Operating Systems

Concurrency and Thread API (Chapter 26 ~ 27)

Young-Woo Kwon

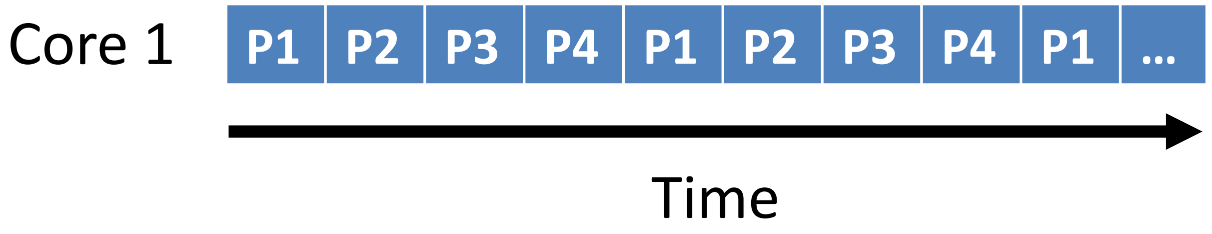
CONCURRENCY

Concurrent/Parallel Programs

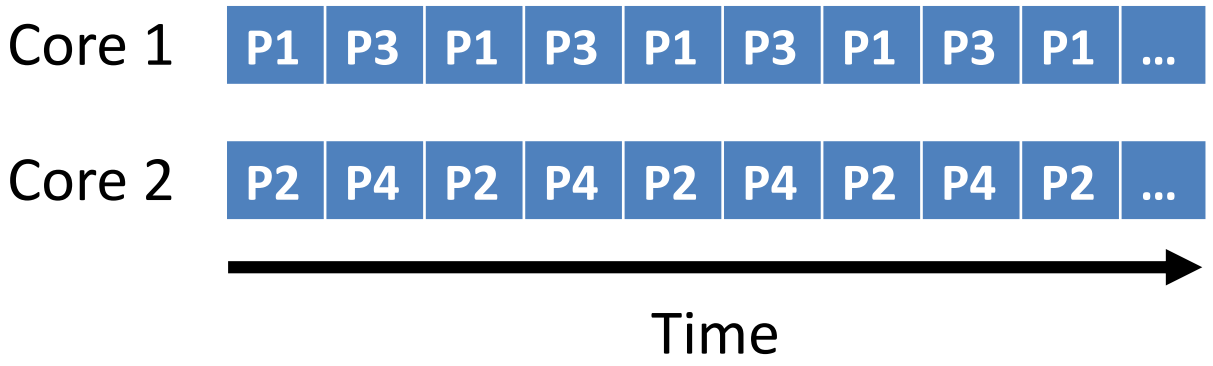
- To execute these programs we need to
  - **Create several processes** that execute in parallel (or concurrently)
  - Map each process to the **same address space to share data**
    - They are all part of the same computation
  - Have the **OS schedule** these processes in parallel
- This situation is very inefficient. Why?

Concurrency vs. Parallelism

- Concurrent execution on a single-core system:



- Parallel execution on a dual-core system:



