Lecture.6.Synchronization.txt

- Why Synchronization?
    - If a program only uses one process, then synchronization is not needed
        - This is because all resources are dedicated to one process
    - Processes can be executed concurrently
        - Multiple applications allow sharing of resources
            - i.e. Sharing memory location, processing time, etc.
        - When working with multiple processes, synchronization is needed to prevent processes from writing to the same memory location
    - Synchronization allows processing time to be shared between a number of processes
    - Structured applications are an extension to modular design and structured programming
        - Usually done as a set of concurrent processes
    - Operating systems are implemented as a set of concurrent processes and procedures
        - This is the structure of all operating systems

- Background
    - Processes can execute concurrently
    - Processes can be interrupted at any time
        - This results in partially completing execution
        - When the process resumes, we need to ensure that it has all the data that it was using before it was stopped
    - Concurrent access to shared data may result in data inconsistency
        - The solution to this problem is synchronization
        - Concurrency issues are caused by multiple processes writing to shared memory
            - Reading the same memory location does not cause inconsistency
    - Maintaining data consistency requires mechanisms to ensure the orderly execution of cooperating processes

- Consumer Producer
    - i.e. Producer Example
      ```
      /*
       * The producer creates an item, puts it into the buffer, and
       *   then increases the counter. The counter is very important
       *   because it shows us how many items are in the buffer.
       * The `counter` variable is initialized to 0, and then
       *   incremented every time we add a new item to the buffer.
       * While `counter` is equal to the size of the buffer then
       *   nothing is done, because the buffer is full.
       * If the buffer is not full, the next item is put into the
       *   buffer, and the index of the buffer is increased.
       */
      ```

```
        while (true) {
            /* Produce an item in next produced */
            while (counter == BUFFER_SIZE); /* Do nothing */
            buffer[in] = next_produced;
            in = (in + 1) % BUFFER_SIZE;
            counter++;
        }
        // The `counter` variable is initialized to 0.
        // It's incremented every time we add a new item to the buffer
    - i.e. Consumer Example
        /*
         * The consumer needs to consume the elements before the
         *   buffer can be emptied. Otherwise, the buffer will be
         *   overwritten or flooded.
         * If the buffer is empty, then the consumer does nothing
         *   because there is nothing to consume.
         * If there is an item in the buffer, consume it, update the
         *   pointer, and decrement the counter.
         */
        while (true) {
            while (counter == 0); /* Do nothing */
                next_consumed = buffer[out];
                out = (out + 1) % BUFFER_SIZE;
                counter--;
                /* Consume the item in `next_consumed` */
        }
        // The `counter` variable is decremented every time we
        // remove one item from the buffer
    - The producer & consumer functions, above, are correct separately
        - But, may not function correctly when executed concurrently
            - Switching execution between producer and consumer can
              create problems
                - i.e.
                    - Suppose counter = 5
                    - The producer and consumer functions concurrently
                        execute the statements counter++ and counter--
                        - The value of `counter` may be 4, 5, or 6
                            - This is a race condition

- Race Condition (1)
    - The expression `counter++` could be implemented as:
        register1 = counter
        register1 = register1 + 1
        counter   = register1
    - The expression `counter--` could be implemented as:
        register2 = counter
        register2 = register2 - 1
        counter   = register2
    - Even though the expressions `counter++` and `counter--` are
      single C instructions, the compiler transforms them into 3
```

separate (lower level) instructions
  - The expressions `counter++` and `counter--` are NOT atomic instructions
    - An atomic instruction is an instruction that is either fully completed or not started, but NEVER partially completed
  - You can start and stop in the middle of `counter++` and `counter--`
    - Other (machine) instructions can be executed immediately after stopping in the middle of these increment and decrement instructions
      - This is the source of the problem
        - If the value of the counter is not updated immediately, then other parts of the program will be forced to work with an incorrect value
- In microprocessors, mathematical operations are done on the arithmetic logic unit
  - Computation takes place on the registers, and not the value in memory
    - Everything is done in the register
    - The ALU performs mathematical operations on the value that is in register
      - Then, the new value is written to memory

