Synchronization

Bojan Nokovic

Based on: "Operating Systems Concepts", 10th Edition Silberschatz Et al.
"Slides 3SH3 '12" - Sanzheng Qiao
"Operating Systems: Internals and Design Principles" - 9e Edition Stallings

Jan. 2021

Why Synchronization ?

Processes can execute concurrently.

Multiple Applications - allow processing time to be shared.

Structured Application - extension of modular design and structured programming a set of concurrent processes.

OS Structure - OS themselves implemented as set of processes and procedures.

Background

Processes can execute concurrently.

May be interrupted at any time, partially completing execution.

Concurrent access to shared data may result in data inconsistency.

Maintaining data consistency requires mechanisms to ensure the orderly execution of cooperating processes

Consumer-Producer

Producer:

```
while (true) {
        /* produce an item in next produced */

        while (counter == BUFFER_SIZE)
                ; /* do nothing */
        buffer[in] = next_produced;
        in = (in + 1) % BUFFER_SIZE;
        counter++;
}
```

Variable, `counter`, initialized to 0. It is incremented every time we add a new item to the buffer

Consumer-Producer

Consumer:

```
while (true) {
        while (counter == 0)
                ; /* do nothing */
        next_consumed = buffer[out];
        out = (out + 1) % BUFFER_SIZE;
        counter--;
        /* consume the item in next consumed */
}
```

Variable, `counter`, is decremented every time we remove one item from the buffer.

Consumer-Producer

The producer and consumer routines shown above are correct separately, but may not function correctly when executed concurrently.

Suppose `counter = 5`

Producer and consumer processes concurrently execute the statements `counter++` and `counter--`

Value of the variable `counter` may be 4, 5, or 6!

Race Condition

"`counter++`" could be implemented as

```
register1 = counter
register1 = register1 + 1
counter = register1
```

"`counter--`" could be implemented as

```
register2 = counter
register2 = register2 - 1
counter = register2
```

Race Condition

Consider this execution interleaving with `counter = 5` initially

```
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}
```

We have arrived at the incorrect state `counter == 4`

What would be the value of `counter` if we revised the order of S4 and S5?

Race Condition

A situation where 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, is called a race condition.

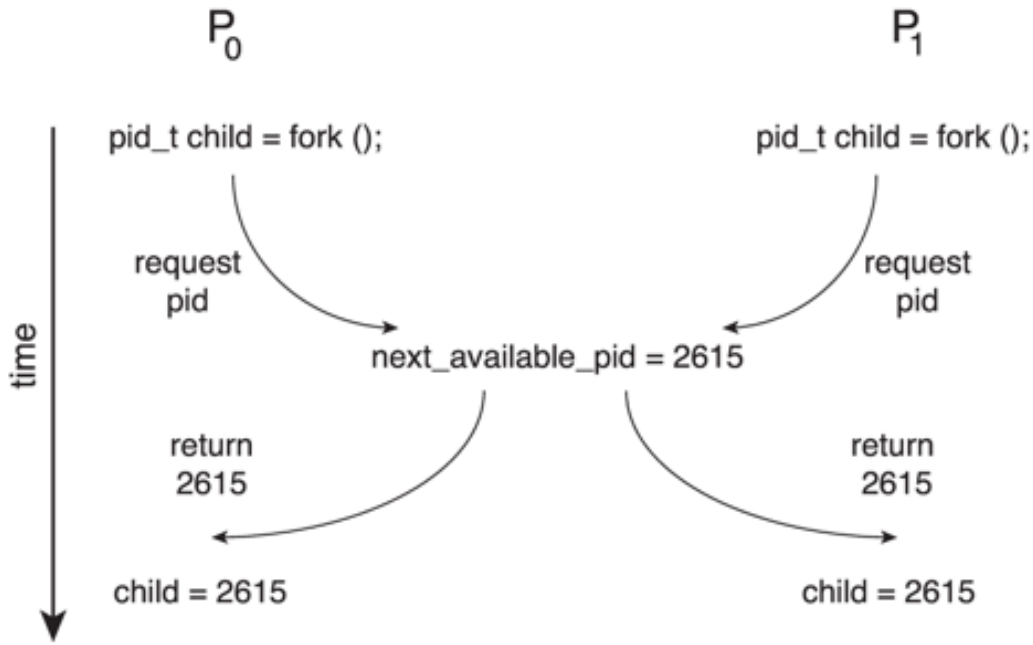
To guard against the race condition above, we need to ensure that only one process at a time can be manipulating the variable `counter`.

