LECTURE 14
FEB 5, 2024

RECAP BEGIN: Hardware atomic instructions

Very hard to ensure atomicity only in software

- Modern architectures provide hardware atomic instructions
- Example of an atomic instruction: test-and-set

–Update a variable and return old value, all in one hardware instruction

```
1  int TestAndSet(int *old_ptr, int new) {
2      int old = *old_ptr; // fetch old value at old_ptr
3      *old_ptr = new;     // store 'new' into old_ptr
4      return old;         // return the old value
5  }
```

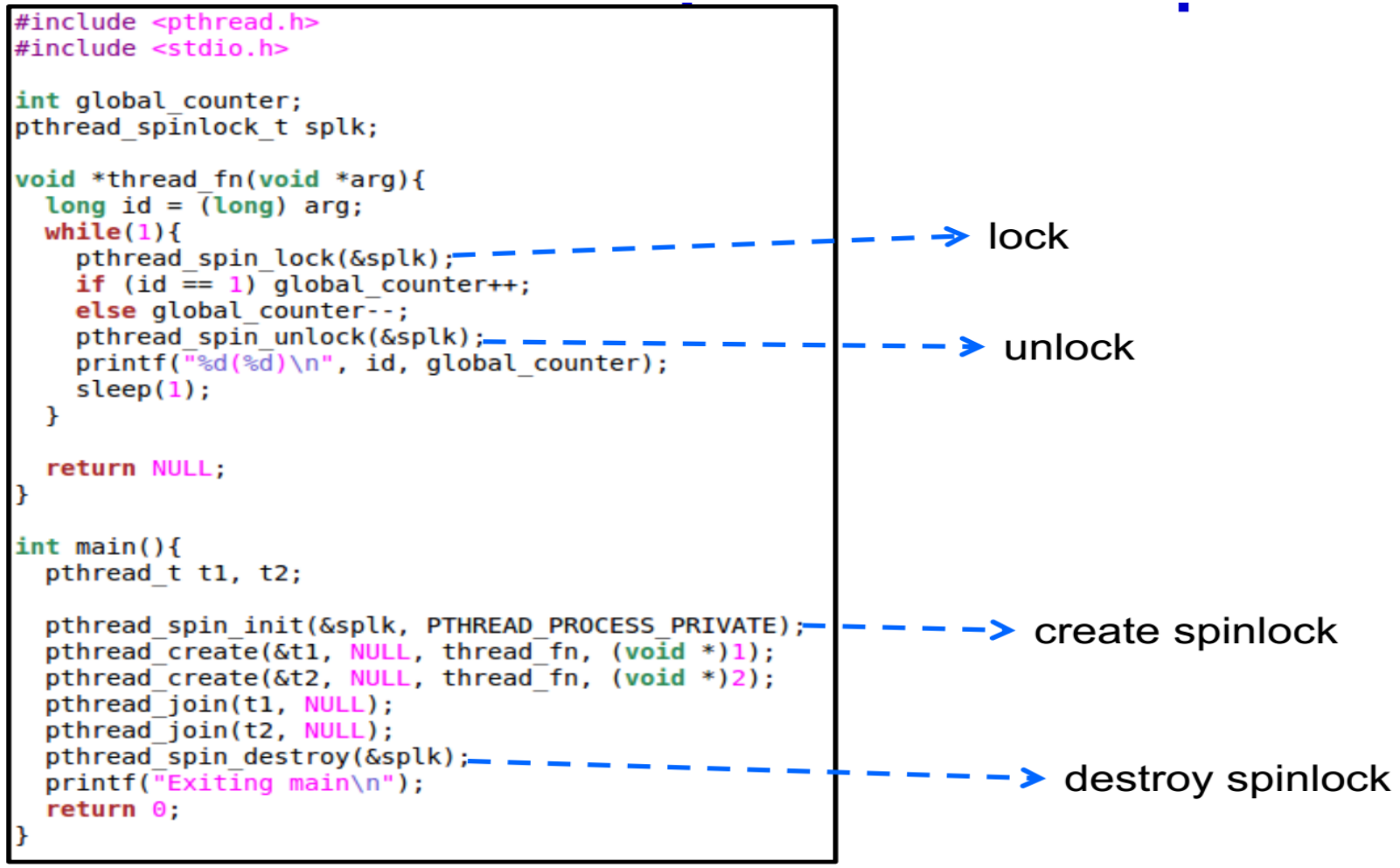
Simple lock using test-and-set

If TestAndSet(flag,1) returns 1, it means the lock is held by someone else, so wait busily
•This lock is called a spinlock–spins until lock is acquired

```
1  typedef struct __lock_t {
2      int flag;
3  } lock_t;
4
5  void init(lock_t *lock) {
6      // 0: lock is available, 1: lock is held
7      lock->flag = 0;
8  }
9
10 void lock(lock_t *lock) {
11     while (TestAndSet(&lock->flag, 1) == 1)
12         ; // spin-wait (do nothing)
13 }
14
15 void unlock(lock_t *lock) {
16     lock->flag = 0;
17 }
```