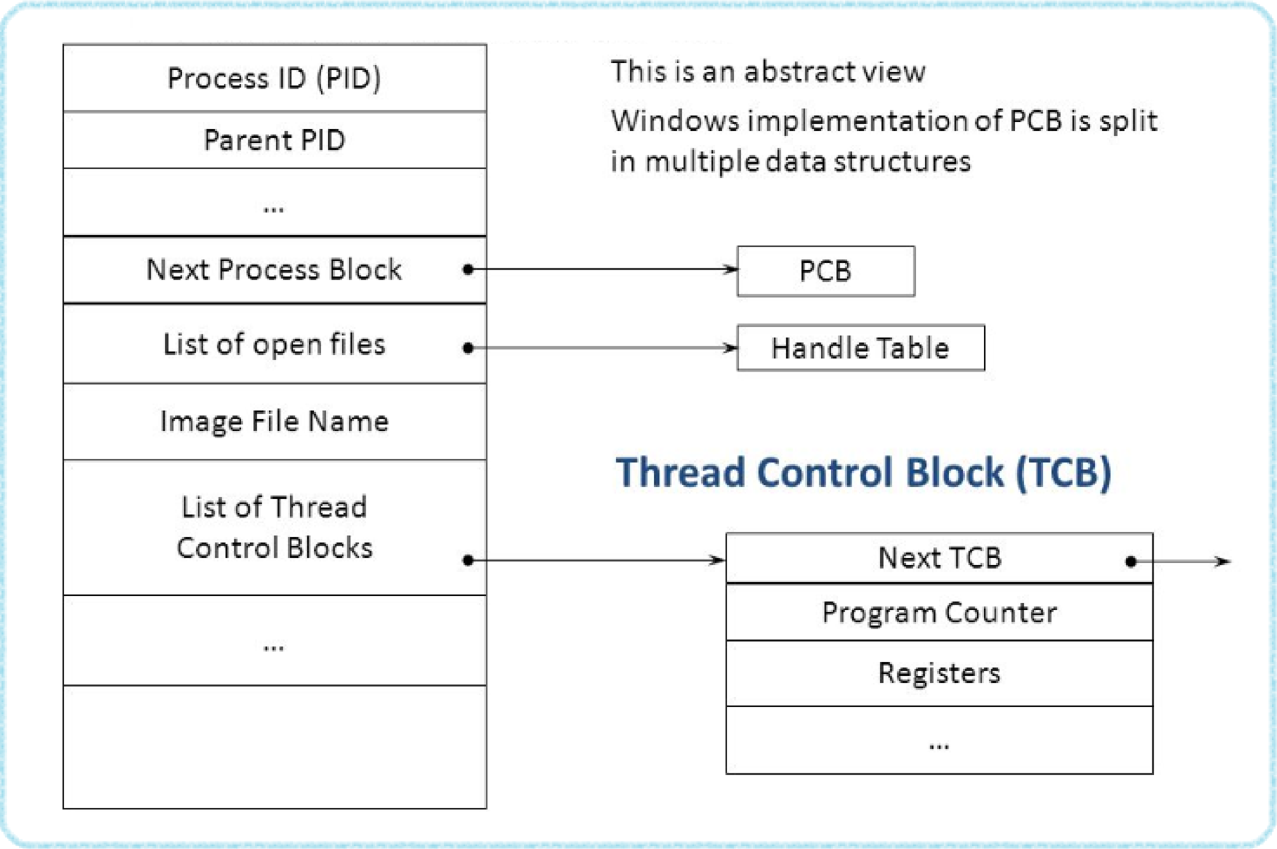
Thread

- A **thread** is a single execution sequence that represents a separately schedulable task

- Multi-threaded program
  - A multi-threaded program has **more than one point of execution**.
  - Multiple PCs (Program Counter)
  - They share the share the same address space.

Thread Control Block (TCB)

- Thread ID
- Thread state
- Pointer to parent PCB
- PC/registers for thread
- Stack location in memory map.



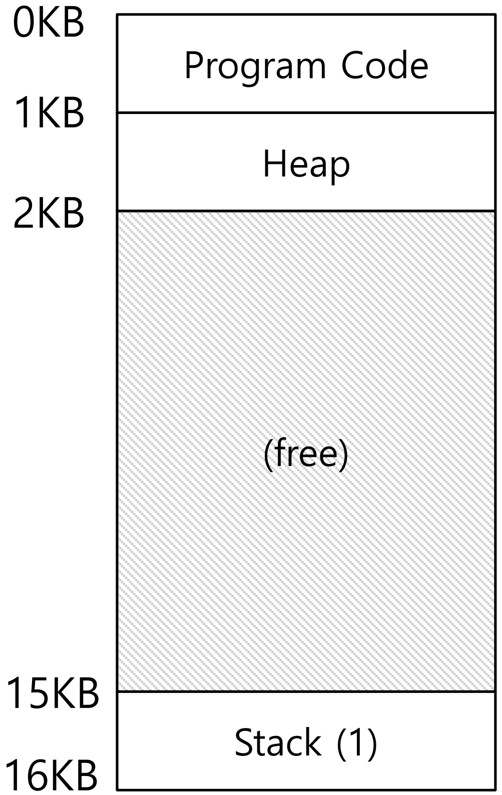
Context switch between threads

- Each thread has its own <u>program counter</u> and <u>set of registers</u>.
  - One or more **thread control blocks(TCBs)** are needed to store the state of each thread.

