Chapter 4: Threads – Outline

- What are threads? How do they compare with processes?
- Why are threads important?
- What are the common multithreading models?
- What are the common multithreading libraries?
- Discussion on threading issues.
- Examples of threads in contemporary OSes.

Process Overview

- The basic unit of CPU utilization is a process.
- To run a program (a sequence of instructions), create a process.
- Process properties
  - `fork()` and `exec()` can be used to start new program execution
  - processes are well protected from each other
  - context-switching between processes is fairly expensive
  - inter-process communication used for information sharing and coordination between processes
    - shared memory
    - message passing

Is the Process Abstraction Always Suitable?

- Consider characteristics for a game software
  - different code sequences for different game objects
    - soldiers, cities, airplanes, cannons, user-controlled heroes, etc.
  - each object is more or less independent
- Problems
  - single monolithic process may not utilize resources optimally
    - can create a process for each object
  - action of an object depends on game state
    - sharing and co-ordination of information necessary
    - IPC is expensive
  - number of objects proceed simultaneously
    - may involve lots of context switches
    - process context switches are expensive

Is the Process Abstraction Always Suitable? (2)

- Ability to run multiple sequences of code (threads of control) for different object
  - individual process only offers one thread of control
- Way for threads of control to share data effectively
  - processes NOT designed to do this
- Protection between threads of control not very important
  - all in one application, anyway!
  - process is an overkill
- Switching between threads of control must be efficient
  - context switching involves a lot of overhead
- Different threads of control may share most information
  - processes duplicate entire address space
Threads to the Rescue

- **Threads** are designed to achieve all the above requirements!
  - do as little as possible to allow execution of a thread of control

- Thread are known as a *lightweight* process
  - only the necessary context information is re-generated
    - thread-context: PC, registers, stack, other misc. info
    - process-context: also includes data and code regions
  - threads are executed within a process
    - code and data shared among different threads
    - reduced communication overhead
  - smaller context
    - faster context switching
  - a single address space for all threads in a process
    - reduced inter-thread protection

Single and Multithreaded Process

Thread Basics

- Thread – *a lightweight process*
  - have their own independent flow of control
  - share process resources with other sibling threads
  - exist within the context space of the same process

- Threads shared data
  - process instructions
  - most data
  - open files (descriptors)
  - signals and signal handlers
  - current working directory
  - user and group id

- Threads specific data
  - thread id
  - registers, stack pointer
  - thread-specific data (stack of activation records)
  - signal mask
  - scheduling properties
  - return value

Thread Benefits

- Responsiveness
  - for an interactive user, if part of the application is blocked

- Resource Sharing
  - easier, via memory sharing
  - be aware of synchronization issues

- Economy
  - sharing reduces creation, context-switching, and space overhead

- Scalability
  - can exploit computational resources of a multicore CPU

source: https://computing.llnl.gov/tutorials/pthreads/
Threads can be created using the Pthreads library

- IEEE POSIX C language thread programming interface
- may be provided either as user-level or kernel-level

Pthreads API

- Thread management – functions to create, destroy, detach, join, set/query thread attributes
- Mutexes – functions to enforce synchronization. Create, destroy, lock, unlock mutexes
- Condition variables – functions to manage thread communication. Create, destroy, wait and signal based on specified variable values

Pthreads Example (2)

.... (cont. from previous page...)

/* The thread will begin control in this function */
void *runner (void *param)
{
    int i, upper = atoi(param);
    sum = 0;
    for(i=1; i<=upper; i++)
        sum += i;
    pthread_exit(0);
}

Pthreads Example – API Calls

- pthread_attr_init – initialize the thread attributes object
  - defines the attributes of the thread created
- pthread_create – create a new thread
  - upon success, a new thread id is returned in thread
- pthread_join – wait for thread to exit
  - calling process blocks until thread exits
- pthread_exit – terminate the calling thread
  - void pthread_exit(void *value_ptr);
User Vs. Kernel Level Threads

- User-level threads – manage threads in user code
  - Advantages – efficient and flexible in space, speed, switching, and scheduling
  - Disadvantages – one thread blocked on I/O can block all threads, difficult to automatically take advantage of SMP
  - Examples of thread libraries – POSIX Pthreads, Windows threads, Java Threads, GNU portable Threads

- Kernel-level threads – kernel manages the threads
  - Advantages – removes disadvantages of user-level threads
  - Disadvantages – greater overhead due to kernel involvement
  - Examples – provided by almost all GP OS
    - Windows, Solaris, Linux, Mac OS, etc.

