Week 3

Synchronization Primitives

Oana Balmau
January 19, 2023

Overview of Assignment 1

- Presented by Jiaxuan.
- Recording will be posted on MyCourses.

Synchronization

Build higher-level synchronization primitives in OS

- Operations that ensure correct ordering of instructions across threads

Motivation: Build them once and get them right

Software	**Monitors** ✅ **Locks (mutex)** **Semaphores** **Condition Variables**
Hardware	Loads Stores Test&Set Disable Interrupts

Condition Variables

- Used when thread A needs to wait for an event done by thread B
- A waits until a certain condition is true
  - First test condition,
  - If condition not true, call `pthread_cond_wait()`
    - A blocks until condition is true.

- At some point B makes the condition true
  - Then B calls `pthread_cond_signal()`, which unblocks A.

Advantage of Condition Variable of Locks

- A waits until a certain condition is true
  - A goes to sleep here.
  - Does not keep CPU busy.

Condition variables use (incorrect)

Thread A

```
x = f (a , b) ;
if (x < 0 || x > 9)
  pthread_cond_wait (&cv);
```

Thread B

```
//change a and b;
x = f (a , b) ;
if (x >= 0 && x <= 9)
  pthread_cond_signal (&cv);
```

Find the data race.

Condition variables use (incorrect)

Thread A

```
x = f ( a , b ) ;
if ( x < 0 || x > 9)
  pthread_cond_wait (&cv);
```

Interrupt

Thread B

```
//change a and b;
x = f (a , b) ;
if ( x >= 0 && x <= 9)
  pthread_cond_signal (&cv);
```

Condition variables use (incorrect)

Thread A

```
x = f ( a , b ) ;
if ( x < 0 || x > 9)
  pthread_cond_wait (&cv);
```

Interrupt

Thread B

```
//change a and b;
x = f (a , b) ;
if ( x >= 0 && x <= 9)
  pthread_cond_signal (&cv);
```

Condition variables use (incorrect)

Thread A

```
x = f ( a , b ) ;
if ( x < 0 || x > 9)
  pthread_cond_wait (&cv);
```

Interrupt

Thread B

```
//change a and b;
x = f (a , b) ;
if ( x >= 0 && x <= 9)
  pthread_cond_signal(&cv);
```

:(Broadcast missed by A

Condition variables use (incorrect)

Thread A

```
x = f ( a , b ) ;
if ( x < 0 || x > 9)
  pthread_cond_wait (&cv);
```

Thread B

```
//change a and b;
x = f (a , b) ;
if ( x >= 0 && x <= 9)
  pthread_cond_signal (&cv);
```

A waits forever…

Condition variables use (still incorrect)

Thread A

```
pthread_mutex lock(&mutex ) ;
x = f ( a , b ) ;
if ( x < 0 || x > 9)
  pthread_cond_wait (&cv, &mutex);
pthread_mutex_unlock(&mutex) ;
```

Thread B

```
pthread_mutex lock(&mutex );
//change a and b;
x = f (a , b) ;
if ( x >= 0 && x <= 9)
  pthread_cond_signal (&cv , &mutex);
pthread_mutex_unlock(&mutex);
```

Remember: Condition variable is a shared resource between A and B

→ Every time you use a condition variable you must also use a mutex to prevent the race condition.

One more issue…

Sometimes, the wait function might return even though the condition variable has not actually been signaled.

Thread A

```
pthread_mutex lock(&mutex ) ;
x = f ( a , b ) ;
if ( x < 0 || x > 9)
  pthread_cond_wait (&cv, &mutex);
pthread_mutex_unlock(&mutex) ;
```

Thread B

```
pthread_mutex lock(&mutex );
//change a and b;
x = f (a , b) ;
if ( x >= 0 && x <= 9)
  pthread_cond_signal (&cv , &mutex);
pthread_mutex_unlock(&mutex);
```

Example:

If process P running A and B receives an OS signal

- Any thread in P can be chosen to process the signal.
- → A might be chosen to process the signal handling function
- → wait returns with an error code → A runs even if condition is not true…

How can we fix this?

Condition variables use (correct)

- Retest the condition after pthread_cond_wait() returns.
  - This is most easily done using a loop.