- When switching from running one (T1) to running the other (T2),
  - The register state of T1 be saved.
  - The register state of T2 restored.
  - The address space remains the same.

The stack of the relevant thread

- There will be one stack per thread.

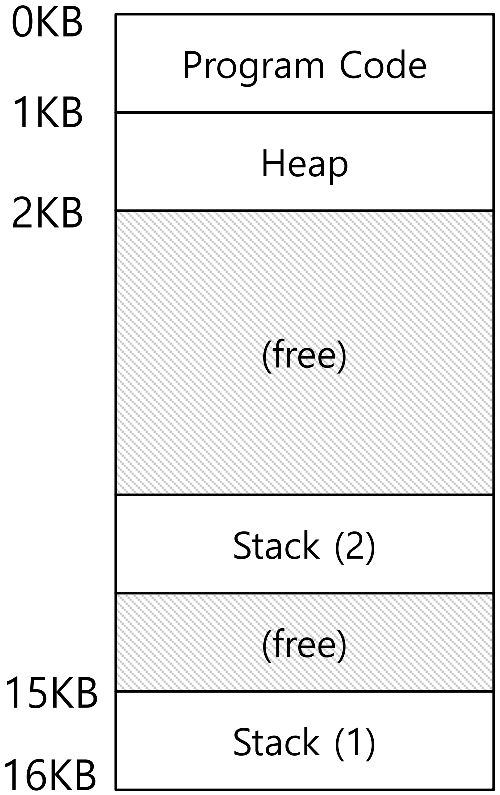


The code segment: where instructions live

The heap segment: contains malloc'd data dynamic data structures (it grows downward)

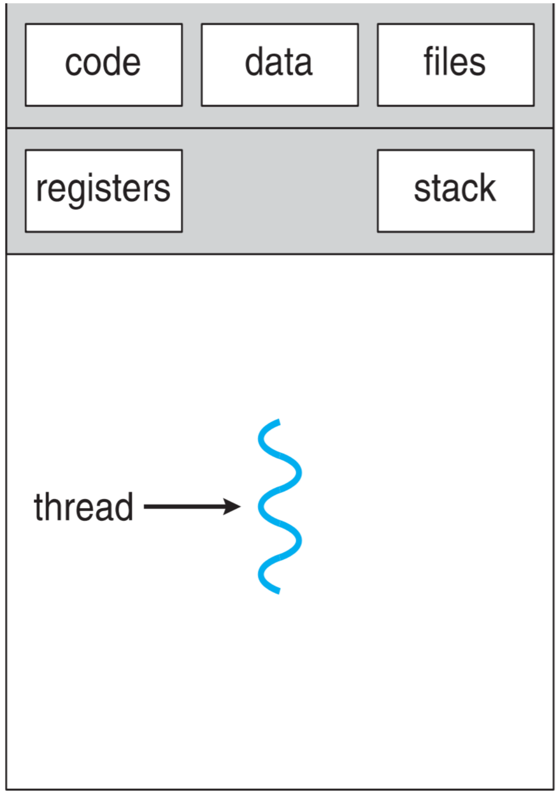
(it grows upward)
The stack segment: contains local variables arguments to routines, return values, etc.