- Race Condition (2)
  - Consider the `counter++` and `counter--` instructions being executed in an interleave manner when `counter` equals 5
    - i.e.

```
S0: register1 = counter        {register1 = 5}
S1: register1 = register1 + 1  {register1 = 6}
-----------------------------------------------
S2: register2 = counter        {register2 = 5}
S3: register2 = register2 - 1  {register2 = 4}
-----------------------------------------------
S4: counter = register1        {counter = 6}
-----------------------------------------------
S5: counter = register2        {counter = 4}
```

    - The first two instructions from `counter++` are executed, then the first two instructions from `counter--` are executed. Finally the last instruction from `counter++` and `counter--` are executed, respectively
      - The final result of counter is 4
        - This is incorrect; the correct answer is 5
      - Alternatively, if the last instruction from `counter--` and `counter++` were executed, the final result of counter would be 6
        - This is also incorrect
    - This is a race condition, and can be a serious problem

- Race Condition (3)

    - A race condition occurs when several processes access and manipulate the same data concurrently, and the outcome of the execution depends on the particular order in which the access takes place
      - i.e. In the previous example, `counter` may be 4, 5, or 6
        - This is why it is difficult to discover race conditions; because the program may be working in some situations, and under a controlled environment, but other times it will not work
          - i.e. Code runs fine in-house, but fails during deployment or in the real world
    - To prevent race conditions, we need to ensure that only one process at a time can be manipulating the shared variable
      - i.e. Only `counter++` has access to `counter`, or only `counter--` has access to `counter`
      - To guarantee this, the processes need to be synchronized in some way

- Race Condition Example
    - Consider the following scenario:
      - Two processes, P0 and P1, are creating a child process using the 'fork()' system call at the same time
        - They both request `next_available_pid` and get a value of 2615
        - Now, both child processes from P0 and P1 will have the same PID
          - Unless there is mutual exclusion, the same PID could be assigned to two different processes
          - This is a problem because each process should have a unique PID
        - The race condition is caused by the kernel variable `next_available_pid`
          - When a process requests this variable, the other processes should not be allowed to access it

- Critical Section Problem Definition
    - Consider a system of 'n' processes; p(0), p(1), ..., p(n-1)
      - Each process has its own critical section segment of code
        - A process may be changing common variables, updating a table, writing to a file, etc.
        - When one process is in a critical section, no other process may be in its critical section
          - Critical sections are defined by the programmer
      - The critical section problem is to design protocol to solve this problem
      - Each process must ask permission to enter critical section in entry section, may follow critical section with exit section, then remainder section
        - i.e. Use a mutex lock at the start of a critical section, and then release the lock at the end of

the critical section

- Solution To Critical Section Problem
    - When protecting a critical section, it is not enough to ensure that multiple processes are not accessing the critical section at the same time
    - There are three things that should be satisfied to ensure that there is no critical section problem; they are:
        1. Mutual Exclusion
            - If process 'P_i' is executing in its critical section, then no other processes can be executing in 'P_i's critical section
        2. Progress
            - If no process is executing in its critical section, and there exists some process that wish to enter that critical section, then the selection of the processes that will enter the critical section next cannot be postponed indefinitely
                - A process needs to give up exclusivity of the critical section when it is done using it
                    - i.e. Putting a tool back in the shed after you are done using it
                    - i.e. Getting off the bench press when you are done lifting weights
                - If a process does no give up the critical section, the other processes won't be able to progress
        3. Bounded Waiting
            - A bound must exist on the number of times that other processes are allowed to enter the critical section after a process has made a request to enter the critical section, and before that request is granted
                - i.e. Assume there are 3 processes. The first two processes are more important than the third one; they have a higher priority. The third process won't be able to enter a critical section because the first two processes have a higher priority.
                - Bounded waiting puts a limit on the number of times a process can enter the critical section
                    - This prevents high priority processes from restricting a lower priority process from accessing a critical section
                        - With bounded waiting, a low priority process does not have to wait forever to access a critical section
            - Bounded waiting solves the starvation problem