Thread A

```
pthread_mutex lock(&mutex ) ;
while (1){
  x = f ( a , b ) ;
  if ( x < 0 || x > 9)
    pthread_cond_wait (&cv, &mutex);
  else break;
}
pthread_mutex_unlock(&mutex) ;
```

Thread B

```
pthread_mutex lock(&mutex );
//change a and b;
x = f (a , b) ;
if ( x >= 0 && x <= 9)
  pthread_cond_signal (&cv, &mutex);
pthread_mutex_unlock(&mutex);
```

Conditional Variables Interface

- `pthread_cond_init(pthread_cond_t *cv, pthread_condattr_t *cattr)`
    - Initialize the conditional variable, cattr can be NULL

- `pthread_cond_wait(pthread_cond_t *cv, pthread_mutex_t *mutex)`
    - Block thread until condition is true, and atomically unblock mutex.

- `pthread_cond_signal(pthread_cond_t *cv)`
    - Unblock one thread at random that is blocked by the condition variable

- `pthread_cond_broadcast(pthread_cond_t *cv)`
    - Unblock all threads that are blocked on the condition variable pointed to by cv.

Condition Variable Example

```
pthread_cond_t is_zero;
pthread_mutex_t mutex;
int shared_data = 100;


void *thread_func(void *arg){
  while(shared_data > 0) {
    pthread_mutex_lock(&mutex);
    shared_data--;
    printf("%d ", shared_data);
    pthread_mutex_unlock(&mutex);
  }

  printf("Signaling main\n");
  pthread_cond_signal(&is_zero);
  return NULL;
}
```

```
int main (void){

  pthread_t tid;
  void * exit_status;
  int i;

  pthread_cond_init(&is_zero, NULL);
  pthread_mutex_init(&mutex, NULL);

  pthread_create(&tid, NULL, thread_func, NULL);

  pthread_mutex_lock(&mutex);
  printf("Start waiting in main\n");
  while(shared_data!=0)
    pthread_cond_wait(&is_zero, &mutex);
  pthread_mutex_unlock(&mutex);

  printf("Done waiting in main!\n");

  pthread_join(tid, &exit_status);
  return 0;
}
```