A Single-Threaded Address Space

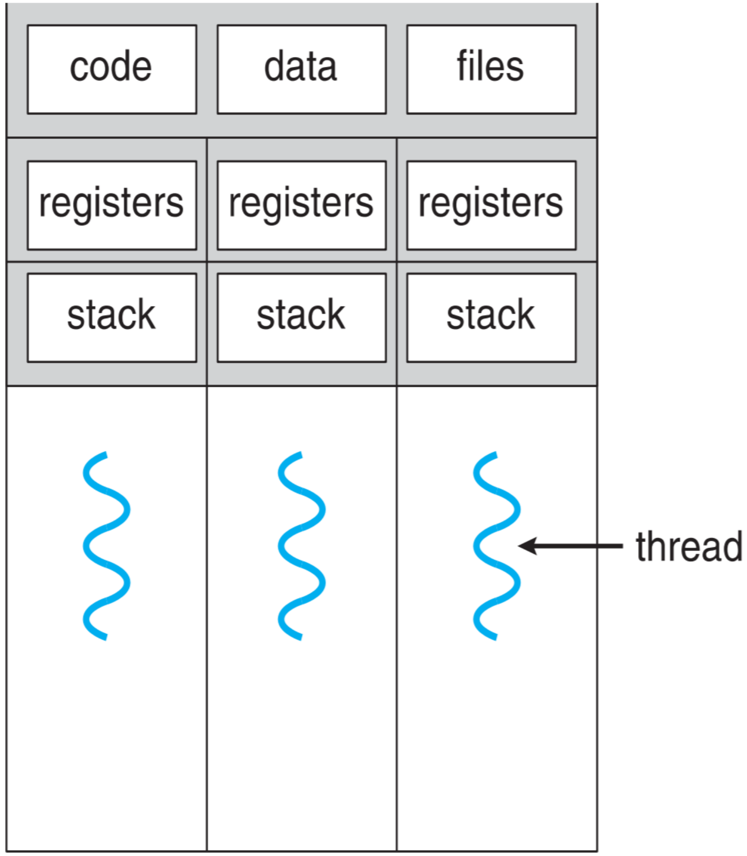


Two threaded Address Space

Single vs. Multithreaded Processes



single-threaded process



multithreaded process

Why Use Threads?

- Parallelism
  - One thread per CPU can make programs **run faster** on multiple processors
- I/O overlapping
  - **Avoid blocking program progress** due to slow I/O
  - While one thread in your program waits (i.e., blocked waiting for I/O), the CPU scheduler can switch to other threads, which are ready to run and do something useful.
  - Similar to the effect of multiprogramming

Why Use Threads?

- You could use multiple processes instead of threads.
  – Processes are a more sound choice for logically separate tasks
- But,
  – **Process creation is heavy-weight** while thread creation is lightweight
  – **Threads share an address space** and thus make it easy to share data
  – Can simplify code, **increase efficiency**

Threads are “lighter weight” than processes

- To make a new thread, we only need a **stack.**
  - Can share all pre-existing resources.
  - Done at user-level without system calls
- To make a new process, we have to talk to the **operating system**
  - Make a new address space
  - Inherit resources from the original space
  - Compete for the same underlying global scheduler
  - Can easily run out, or thrash

Benefits

- **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, thread switching lower overhead than context switching
- **Scalability**
  – Threads can take advantage of multiprocessor architectures

An Example: Thread Creation

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

void *mythread (void *arg) {
    printf (“%s\n”, (char *) arg);
    return NULL;
}