- Critical Section Handling In OS
    - There are two approaches to handling critical sections in the OS
        - Both depend on if the kernel is pre-emptive or non pre-

emptive
  1. Preemptive
      - Preemptive critical section handling in operating systems allows preemption of process when running in kernel mode
  2. Non-preemptive
      - Runs until exists kernel mode, blocks, or voluntarily yields CPU

- Peterson's Solution (1)
  - This solution is elegant, but it does not always work
    - It is a two process solution
  - Assume that the `load` and `store` machine-language instructions are atomic
    - An atomic instruction cannot be interrupted; once the execution has begun, nothing else can be started until it has completed
  - In Peterson's solution, two processes share two variables:
    - int turn
      - Indicates whose turn it is to enter the critical section
    - boolean flag[2]
      - Can be true or false
      - Indicates if a process is ready to enter the critical section
        - flag[i] = true implies that process P_i is ready

- Algorithm For Processes P1 & P2
  - i.e. Process P0

```
while (true) {
    flag[0] = true;
    turn = 1;
    while (flag[1] && turn == 1);
    /* Critical section */

    flag[0] = false;

/* Remainder section */

}
```

  - i.e. Process P1

```
while (true) {
    flag[1] = true;
    turn = 0;
    while (flag[0] && turn == 0);
    /* Critical section */

    flag[1] = false;

/* Remainder section */
```

```
        }
    - The algorithm stalls the process if flag[j] is set to true
        - flag[j] corresponds to a different process
            - i.e. If flag[1] in Process P0 is true, then P0 cannot
                   enter the critical section

- Peterson's Solution (2)
    - The three critical section requirements are met:
        - Mutual exclusion is preserved
            - 'P_i' enters critical section only if either:
               flag[j] == false OR turn == i
        - Progress requirement is satisfied
            - Once a process has finished with the critical section,
              another process is allowed to enter
        - Bounded waiting (no deadlock) requirements are met

- Peterson's Solution (3)
    - Peterson's Solution is not guaranteed to work on modern
      (multithreaded) architectures
        - This is because processors and/or compilers may reorder
          operations that have no dependencies
            - This is done to improve performance
            - i.e. The statement `x,y := a,b` may be executed as:
                   x := a;
                   y := b;
                     OR
                   y := b;
                   x := a;
            - Since there is no dependency in this instruction, the
              compiler may decide to reorder it
            - For single-threaded this is acceptable, but for multi-
              threaded this is not acceptable because the reordering
              may produce inconsistent or unexpected results

- Peterson's Solution (4)
    - Consider the following:
        - Two threads share the same data:
            boolean flag = false;
            int x = 0;
        - Thread 1:
            while (!flag);
            print x;
        - Thread 2:
            x = 100;
            flag = true;
    - What is the expected output?
        - Output is 100, because Thread #1 is waiting for flag to be
          true
    - If thread 2 is reordered:
        flag = true;
```

    x = 100;
  - The expected output of this is 0
  - The effects of instruction reordering in Peterson's solution is that two processes, P0 and P1, may enter the critical section at the same time
    - The reordering of instructions may be done by the compiler

- Synchronization Hardware
  - Another way to solve the synchronization problem is hardware support for implementing the critical section code
    - There are three forms of hardware support:
      1. Memory barriers
      2. Hardware instructions
      3. Atomic variables
  - In a uniprocessor, interrupts could be disabled
    - The result is that currently running code would execute without preemption
    - However, this approach is too inefficient on multiprocessor systems, and it is very hard to scale the operating system when using this approach

- Memory Barriers (1)
  - Memory models are the memory guarantees a computer architecture makes to application programs
  - An instruction that forces any change in memory to be propagated (made visible) to all other processors
    - If other processors are not aware of modified memory, then they will think that the original value is still there, and continue to work with the wrong value
      - But if they are aware of the change, then they will know that the information in memory is no longer valid
  - There are two types of memory models:
    1. Strongly Ordered
       - Memory modification of one processor is immediately visible to all other processors
    2. Weakly Ordered
       - Memory modification of one processor may not be immediately visible to other processors

