Operating Systems COMS(3010A) Concurrency and Threads

Branden Ingram

branden.ingram@wits.ac.za

Office Number : ???

Recap

- Scheduling
- Basic Schemes
- Metrics
- MLFQ
- Proportional Share

Concurrency

- In the real world different activities often proceed at the same time
  - Members of a band are all playing their own instruments and reacting to each other's pacing
  - Cars driving on the road

Concurrency

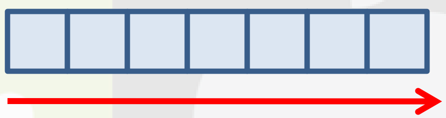
- In the real world different activities often proceed at the same time
  - Members of a band are all playing their own instruments and reacting to each other's pacing
  - Cars driving on the road
- We use the word "Concurrency" to refer to multiple activities at the same time
  - The real world is concurrent
  - And so are modern computers systems

Concurrency

- Examples in applications
  - Network services handle multiple requests from their clients
    - Imagine Google only handling one page request at a time
  - Most applications have multiple user interfaces
  - Parallel programming and utilisation of multiple processors

How do we achieve concurrency in computers?

- From the programmers perspective its much easier to think sequentially



How do we achieve concurrency in computers?

- From the programmers perspective its much easier to think sequentially

- The question is how can you write a correct program with dozens of events happening at once

- The key idea is to write a concurrent program, one with many simultaneous activities

How do we achieve concurrency in computers?

- From the programmers perspective its much easier to think sequentially

- The question is how can you write a correct program with dozens of events happening at once

- The key idea is to write a concurrent program, one with many simultaneous activities

- The solution to these are “Threads”

Threads

- Threads are a set of sequential streams of execution that interact and share results in very precise ways

Threads

- Threads are a set of sequential streams of execution that interact and share results in very precise ways

- Threads let us define a set of tasks that run concurrently while the code for each task is sequential

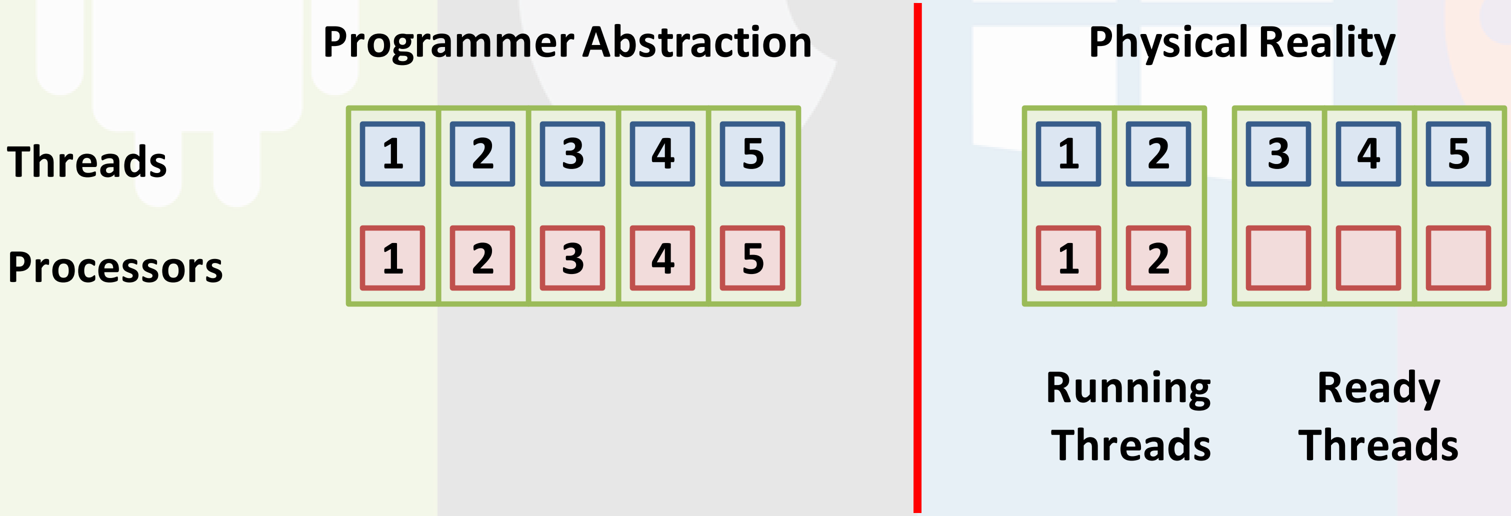
- Each thread behaves as if it has its own dedicated processor

therefore each thread has its own address space

- Utilising threads often requires additional work from the application programmer for coordinating

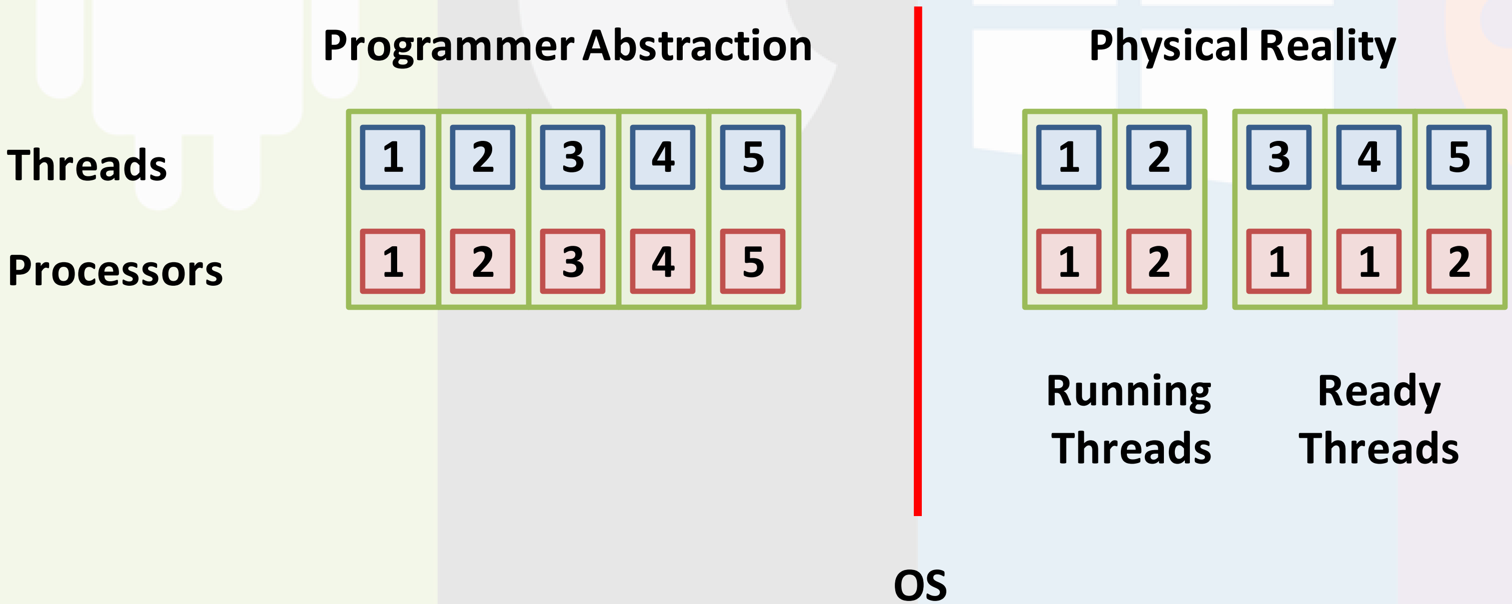
Threads

- The OS provides the illusion that programmers can make as many threads as needed
- Each thread runs on its own dedicated virtual processor



Threads

- In reality, a machine only has a finite number of processors
- It is the job of the OS to manage/link the running of threads onto actual hardware



What is the intuition behind using threads?

