Threads

CSE 333 Winter 2021

Instructor: John Zahorjan

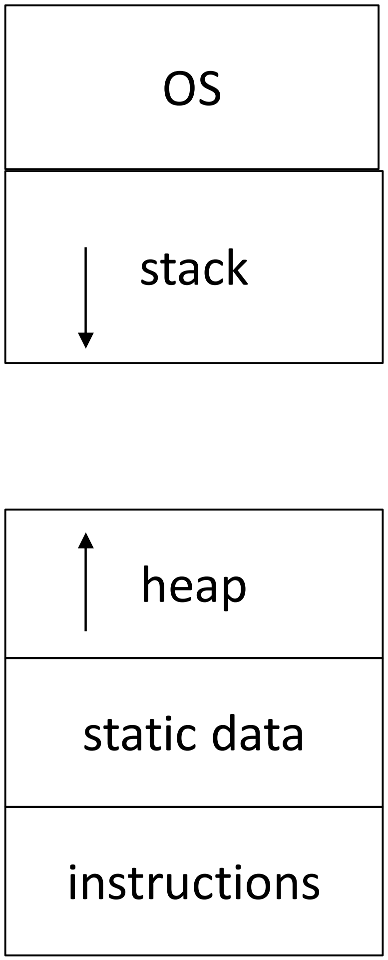
Teaching Assistants:

Matthew Arnold	Nonthakit Chaiwong	Jacob Cohen
Elizabeth Haker	Henry Hung	Chase Lee
Leo Liao	Tim Mandzyuk	Benjamin Shmidt
Guramrit Singh		

Process

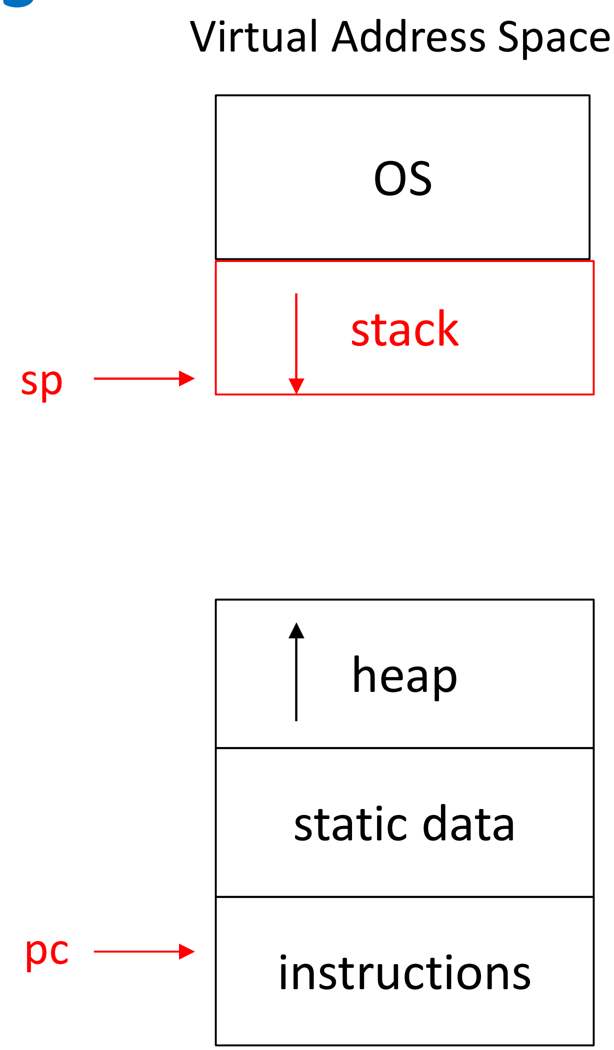
- A process is a program in execution
  - A process is associated with an address space
- A process provides isolation