Synchronization

Build higher-level synchronization primitives in OS

- Operations that ensure correct ordering of instructions across threads

Motivation: Build them once and get them right

Software

Monitors ✅ **Locks (mutex)** **Semaphores**

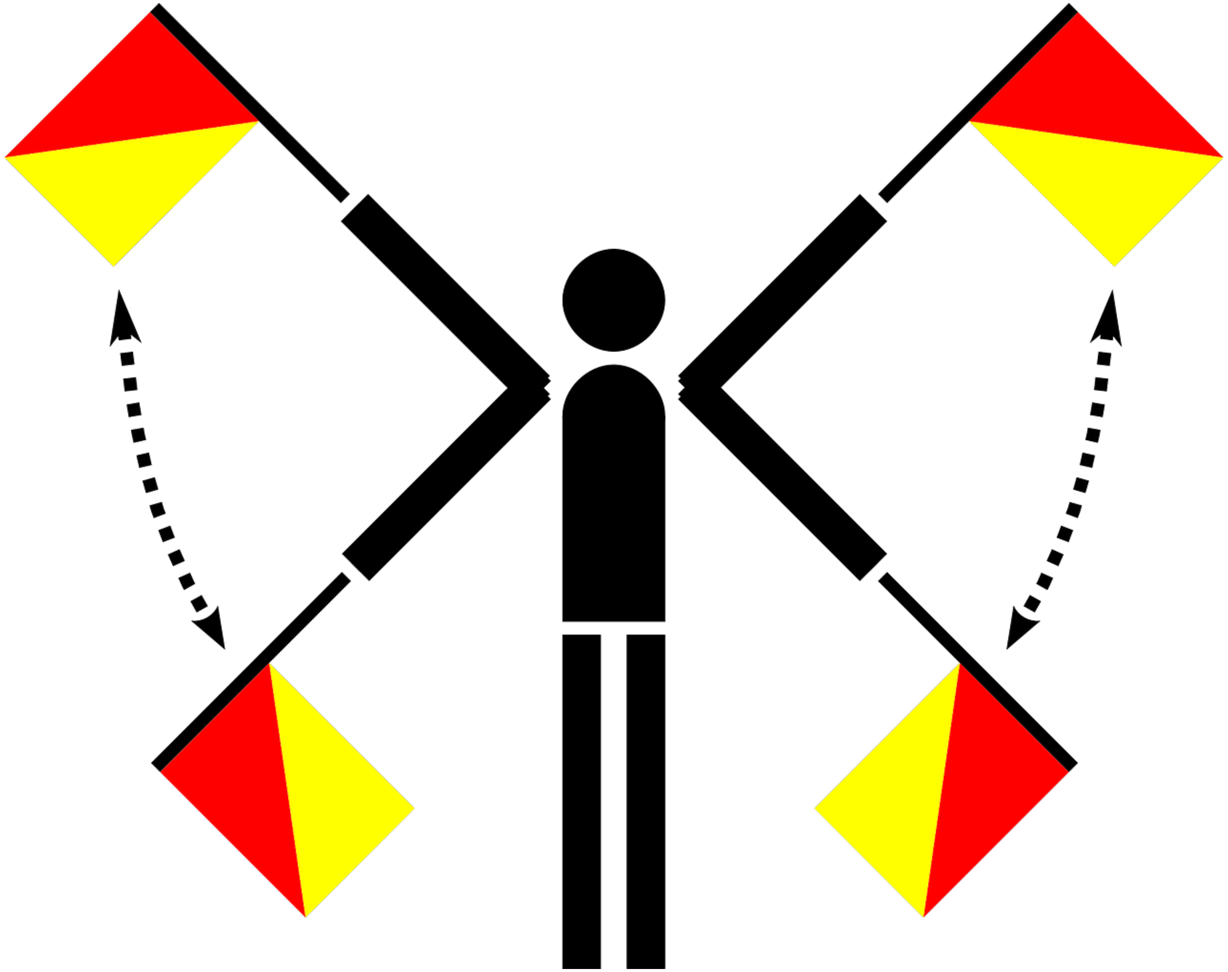
✅ **Condition Variables**

Hardware

Loads Stores Test&Set

Disable Interrupts

Semaphores



What are Semaphores?

- A shared, non-negative counter.

- Two primary operations:
  - Wait → attempts to decrement the counter; blocks when counter is 0.
  - Post (or Signal) → attempts to increment the counter.

- #include<semaphore.h>

Changes are atomic

```
unsigned int
```

Semaphore

Changes are atomic

2 operations

`unsigned int`

Semaphore

`wait()`

`post()`

unsigned int

Semaphore

Changes are atomic

2 operations

`wait()`

try to decrement semaphore value

block if value = 0

`post()`

increment semaphore value

unsigned int v

3

Semaphore

```
wait() {

  while(1){
    if (v>0){
      v--;
      return;
    }
  }

}
```

Atomic execution

Note: wait() is not actually implemented like this. This is how the behavior looks like to the programmer

```
unsigned int v

2
```

Semaphore

```
wait() {

  while(1){
    if (v>0){
      v--;
      return;
    }
  }
}
```

Atomic execution

Note: wait() is not actually implemented like this. This is how the behavior looks like to the programmer

```
unsigned int v
```

0

Semaphore

```
wait() {

  while(1){
    if (v>0){
      v--;
      return;
    }
  }
}
```

Atomic execution

Wait until semaphore value becomes positive again

Note: wait() is not actually implemented like this. This is how the behavior looks like to the programmer

```
unsigned int v

1
```

Semaphore

```
wait() {

  while(1){
    if (v>0){
      v--;
      return;
    }
  }
}
```

Atomic execution

Once the value is positive, the thread that
Was waiting is able to decrement the value.

Note: wait() is not actually implemented like this. This is how the behavior looks like to the programmer

```
unsigned int v
```

0

Semaphore

```
post() {
  v++;
  return;
}
```

Atomic execution

Note: post() is not actually implemented like this. This is how the behavior looks like to the programmer

```
unsigned int v

1
```

Semaphore

```
post() {
    v++;
    return;
}
```

Atomic execution

Note: post() is not actually implemented like this. This is how the behavior looks like to the programmer

```
unsigned int
```

Semaphore

One last thing:

- Semaphore value can be initialized upon creation to a positive value

- or 0

Semaphore Uses

Mutual exclusion

- A semaphore with its counter initialized to 1 is equivalent to a lock.

Bound the concurrency

- Only allow X threads out of N to proceed.

Producer-consumer problem

- More complex use of semaphores. Will see in 2 weeks.