int main (int argc, char *argv[]) {
    pthread_t p1, p2;
    int rc;
    printf(“main: begin\n”);
    pthread_create(&p1, NULL, mythread, “A”);
    pthread_create(&p2, NULL, mythread, “B”);
    // join waits for thre threads to finish
    pthread_join(p1, NULL);
    pthread_join(p2, NULL);
    printf(“main: end\n”);
    return 0;
}
```

Thread Trace (1)

main	**Thread 1**	**Thread 2**
starts running		
prints “main: begin”		
creates Thread 1		
creates Thread 2		
waits for T1		
	runs	
	prints “A”	
	returns	
waits for T2		
		runs
		prints “B”
		returns
prints “main: end”		

Thread Trace (2)

main	**Thread 1**	**Thread 2**
starts running		
prints “main: begin”		
creates Thread 1		
	runs	
	prints “A”	
	returns	
creates Thread 2		
		runs
		prints “B”
		returns
waits for T1		
returns immediately; T1 is done		
waits for T2		
returns immediately; T2 is done		
prints “main: end”		

Thread Trace (3)

main	Thread 1	Thread 2
starts running		
prints "main: begin"		
creates Thread 1		
creates Thread 2		
		runs
		prints "B"
		returns
waits for T1		
	runs	
	prints "A"	
	returns	
waits for T2		
returns immediately; T2 is done		
prints "main: end"		

Issues when using threads

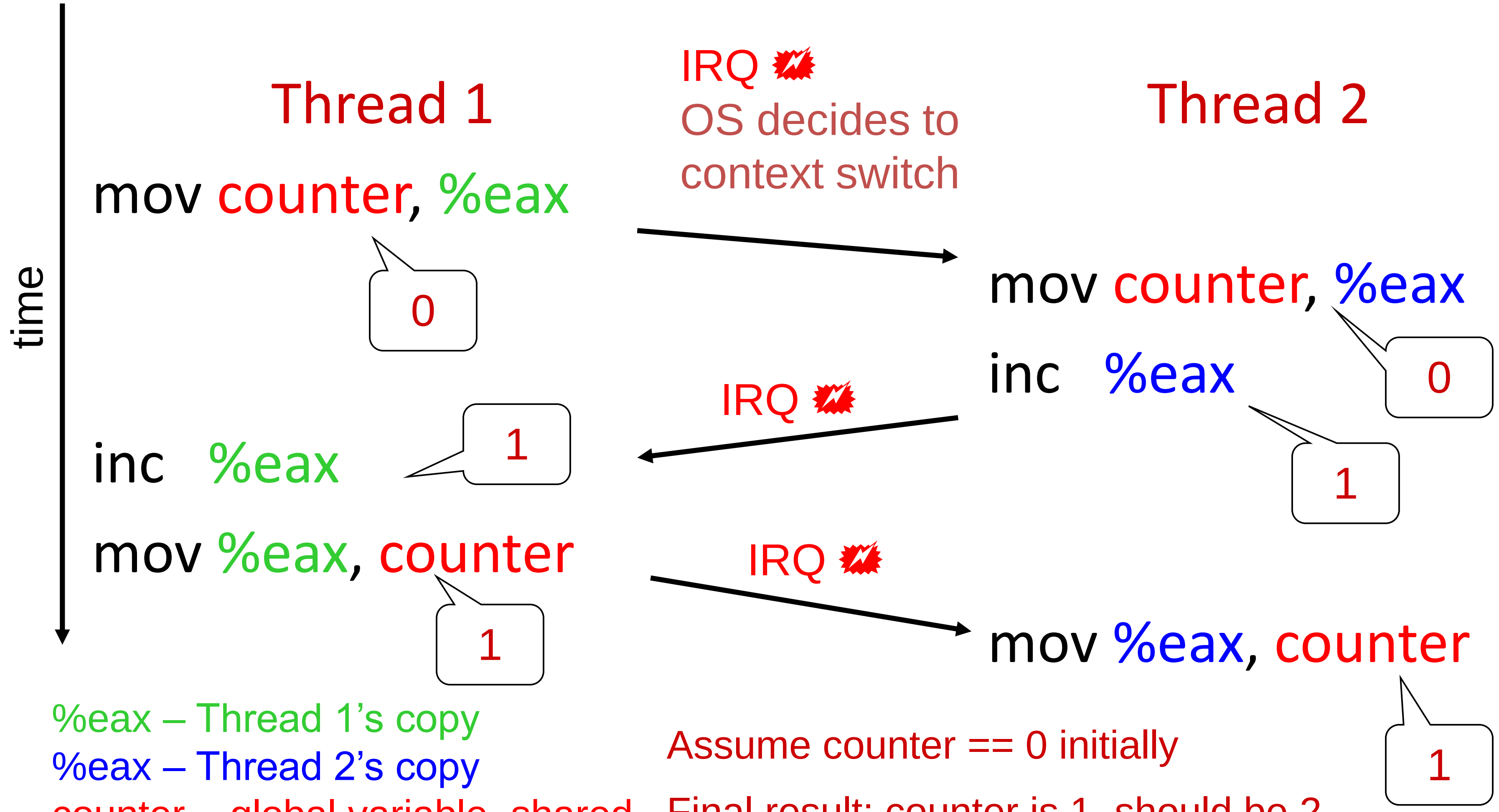
```
#include <stdio.h>
#include <assert.h>
#include <pthread.h>jjj
int counter = 0;
void *mythread (void *arg) {
    printf ("%s\n", (char *) arg);
    for(int i = 0; i < 1e7; i++)
        counter = counter + 1;
    printf ("%s: done\n", (char *) arg);
    return NULL;
}

int main (int argc, char *argv[]) {
    pthread_t p1, p2;
    int rc;
    printf("main: begin\n");
    pthread_create(&p1, NULL, mythread, "A");
    pthread_create(&p2, NULL, mythread, "B");
    // join waits for thre threads to finish
    pthread_join(p1, NULL);
    pthread_join(p2, NULL);
    printf("main: end\n");
    return 0;
}
```

Result