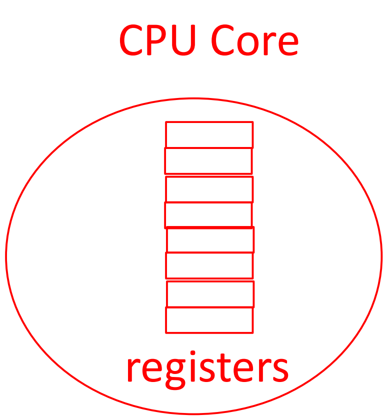
Virtual Address Space



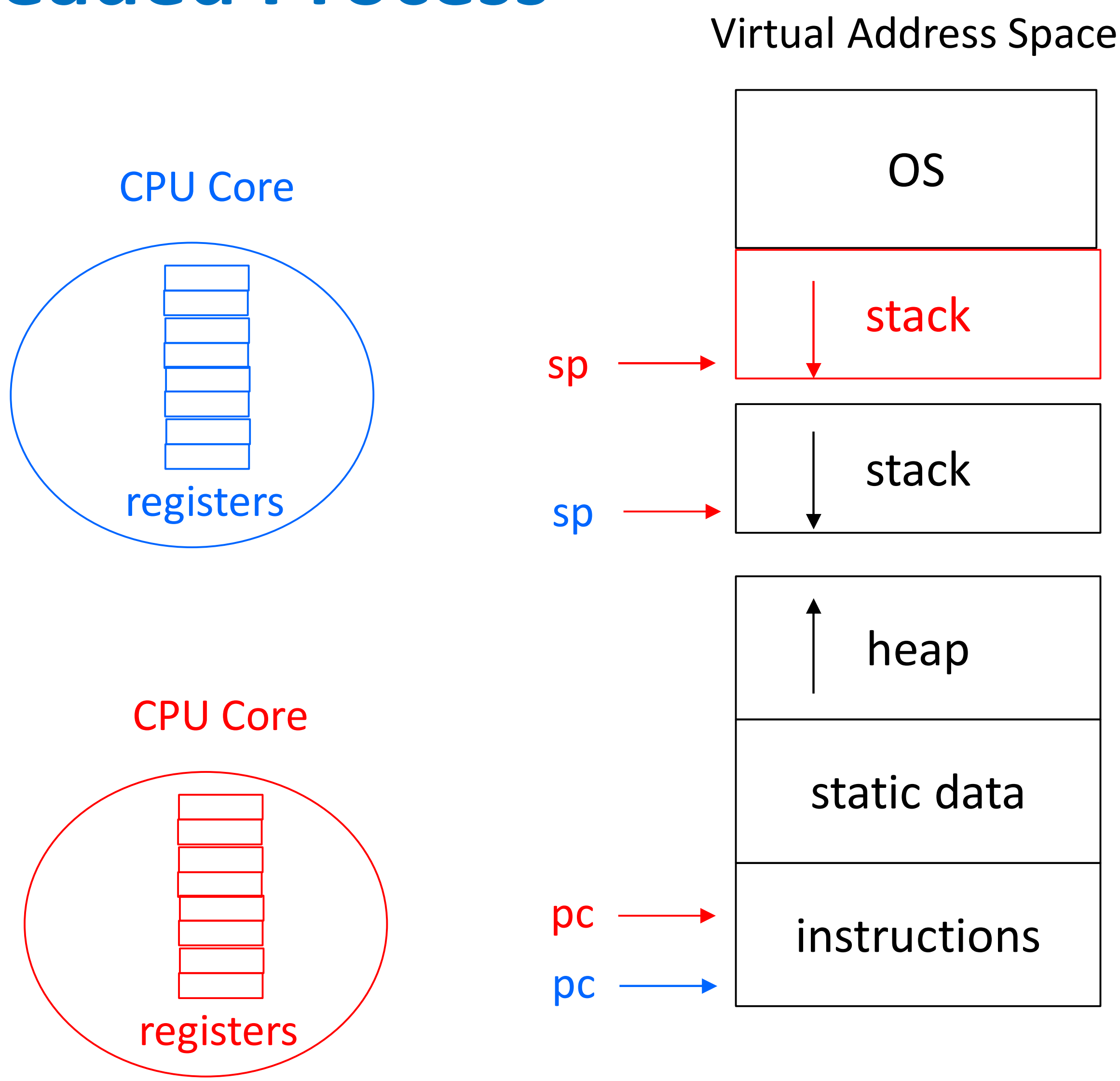
Single Threaded Process

- A process is a program in execution
- A process contains one or more threads of execution



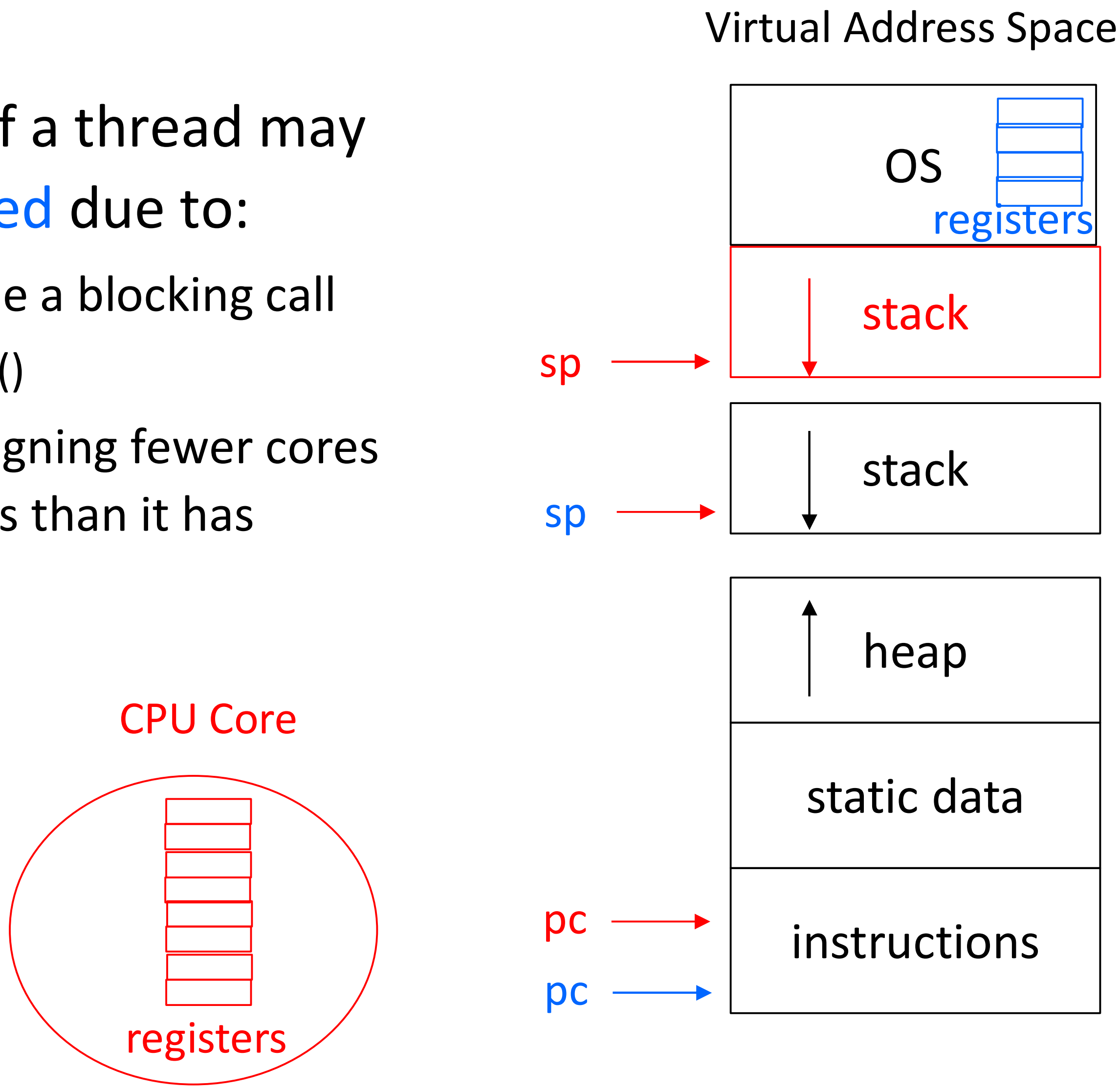


Multi-Threaded Process



Multi-Threaded Process

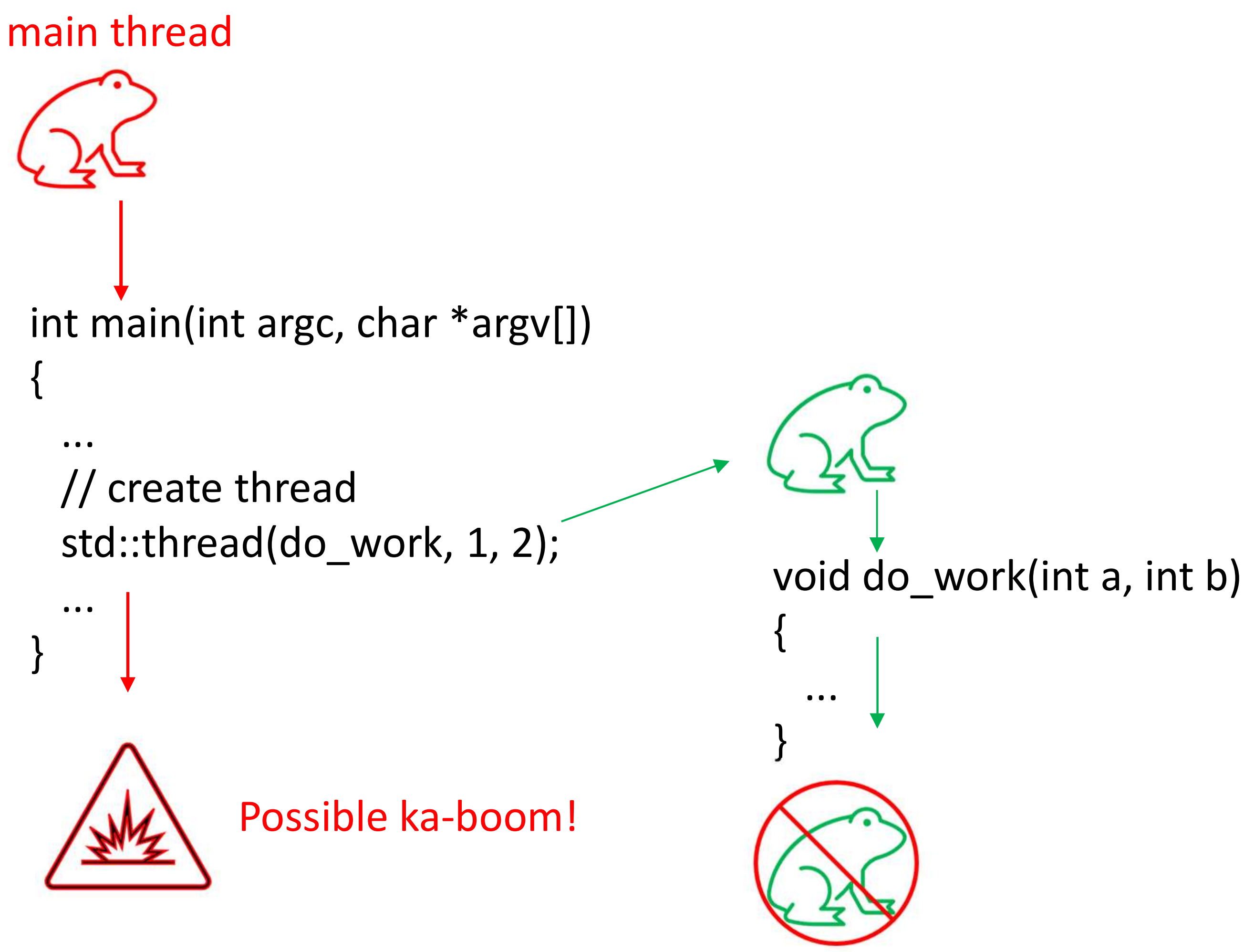
- Execution of a thread may be suspended due to:
  - Having done a blocking call
    - e.g., read()
  - The OS assigning fewer cores to a process than it has threads



