DCS216 Operating Systems

Lecture 14
Synchronization (3)

Apr 15th, 2024

Instructor: Xiaoxi Zhang

Sun Yat-sen University

- **Content**
  - Monitors
    - Condition Variables
  - Synchronization within the Kernel
  - POSIX Synchronization
  - Alternative Approaches

Problems with Semaphores

- Although semaphores provide a convenient and effective mechanism for synchronization, using them incorrectly can result in errors that are difficult to detect.
- Incorrect use of semaphore operations:
- Examples 1: Suppose that a program interchanges the order in which the `wait()` and `signal()` operations on the semaphore `mutex` are executed, resulting in the following execution (**mutual exclusion** no longer holds):

```
signal(mutex);
...
// critical section
...
wait(mutex);
```

Problems with Semaphores

- Although semaphores provide a convenient and effective mechanism for synchronization, using them incorrectly can result in errors that are difficult to detect.
- Incorrect use of semaphore operations:
- Examples 2: Suppose that a program replaces `signal()` with `wait()` : (thread will permanently block on 2nd call to `wait()`)

```
wait(mutex);
...
// critical section
...
wait(mutex);
```

Problems with Semaphores

- Although semaphores provide a convenient and effective mechanism for synchronization, using them incorrectly can result in errors that are difficult to detect.
- Incorrect use of semaphore operations:
- Examples 3: the same API (`sem_wait`, `sem_post`) for different purposes leads to confusion

```
void* producer(void* arg) {
    int next_produced;
    for (int i = 0; i < 10; i++) {
        next_produced = i;
        sem_wait(&empty); // Decrement empty count
        sem_wait(&mutex);
        // Add the item to the buffer
        buffer[in] = next_produced;
        in = (in + 1) % BUFFER_SIZE;
        printf("--> Produced %d. ", next_produced);
        printBuffer();

        sem_post(&mutex);
        sem_post(&full);
    }
}
```

Problems with Semaphores

- Although semaphores provide a convenient and effective mechanism for synchronization, using them incorrectly can result in errors that are difficult to detect.
- Incorrect use of semaphore operations:
- Examples 3: the same API (`sem_wait`, `sem_post`) for different purposes leads to confusion

Programmers can get confused and interchange their orders by mistake without noticing, leading to deadlocks or other bugs.

```
void* producer(void* arg) {
    int next_produced;
    for (int i = 0; i < 10; i++) {
        next_produced = i;
        sem_wait(&mutex);
        sem_wait(&empty); // Decrement empty count
        // Add the item to the buffer
        buffer[in] = next_produced;
        in = (in + 1) % BUFFER_SIZE;
        printf("--> Produced %d. ", next_produced);
        printBuffer();

        sem_post(&mutex);
        sem_post(&full);
    }
}
```

Problems with Semaphores

- Although semaphores provide a convenient and effective mechanism for synchronization, using them incorrectly can result in errors that are difficult to detect.
- Incorrect use of semaphore operations
- **Solution**: (a cleaner separation of functions)
  - Avoid using semaphores for *both* **Mutual Exclusion** and **Scheduling Constraints**
  - Use Locks for **Mutual Exclusion**
  - Use Condition Variables for **Scheduling Constraints**

- **Monitors (管程)**
  - To deal with such errors caused by incorrect use of semaphores, one strategy is to incorporate simple synchronization tools as high-level constructs.

Monitors (管程)