Semaphores interface

- int sem_init(sem_t **sem*, int *pshared*, unsigned *value*);
- int sem_post(sem_t **sem*);
- int sem_wait(sem_t **sem*);

For more details: https://man7.org/linux/man-pages/man0/semaphore.h.0p.html

int sem_init(sem_t **sem*, int *pshared*, unsigned *value*);

- Initializes the semaphore * *sem*. The initial value of the semaphore is *value*.
- If *pshared* is 0, the semaphore is shared among all threads of a process.
- If *pshared* is not zero, the semaphore is shared but should be in shared memory.
- Return 0 on success, -1 on failure.

int sem_wait(sem_t **sem*);

- If the *sem* has a value > 0, decrement the value by 1.
- If *sem* has value 0, the caller will be blocked (busy-waiting or more likely on a queue) until *sem* has a value larger than 0.
- Return 0 on success, -1 on failure.

int sem_post(sem_t **sem*);

- Increment the value of *sem* by 1.
- If threads are blocked waiting for the semaphore, one of them (at random) will return successfully from its call to *sem_wait*(); the semaphore value is immediately decremented.
- Return 0 on success, -1 on failure.

Semaphores example

Note: If 2 is changed to 1, we have a lock behavior

```
#include <pthread.h>
#include <semaphore.h>


pthread_t threads[5];
int tid[5];
sem_t sem;

void * thread_func(void *arg){
  int tid_ = tid[* (int *) arg];
  printf("Thread %d created\n", tid_);
  int j;

  sem_wait(&sem);
  for (j=0; j<3; j++){
    printf("T%d run %d\n", tid_, j);
    sleep(2);
  }
  sem_post(&sem);
}
```

```
int main(){

  sem_init(&sem, 0, 2);
  //sem initialized for all threads in the
process; allow only 2 threads in the critical
section at a time;

  int i;
  for (i=0; i<5; i++){
    tid[i]=i;
    pthread_create(&threads[i], NULL,
thread_func, &tid[i]);
  }

  for (i=0; i<5; i++){
    pthread_join(threads[i], NULL);
  }
  sem_destroy(&sem);
  return 0;
}
```

Synchronization

Build higher-level synchronization primitives in OS

- Operations that ensure correct ordering of instructions across threads

Motivation: Build them once and get them right

Software

Monitors ✅ **Locks (mutex)** ✅ **Semaphores**

✅ **Condition Variables**

Hardware

Loads Stores Test&Set

Disable Interrupts

Monitors

- Collection of variables and functions
- Threads can only access monitor functions
  - Variables are private to the monitor
- Only one process at a time can execute code inside the monitor.

Monitors

- Pthreads does not offer a monitor primitive ☹
- … but possible to implement monitor semantics using mutex and condition variables ☺
- We will see how in 2 weeks.

Synchronization

How are locks, semaphores, cond var, etc **implemented**?

Software

Monitors Locks (mutex) Semaphores

Condition Variables

Hardware

Loads **Stores** **Test&Set**

Disable Interrupts

Lock implementation motivating example “Too much milk”

- Alice and Bob are roommates
- They want to coordinate grocery shopping
  - Need to be careful to not buy too much of perishable items, like milk.

Schedule that leads to too much milk

Time	Alice	Bob
3:00	Look in Fridge. Out of milk.	
3:05	Leave for store.	
3:10	Arrive at store.	Look in Fridge. Out of milk.
3:15	Buy milk.	Leave for store.
3:20	Arrive home, put milk away.	Arrive at store.
3:25		Buy milk.
3:30		Arrive home, put milk away.

Problem specifications

Safety

- Never more than one person buys

Liveness

- Someone buys if needed

Lock implementation: first attempt

Restrict ourselves to use only atomic load and store operations as building blocks.

Idea: Use **a note** to avoid buying too much milk:

- Leave a note on fridge before buying (kind of "lock")
- Remove note after buying (kind of "unlock")
- Don't buy if note (wait)

Lock implementation: first attempt (incorrect)

```
if (noMilk) {
  if (noNote) {
    leave Note;
    buy milk;
    remove note; }
}
```

Lock implementation: first attempt (incorrect)

```
if (noMilk) {
  if (noNote) {
    leave Note;
    buy milk;
    remove note; }
}
```

load (atomic)

store (atomic)

Lock implementation: first attempt (incorrect)

Thread A

```
if (noMilk) {

  if (noNote) {
    leave Note;
    buy milk;
    remove note;
  }
}
```

Thread B

```
if (noMilk) {
  if (noNote) {




    leave Note;
    buy milk;
    remove note; }
}
```

Lock implementation: first attempt (incorrect)

Thread A

```
if (noMilk) {

  if (noNote) {
    leave Note;
    buy milk;
    remove note;
  }
}
```

Remember: scheduler can create any interleaving
We are assuming a malicious scheduler

Thread B

```
if (noMilk) {
  if (noNote) {




    leave Note;
    buy milk;
    remove note; }
}
```

First attempt result

- Still too much milk but only occasionally!
  - This is worse than a consistent error, because it is harder to catch.
- Thread can get context switched after checking milk and note but before buying milk!

