Parallel Computing

Ekkapot Charoenwanit

Software Systems Engineering
TGGS
KMUTNB

Lecture 5:

- Design Models for Multithreaded Program
- Additional Features of the Pthreads Library

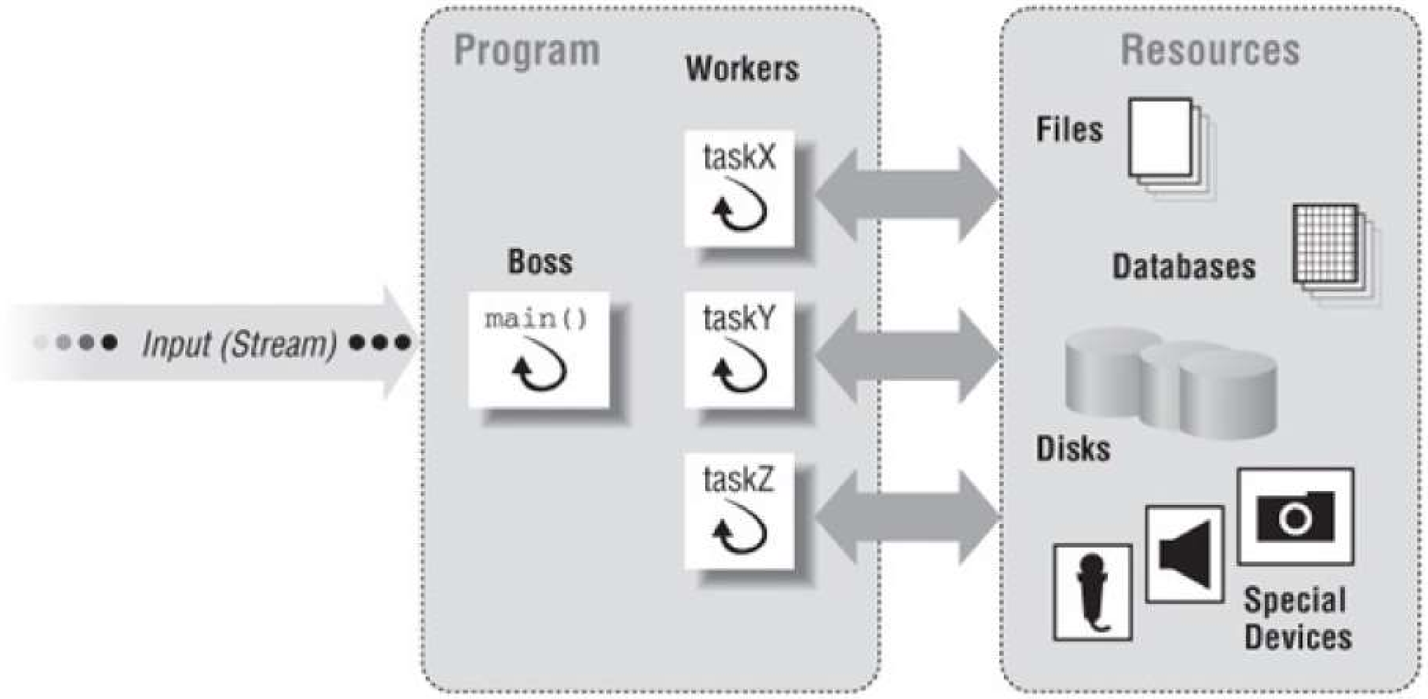
Threading Models

There is no set rule for threading a program.

However, the following three models are ones that are commonly used to thread programs:

- ***The Boss-Worker Model***
- ***The Peer Model***
- ***The Pipeline model***

The Boss-Worker Model



The Boss-Worker Model

A single thread, ***the boss***, accepts input for the entire program.

- Based on that input, the boss passes off specific tasks to one or more worker threads.
- The boss creates each worker thread, assigns it tasks, and, if necessary, waits for it to finish.

In the pseudo code on the next slide, the boss ***dynamically*** creates a new worker thread when it receives a new request.

- In the $pthread\_create()$ call, the boss specifies task-specific routine the newly created worker thread will execute.
- After creating each worker, the boss returns to the top of its loop to process the next request.
- If no requests are pending, the boss loops until one arrives.
- Once finished, each worker can be made responsible for any output resulting from its task, or it can synchronize with the boss and let it handle the output.

The Boss-Worker Model

```
main()

/* The boss */
{
   forever {
             get a request
             switch request
             case X : pthread_create( ... taskX)
             case Y : pthread_create( ... taskY)
              .
              .
              .
    }
 }
 taskX() /* Workers processing requests of type X */
 {
    perform the task, synchronize as needed if accessing shared resources
    done
 }

 taskY() /* Workers processing requests of type Y */
 {
    perform the task, synchronize as needed if accessing shared resources
    done
 }
 .
 .
 .
```

The Boss-Worker Model: Thread Pool

Rather than dynamically create worker threads, the boss can alternatively save some ***runtime overhead*** by creating all worker threads upfront.

