Example

```c
#include <stdio.h>
#include <assert.h>
#include <pthread.h>
#include "common.h"
#include "common_threads.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 the threads to finish
    Pthread_join(p1, NULL);
    Pthread_join(p2, NULL);
    printf("main: end\n");
    return 0;
}
```

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

void *thread_fn(void *arg){
  long id = (long) arg;
  printf("Starting thread %ld\n", id);
  sleep(5);
  printf("Exiting thread %ld\n", id);
  return NULL;
}

int main(){
  pthread_t t1, t2;

  pthread_create(&t1, NULL, thread_fn, (void *)1);
  pthread_create(&t2, NULL, thread_fn, (void *)2);
  pthread_join(t1, NULL);
  pthread_join(t2, NULL);
  printf("Exiting main\n");
  return 0;
}
```

Thread Trace

main	Thread 1	Thread2
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"		

main	Thread 1	Thread2
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"		

main	Thread 1	Thread2
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"		

Example: With Shared data

```c
#include <stdio.h>
#include <pthread.h>
#include "common.h"
#include "common_threads.h"

static volatile int counter = 0;

// mythread()
//
// Simply adds 1 to counter repeatedly, in a loop
// No, this is not how you would add 10,000,000 to
// a counter, but it shows the problem nicely.
//
void *mythread(void *arg) {
    printf("%s: begin\n", (char *) arg);
    int i;
    for (i = 0; i < 1e7; i++) {
        counter = counter + 1;
    }
    printf("%s: done\n", (char *) arg);
    return NULL;
}

// main()
//
// Just launches two threads (pthread_create)
// and then waits for them (pthread_join)
//
int main(int argc, char *argv[]) {
    pthread_t p1, p2;
    printf("main: begin (counter = %d)\n", counter);
    Pthread_create(&p1, NULL, mythread, "A");
    Pthread_create(&p2, NULL, mythread, "B");

    // join waits for the threads to finish
    Pthread_join(p1, NULL);
    Pthread_join(p2, NULL);
    printf("main: done with both (counter = %d)\n",
        counter);
    return 0;
}
```

1M

T1

T2

c=0

c=?

Mutual Exclusion

Mutex

-lpthread

With shared data – What happens?

```
prompt> gcc -o main main.c -Wall -pthread; ./main
main: begin (counter = 0)
A: begin
B: begin
A: done
B: done
main: done with both (counter = 20000000)
```

What do we expect? Two threads, each increments counter by 10^7, so 2X10^7

```
prompt> ./main
main: begin (counter = 0)
A: begin
B: begin
A: done
B: done
main: done with both (counter = 19345221)
```

Sometimes, a lower value. Why?

objdump -d main

disassembler

Assembly code

OS	Thread 1	Thread 2	(after instruction) PC	eax	counter
	before critical section		100	0	50
	`mov 8049a1c,%eax`		105	**50**	50
	`add $0x1,%eax`		108	**51**	50
interrupt					
save T1					
restore T2			100	0	50
		`mov 8049a1c,%eax`	105	**50**	50
		`add $0x1,%eax`	108	**51**	50
		`mov %eax,8049a1c`	113	51	**51**
interrupt					
save T2					
restore T1			108	51	51
	`mov %eax,8049a1c`		113	51	**51**

52

Assembly code of