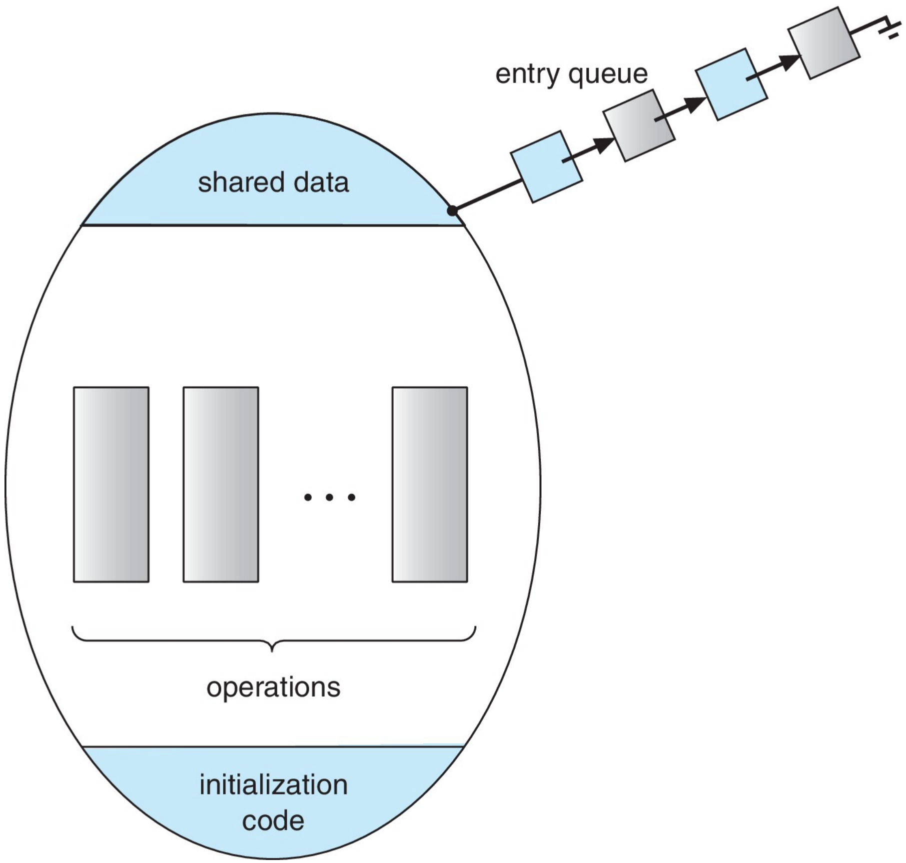
- To deal with such errors caused by incorrect use of semaphores, one strategy is to incorporate simple synchronization tools as high-level constructs.
- **Monitor:** A high-level abstraction that provides a convenient and effective mechanism for synchronization
- A Monitor type is an **Abstract Data Type** (**ADT**) that includes a set of programmer-defined operations that are provided with mutual exclusion within the monitor.

```
Monitor monitor_name {
    /* shared variable declarations */

    function P1 (...) { ... }
    function P2 (...) { ... }
    function P3 (...) { ... }

    initialization_code (...) {...}
}
```

- **Monitors (管程)**
  - Sharing: A monitor is shared by concurrent processes/threads
  - Encapsulation & Security: The representation of a monitor cannot be used directly by the processes/threads
    - Only the functions defined within a monitor can access the shared variables declared within the monitor
  - Mutual Exclusion: The monitor construct ensures that only one process/thread at a time is active within the monitor.
    - Programmers does not need to code this synchronization explicitly
    - Shared data are protected by placing them within the monitor
    - The monitor locks shared data upon process entry

Monitors

- Schematic View of a Monitor



Condition Variables

- With only Mutex Locks and Semaphores, the **monitor** is not sufficiently powerful for modeling some synchronization schemes.
- Sometimes, we wish to check whether a **condition** is true before we continue its execution.

```
Monitor bounded_buffer {
    /* shared variables */
    Condition not_full;
    Condition not_empty;

    void Producer() {
        wait(not_full);
        ...
        signal(not_empty);
    }
    void Consumer() {
        wait(not_empty);
        ...
        signal(not_full);
    }
}
```

Condition Variables

- With only Mutex Locks and Semaphores, the **monitor** is not sufficiently powerful for modeling some synchronization schemes.
- Sometimes, we wish to check whether a **condition** is true before we continue its execution.

```
Monitor bounded_buffer {
    /* shared variables */
    Condition not_full;
    Condition not_empty;

    void Producer() {
        wait(not_full);
        ...
        signal(not_empty);
    }
    void Consumer() {
        wait(not_empty);
        ...
        signal(not_full);
    }
}
```

```
Producer should wait when the
buffer is full. It should continue
only if (not_full == true)
```

Condition Variables

- With only Mutex Locks and Semaphores, the **monitor** is not sufficiently powerful for modeling some synchronization schemes.
- Sometimes, we wish to check whether a **condition** is true before we continue its execution.

```
Monitor bounded_buffer {
    /* shared variables */
    Condition not_full;
    Condition not_empty;

    void Producer() {
        wait(not_full);
        ...
        signal(not_empty);
    }
    void Consumer() {
        wait(not_empty);
        ...
        signal(not_full);
    }
}
```

After the Producer creates an item, it should notify (signal) the Consumer that the buffer is now not_empty.

Condition Variables

- With only Mutex Locks and Semaphores, the **monitor** is not sufficiently powerful for modeling some synchronization schemes.
- Sometimes, we wish to check whether a **condition** is true before we continue its execution.

```
Monitor bounded_buffer {
    /* shared variables */
    Condition not_full;
    Condition not_empty;

    void Producer() {
        wait(not_full);
        ...
        signal(not_empty);
    }
    void Consumer() {
        wait(not_empty);
        ...
        signal(not_full);
    }
}
```

After the Producer creates an item, it should notify (signal) the Consumer that the buffer is now not_empty, potentially unblocking Consumer.