Multithreading Models

- Relationships between user and kernel threads
  - Many-to-One
  - One-to-One
  - Many-to-Many

Many-to-One Multithreading Model

- Many user-level threads mapped to single kernel thread
  - Examples – Solaris Green Threads, GNU Portable Threads

One-to-One Multithreading Model

- Each user-level thread maps to kernel thread
  - Examples – Windows NT/XP/2000, Linux, Solaris 9 and later
Many-to-Many Multithreading Model

- User level threads mapped to \( n \) kernel threads
  - Operating system can create a sufficient number of kernel threads
  - Examples – Solaris prior to v9, Windows NT/2000 ThreadFiber package

Two-level Multithreading Model

- Similar to M:M, except that it also allows a user thread to be bound to kernel thread
  - Examples – IRIX, HP-UX, Tru64 UNIX, Solaris 8 and earlier

Threading Issues

- Semantics of \texttt{fork()} and \texttt{exec()} system calls
- Thread cancellation of target thread
- Signal handling
- Thread pools
- Thread-specific data
- Scheduler activations

Semantics of \texttt{fork()} and \texttt{exec()}

- Does \texttt{fork()} duplicate only the calling thread or all threads?
  - Some systems provide two versions of \texttt{fork()}
- How about \texttt{exec()}?
  - Most systems maintain the semantics of \texttt{exec()}
- Observations
  - \texttt{exec()} called immediately after \texttt{fork}
    - Duplicating all threads is unnecessary
  - \texttt{exec()} not called after \texttt{fork}
    - New process should duplicate all threads
**Thread Cancellation**

- Terminating a thread before it has finished
- Asynchronous cancellation
  - terminates the target thread immediately
  - allocated resources may not all be freed easily
  - status of shared data may remain ill-defined
- Deferred cancellation
  - target thread terminates itself
  - orderly cancellation can be easily achieved
  - failure to check cancellation status may cause issues

**Signal Handling**

- Signals are used in UNIX systems to notify a process that a particular event has occurred
- A *signal handler* is used to process signals
  - OS may deliver the signal to the appropriate process
  - OS or process handles the signal
- Types of signals
  - synchronous – generated by some event in the process
  - asynchronous – generated by an event outside the process
- Where to deliver a signal in multithreaded programs?
  - deliver the signal to the thread to which the signal applies
  - deliver the signal to every thread in the process
  - deliver the signal to certain threads in the process
  - assign a specific thread to receive all signals for the process

**Thread Pools**

- Concerns with multithreaded applications
  - continuously creating and destroying threads is expensive
  - overshooting the bound on concurrently active threads
- Thread Pools
  - create a number of threads in a pool where they await work
  - number of threads can be proportional to the number of processors
- Advantages
  - faster to service a request with an existing thread than create a new thread every time
  - allows the number of threads in the application(s) to be bound to the size of the pool

**OpenMP**

- Compiler directives and an API for C, C++, FORTRAN
- Supports parallel programming in shared-memory environments
- User identifies parallel region
- Create as many threads as there are cores
  ```c
  #pragma omp parallel
  { 
    printf("I am a parallel region.");
  }
  
  /* sequential code */
  int main(int argc, char *argv[])
  { 
    /* sequential code */
    for(i=0 ; i<N ; i++)
    { 
      c[i] = a[i] + b[i];
    }
  }
  ```
Linux Thread Implementation

- Linux refers to them as **tasks** rather than **threads**
- Thread creation is done through **clone()** system call
- **clone()** allows a child task to share the address space of the parent task (process)

<table>
<thead>
<tr>
<th>flag</th>
<th>meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>CLONE_FS</td>
<td>File-system information is shared.</td>
</tr>
<tr>
<td>CLONE_VM</td>
<td>The same memory space is shared.</td>
</tr>
<tr>
<td>CLONE_SIGHAND</td>
<td>Signal handlers are shared.</td>
</tr>
<tr>
<td>CLONE_FILES</td>
<td>The set of open files is shared.</td>
</tr>
</tbody>
</table>

Windows XP Thread Implementation

- Implements the one-to-one mapping, kernel-level
- Each thread contains
  - A thread id
  - Register set
  - Separate user and kernel stacks
  - Private data storage area
- The register set, stacks, and private storage area are known as the **context** of the threads
- The primary data structures of a thread include:
  - ETHREAD (executive thread block)
  - KTHREAD (kernel thread block)
  - TEB (thread environment block)

Windows XP Threads

Multicore Processors

- Multiple processing cores on a single chip.
- Reasons for a shift to multicore processors
  - power wall
  - limits to frequency scaling
  - transistor scaling still a reality
- Multicore programming Vs. multicomputer programming
  - same-chip communication is faster
  - memory sharing is easier and faster
Single Core Vs. Multicore Execution

**Single core execution**

- L1, L2, L3, L4, L1, L2, L3, L4, L1, ...

**Multiple core execution**

- Core 1: T1, T3, T1, T3, T1, ...
- Core 2: T2, T4, T2, T4, T2, ...

Challenges for Multicore Programming

- Dividing activities
- Balance
- Data splitting
- Data dependency
- Testing and debugging