C++ and Threads

- Every C++ program starts with a single main thread that begins execution in main()
- Additional threads can be created as std::thread objects
- A new thread starts execution by calling a method provided as an argument to the thread constructor
- The new thread terminates when it returns from that method

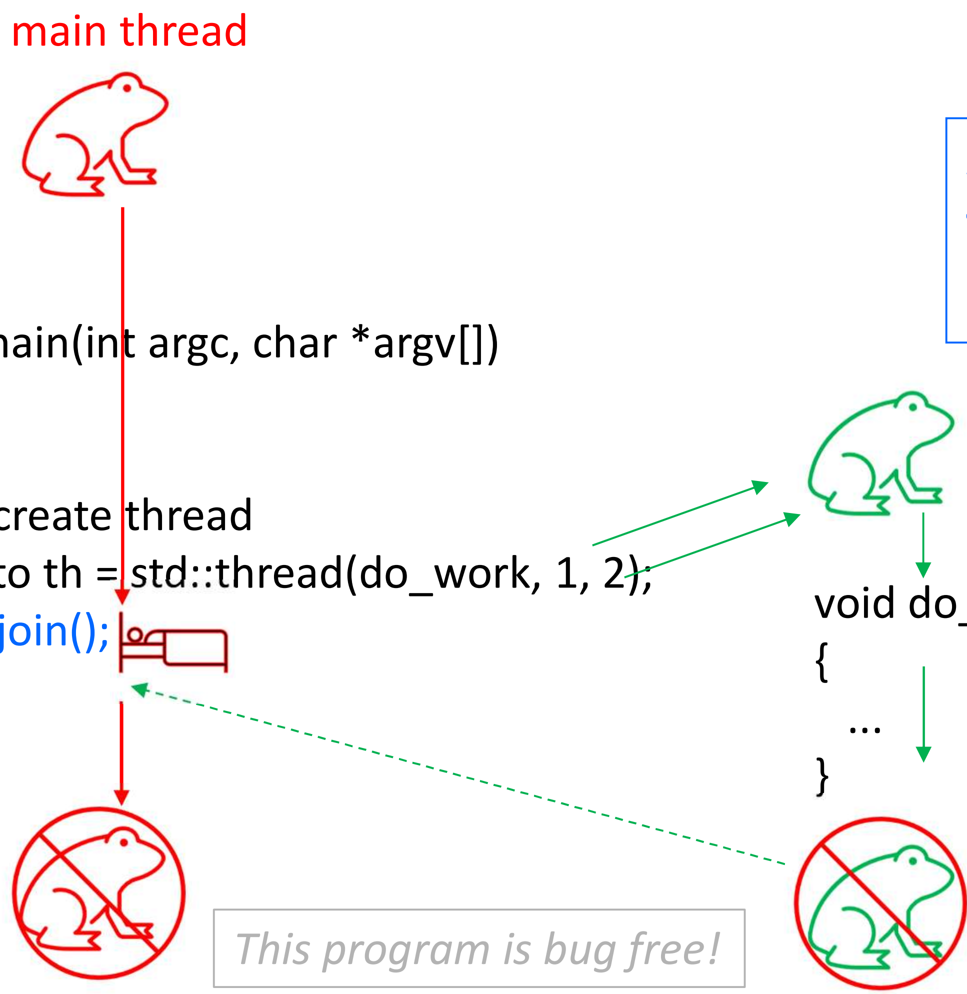
Creating Threads



Join-ing Threads

Join: One thread waits for another to terminate

A program **must not terminate** while there are any joinable threads




```
int main(int argc, char *argv[])
{
  ...
  // create thread
  auto th = std::thread(do_work, 1, 2);
  th.join();
  ...
}
```

```
void do_work(int a, int b)
{
   ...
}
```

Detach-ing Threads

Detach: Indicate that join() will never be called on this thread

main thread

A detached thread is not joinable.

```
int main(int argc, char *argv[])
{
  ...
  // create thread
  std::thread(do_work, 1, 2).detach();
  ...
}
```

```
void do_work(int a, int b)
{
  ...
}
```

This program is bug free!

Performance and Threads

- It is tempting to think of threads as a mechanism for parallel execution
  - Parallel: the goal is to obtain a result quicker
  - Sometimes threads simplify program structure: concurrency
- Because a single thread can use only a single core, to use more than one core at a time there must be threads
  - That doesn't necessarily mean your code has to manage them...
- The relationship between number of threads and performance is complicated
  - more threads => more potential parallelism
  - more parallelism => more contention
  - more threads => more thread management overhead

Example Parallel Code

```
std::array<int,100000000> global_array;

void init_array (decltype(global_array.size()) start, decltype(global_array.size()) end) {
  if ( end > global_array.size() )  end = global_array.size();

  std::cout << std::this_thread::get_id() << ": "  << start << " -- " << end << std::endl;
  for (decltype(start) i=start; i<end; ++i)
    global_array[i] = i;
}

int main (int argc, char *argv[])
{
  if ( argc != 2 ) usage(argv[0]);

  auto N_threads = atol(argv[1]);
  if ( N_threads <= 0 ) usage(argv[0]);
  auto stride = (global_array.size() + N_threads - 1) / N_threads;
  decltype(stride) start = 0;

  std::vector<std::thread> threads;
  for (int i=0; i<N_threads; ++i) {
    threads.push_back(std::thread(init_array, start, start+stride));
    start += stride;
  }

  std::cout << "synchronizing all threads...\n";
  for (auto& th : threads) th.join();

  return 0;
}
```