- Memory Barriers (2)
  - A memory barrier can be added to the following instructions to ensure Thread 1 outputs 100
    - i.e.

```
// Thread 1
while (!flag) {
    memory_barrier();
}
print(x);

// Thread 2
```

```
            x = 100;
            memory_barrier();
            flag = true;
        - The 'memory_barrier()' function ensures that the assignment
          to `x` occurs before the assignment to `flag` in Thread 2
            - Even if the compiler thinks that the `x = 100;` and
              `flag = true;` instructions can be executed in reverse
              order to improve throughput, the order will be kept as
              it is written
            - Guarantees the order of assignment for `x` and `flag`
        - The 'memory_barrier()' function guarantees that the value of
          `flag` is loaded before the value of `x` in Thread 1
            - Ensures that the output of Thread 1 is 100
    - This method does not solve the critical section problem
        - All it does is prevent the reordering of instructions during
          execution

- Hardware Instructions
    - Some hardware instructions to address the critical section
      problem are:
        - test_and_set()
            - Test and modify the content of a word
        - compare_and_swap()
            - Swap the contents of two words
    - These instructions are executed atomically
        - In other words, their execution is not interrupted; once
          execution starts, it does not stop until completion

- Test And Set Instruction
    - Executed atomically
        - Once `test_and_set()` has been called, nothing can interrupt
          its execution
    - Returns the original value of passed parameter, and sets the new
      value of passed parameter to 'true'
    - i.e.
        boolean test_and_set (boolean *target) {
            boolean rv = *target;
            *target = true;
            return rv;
        }

- Mutual Exclusion Implementation For Test And Set
    - The `test_and_set()` instruction can be implemented in the
      Mutual exclusion problem
    - i.e.
        // Assume shared boolean variable `lock` is `false`
        do {

            while (test_and_set(&lock)); // Do nothing
```

```
            /* Critical section */

            lock = false;

            /* Remainder section */

        } while (true);
```

  - If two `test_and_set()` instructions are executed simultaneously (each on a different core), they will be executed sequentially in some arbitrary order
    - They won't enter each other's critical section

- Compare And Swap Instruction
  - Executed atomically
  - Returns the original value of passed parameter `value`
  - `value` is set to `new_value` only if `*value == expected` is true
  - i.e.

```
    int compare_and_swap(int *value, int expected,
                                     int new_value) {
        int temp = *value;

        if (*value = expected) {
            *value = new_value;
        }

        return temp;
    }
```

- Mutual Exclusion Implementation For Compare And Swap
  - The `compare_and_swap()` instruction can be implemented in the Mutual exclusion problem
  - i.e.

```
    // Assume shared integer `lock` is initialized to 0
    while (true) {
        while (compare_and_swap(&lock, 0, 1) != 0); // Do nothing

        /* Critical section */

        lock = 0;

        /* Remainder section */
    }
```

  - The first process that invokes `compare_and_swap()` will set lock to 1
    - It will then enter its critical section, because the original value of lock was equal to the expected value of 0

- Mutual Exclusion Implementation
  - Although the previous algorithm satisfies the mutual exclusion

requirement, it does not satisfy the bounded waiting requirement
  - Recall from the previous slide that there are 3 requirements to the critical section problem
    1. Mutual Exclusion
    2. Progress
    3. Bounded Waiting
- i.e.

```
while (true) {
    waiting[i] = true;
    key = 1;
    while (waiting[i] && key == 1) {
        key = compare_and_swap(&lock, 0, 1);
    }
    waiting[i] = false;

    /* Critical section */

    j = (i + 1) % n;

    while ((j != i) && !waiting[j]) {
        j = (j + 1) % n;
    }

    if (j == i) {
        lock = 0;
    } else {
        waiting[j] = false;
    }

    /* Remainder section */
}
// `compare_and_swap()` is an atomic instruction
// There are more conditions to satisfy before the critical
//   section can be entered
// Bounded waiting ensures that every process waiting to
//   enter the critical section will eventually get access.
//   In the example above, the process after `i` will get
//   access to the critical section because `waiting[j] ==
//   false`, and `j` is the next process
// `n` is number of processes
// `i` is the current process
// `j` is used to set waiting process to false
```