- This variant of the Boss-Worker model is known as, ***a thread pool***.
- The boss creates all the worker threads at program initialization.
- After being created, each worker immediately suspends itself to a ***wake-up call*** from the boss when a request arrives for it to process.
- The boss advertises work by placing requests on a shared buffer from which the workers can retrieve.

The Boss-Worker Model: Thread Pool

```
main()

/* The boss */
{
 for the number of workers
         pthread_create( ... pool_base )

 forever {
         get a request
         place request in work queue
         signal sleeping threads that work is available
 }
}
pool_base() /* All workers */
{
 forever {
         sleep until awoken by boss
         dequeue a work request
         switch
           case request X: taskX()
           case request Y: taskY()
                   .
                   .
                   .
 }
}
```

The Boss-Worker Model

The boss-worker model works well with ***servers*** e.g. database servers.

- The complexity of dealing with asynchronously arriving requests and communications are encapsulated within the boss.
- The specifics of handling requests are delegated to the workers.

In this model, it is important that you ***minimize the frequency*** with which the boss and the workers communicate.

- The boss cannot afford to be frequently blocked by its workers and allow new requests to pile up at the inputs.
- Similarly, interdependencies among the workers must be avoided as much as possible.
- If every request requires every worker to synchronize because they share some data, all workers will experience a slow-down.

The Peer Model

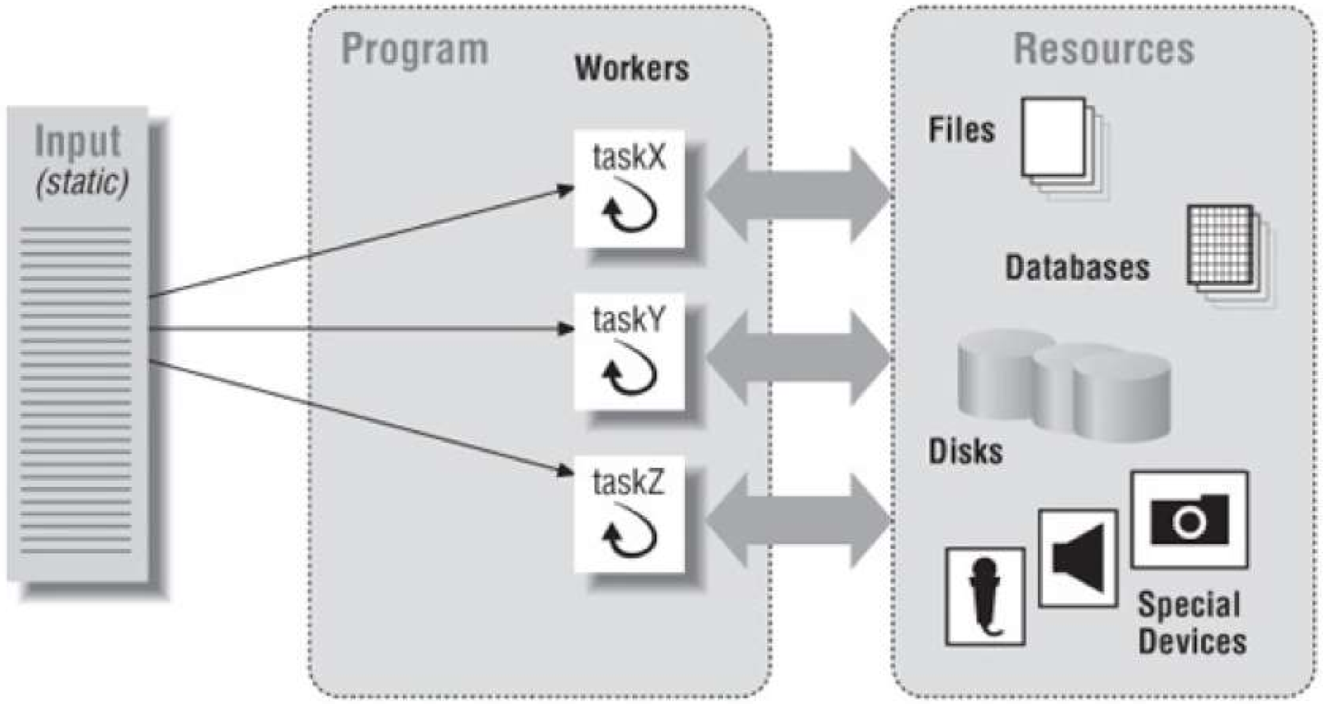
In ***the peer model***, all the threads work concurrently on their tasks ***without a specific leader***.

- thread must create all the other threads when the program starts.
- However, unlike the boss-worker model, this thread acts as just another peer thread that processes requests or suspends itself to wait for the other threads to finish.