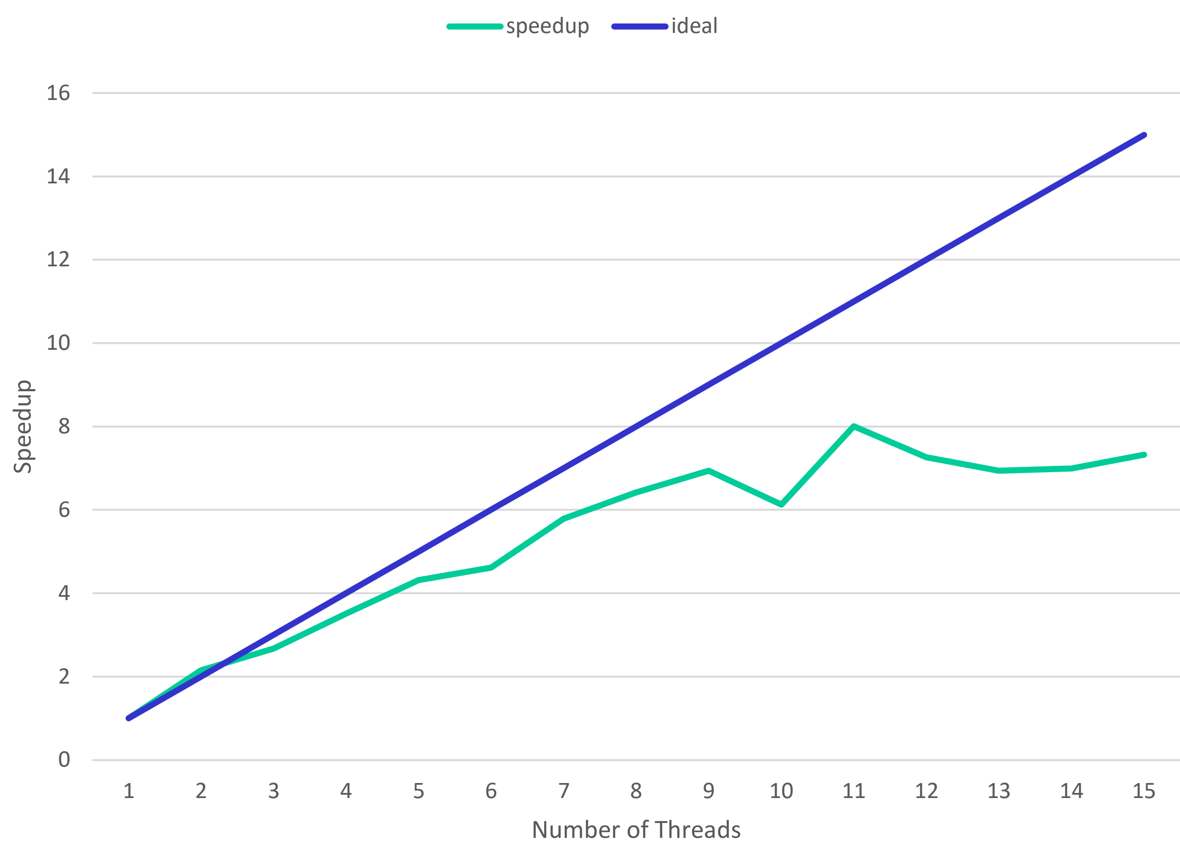
Initialize an array of 100,000,000 ints

Speedup

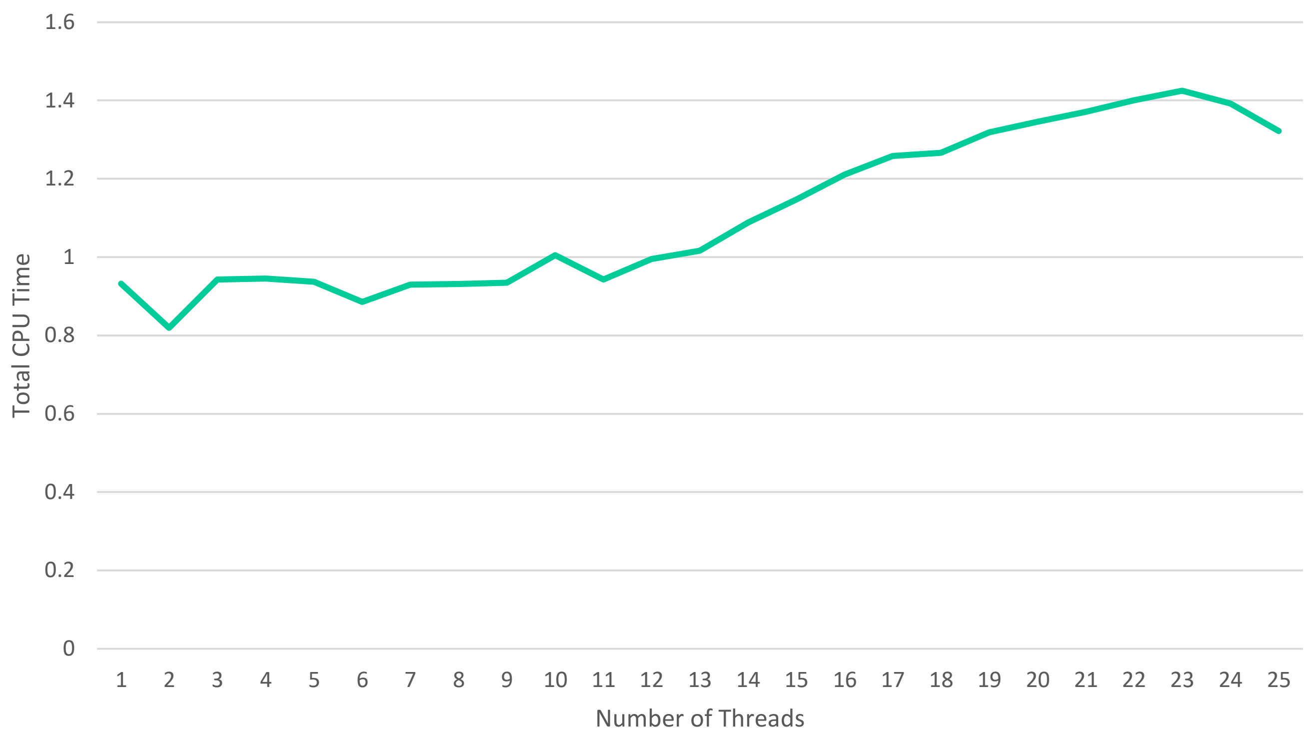
- S(n) = T(1) / T(n), where
  - S(n) is the speedup using n threads
  - T(k) is the elapsed time required to complete using k threads
- Ideal S(n) == n
  - S(n) is normally less than n
    - Sometimes much less...
  - It's not impossible for it to be greater than n

