Threads
Parallel computing

• Goal: concurrent or parallel computing
  - Take advantage of multiple hardware units to solve multiple problems simultaneously

• Motivations:
  - Maintain high utilization during slow I/O downtime
  - Maintain UI responsiveness during computation
  - Respond simultaneously to multiple realtime events
  - Split up a large problem and solve sub-pieces concurrently to achieve faster time-to-solution (strong scaling)
  - Solve larger problems by adding more hardware (weak scaling)
Parallel computing

- **Process**: currently-executing program
  - Code and state (PC, stack, data, heap)
  - Private address space

- **Thread**: unit of execution or logical flow
  - Exists within the context of a single process
  - Shares code/data/heap/files w/ other threads
  - Keeps private PC, stack, and registers
    - Stacks are technically shared, but harder to access
Threads

- One main thread for each process
  - Can create multiple peer threads
POSIX threads

- **Pthreads** – POSIX standard interface for threads in C
  - Not part of the standard library
    - Requires "-lpthread" flag during linking
  - `pthread_create`: spawn a new child thread
    - `pthread_t` struct for storing thread info
    - attributes (or NULL)
    - thread work routine (function pointer)
    - thread routine parameter (void*, can be NULL)
  - `pthread_self`: get current thread ID
  - `pthread_exit`: terminate current thread
    - can also terminate implicitly by returning from the thread routine
  - `pthread_join`: wait for another thread to terminate
    - requires a `pthread_t` to wait for
```c
#include <stdio.h>
#include <pthread.h>

void* work (void* arg)
{
    printf("Hello from work routine!\n");
    return NULL;
}

int main ()
{
    printf("Spawning single child ...\n");

    pthread_t child;
    pthread_create(&child, NULL, work, NULL);
    pthread_join(child, NULL);

    printf("Done!\n");

    return 0;
}
```
Shared memory

- Global variables (shared, single static copy)
  - Often used for communication between threads
  - Requires careful coordination
- Local “automatic” variables (multiple copies, one on each stack)
  - Technically still shared if in memory, but harder to access
  - Not shared if cached in register
  - Safer to assume they're private; this is conventional
- Local static variables (shared, single static copy)
  - Similar to global variables but with reduced scope
- Heap-allocated variables (shared, dynamic)
  - Requires coordination if threads share pointers to same memory
Processes vs. threads

- **Process**: currently-executing program
  - Code and state (PC, stack, data, heap)
  - Created via system call (*fork*); parent and child continue from call site
  - **Private address space** not shared w/ other processes
  - **Advantages**: isolation, safety, and mutual exclusion

- **Thread**: unit of execution or logical flow
  - Private PC, registers, condition codes, and stack
  - Created via library call (*pthread_create*); child runs separate routine
  - **Shared address space** w/ other threads
  - **Advantages**: faster context switching, more shared resources
Issues with shared memory

foo:
  irmovq x, %rcx
  irmovq 7, %rax
  mrmovq (%rcx), %rdx
  addq %rax, %rdx
  rmmovq %rdx, (%rcx)
  ret

x:
  .quad 0

This interleaving is ok.
Issues with shared memory

foo:
irmovq x, %rcx
irmovq 7, %rax
mrmovq (%rcx), %rdx
addq %rax, %rdx
rmmovq %rdx, (%rcx)
ret

x:
.quad 0

This one is not!
Issues with shared memory

- A program is **non-deterministic** when it can produce different outputs given the same inputs.
- A **data race** occurs when correct output relies on a particular ordering during execution.
- **Deadlock** occurs when threads or processes are blocked waiting on a condition that will never happen.
Mutual exclusion

- Fixing a data race requires some form of mutual exclusion
  - Only one thread at a time should update shared memory
  - In Pthreads, this can be accomplished using either a mutex or a semaphore (more details in CS 361)
  - However, these mechanisms introduce overhead!
    - Threads must perform additional checks before updating memory
    - Some threads may have to pause and wait before they may continue
  - If not implemented carefully, the additional overhead may defeat the purpose of using multiple threads
    - Theme: Systems design requires tradeoffs
    - Theme: Details matter (a LOT!)
  - Efficient parallel and distributed computing can be very difficult
Wouldn’t it be great if the compiler could automatically parallelize our programs?

- This is a HARD problem
- In some cases, it is (kind of) possible
- Approach #1: code annotations in existing language
  - Example: OpenMP (CS 450, CS 470)
- Approach #2: new language designed for parallelism
  - Example: HPF and Chapel (CS 430, CS 470)

```c
int a[100]; #pragma omp parallel for
for (int i=0; i < 100; i++)
a[i] = i*i;
```

```chapel
var a: [100] int;
forall i in 0..100 do
   a[i] = i*i;
```

OpenMP example  Chapel example
Parallel systems

- **Uniprogramming / batch (1950s)** - CS 261
  - One process at a time w/ complete control of CPU
  - Minimal OS (mostly for launching programs)

- **Multiprogramming / multitasking / time sharing (1960s)** - CS 261, CS 450
  - Multiple processes taking turns on a single CPU
  - Increased utilization, lower response time
  - OS handles scheduling and context switching

- **(Symmetric) multiprocessing (1970s)** - CS 361, CS 450, CS 470
  - Multiple processes share multiple CPUs or cores
  - Increased throughput, increased parallelism
  - OS handles scheduling, context switching, and communication

- **Distributed processing (1980s and onward)** - CS 361, CS 470
  - Multiple processes share multiple computers
  - Massive scaling; OS no longer sufficient (other middleware required)