```
counter = counter + 1
```

SRC DE

```
100 mov    0x8049a1c, %eax
105 add    $0x1, %eax
108 mov    %eax, 0x8049a1c
```

Updated

Race condition & Synchronization

What just happened is called a race condition –Concurrent execution can lead to different results

- Critical section: portion of code that can lead to race conditions
- What we need: mutual exclusion–Only one thread should be executing critical section atany time
- What we need: atomicity of the critical section –The critical section should execute like one uninterruptible instruction
- How is it achieved? Locks

Who manages threads?

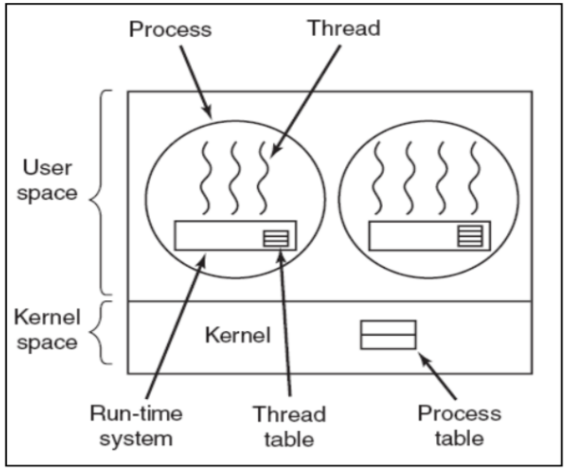
- Two strategies
  - User threads
    - Thread management done by user level thread library. Kernel knows nothing about the threads.
  - Kernel threads
    - Threads directly supported by the kernel.
    - Known as light weight processes.

User level threads

- Advantages:
  - Fast (really lightweight)

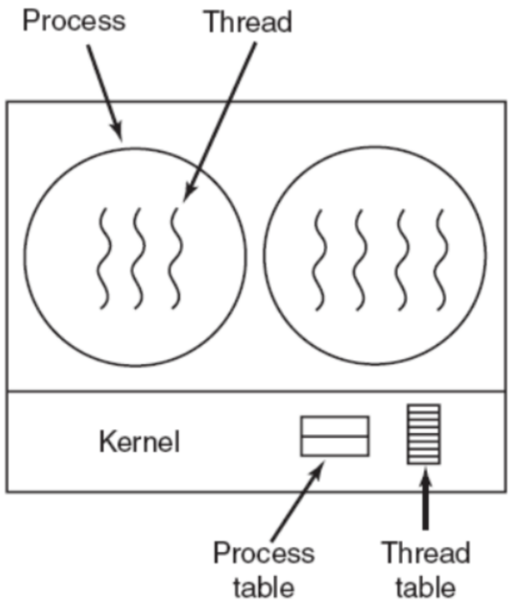
  (no system call to manage threads. The thread library does everything).
  - Can be implemented on an OS that does not support threading.
  - Switching is fast. No, switch from user to protected mode.

- Disadvantages:
  - Scheduling can be an issue. (Consider, one thread that is blocked on an IO and another runnable.)
  - Lack of coordination between kernel and threads. (A process with 1000 threads competes for a timeslice with a process having just 1 thread.)
  - Requires non-blocking system calls. (If one thread invokes a system call, all threads need to wait)



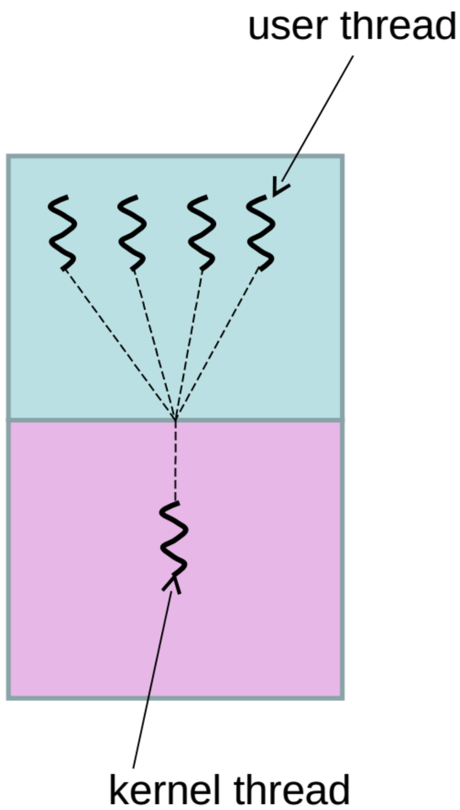
Kernel level threads

- **Advantages:**
  - Scheduler can decide to give more time to a process having large number of threads than process having small number of threads.
  - Kernel-level threads are especially good for applications that frequently block.
- **Disadvantages:**
  - The kernel-level threads are slow (they involve kernel invocations.)
  - Overheads in the kernel. (Since kernel must manage and schedule threads as well as processes. It require a full thread control block (TCB) for each thread to maintain information about threads.)



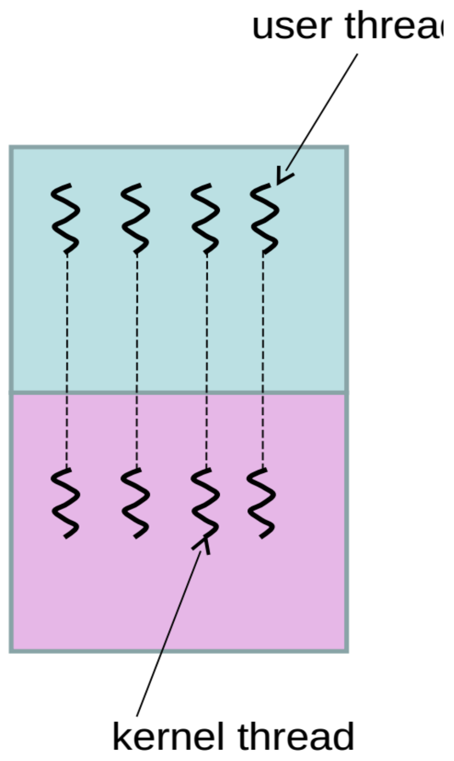
Many-to-One-Model

- Many user level threads map to a single kernel thread
- Pros:
  - Fast. No system calls to manage threads.
  - No mode change for switching threads
- Cons:
  - No parallel execution of threads. All threads block when one has a system call.
  - Not suited for multi-processor systems.



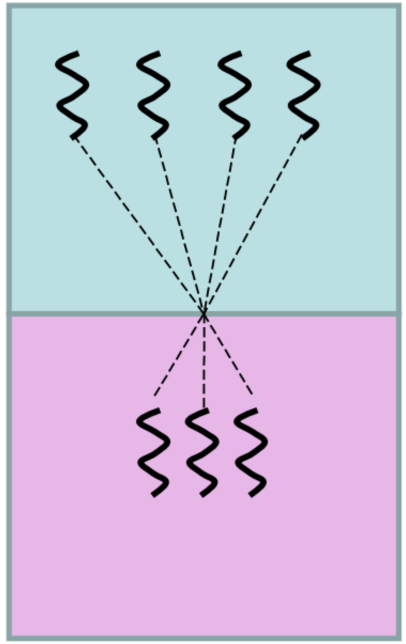
One-to-One Model

- Each user thread associated with one kernel thread.
- Pros.
  - Better suited for multiprocessor environments.
  - When one thread blocks, the other threads can continue to execute.
- Cons.
  - Expensive. Kernel is involved.



Many-to-many

- Many user threads mapped to many kernel threads
  - Supported by some unix and windows versions
- Pros: flexible
  - OS creates kernel threads as required
  - Process creates user threads as needed
- Cons: Complex
  - Double management



Summary

- A **critical section** is a piece of code that accesses a *shared* resource, usually a variable or data structure.
- A **race condition** (or **data race** [NM92]) arises if multiple threads of execution enter the critical section at roughly the same time; both attempt to update the shared data structure, leading to a surprising (and perhaps undesirable) outcome.
- An **indeterminate** program consists of one or more race conditions; the output of the program varies from run to run, depending on which threads ran when. The outcome is thus not **deterministic**, something we usually expect from computer systems.
- To avoid these problems, threads should use some kind of **mutual exclusion** primitives; doing so guarantees that only a single thread ever enters a critical section, thus avoiding races, and resulting in deterministic program outputs.

Any solution should satisfy the following requirements

– Mutual Exclusion : No more than one process in critical section at a given time

– Progress : When no process is in the critical section, any process that requests entry into the critical section must be permitted without any delay

– No starvation (bounded wait): There is an upper bound on the number of times a process enters the critical section, while another is waiting

Locks

Consider update of shared variable **`balance = balance + 1`**

We can use a special lock variable to protect it

