released?
Monitors and Condition Variables

Monitor: a higher-level synchronization primitive
- Only one process can be active in a monitor at any instant, with compiler’s help; thus, how about to put all the critical regions into monitor procedures for mutual exclusion?

```
monitor example
  integer i;
  condition c;
  procedure producer();
    .
    .
    .
  end;

  procedure consumer();
    .
    .
    .
  end;
end monitor:
```

But, how processes block when they cannot proceed?

Condition variables, and two operations: `wait()` and `signal()`

Condition Variables and Mutexes

- Mutexes: support synchronization by controlling thread access to data
- Condition variables: another way for threads to synchronize
  - Allows thread to synchronize based on the actual value of data
  - Always used in conjunction with a mutex lock, why?
    - In Monitors, mutual exclusion is achieved with compiler’s help which ensures at any time only one process can be active in a monitor
    - `wait()` and `signal()`
A Representative Sequence Using Condition Variables

<table>
<thead>
<tr>
<th>Main Thread</th>
<th>Thread A</th>
<th>Thread B</th>
</tr>
</thead>
<tbody>
<tr>
<td>Declare and initialize global data/variables which require synchronization (such as &quot;count&quot;)</td>
<td>Do work up to the point where a certain condition must occur (such as &quot;count&quot; must reach a specified value)</td>
<td>Do work</td>
</tr>
<tr>
<td>Declare and initialize a condition variable object;</td>
<td>Lock associated mutex and check value of a global variable</td>
<td>Lock associated mutex</td>
</tr>
<tr>
<td>Declare and initialize an associated mutex</td>
<td>Call \texttt{pthread_cond_wait()} to perform a blocking wait for signal from Thread-B. Note that a call to \texttt{pthread_cond_wait()} automatically and atomically unlocks the associated mutex variable so that it can be used by Thread-B.</td>
<td>Change the value of the global variable that Thread-A is waiting upon.</td>
</tr>
<tr>
<td>Create threads A and B to do work</td>
<td>4. When signalled, wake up. Mutex is automatically and atomically locked.</td>
<td>4. Check value of the global Thread-A wait variable. If it fulfills the desired condition, signal Thread-A</td>
</tr>
<tr>
<td></td>
<td>5. Explicitly unlock mutex</td>
<td>5. Unlock mutex.</td>
</tr>
<tr>
<td></td>
<td>6. Continue</td>
<td>6. Continue</td>
</tr>
<tr>
<td>Main Thread</td>
<td>Join / Continue</td>
<td></td>
</tr>
</tbody>
</table>

Condition Variables – Creating and Destroying Con. Variables

- \texttt{pthread\_cond\_init} (condition, attr)
- \texttt{pthread\_cond\_destroy} (condition)
- \texttt{pthread\_condattr\_init} (attr)
- \texttt{pthread\_condattr\_destroy} (attr)

1. Mutex variables must be declared with type \texttt{pthread\_cond\_t}, and must be initialized before can be used. The mutex is initially not locked.
   
   Statically: \texttt{pthread\_cond\_t myconvar = PTHREAD\_COND\_INITIALIZER}
   
   Dynamically, with \texttt{pthread\_cond\_init} (condition, attr)

2. The attr object must be declared with type \texttt{pthread\_condattr\_t}

3. Programmers should free a condition variable that is no longer used
**Condition Variables – Waiting and Signaling Con. Variables**

- `pthread_cond_wait` (condition, mutex)
- `pthread_cond_signal` (condition)
- `pthread_cond_broadcast` (condition)

1. `waiting()` blocks the calling thread until the specified condition is signaled. It should be called while mutex is locked.
2. `signal()` is used to signal (or wake up) another thread which is waiting on the condition variable. It should be called after mutex is locked, and programmers must unlock mutex in order for `wait()` routine to complete.
3. `broadcast()` is useful when multiple threads are in blocked waiting.
4. `wait()` should come before `signal()` – conditions are not counters, signal would be lost if not waiting.
5. Example code; see the reference link.

**Re: Semaphores and P&V Operations**

- Semaphores: a variable to indicate the # of pending wakeups
- **Down** operation (P):
  - Checks if a semaphore is > 0,
  - if so, it decrements the value and just continue
  - Otherwise, the process is put to sleep without completing the down for the moment
- **Up** operation (V)
  - Increments the value of the semaphore
  - if one or more processes are sleeping on the semaphone, one of them is chosen by the system (by random) and allowed to complete its down
- P & V operations are atomic
POSIX Semaphores

- Semaphores are counters for resources shared between threads. The basic operations on semaphores are: increment the counter atomically, and wait until the counter is non-null and decrement it atomically.
- The pthreads library implements POSIX 1003.1b semaphores. These should not be confused with System V semaphores (ipc, semctl and semop).
- All the semaphore functions & macros are defined in semaphore.h.
  - `int sem_init(sem_t *sem, int pshared, unsigned int value)`
  - `int sem_destroy(sem_t *sem)`
  - `int sem_wait(sem_t *sem)`
  - `int sem_trywait(sem_t *sem)`
  - `int sem_post(sem_t *sem)`
  - `int sem_getvalue(sem_t *sem, int *sval)`


Thread Scheduling and Thread-specific Data

- Thread scheduling
  - implementation differs. In most cases, the default mechanism is adequate
  - Pthreads API provides routines to explicitly set thread scheduling policies and priorities which may override the default mechanism
- Thread-specific Data
  - The local data on a threads' stack comes and goes as threads call and return from different routines
  - To preserve stack data, you can usually pass it as an argument from one routine to the next, or store the data in a global variable associated with a thread
  - Pthreads provide a convenient way: Keys
### Windows 2000 Thread Object

<table>
<thead>
<tr>
<th>Object Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Thread ID</td>
</tr>
<tr>
<td>Thread context</td>
</tr>
<tr>
<td>Dynamic priority</td>
</tr>
<tr>
<td>Base priority</td>
</tr>
<tr>
<td>Thread processor affinity</td>
</tr>
<tr>
<td>Thread execution time</td>
</tr>
<tr>
<td>Alert status</td>
</tr>
<tr>
<td>Suspension count</td>
</tr>
<tr>
<td>Impersonation token</td>
</tr>
<tr>
<td>Termination port</td>
</tr>
<tr>
<td>Thread exit status</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Services</th>
</tr>
</thead>
<tbody>
<tr>
<td>Create thread</td>
</tr>
<tr>
<td>Open thread</td>
</tr>
<tr>
<td>Query thread information</td>
</tr>
<tr>
<td>Set thread information</td>
</tr>
<tr>
<td>Current thread</td>
</tr>
<tr>
<td>Terminate thread</td>
</tr>
<tr>
<td>Get context</td>
</tr>
<tr>
<td>Set context</td>
</tr>
<tr>
<td>Suspend</td>
</tr>
<tr>
<td>Resume</td>
</tr>
<tr>
<td>Alert thread</td>
</tr>
<tr>
<td>Test thread alert</td>
</tr>
<tr>
<td>Register termination port</td>
</tr>
</tbody>
</table>

### Reference Links

  - Many examples and examples with bugs
- [http://yolinux.com/TUTORIALS/LinuxTutorialPosixThreads.html](http://yolinux.com/TUTORIALS/LinuxTutorialPosixThreads.html)
- Textbook P. 696 – 698, 701-704