What is a Thread?

- A thread is
  - An abstraction of a "process activity"
  - An active sequence of instructions in a process

- A process activity is defined by its context:
  - Program counter: Tells which instruction is being executed
  - Stack: Determines where in the program we are (e.g.: which function, what parameters)

Outline

- Threads:
  - Thread Concept and Usage
  - Thread Models and Implementation
  - Implicit Threading
  - Thread Issues

Threads

- Threads exist within a process
  - Multiple threads run concurrently within the same process

- Threads share
  - Process code, data, heap
  - Files, signals

- Each thread has its own
  - Program counter, stack, registers, signal mask
Process vs. Threads

<table>
<thead>
<tr>
<th>Process</th>
<th>Threads</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data</td>
<td>Data</td>
</tr>
<tr>
<td>Code</td>
<td>Code</td>
</tr>
<tr>
<td>Heap</td>
<td>Heap</td>
</tr>
<tr>
<td>Stack</td>
<td>Stack</td>
</tr>
<tr>
<td>Stack</td>
<td>Stack</td>
</tr>
<tr>
<td>Stack</td>
<td>Stack</td>
</tr>
</tbody>
</table>

Multithreading Example

```c
foo () {
    ... 
}
bar () {
    ... 
} 
void main () {
    ... 
create_thread (foo);
create_thread (bar);
    ... 
}
```

Thread Operations

- Create a thread
  - Pass it a function and arguments, attributes
- Threads run concurrently
- Join a thread
  - Makes the calling thread wait for a child
- Can also:
  - Detach a thread: Lets the thread release its resources when done
  - Cancel a thread: Kill a thread without exiting the process

Thread Example: Pthreads

```c
#include <pthread.h>

int main() { 
    pthread_t tid; 
    int sum, number=10;
    
    void *summation(void *arg){
        int i, upper = *(int *) arg;
        for (i=1; i<=upper; i++)
            sum = ... /* Compute sum */
    } 

    int main(){
        ... 
        pthread_create(&tid, NULL, summation, &number);
        pthread_join(tid, NULL); 
        ... 
    }
```
Thread Example: Win32 Threads

DWORD ThreadId;
HANDLE ThreadHandle;
int sum, number=10;

int Summation(LPVOID arg){
    int i, upper = *(int*) arg;
    for (i=1; i<=upper; i++)
        sum = ... /* Compute sum */
    }

int main(){
    ThreadHandle = CreateThread(..., Summation, &number, ...., &ThreadId);
    WaitForSingleObject(ThreadHandle, INFINITE);
}

Thread Example: Java Threads

class Summation implements Runnable {
    ...
    public void run(){
        /* Compute sum */
    }
}

class Driver{
    public static void main(String[] args){
        ...
        Thread thrd = new Thread (new Summation(...));
        thrd.start();
        thrd.join();
        ...
    }
}

Thread Benefits

- Concurrency and Responsiveness
  - When one thread is blocked, another can run
  - Great for multi-tasking applications (Web servers, file servers)
- Resource Sharing
  - Threads share resources of the process (e.g., code, data, files)
  - Less OS resources used up (e.g., memory, buffers, kernel data structures)

Thread Benefits (contd.)

- Economy
  - Thread operations cheaper than processes
  - Creating/destroying, context switches, scheduling
  - Communication (Common address-space)
- Parallelism
  - Multithreading gives real parallelism on multiprocessor machines
  - Can run multiple threads on multiple processors
Thread Problems

- Programming Complexity
  - Non-deterministic behavior
  - Need to be synchronized
  - Difficult to debug
- Scalability
  - With many threads, stacks could still use up lot of memory
  - Context switch has overhead
- Portability problems due to different implementations

Concurrency vs. Parallelism

- Concurrency: Multiple threads make progress and execution is interleaved
- Parallelism: Threads can run simultaneously on multiple CPUs
- Types of parallelism:
  - Data parallelism: Same operation executed on different parts of input data
  - Task parallelism: Different operations executed by different threads

Amdahl’s Law

- Provides a theoretical bound on achievable parallel speedup
- \[ \text{Speedup} = \frac{\text{Sequential runtime}}{\text{Parallel runtime}} \]
  \[ \text{speedup} \leq \frac{1}{S + \frac{(1-S)}{N}} \]
  \( S = \) Serial portion of program, \( N = \) num CPUs

Thread Implementation

- Can be implemented in user or kernel space
- User threads are implemented by a user-level runtime system
  - Language support or thread-package library
  - E.g.: Java, Pthreads
- Kernel threads are implemented directly inside the kernel
  - Like processes with shared address-space
  - E.g.: Linux kernel threads
User Threads

- User threads are more light-weight and efficient
  - No kernel scheduling, context-switching
- User threads are more flexible
  - Application-specific scheduling policy, concurrency model
- Blocking I/O Problem
  - If a single user thread blocks, the whole process and hence, all threads block
- User threads cannot exploit parallelism of multiprocessors

Kernel Threads

User vs. Kernel Threads

Multithreading Models

- Many systems provide support for both user and kernel threads
- Use a combination of user and kernel threads
- Map user threads to kernel threads
- Dependent on
  - OS thread support
  - User thread library implementation
Many-to-One Model