Condition Variables

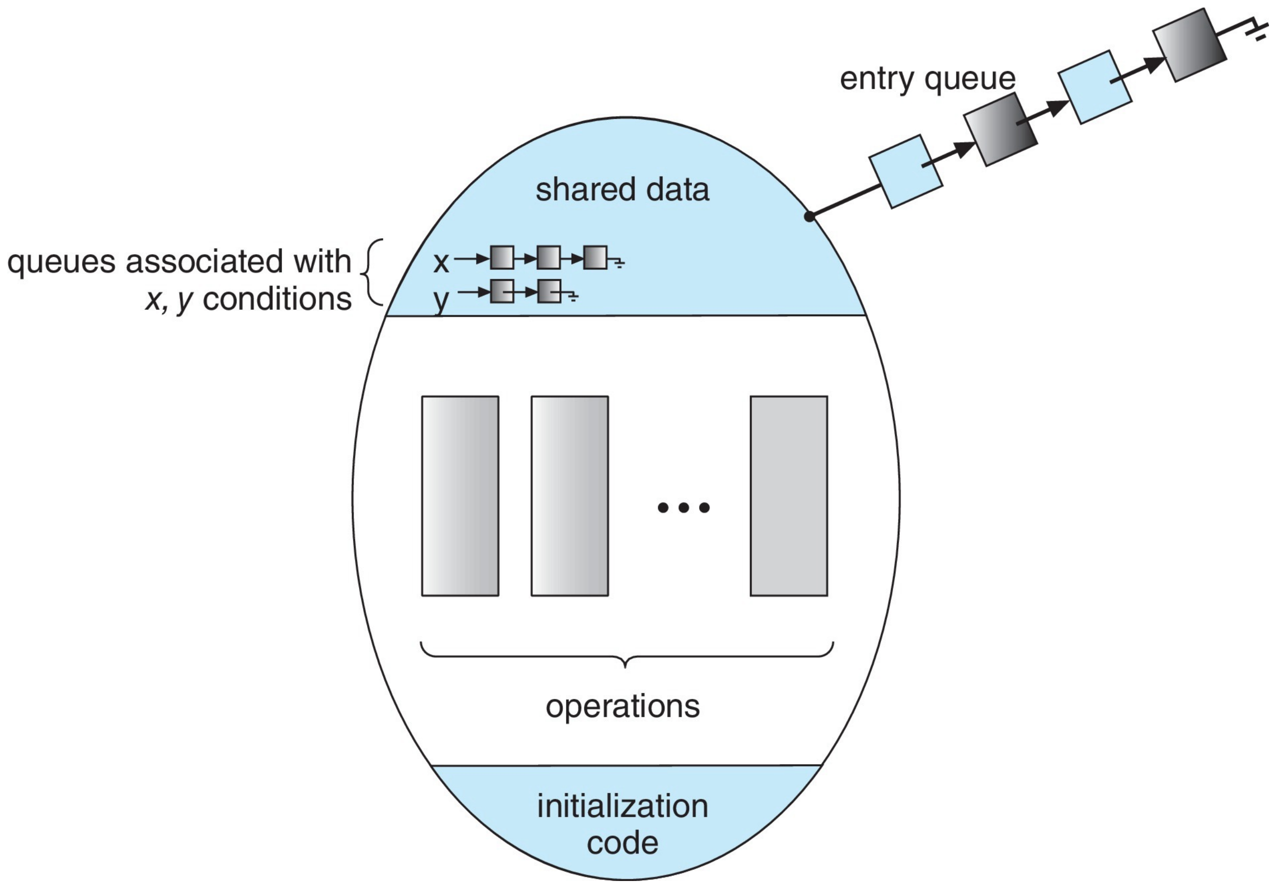
- `condition x, y;`
- Two operations are allowed on a condition variable:
  - `x.wait()` : the calling thread that invokes `wait()` is suspended
    - ...until `x.signal()`.
  - `x.signal()` : resumes (wakes up) one of the threads (if any)
    - ...that invoked `x.wait()`
    - If no `x.wait()` on the condition variable x, then it has no effect on condition variable x.
    - In contrast to ``**semaphore**.signal()``, which always affects the **state** of the **semaphore**.
    - In other words, condition variables are stateless, while semaphores are stateful.