Measured Speedup on attu4

- Xeon E5-2670 v3
- 12 cores, 24 threads



Example Parallel Code Total CPU Time



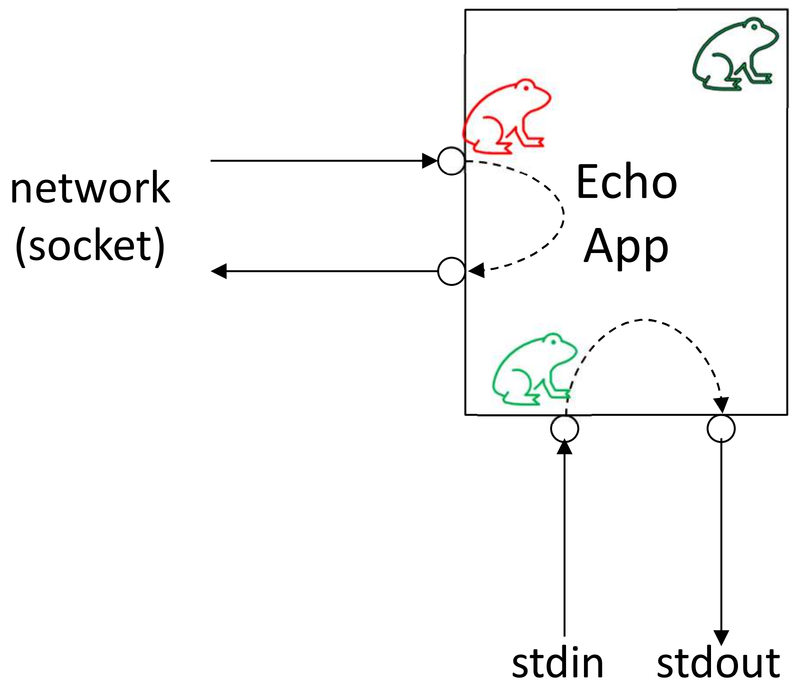
Threads for Concurrency

- Sometimes the code needs to do a number of largely separate tasks, each of which is nice represented as a single thread of control

- In some of these cases, threads need to make blocking system calls
  - If there's only one thread, the application is completely inert when that one thread blocks on a system call (e.g., read)

- Threads for concurrency are about making it easier to write the program

Example Concurrent Application



- Main thread creates two threads
- One sits in a loop accepting connections
  - it then sits in a loop reading from connection and writing back to connection
- The other sits in a loop reading from stdin and writing to stdout
- The main thread join's the two threads it has created, and then exits

Main thread code

```
bool done = false;

int main(int argc, char const *argv[])
{
  int server_fd;
  try
    {
      server_fd = make_server_socket(PORT);
    }
  catch (std::exception &e)
    {
      std::cout << e.what() << std::endl;
      exit (1);
    }

  std::thread network_thread(accept_connection, server_fd);
  std::thread keyboard_thread(read_keyboard);

  network_thread.join();
  close(server_fd);

  keyboard_thread.join();

  return 0;
}
```

Keyboard (stdin/stdout) thread code

```
void read_keyboard()
{
  char buffer[1024];
  while ( fgets(buffer, 1023, stdin) ) {
    printf("stdin: %s", buffer);
    if ( !strcmp(buffer, "q\n") )
        break;
  }
  done = true;
  std::cout << "--- Keyboard thread exiting" << std::endl;
}
```

EOF from keyboard can shut down the app.
(The app cannot be shut down from its network connection.)

Network thread code (Part 1)

```
void accept_connection(int server_fd)
{
  int new_socket, addrlen, valread;
  struct sockaddr_in address;
  char buffer[1024] = {0};

  struct timeval timeout;
  timeout.tv_sec = 5;
  timeout.tv_usec = 0;

  if (setsockopt (server_fd, SOL_SOCKET, SO_RCVTIMEO, (char *)&timeout,
                 sizeof(timeout)) < 0)
   {
    perror("setsockopt failed");
    return;
   }
```

Can't kill this thread from another thread, so it has to wake up from waiting for a connection every so often. (Every 5 seconds here.)

Network thread code (Part 2)

```
while(!done)
  {
    new_socket = accept(server_fd, (struct sockaddr *)&address,(socklen_t*)&addrlen);
    if ( new_socket < 0 )
     {
       if ( errno == EAGAIN || errno == EWOULDBLOCK)
        {
          std::cout << "Network thread continuing..." << std::endl;
          continue;
        }
       std::cout << std::strerror(errno) << std::endl;
       perror("accept failed");
       return;
     }
    std::cout << "--- Have network connection" << std::endl;
    while ( (valread = read( new_socket , buffer, 1024)) > 0 )
     {
       send(new_socket , buffer, valread , 0 );
       printf("network: %s", buffer);
     }
    close(new_socket);
    std::cout << "--- Network connection closed" << std::endl;
  }
 std::cout << "--- Network thread exiting" << std::endl;
}
```

Wake up regularly to check done flag.

This code blocks indefinitely if connected and the other end doesn't send anything. Maybe...

Threads for Concurrency Summary: Pro's

- We used many threads for concurrency because it simplified the programming model
  - Each thread represented a largely independent computation
  - The state of the computation (thread) was reflected "in the usual way" – in the call stack of the thread
  - The computations involved high latency operations
  - We addressed the high latency operation using blocking calls
    - Rather than "polling"
  - Overall efficiency is good because one thread blocking doesn't interfere with the progress of other thread
  - Have the possibility for physical parallelism (use more than one core)

Threads for Concurrency: Con's

- When the per-thread computations aren't so independent, probably have race conditions that must be addressed
  - We'll look at this a bit in a bit
  - For now, read this as "when computations aren't entirely independent, we need synchronization, and that is a new level of complexity and difficult bugs"
- It's hard to know how many threads to use
  - Too many results in high thread management overhead
  - Too few results in insufficient concurrency and resultant delays

An Alternative: Event-Driven Execution

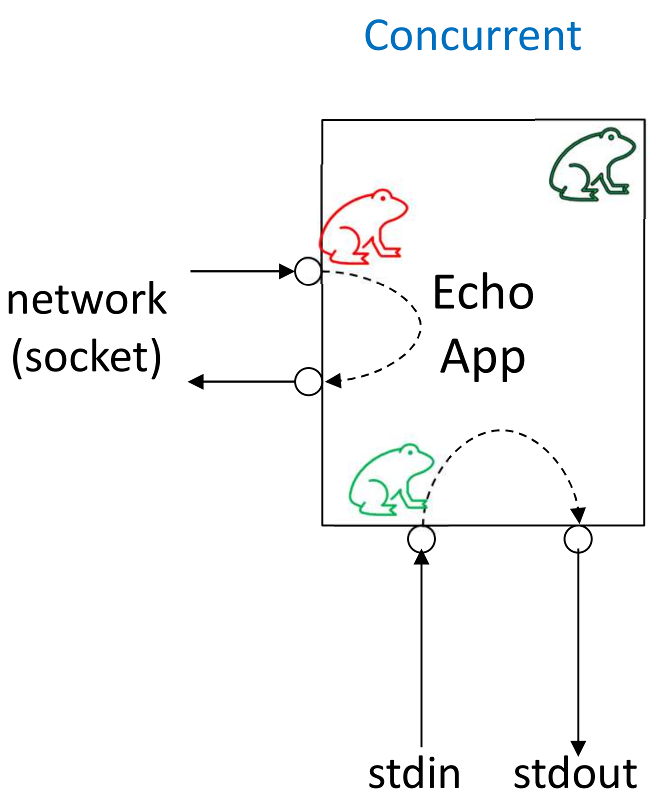
- It isn't inherent in the idea, but typically this implies using only a single thread
  - Minimal thread management overhead
  - No race conditions
- Instead of one thread per blocking call (e.g., socket or file read), a single thread waits for any of them to become available
- Having available data is one example of "an event"
  - Events can be logical/software induced – Java Observer/Observable
- A handler routine is called when an event happens
- Program execution is a succession of events firing (asynchronously) and event handlers being invoked

