ECS 150 - Synchronization

Prof. Joël Porquet-Lupine

UC Davis - 2020/2021

Threads (recap)

Memory sharing

- Private processor registers
- Private stack
- Shared *global* memory

Type of sharing

Independent

- Threads work on distinct areas of shared data

```
int a[N], b[N], c[N], d[N], e[N];

do_mult(p, m)
{
    for (i = p; i < m; i++)
        a[i] += b[i] * c[i] + d[i] * e[i];
}
int main(void)
{
    ...
    thread_create(do_mult, 0, N/2);
    thread_create(do_mult, N/2, N);
    ...
}
```

Cooperating

- Threads work on same areas of shared data

```
int a[N], b[N], c[N], d[N], e[N];

do_mult1()
{
    for (i = 0; i < n; i++)
        a[i] += b[i] * c[i];
}
do_mult2()
{
    for (i = 0; i < n; i++)
        a[i] += d[i] * e[i];
}
int main(void)
{
    ...
    thread_create(do_mult1);
    thread_create(do_mult2);
    ...
}
```

Concurrency issues

Example

```c
int x = 0;

void thread_a(void)
{
    x = x + 1;
}

void thread_b(void)
{
    x = x + 2;
}

int main(void)
{
    thread_create(thread_a);
    thread_create(thread_b);
    thread_join(thread_a);
    thread_join(thread_b);
    printf("%d\n", x);
    return 0;
}
```

Execution

- Typical output:

```
$ ./a.out
3
```

- Also possible (yet probably very rare)...

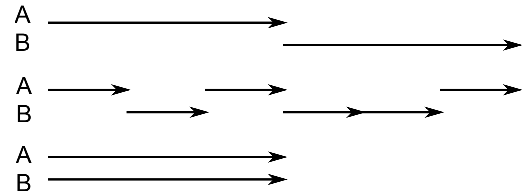
```
$ ./a.out
2
$ ./a.out
1
```

Concurrency issues

Indeterministic interleavings

Thread scheduling

- Indeterministic scheduling
- Sequential execution, concurrent execution, parallelism, etc.



Instruction reordering

- Compiler instruction reordering
- Hardware instruction reordering

```
void thread1(void) {
    ...
    p = init();
    init = true;
    ...
}
void thread2(void) {
    ...
    while (init == false);
    q = docomputation(p);
    ...
}
```


Multi-word operations

- Multi-word operations are not atomic

```
uint64_t num = 2;
struct {
    char c;
    short int b;
} s = { 'j', 23 };
```

Concurrency issues

Race conditions

Definition

- **Race condition** when the output of a concurrent program depends on the order of operations between threads.

```
void thread_a(void)
{
    x = x + 1;
}
```

```
void thread_b(void)
{
    x = x + 2;
}
```


Difficulties

- Number of possible "interleavings" can be huge
- Some interleavings are good
    - Vast majority of them usually are
- Some interleavings are bad
    - They may even be extremely rare...

Solution?

- Synchronization!

Too Much Milk!

Example

Roommate 1	Roommate 2
Arrive home	
Look in fridge, out of milk	
Leave for store	
	Arrive home
Arrive at store	Look in fridge, out of milk
Buy milk	Leave for store
Arrive home, put milk away	Buy milk
	Arrive home, put milk away
	Oh, no!

Required correctness properties

- **Safety**: at most one person buys milk at a time
- **Liveness**: someone buys milk if needed

Too Much Milk!

1. Leaving a note

Roommate 1 (thread 1)

```
if (note == 0) {
    if (milk == 0) {
        note = 1;
        milk++;
        note = 0;
    }
}
```

Roommate 2 (thread 2)

```
if (note == 0) {
    if (milk == 0) {
        note = 1;
        milk++;
        note = 0;
    }
}
```


Safety and liveness

- Not safe if threads are descheduled right after the two tests
  - They would both put a note and get milk, resulting in two bottles of milk!