To make such a guarantee, we require that the processes be synchronized in some way.

Race Condition - Example

Processes P0 and P1 are creating child processs using the `fork()` system call

Race condition on kernel variable `next_available_pid` which represents the next available process identifier (pid)



Unless there is mutual exclusion, the same pid could be assigned to two different processes!

Critical Section Problem Definition

Consider system of n processes p_0, p_1 ... p_{n-1} Each process has critical section segment of code

- Process may be changing common variables, updating table, writing file, etc.
- When one process is in critical section, no other may be in its critical section! 

Critical section problem is to design protocol to solve this.

Each process must ask permission to enter critical section in entry section, may follow critical section with exit section, then remainder section.

Solution to Critical-Section Problem

1) Mutual Exclusion - If process P_i is executing in its critical section, then no other processes can be executing in their critical sections.

Solution to Critical-Section Problem

1) Mutual Exclusion - If process P_i is executing in its critical section, then no other processes can be executing in their critical sections.
2) Progress - If no process is executing in its critical section and there exist some processes that wish to enter their critical section, then the selection of the processes that will enter the critical section next cannot be postponed indefinitely.

Solution to Critical-Section Problem

1) Mutual Exclusion - If process P_i is executing in its critical section, then no other processes can be executing in their critical sections.
2) Progress - If no process is executing in its critical section and there exist some processes that wish to enter their critical section, then the selection of the processes that will enter the critical section next cannot be postponed indefinitely.
3) Bounded Waiting - A bound must exist on the number of times that other processes are allowed to enter their critical sections after a process has made a request to enter its critical section and before that request is granted.

Critical-Section Handling in OS

Two approaches depending on if kernel is preemptive or nonpreemptive.

a) Preemptive - allows preemption of process when running in kernel mode
b) Non-preemptive - runs until exits kernel mode, blocks, or voluntarily yields CPU

Peterson's Solution

Two process solution

Assume that the `load` and `store` machine-language instructions are atomic; that is, cannot be interrupted

The two processes share two variables

- `int turn`
- `boolean flag[2]`

The variable `turn` indicates whose turn it is to enter the critical section.

The `flag` array is used to indicate if a process is ready to enter the critical section. `flag[i] = true` implies that process P_i is ready!

Algorithm for Processes P_1 and P_2

Process P_0

```
while (true){
  flag[0] = true;
  turn = 1;
  while (flag[1] && turn == 1)
        ;

  /* critical section */

  flag[0] = false;

/* remainder section */
}
```

Process P_1

```
while (true){
  flag[1] = true;
  turn = 0;
  while (flag[0] && turn == 0)
        ;

  /* critical section */

  flag[1] = false;

/* remainder section */
}
```

Peterson's Solution

Provable that the three Critical-Section requirements are met:

- ✓ Mutual exclusion is preserved
  P_i enters CS only if:
  either `flag[j] == false` or `turn == i`
- ✓ Progress requirement is satisfied.
- ✓ Bounded-waiting (no deadlock) requirement is met.

Peterson's Solution

Peterson's Solution is not guaranteed to work on modern (multithreaded) architectures.

The primary reason: To improve performance, processors and/or compilers may reorder operations that have no dependencies.

The statement `x,y := a,b` may be executed either as

```
x := a;
y := b;
```

or

```
y := b;
x := a;
```

For single-threaded this is ok, but for multithreaded the reordering may produce inconsistent or unexpected results!

