Threads
CS 3113
Concurrency vs Parallelism
Concurrent execution on single-core system:

Parallelism on a multi-core system:
Concurrency vs Parallelism

• Concurrency: rapid switching of processes onto the CPU, which gives the illusion that multiple processes are executing at once

• Parallelism: have hardware support to execute multiple things at once
Concurrency vs Parallelism

Parallelism: have hardware support to execute multiple things at once

• Core: physical unit of code execution (instruction decoding, registers, etc.)
• CPU: Nominally (today), the computing hardware on a single chip. Can contain multiple cores
• Today, we often have multiple CPU machines, with each CPU containing multiple cores
Multi-Core Programming

We can explicitly write a program that has multiple execution contexts

• Distribute across the available cores
• Do different parts of the work in parallel
Processes

Often exist in isolation:

• Separate memory
  • Program
  • Data (globals, heap, etc)

• Separate execution context
  • Program counter
  • Registers
Threads

• Memory is shared!
  • Program
  • Data (globals, heap, etc)

• Separate execution context
  • Program counter
  • Registers

We refer to an execution context as a \textit{thread}
Process vs Threads within a Process

![Diagram showing single-threaded and multithreaded processes](image-url)

**Single-threaded process**
- Code
- Data
- Files
- Registers
- Stack
- Thread

**Multithreaded process**
- Code
- Data
- Files
- Registers
- Registers
- Registers
- Stack
- Stack
- Stack
- Thread
Advantages of Multi-Thread Programming

• **Implementation:** Can divide a big task into many, easy to implement tasks

• **Responsiveness:** may allow continued execution if part of process is blocked, especially important for user interfaces

• **Resource Sharing:** threads share resources of process, easier than shared memory or message passing

• **Economy:** cheaper than process creation, and thread switching has lower overhead than context switching

• **Scalability:** process can take advantage of multiprocessor architectures
Multi-Thread Programming

Types of parallelism

• **Data parallelism** – distributes subsets of the same data across multiple cores, same operation on each

• **Task parallelism** – distributing threads across cores, each thread performing unique operation
Multi-Thread Programming

Architectural support for threading has increased in the last couple of decades

• Core: hardware pipeline for execution of instructions
  • Single instruction in CISC processors requires many steps (including operand fetch, multiple execution step, store of result)

• Hardware thread:
  • One physical thread appears to the OS as multiple independent cores
  • Implementation: have instructions in the pipeline from more than one hardware thread

• Oracle SPARC T4 with 8 cores, and 8 hardware threads per core
Multi-Thread Programming

How much faster can work be done with parallelism?
Amdahl’s Law

Performance speedup with parallelization

• S: fraction of task that is necessarily serial (rest is parallel)

• N: number of processors/cores

\[
\text{speedup} \leq \frac{1}{S + \frac{(1-S)}{N}}
\]
Amdahl’s Law

Performance speedup with parallelization

• What happens as S approaches 0?
• What happens as S approaches 1?
• What happens as N approaches infinity?

\[
\text{speedup} \leq \frac{1}{S + \frac{(1-S)}{N}}
\]
Support for Threads

Management of threads: must address the scheduling of threads for execution

• User space
  • Managed by libraries that live entirely in the user space
  • More general / portable

• Kernel space
  • Managed through systems calls to the kernel
  • Allows us to take more advantage of the available hardware
  • But: can be more hardware specific
Support for Threads

User space examples
• POSIX Pthreads
• Windows threads
• Java threads
Support for Threads

Kernel space provided by all modern OSes, including:

• Windows
• Solaris
• Linux
• Tru64 UNIX
• Mac OS X
Multithreading Models

What is the relationship between programming of threads in the user space and the implementation in the kernel space?
Multithreading Models

Relationship between user space threads and kernel threads. Options include:
- Many-to-One
- One-to-One
- Many-to-Many
Many-to-One

• Many user-level threads mapped to single kernel thread
• 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
  • GNU Portable Threads
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 many kernel threads
- Allows the operating system to create a sufficient number of kernel threads
- Solaris prior to version 9
- Windows with the **ThreadFiber** package
Thread Libraries

Programmer API for doing multithreading
Pthreads

• May be provided either as user-level or kernel-level
• A POSIX standard (IEEE 1003.1c) API for thread creation and synchronization

• 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)
Pthreads

Set up:

• Global variable(!): sum

• Function prototype: runner

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

int sum; /* this data is shared by the thread(s) */
void *runner(void *param); /* threads call this function */

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

    if (argc != 2) {
        fprintf(stderr,"usage: a.out <integer value>\n");  
        return -1;
    }
    if (atoi(argv[1]) < 0) {
        fprintf(stderr,"%d must be >= 0\n",atoi(argv[1]));  
        return -1;
    }
```
Pthreads

Parent:
• Create a single thread
  • Starts execution
• Join: parent waits for the child to exit

Child:
• Writes result to global variable

```c
/* get the default attributes */
pthread_attr_init(&attr);
/* create the thread */
pthread_create(&tid,&attr,runner,argv[1]);
/* 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;

    for (i = 1; i <= upper; i++)
        sum += i;
    pthread_exit(0);
}
```
pthreads demo