Too Much Milk!

2. Using two notes

Roommate 1 (thread 1)

```
note1 = 1;
if (note2 == 0) {
    if (milk == 0) {
        milk++;
    }
}
note1 = 0;
```

Roommate 2 (thread 2)

```
note2 = 1;
if (note1 == 0) {
    if (milk == 0) {
        milk++;
    }
}
note2 = 0;
```


Safety and liveness

- Not live if threads are descheduled right after setting their personal notes
  - They both think the other is on their way getting milk but no does eventually does

Too Much Milk!

3. Using asymmetric notes

Roommate 1 (thread 1)

```
note1 = 1;
while (note2 == 1)
    ;
if (milk == 0) {
    milk++;
}
note1 = 0;
```

Roommate 2 (thread 2)

```
note2 = 1;
if (note1 == 0) {
    if (milk == 0) {
        milk++;
    }
}
note2 = 0;
```


Safety and liveness

- Yes, safe and live!

Too Much Milk!

3. Using asymmetric notes: explained

Roommate 1 (thread 1)

```
note1 = 1;
while (note2 == 1)
      ;
if (milk == 0) {
    milk++;
}
note1 = 0;
```

Roommate 2 (thread 2)

```
note2 = 1;
if (note1 == 0) {
    if (milk == 0) {
        milk++;
    }
}
note2 = 0;
```


Issues

1. Way too over-engineered!
2. Asymmetrical, non-scalable
3. Involves *busy-waiting*

Peterson's algorithm

- Share a single resource using only shared memory
- Symmetric and scalable
- But still quite complex...
- More here

Critical section

Definition

- Piece of code where the shared resource is accessed
- Needs to be properly protected to avoid race conditions
- Cannot be executed by more than one thread at a time

Thread 1

```
...
CS_enter();
    Critical section
CS_exit();
...
```

Thread 2

```
...
CS_enter();
    Critical section
CS_exit();
...
```


Correctness properties

1. **Safety**
2. **Liveness**
3. Bounded waiting
4. Failure atomicity

Mutual exclusion

- Property of concurrency control
- Requirement that only one thread can enter critical section at a time
- Active thread excludes its peers

Critical section

Formalizing "Too Much Milk!"

- Shared variable
- Safety property
- Liveness property

Roommate 1 (thread 1)

```
note1 = 1;            /*
while (note2 == 1)     * CS_enter()
    ;                  */
if (milk == 0) {      /*
    milk++;            * CS
}                      */
note1 = 0;            /* CS_exit() */
```

Roommate 2 (thread 2)

```
note2 = 1;            /* CS_enter()
if (note1 == 0) {      */
    if (milk == 0) {  /*
        milk++;        * CS
    }                  */
}
note2 = 0;            /* CS_exit() */
```

ECS 150 - Synchronization

Prof. Joël Porquet-Lupine

UC Davis - 2020/2021

Recap

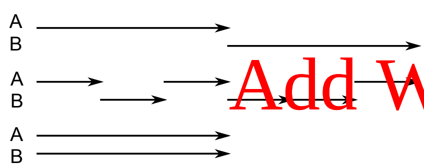
Race condition

- Output of concurrent program depends on order of operations between threads

```
void thread_a(void)                void thread_b(void)
{                                  {
    x = x + 1;                         x = x + 2;
}                                  }
```

```
$ ./a.out                          $ ./a.out
3                                  2
                                   $ ./a.out
                                   1
```

- Indeterministic concurrent execution
  - Thread scheduling



  - Instruction reordering
  - Multi-word operations

Critical section

```
void thread_1(void) {
    note1 = 1;            // v
    while (note2 == 1)    // CS_enter()
        ;                 // ^
    if (milk == 0) {          // v
        milk++;               // CS
    }                         // ^
    note1 = 0;            // CS_exit()
}
void thread_2(void) {
    note2 = 1;                // CS_enter()
    if (note1 == 0) {         // ^
        if (milk == 0) {      // v
            milk++;           // CS
        }                     // ^
    }
    note2 = 0;                // CS_exit()
}
```

