Synchronization
Software-based Mutual Exclusion

Would a software-based solution work?
Two Flag and Turn Mutual Exclusion

int owner[2]={false, false};
int turn;
...
owner[my_process_id] = true;
turn = other_process_id;
while (owner[other_process_id] and
    turn == other_process_id) {
    /* wait your turn */
}

access shared variables;
owner[my_process_id] = false;
...
Two Flag and Turn Mutual Exclusion

```
int owner[2] = {false, false};
int turn;
...
owner[my_process_id] = true;
turn = other_process_id;
while (owner[other_process_id] and
       turn == other_process_id) {
    /* wait your turn */
}

// access shared variables;
owner[my_process_id] = false;
...```

Peterson's Solution
Are we done?

- Peterson’s algorithm works
  - It guarantees mutual exclusion
  - no thread can monopolize use of shared resource, because each thread has to give an opportunity to the other thread by setting “turn=other_process_id” before each attempt to enter its critical section

- But....
In case you test Peterson sol.

- If everything worked...

  ```
  $ ./peterson
  Final value: 100000
  ```

  Output

  ![](6)

  ```
  mcaccamo$ ./peterson
  Final value: 100000
  mcaccamo$ ./peterson
  Final value: 100000
  mcaccamo$ ./peterson
  Final value: 99999
  mcaccamo$ ./peterson
  Final value: 100000
  ```

I am confused…
The perilous landscape of relaxed memory multicores

Required assumptions for correctness of Peterson’s alg.:
- We consider only two threads
- [Topic for computer architecture class] CPU does not perform memory operations in an out-of-order fashion. The programmer needs to rely on strict ordering for the memory operations within a thread.
  - Guess what… x86 performs the following re-ordering:
    
    Loads may be reordered with older stores to different locations

  ➞ Peterson’s algorithm is broken on x86. Test it yourself…..

Problem: software-based solutions are slow
  ➞ Solution: leverage CPU atomic operations like test-and-set
Hardware support for mutual exclusion

- We need **hardware support**: an **atomic** operation like test-and-set is needed to implement mutual exclusion.
int TestAndSet(int* plock) atomic {
    int initial;
    initial = *plock;
    *plock = 1;
    return initial;
}

atomic = executed in a single shot without any interruption

Note: this pseudo-code only serves to help explain the behavior of test-and-set; atomicity requires explicit hardware support and hence can't be implemented as a simple C function.
volatile long lock = 0; // lock is initially set to free

// Calling TestAndSet(&lock) sets lock to 1 and returns the old value of lock. 
// So, if lock is zero, then TestAndSet(&lock) returns zero and sets lock to 
// one. This means the lock has been successfully acquired. On the other hand, 
// if the lock had already been set to one by another process or thread, 
// then 1 would be returned. This would indicate to the caller that the lock 
// is already being held by another process or thread.

// This code is gcc/linux/intel x86 specific.
long TestAndSet (volatile long * lock) {
    long retval;
    // Atomically exchange value of register %0 with lock. The 
    // atomicity of xchg is what guarantees that at most one 
    // process or thread can be holding the lock at any point in time.
    __asm__ __volatile__ (
        "movl $0x1, %0 \n"
        "xchg %0, (%1) \n"
        : "=&r"(retval) : "r"(lock) : "memory" );

    return retval;
}
TestAndSet function

- **int TestAndSet(int *lock)**
  - **Pros:**
    - Very fast to entry to unlocked region
    - Guarantees safety & progress
  - **Cons:**
    - Wastes CPU cycles if used with busy waiting (spin lock)
    - Extremely high memory (i.e., bus) traffic if used with busy waiting

- TestAndSet can be used to implement higher-level synchronization constructs.
Back to the counter example

compile with gcc -m32 -lpthread -o test_test.c

```c
#include <stdio.h>
#include <pthread.h>
#define NUM_THREADS 2

// lock is initially set to free
volatile long lock = 0;

int cnt = 0;

void * worker( void *ptr ) {
    int i;
    for (i = 0; i < 50000; i++) {
        // spin until it locks
        while(TestAndSet(&lock)) {};
        // locked in mutual exclusion
        cnt++;
        lock = 0; // lock is released
    }
    pthread_exit(NULL);
}

int main(void) {
    pthread_t threads[NUM_THREADS];
    int i, res;
    for (i=0; i < NUM_THREADS; i++) {
        res=pthread_create(&threads[i],
         NULL, worker, NULL);
    }
    for (i=0; i < NUM_THREADS; i++) {
        res=pthread_join(threads[i],NULL);
    }
    /* Print result */
    printf("Final value: %d\n", cnt);
}
```

Copyright ©: University of Illinois CS 241 Staff
Simple implementation of P(s)

- P(s) and V(s) need to execute atomically.
- How can I implement them?
- **Solution**: use `TestAndSet`