Condition Variables

- `condition x, y;`
- Two operations are allowed on a condition variable:
  - `x.wait()` : the calling thread that invokes `wait()` is suspended
    - ...until `x.signal()`.
  - `x.signal()` : resumes (wakes up) one of the threads (if any)
    - ...that invoked `x.wait()`
    - If no `x.wait()` on the condition variable x, then it has no effect on condition variable x.
    - In contrast to ``semaphore.signal()``, which always affects the **state** of the **semaphore**.
    - In other words, condition variables are stateless, while semaphores are stateful.
  - `x.broadcast()` : resumes all threads that invoked `x.wait()`.
    - Often seen in other implementations, such as `pthread_cond_broadcast()` and Java Threads API

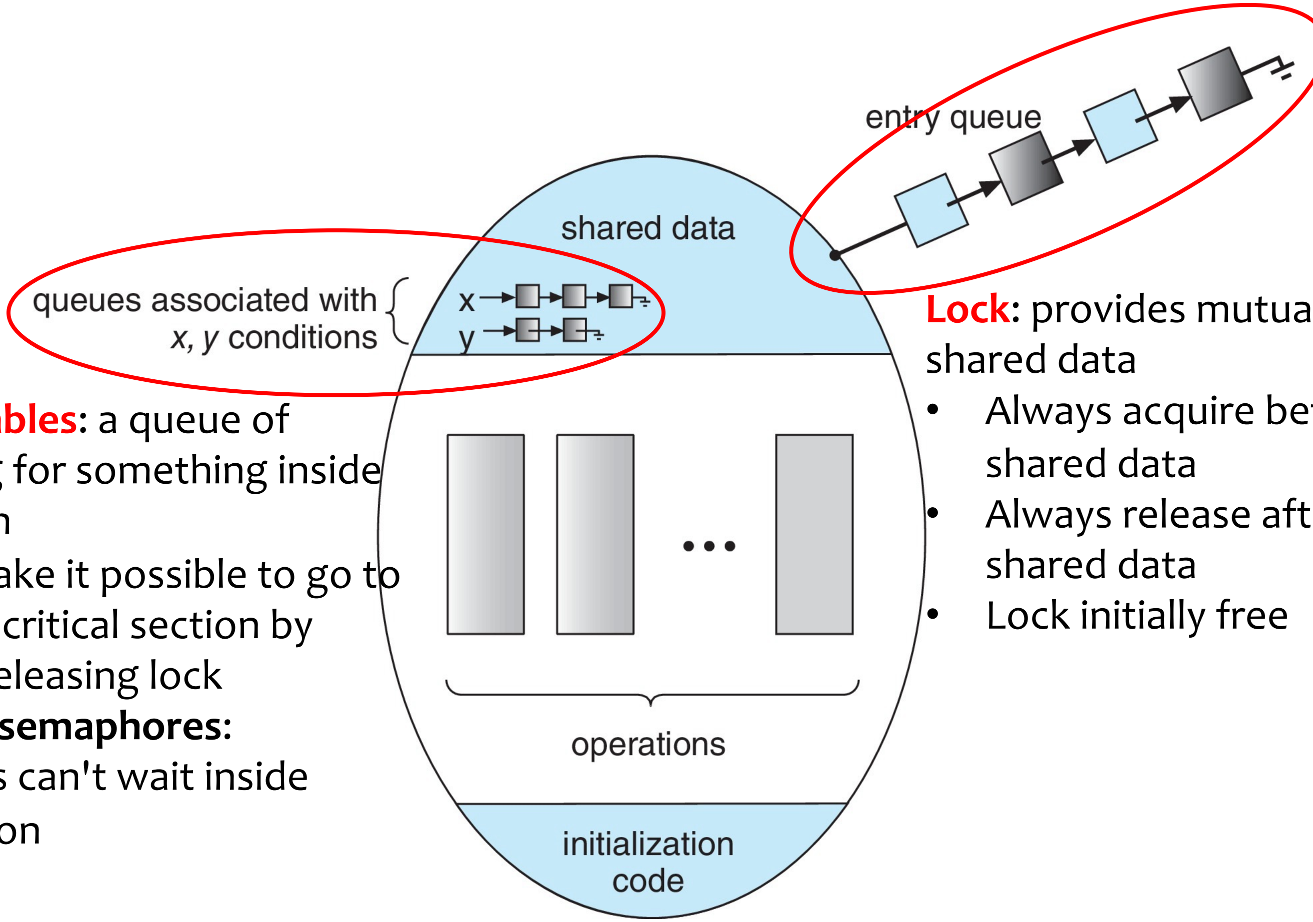
Monitors

- Schematic View of a Monitor with Condition Variables



Monitors

Schematic View of a Monitor with Condition Variables



Condition Variables: a queue of threads waiting for something inside a critical section

- **Key idea:** make it possible to go to sleep inside critical section by atomically releasing lock
- **Contrast to semaphores:** Semaphores can't wait inside critical section

Lock: provides mutual exclusion to shared data

- Always acquire before accessing shared data
- Always release after finishing with shared data
- Lock initially free

Monitor: a lock with zero or more condition variables for managing concurrent access to shared data.