- Shared variable
- Safety property
- Liveness property
- Mutual exclusion

Locks

Definition

- A lock is a *synchronization* variable that provides *mutual exclusion*
- Two states: *locked* and *free* (initial state is generally *free*)

API

- `lock()` or `acquire()`
  - Wait until lock is free, then grab it
- `unlock()` or `release()`
  - Unlock, and allow one of the threads waiting in *acquire* to proceed


```c
int milk;

int main(void)
{
    thread_create(roommate_fcn);
    thread_create(roommate_fcn);
    ...
}
```

```c
void roommate_fcn(void)
{
    ...
    lock();

    /* Critical section */
    if (!milk)
        milk++;

    unlock();
    ...
}
```

Locks

Simple uniprocessor implementation

- Race conditions are coming from indeterministic scheduling
  - Breaks atomicity of instruction sequence
  - Caused by preemption (i.e. timer interrupt)
- Solution: disable the interrupts!

```
void lock()
{
    disable_interrupts();
}
```

```
void unlock()
{
    enable_interrupts();
}
```


Issues

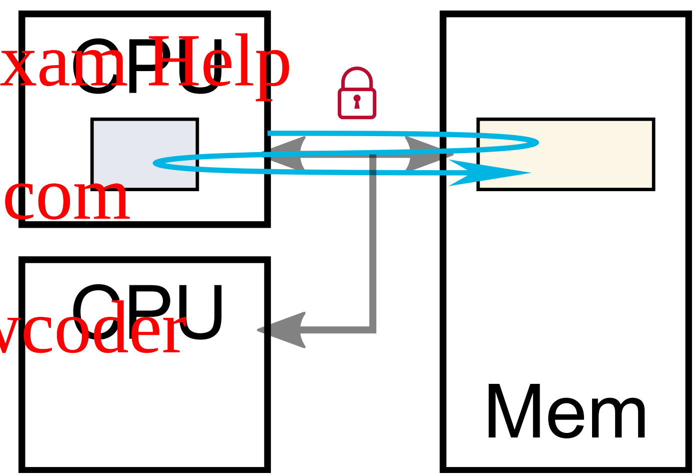
- Only works on uniprocessor systems
- Dangerous to have unpreemptable code
- Cannot be used by user applications

Multiprocessor spinlocks

Hardware support

- *Test-and-set* hardware primitive to provide mutual exclusion
  - a.k.a *Read-modify-write* operation
- Typically relies on a multi-cycle bus operation that **atomically** reads and updates a memory location
  - Multiprocessor support

```
/* Equivalent of a test&set hardware instruc
tion in software */
ATOMIC int test_and_set(int *mem)
{
    int oldval = *mem;
    *mem = 1;
    return oldval;
}
```

Implementation

```
void spinlock_lock(int *lock)
{
    while (test_and_set(lock) == 1);
}
```

```
void spinlock_unlock(int *lock)
{
    *lock = 0;
}
```

Multiprocessor spinlocks

Revisiting "Too Much Milk!"

```
int lock = 0;
```

Thread 1

```
spinlock_lock(&lock);
if (milk == 0) {
    milk++;
}
spinlock_unlock(&lock);
```

Thread 2

```
spinlock_lock(&lock);
if (milk == 0) {
    milk++;
}
spinlock_unlock(&lock);
```


Thread 3

```
spinlock_lock(&lock);
if (milk == 0) {
    milk++;
}
spinlock_unlock(&lock);
```

Thread 4

```
spinlock_lock(&lock);
if (milk == 0) {
    milk++;
}
spinlock_unlock(&lock);
```

Multiprocessor spinlocks

Issue

- Busy-waiting wastes cpu cycles
  - Only to reduce latency

```
void spinlock_lock(int *lock)
{
    while (test_and_set(lock) == 1);
}
```