Pthread example

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

int global_counter;
pthread_spinlock_t splk;

void *thread_fn(void *arg){
  long id = (long) arg;
  while(1){
    pthread_spin_lock(&splk);
    if (id == 1) global_counter++;
    else global_counter--;
    pthread_spin_unlock(&splk);
    printf("%d(%d)\n", id, global_counter);
    sleep(1);
  }

  return NULL;
}

int main(){
  pthread_t t1, t2;

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



When to use Spinlocks?

- Characteristic : busy waiting
  - Useful for short critical sections, where much CPU time is not wasted waiting
    - eg. To increment a counter, access an array element, etc.
  - Not useful, when the period of wait is unpredictable or will take a long time
    - eg. Not good to read page from disk.
    - Use mutex instead (…mutex)

Alternative to Spinning

- Alternative to spinlock: a (sleeping) mutex
- Instead of spinning for a lock, a contending thread could simply give up the CPU and check back later – yield() moves thread from running to ready state

```
1   void init() {
2       flag = 0;
3   }
4
5   void lock() {
6       while (TestAndSet(&flag, 1) == 1)
7           yield(); // give up the CPU
8   }
9
10  void unlock() {
11      flag = 0;
12  }
```

Mutexes

- Can we do better than busy waiting?
  - If critical section is locked then yield CPU
    - Go to a SLEEP state
  - While unlocking, wake up sleeping process

```
int xchg(addr, value){
  %eax = value
  xchg %eax, (addr)
}

void lock(int *locked){
  while(1){
    if(xchg(locked, 1) == 0)
      break;
    else
      sleep();
  }
}

void unlock(int *locked){
    locked = 0;
    wakeup();
}
```

Thundering Herd Problem

- A large number of processes wake up (almost simultaneously) when the event occurs.
  - All waiting processes wake up
  - Leading to several context switches
  - All processes go back to sleep except for one, which gets the critical section
    - Large number of context switches
    - Could lead to starvation

```
int xchg(addr, value){
  %eax = value
  xchg %eax, (addr)
}

void lock(int *locked){
  while(1){
    if(xchg(locked, 1) == 0)
      break;
    else
      sleep();
  }
}

void unlock(int *locked){
    locked = 0;
    wakeup();
}
```

RECAP ENDS

- The Solution
  - When entering critical section, push into a queue before blocking
  - When exiting critical section, wake up only the first process in the queue

```
int xchg(addr, value){
  %eax = value
  xchg %eax, (addr)
}

void lock(int *locked){
  while(1){
    if(xchg(locked, 1) == 0)
      break;
    else{
      // add this process to Queue
       sleep();
    }
  }
}

void unlock(int *locked){
    locked = 0;
    // remove process P from queue
    wakeup(P)
}
```

Locks and Priorities

- What happens when a high priority task requests a lock, while a low priority task is in the critical section
  - Priority Inversion
  - Possible solution
    - Priority Inheritance

Spinlock vs Sleeping Mutex

- Most userspace lock implementations are of the sleeping mutex kind
  - CPU wasted by spinning contending threads
  - More so if a thread holds spinlock and blocks for long
- Locks inside the OS are always spinlocks – Why? Who will the OS yield to?
- When OS acquires a spinlock:
  - It must disable interrupts (on that processor core) while the lock is held. Why? An interrupt handler could request the same lock, and spin for it forever.
  - It must not perform any blocking operation–never go to sleep with a locked spinlock!
- In general, use spinlocks with care, and release as soon as possible

How locks should be used?

A lock should be acquired before accessing any variable or data structure that is shared between multiple threads of a process–“Thread-safe” data structures

•All shared kernel data structures must also be accessed only after locking

•Coarse-grained vs. fine-grained locking: one biglock for all shared data vs. separate locks

–Fine-grained allows more parallelism

–Multiple fine-grained locks may be harder to manage

•OS only provides locks, correct locking discipline is left to the user

Homework

The program introduces two employees competing for the "employee of the day" title, and the glory that comes with it.

