Threads

Disclaimer: some slides are adopted from the book authors’ slides with permission
Recap

• Process
  – Address space
  – CPU context
  – OS resources

• IPC
  – Shared memory
  – Message passing
Quiz

• A process produces 100MB data in memory. You want to share the data with two other processes so that each of which can access half the data (50MB each). What IPC mechanism will you use and why?
Today

- 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.
Concurrent Programs

- Objects (tanks, planes, ...) are moving simultaneously
- Now, imagine you implement each object as a process. Any problems?
Why Processes Are Not Always Ideal?

• Not memory efficient
  – Each process consumes memory (PCB and OS resources)
    • # cat /sys/kernel/slab/task_struct/slab_size
      5904  ← 5KB for each task_struct (PCB in Linux)

• Sharing data between processes is not easy
  – No direct access to others’ address space
  – Need to use IPC mechanisms
Better Solutions?

• We want to run things concurrently
  – i.e., multiple independent threads of control

• We want to share memory easily
  – Protection is not really big concern

• We want to do these things efficiently
  – Don’t want to waste memory
Threads

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

Threads are known as a lightweight process
- only the necessary context information is re-generated
  - thread-context: PC, registers, stack, other misc. info
  - process-context: the entire address space
- 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
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
Single and Multithreaded Process

Source: https://computing.llnl.gov/tutorials/pthreads/
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
Thread Programming In Linux

- Threads can be created using the *Pthreads* library
  - IEEE POSIX C language thread programming interface

- 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

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

int sum; /* data shared by all threads */
void runner(void param); /* thread function prototype */

int main (int argc, char *argv[])
{
    pthread_t tid; /* thread identifier */
    pthread_attr_t attr /* set of thread attributes */

    if(atoi(argv[1]) < 0) {
        fprintf(stderr, "%d must be >=0\n", atoi(argv[1]));
        return -1;
    }

    pthread_attr_init(&attr);
    /* create the thread */
pthread_create(&tid, &attr, runner, argv[1]);
    /* wait for the thread to exit */
pthread_join(tid, NULL);
    fprintf(stdout, "sum = %d\n", sum);
}
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);
}
Pthread Example – API Calls

- **pthread_attr_init** – initialize the thread attributes object
  - int pthread_attr_init(pthread_attr_t *attr);
  - defines the attributes of the thread created

- **pthread_create** – create a new thread
  - int pthread_create(pthread_t *restrict thread, const pthread_attr_t *restrict attr, void *(*start_routine)(void*), void *restrict arg);
  - upon success, a new thread id is returned in `thread`

- **pthread_join** – wait for thread to exit
  - int pthread_join(pthread_t thread, void **value_ptr);
  - calling process blocks until thread exits

- **pthread_exit** – terminate the calling thread
  - void pthread_exit(void *value_ptr);
  - make return value available to the joining thread
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
    - GNU Portable Threads, Free BSD's userland threads, QuickThreads

- **Kernel-level threads** – kernel manages the threads
  - Advantages
    - removes disadvantages of user-level threads
  - Disadvantages
    - greater overhead due to kernel involvement
  - Examples -- Pthreads, Windows XP threads, 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

![Diagram showing Many-to-One Multithreading Model]
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

- $m$ 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 `fork()` and `exec()` system calls
- Thread cancellation of target thread
- Signal handling
- Thread pools
- Thread-specific data
- Scheduler activations
Semantics of fork() and exec()

Does fork() duplicate only the calling thread or all threads?
- some systems provide two versions of fork()

How about exec()?
- most systems maintain the semantics of exec()

Observations
- exec() called immediately after fork
  - duplicating all threads is unnecessary
- exec() not called after 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
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

ETHREAD
- thread start address
- pointer to parent process

KTHREAD
- scheduling and synchronization information
- kernel stack

TEB
- thread identifier
- user stack
- thread-local storage

kernel space

user space
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**

**Multiple core execution**
Challenges for Multicore Programming

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