Whereas the boss-worker model employs a stream of input requests to the boss, the peer model makes each thread responsible for its own input.

- Each peer knows its own input ahead of time.
- It has its own private way to obtain its input.

The Peer Model



The Peer Model

```
main()
{
   pthread_create( ... thread1 ... task1 )
   pthread_create( ... thread2 ... task2 )
   .
   .
   .
   signal all workers to start
   wait for all workers to finish
   do any clean up
}

task1()
{
   wait for start
   perform task, synchronize as needed if accessing shared resources
   done
}

task2()
{
   wait for start
   perform task, synchronize as needed if accessing shared resources
   done
}
```

The Peer Model

The peer model is suitable for applications that have a ***fixed*** or ***well-defined*** set of inputs such as matrix multiplication, database search engines, and prime number generators.

- Because there is no boss, peers must synchronize their access to any common sources of input.
- However, like the boss-worker model, peers can experience a slow-down if they must frequently synchronize to access shared resources.
- The parallel program that estimates the value of π using Monte Carlo from the last lecture is also based on the peer model.

The Pipeline Model

The pipeline model assumes:

- a long stream of input
- a series of suboperations (know as ***stages***) through which every unit of input must be processed
- each stage can handle a different unit of input at a time.

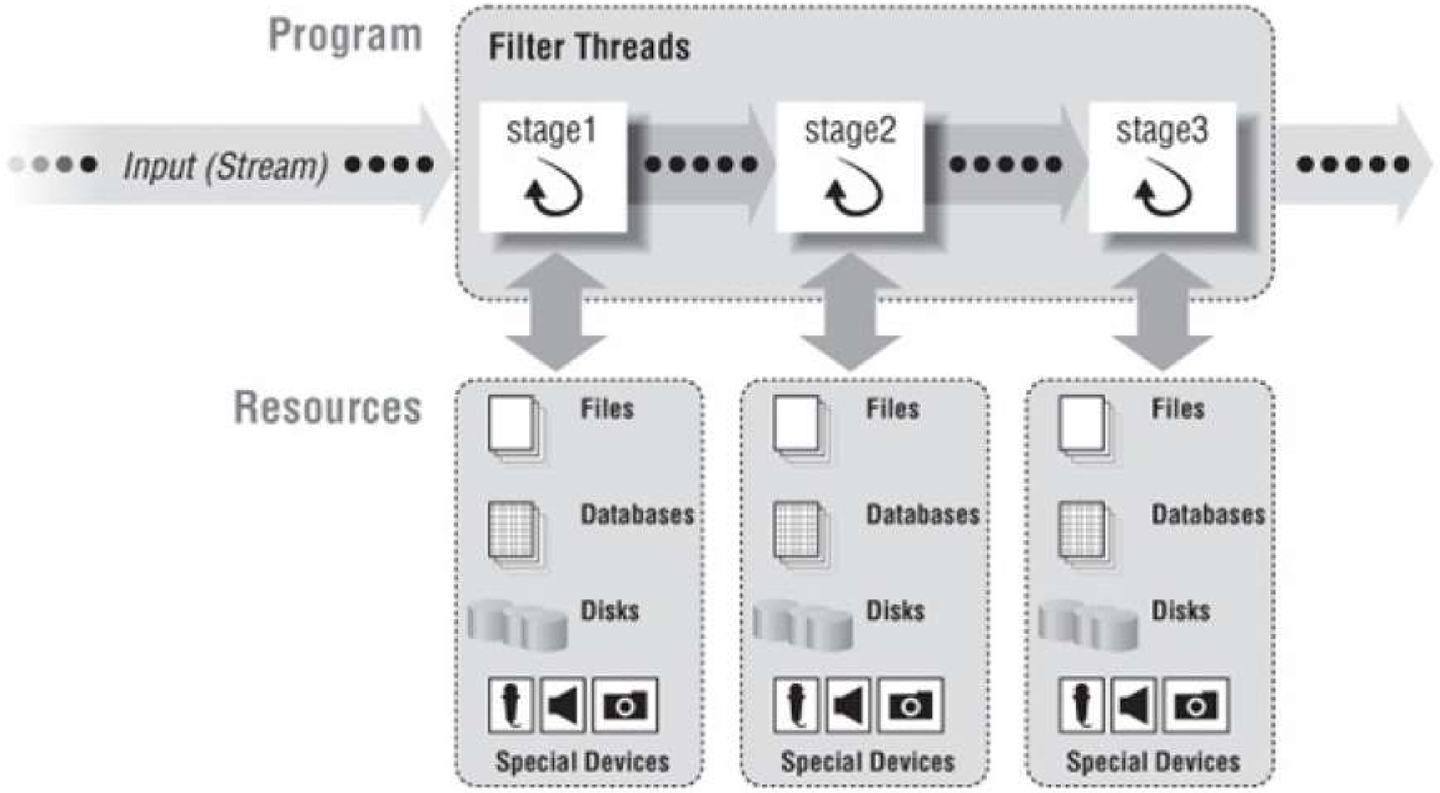
An automotive assembly line is a good example of a pipeline.