- Atomic Variables (1)
  - Typically, instructions such as `compare_and_swap()` are used as building blocks for other synchronization tools
  - One tool is an 'atomic variable' that provides atomic (uninterruptible) updates on basic data types such as integers and booleans
    - In other words, once an operation on a variable has started,

  it cannot be interrupted by anything
- i.e. The `increment()` operation on the atomic variable `sequence` ensures incrementation without interruption
  - i.e. increment(&sequence);
- Recall from earlier that the `counter++;` function could be implemented as 3 smaller instructions
  - The processor may switch context and execute other instructions before completing the `counter++` instruction
  - Using an atomic variable on the `counter++` and `counter--` example from earlier will solve the potential race condition

- Atomic Variables (2)
  - The `increment()` function can be implemented as follows:
    - i.e.

```
void increment(atomic_int *v) {
    int temp;

    do {
        temp = *v;
    } while (temp != (compare_and_swap(*v, temp,
                                       temp +1)));
}
// This example shows how to use atomic variables to avoid
//   encountering a race condition
```

  - Although atomic variables provide atomic updates, they do not entirely solve race conditions in all circumstances
    - i.e. If there are multiple producers and multiple consumers, and they are looping, waiting for `count` to equal 0. At the same time, both will get information that `count` is greater than zero, and they may start at the same time to consume that element

- Mutex Locks
  - Previous solutions are complicated and generally inaccessible to application programmers
    - Programmers should not have to dissect functions, understand what loops are doing, and how things are changed
      - The ideal solution is to give programmers a simple instruction that solves the critical section problem, and encapsulates the complexity of the solution
  - Operating system designers build software tools to solve the critical section problem
    - The simplest solution is 'mutex lock'
      - i.e. If something wants to access a critical section, the first step is to `acquire()` the lock, and then `release()` it when exiting the critical section
      - Critical sections are protected by `acquire()` and 'release()' functions
        - Only one thread may be able to have the mutex lock

            at a time
        - Calls to `acquire()` and `release()` must be atomic
            - They are usually implemented via hardware atomic
              instructions such as `compare_and_swap`
- The mutex lock solutions requires 'busy waiting'
    - If a critical section cannot be accessed, then the thread
      continues to wait until the lock is acquire-able
    - Busy waiting consumes computing power
        - Thus, mutex locks are not an efficient way to solve the
          critical problem
        - This type of lock is called spinlock

- Mutex Lock Example
    - i.e.

```
while (true) {
    |--------------|
    | Acquire Lock |
    |--------------|
       // Critical section
    |--------------|
    | Release Lock |
    |--------------|
       // Remainder section
}
// `acquire()` and `release()` are atomic instructions that
//   are implemented by the hardware. These functions are
//   very reliable because they have been extensively tested
//   and proven to work
// Entering the critical section requires the lock, which
//   must be acquired and it must be available
// When exiting the critical section, the lock must be
//   released via `release()`
```

- Mutex Lock Definitions
    - A thread waiting to enter the critical section is called 'busy
      waiting'
        - This is not an efficient solution because waiting can take
          a long time, and use a lot of computing power
    - i.e.

```
acquire() {
    while (!available); // Busy wait //
    available = false;
}

release() {
    available = true;
}
```

    - The `acquire()` and `release()` functions must be implemented
      atomically
        - Both `test_and_set()` and `compare_and_swap()` can be used

      to implement these functions
  - There needs to be a way to avoid 'busy waiting'
    - i.e. Do something else instead of checking the status of the lock all the time

- Semaphore
  - Synchronization tool that provides most sophisticated ways, than Mutex locks, for processes to synchronize their activities
    - May help avoid 'busy waiting'
    - Can be used on both threads and processes
  - Semaphores are represented by an integer variable called S
    - Can be accessed via two atomic (indivisible) operations:
      1. wait()
         - Specifies that a process is waiting for a resource
      2. signal()
         - Specifies that the resource is available by using semaphore
  - i.e.

```
wait(S) {
    while (S <= 0); // Busy wait //
    S--;
}

signal(S) {
    S++;
}
```

