CS370 Operating Systems
Colorado State University
Yashwant K Malaiya
Spring 20 Lecture 7
Threads

Slides based on
- Text by Silberschatz, Galvin, Gagne
- Various sources
FAQ

- Is main in C the process and functions are threads?
  - The whole program runs as a process, some functions may run as separate threads concurrently/in parallel.

- Why use pipes: inter-process communications

- What are pipes? Functions, arrays, strings?
  - Special kind of files

- Who ensures that pipes work as expected?

- Why ordinary pipes require a parent-child relationship?
FAQ Threads

• **Process vs threads**
  - Threads within a program share the same address space, files etc. Each thread has its own registers (inc PC) and stack.
  - Threads can run
    - concurrently or
    - in parallel (need extra hardware/multiple cores)

• **What are pipes?** Special kind of files

• **“Ordinary pipes”** use a parent-child relationship and cannot be used by outside processes.
Today

• Amdahl’s law
• Kernel support for threads
• Pthreads
• Java Threads
• Implicit threading
• Signals, thread cancellation
• Start CPU scheduling
Amdahl’s Law

Identifies performance gains from adding additional cores to an application that has both serial and parallel components.

- $S$ is serial portion (as a fraction) that cannot be broken into parallel operations.
- Some things can possibly be done in parallel.
- $N$ processing cores

$$speedup \leq \frac{1}{S + \frac{(1-S)}{N}}$$

- **Example**: if application is 75% parallel / 25% serial, moving from 1 to 2 cores results in speedup of 1.6 times
- As $N$ approaches infinity, speedup approaches $1 / S$

Serial portion of an application has disproportionate effect on performance gained by adding additional cores
Amdahls law: ordinary life example

Which of the two option is faster?

– Person A cooks, person B eats and then Person C eats.
– Person A cooks, then both person B and person C eat at the same time.
User Threads and Kernel Threads

- **User threads** - management done by user-level threads library
- Three main thread libraries:
  - POSIX **Pthreads**
  - Windows threads
  - Java threads
- **Kernel threads** - Supported by the Kernel
- Examples – virtually all general purpose operating systems, including:
  - Windows
  - Linux
  - Mac OS X
Multithreading Models

How do kernel threads support user process threads?

• Many-to-One

• One-to-One  \textit{(now common)}

• Many-to-Many
Many-to-One

- Many user-level threads mapped to single kernel thread *(thread library in user space older model)*
- One thread blocking causes all to block
- Multiple threads may not run in parallel on multicore system because only one may be in kernel at a time
- Few systems currently use this model
- Examples:
  - Solaris Green Threads for Java 1996
  - GNU Portable Threads 2006
One-to-One

- Each user-level thread maps to kernel thread
- Creating a user-level thread creates a kernel thread
- More concurrency than many-to-one
- Number of threads per process sometimes restricted due to overhead

Examples
  - Windows
  - Linux
  - Solaris 9 and later
Many-to-Many Model

- Allows many user level threads to be mapped to smaller or equal number of kernel threads
- Allows the operating system to create a sufficient number of kernel threads
- Solaris prior to version 9 (2002-3)
- Windows with the *ThreadFiber* package (NT/2000)
Two-level Model

• Similar to M:M, except that it allows a user thread to be **bound** to a kernel thread

• Examples
  – IRIX -2006
  – HP-UX
  – Tru64 UNIX
  – Solaris 8 and earlier
Thread Libraries

• **Thread library** provides programmer with API for creating and managing threads

• Two primary ways of implementing
  – Library entirely in user space
  – Kernel-level library supported by the OS
• May be provided either as user-level or kernel-level
• A POSIX standard (IEEE 1003.1c) API for thread creation and synchronization 1991
• *Specification*, not *implementation*
• API specifies behavior of the thread library, implementation is up to development of the library
• Common in UNIX operating systems (Solaris, Linux, Mac OS X)
Some Pthread management functions

<table>
<thead>
<tr>
<th>POSIX function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>pthread_cancel</td>
<td>Terminate a thread</td>
</tr>
<tr>
<td>pthread_create</td>
<td>Create a thread</td>
</tr>
<tr>
<td>pthread_detach</td>
<td>Set thread to release resources</td>
</tr>
<tr>
<td>pthread_exit</td>
<td>Exit a thread without exiting process</td>
</tr>
<tr>
<td>pthread_kill</td>
<td>Send a signal to a thread</td>
</tr>
<tr>
<td>pthread_join</td>
<td>Wait for a thread</td>
</tr>
<tr>
<td>pthread_self</td>
<td>Find out own thread ID</td>
</tr>
</tbody>
</table>

• Return 0 if successful
POSIX: Thread creation  

- Automatically makes the thread runnable without a start operation
- Takes 3 parameters:
  - Points to ID of newly created thread
  - Attributes for the thread
    - Stack size, scheduling information, etc.
  - Name of function that the thread calls when it begins execution with argument