- In a program we can represent each concurrent task as a thread

- Each thread provides the abstraction of a sequential execution similar to a traditional program

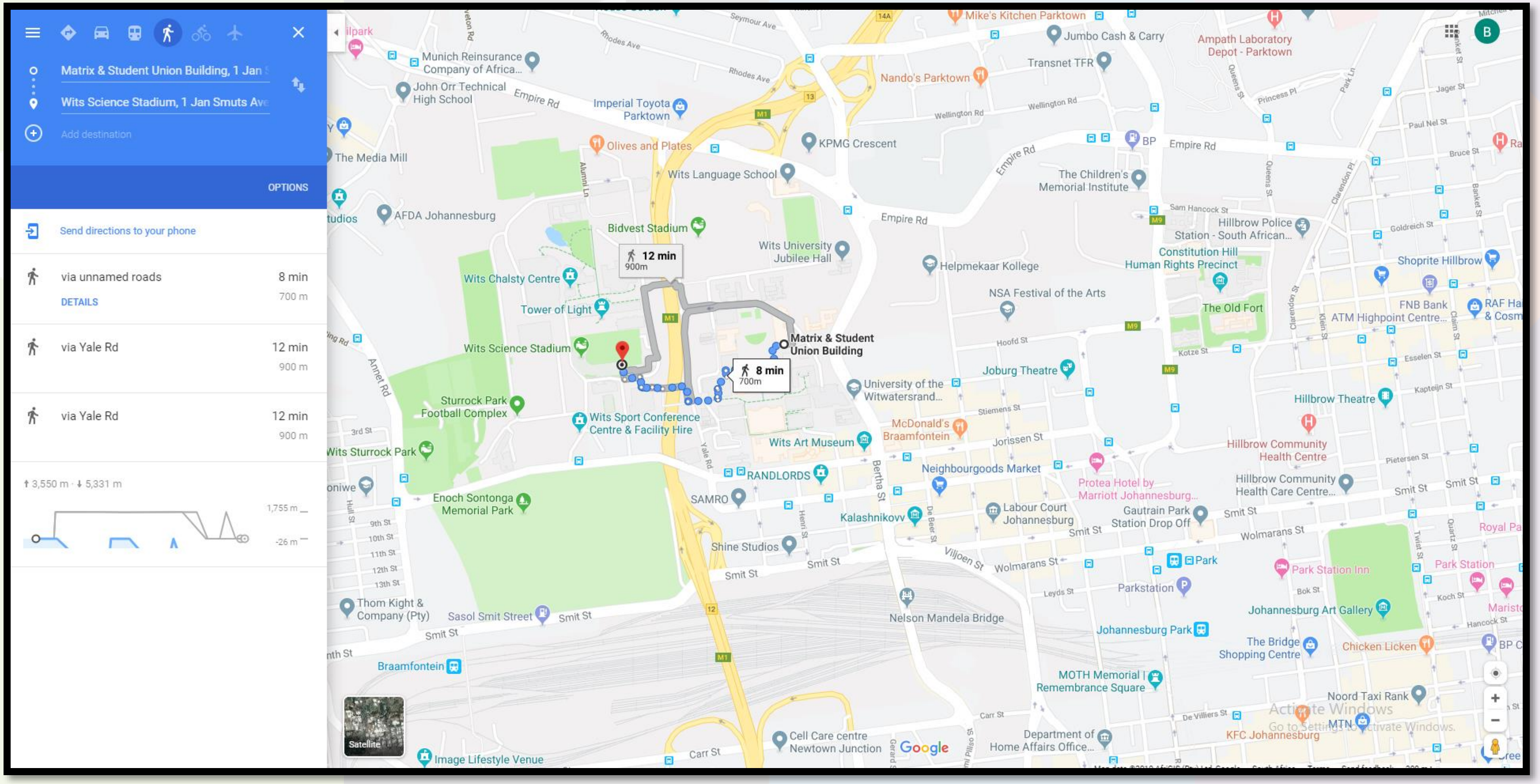
- Traditional programs can be considered as a “single-threaded program”
  - each instruction follows the next

- A “multi-threaded program” is a generalization of the same programming model
  - each thread follows a single sequence of steps

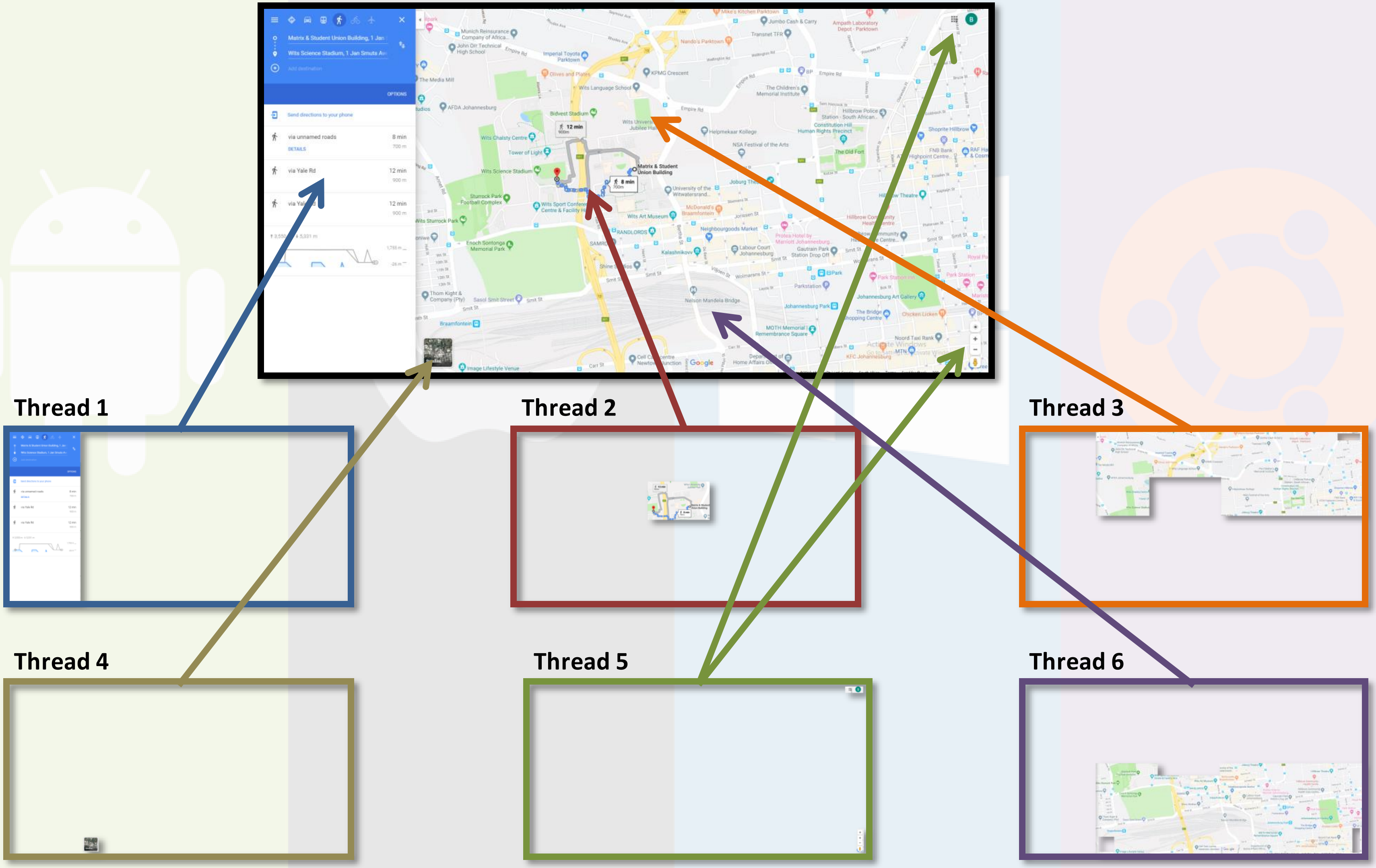
- However, a program can now have several threads executing at the same time

Uses of Threads

- Google Maps



Uses of Threads



Uses of Threads

- Four Reasons to use Threads
  - Programming structure
    - In order to keep track of multiple activities
  - Responsiveness
    - In order to maintain a high level of interface responsiveness
  - Performance - Processing
    - In order to exploit parallel code and multiple processors
  - Performance – I/O
    - In order to hide I/O latency

Programming structure

- Threads let you express an applications natural concurrency by writing each concurrent task as a separate thread

Programming structure

- **Threads let you express an applications natural concurrency by writing each concurrent task as a separate thread**
- **Example**
  - To get the screen input while also redrawing the screen pixels requires the physical processors to split their time
  - You could manually implement a program that interleaves these activities
    - i.e. Draw some pixels, check mouse input, draw some pixels
  - However, using threads greatly simplifies this process
    - The OS handles the threads as if they were individual processes