POSIX API for Condition Variables

- Data type:
  - `pthread_cond_t cond;`
- Basic operations:
  - `pthread_cond_wait(&cond, &mutex);`
    - Atomically put current thread to block on the condition variable `cond` and release the *mutex lock* `mutex`
  - `pthread_cond_signal(&cond);`
    - Wake up one of the threads (if any) that blocks on `cond`.
  - `pthread_cond_broadcast(&cond);`
    - Wake up all threads (if any) that block on `cond`.

POSIX Condition Variables Example

```c
/* parent_wait.c */
#include <stdio.h>
#include <unistd.h>
#include <pthread.h>

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

int main() {
    pthread_t p;
    printf("Parent BEGIN\n");
    pthread_create(&p, NULL, child, NULL);
    // How to wait for child?
    printf("Parent END\n");
    sleep(1);
    return 0;
}
```

```
$ ./parent_wait
Parent BEGIN
Parent END
Child
```

POSIX Condition Variables Example

```c
/* parent_wait2.c */
#include <stdio.h>
#include <unistd.h>
#include <pthread.h>

int done = 0;

void *child(void *arg) {
    printf("Child\n");
    // How to indicate we are done?
    done = 1;
    return NULL;
}
int main() {
    pthread_t p;
    printf("Parent BEGIN\n");
    pthread_create(&p, NULL, child, NULL);
    pthread_join(p, NULL);
    // How to wait for child?
    while (done == 0)
        ;     // Spin
    printf("Parent END\n");
    sleep(1);
    return 0;
}
```

```
$ ./parent_wait2
Parent BEGIN
Child
Parent END
```

1. The most natural approach is to use a shared global variable done as a flag to indicate whether child thread is done.
2. Before child process terminate, set `done = 1`
3. In the parent thread, keep checking the value of done before proceeding.

Disadvantage: Hugely inefficient as the parent spins and wastes CPU cycles.

POSIX Condition Variables Example

```c
/* parent_wait3.c */
#include <stdio.h>
#include <unistd.h>
#include <pthread.h>

void *child(void *arg) {
    printf("Child\n");
    // How to indicate we are done?
    pthread_exit(NULL);
}

int main() {
    pthread_t p;
    printf("Parent BEGIN\n");
    pthread_create(&p, NULL, child, NULL);
    // How to wait for child?
    pthread_join(p, NULL);
    printf("Parent END\n");
    sleep(1);
    return 0;
}
```

```
$ ./parent_wait3
Parent BEGIN
Child
Parent END
```

Remember `pthread_join()` and `pthread_exit()`?

Obviously, they solve this problem perfectly.

But, how are they implemented under the hood?

POSIX Condition Variables Example

```
/* parent_wait4.c */
#include <stdio.h>
#include <unistd.h>
#include <pthread.h>

void *child(void *arg) {
    printf("Child\n");
    // How to indicate we are done?
    my_thread_exit();
}

int main() {
    pthread_t p;
    printf("Parent BEGIN\n");
    pthread_create(&p, NULL, child, NULL);
    // How to wait for child?
    my_thread_join();
    printf("Parent END\n");
    sleep(1);
    return 0;
}
```

Here, we demonstrate a simple implementation of `thread_exit()` and `thread_join()` using Monitors, which is basically {**lock** + **condition variables**}.

POSIX Condition Variables Example

```c
/* parent_wait4.c */
#include <stdio.h>
#include <unistd.h>
#include <pthread.h>

void *child(void *arg) {
    printf("Child\n");
    // How to indicate we are done?
    my_thread_exit();
}

int main() {
    pthread_t p;
    printf("Parent BEGIN\n");
    pthread_create(&p, NULL, child, NULL);
    // How to wait for child?
    my_thread_join();
    printf("Parent END\n");
    sleep(1);
    return 0;
}
```

```
$ ./parent_wait4
Parent BEGIN
Child
Parent END
```

```c
int done = 0;
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t cond = PTHREAD_COND_INITIALIZER;

void my_thread_exit() {
    pthread_mutex_lock(&mutex);
    done = 1;
    pthread_cond_signal(&cond);
    pthread_mutex_unlock(&mutex);
}
void my_thread_join() {
    pthread_mutex_lock(&mutex);
    while (done == 0)
        pthread_cond_wait(&cond, &mutex);
    pthread_mutex_unlock(&mutex);
}
```

POSIX Condition Variables Example

```c
/* parent_wait4.c */
#include <stdio.h>
#include <unistd.h>
#include <pthread.h>

void *child(void *arg) {
    printf("Child\n");
    // How to indicate we are done?
    my_thread_exit();
}

int main() {
    pthread_t p;
    printf("Parent BEGIN\n");
    pthread_create(&p, NULL, child, NULL);
    // How to wait for child?
    my_thread_join();
    printf("Parent END\n");
    sleep(1);
    return 0;
}
```

```
$ ./parent_wait4
Parent BEGIN
Child
Parent END
```

```c
int done = 0;
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t cond = PTHREAD_COND_INITIALIZER;

void my_thread_exit() {
    pthread_mutex_lock(&mutex);
    done = 1;
    pthread_cond_signal(&cond);
    pthread_mutex_unlock(&mutex);
}
void my_thread_join() {
    pthread_mutex_lock(&mutex);
    while (done == 0)
        pthread_cond_wait(&cond, &mutex);
    pthread_mutex_unlock(&mutex);
}
```

Question: Does it spin?