- Each car goes through a series of stages on its way to the exit gate.
- At any given time, a number of cars are processed at those different stages.

A pipeline improves ***throughput*** because it can accomplish the many different stages of a process on different units of input concurrently.

- In stead of taking each car from start to finish before starting the next one, a pipeline allows as many cars to be simultaneously worked on as there are stages to process them.
- Note that it still takes the same amount of time from start to finish for a specific car to be processed, but the overall throughput is greatly increased.

The Pipeline Model



The Pipeline Model

As the pseudocode illustrates,

- a single thread receives input for the entire program, always passing it to the thread that handles the ***first stage*** of processing.
- Similarly, a single thread at the end of the pipeline produces all final output of the program.
- Each thread in between performs its own stage of processing on the input it receives from the thread that performed the ***previous stage***, and passes its output to the thread performing the ***next stage***.

Applications in which the pipeline model might be useful are image processing and text processing or any application that can be broken down into a series of filter steps on a stream of input.

```
main()
{
    pthread_create( ... stage1 )
    pthread_create( ... stage2 )
    .
    .
    .
    wait for all pipeline threads to finish
    do any clean up
}

stage1()
{
   forever {
           get next input for the program
           do stage 1 processing of the input
           pass result to next thread in pipeline
           }
}

stage2()
{
   forever {
           get input from previous thread in pipeline
           do stage 2 processing of the input
           pass result to next thread in pipeline
           }
}
.
.
.
```

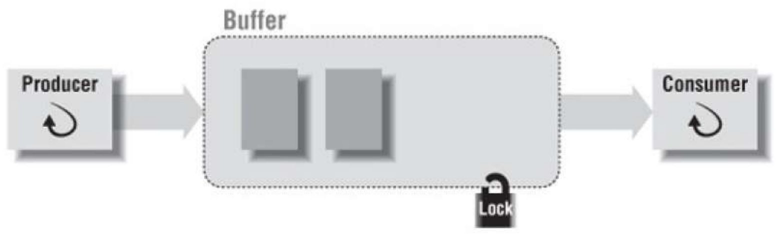
Buffering Data between Threads

The boss-worker, peer and pipeline are models for complete multithreaded programs.

- Within any of these models, threads transfer data to each other using ***buffers***.
- In the boss-worker model, the boss must transfer requests to the workers.
- In the pipeline model, each thread must pass input to the thread that performs the next stage of processing.
- Even in the peer model, peers may often exchange data.

A thread assumes either one of two roles when it exchanges data in a buffer with another thread.

- The thread that passes the data to another is known as the ***producer***.
- The one that receives that data is known as the ***consumer***.



Buffering Data between Threads

The ideal ***producer-consumer*** relationship requires:

- A buffer:
  - The buffer can be any data structure accessible to both producer and consumer.
- A lock:
  - Since the buffer is shared, a mutex is required to synchronize access to the buffer.
- A suspend/resume mechanism:
  - The consumer suspends itself when the buffer is empty.
  - If so the producer must be able to resume it when it places a new item in the buffer.
  - With Pthreads, we can implement this mechanism using a ***condition variable***.
- State of information:
  - A flag or a variable should indicate how much data is in the buffer.

Buffering Data between Threads

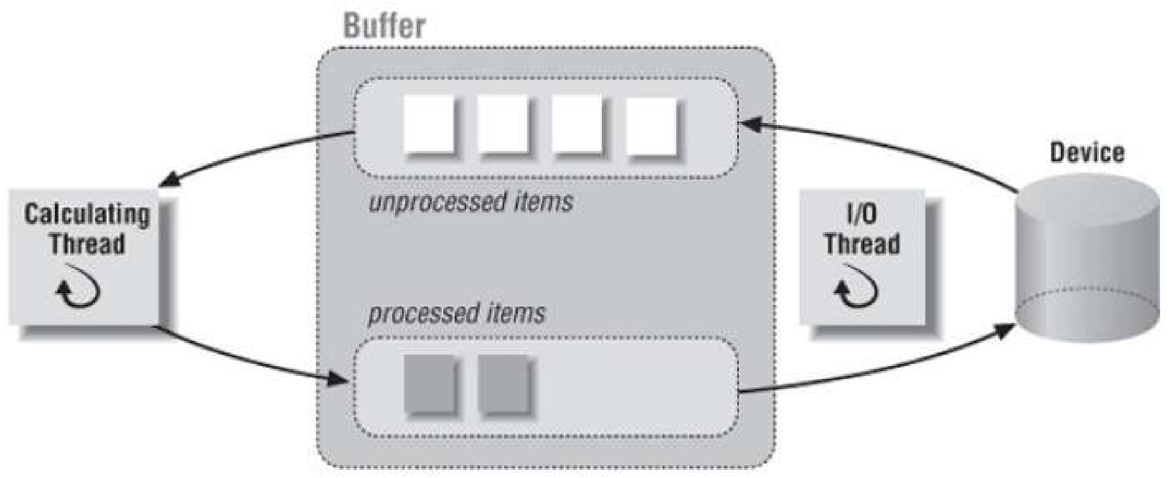
```
producer()
{
    .
    .
    .
  lock shared buffer
  place results in buffer
  unlock buffer
  wake up any consumer threads
    .
    .
    .
}

consumer()
{
    .
    .
    .
  lock shared buffer
  while state is not full {
        release lock and sleep
        awake and reacquire lock
        }
  remove contents
  unlock buffer
    .
    .
    .
}
```