To simulate that in a rapid pace, the program employs 3 threads: one that promotes Danny to "employee of the day", one that promotes Moshe to that situation, and a third thread that makes sure that the employee of the day's contents is consistent (i.e. contains exactly the data of one employee).

Two versions of the program are given to you. One that uses a mutex, and one that does not.

In this HW, you will report the output of each program and will analyze and write the differences between both the programs

Link : https://drive.google.com/drive/folders/1RrLsvKJJjkjKTefnqMxQ1vdozaLMHIzS

Conditional Variables

- Locks allow one type of synchronization between threads
  - Mutual exclusion
- Another common requirement in multi-threaded applications

There are many cases where a thread wishes to check whether a condition is true before continuing its execution –waiting and signaling

  –E.g., Thread T1 wants to continue only after T2 has finished some task

- Can accomplish this by busy-waiting on some variable, but inefficient
- Need a new synchronization primitive: **condition variables**

```
void *child(void *arg) {
    printf("child\n");
    // XXX how to indicate we are done?
    return NULL;
}

int main(int argc, char *argv[]) {
    printf("parent: begin\n");
    pthread_t c;
    Pthread_create(&c, NULL, child, NULL); // create child
    // XXX how to wait for child?
    printf("parent: end\n");
    return 0;
}
```

parent thread might wish to check whether a child thread has completed before continuing

```
parent: begin
child
parent: end
```

```
volatile int done = 0;

void *child(void *arg) {
    printf("child\n");
    done = 1;
    return NULL;
}

int main(int argc, char *argv[]) {
    printf("parent: begin\n");
    pthread_t c;
    Pthread_create(&c, NULL, child, NULL); // create child
    while (done == 0)
        ; // spin
    printf("parent: end\n");
    return 0;
}
```

Conditon Variable

A condition variable (CV) is a queue that a thread can put itself into when waiting on some condition

•Another thread that makes the condition true can signal the CV to wake up a waiting thread

•Pthreads provides CV for user programs –OS has a similar functionality of wait/signal for kernel threads

•Signal wakes up one thread, signal broadcast wakes up all waiting threads

`pthread_cond_t c` which declares `c` as a condition variable

Wait() and Signal()

A condition variable has two operations associated with it: **`wait()`** and **`signal()`**.

The **`wait()`** call is executed when a thread wishes to put itself to sleep.

The **`signal()`** call is executed when a thread has changed something in the program and thus wants to wake a sleeping thread waiting on this condition.

```
pthread_cond_wait(pthread_cond_t *c, pthread_mutex_t *m);
pthread_cond_signal(pthread_cond_t *c);
```

Example

```
int done  = 0;
pthread_mutex_t m = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t c  = PTHREAD_COND_INITIALIZER;

void thr_exit() {
    Pthread_mutex_lock(&m);
    done = 1;
    Pthread_cond_signal(&c);
    Pthread_mutex_unlock(&m);
}

void *child(void *arg) {
    printf("child\n");
    thr_exit();
    return NULL;
}

void thr_join() {
    Pthread_mutex_lock(&m);
    while (done == 0)
        Pthread_cond_wait(&c, &m);
    Pthread_mutex_unlock(&m);
}

int main(int argc, char *argv[]) {
    printf("parent: begin\n");
    pthread_t p;
    Pthread_create(&p, NULL, child, NULL);
    thr_join();
    printf("parent: end\n");
    return 0;
}
```

Why while loop?

In the example code, why do we check condition before calling wait?–In case the child has already run and done is true, then no need to wait

- Why check condition with “while” loop and not “if”?

–To avoid corner cases of thread being woken up even when condition not true (may be an issue with some implementations)

Why use lock when calling wait?

What if no lock is held when calling wait/signal?

- Race condition: missed wakeup

–Parent checks done to be 0, decides to sleep, gets interrupted

–Child runs, sets done to 1, signals, but no one sleeping yet

–Parent now resumes and goes to sleep forever

- Lock must be held when calling wait and signal with CV
- The wait function releases the lock before putting thread to sleep, so lock is available for signaling thread

```
void thr_exit() {
    done = 1;
    Pthread_cond_signal(&c);
}

void thr_join() {
    if (done == 0)
        Pthread_cond_wait(&c);
}
```

The responsibility of wait() is to release the lock and put the calling thread to sleep (atomically).

when the thread wakes up (after some other thread has signaled it), it must re-acquire the lock before returning to the caller.