Responsiveness

- In order to preserve responsiveness and performance, a common design pattern is to create threads to perform work in the background

Responsiveness

- **In order to preserve responsiveness and performance, a common design pattern is to create threads to perform work in the background**

- **Example**
  - Many applications have a main loop; execute command; wait for next command
  - If some command takes a long time, the user will have to wait for that execution to finish before a new one can be given

  - However, with threads we can shift those time consuming commands onto another thread, and continue cycling that main loop

Performance - Processing

- Programs can use threads on a multiprocessor machine to do work in parallel
    - Thus doing the same amount of work in less time

Performance - Processing

- **Programs can use threads on a multiprocessor machine to do work in parallel**
  - **Thus doing the same amount of work in less time**

- **Example**
  - On Google Maps a single threaded machine would have to renderer each pixel on its own

  - However, on an 8 processor machine we can divide that task up into 8 pieces and spread it over the 8 processors
    - Thus decreasing the time taken to complete the job

Performance – I/O

- Computers are constantly interacting with the outside world with I/O devices
- Now by running tasks as separate threads, when one task is waiting for I/O the processor can make progress on the other

Performance – I/O

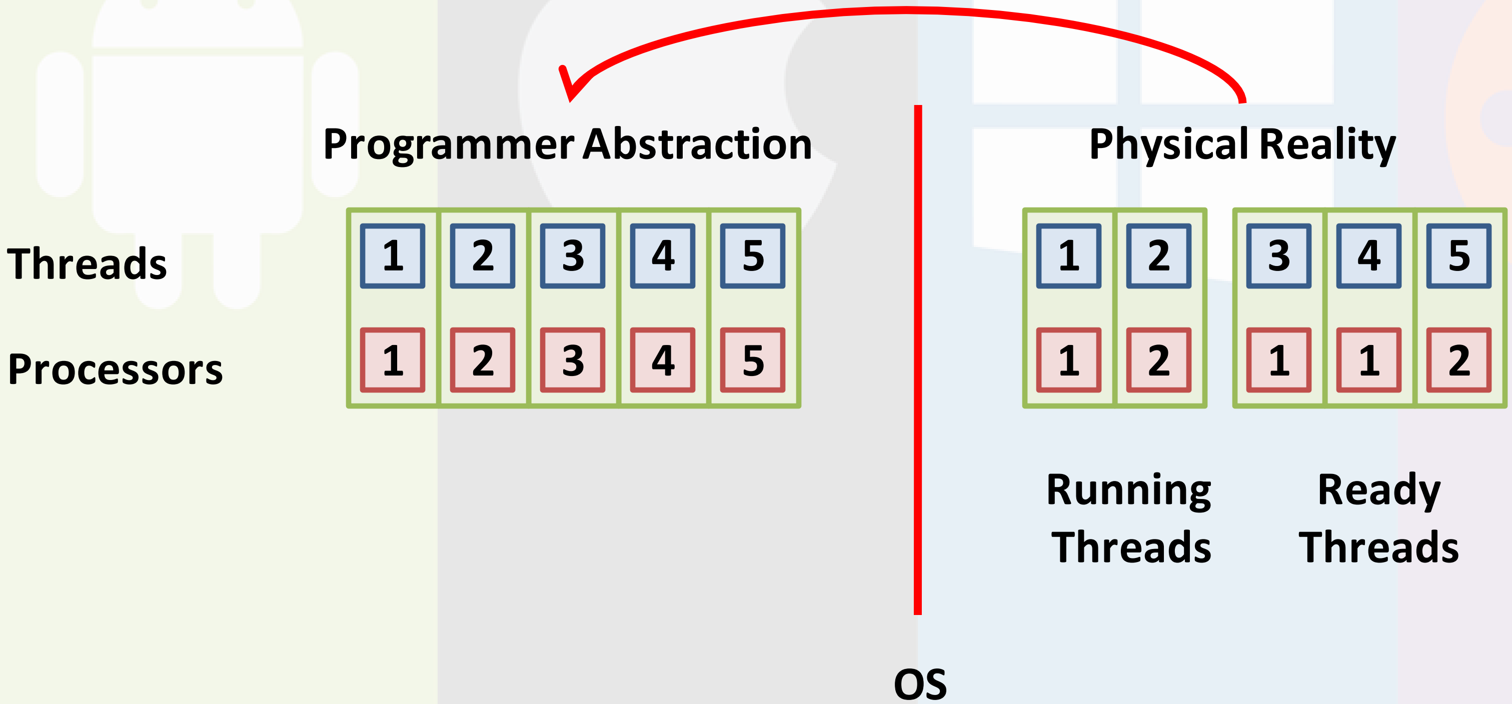
- **Computers are constantly interacting with the outside world with I/O devices**
- **Now by running tasks as separate threads, when one task is waiting for I/O the processor can make progress on the other**

- **Example**
    - The latency to read from disk can be 10ms, which is enough time to execute millions of instructions on modern processors

- However, with thread based programs the processor can quickly switch to another process or thread in the interim time

Thread Abstraction

- A thread is a single execution sequence that represents a separately schedulable task
  - How does the programmer see the threads
- This is implemented by the abstraction process the OS performs



Thread Abstraction

- OS provide the illusion of an infinite number of processors but the underlying hardware has only a limited number of processors

- It is the OS's job to map these threads to processors
    - The OS uses a "thread scheduler" to switch between threads

Thread Abstraction

- Threads provide the illusion of an infinite number of processors but the underlying hardware has only a limited number of processors

- It is the OS's job to map these threads to processors
    - The OS uses a "thread scheduler" to switch between threads

Programmers View	Possible Execution	Possible Execution
x = x + 1	x = x + 1	x = x + 1
y = y + x	y = y + x	Thread suspended
z = x + 5y	z = x + 5y	Other threads run
		Thread resumed
		y = y + x
		z = x + 5y

Thread Abstraction

- Threads provide the illusion of an infinite number of processors but the underlying hardware has only a limited number of processors

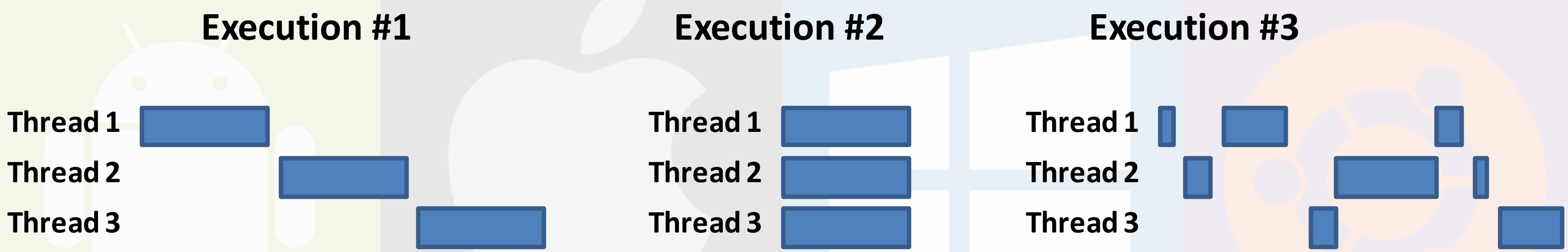
- It is the OS's job to map these threads to processors
  - The OS uses a "thread scheduler" to switch between threads

Programmers View	Possible Execution	Possible Execution
x = x + 1	x = x + 1	x = x + 1
y = y + x	y = y + x	Thread suspended
z = x + 5y	z = x + 5y	Other threads run
		Thread resumed
		y = y + x
		z = x + 5y

- Each thread runs on a dedicated virtual processor with unpredictable and variable speed

Why is the speed unpredictable?

- Threads can be interleaved in many possible ways during runtime



- Multi-threaded programs should make no assumptions about the behaviour of the thread scheduler

cant predict when stuff will finish in threading, we have to enforce it

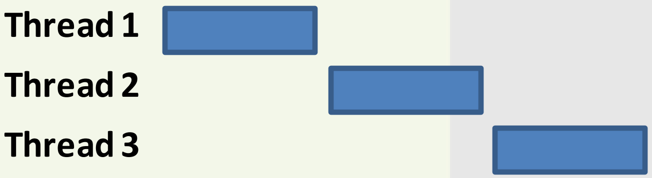
- Additionally, execution speed may vary based on normal operations, such as accessing the disk

Does the order of execution even matter?