```
main: begin (counter = 0)
A
B
B: done
A: done
main: end (counter = 10313459)
```

```
main: begin (counter = 0)
A
B
A: done
B: done
main: end (counter = 10201521)
```

Race Conditions



%eax – Thread 1's copy
%eax – Thread 2's copy
counter – global variable, shared

Assume counter == 0 initially

Final result: counter is 1, should be 2

Race Conditions

- Race condition
  - Two threads “race” to execute code and update shared (dependent) data
  - Errors emerge based on the ordering of operations, and the scheduling of threads
  - Thus, errors are nondeterministic

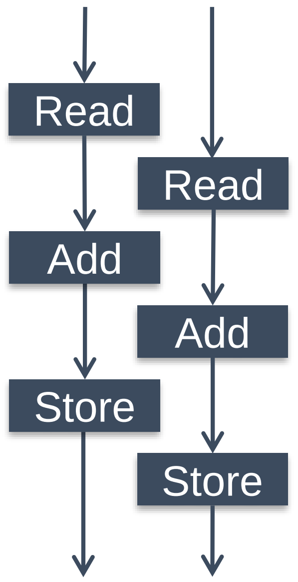
Critical Sections

- Classical definition of a critical section:
  - *"A piece of code that **accesses a shared resource** that must not be concurrently accessed by more than one thread of execution."*
  - Multiple threads executing critical section can result in a race condition.
  - Need to support **atomicity** for critical sections (**mutual exclusion**)

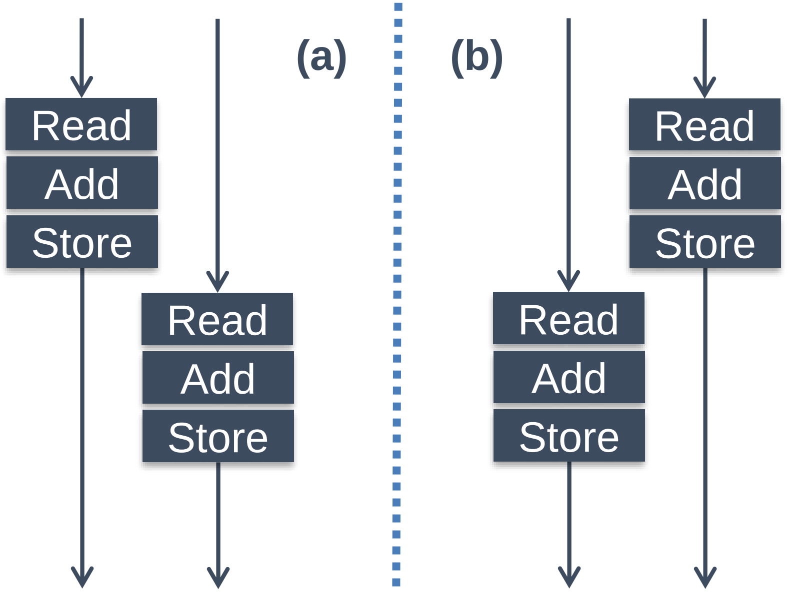
- Two problems
  - Code was not designed for concurrency