Solution

"Cheap" busy-waiting

- Yield/sleep when unable to get the lock, instead of looping

```
void lock(int *lock)
{
    while (test_and_set(lock) == 1)
        yield(); //or, sleep(N);
}
```

Cons

- Yielding still wastes cpu cycles
- Sleeping impacts latency as well

Better primitives

- *Block* waiting threads until they can proceed

```
void lock(int *lock)
{
    while (test_and_set(lock) == 1)
        block(lock);
}
```

Examples

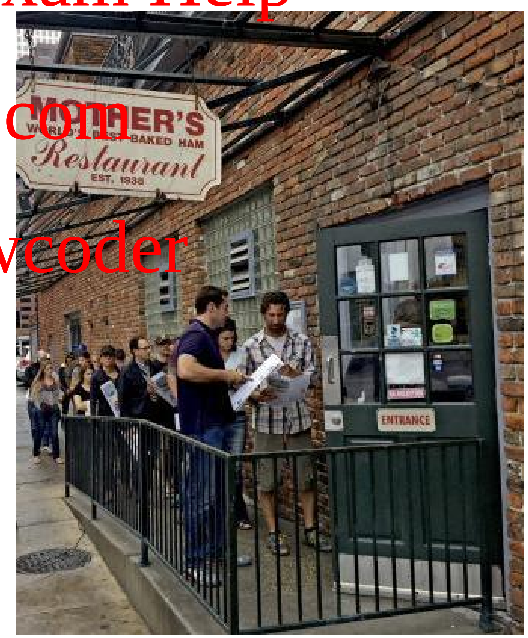
- Semaphores
- Mutexes (equivalent to binary semaphore with the notion of ownership)

Semaphores

Definition

- Invented by Dijkstra in the 60's
- A semaphore is a generalized lock
    - Used for different types of synchronization (including mutual exclusion)
    - Keeps track an arbitrary resource count
        - Queue of threads waiting to access resource

One resource to share



Multiple resources to share

Semaphores

API

- Initial count value *(but not a maximum value)*

```
sem = sem_create(count);
```

- `down()` or `P()`
  - Decrement by one, or block if already 0
- `up()` or `V()`
  - Increment by one, and wake up one of the waiting threads if any

Possible implementation:


```
void sem_down(sem)
{
    spinlock_lock(sem->lock);
    while (sem->count == 0) {
        /* Block self */
        ...
    }
    sem->count -= 1;
    spinlock_unlock(sem->lock);
}
```

```
void sem_up(sem)
{
    spinlock_lock(sem->lock);
    sem->count += 1;
    /* Wake up first in line */
    /* (if any)              */
    ...
    spinlock_unlock(sem->lock);
}
```

Semaphores

Binary semaphore

- Semaphore which count value is either 0 or 1
- Can be used similarly as a lock
  - But no busy waiting, waiting thread are blocked until they can get the lock
- Guarantees mutually exclusive access to a shared resource
- Initial value is generally 1 (ie *free*)

Example


```
sem = sem_create(1);
```


Thread 1

```
...
down(sem);
    Critical section
up(sem);
...
```

Thread 2


```
...
down(sem);
    Critical section
up(sem);
...
```

Mutexes are a similar concept except that they also have the notion of ownership

Semaphores

Counted semaphore

- Semaphore which count value can be any *positive* integer
    - Represents a resource with many "units" available
- Initial count is often the number of initial resources (if any)
- Allows a thread to continue as long as enough resources are available
- Used for synchronization

Example


```
sem_packet = sem_create(0);
```


Thread 1

```
while (1) {
    x = get_network_packet();
    enqueue(packetq, x);
    up(sem_packet);
}
```

Thread 2


```
while (1) {
    down(sem_packet);
    x = dequeue(packetq);
    process_contents(x);
}
```