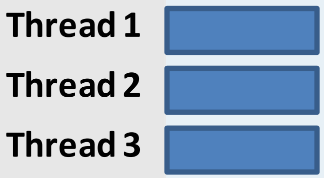
- If threads are completely independent of each other, shares no memory or other resources, then the order of execution does not matter

- However, most multi-threaded programs share data structures
  - Here the programmer must use explicit synchronisation to ensure program correctness regardless of the order of execution

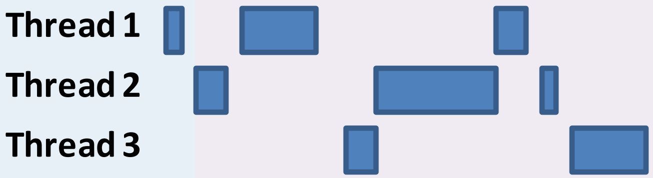
Execution #1



Execution #2



Execution #3



Simple Thread API

- void thread_create(thread,func,arg)
  - Creates a new thread
  - Concurrently with the calling thread, thread executes the function func with arguments arg

Simple Thread API

- void thread_create(thread,func,arg)
  - Creates a new thread
  - Concurrently with the calling thread, thread executes the function func with arguments arg

- void thread_yield()
  - The calling thread voluntarily gives up the processor to let other threads run

Simple Thread API

- void thread_create(thread,func,arg)
  - Creates a new thread
  - Concurrently with the calling thread, thread executes the function func with arguments arg
- void thread_yield()
  - The calling thread voluntarily gives up the processor to let other threads run
- int thread_join(thread)
  - Wait for thread to finish, then return the value passed to thread_exit by that thread

Simple Thread API

- void thread_create(thread,func,arg)
  - Creates a new thread
  - Concurrently with the calling thread, thread executes the function func with arguments arg
- void thread_yield()
  - The calling thread voluntarily gives up the processor to let other threads run
- int thread_join(thread)
  - Wait for thread to finish, then return the value passed to thread_exit by that thread
- void thread_exit(ret)
  - Finish the current thread
  - Store the value ret in the current threads data structure

Simple Thread API

- void thread_create(thread,func,arg)
- void thread_yield()
- int thread_join(thread)
- void thread_exit(ret)

- This API is based on the POSIX standard threads API
- These API's are what allow programmers to utilise threads

Simple Thread API

- **A good way to understand the simple threads API**
  - **Is that it provides a way to invoke asynchronous procedure calls**

- **A normal procedure call works by:**
  - Passing a set of arguments to a function
  - Runs the function immediately on the caller's stack
  - When the function is completed
    - It returns the result back

Simple Thread API

- **A good way to understand the simple threads API**
    - **Is that it provides a way to invoke asynchronous procedure calls**

- **A normal procedure call works by:**
    - Passing a set of arguments to a function
    - Runs the function immediately on the caller's stack
    - When the function is completed
        - It returns the result back

- **An asynchronous procedure call works by:**
    - Separating the call from the return
        - With thread_create() the caller starts the function
        - The caller continues execution concurrently with the called function
        - Later the caller can wait for the called function to continue with thread_join

Simple Thread API

- **A good way to understand the simple threads API**
  - **Is that it provides a way to invoke asynchronous procedure calls**

- **A normal procedure call works by:**
  - Passing a set of arguments to a function
  - Runs the function immediately on the caller's stack
  - When the function is completed
    - It returns the result back

- **An asynchronous procedure call works by:**
  - Separating the call from the return
    - With thread_create() the caller starts the function
    - The caller continues execution concurrently with the called function
    - Later the caller can wait for the called function to continue with thread_join()

- **thread_create() – is similar to fork, exec**
- **thread_join() – is similar to wait**

Thread Data Structures and Life Cycle

- What we know
  - Each thread represents a sequential stream of execution
  - The OS provides the illusion of each thread owning its own processor
  - This is handled through transparently suspending and resuming threads

- For this illusion to work
  - The OS must precisely load and save the state of a thread

Thread Data Structures and Life Cycle

- What we know
  - Each thread represents a sequential stream of execution
  - The OS provides the illusion of each thread owning its own processor
  - This is handled through transparently suspending and resuming threads

- For this illusion to work
  - The OS must precisely load and save the state of a thread

- However, because threads run either within processes or the kernel there is also a "shared state"
  - this state is not saved or restored when switching between threads

- Thus we must first define this shared and separated states

Thread Data Structures and Life Cycle

- In a running programming, threads share the; programs code, global static variables and the heap