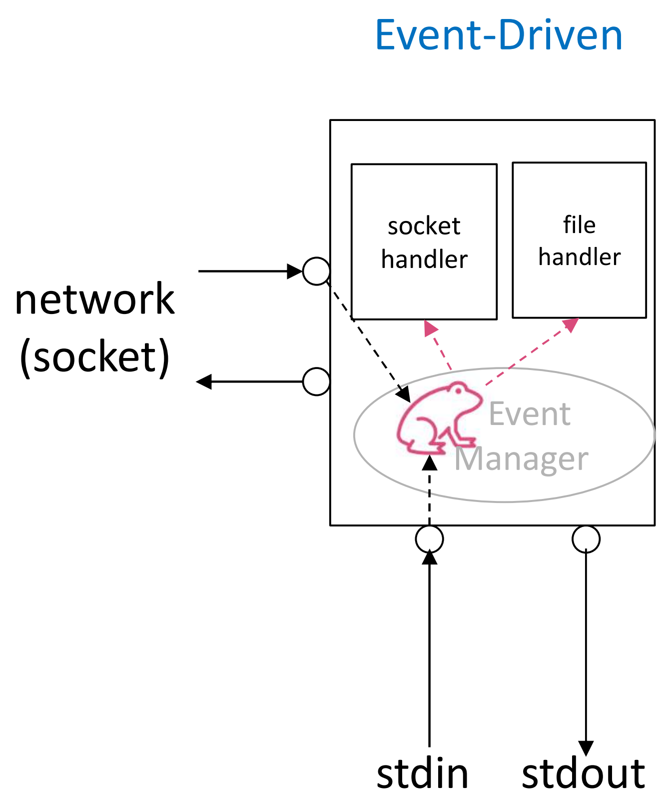
An Alternative: Event-Driven Execution

- Often the code that knows how to accept event handler registrations, wait for an event, and invoke the appropriate handler is infrastructure
  - E.g., Windows message loop, Java Observer/Observable, any number of language runtimes
- The application is (largely) composed of a set of handlers

- C++ does not have a generally accepted event infrastructure
- In the examples I'll show you, I've built a crude one as part of the app

Concurrent vs. Event-Driven





Event-Driven App Code: main()

```
/* Single threaded, event-driven implementation
  of multiple input stream example echo app */
int main(int argc, char const *argv[])
{
  try
   {
    listener.RegisterStream(fileno(stdin), read_file );

    int server_fd = make_server_socket(PORT);
    listener.RegisterStream(server_fd, accept_connection);

    listener.run();
   }
  catch (std::exception &e)
   {
    std::cout << e.what() << std::endl;
    exit (1);
   }
  return 0;
}
```

"listener" is the event infrastructure object

When stdin has input, call read_file()

When client connection arrives, call accept_connection()

Okay, go into event loop (and don't return until it's time to terminate execution)

Event-Driven App Code: file handler

```
/* Callback for reading from a file */
void read_file(int fd)
{
  char buffer[1024] = {0};
  /* Get FILE* from fd to use with fgets() */
  FILE* in_file = fdopen(fd, "r");
  fgets(buffer, 1023, in_file );
  std::cout << "From file stream: " << buffer;
  if ( !strcmp(buffer, "q\n") )
    listener.done();
}
```

Called by listener whenever user types something (annotates `void read_file(int fd)`)

Essential stdin functionality of app (with a bug) (annotates the `fgets` and `std::cout` lines)

Program exit logic. Will cause listener::run() to return. (annotates the `if ( !strcmp(buffer, "q\n") )` / `listener.done();` lines)

Event-Driven App Code: socket handler

```
/* Callback for listener socket */
void accept_connection(int server_fd)
{
  int new_socket, addrlen;
  struct sockaddr_in address;

  new_socket = accept(server_fd, (struct sockaddr *)&address,(socklen_t*)&addrlen);
  if ( new_socket < 0 )
    throw std::runtime_error("accept failed");

  listener.RegisterStream(new_socket, read_socket);
}
```

Event-Driven App Code: client socket handler

```
/* Callback for reading from a connected socket */
void read_socket(int client_socket)
{
 char buffer[1024] = {0};

 int n_read = recv( client_socket , buffer, 1023, MSG_DONTWAIT);
 if ( n_read > 0 )
  {
   send(client_socket , buffer, n_read , 0 );
   buffer[n_read] = '\0';
   printf("From network stream: %s", buffer);
  }
 else if ( n_read == 0 )
  {
   listener.UnregisterStream(client_socket);
  }
 else if (/* n_read < 0 */ errno != EWOULDBLOCK )
  {
    throw std::runtime_error("Socket not ready for recv?");
  }
}
```

App Code Summary

- The app is basically a set of event handlers
  - There is a setup phase that registers the handlers
  - Then the app sits in the event handler infrastructure calling handlers as events happen
  - Works beautifully when handling an event is independent of everything else that has or will happen...

- Reminder: single threaded execution, so no race conditions

Infrastructure Implementation

- select() is a somewhat deprecated call whose input is a list of file descriptors
  - select() blocks until any one (or more) of the file descriptors indicates it "is ready"
    - has input to read, and/or is capable of accepting new output to write
  - select() returns an indication of which file descriptors are ready
    - plus it can do more, so look at the man page if you want to know more

- The modern version is poll()

- Despite that, you're hear the term "select loop" – that's the heart of the event handler infrastructure

Infrastructure Implementation

Some details have been left out to fit this on the slide.

```
typedef std::function<void(int)> SLCallback;
class StreamListener {
public:
  /* n_fds is the maximum number of file descriptors the listener is configured to monitor */
  StreamListener(unsigned int n_fds = 10) : max_fds_(n_fds) {...}
  ~StreamListener();

  bool RegisterStream( int fd, SLCallback event_callback, SLCallback destroy_callback=CloseFd );
  bool UnregisterStream( int fd );

  void run();
  /* tells run to return */
  void done() { ... }

private:
  static void CloseFd(int fd) { close(fd); }

  std::map<int, std::pair<SLCallback, SLCallback>> listener;
  unsigned int max_fds_;
};
```

Infrastructure Implementation

```
StreamListener::~StreamListener()
{
  if ( poll_fds_ )
    delete [] poll_fds_;

  /* Invoke destroy callback on registered streams */
  for (auto&& [first,second] : listener)
      second.second(first);
}
```