ECS 150 - Synchronization

Prof. Joël Porquet-Lupine

UC Davis - 2020/2021

Recap

Locks

```
void thread(void) {
    lock();
        /* Critical section */
        ...
    unlock();
}
```

Atomic spinlocks

- Based on atomic *test-and-set* instruction
- Compatible with multiprocessor systems
- Accessible to user processes

```
void spinlock_lock(int *lock) {
    while (test_and_set(lock) == 1);
}
```

```
void spinlock_unlock(int *lock) {
    *lock = 0;
}
```

- But based on busy-waiting

Semaphores

- Internal count
- `down()` decrements count by one, or blocks if count is 0
- `up()` increments count by one, and wakes up first blocked thread if any

Binary semaphore

```
sem = sem_create(1);
void thread(void) {
    sem_down(sem);
        /* Critical section */
        ...
    sem_up(sem);
}
```

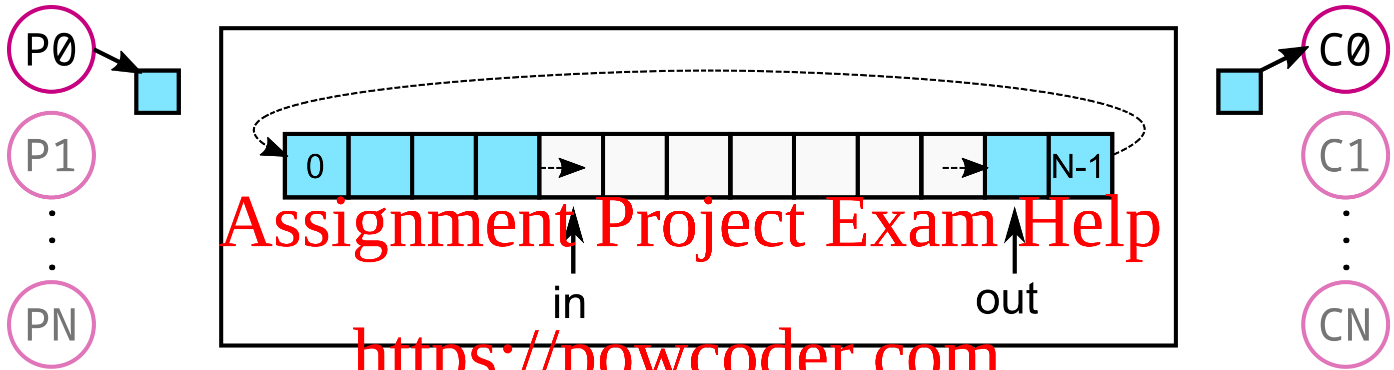
Counted semaphore

```
sem = sem_create(0);
void thread1(void) {          void thread2(void) {
    x = get_packet();             down(sem);
    enqueue(q, x);                x = dequeue(q);
    up(sem);                      process_packet(x);
}                             }
```

Producer-consumer problem

Definition

Two or more threads communicate through a circular data buffer: some threads *produce* data that others *consume*.



- Bounded buffer of size N
- Producers write data to buffer
  - Write at `in` and moves rightwards
  - Don't write more than the amount of available space
- Consumers read data from buffer
  - Read at `out` and moves rightwards
  - Don't consume if there is no data
- Allow for multiple producers and consumers

Producer-consumer problem

Solution 1: no protection

```
int buf[N], in, out;
```

```
void produce(int item)
{
    buf[in] = item;
    in = (in + 1) % N;
}
```

```
int consume(void)
{
    int item = buf[out];
    out = (out + 1) % N;
    return item;
}
```


Issues

- Unprotected shared state
  - Race conditions on all shared variables
- No synchronization between consumers and producers