  - The `wait()` function was originally termed 'P'
    - From the Dutch word 'proberen', which means: 'to test'
  - The `signal()` function was originally called 'V'
    - From the Dutch word 'verhogen', which means: 'to increment'
  - Edsger W. Dijkstra was a Dutch scientist
    - He contributed a lot of knowledge and research to the field of computer science; especially semaphores
    - One of his famous statements is: "By testing you cannot prove correctness. All you can do is find a problem"
      - This caused computer science to shift from testing to formal proofs

- Semaphore Usage
  - When constructing a semaphore, some portion of the code, or process, is executed before another
    - i.e. Consider P_1 and P_2 that require S_1 to happen before S_2. Create a semaphore `synch` initialized to 0
      - i.e.

```
P1:
    S1;
    signal(synch);
P2:
    wait(synch);
    S2;
```

        - The `wait()` and `signal()` instructions guarantee that S_1 is executed before S_2
        - The semaphore `synch` is (initially) initialized to 0
    - Guarantees that no two processes can execute the `wait()` and `signal()` functions on the same semaphore at the same time
    - There are two kinds of semaphores:
      1. Counting semaphore
          - Integer value can range over an unrestricted domain
      2. Binary semaphore
          - Integer value can range only between 0 and 1
              - Same as a mutex lock

- Semaphore Implementation
    - The semaphore implementation using `wait()` and `signal()` still suffers from the 'busy waiting' problem
      - When a thread is stuck in a `wait()` loop, it is doing nothing other than checking the value of `S`
          - This is a waste of computing power
      - To overcome this problem, this operation needs to be suspended
          - i.e. Enter some kind of sleep mode until we get some information about the availability of resources
    - To solve the 'busy wait' problem, the definition of `wait()` and `signal()` operations need to be modified as follows:
      - When a process executes the `wait()` operation, and finds that the semaphore value is not positive, it must wait
      - Rather than engaging in 'busy waiting', the process can suspend itself
      - The suspended operation places a process into a waiting queue associated with the semaphore, and the state of the process is switched to the waiting state
      - Then, control is transferred to the CPU scheduler, which selects another process to execute

- Semaphore Implementation With No Busy Waiting
    - All processes are put into a waiting state
      - The process scheduler decides what process is executed next
          - Could be based on some kind of priority
              - i.e. How long process is been sitting in the queue, How much CPU time does the process need, etc.
    - With each semaphore there is an associated waiting queue
      - Each entry in a waiting queue has two data items:
          - value
              - Of type integer
          - pointer
              - Points to the next record in the list
      - i.e. Waiting Queue Structure

```
typedef struct {
    int value;
    struct process *list;
```

```
            } semaphore;
```

- Implementation With No Busy Waiting
    - The following is an implementation of `wait()` and `signal()` to fix the 'busy waiting' problem
        - 'busy waiting' wastes processing power/time
            - Hence, it is not a good design
    - i.e.

```
        // Wait:
        wait (semaphore *S) {
            S->value--;
            if (S->value < 0) {
                add this process to S->list;
                sleep();
            }
        }

        // Signal:
        signal(semaphore *S) {
            S->value++;
            if (S->value <= 0) {
                remove a process P from S->list;
                wakeup(P);
            }
        }
        // The semaphore `S` is now a pointer to a struct that holds
        //   a value and a pointer
        // Instead of waiting in a while loop, the process is added to
        //   a wait list, and then enters sleep mode via `sleep()`
        // When resources become available, the process is removed
        //   from the waiting list, and executed
```

    - The `sleep()` and `wait()` functions are different

- Problems With Semaphores
    - Some issues with semaphores are related to implementation
        - Using semaphores incorrectly can result in timing errors that are difficult to detect
            - Sometimes the error may show up, and sometimes it may not
        - i.e.