Peterson's Solution

Two threads share the data:

```
boolean flag = false;
int x = 0;
```

Thread 1

```
while (!flag)
;
print x;
```

Thread 2

```
x = 100;
flag = true;
```

- What is the expected output?

Peterson's Solution

Two threads share the data:

```
boolean flag = false;
int x = 0;
```

Thread 1

```
while (!flag)
;
print x;
```

Thread 2

```
x = 100;
flag = true;
```

- What is the expected output? 100

Peterson's Solution

If Tread 2 is reordered

```
flag = true;
x = 100;
```

- What is the expected output?

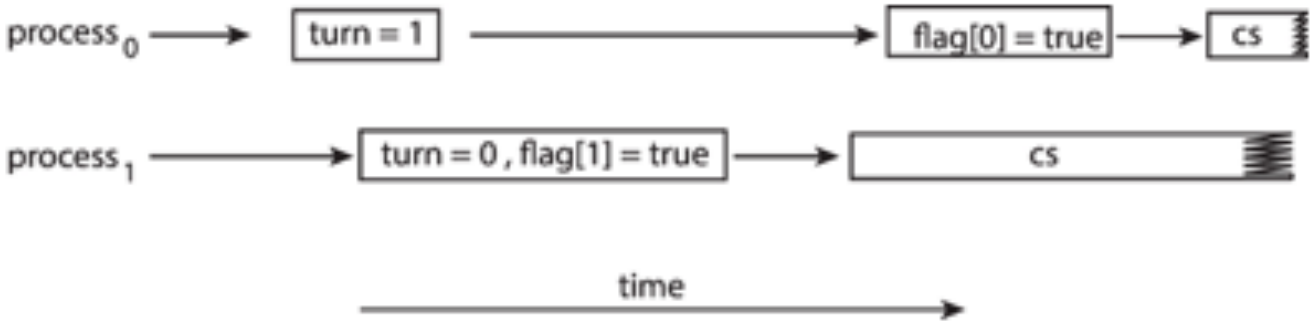
Peterson's Solution

If Tread 2 is reordered

```
flag = true;
x = 100;
```

- What is the expected output? 0!

The effects of instruction reordering in Peterson's Solution



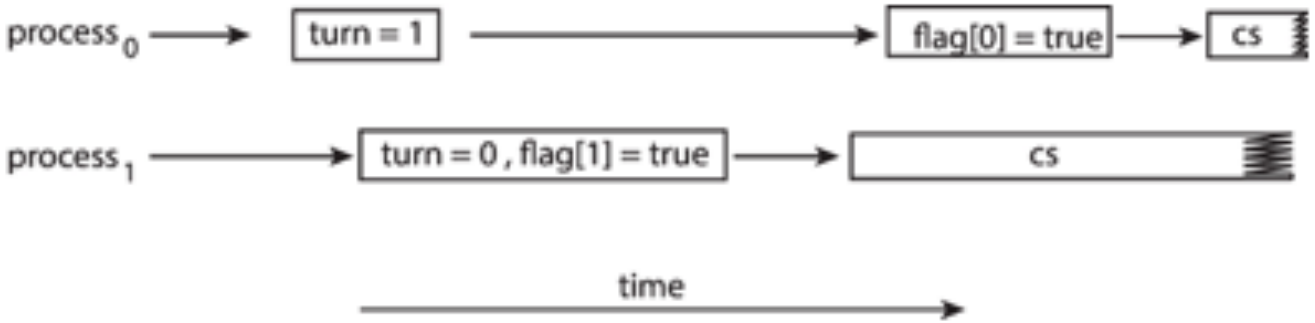
Peterson's Solution

If Tread 2 is reordered

```
flag = true;
x = 100;
```

- What is the expected output? 0!

The effects of instruction reordering in Peterson's Solution



Both processes to be in their critical section at the same time!