Shared State

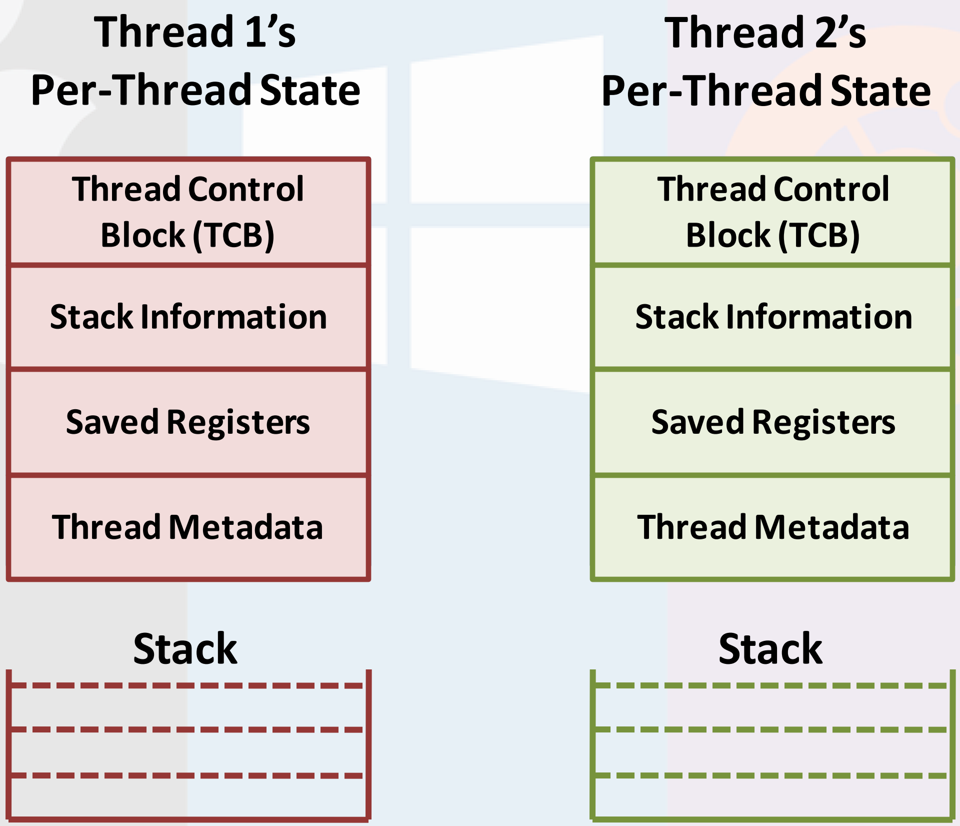
Code

Global Variables

Heap

Thread Data Structures and Life Cycle

- In a running programming, each thread also stores a per-thread state

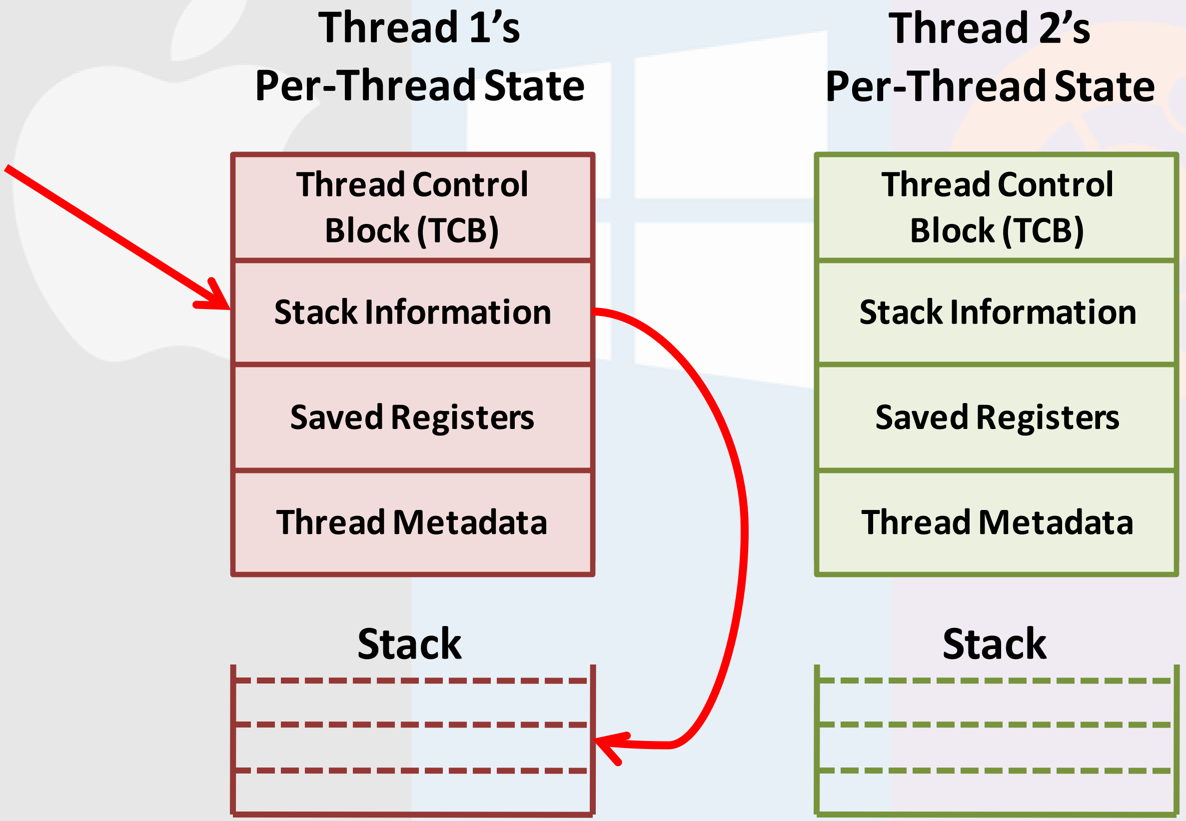


Thread Data Structures and Life Cycle

- In a running programming, each thread also stores a per-thread state

The TCB stores :

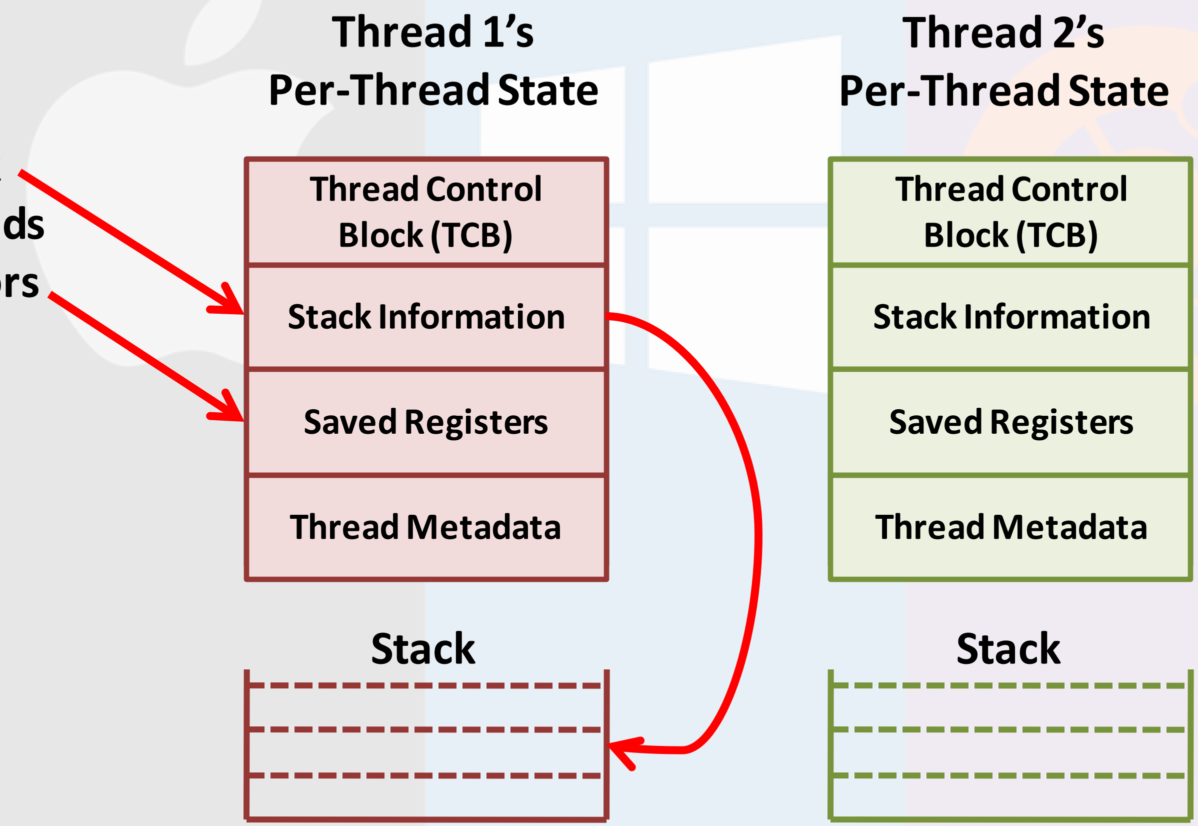
- A pointer to the stack



Thread Data Structures and Life Cycle

- In a running programming, each thread also stores a per-thread state

The TCB stores :
- A pointer to the stack
- Current state of threads computation, processors register values



Thread Data Structures and Life Cycle

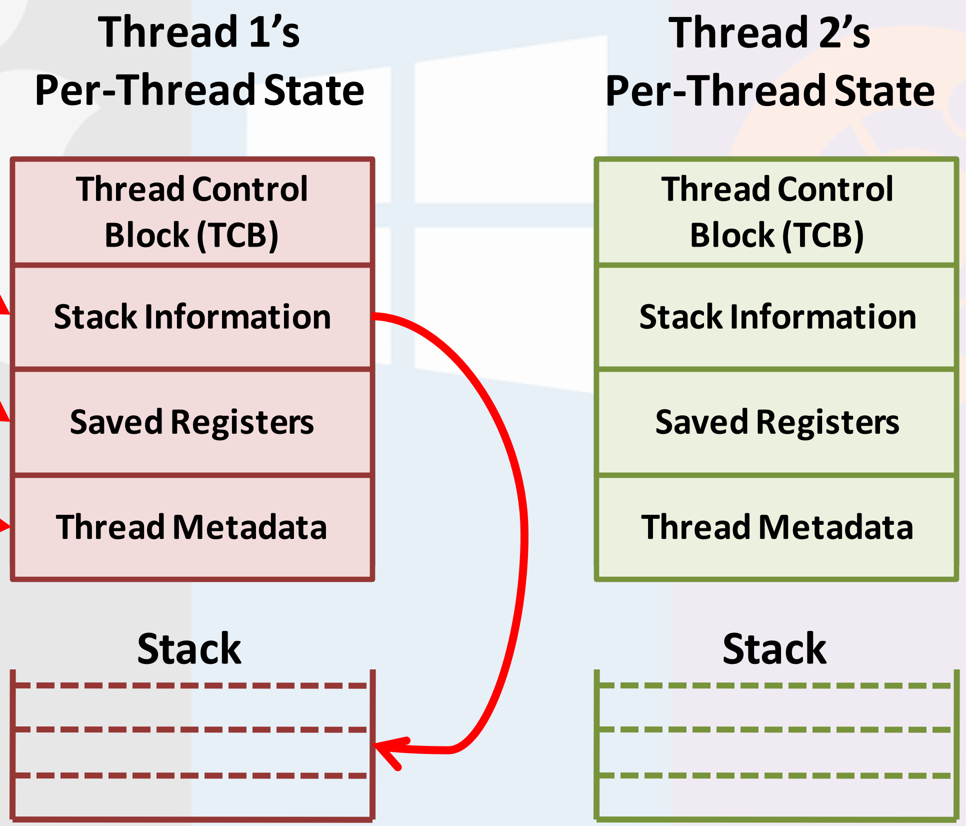
• In a running programming, each thread also stores a per-thread state