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

            // Replaces `signal()` with `wait()`
            wait(mutex);
                ...
```

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

        // Mutual exclusion is violated if `wait(mutex)` and/or
        //   `signal(mutex)` is omitted
        // The compiler does not complain or warn about this
        // If other threads require the critical section, then a
        //   problem may appear (i.e. process is blocked)
```
      - Either mutual exclusion is violated or the process will be permanently blocked

- Monitors
  - Are a high-level abstraction that provide a convenient and effective mechanism for process synchronization
    - They are an abstract data type (ADT)
      - Internal variables are only accessible by code within the procedure
    - Only one process may be active within the monitor at a time
      - When a process is inactive, it leaves the monitor
  - i.e.

```
    monitor monitor_name {
        // Shared variable declarations
        function P1 (...) {...}
        function P2 (...) {...}
            ...
        function Pn (...) {...}

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

- Schematic View Of A Monitor
  - All processes are put into an entry queue
  - The monitor construct ensures that only one process at a time is active/running within the monitor
    - Only one process can access the critical section at a time
    - Consequently, the programmer does not need to code this synchronization constraint explicitly
      - This is part of the design of the monitor

- Condition Variables
  - A programmer that needs to write a tailor-made synchronization scheme can define one or more variables of type `condition`
    - i.e. condition x, y;
  - Two operations are allowed on a condition variable:
    - x.wait()
      - A process that invokes this operation is suspended until `x.signal()`
    - x.signal()
      - Resumes one of processes (if any) that invoked

`x.wait()`
  - If there is no `x.wait()` on the variable, then it has no effect (on the variable)

- Monitor With Condition Variables
  - A monitor with condition variables helps in creating a tailor-made synchronization scheme
  - Condition variables are shared data
    - Multiple processes can share same condition variable
  - Monitor with `condition` variables example: https://www.youtube.com/watch?v=15Q8PILXkQ0

- Condition Variables Choices
  - Some ambiguous situations may arise when using condition variables
    - i.e. If process 'P' invokes `x.signal()`, and process Q is suspended in `x.wait()`, what should happen next?
      - Since only one process can run in the monitor, there are two different options that can occur:
        - Signal and wait
          - 'P' either waits until 'Q' leaves the monitor or, it waits for another condition
        - Signal and continue
          - 'Q' waits until 'P' leaves the monitor, or it waits for another condition
        - Both approaches have pros and cons
          - The implementation depends on the programming language
            - In C, we use signal and wait; first option
            - Monitors implemented in concurrent Pascal can be compromised
              - 'P' executing `signal()` immediately leaves the monitor, and Q is resumed
            - Monitors are implemented in many languages including C#, Java, etc.

- Monitor Implementation Using Semaphores
  - For the signal and wait scheme:
    - An additional binary semaphore `next` is introduced
    - `next_count` counts the number of suspended processes
    - i.e. Example code

```
semaphore mutex;    // initially = 1
semaphore next;     // initially = 0
int next_count = 0;

// The semaphores are structures (defined in the previous
//   slides)
```

    - Each function 'F' will be replaced by:
      - i.e. Example code

```
wait(mutex)
```

```
            ...
        body of F;
            ...
        if (next_count > 0) {
            signal(next);
        } else {
            signal(mutex);
        }