Synchronization Hardware

Hardware support for implementing the critical section code.

Uniprocessors could disable interrupts.

- Currently running code would execute without preemption.
- Generally too inefficient on multiprocessor systems; OS using this not broadly scalable.

Three forms of hardware support

1. Memory barriers
2. Hardware instructions
3. Atomic variables

Memory Barriers

Memory model are the memory guarantees a computer architecture makes to application programs.

Memory models may be either

- Strongly ordered - memory modification of one processor is immediately visible to all other processors.
- Weakly ordered - where a memory modification of one processor may not be immediately visible to all other processors.

A memory barrier is an instruction that forces any change in memory to be propagated (made visible) to all other processors.

Memory Barriers

We could add a memory barrier to the following instructions to ensure Thread 1 outputs 100

Thread 1

```
while (!flag)
 memory_barrier();
print x;
```

Guarantee that the value of `flag` is loaded before the value of `x`.

Thread 2

```
x = 100;
memory_barrier();
flag = true;
```

Ensure that the assignment to `x` occurs before the assignment to `flag`.

Hardware Instructions

Special hardware instructions that allow us to either

- test-and-modify the content of a word
- swap the contents of two words

atomically (uninterruptibly)

test_and_set() instruction
compare_and_swap() instruction

Test-and-Set Instruction

Definition:

```
boolean test_and_set (boolean *target)
{
  boolean rv = *target;
  *target = true;
  return rv;
}
```

Executed atomically

Returns the original value of passed parameter

Set the new value of passed parameter to `true`

Mutual-exclusion Implementation

```
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.

Shared `boolean` variable `lock`, initialized to `false`

Compare And Swap

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

  if (*value == expected)
    *value = new_value;
   return temp;
}
```

Executed atomically

Returns the original value of passed parameter `value`

`value` is set to `new_value` only if `*value == expected` is true.

Mutual-exclusion Implementation

Shared integer `lock` 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 previous algorithm satisfies the mutual-exclusion requirement, it does not satisfy the bounded-waiting requirement.

```
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 */
}
```

Atomic Variables

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

For example, the `increment()` operation on the atomic variable `sequence` ensures incrementation without interruption

```
increment(&sequence);
```

Atomic Variables

The `increment()` function can be implemented as follows:

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

    do {
        temp = *v;
    }
   while (temp != (compare_and_swap(*v,temp,temp+1));
}
```

Although atomic variables provide atomic updates, they do not entirely solve race conditions in all circumstances.

Example: `count` in producer-consumer when buffer is empty and two consumers are looping while waiting for `count > 0`

Mutex Locks

Previous solutions are complicated and generally inaccessible to application programmers

OS designers build software tools to solve critical section problem, simplest is mutex lock

Protect a critical section by first `acquire()` a lock then `release()` the lock

Calls to `acquire()` and `release()` must be atomic, usually implemented via hardware atomic instructions such as compare-and-swap.

But this solution requires busy waiting. This lock is therefore called spinlock.

Atomic Variables

```
while (true) {
    --------------
   | acquire lock |
    --------------
        critical section
    --------------
   | release lock |
    --------------
       remainder section
}
```

Mutex Lock Definitions

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

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

These two functions must be implemented atomically.

Both test-and-set and compare-and-swap can be used to implement these functions.

Strategy that avoid busy waiting ?

Semaphore

Synchronization tool that provides more sophisticated ways (than Mutex locks) for processes to synchronize their activities.

Semaphore S - integer variable; Can only be accessed via two indivisible (atomic) operations, `wait()` and `signal()`

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

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

`wait()` operation was originally termed P (from the Dutch *proberen*, "to test")
`signal()` was originally called V (from *verhogen*, "to increment").
`https://en.wikipedia.org/wiki/Edsger_W._Dijkstra`

Semaphore Usage

Counting semaphore - integer value can range over an unrestricted domain

Binary semaphore - integer value can range only between 0 and 1 (same as a mutex lock)

Consider P_1 and P_2 that require S_1 to happen before S_2. Create a semaphore "synch" initialized to 0