The TCB stores :

• A pointer to the stack

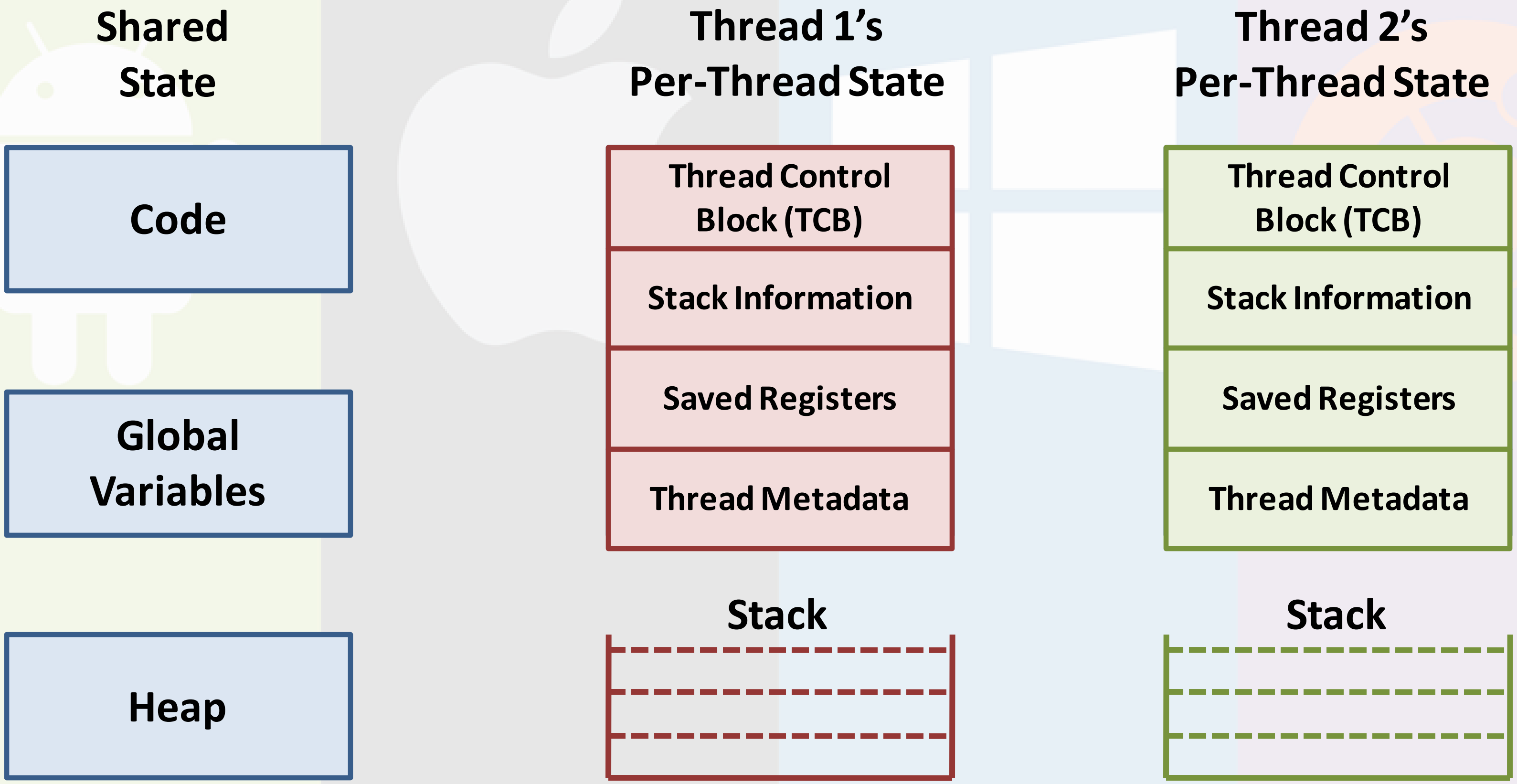
• Current state of threads computation, processors register values

• Metadata required to manage threads (thread id, scheduling priority, owner and resource usage)



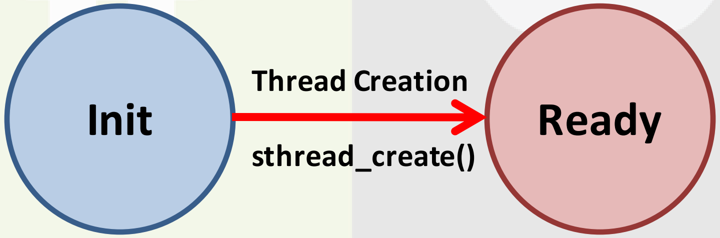
Thread Data Structures and Life Cycle

- Overall Picture



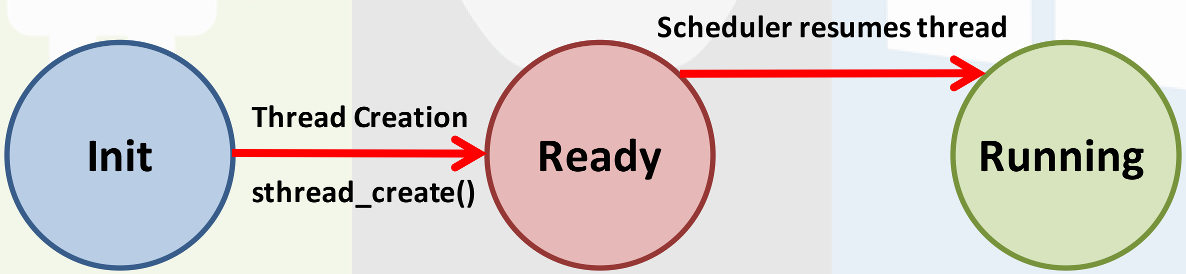
Thread Life Cycle

- INIT
  - Thread creation puts a thread into its INIT state
  - Allocates per-thread data structures
    - Once done puts thread into READY state by adding thread to a “ready list”



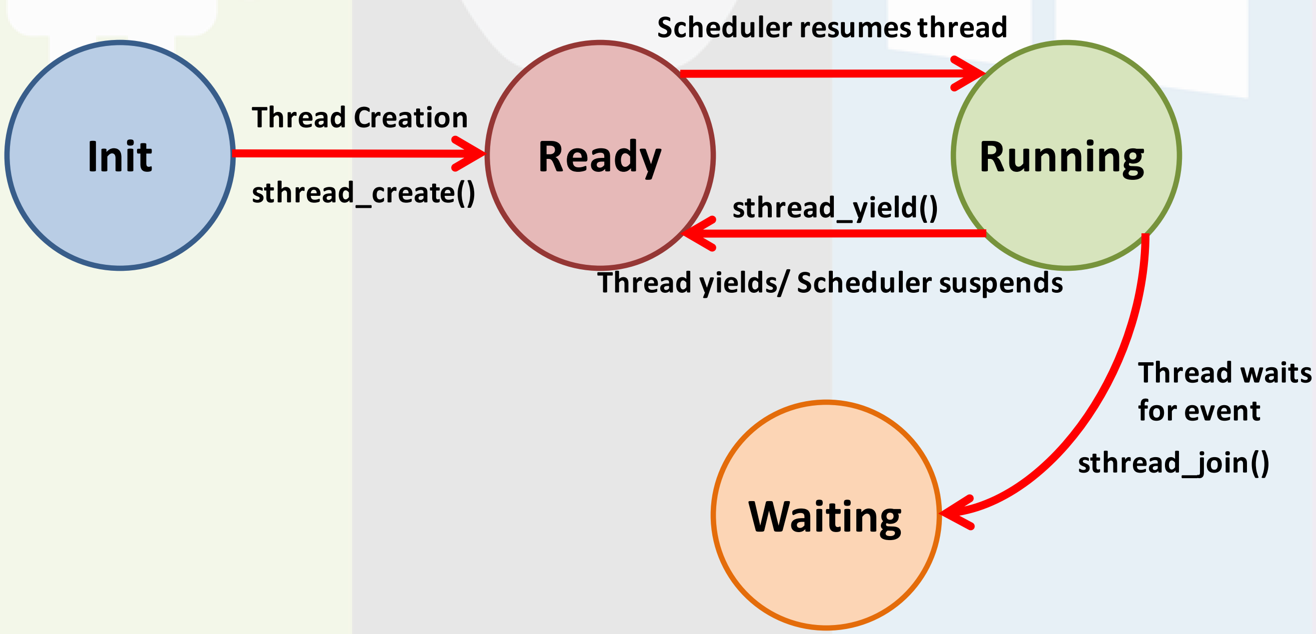
Thread Life Cycle

- READY
  - Available to run but not being run
  - Scheduler can at anytime cause a thread to transition to the RUNNING state