Lock implementation: second attempt (incorrect)

- How about labeled notes?

Thread A

```
leave note A;
if (noNote B) {
  if (noMilk) {
    buy milk;
  }
}
remove note A;
```

Thread B

```
leave note B;
if (noNote A) {
  if (noMilk) {
    buy milk;
  }
}
remove note B;
```

Problem solved?

Lock implementation: second attempt (incorrect)

- How about labeled notes?

Thread A

Proc switch

```
leave note A;
if (noNote B) {
  if (noMilk) {
    buy milk;
  }
}
remove note A;
```

Thread B

```
leave note B;
if (noNote A) {
  if (noMilk) {
    buy milk;
  }
}
remove note B;
```

Not quite…

Lock implementation: second attempt (incorrect)

- How about labeled notes?

Thread A

```
leave note A;
if (noNote B) {
  if (noMilk) {
    buy milk;
  }
}
remove note A;
```

Proc switch

Thread B

```
leave note B;
if (noNote A) {
  if (noMilk) {
    buy milk;
  }
}
remove note B;
```

Not quite…

Lock implementation: second attempt (incorrect)

- How about labeled notes?

Thread A

```
leave note A;
if (noNote B) {
  if (noMilk) {
    buy milk;
  }
}
remove note A;
```

Thread B

```
leave note B;
if (noNote A) {
  if (noMilk) {
    buy milk;
  }
}
remove note B;
```

Not quite…

Lock implementation: second attempt (incorrect)

- How about labeled notes?

Thread A

```
leave note A;
if (noNote B) {
  if (noMilk) {
    buy milk;
  }
}
remove note A;
```

Proc switch

Thread B

```
leave note B;
if (noNote A) {
  if (noMilk) {
    buy milk;
  }
}
remove note B;
```

Not quite…

Lock implementation: second attempt (incorrect)

- How about labeled notes?

Thread A

```
leave note A;
if (noNote B) {
  if (noMilk) {
    buy milk;
  }
}
remove note A;
```

Proc switch

Thread B

```
leave note B;
if (noNote A) {
  if (noMilk) {
    buy milk;
  }
}
remove note B;
```

Not quite…

Lock implementation: second attempt (incorrect)

- How about labeled notes?

Thread A

```
leave note A;
if (noNote B) {
  if (noMilk) {
    buy milk;
  }
}
remove note A;
```

Thread B

```
leave note B;
if (noNote A) {
  if (noMilk) {
    buy milk;
  }
}
remove note B;
```

Not quite…

Nobody buys milk ☹

Lock implementation: third attempt (correct)

- How about labeled notes?

Thread A

```
leave note A;
while (note B)
  do nothing;
if (noMilk)
  buy milk;
remove note A;
```

Thread B

```
leave note B;
if (noNote A) {
  if (noMilk)
    buy milk;
}
remove note B;
```

This works!

B has priority to buy

Correctness argument. Case 1

Thread A

```
leave note A;
while (note B)
  do nothing;
if (noMilk)
  buy milk;
remove note A;
```

Happened before

Thread B

```
leave note B;
if (noNote A) {
  if (noMilk)
    buy milk;
}
remove note B;
```

Correctness argument. Case 1

Thread A

```
leave note A;
while (note B)
  do nothing;




if (noMilk)
  buy milk;
remove note A;
```

Happened before

Thread B

```
leave note B;
if (noNote A) {
  if (noMilk)
    buy milk;
}
remove note B;
```

Correctness argument. Case 1

Thread A

```
leave note A;
while (note B)
  do nothing;
```

Wait for note B
to be removed

```
if (noMilk)
  buy milk;
remove note A;
```

Happened before

Thread B

```
leave note B;
if (noNote A) {
  if (noMilk)
    buy milk;
}
remove note B;
```

Correctness argument. Case 1

Thread A

```
leave note A;
while (note B)
  do nothing;
```

Happened before

Wait for note B
to be removed

```
if (noMilk)
  buy milk;
remove note A;
```

Thread B

```
leave note B;
if (noNote A) {
  if (noMilk)
    buy milk;
}
remove note B;
```