```
P1:
   S1;
   signal(synch);
```

```
P2:
   wait(synch);
   S2;
```

Guarantee that no two processes can execute the `wait()` and `signal()` on the same semaphore at the same time

Semaphore Implementation

The implementation still suffers from busy waiting problem.

To overcome this problem we need to modify the definition of the `wait()` and `signal()` operations 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 suspend 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

With each semaphore there is an associated waiting queue

Each entry in a waiting queue has two data items:

- value (of type integer)
- pointer to next record in the list

Two operations: block (or sleep) and wakeup

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

Implementation with no Busy waiting

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);
    }
}
```

Problems with Semaphores

Using semaphores incorrectly can result in timing errors that are difficult to detect.

Order interchanged

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

Replaces `signal` with `wait`

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

Omitting of `wait(mutex)` and/or `signal(mutex)`

Either mutual exclusion is violated or the process will permanently blocked.

Monitors

A high-level abstraction that provides a convenient and effective mechanism for process synchronization

Abstract data type, internal variables only accessible by code within the procedure

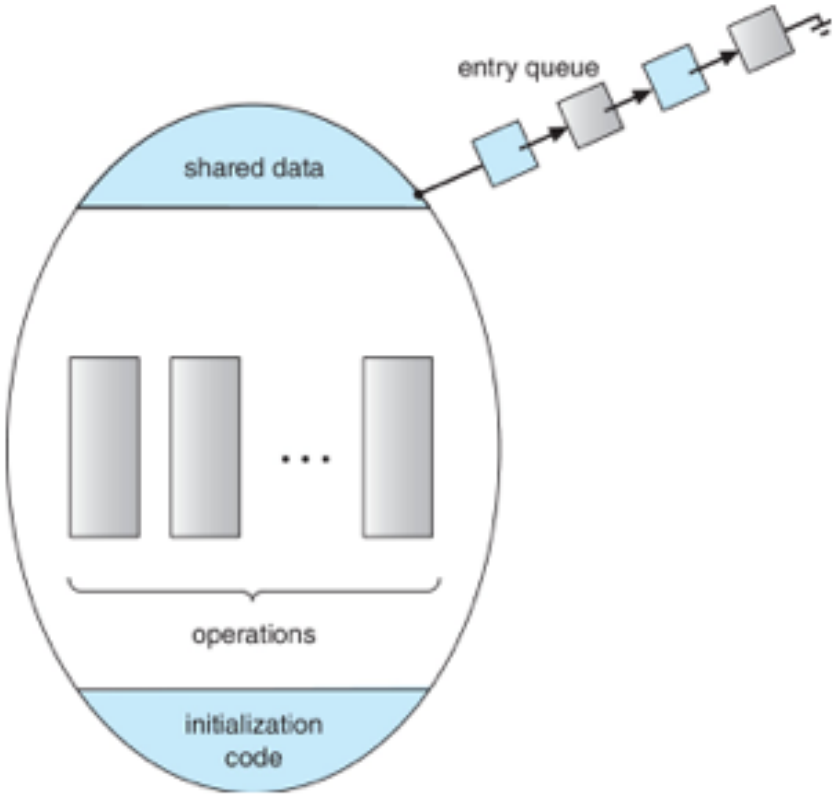
Only one process may be active within the monitor at a time 