Free dynamically allocated memory (annotation pointing to `delete [] poll_fds_;`)

Invoke file descriptor destruction method callback (annotation pointing to `second.second(first);`)

Infrastructure Implementation

```
void StreamListener::run() {
 while(!done_) {
   /* Convert map to array structure needed by poll() */
   RepopulatePollFds();
   /* Wait for something to happen... */
   int rc = poll(poll_fds_, n_fds_, -1);
   if ( rc < 0 )  throw std::runtime_error("poll failed");

   /* Figure out what happened */
   for ( unsigned int i=0; i<n_fds_; i++ )  {
     if ( poll_fds_[i].revents == 0 )  continue;
     if ( poll_fds_[i].revents == POLLIN )   {
       auto it = listener.find( poll_fds_[i].fd );
       if ( it == listener.end() )
        throw std::runtime_error("Got poll event on file descriptor that isn't registered");
       it->second.first(it->first);
      }

     else if ( poll_fds_[i].revents == POLLNVAL )  UnregisterStream(poll_fds_[i].fd);

     else std::cout << "Bad revents: " << poll_fds_[i].revents << " on " << poll_fds_[i].fd << std::endl;
    }
  }
}
```

Wait for some source to become ready

Find the ready descriptor(s) and event type

Accessing the Example Code

- attu:/cse/courses/cse333/21wi/public/concurrency/
- attu:/cse/courses/cse333/21wi/public/event-driven/

- *Note: there are known bugs having to do with robustness and error detection/resolution*

- Make sure to include the pthread library in the build:

  g++ -std=c++17 –g –Wall *.cc –l pthread

Bonus Topic: The Problem, In Real Life

(approximately)

- You order dinner delivered to your front door
- How do you know when it arrives?
- You can
  - Stand at the front door and wait
  - Do something else, but go to the door every once in a while and check
    - If it's not there you can go back inside, or
    - If it's not there you can just wait because you have nothing better to do
  - Arrange for the delivery person to text you when your dinner arrives
  - Train your dog to wait at the door for your dinner (but now you're waiting for the dog, so you have the same problem)
    - *Note: If your dog could eat your dinner for you that would solve the waiting problem*

Key: synchronous single-threaded | asynchronous | multi-threaded

Long Latency Operations

- When your code calls read(), it stops executing until something has been read (or an error has occurred or EOF has occurred or a signal is received or...)
- Why?

- It can be useful to think of long-latency operations as having two distinct sub-operations
  - start
  - done
- Why?

Long Latency Operation Completion Detection

- How can the originator of the operation know when it has completed?

- Depends on how execution is done
  - Synchronous execution - the thread originating the operation doesn't run again until the operation finishes
  - Asynchronous execution – the thread originating the operation continues running

- Depends on how notification is done
  - Synchronous notification – the initiating thread takes some action to check whether the operation has completed
  - Asynchronous notification – a method is registered to be run when completion occurs, and then is run when completion occurs
  - No notification

Procedure Call Semantics: Sync / Sync

- Synchronous Execution / Synchronous Notification
  - Example: procedure call
    - calling thread carries out the long latency work (procedure execution)
  - Example: (blocking) read()
    - operating system suspends execution of calling thread until data is available to be read
  - Continuing execution == operation has finished

- This is the simplest model for programmers
- "Remote Procedure Call" (RPC) is a(ny) network protocol whose semantics are those of local procedure call

Polling: Async / Sync

- A invoking thread starts an operation and then goes on executing without waiting for operation to complete
- The operation sets some state (e.g., a variable) to indicate when it has completed
- The invoking thread checks the state variable whenever it feels like
  - Could be in a tight "polling loop" (doing nothing but checking)
  - Could check "every once in a while" (every 5 msec., every 10 sec., once per day, ...)

- Polling mostly make sense for operations whose latency (time to completion) is predictable

Polling: Example

- First of all, you should feel very uneasy if you find yourself writing code that does polling!
  - In most circumstances, there's some better (more efficient/simpler) solution

- Example: sockets
  - you can set a network socket to be "non-blocking"
  - When you perform a read() operation on it, you get an answer back immediately
    - The answer might be the data you wanted
    - Otherwise the answer is an error (EWOULDBLOCK)
    - Either way, your thread continues running and can do whatever you want

Join: Async / Sync

- Threads:
  - create a thread (as a C++ std::thread object, say, th), which causes it to start running
  - th.join() suspends the calling thread until thread th terminates

- Processes
  - fork() a process. You get back the new (child) process's process id (pid)
  - wait(pid) to wait for it to terminate
  - The difference between
    - emacs myfile.txt
    - emacs myfile.txt &

Async / synch

- When you start some operation asynchronously, there will almost aways be two things you can do to check on its completion status
  - "wait" (or some other name): suspend my execution until the operation has finished
  - "test" (or some other name): return an indication of whether or not it has finished, but don't block me no matter what

Example using C++ futures/async

- Synch/Synch
  - procedure call
- Asynch / Synch
  - wait
  - poll

- Plus bonus features (and C++ qualitative review)

Example App

```
int main(int argc, char *argv[])
{
   <start delay_sub(args);
   <do no work or do some work>
    <obtain result from delay_sub()>
}
```

```
int delay_sub(args)
{
   <do some work that
     takes a while>
    return value;
}
```

Execution Scenarios:

- procedure call
- async / sync where main waits
- async / sync where result is ready when main asks for it
- polling

C++ features

- std::async, std::future
- std::this_thread
- time – std::chrono
- function object
- method chaining
- friend function

A Design Issue

- I want to print log messages indicating what each "thread" is doing
- I want to print elapsed time with each message
- I want syntax something like this:

  `LOG() << "Main thread start operation(0, 1, 2)" << std::endl;`

  to produce output like this:

  `0.0000410800 -- Main thread start operation(0, 1, 2)`

IntervalTimer Utility Class

```
class IntervalTimer
{
 public:
  IntervalTimer() { reset(); }
  IntervalTimer& reset()
    {
      start_ = std::chrono::steady_clock::now();
      return *this;
    }
 private:
  std::chrono::time_point<std::chrono::steady_clock> start_;
  friend std::ostream& operator<<(std::ostream&, IntervalTimer&);
};

std::ostream& operator<<(std::ostream& os, IntervalTimer &timer)
{
  std::chrono::duration<float> elapsed_time = std::chrono::steady_clock::now() - timer.start_;
  os << elapsed_time.count();
  return os;
}
```

Logger Utility Class

```
class Logger
{
 public:
 Logger(std::ostream& os) : os_(os) {}
  std::ostream& operator()()
   {
     os_ << std::fixed << std::setprecision(10) << timer_ << " -- ";
     return os_;
   }
  Logger& reset()
   {
     timer_.reset();
     return *this;
   }
  std::ostream& ostream()
   {
     return os_;
   }
 private:
  std::ostream& os_;
  IntervalTimer timer_;
};
```

delay_sub()

```
int delay_sub(int x, int y, int z)
{
  LOG() << "delay_sub thread (" << std::this_thread::get_id() << ") sleeping for 5 seconds" << std::endl;
  std::this_thread::sleep_for(std::chrono::seconds(5)); // never do this!
  LOG() << "delay_sub thread awake" << std::endl;
  if ( x+y+z < 0 )
     throw std::runtime_error("Result is negative!");
  return x+y+z;
}
```