Correctness argument. Case 2

Thread A

```
leave note A;
while (note B)
  do nothing;
if (noMilk)
  buy milk;
remove note A;
```

Happened before

Thread B

```
leave note B;
if (noNote A) {
  if (noMilk)
    buy milk;
}
remove note B;
```

Correctness argument. Case 2

Thread A

```
leave note A;
while (note B)
  do nothing;
if (noMilk)
  buy milk;
remove note A;
```

Happened before

Thread B

```
leave note B;
if (noNote A) {
  if (noMilk)
    buy milk;
}
remove note B;
```

Correctness argument. Case 2

Thread A

```
leave note A;
while (note B)
  do nothing;

if (noMilk)
  buy milk;
remove note A;
```

Happened before

Wait for note B
to be removed

Thread B

```
leave note B;
if (noNote A) {
  if (noMilk)
    buy milk;
}
remove note B;
```

Attempt 3 discussion

Solution 3 works, but it’s really unsatisfactory.

- Complex, even for this simple example.
  - Hard to convince yourself that this really works.

- A’s code is different from B’s –what if lots of threads?
  - Code would have to be slightly different for each thread.

- While A is waiting, it is consuming CPU time.
  - This is called “busy-waiting”.

There must be a better way!

- Have higher-level hardware primitives than atomic load & store
- Build higher-level abstractions on this hardware support

Disabling interrupts

- Lock implementation code executed in kernel mode.
- Key idea: maintain a lock variable and impose mutual exclusion only during operations on that variable

Disabling interrupts

```
int value = FREE;

Lock() {
  disable interrupts;
  if (value == BUSY) {
    put thread on wait queue;
    Go to sleep();
    // Enable interrupts? }
  else {
    value = BUSY;}
  enable interrupts;}
```

```
Unlock() {
  disable interrupts;
  if (anyone on wait queue) {
    take thread off wait queue;
    place on ready queue;}
  else {
    value = FREE;}
  enable interrupts; }
```

Disabling interrupts discussion

Why do we need to disable interrupts?

- Avoid interruption between checking and setting lock value
- Otherwise two threads could think that they both have lock

```
Lock() {
  disable interrupts;
  if (value == BUSY) {
    put thread on wait queue;
    Go to sleep();
    // Enable interrupts? }
  else {
    value = BUSY;}
  enable interrupts;}
```

Critical section is short in kernel mode

What about enabling interrupts when going to sleep?

```
Lock() {
  disable interrupts;
  if (value == BUSY) {
    put thread on wait queue;
    Go to sleep();
    // Enable interrupts? }
  else {
    value = BUSY;}
  enable interrupts;}
```

Enable interrupts here?

☹ Release can check the queue and not wake up thread

What about enabling interrupts when going to sleep?

```
Lock() {
  disable interrupts;
  if (value == BUSY) {
    put thread on wait queue;
    Go to sleep();
    // Enable interrupts? }
  else {
    value = BUSY;}
  enable interrupts;}
```

Enable interrupts here?

- Release puts the thread on the ready queue, but the thread still thinks it needs to go to sleep
- Misses wakeup and still holds lock (deadlock!) ☹☹

What about enabling interrupts when going to sleep?

```
Lock() {
  disable interrupts;
  if (value == BUSY) {
    put thread on wait queue;
    Go to sleep();
    // Enable interrupts? }
  else {
    value = BUSY;}
  enable interrupts;}
```

Want to enable interrupts after sleep()

- But how?

What about enabling interrupts when going to sleep?