POSIX Condition Variables Example

```c
int done = 0;
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t cond = PTHREAD_COND_INITIALIZER;

void my_thread_exit() {
    pthread_mutex_lock(&mutex);
    done = 1;
    pthread_cond_signal(&cond);
    pthread_mutex_unlock(&mutex);
}
void my_thread_join() {
    pthread_mutex_lock(&mutex);
    while (done == 0)
        pthread_cond_wait(&cond, &mutex);
    pthread_mutex_unlock(&mutex);
}
```

```c
int done = 0;
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t cond = PTHREAD_COND_INITIALIZER;

void my_thread_exit() {

    done = 1;
    pthread_cond_signal(&cond);

}
void my_thread_join() {

    while (done == 0)
        pthread_cond_wait(&cond, &mutex);

}
```

Is the Mutex Lock necessary?

POSIX Condition Variables Example

```
// Parent Thread                // Child Thread

// call thread_join()
while (done == 0) {
--------------------------------------------------
                                // call thread_exit()
                                done == 1;

                                pthread_cond_signal();
                                // no effect, since no
                                // thread is waiting
--------------------------------------------------
  pthread_cond_wait();
  // waits forever
}
```

```
int done = 0;
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t cond = PTHREAD_COND_INITIALIZER;

void my_thread_exit() {

    done = 1;
    pthread_cond_signal(&cond);

}
void my_thread_join() {

    while (done == 0)
        pthread_cond_wait(&cond, &mutex);

}
```

In **Monitors,** it is mandatory to acquire the lock before any operation on condition variables. After `wait()` or `signal()`, we must release the lock.

Remember, always hold the lock while `signal()` or `wait()`.

Mesa vs. Hoare Semantics

- Why do we `CV::wait()` inside a **while** loop?

```
void my_thread_join() {
    pthread_mutex_lock(&mutex);
    while (done == 0)
        pthread_cond_wait(&cond, &mutex);
    pthread_mutex_unlock(&mutex);
}
```

- Why can't we do this (check only once with an **if** statement)?

```
void my_thread_join() {
    pthread_mutex_lock(&mutex);
    if (done == 0)
        pthread_cond_wait(&cond, &mutex);
    pthread_mutex_unlock(&mutex);
}
```

- **Answer**: depends on the semantics of Monitor/Condition Variable.
  - **Hoare** Semantics: `CV::signal()` is immediately (atomically) followed by `CV::wait()`
  - **Mesa** Semantics: No such guarantee.
    - Most modern OSes use Mesa Semantics!

Monitors

Hoare Semantics

- Signaler gives up lock, CPU to waiter; waiter runs immediately
- Then waiter gives up lock, CPU back to signaler when it exits critical section or if it waits again.

```c
void my_thread_exit() {
    pthread_mutex_lock(&mutex);
    done = 1;

    pthread_cond_signal(&cond);

    pthread_mutex_unlock(&mutex);

}
```

```c
void my_thread_join() {
    pthread_mutex_lock(&mutex);

    if (done == 0)
        pthread_cond_wait(&cond, &mutex);


    pthread_mutex_unlock(&mutex);
}
```

- On first glance, this seems like good semantics
  - Waiter gets to run immediately, condition is still correct!
- Most textbooks talk about Hoare semantics scheduling
  - However, hard to implement, not really necessary
  - Forces a lot of context switching (inefficient)

Mesa Semantics

- Signaler keeps lock and CPU
- Waiter placed on ready queue with no special priority

```
void my_thread_exit() {
    pthread_mutex_lock(&mutex);
    done = 1;

    pthread_cond_signal(&cond);

    pthread_mutex_unlock(&mutex);

}
```

Put waiting thread on ready queue

Schedule thread (sometime later)

```
void my_thread_join() {
    pthread_mutex_lock(&mutex);

    while (done == 0)
        pthread_cond_wait(&cond, &mutex);


    pthread_mutex_unlock(&mutex);
}
```

- Practically, need to check condition again after `wait()`
  - by the time the waiter gets scheduled, condition may be false again. So, just check again with the while loop
- Most Modern OSes adopt Mesa Semantics
  - Efficient, easier to implement
  - No need for special treatment in the Scheduler.