- Many user threads mapped to a single kernel thread

User threads  

Kernel thread

One-to-One Model

- Each user thread mapped to a single kernel thread

User threads  

Kernel threads

Many-to-Many Model

- Multiple user threads mapped to multiple kernel threads

User threads  

Kernel threads

Light-weight Process

- Virtual processor: Maps user threads to kernel threads

User threads  

LWPs  

Kernel threads
Scheduler Activations

- Each application assigned a certain number of virtual processors
- User-level scheduler can run user threads as it wants
- Kernel maintains a one-to-one mapping between virtual processors and CPUs
- Sends an upcall to user-level lib on specific events (e.g., thread blocked/unblocked, pre-emption)

Thread Scheduling

- Two-level scheduling: User-level and kernel-level
  - Process-contention scope: Mapping user-level threads to LWPs
  - System-contention scope: Scheduling kernel-level threads on CPUs

Multicore Scheduling

- Chip Multithreading (CMT):
  - Hyperthreading or Simultaneous multithreading (SMT)
  - Two or more hardware threads assigned to each core
  - One of the threads can execute at a time
- Hardware scheduling: When to switch between the threads on a core
  - Fine-grained vs. coarse-grained
  - Round-robin vs. priority/urgency
- OS scheduling: Which kernel threads to assign to each core
Thread Libraries

- Pthreads: POSIX Threads library
  - Specification of thread API
  - Implementation is dependent on system support
  - Could be a combination of user/kernel threads
- Win32 Threads: Windows Thread library
  - One-to-one mapping
- Java Threads: Language-level support
  - Uses host thread package/library
  - E.g.: might use Pthreads or Win32 threads

Implicit Threading

- Support from compilers and runtime systems
  - Make it easier for programmers to specify concurrency
  - System can implement threading and parallelism

Thread Pools

- Number of threads are pre-created
- Programmer creates tasks
  - Run-time system allocates threads on demand
- E.g.: multi-threaded Web server
- Benefits?

OpenMP

- API for C, C++, Fortran
  - Used for scientific computing
- Compiler directives inserted into program
  - Identify parts of code that can be parallelized
  - E.g.: #pragma omp parallel
- Compiler will parallelize code and create no. of threads based on no. of CPUs
Thread Issues

- What are the issues when a process is multi-threaded?
  - Process creation
  - Thread cancellation
  - Signal handling

fork and exec

- Single-threaded process:
  - fork makes a copy of whole process
- What happens when you have multiple threads?
  - Should new process have one thread?
  - Should new process have multiple threads?
- What happens when we call exec?

Thread Cancellation

- Cancellation:
  - Terminate a thread
  - Reclaim thread resources
- When can a thread be cancelled?
  - Asynchronous cancellation: cancel immediately
    - What if thread is modifying shared data?
  - Deferred cancellation: cancel later
    - When the target thread is ready to terminate
    - Cancellation point: Safe point in execution

Signals

- Signal: A software interrupt generated by the OS or a process
  - E.g.: SegFault, Alarm, Ctrl-C
- Signal generation and delivery
  - Signal generated when event occurs
  - Delivered when process runs and takes action
- Signal handling
  - Default action: E.g.: Terminate on Segfault
  - Ignore/Block: Throw away or delay delivery
  - Catch: User-defined signal handler
Signals and Threads

- Which thread gets the signal?
  - Specific thread?
  - All threads?
  - One signal-handling thread?
- UNIX:
  - Allow threads to block specific signals
  - Signal delivered to first thread not blocking signal
  - Pthreads allows signal delivery to thread

Thread Implementation

- What data structures does the OS maintain for a thread?
- How does thread relate to a process?
- OS Examples:
  - Windows
  - Linux

Windows Threads

- Traditional: One-to-one mapping model
- Thread context:
  - Register set
  - User and kernel stack
  - Private storage area

Windows Threads: Data Structures

- ETHREAD: Executive thread block
  - Pointer to process
  - Address of start routine
- KTHREAD: Kernel thread block
  - Scheduling, synchronization information
  - Kernel stack
- TEB: Thread environment block
  - User-space structure
  - Thread id
  - User stack
  - Thread-local storage
**Windows Threads: Other Models**

- Fibers
  - Provides many-to-many model
  - Multiple fibers exist within a single thread
  - Only one fiber within a thread runs at a time
  - Programmer has to schedule fibers
- User-mode scheduling
  - Similar to scheduler activations
  - User-level scheduler for each processor
  - Completion list: List of ready threads
  - Kernel gives notifications of specific events (thread creation, blocking, yield)

**Linux Threads**

- Task: Generalization of process/thread
- Task creation: `clone()`
  - Specify what *to share* between parent and child and what *to copy*
  - E.g.: memory space, open files, signal handlers
- What do you get if:
  - Share all?
  - Share none?

**Linux Threads: Implementation**

- Task data structure: `struct task_struct`
  - Contains pointers to other data structures (open files, memory management)
- How to enforce sharing or non-sharing of certain things and not others?
  - No sharing => copy data structures
  - Sharing => copy *pointers* to data structures