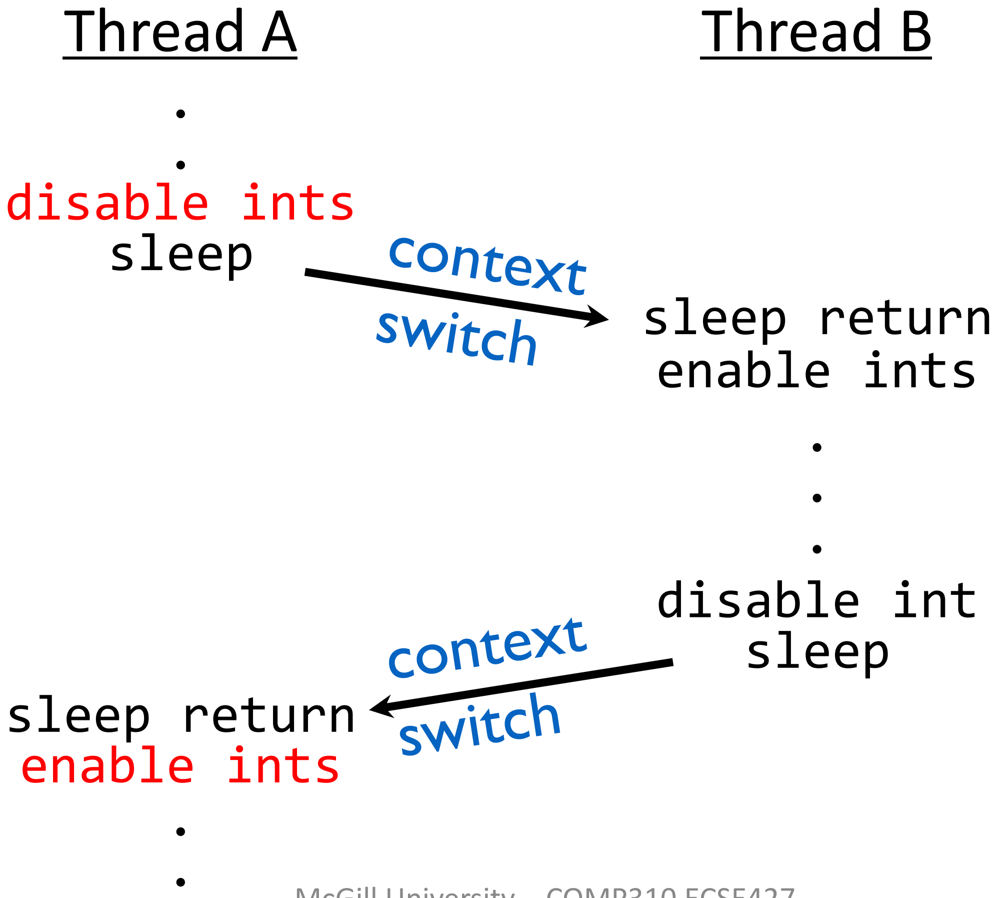
```
Lock() {
  disable interrupts;
  if (value == BUSY) {
    put thread on wait queue;
    Go to sleep();
    // Enable interrupts? }
  else {
    value = BUSY;}
  enable interrupts;}
```

Want to enable interrupts after sleep()

- But how? **In scheduler.**

In scheduler, since interrupts are disabled when you call sleep():

- Responsibility of the next thread to re-enable interrupts
- When the sleeping thread wakes up, returns to lock() and re-enables interrupts



Simulation of locks with disabled interrupts

Value: 0 | waiters | owner | READY

Running

Thread A

```
lock.Lock();
 …
 critical section;
 …
lock.Unlock();
```

INIT

```
  int value = 0;

Lock() {
  disable interrupts;
  if (value == 1) {
    put thread on wait-queue;
    go to sleep() //??
  } else {
    value = 1;
  }
  enable interrupts;
}

Unlock() {
  disable interrupts;
  if anyone on wait queue {
    take thread off wait-queue
    Place on ready queue;
  } else {
    value = 0;
  }
  enable interrupts;
}
```

Ready

Thread B

```
lock.Lock();
 …
 critical section;
 …
lock.Unlock();
```

Simulation of locks with disabled interrupts

Value: 1 | waiters | owner | READY

Running

Thread A

```
lock.Lock();
 …
 critical section;
 …
lock.Unlock();
```

```
INIT
   int value = 0;

Lock() {
  disable interrupts;
  if (value == 1) {
    put thread on wait-queue;
    go to sleep() //??
  } else {
    value = 1;
  }
  enable interrupts;
}

Unlock() {
  disable interrupts;
  if anyone on wait queue {
    take thread off wait-queue
    Place on ready queue;
  } else {
    value = 0;
  }
  enable interrupts;
}
```

Ready

Thread B

```
lock.Lock();
 …
 critical section;
 …
lock.Unlock();
```

Simulation of locks with disabled interrupts

Value: 1 | waiters | owner

READY

Ready Running

Thread A

Ready Running

Thread B

```
INIT
   int value = 0;
```

Thread A:

```
lock.Lock();
 …
 critical section;
 …
lock.Unlock();
```

```
Lock() {
  disable interrupts;
  if (value == 1) {
    put thread on wait-queue;
    go to sleep() //??
  } else {
    value = 1;
  }
  enable interrupts;
}

Unlock() {
  disable interrupts;
  if anyone on wait queue {
    take thread off wait-queue
    Place on ready queue;
  } else {
    value = 0;
  }
  enable interrupts;
}
```