Producer-consumer problem

Solution 2: Lock semaphores

- Add protection of share state
  - Mutual exclusion around critical sections
  - Guarantees one producer and one consumer at a time

```
int buf[N], in, out;
sem_t lock_prod = sem_create(1), lock_cons = sem_create(1);
```



```
void produce(int item)
{
    sem_down(lock_prod);
    buf[in] = item;
    in = (in + 1) % N;
    sem_up(lock_prod);
}
```

```
int consume(void)
{
    sem_down(lock_cons);
    int item = buf[out];
    out = (out + 1) % N;
    sem_up(lock_cons);
    return item
}
```

Producer-consumer problem

Solution 3: Communication semaphores

- Add synchronization between producers and consumers
  - Producers wait if buffer is full
  - Consumers wait if buffer is empty

```
int buf[N], in, out;
sem_t lock_prod = sem_create(1), lock_cons = sem_create(1);
sem_t empty = sem_create(N), full = sem_create(0);
```

```
void produce(int item)
{
    sem_down(empty); //need empty spot
    sem_down(lock_prod);
    buf[in] = item;
    in = (in + 1) % N;
    sem_up(lock_prod);
    sem_up(full); //new item avail
}
```

```
int consume(void)
{
    sem_down(full); //need new item
    sem_down(lock_cons);
    int item = buf[out];
    out = (out + 1) % N;
    sem_up(lock_cons);
    sem_up(empty); //empty slot avail
    return item
}
```

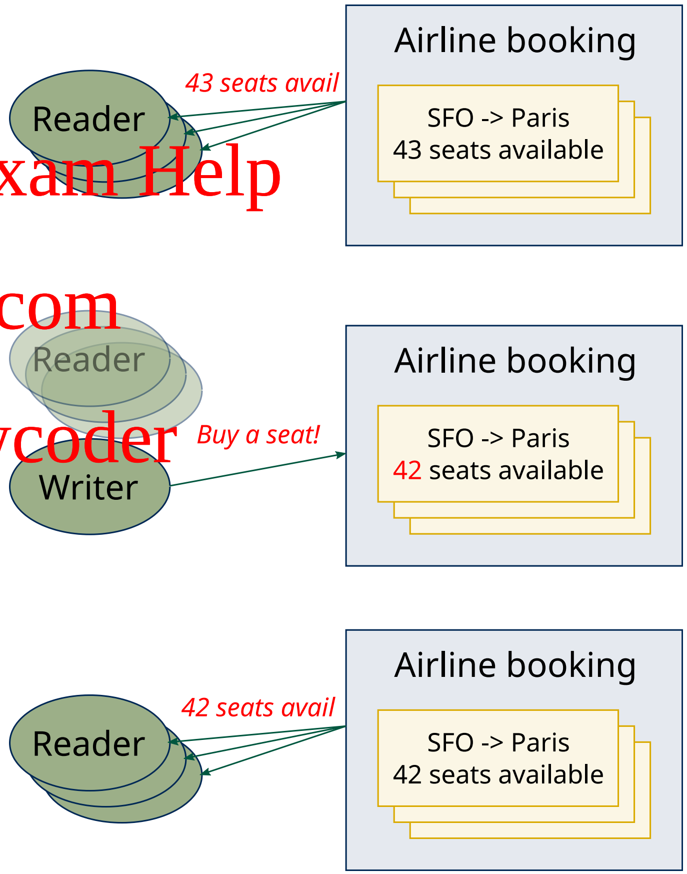
Readers-writers problem

Definition

- Multiple threads access the same shared resource, but differently
  - *Many* threads only *read* from it
  - *Few* threads only *write* to it
- Readers vs writers
  - Multiple *concurrent* readers at a time
  - Or single writer at a time

Examples

- Airline ticket reservation
- File manipulation



Readers-writers problem

Solution 1: Protect resource

```
sem_t rw_lock = sem_create(1);
```

```
void writer(void)
{
    sem_down(rw_lock);
    ...
    /* perform write */
    ...
    sem_up(rw_lock);
}
```

```
int reader(void)
{
    sem_down(rw_lock);
    ...
    /* perform read */
    ...
    sem_up(rw_lock);
}
```