```c
#include <sched.h>
typedef struct SEMAPHORE {
    volatile long lock;
    volatile long sem;
} semaphore;

void P(semaphore *p) {
    while(1) {
        while(TestAndSet(&p->lock))
            sched_yield();
        // locked in mutual exclusion
        if (p->sem > 0) {
            p->sem--;
            p->lock = 0; // lock is released
            break; // entering crit. sec.
        }
        p->lock = 0; // lock is released
        sched_yield(); // relinquish CPU
    }
}
```
Simple implementation of V(s)

- How can I implement V(s)?
- **Solution**: use TestAndSet!

```c
void V(semaphore *p) {
    while(TestAndSet(&p->lock))
        sched_yield();

    // locked in mutual exclusion
    p->sem++;
    p->lock = 0;  // lock is released
}
```
Back to the counter example: solution with primitives P & V

```c
#include <stdio.h>
#include <pthread.h>
#include <sched.h>
#define NUM_THREADS 2

semaphore s = {.lock =0, .sem =1};
int cnt = 0;

void * worker( void *ptr ) {
    int i;
    for (i = 0; i < 50000; i++) {
        P(&s);
        cnt++;
        // critical section
        V(&s);
    }
    pthread_exit(NULL);
}

int main(void) {
    pthread_t threads[NUM_THREADS];
    int i, res;
    for (i=0; i < NUM_THREADS; i++) {
        res=pthread_create(&threads[i],
                           NULL, worker, NULL);
    }
    for (i=0; i < NUM_THREADS; i++) {
        res=pthread_join(threads[i],NULL);
    }
    /* Print result */
    printf("Final value: %d\n", cnt);
}
```

Copyright ©: University of Illinois CS 241 Staff
Synchronization Primatives

- **Pthread mutex**
  - Permits only one thread to execute a critical section

- **PosixSemaphore**
  - Permits up to a limited number of threads to execute a critical section

- **Pthread condition variable**
  - Wait for event
  - Signal occurrence of event to one waiting thread
  - Broadcast occurrence of event to all waiting threads
Creating a mutex

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

- Initialize a pthread mutex: the mutex is initially unlocked
- Returns
  - 0 on success
  - Error number on failure
    - **EAGAIN**: The system lacked the necessary resources; **ENOMEM**: Insufficient memory; **EPERM**: Caller does not have privileges; **EBUSY**: An attempt to re-initialise a mutex; **EINVAL**: The value specified by attr is invalid

- Parameters
  - **mutex**: Target mutex
  - **attr**:
    - NULL: the default mutex attributes are used
    - Non-NULL: initializes with specified attributes
Creating a mutex

- **Default attributes**
  - Use `PTHREAD_MUTEX_INITIALIZER`
    - Statically allocated
    - Equivalent to dynamic initialization by a call to `pthread_mutex_init()` with parameter `attr` specified as NULL
    - No error checks are performed
Destroying a mutex

```c
int pthread_mutex_destroy(pthread_mutex_t *mutex);
```

- Destroy a pthread mutex
- Returns
  - 0 on success
  - Error number on failure
    - `EBUSY`: Mutex is locked by a thread; `EINVAL`: The value specified by mutex is invalid
- Parameters
  - `mutex`: Target mutex
Locking/unlocking a mutex

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

- **Returns**
  - 0 on success
  - Error number on failure
    - **EBUSY**: Mutex is already locked; **EINVAL**: The value specified by mutex is invalid; **EDEADLK**: The current thread already owns the mutex; **EPERM**: The current thread does not hold a lock on mutex.

⇒ If a signal is delivered to a thread while that thread is waiting for a mutex, when the signal handler returns, the wait resumes. `pthread_mutex_lock()` does not return **EINVAL**!
Back to the counter example: solution with pthread_mutex

```c
#include <stdio.h>
#include <pthread.h>
#define NUM_THREADS 2

pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
int cnt = 0;

void * worker( void *ptr ) {  
    int i;
    for (i = 0; i < 50000; i++) {  
        pthread_mutex_lock(&mutex);
        cnt++;
        // critical section
        pthread_mutex_unlock(&mutex);
    }
    pthread_exit(NULL);
}

int main(void) {
    pthread_t threads[NUM_THREADS];
    int i, res;
    for (i=0; i < NUM_THREADS; i++) {
        res=pthread_create(&threads[i],
                          NULL, worker, NULL);
    }
    for (i=0; i < NUM_THREADS; i++) {
        res=pthread_join(threads[i],NULL);
    }
    /* Print result */
    printf("Final value: %d\n", cnt);
}
```