Double Buffering

A more specialized producer-consumer relationship uses a technique known as ***double buffering***.

- Using double buffering, threads act both as producer and consumer to each other.
- In the figure, one set of buffers contains ***unprocessed data*** and another set contains ***processed data***.
- One thread, the IO thread, obtains unprocessed data from an IO device and places it in a shared buffer: in other words, it is the producer of unprocessed data.
- The IO thread also obtains processed data from another shared buffer and writes it to an IO device: in other words, it is the consumer of processed data.
- A second thread, the calculating thread, obtains unprocessed data from the shared buffer filled by the IO thread, processes it, and place its results in another shared buffer: the calculating thread is thus the consumer of unprocessed data and producer of processed data.

Double Buffering



Additional Features

In the second half of the lecture, we will discuss the following ***additional features*** provided by the Pthreads library:

- Thread Attributes
- One-Time Initialization

Thread Attributes

Threads have certain properties, called ***attributes***, that you can request through the Pthread library.

The Pthread standard defines attributes that determine the following characteristics of any given thread:

- Whether the thread is ***detached*** or ***joinable***:
  - All Pthread implementations provide this attribute.
- Size of the thread's ***private stack***:
  - An implementation provides this attribute if $\_POSIX\_THREAD\_ATTR\_STACKSIZE$ compile-time constant is defined.
- Location of the thread's ***private stack***:
  - An implementation provides this attribute if $\_POSIX\_THREAD\_ATTR\_STACKADDR$ compile-time constant is defined.

Thread Attributes

As we mentioned in the last lecture, a thread is created with a set of ***default attributes***.

So far, we have been using the default thread attributes by passing $NULL$ as an attribute parameter to the $pthread\_create()$ call.

To set a thread's attributes to something other than the default, we must perform the following steps:

- Define an attributes object of type $pthread\_attr\_t$
- Call $pthread\_attr\_init()$ to initialize the attributes object
- Make calls to specific Pthread functions to set individual attributes in the attributes object
- Specify the fully initialized attributes object to the $pthread\_create()$ call that creates the thread

Thread Attributes: Stack Size

A thread uses its ***stack*** to store local variables for each routine that it has called (but not yet exited) to its current point of execution.

- The thread also leaves various pieces of routine context information on the stack so that it can find its way back to the caller routine when it returns from the current one

Two factors can affect whether a thread will have enough room on its stack:

- The total memory required by all the local variables created during each routine call
- The number of routines that may be in its call chain at any one time

Even a single-thread program can sometimes run out of stack space.

- An individual thread’s stack is much smaller than that devoted to the entire process

Thread Attributes: Stack Size

With a ***process***, the amount of virtual address space is fixed.

