```c
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;
}
```

```
main: begin (counter = 0)
A: begin
B: begin
A: done
B: done
main: done with both (counter = 20000000)
```

```
main: begin (counter = 0)
A: begin
B: begin
A: done
B: done
main: done with both (counter = 19345221)
```

```
main: begin (counter = 0)
A: begin
B: begin
A: done
B: done
main: done with both (counter = 19221041)
```

```
mov 0x8049a1c, %eax
add $0x1, %eax
mov %eax, 0x8049a1c
```

- this can be a problem when shared data is involved

Locks

- a *lock* has two states
  - *locked* (or *held* or *acquired*)
  - *unlocked* (or *free* or *available*)
- a particular lock can only be held by one thread at a time
- locks support mutual exclusion to protect critical sections
  - a thread entering a critical section acquires a lock
  - another thread attempting to acquire the same lock must wait
  - the thread releases the lock when it exits the critical section, allowing a waiting thread to acquire it

Using pthread Locks

```c
pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER;

Pthread_mutex_lock(&lock); // wrapper; exits on failure
balance = balance + 1;
Pthread_mutex_unlock(&lock);
```

use separate lock variables for more fine-grained control over different things needing protection

- `PTHREAD_MUTEX_INITIALIZER` initializes a mutex with default attribute values
  - *mutex* is the pthreads terminology for a lock
  - does not do error checking
  - should only be used for static mutexes
    - *static* means that the lifetime of the variable is the entire duration of the program

```c
#include <pthread.h>

int pthread_mutex_init(pthread_mutex_t *mutex,
    const pthread_mutexattr_t *attr);
int pthread_mutex_destroy(pthread_mutex_t *mutex);
```

- create and destroy mutexes
  - allows setting of attributes
  - does error checking

Evaluating Lock Implementations

- a working lock –
  - must provide *mutual exclusion*
    - holding a lock must successfully prevents other threads from entering a section of code
  - should be *fair*
    - all threads trying to acquire a lock should have an equal chance of getting it when it is free
    - no thread should starve
  - should have minimal impacts on *performance*
    - more time spent managing a lock means less time actually running processes

Lock Implementations

- controlling interrupts
- software-only
  - simple flag
  - Peterson's algorithm
- hardware supported
  - test-and-set
  - compare-and-swap
  - load-linked, store-conditional
  - fetch-and-add

Controlling Interrupts

- on single-processor systems, race conditions are a problem entirely because of interrupts
  - without an interrupt, there is no context switch to another process

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**

→ disable interrupts before entering a critical section, re-enable on exit

Controlling Interrupts

- strategy: disable interrupts before entering a critical section, re-enable on exit
- challenges
  - doesn't work for multiprocessor systems
    - multiple threads can run simultaneously on different CPUs
  - turning off interrupts is a privileged operation but locks are needed by user programs
    - a badly behaved thread could turn interrupts off and then go into an infinite loop
  - turning off interrupts for long periods of time risks losing interrupts that arrive in the interim
    - timer interrupts
    - completed I/O
- as a result, turning off interrupts is limited to only within the kernel and for very limited situations

A Simple Flag

- idea: the lock is represented by a boolean variable
  - set the variable to 1 if the lock is held, 0 if the lock is free
  - a thread attempting to acquire a held lock *spin-waits* until the lock becomes available
- effectiveness: poor
  - does not ensure mutual exclusion
    - a thread can be interrupted between seeing that the lock is available and acquiring it, during which time another thread can grab the lock
  - spin-waiting ties up the CPU doing nothing productive

```
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	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!)	

Peterson's Algorithm

```
int flag[2];
int turn;

void init() {
    // indicate you intend to hold the lock w/ 'flag'
    flag[0] = flag[1] = 0;
    // whose turn is it? (thread 0 or 1)
    turn = 0;
}
void lock() {
    // 'self' is the thread ID of caller
    flag[self] = 1;
    // make it other thread's turn
    turn = 1 - self;
    while ((flag[1-self] == 1) && (turn == 1 - self))
        ; // spin-wait while it's not your turn
}
void unlock() {
    // simply undo your intent
    flag[self] = 0;
}
```

the only shared variable involved is `turn` (each thread accesses a different slot of `flag`) and load and store are atomic – no chance for the other thread to grab the lock between this thread seeing it is available and taking it

- achieves mutual exclusion, assuming load and store are atomic operations (and assuming a certain memory consistency model)
  - (this assumption was valid with older hardware, but modern systems have more relaxed handling of memory consistency)

Test-And-Set

- a *test-and-set* (or *atomic exchange*) instruction fetches a value and replaces it with a new value in a single step

```
int TestAndSet(int *old_ptr, int new) {
    int old = *old_ptr; // fetch old value at old_ptr
    *old_ptr = new;     // store 'new' into old_ptr
    return old;         // return the old value
}
```

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

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

void lock(lock_t *lock) {
    while (TestAndSet(&lock->flag, 1) == 1)
        ; // spin-wait (do nothing)
}

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

atomic test-and-set ensures no chance of another thread grabbing the lock between you seeing it is available and taking it

spin lock implemented with test-and-set