  - Shared resource (data) does not support concurrent access

Wish for Atomicity

- We need more powerful instructions
  - do exactly whatever we needed done in a single step → atomic
  - remove the possibility of an untimely interrupt



Interleaved Execution

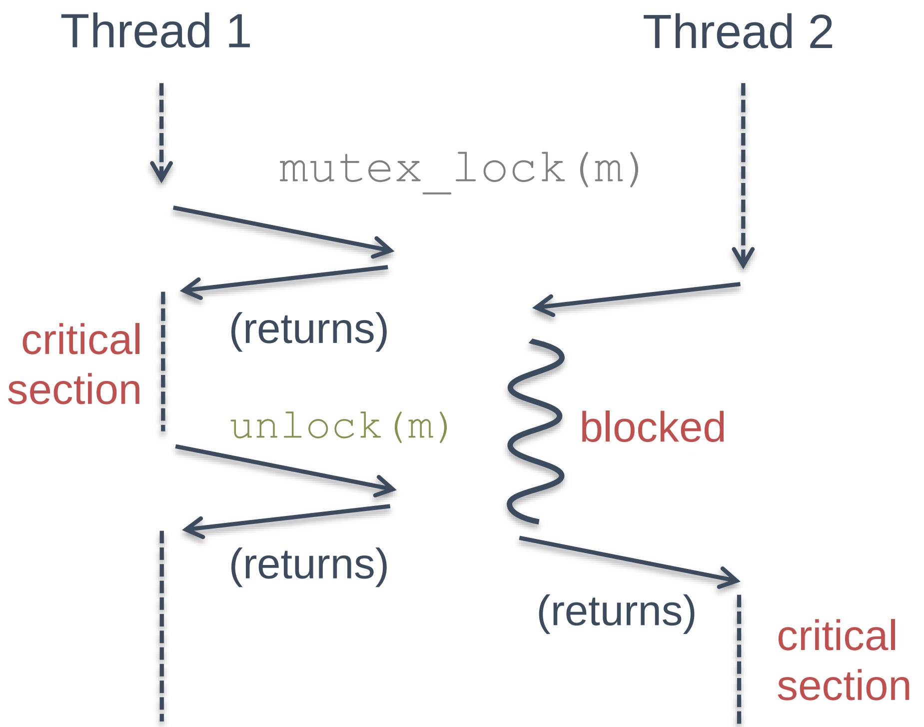


Non-Interleaved (Atomic) Execution

Mutexes for Atomicity

- Mutual exclusion lock (mutex) is a construct that can enforce atomicity in code

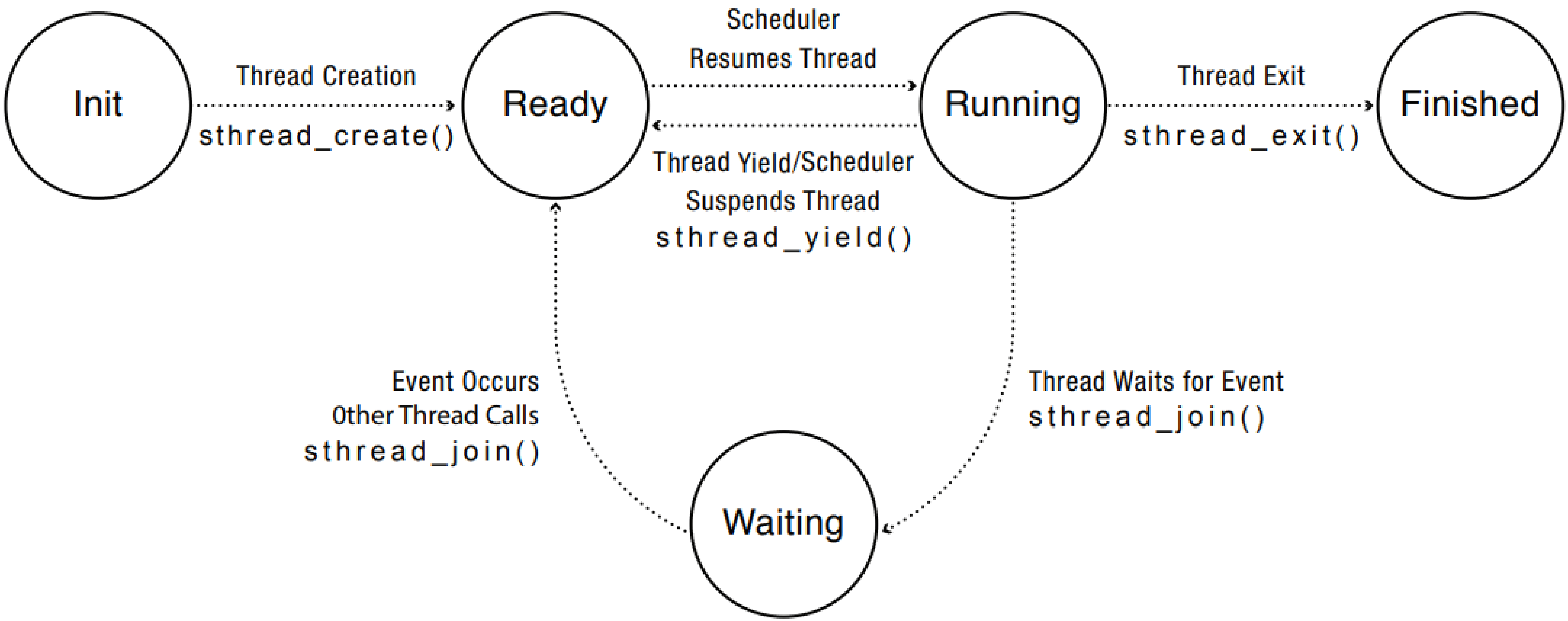
```
m = mutex_create();
…
mutex_lock(m);
// do some stuff
mutex_unlock(m);
```



One More Problem: Waiting For Another

- Another common interaction
  - One thread must wait for another to complete some action before it continues
  - e.g., when a process performs a disk I/O and is put to sleep; when the I/O completes, the process needs to be roused from its slumber so it can continue

Thread Lifecycle



27. Thread API

Thread Creation

- 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.

Thread Creation (Cont.)

- If `start_routine` instead requires 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: Creating a Thread