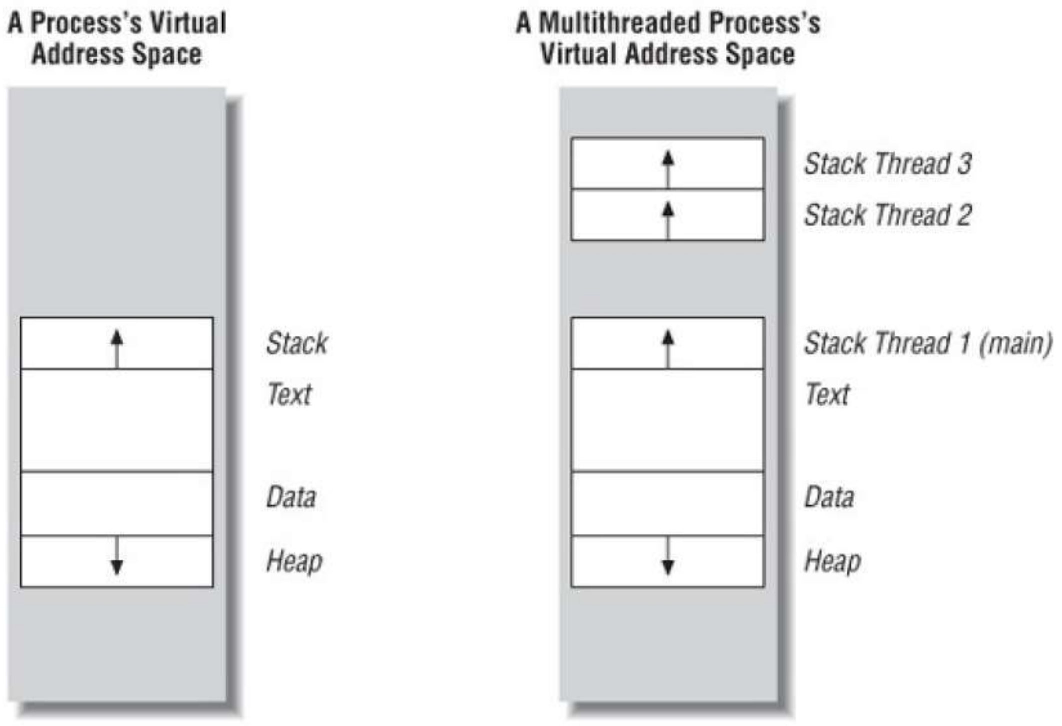
- Since there is only one stack, its size usually isn’t a problem.

With ***threads***, however, the same amount of virtual address space must be shared by all the thread stacks.

- You might have to reduce your default thread stack size if your application uses so many threads that the cumulative size of their stacks exceeds the available virtual address space.

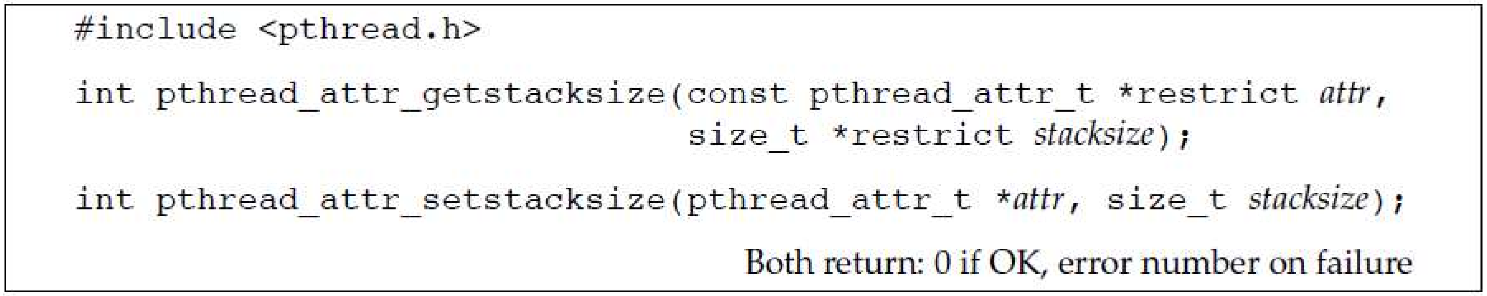
- On the other hand, if your threads call functions that allocate large local variables or call functions many stack frames deep, you might need more than the default stack size.

Thread Attributes: Stack Size


A Process's Virtual Address Space
Stack
Text
Data
Heap
A Multithreaded Process's Virtual Address Space
Stack Thread 3
Stack Thread 2
Stack Thread 1 (main)
Text
Data
Heap