        // Multiple processes are executed
        // Only one process can be in the monitor at a time
```
  - Mutual exclusion within a monitor is ensured

- Monitor Implementation Using Condition Variables
  - For each condition variable `x`, we have:
    - i.e. Example code
      ```
      semaphore x_sem; // Initially = 0
      int x_count = 0;
      ```
    - i.e. `x.wait()` code
      ```
      x_count++;
      if (next_count > 0) {
          signal(next);
      } else {
          signal(mutex);
      }
      wait(x_sem);
      x_count--;
      ```
    - i.e. `x.signal()` code
      ```
      if (x_count > 0) {
          next_count++;
          signal(x_sem);
          wait(next);
          next_count--;
      }
      ```
    ```
    // This ensures that there will never be a critical section
    // problem, because only one process is running at a time
    ```

- Resuming Processes Within A Monitor
  - If several processes queued on condition variable `x`, and `x.signal()` is executed, which process should be resumed first?
    - Usually implemented using First Come, First Served (FCFS)
      - FCFS may not be adequate in some cases
        - i.e. Process with higher priority are resumed first
    - When priority is an issue:
      - A conditional wait construct of the form `x.wait(c)` is used, where:
        - `c` is a priority number
          - The process with the lowest number (highest priority) is scheduled (to run) next

- Single Resource Allocation
    - If a single resource needs to be acquired, the `acquire()` and `release()` functions are used
        - `acquire()` accepts an argument that specifies the maximum time the process needs to be executed
    - Allocate a single resource among competing processes using priority numbers that specify the maximum time a process plans to use the resource
        - i.e. Example code
            ```
            R.acquire(t);
                ...
            Access the resource;
                ...
            R.release;
            ```
        - 'R' is an instance of type `ResourceAllocator`
            - It contains the `acquire()` and `release()` methods

- A Monitor To Allocate Single Resource
    - i.e. Example code
        ```
        monitor ResourceAllocator {
            boolean busy;
            condition x;

            void acquire(int time) {
                if (busy) {
                    x.wait(time);
                }
                busy = true;
            }

            void release() {
                busy = false;
                x.signal();
            }

            initialization_code() {
                busy = false;
            }
        }
        ```
    - When a resource is acquired, the parameter specifies the maximum time that the resource is needed
        - Processes that require little time with a resource may have higher priority
    - When the `release()` function is called, the resource(s) are once again available

- Liveness (1)
    - Processes may have to wait indefinitely while trying to acquire a synchronization tool such as MuTex lock or semaphore
        - This is not acceptable because it violates the progress

      and bounded waiting criteria
  - Liveness refers to a set of properties that a system must satisfy to ensure processes make progress
    - Indefinite waiting is an example of liveness failure

- Liveness (2)
  - A deadlock is caused when two or more processes are waiting indefinitely for an event that can be caused by only one of the waiting processes
  - i.e. Let P0 and P1 be two processes that are using 2 semaphores, S and Q, respectively
    - i.e. P0 process

```
// P0 code
wait(S);
wait(Q);
...
signal(S);
signal(Q);
```

    - i.e. P1 process

```
// P1 code
wait(Q);
wait(S);
...
signal(Q);
signal(S);
```

    - If P0 executes `wait(S)` and P1 executes `wait(Q)`, then when P0 executes `wait(Q), it must wait until P1 executes `signal(Q)`
      - P1 is waiting until P0 executes `signal(S)`
      - Both processes are stuck, waiting on each other to do something
        - Since the `signal()` functions are never executed, P0 and P1 are deadlocked

- Liveness & Other Forms Of Deadlock
  - Starvation is indefinite blocking process
    - A process may never be removed from the semaphore queue in which it is suspended
      - i.e. The priority of the process is to low for the scheduler to execute it, and processes with higher priority get access
    - This violates the bounded waiting criteria
  - Priority inversion is used to solve starvation
    - However, it can cause another scheduling problem where a lower priority process holds a lock needed by a higher priority process
      - This is solved via priority-inheritance protocol
  - A system should not have deadlock, starvation, priority inversion, etc.
    - All of these are signs of poor design

- Priority Inheritance Protocol
    - Consider the processes: P1, P2, and P3
        - P1 has the highest priority, P2 is the second highest, and P3 has the lowest priority
            - Assume that P3 is assigned a resource 'R' that P1 wants
            - Thus, P1 must wait for P3 to finish using the resource
            - However, P2 becomes runnable and preempts P3
                - The result is that P2, a process with lower priority than P1, has indirectly prevented P1 from gaining access to the resource 'R'
    - To prevent starvation due to priority inversion, a priority inheritance protocol is used
        - It allows the priority of the highest thread (P1) waiting to access a shared resource to be assigned to the thread currently using the resource (P3)

- Priority Inversion Example
    - The Mars Pathfinder had a problem due to priority inversion
        - The highest priority task was not able to run
            - This triggered the watchdog timer, causing the system to reset itself
        - Even though the system was designed by some of the best engineers, this problem still occurred

- End
    - Operating Systems are mong the most complex pieces of software ever developed

- Q/A
    - Q: What is the difference between a mutex and semaphore?
        - A: Essentially they are the same, but semaphores give you more control. Also, binary semaphores are the same as a mutex. Counting semaphores are more powerful than mutexes because you can control more processes, and have more integer values