Monitors

- Monitors represent the synchronization logic of the program
  - Wait if necessary
  - Signal when change something so any waiting threads can proceed
- Typical structure of monitor-based program (Mesa Semanatics):

```
lock
while (need to wait) {
    cond.wait();
unlock
```

```
do something so no need to wait
lock

cond.signal();

unlock
```

Monitors

- Implementing a Monitor Using Semaphores
  - **Monitors**: a high-level synchronization construct that allows threads to have both mutual exclusion and the ability to wait(block) for a certain condition to become true. Monitors also automatically handle waking up threads when conditions change.
  - **Semaphores**: a low-level synchronization primitive that manages access to common resources by using counters. If a semaphore's counter is positive, accessing the resource decrements the counter. If the counter is zero, the thread attempting access is blocked.

Monitors

- Implementing a Monitor (**Hoare**) Using Semaphores

```c
#include <stdio.h>
#include <pthread.h>
#include <semaphore.h>
// Declare semaphores
sem_t mutex, next, sem_done;
int next_count = 0, count_done = 0;

void init_monitor() {
    // Binary semaphore for mutex
    sem_init(&mutex, 0, 1);
    // Binary semaphore for orderly access
    sem_init(&next, 0, 0);
    // Semaphore to signal child completion
    sem_init(&sem_done, 0, 0);
}
void enter_monitor() {
    sem_wait(&mutex);
}
void exit_monitor() {
    if (next_count > 0)
        sem_post(&next);
    else
        sem_post(&mutex);
}
```

```c
void wait_child_done() {
    count_done++;
    if (next_count > 0)
        sem_post(&next);
    else
        sem_post(&mutex);
    sem_wait(&sem_done);
    count_done--;
}
void signal_child_done() {
    if (count_done > 0) {
        next_count++;
        sem_post(&sem_done);
        sem_wait(&next);
        next_count--;
    }
}
```

Monitors

- Implementing a Monitor (**Hoare**) Using Semaphores

```c
void *child(void *arg) {
    enter_monitor();
    printf("Child\n");
    signal_child_done();
    exit_monitor();
    return NULL;
}

int main() {
    pthread_t p;

    init_monitor();

    pthread_create(&p, NULL, child, NULL);

    enter_monitor();
    printf("Parent BEGIN\n");
    wait_child_done();
    printf("Parent END\n");
    exit_monitor();

    pthread_join(p, NULL);
    return 0;
}
```

```c
void wait_child_done() {
    count_done++;
    if (next_count > 0)
        sem_post(&next);
    else
        sem_post(&mutex);
    sem_wait(&sem_done);
    count_done--;
}
void signal_child_done() {
    if (count_done > 0) {
        next_count++;
        sem_post(&sem_done);
        sem_wait(&next);
        next_count--;
    }
}
```

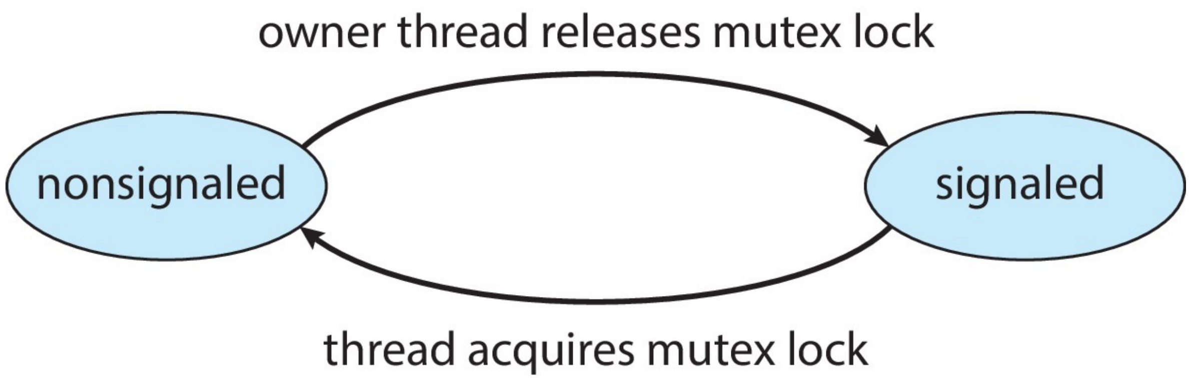
```
$ ./monitor
Parent BEGIN
Child
Parent END
```

Synchronization within the Kernel

- **Kernel Synchronization in Windows**
  - Uses interrupt masks to protect access to global resources on uniprocessor systems
  - Use spinlocks on multiprocessor systems
    - Spinlocking-thread will never be preempted
  - Also provides dispatch objects user-land which may act mutexes, semaphores, events, and timers
    - Events
      - An event acts much like a condition variable
    - Timers notify one or more thread when time quantum expired
    - Dispatcher objects either signaled-state (object available) or non-signaled state (thread will block)