/* create the thread */

pthread_create(&tid, &attr, runner, argv[1]);
POSIX: Detaching and Joining

• **pthread_detach()**
  – Sets internal options to specify that storage for thread can be reclaimed when it exits
  – 1 parameter: Thread ID of the thread to detach
  – Undetached threads don’t release resources until
    • Another thread calls `pthread_join` for them
    • Or the whole process exits

• **pthread_join**
  – Takes ID of the thread to wait for
  – Suspends calling thread till target terminates
  – Similar to `waitpid` at the process level
  `pthread_join(tid, NULL);`
POSIX: Exiting and cancellation

• If a process calls exit, **all** threads terminate
• Call to pthread_exit causes only the calling thread to terminate

`pthread_exit(0)`

• Threads can force other threads to return through a cancellation mechanism
  – pthread_cancel: takes thread ID of target
  – Actual cancellation depends on type and state of thread
Pthreads Example (next 2 slides)

• This process will have two threads
  – Initial/main thread to execute the main() function. It creates a new thread and waits for it to finish.
  – A new thread that runs function runner()
    • It will get a parameter, an integer, and will compute the sum of all integers from 1 to that number.
    • New thread leaves the result in a global variable sum.
  – The main thread prints the result.
#include <pthread.h>
#include <stdio.h>

int sum; /* this global data is shared by the thread(s) */

void *runner(void *param); /* the thread */

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

    if (argc != 2) {
        fprintf(stderr,"usage: a.out <integer value>\n");
        /*exit(1);*/
        return -1;
    }

    if (atoi(argv[1]) < 0) {
        fprintf(stderr,"Argument %d must be non-negative\n",atoi(argv[1]));
        /*exit(1);*/
        return -1;
    }
}
/* get the default attributes */
pthread_attr_init(&attr);
/* create the thread */
pthread_create(&tid, &attr, runner, argv[1]);
/* now wait for the thread to exit */
pthread_join(tid, NULL);

printf("sum = %d\n", sum);
}
/* The thread will begin control in this function */
void *runner(void *param)
{
    int i, upper = atoi(param);
    sum = 0;
        if (upper > 0) {
            for (i = 1; i <= upper; i++)
                sum += i;
        }
    pthread_exit(0);
}
/* create the threads */
for (i = 0; i < NUM_THREADS; i++)
    pthread_create(&tid[i],&attr,runner,NULL);

/* now join on each thread */
for (i = 0; i < NUM_THREADS; i++)
    pthread_join(tid[i], NULL);
}

/* Each thread will begin control in this function */

void *runner(void *param)
{
    /* do some work ... */
    pthread_exit(0);
}
Java Threads

- Java threads are managed by the JVM
- Typically implemented using the threads model provided by underlying OS
- Java threads may be created by:
  - Extending Thread class
    - Override its run() method
  - More commonly, implementing the Runnable interface
    1. Has 1 method run()
    2. Create new Thread class by passing a Runnable object to its constructor
    3. start() method creates a new thread by calling the run() method.
      - new features available in java.util.concurrent package

Runnable interface is defined by

```java
public interface Runnable
{
    public abstract void run();
}
```
Java Thread States

New

Runnable

Running

Non Runnable (Blocked)

Terminated

start()

sleep() done, i/o complete, lock available, resume(), notify() or notifyAll()

run() method exits or stop()

sleep(), block on I/O, wait for lock, suspend(), wait()

https://www.javatpoint.com/life-cycle-of-a-thread
Java version of a multithreaded program that computes summation of a non-negative integer.
This program creates a separate thread by implementing the Runnable interface.

class Sum
{
    private int sum;

    public int get() {
        return sum;
    }

    public void set(int sum) {
        this.sum = sum;
    }
}

Program Overall Structure

```java
class sum {
 ...
}
class summation implements runnable {
    ...
    public void run() {
    }
}

Public class Driver {
    ...
    public static void main(String[] args) {
        Thread worker = new Thread(new summation(...
        worker.start();
        try {
            worker.join();
        }
    }
```
class Summation implements Runnable
{
    private int upper;
    private Sum sumValue;

    public Summation(int upper, Sum sumValue) {
        if (upper < 0)
            throw new IllegalArgumentException();
        this.upper = upper;
        this.sumValue = sumValue;
    }

    public void run() {
        int sum = 0;
        for (int i = 0; i <= upper; i++)
            sum += i;
        sumValue.set(sum);
    }
}
public class Driver {
    public static void main(String[] args) {
        if (args.length != 1) {
            System.err.println("Usage Driver <integer>");
            System.exit(0);
        }

        Sum sumObject = new Sum();
        int upper = Integer.parseInt(args[0]);

        Thread worker = new Thread(new Summation(upper, sumObject));
        worker.start();
        try {
            worker.join();
        } catch (InterruptedException ie) {
        }
        System.out.println("The sum of "+ upper + " is "+ sumObject.get());
    }
}
Implicit Threading

- Growing in popularity as numbers of threads increase, program correctness more difficult with explicit threads
- Creation and management of threads done by compilers and run-time libraries rather than programmers
- Three methods explored
  - Thread Pools
  - OpenMP
  - Grand Central Dispatch
- Other methods include Microsoft Threading Building Blocks (TBB), java.util.concurrent package
Implicit Threading1: Thread Pools