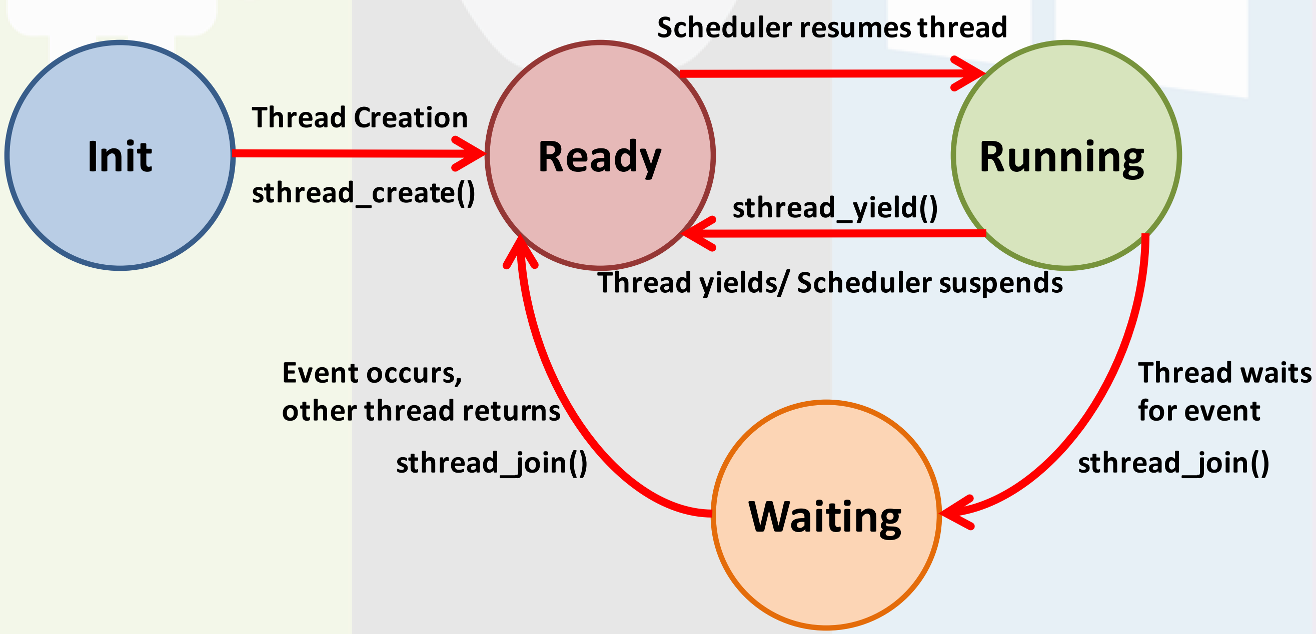
Thread Life Cycle

- RUNNING
  - A thread in the RUNINNG state is running on the processor
  - Register values are stored on the processor not in the TCB
  - RUNNING -> READY
    - Voluntarily give up the processor (yield)
    - Forcefully moved out by the scheduler moving a new thread into RUNNING
  - RUNNING -> WAITING
    - Moved to waiting when in need of input/other influences



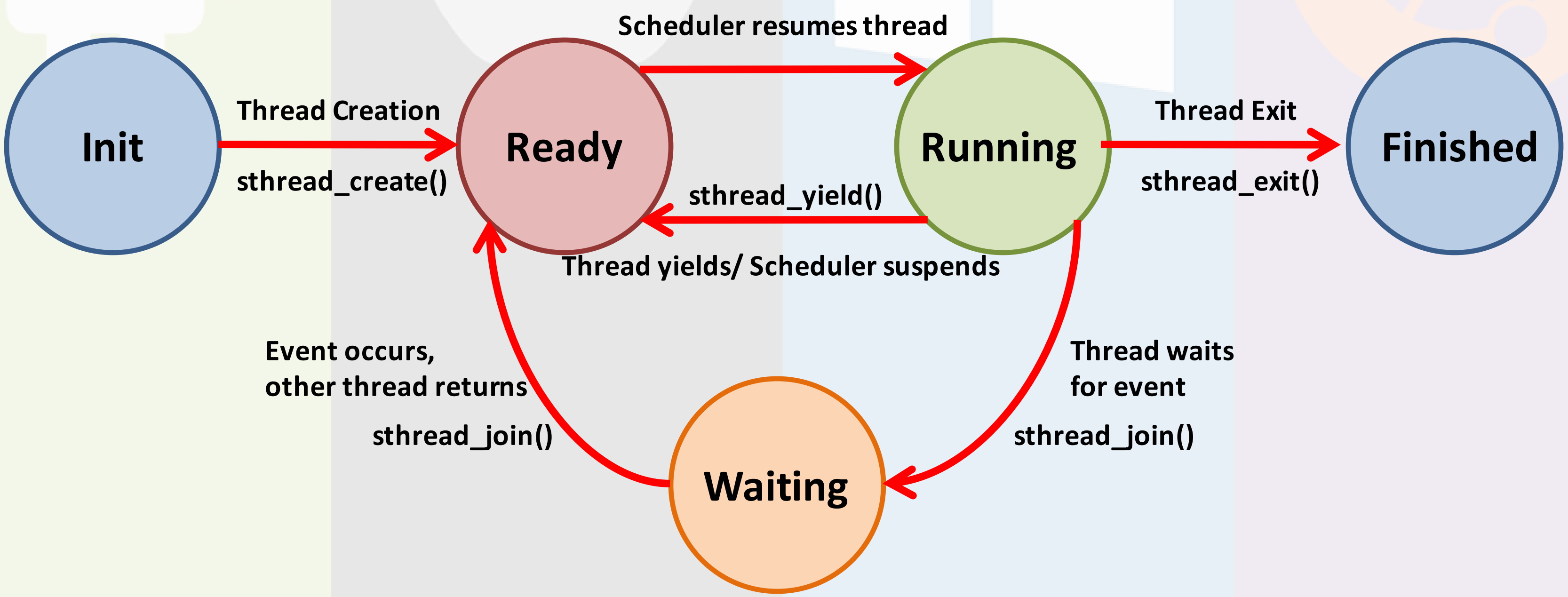
Thread Life Cycle

- WAITING
  - A thread in the WAITING state is waiting for some event
  - Cannot be moved until an action occurs
  - Example – after creating a child thread, the main thread must wait for them to complete by calling thread_join()
  - While in the WAITING state a thread cannot progress and therefore isn't useful to run
  - TCB is stored on the schedulers "waiting list"
  - When event occurs OS moves TCB to the "ready list"



Thread Life Cycle

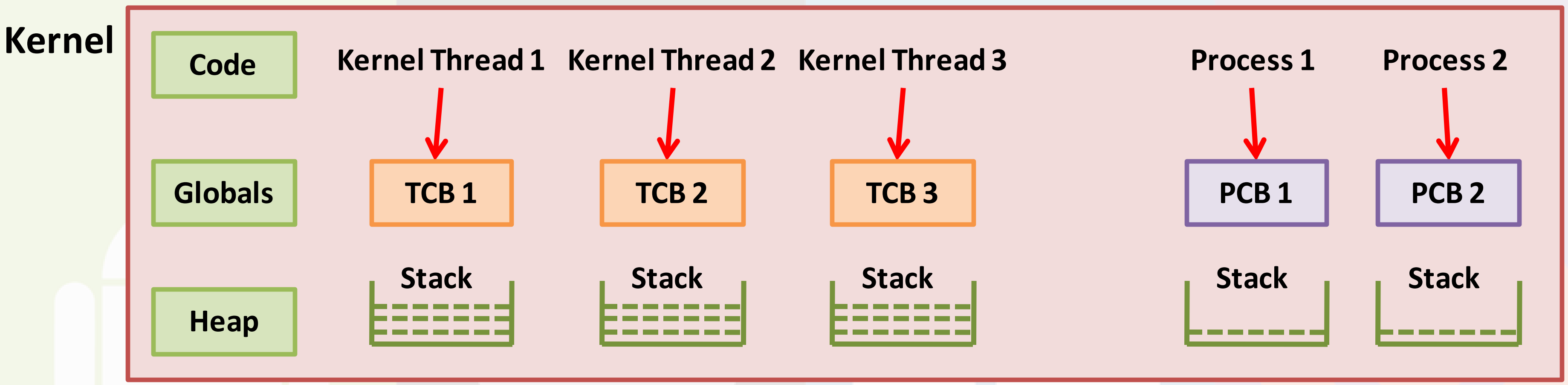
- FINISHED
  - A thread in the FINISHED state never runs again
  - System may free all or some of its state
  - The remnants of the TCB are put on the "finished list"
  - Passes its exit value to the parent thread