Thread B:

```
lock.Lock();
 …
 critical section;
 …
lock.Unlock();
```

Simulation of locks with disabled interrupts

Value: 1 | waiters | owner | READY

Ready
Running

Thread A

Waiting
Running

Thread B

INIT
```
int value = 0;
```

Thread A:
```
lock.Lock();
 …
 critical section;
 …
lock.Unlock();
```

```
Lock() {
  disable interrupts;
  if (value == 1) {
    put thread on wait-queue;
    go to sleep() //??
  } else {
    value = 1;
  }
  enable interrupts;
}

Unlock() {
  disable interrupts;
  if anyone on wait queue {
    take thread off wait-queue
    Place on ready queue;
  } else {
    value = 0;
  }
  enable interrupts;
}
```

Thread B:
```
lock.Lock();
 …
 critical section;
 …
lock.Unlock();
```

Simulation of locks with disabled interrupts

Value: 1 | waiters | owner | READY

Running

Thread A

```
lock.Lock();
 …
 critical section;
 …
lock.Unlock();
```

Ready / Waiting

Thread B

```
lock.Lock();
 …
 critical section;
 …
lock.Unlock();
```

```
INIT
   int value = 0;

Lock() {
  disable interrupts;
  if (value == 1) {
    put thread on wait-queue;
    go to sleep() //??
  } else {
    value = 1;
  }
  enable interrupts;
}

Unlock() {
  disable interrupts;
  if anyone on wait queue {
    take thread off wait-queue
    Place on ready queue;
  } else {
    value = 0;
  }
  enable interrupts;
}
```

Simulation of locks with disabled interrupts

Value: 1 | waiters | owner | READY

Running Ready

Thread A

INIT
```
int value = 0;
```

Running Ready

Thread B

```
lock.Lock();
 …
 critical section;
 …
lock.Unlock();
```

```
Lock() {
disable interrupts;
if (value == 1) {
  put thread on wait-queue;
  go to sleep() //??
} else {
  value = 1;
}
enable interrupts;
}
```

```
Unlock() {
disable interrupts;
if anyone on wait queue {
  take thread off wait-queue
  Place on ready queue;
} else {
  value = 0;
}
enable interrupts;
}
```

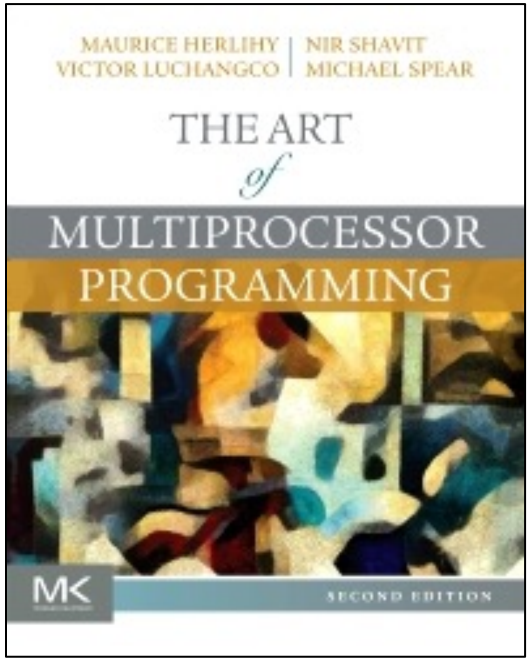
```
lock.Lock();
 …
 critical section;
 …
lock.Unlock();
```

Further Optional Reading

Operating Systems: Three Easy Pieces by R. & A. Arpaci-Dusseau

Chapters 25 – 31 (inclusive)
https://pages.cs.wisc.edu/~remzi/OSTEP/

Reading on concurrency: Herlihy & Shavit: The Art of Multiprocessor Programming, 2nd edition.



Credits:

Some slides adapted from the OS courses of Profs. Remzi and Andrea Arpaci-Dusseau (University of Wisconsin-Madison), Prof. Willy Zwaenepoel (University of Sydney), and Prof. Maurice Herlihy (Brown University), Prof. Natacha Crooks (UC Berkeley).