Main: procedure call

```
LOG.reset()() << "Procedure call test" << std::endl;
LOG() << "Main thread (" << std::this_thread::get_id() << ") start operation(0, 1, 2)" << std::endl;
wait_val = delay_sub(0, 0, 0);
LOG() << "Main thread got value: " << wait_val << std::endl;
```

```
0.0000045240 -- Procedure call test
0.0000432160 -- Main thread (140487854065472) start operation(0, 1, 2)
0.0000488950 -- delay_sub thread (140487854065472) sleeping for 5 seconds
5.0001621246 -- delay_sub thread awake
5.0001931190 -- Main thread got value: 0
```

Main: async / synch (wait)

```
LOG.ostream() << std::endl;
LOG.reset()() << "First wait test" << std::endl;
LOG() << "Main thread (" << std::this_thread::get_id() << ") start operation(0, 1, 2)" << std::endl;
std::future<int> v1 = std::async(&delay_sub, 0, 1, 2);
LOG() << "Main thread sleeping for 2 seconds" << std::endl;
std::this_thread::sleep_for(std::chrono::seconds(2));
LOG() << "Main thread get()" << std::endl;
wait_val = v1.get();
LOG() << "Main thread got value: " << wait_val << std::endl;
```

```
0.0000001100 -- First wait test
0.0000050920 -- Main thread (140487854065472) start operation(0, 1, 2)
0.0002516000 -- Main thread sleeping for 2 seconds
0.0002674270 -- delay_sub thread (140487836149504) sleeping for 5 seconds
2.0003676414 -- Main thread get()
5.0004024506 -- delay_sub thread awake
5.0005426407 -- Main thread got value: 3
```

Main: async / sync (wait) Part 2

```
LOG.ostream() << std::endl;
 LOG.reset()() << "Second wait test" << std::endl;
 LOG() << "Main thread (" << std::this_thread::get_id() << ") start operation(3, 4, 5)" << std::endl;
 auto v2 = std::async(delay_sub, 3, 4, 5);  // this is an easier way to declare the std::future
 LOG() << "Main thread sleeping for 9 seconds" << std::endl;
 std::this_thread::sleep_for(std::chrono::seconds(9));
 LOG() << "Main thread get()" << std::endl;
 wait_val = v2.get();
 LOG() << "Main thread got value: " << wait_val << std::endl;
```

```
0.0000001000 -- Second wait test
0.0000039960 -- Main thread (140487854065472) start operation(3, 4, 5)
0.0000397140 -- Main thread sleeping for 9 seconds
0.0000452840 -- delay_sub thread (140487836149504) sleeping for 5 seconds
5.0001139641 -- delay_sub thread awake
9.0001583099 -- Main thread get()
9.0001926422 -- Main thread got value: 12
```

Main: async / sync (polling)

```
LOG.ostream() << std:: endl;
LOG.reset()() << "Polling test" << std::endl;
LOG() << "Main thread (" << std::this_thread::get_id() << ") start operation(6, 7, 8)" << std::endl;
auto v3 = std::async(delay_sub, 6, 7, 8);
while(1) {
  auto status = v3.wait_for(std::chrono::seconds(0));
  if ( status == std::future_status::ready)
   break;
  LOG() << "Main thread sleeping for four seconds" << std::endl;
  std::this_thread::sleep_for(std::chrono::seconds(4));
 }
wait_val = v3.get();
LOG() << "Main thread got value: " << wait_val << std::endl;
```

```
0.0000000800 -- Polling test
0.0000031500 -- Main thread (140487854065472) start operation(6, 7, 8)
0.0000432180 -- Main thread sleeping for four seconds
0.0000984170 -- delay_sub thread (140487836149504) sleeping for 5 seconds
4.0001163483 -- Main thread sleeping for four seconds
5.0001783371 -- delay_sub thread awake
8.0002136230 -- Main thread got value: 21
```

C++ bonus material: delayed exception

```
LOG.ostream() << std::endl;
 LOG.reset()() << "Exception test" << std::endl;
 try {
   LOG() << "Main thread (" << std::this_thread::get_id() << ") start operations(-1, -2, -3)" << std::endl;
   v3 = std::async(delay_sub, -1, -2, -3);
   LOG() << "Main thread sleeping for eight seconds" << std::endl;
   std::this_thread::sleep_for(std::chrono::seconds(8));
   int result = v3.get();
   LOG() << "Exception test got result " << result << std::endl;
  }
 catch (std::exception &e) {
   LOG() << "Exception: " << e.what() << std::endl;
  }
```

```
0.0000000800 -- Exception test
0.0000022680 -- Main thread (140487854065472) start operations(-1, -2, -3)
0.0000613190 -- Main thread sleeping for eight seconds
0.0000692290 -- delay_sub thread (140487836149504) sleeping for 5 seconds
5.0001440048 -- delay_sub thread awake
8.0001573563 -- Exception: Result is negative!
```

Computing Bonus Material: Signals

- What about async execution/ async notification?
  - What does it even mean?
    - Event-based programming (sort of)
- Signals
  - Process-level event handlers
  - The "events" are integers, most of which have well-known semantics
    - For instance, ctrl-C is a signal (SIGINT == 2)
- A process registers a signal handler method for a signal
- When the signal is sent/received, that method is invoked
- The signals I'll show allow one process to signal another process

signal.cc

```
int counter = 0;

void signal_handler(int signal)
{
  std::cout << std::endl
          << "Thread " << std::this_thread::get_id()
          << " caught signal: " << signal << std::endl
          << "counter = " << counter << std::endl;
}

int main()
{
  std::cout << "Process id: " << getpid() << std::endl;

  // Install a signal handler
  std::cout << "Installing handler for " << SIGUSR1 << std::endl;
  std::signal(SIGUSR1, signal_handler);

  std::cout << "Thread " << std::this_thread::get_id() << " going into infinite loop." << std::endl;
  for (unsigned int i=0; i>=0; i++) { counter++; }

  return 0;
}
```

Register handler method for SIGUSR1 signal

Example Execution

One Shell

```
attu8> ./a.out
Process id: 801807
Installing handler for 10
Thread 1 going into infinite loop.

Thread 1 caught signal: 10
counter = 1054351402

Thread 1 caught signal: 10
counter = -592577485

Thread 1 caught signal: 10
counter = 155031883

attu8>
```

Another Shell

```
attu8> kill -SIGUSR1 801807
attu8> kill -SIGUSR1 801807
attu8> kill -SIGUSR1 801807
attu8> kill -SIGINT 801807
```

Accessing the Example Code

- attu:/cse/courses/cse333/21wi/public/concurrency/
- attu:/cse/courses/cse333/21wi/public/event-driven/
- attu:/cse/courses/cse333/21wi/public/async/
- attu:/cse/courses/cse333/21wi/public/signal/

- *Note: there are known bugs having to do with robustness and error detection/resolution*

- Make sure to include the pthread library in the build:

  g++ -std=c++17 –g –Wall *.cc –l pthread