Analysis

- Mutual exclusion between readers and writers: *Yes*
- Only one writer can access the critical section: *Yes*
- Multiple readers can access the critical section at the same time: *No!*

Readers-writers problem

Solution 2: Enable multiple readers

```
int rcount = 0;
sem_t rw_lock = sem_create(1);
```

```
void writer(void)
{
    sem_down(rw_lock);
    ...
    /* perform write */
    ...
    sem_up(rw_lock);
}
```

```
int reader(void)
{
    rcount++;
    if (rcount == 1)
        sem_down(rw_lock);
    ...
    /* perform read */
    ...
    rcount--;
    if (rcount == 0)
        sem_up(rw_lock);
}
```


Issue

- Race condition between readers on variable `rcount`!

Readers-writers problem

Solution 3: Protect multiple readers

```
int rcount = 0;
sem_t rw_lock = sem_create(1), count_mutex = sem_create(1);
```

```
void writer(void)
{
    sem_down(rw_lock);
    ...
    /* perform write */
    ...
    sem_up(rw_lock);
}
```

```
int reader(void)
{
    sem_down(count_mutex);
    rcount++;
    if (rcount == 1)
        sem_down(rw_lock);
    sem_up(count_mutex);
    ...
    /* perform read */
    ...
    sem_down(count_mutex);
    rcount--;
    if (rcount == 0)
        sem_up(rw_lock);
    sem_up(count_mutex);
}
```

Analysis

- Correct solution
- But suffers from potential starvation of writers

Semaphores

Concluding notes

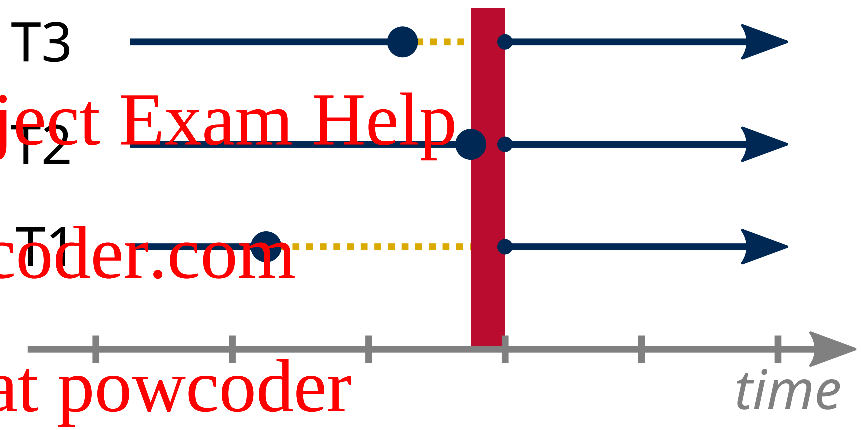
- *Semaphores considered harmful* (Dijkstra, 1968)
  - Simple algorithms can require more than one semaphore
    - Increase of complexity to manage them all
  - Semaphores are low-level primitives
    - Easy to make programming mistakes (e.g. `down()` followed by `down()`)
  - Programmer must keep track of the order of all semaphores operations
    - Avoid deadlocks
  - Semaphores are used for both mutual exclusion and synchronization between threads
    - Difficult to determine which meaning a given semaphore has
- Need for another abstraction
  - Clear distinction between mutual exclusion and synchronization aspects
  - Concept of *monitor* developed in early 70's

Synchronization barriers

Concept

- Enables multiple threads to wait until all threads have reached a particular point of execution before being able to proceed further