```
lock_t mutex; // some globally-allocated lock 'mutex'
...
lock(&mutex);
balance = balance + 1;
unlock(&mutex);
```

CS

All threads accessing a critical section share a lock

Waiting (Blocked)

- One threads succeeds in locking – owner of lock
- Other threads that try to lock cannot proceed further until lock is released by the owner
- Pthreads library in Linux provides such locks

T_1

program 0

```
{
  *
  *
lock(L)
counter++
unlock(L)
  *
}
```

shared variable

```
int counter=5;
lock_t L;
```

T_2

program 1

```
{
  *
  *
lock(L)
counter--
unlock(L)
  *
}
```

- lock(L) acquire lock L exclusively
  - Only the process with L can access the critical section
- unlock(L) : release exclusive access to lock L
  - Permitting other processes to access the critical section

Building a Lock

Goals of a lock implementation

– Mutual exclusion

– Fairness: all threads should eventually get the lock, and no thread should starve

– Low overhead: acquiring, releasing, and waiting for lock should not consume too many resources

- Implementation of locks are needed for both userspace programs (e.g., pthreads library) and kernel code
- Implementing locks needs support from hardware and OS

Using Interrupts

Process 1

```
        while(1){
lock -->    disable interrupts ()
            critical section
unlock -->  enable interrupts ()
             other code
        }
```

Process 2

```
while(1){
    disable interrupts ()
    critical section
    enable interrupts ()
     other code
}
```

- Simple
  - When interrupts are disabled, context switches won’t happen
- Requires privileges
  - User processes generally cannot disable interrupts
- Not suited for multicore systems

Is disabling interrupts enough ?

Is this enough?
• No, not always!
• Many issues here:
– Disabling interrupts is a privileged instruction and program can misuse it (e.g., run forever)
– Will not work on multiprocessor systems, since another thread on another core can enter critical section
• This technique is used to implement locks on single processor systems inside the OS
– Need better solution for other situations

Implementation Attempt -1

- Lock: spin on a flag variable until it is unset, then set it to acquire lock
- Unlock: unset flag variable

```
typedef struct __lock_t { int flag; } lock_t;

void init(lock_t *mutex) {
    // 0 -> lock is available, 1 -> held
    mutex->flag = 0;
}

void lock(lock_t *mutex) {
    while (mutex->flag == 1) // TEST the flag
        ; // spin-wait (do nothing)
    mutex->flag = 1;         // now SET it!
}

void unlock(lock_t *mutex) {
    mutex->flag = 0;
}
```

- Thread 1 spins, lock is released, ends spin
- Thread 1 interrupted just before setting flag
- Race condition has moved to the lock acquisition code!

Thread 1	Thread 2
call `lock()`	
while (flag == 1)	
interrupt: switch to Thread 2	
	call `lock()`
	while (flag == 1)
	flag = 1;
	interrupt: switch to Thread 1
flag = 1; // set flag to 1 (too!)	