Thread Attributes: Stack Size

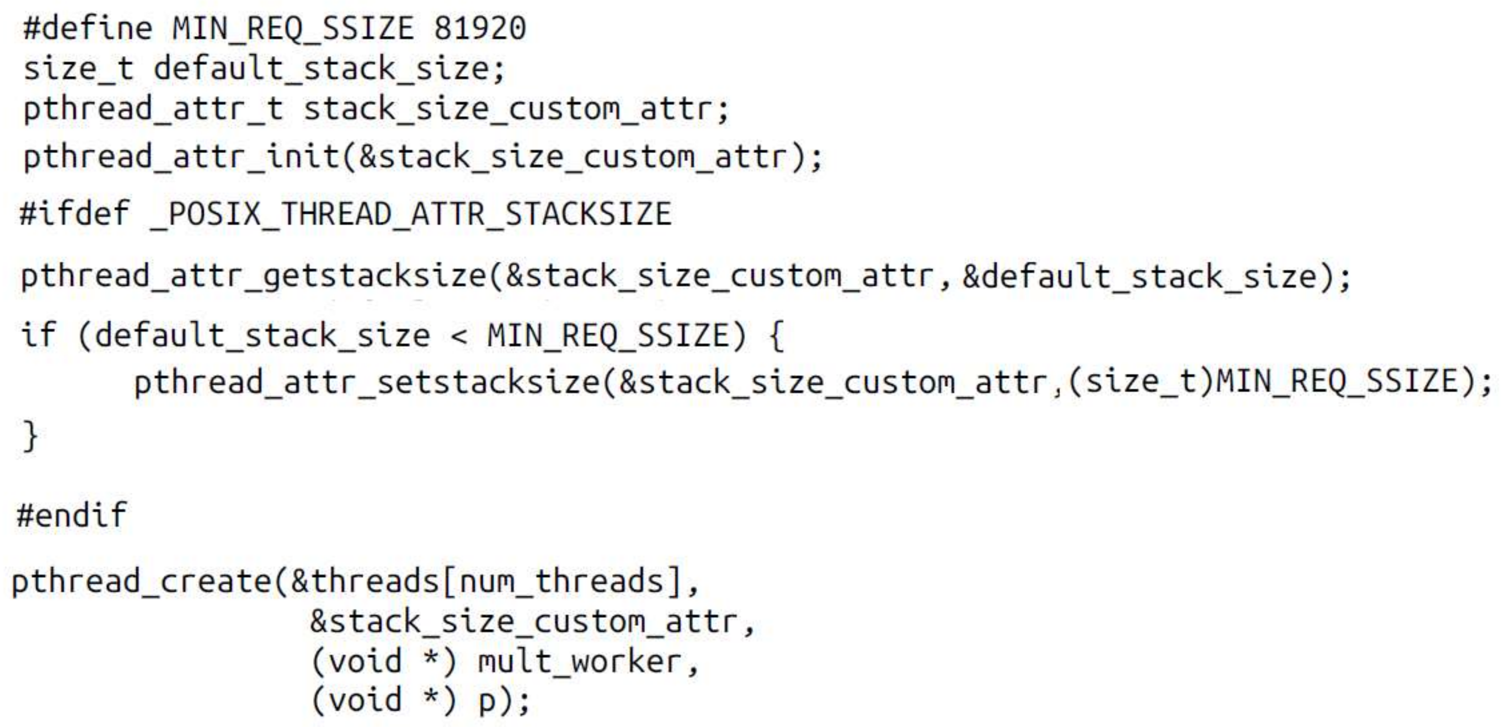
We can ***check*** and ***set*** the size of the stack of the ***calling thread*** using the following two Pthreads function.



```
#include <pthread.h>

int pthread_attr_getstacksize(const pthread_attr_t *restrict attr,
                              size_t *restrict stacksize);

int pthread_attr_setstacksize(pthread_attr_t *attr, size_t stacksize);
```

Both return: 0 if OK, error number on failure

Thread Attributes: Stack Size



```
#define MIN_REQ_SSIZE 81920
size_t default_stack_size;
pthread_attr_t stack_size_custom_attr;
pthread_attr_init(&stack_size_custom_attr);
#ifdef _POSIX_THREAD_ATTR_STACKSIZE
pthread_attr_getstacksize(&stack_size_custom_attr, &default_stack_size);
if (default_stack_size < MIN_REQ_SSIZE) {
      pthread_attr_setstacksize(&stack_size_custom_attr,(size_t)MIN_REQ_SSIZE);
}

#endif
pthread_create(&threads[num_threads],
               &stack_size_custom_attr,
               (void *) mult_worker,
               (void *) p);
```

Thread Attributes

Throughout this discussion, we've declared and initialized thread attribute objects using the $pthread\_attr\_init()$ call.

When we're finished using a thread attribute object, we can call $pthread\_attr\_destroy()$ to destroy it.

Note that ***existing threads*** that were created using this object are ***not affected*** when the object is destroyed.

One-Time Initialization

Sometimes, a multithreaded application wants to make sure that some initialization occurs ***only once***.

For example, a mutex may need to be initialized with special attributes using $pthread\_mutex\_init()$ and that initialization must occur only once.

- If we are creating threads from the main function, this is generally easy to achieve by performing initialization before creating any threads that depend on the initialization.

However, in a ***library function***, this is not always possible because the calling program may create the threads before the first call to the library function.

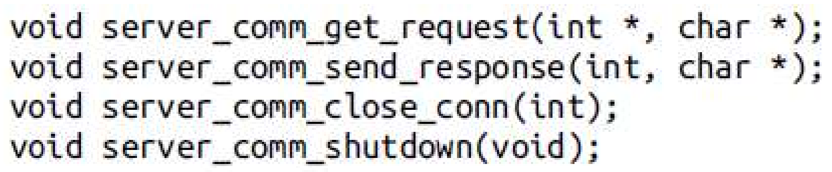
- Therefore, the library function needs a method of performing the initialization the first time that it is called from any thread.

One-Time Initialization

Let’s walk through a scenario to illustrate the point of the ***one-time initialization*** mechanism.

Suppose that we have an API library that is used to implement a communication module for an ATM server, whose purpose is to receives request from clients and unpacks them.