```
#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);
        …
}
```

Wait for a thread to complete

```
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 the pointer to that value.

Example: Waiting for Thread Completion

```c
#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: Waiting for Thread Completion (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;
    r.x = 1;
    r.y = 2;
    return (void *) &r;
}
```

  - The variable `r` is allocated in a stack space. So?

Example: Simpler Argument Passing to a Thread

- 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;
}
```

Locks

- Provide mutual exclusion to a **critical section**
  - Interface

```
int pthread_mutex_lock(pthread_mutex_t *mutex);
int pthread_mutex_unlock(pthread_mutex_t *mutex);
```

  - Usage (w/o *lock initialization* and *error check*)

```
pthread_mutex_t lock;
pthread_mutex_lock(&lock);
x = x + 1; // or whatever your critical section is
pthread_mutex_unlock(&lock);
```

    - No other thread holds the lock → the thread will acquire the lock and enter the critical section.
    - If another thread hold the lock → the thread will not return from the call until it has acquired the lock.

Condition Variables

- **Condition variables** are useful when some kind of signaling must take place between threads.

```
int pthread_cond_wait(pthread_cond_t *cond,
                      pthread_mutex_t *mutex);
int pthread_cond_signal(pthread_cond_t *cond);
```

  - `pthread_cond_wait`:
    - Put the calling thread to sleep.
    - Wait for some other thread to signal it.
  - `pthread_cond_signal`:
    - Unblock at least one of the threads that are blocked on the condition variable

Condition Variables (Cont.)

- The waiting thread **re-checks** the condition in a while loop, instead of a simple if statement.

```
pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t init = PTHREAD_COND_INITIALIZER;

pthread_mutex_lock(&lock);
while (initialized == 0)
        pthread_cond_wait(&init, &lock);
pthread_mutex_unlock(&lock);
```

  - Without rechecking, the waiting thread will continue thinking that the condition has changed *even though it has not*.

Condition Variables (Cont.)

- Don't ever to this.x
  - A thread calling wait routine:

```
while(initialized == 0)
        ; // spin
```

  - A thread calling signal routine:

```
initialized = 1;
```

  - It performs poorly in many cases. → just wastes CPU cycles.
  - It is error prone.

Implementing a Lock

```
do {
    while (flag);
    flag = true;
        critical section
    flag = false;
        remainder section
} while (true);
```

Problem?

```
flag = false;
```

```
do {
    while (flag);
    flag = true;
        critical section
    flag = false;
        remainder section
} while (true);
```

```
do {
    while (flag);
    flag = true;
        critical section
    flag = false;
        remainder section
} while (true);
```