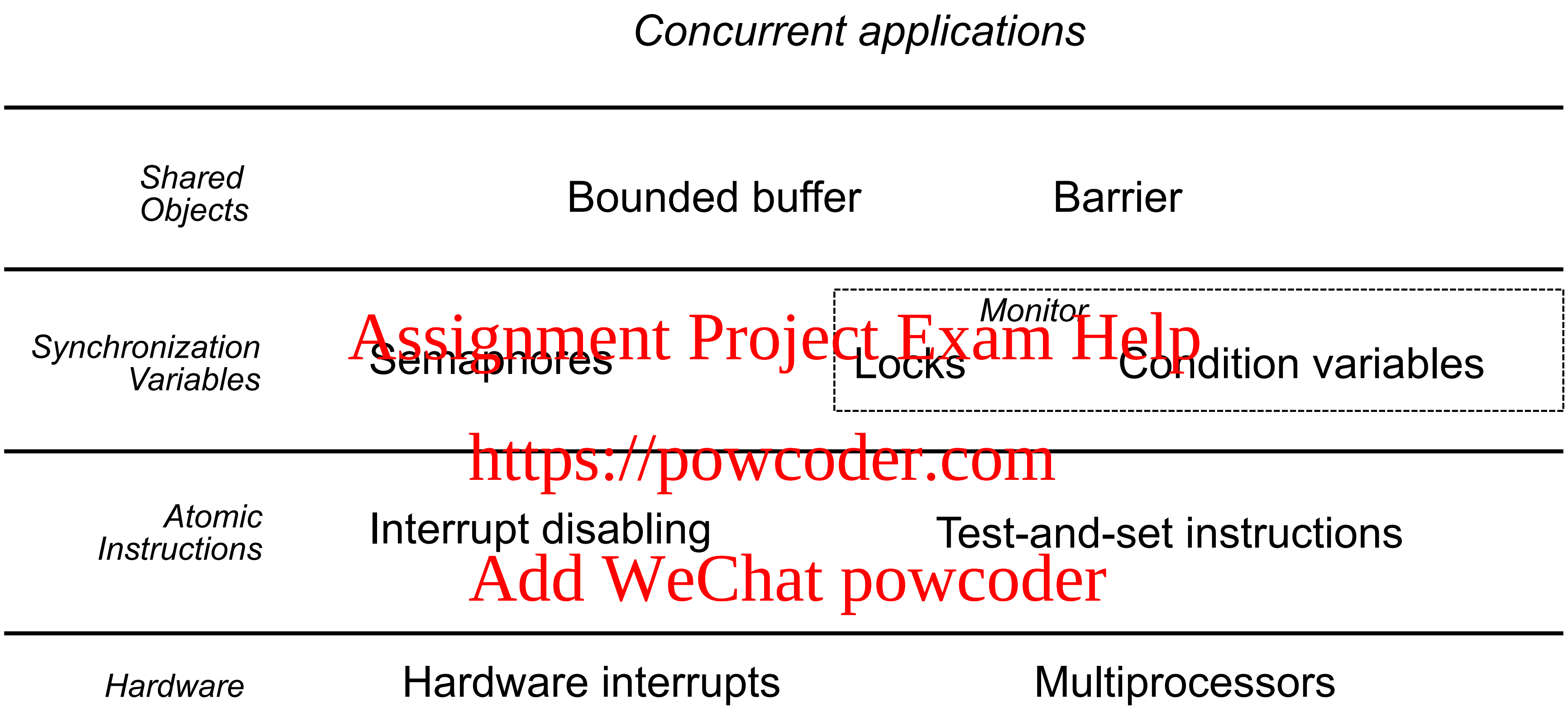
```
void main(void)
{
    barrier_t b = barrier_create(3);
    thread_create(thread_func, b);
    thread_create(thread_func, b);
    thread_create(thread_func, b);
}

void thread_func(barrier_t b)
{
    while (/* condition */) {
        /* ... do some computation ... */
        ...
        /* Wait for the other threads */
        barrier_wait(b);
    }
}
```



Implementation

- Using semaphores
- Using condition variables and the `broadcast()` feature

Synchronization: the big picture



Best practices

- Basic Threads Programming:Standards and Strategy
- The 12th Commandments of Synchronization (Cornell University, 2011)