The interface to the module is as follows:



```
void server_comm_get_request(int *, char *);
void server_comm_send_response(int, char *);
void server_comm_close_conn(int);
void server_comm_shutdown(void);
```

Let’s pretend that this is legacy code that we have been asked to incorporate into a multithreaded application.

One-Time Initialization

```
void server_comm_get_request(int *conn, char *req_buf)
{
  int i, nr, not_done = 1;
  fd_set read_selects;

  if (!srv_comm_inited) {
    server_comm_init();
    srv_comm_inited = TRUE;
  }

  /* loop, processing new connection requests until a client
     buffer is read in on an existing connection. */

  while (not_done) {
  .
  .
  .
}
```

One-Time Initialization

If the $server\_comm\_inited$ flag is ***FALSE***, the $server\_comm\_get\_request()$ routine calls an initialization routine ($server\_comm\_init()$) and sets the flag to ***TRUE***.

If we allow ***multiple threads*** to call $server\_comm\_init()$ ***concurrently***, we introduce a ***data race*** on the $server\_comm\_inited$ flag and on all of $server\_comm\_init()$'s global variables and initializations.

Consider the following scenario:

- Threads ***A*** and ***B*** enter the routine at the same time.
- Thread ***A*** checks the value of $server\_comm\_inited$ and finds ***FALSE***.
- Thread ***B*** checks the value and also finds it ***FALSE***.
- Then they both go forward and call $server\_comm\_init()$.

One-Time Initialization

We will consider ***two possible solutions***:

- Adding a statically initialized mutex to protect the $server\_comm\_inited$ flag and $server\_comm\_init()$ routine

- Designating that the entire routine needs special synchronization handling using the ***One-Time Initialization*** mechanism

One-Time Initialization

Statically Initialized Mutex:

```
pthread_mutex_t init_mutex = PTHREAD_MUTEX_INITIALIZER;

void server_comm_get_request(int *conn, char *req_buf)
{
  int i, nr, not_done = 1;
  fd_set read_selects;

  pthread_mutex_lock(&init_mutex)
  if (!srv_comm_inited) {
    server_comm_init();

    srv_comm_inited = TRUE;
  }
  pthread_mutex_unlock(&init_mutex);

  /* loop, processing new connection requests until a client
     buffer is read in on an existing connection. */

  while (not_done) {
    .
    .
    .
}
```

One-Time Initialization

One-Time Initialization Mechanism:

```
pthread_once_t     srv_comm_inited_once = PTHREAD_ONCE_INIT;

void server_comm_get_request(int *conn, char *req_buf)
{
  int i, nr, not_done = 1;
  fd_set read_selects;

  pthread_once(&srv_comm_inited_once, server_comm_init);

  /* loop, processing new connection requests until a client
     buffer is read in on an existing connection. */

  while (not_done) {
     .
     .
     .
}
```

```
#include <pthread.h>

int pthread_once(pthread_once_t *once_control, void (*init)(void));
```

Returns 0 on success, or a positive error number on error

One-Time Initialization

If we use the $server\_comm\_init()$ routine only through the $pthread\_once$ mechanism, we can make the following synchronization guarantees:

- No matter how many times it is invoked by one or more threads, the routine will be executed only once by its ***first caller***.
- No caller will exit from the $pthread\_once$ mechanism until the routine's first caller has returned.

To use the $pthread\_once$ mechanism,

- you must declare a variable known as a ***once block*** ($pthread\_once\_t$), and you must statically initialize it to the value $PTHREAD\_ONCE\_INIT$.
- The Pthreads library uses a ***once block*** to maintain the state of $pthread\_once$ synchronization for a particular routine.
- Note that we are statically initializing the once block to the $PTHREAD\_ONCE\_INIT$ value.

One-Time Initialization

You can declare ***multiple once blocks*** in a program, associating each with a different routine.

Be careful, though !!!

- Once you associate a routine with the $pthread\_once$ mechanism, you must always call it through a the $pthread\_once$ call, using the ***same once block***.
- You cannot call the routine directly elsewhere in your program.

Notice that the $pthread\_once$ interface does not allow you to pass arguments to the routine that is protected by the once block.

- If you're trying to fit a predefined routine with arguments into the $pthread\_once$ mechanism, you'll have to fiddle a bit with global variables, wrapper routines, or environment variables to get it to work properly without subverting the synchronization the $pthread\_once$ mechanism is meant to provide

Advanced Features

There are other ***advanced features*** that are not covered in this lecture:

- Thread-Local Storage (TLS)
- Cancellation
- Mutex Attributes
- Scheduling Policy and Priority
- Threads and Signals

You may want to refer to **[1]**.

Reference

[1] Bradford Nichols, Dick Buttlar, and Jacqueline Proulx Farrell. 1996. Pthreads programming. O'Reilly & Associates, Inc., USA.