Threads

Jin-Soo Kim (jinsookim@skku.edu)
Computer Systems Laboratory
Sungkyunkwan University
http://csl.skku.edu
Concurrency

- Virtualization
  - Virtual CPUs
  - Virtual memory

- Concurrency
  - In the user space by running multi-threaded programs
  - In the kernel space too!

- OS Issues
  - How to support multi-threaded programs?
  - How to coordinate accesses to shared resources?
Motivation

- Process is a cool abstraction to run a new program
  - OS provides protection and isolation among processes
- But, …
  - A single process cannot benefit from multi-cores
  - Very cumbersome to write a program with many cooperating processes
  - Expensive to create a new process
  - High communication overheads between processes
  - Expensive context switching between processes
- How can we increase concurrency within a process cheaply?
What is a Thread?

- A thread of control: A sequence of instructions being executed in a program
- A thread has its own
  - Thread ID
  - Set of registers including PC & SP
  - Stack
- Threads share an address space
- Separate the concept of a process from its execution state
Using Threads

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

void *hello (void *arg)
{
    printf ("hello, world\n");
    ...
}

int main()
{
    pthread_t tid;

    pthread_create (&tid, NULL, hello, NULL);
    printf ("hello from main thread\n");
    ...
}
```
Address Space with Threads

![Diagram of address space with threads]

- PC (T2) → Code
- PC (T1) → Data
- PC (T3) → Stack
- SP (T2) → Heap
- SP (T3) → Stack
- SP (T1) → Stack

program

code
data
Processes vs. Threads

- A thread is bound to a single process
- A process, however, can have multiple threads
- Sharing data between threads is cheap; all see the same address space
- Threads are the unit of scheduling
- Processes are containers in which threads execute
  - PID, Address space, user and group ID, open file descriptors, current working directory, etc.
- Processes are static, while threads are the dynamic entities
Benefits of Multi-threading

- Creating concurrency is cheap
- Improves program structure
  - Divide large task across several cooperative threads
- Throughput
  - By overlapping computation with I/O operations
- Responsiveness
  - Can handle concurrent events (e.g. web servers)
- Resource sharing
- Utilization of multi-core architectures
  - Allows building parallel programs
# OS Classification

<table>
<thead>
<tr>
<th># threads per addr space:</th>
<th># of addr spaces:</th>
<th>One</th>
<th>Many</th>
</tr>
</thead>
<tbody>
<tr>
<td>One</td>
<td>One</td>
<td>MS/DOS Early Macintosh</td>
<td>Traditional UNIX Xv6</td>
</tr>
<tr>
<td>Many</td>
<td>Many</td>
<td>Many embedded OSes (VxWorks, uClinux, ..)</td>
<td>Mach, OS/2, Linux, Windows, Mac OS X, Solaris, HP-UX</td>
</tr>
</tbody>
</table>
Threads Interface

▪ Pthreads (POSIX Threads)
  • A POSIX standard (IEEE 1003.1c) API for thread creation and synchronization
  • API specifies the behavior of the thread library
  • Implementation is up to development of the library
  • Common in Unix-like operating systems
    – Linux, Mac OS X, Solaris, FreeBSD, NetBSD, OpenBSD, etc.

▪ Microsoft Windows has its own Thread API
  • Win32/Win64 threads
Pthreads (I)

- Thread creation/termination

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

void pthread_exit (void *retval);

int pthread_join (pthread_t tid,
                   void **thread_return);
```
Pthreads (2)

- Mutexes

```c
int pthread_mutex_init
    (pthread_mutex_t *mutex,
    const pthread_mutexattr_t *mattr);

void pthread_mutex_destroy
    (pthread_mutex_t *mutex);

void pthread_mutex_lock
    (pthread_mutex_t *mutex);

void pthread_mutex_unlock
    (pthread_mutex_t *mutex);
```
Pthreads (3)

- Condition variables

```c
int pthread_cond_init
    (pthread_cond_t *cond,
     const pthread_condattr_t *cattr);

void pthread_cond_destroy
    (pthread_cond_t *cond);

void pthread_cond_wait
    (pthread_cond_t *cond,
     pthread_mutex_t *mutex);

void pthread_cond_signal
    (pthread_cond_t *cond);

void pthread_cond_broadcast
    (pthread_cond_t *cond);
```
Kernel-level Threads

- **OS-managed threads**
  - OS manages threads and processes
  - All thread operations are implemented in the kernel
  - Thread creation and management requires system calls
  - OS schedules all the threads
  - Creating threads are cheaper than creating processes
  - Windows, Linux, Solaris, Mac OS X, AIX, HP-UX, …
Kernel-level Threads: Limitations

- They can still be too expensive
- Thread operations are all system calls
- Must maintain kernel state for each thread
  - Can place limit on the number of simultaneous threads
- OS must scale well with increasing number of threads
- Kernel-level threads have to be general to support the needs of all programmers, languages, runtime systems, etc.
User-level Threads

- Threads are implemented at the user level
  - A library linked into the program manages the threads
  - Threads are invisible to the OS
  - All the thread operations are done via procedure calls (no kernel involvement)
  - Small and fast: 10-100x faster than kernel-level threads
  - Portable
  - Tunable to meet application needs
User-level Threads: Limitations

- Usually, rely on non-preemptive scheduling
  - Preemptive scheduling can be emulated using Unix signals

- OS can make poor decisions as it is not aware of user-level threads
  - Scheduling a process with only idle threads
  - Blocking the entire process when a thread initiates I/O
  - Unscheduling a process with a thread holding a lock

- All blocking system calls should be emulated in the library via non-blocking calls to the kernel
  - Requires coordination between kernel and thread manager

- Cannot leverage multi-core CPUs