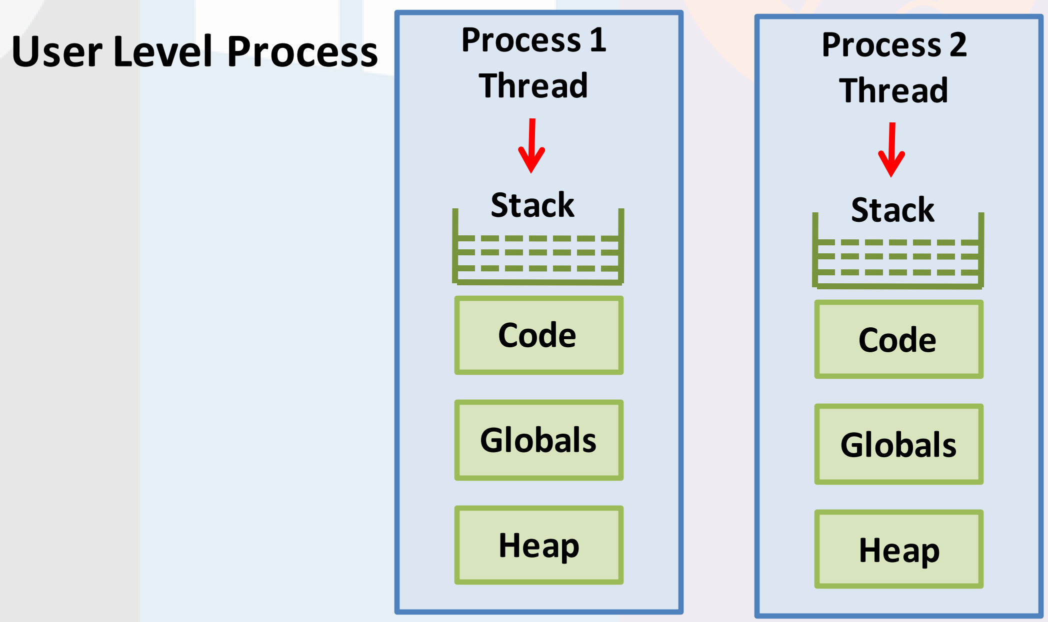


Kernel Threads

- This is the application of the thread abstraction within the kernel itself
- A kernel thread executes kernel code and modifies kernel data structures

Kernel Threads





- This is the application of the thread abstraction within the kernel itself
- A kernel thread executes kernel code and modifies kernel data structures

Threads Implementation

- **How to create and control threads?**

```
#include <pthread.h>

int
pthread_create(       pthread_t*          thread,
                const pthread_attr_t*  attr,
                      void*               (*start_routine)(void*),
                      void*               arg);
```

Threads Implementation

- **How to create and control threads?**

```
#include <pthread.h>

int
pthread_create(      pthread_t*         thread,
               const pthread_attr_t*   attr,
                     void*              (*start_routine)(void*),
                     void*              arg);
```

thread: Used to interact with this thread.

Threads Implementation

- **How to create and control threads?**

```
#include <pthread.h>

int
pthread_create(      pthread_t*       thread,
               const pthread_attr_t*  attr,
                     void*            (*start_routine)(void*),
                     void*            arg);
```

thread: Used to interact with this thread.
attr: Used to specify any attributes this thread might have.
Stack size, Scheduling priority, …

Threads Implementation

- **How to create and control threads?**

```
#include <pthread.h>

int
pthread_create(       pthread_t*        thread,
                const pthread_attr_t*  attr,
                      void*             (*start_routine)(void*),
                      void*             arg);
```

thread: Used to interact with this thread.
attr: Used to specify any attributes this thread might have.
Stack size, Scheduling priority, …
start_routine: the function this thread start running in.

Threads Implementation

- **How to create and control threads?**

```
#include <pthread.h>

int
pthread_create(       pthread_t*         thread,
                const pthread_attr_t*   attr,
                      void*              (*start_routine)(void*),
                      void*              arg);
```

thread: Used to interact with this thread.
attr: Used to specify any attributes this thread might have.
Stack size, Scheduling priority, …
start_routine: the function this thread start running in.
arg: the argument to be passed to the function (start routine)
a void pointer allows us to pass in any type of argument.

Threads Implementation

- If `start_routine` instead required another type argument, the declaration would look like this:

An integer argument:

```
int
pthread_create(…, // first two args are the same
                  void*  (*start_routine)(int),
                  int    arg);
```

Return an integer:

```
int
pthread_create(…, // first two args are the same
                  int  (*start_routine)(void*),
                  void*    arg);
```

Example : Create

```
#include <pthread.h>

typedef struct __myarg_t {
        int a;
        int b;
} myarg_t;

void *mythread(void *arg) {
        myarg_t *m = (myarg_t *) arg;
        printf("%d %d\n", m->a, m->b);
        return NULL;
}

int main(int argc, char *argv[]) {
        pthread_t p;
        int rc;

        myarg_t args;
        args.a = 10;
        args.b = 20;
        rc = pthread_create(&p, NULL, mythread, &args);
        …
}
```

Threads Implementation

- **How to create and control threads?**

```
int pthread_join(pthread_t thread, void **value_ptr);
```

Threads Implementation

- **How to create and control threads?**

```
int pthread_join(pthread_t thread, void **value_ptr);
```

thread: Specify which thread to wait for

Threads Implementation

- **How to create and control threads?**

```
int pthread_join(pthread_t thread, void **value_ptr);
```

thread: Specify which thread to wait for
value_ptr: A pointer to the return value
Because pthread_join() routine changes the value, you need to pass in a pointer to that value.

Example : Wait

```
#include <stdio.h>
#include <pthread.h>
#include <assert.h>
#include <stdlib.h>

typedef struct __myarg_t {
    int a;
    int b;
} myarg_t;

typedef struct __myret_t {
    int x;
    int y;
} myret_t;

void *mythread(void *arg) {
    myarg_t *m = (myarg_t *) arg;
    printf(“%d %d\n”, m->a, m->b);
    myret_t *r = malloc(sizeof(myret_t));
    r->x = 1;
    r->y = 2;
    return (void *) r;
}

```

Example : Wait (cont)

```
int main(int argc, char *argv[]) {
    int rc;
    pthread_t p;
    myret_t *m;

    myarg_t args;
    args.a = 10;
    args.b = 20;
    pthread_create(&p, NULL, mythread, &args);
    pthread_join(p, (void **) &m);  // this thread has been
                                     // waiting inside of the
                                     // pthread_join() routine.
    printf("returned %d %d\n", m->x, m->y);
    return 0;
}
```

Example : Dangerous Code

- **Be careful with how values are returned from a thread.**

```
void *mythread(void *arg) {
    myarg_t *m = (myarg_t *) arg;
    printf("%d %d\n", m->a, m->b);
    myret_t r; // ALLOCATED ON STACK: BAD!
    r.x = 1;
    r.y = 2;
    return (void *) &r;
}
```

When the variable `r` returns, it is automatically de-allocated.

Example : Simpler Argument Passing

- Just passing in a single value.

```
void *mythread(void *arg) {
    int m = (int) arg;
    printf(“%d\n”, m);
    return (void *) (arg + 1);
}

int main(int argc, char *argv[]) {
    pthread_t p;
    int rc, m;
    pthread_create(&p, NULL, mythread, (void *) 100);
    pthread_join(p, (void **) &m);
    printf(“returned %d\n”, m);
    return 0;
}
```