- Create a number of threads in a pool where they await work
- Advantages:
  - Usually slightly faster to service a request with an existing thread than create a new thread
  - Allows the number of threads in the application(s) to be bound to the size of the pool
  - Separating task to be performed from mechanics of creating task allows different strategies for running task
    - i.e. Tasks could be scheduled to run periodically
- Posix thread pools
- Windows API supports thread pools.
Implicit Threading2: OpenMP

- Set of compiler directives and an API for C, C++, FORTRAN
- Provides support for parallel programming in shared-memory environments
- Identifies **parallel regions** – blocks of code that can run in parallel

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

int main(int argc, char *argv[]) {
    /* sequential code */
    #pragma omp parallel
    {
        printf("I am a parallel region.");
    }
    /* sequential code */
    return 0;
}
```

- #pragma omp parallel
  - Create as many threads as there are cores
- #pragma omp parallel for
  - for(i=0; i<N; i++) {
    - c[i] = a[i] + b[i];
  }
  - Run for loop in parallel

Compile using
```sh
gcc -fopenmp openmp.c
```
Implicit Threading#: Grand Central Dispatch

- Apple technology for Mac OS X and iOS operating systems
- Extensions to C, C++ languages, API, and run-time library
- Allows identification of parallel sections
- Manages most of the details of threading
- Block is in “^{ }”
  - ^{ printf("I am a block"); }
- Blocks placed in dispatch queue
  - Assigned to available thread in thread pool when removed from queue
Threading Issues

- Semantics of `fork()` and `exec()` system calls?
- Signal handling
  - Synchronous and asynchronous?
- Thread cancellation of target thread
  - Asynchronous or deferred?
- Thread-local storage?
Semantics of fork() and exec()

• Does fork() duplicate only the calling thread or all threads?
  – Some UNIXes have two versions of fork
    – 1. when exec() will replace the entire process, dup just that thread
    – 2. duplicate all threads

• exec() usually works as normal – replace the running process including all threads
• **Signals** are used in UNIX systems to notify a process that a particular event has occurred.

• A **signal handler** is used to process signals
  1. Signal is generated by particular event
  2. Signal is delivered to a process
  3. Signal is handled by one of two signal handlers:
     1. default
     2. user-defined

• Every signal has **default handler** that kernel runs when handling signal
  – **User-defined signal handler** can override default
  – For single-threaded, signal delivered to process
• Where should a signal be delivered for multi-threaded process?
  – 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? common
Thread Cancellation

• Terminating a thread before it has finished
• Thread to be canceled is **target thread**
• Two general approaches:
  – **Asynchronous cancellation** terminates the target thread immediately
  – **Deferred cancellation** allows the target thread to periodically check if it should be cancelled
• Pthread code to create and cancel a thread:

```
pthread_t tid;

/* create the thread */
pthread_create(&tid, 0, worker, NULL);

... 

/* cancel the thread */
pthread_cancel(tid);
```
Invoking thread cancellation requests cancellation, but actual cancellation depends on thread state.

<table>
<thead>
<tr>
<th>Mode</th>
<th>State</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Off</td>
<td>Disabled</td>
<td></td>
</tr>
<tr>
<td>Deferred</td>
<td>Enabled</td>
<td>Deferred</td>
</tr>
<tr>
<td>Asynchronous</td>
<td>Enabled</td>
<td>Asynchronous</td>
</tr>
</tbody>
</table>

If thread has cancellation disabled, cancellation remains pending until thread enables it.

Default type is deferred.

- Cancellation only occurs when thread reaches `cancellation point`
  - i.e. `pthread_testcancel()`
  - Then `cleanup handler` is invoked

On Linux systems, thread cancellation is handled through signals.
Thread-local storage (TLS) allows each thread to have its own copy of data

- Useful when you do not have control over the thread creation process (i.e., when using a thread pool)
  - Ex: Each transaction has a thread and a transaction identifier is needed.

- Different from local variables
  - Local variables visible only during single function invocation
  - TLS visible across function invocations

- Similar to static data
  - TLS is unique to each thread
Is complexity always good?

• Is something that is
  – More advanced
  – More complex

Generally better?
Hyper-threading

• “Hyper-threading”: simultaneous multithreading:
  – Hardware support for multiple threads in the same core (CPU)

• Performance:
  – performance improvements are very application-dependent
  – Higher energy consumption
  – Not better than out-of-order execution
  – Intel has dropped it

Core i7-9700K 2018 8 cores, 8 threads
Parallelism

Forms of parallelism
– Pipelining: instruction flows though multiple levels
– Multiple issue: Instruction level Parallelism (ILP)
  • Static: compiler scheduling of instructions
  • Dynamic: hardware assisted scheduling of operations
    – “Superscalar” processors
    – CPU decides whether to issue 0, 1, 2, ... instructions each cycle
– Thread or task level parallelism (TLP)
A thinking exercise

• Compare the following:
  – Calling a function
  – Starting a child
  – Creating a thread