Kernel Synchronization in Windows

- Mutex dispatcher object



Kernel Synchronization in Linux

- Prior to kernel version 2.6, disables interrupts to implement short critical sections
- Version 2.6 and later: fully preemptive
- Linux provides:
  - Semaphores
  - Atomic integers
  - Spinlocks
  - Reader-writer versions of both
- On single-processor systems, **spinlocks** replaced by enabling and disabling kernel **preemption**

Single Processor	Multiple Processors
Disable kernel preemption	Acquire spin lock
Enable kernel preemption	Release spin lock

Kernel Synchronization in Linux

- Atomic variables:
  - **atomic_t** is the type for atomic integer
- Consider the variables:

```
atomic_t counter;
int value;
```

Atomic Operation	*Effect*
atomic_set(&counter,5);	counter = 5
atomic_add(10,&counter);	counter = counter + 10
atomic_sub(4,&counter);	counter = counter - 4
atomic_inc(&counter);	counter = counter + 1
value = atomic_read(&counter);	value = 12

- **POSIX Synchronization**
  - POSIX Mutex Locks
  - POSIX Semaphores
  - POSIX Condition Variables

POSIX Synchronization

- POSIX Mutex Locks

```
#include <pthread.h>

pthread_mutex_t mutex;

/* Create and initialize the mutex lock */
pthread_mutex_init(&mutex, NULL);

/* Acquire the mutex lock */
pthread_mutex_lock(&mutex);


/* Critical section */


/* Release the mutex lock */
pthread_mutex_unlock(&mutex);
```

POSIX Synchronization

- POSIX Semaphores
  - **Named** Semaphores
  - **Unnamed** Semaphores

```c
#include <semaphore.h>

sem_t *sem;

/* Named Semaphores */
/* Create the semaphore and initialize it to 1 */
sem = sem_open("SEM", O_CREAT, 0666, 1);

/* Unnamed Semaphores */
/* Create the semaphore and initialize it to 1 */
sem_init(&sem, 0, 1);

/* Semaphore Wait() / Down() / P() */
sem_wait(sem);

/* Semaphore Signal() / Up() / V() */
sem_post(sem);
```

POSIX Synchronization

- POSIX Condition Variables

```
#include <pthread.h>

pthread_mutex_t mutex;
pthread_cond_t cond;

/* Initialize mutex lock */
pthread_mutex_init(&mutex, NULL);
/* Initialize condition variable */
pthread_cond_init(&cond, NULL);

/* Typical Monitor Construct: *wait* part */
pthread_mutex_lock(&mutex);
while (a != b)
    pthread_cond_wait(&cond, &mutex);
pthread_mutex_unlock(&mutex);

/* Typical Monitor Construct: *signal* part */
pthread_mutex_lock(&mutex);
a = b;
pthread_cond_signal(&cond);
pthread_mutex_unlock(&mutex);
```

Alternative Approaches

- **Alternative Approaches**
  - Transactional Memory
  - OpenMP
  - Functional Programming Languages

Alternative Approaches

Transactional Memory

- Consider a function `update()` that must be called atomically. One option is to use mutex locks:

```
void update ()
{
   acquire();

   /* modify shared data */

   release();
}
```

- However, using synchronization mechanisms such as mutex locks and semaphores involve many potential problems, including deadlock.
- Additionally, as the number of threads increases, traditional locking doesn't scale well, because the level of contention among threads for lock ownership becomes very high.

Transactional Memory

- A memory transaction is a sequence of read-write operations to memory that are performed **atomically**. A transaction can be completed by adding `atomic{S}` which ensure statements in S are executed atomically.

```
void update ()
{
   atomic {
      /* modify shared data */
   }
}
```

- Advantage over locks: the **transactional memory system**, rather than the developer, is responsible for guaranteeing atomicity.
- Transactional memory can be implemented in:
  - Software transactional memory (STM)
  - Hardware transactional memory (HTM)

Alternative Approaches

OpenMP

- OpenMP is a set of compiler directives and API that support parallel programming.

```
void update(int value) {
    #pragma omp critical
    {
        count += value;
    }
}
```

- The code contained within the `#pragma omp critical` directive is treated as a critical section and performed atomically.

Alternative Approaches

- **Functional Programming Languages**
  - Functional programming languages offer a different paradigm than procedural languages in that they do not maintain state.
  - Variables are treated as immutable and cannot change state once they have been assigned a value.
  - There is increasing interest in functional languages such as Erlang and Scala for their approach in handling data races.

Thank you!