```
monitor monitor_name
{
  // shared variable declarations

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

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

Schematic view of a Monitor



The monitor construct ensures that only one process at a time is active within the monitor.

Consequently, the programmer does not need to code this synchronization constraint explicitly.

Condition Variables

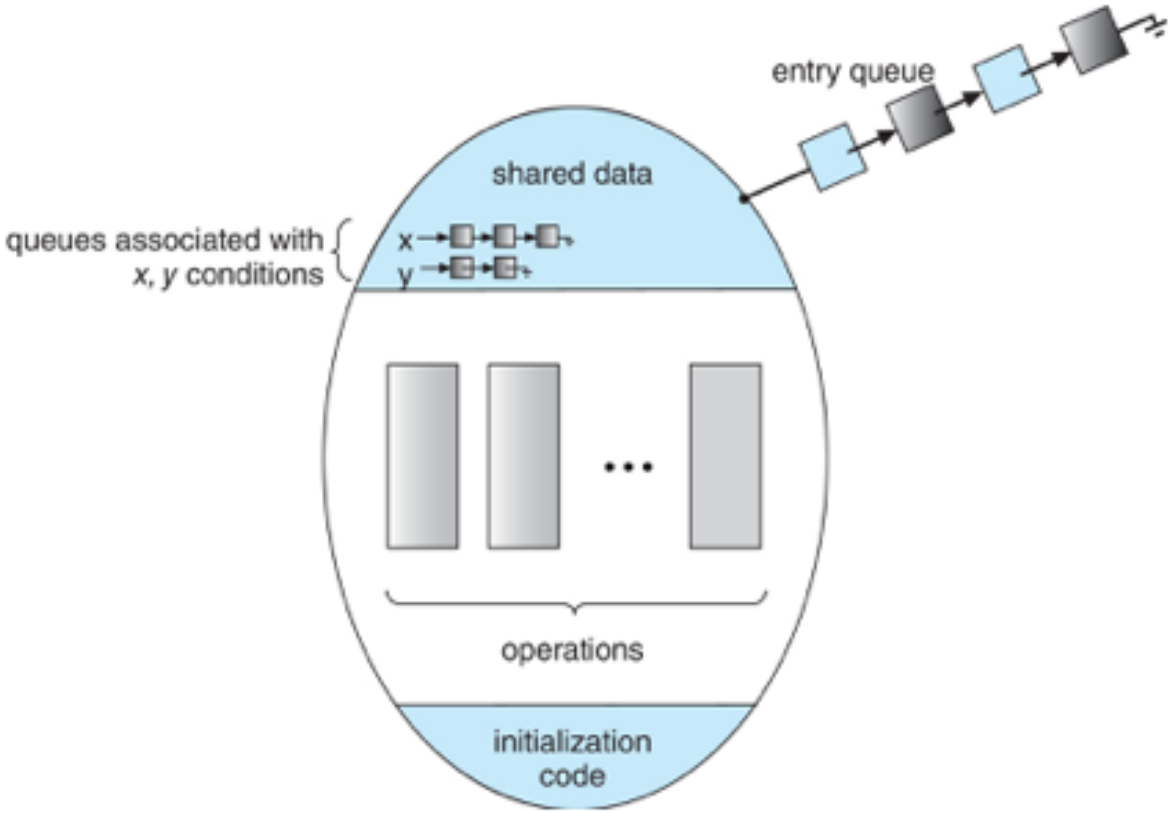
A programmer who needs to write a tailor-made synchronization scheme can define one or more variables of type `condition`:

```
condition x, y;
```

Two operations are allowed on a condition variable:

- `x.wait()` - a process that invokes the operation is suspended until `x.signal()`
- `x.signal()` - resumes one of processes (if any) that invoked `x.wait()`
  - If no `x.wait()` on the variable, then it has no effect on the variable

Monitor with Condition Variables



Monitor with `condition` variables.

`https://www.youtube.com/watch?v=15Q8PILXkQ0`

Condition Variables Choices

If process P invokes `x.signal()`, and process Q is suspended in `x.wait()`, what should happen next?

- Signal and wait - P either waits until Q leaves the monitor or it waits for another condition
- Signal and continue - Q waits until P either leaves the monitor or it waits for another condition

Both have pros and cons - language implementer can decide

Monitors implemented in Concurrent Pascal compromise; P executing signal immediately leaves the monitor, Q is resumed

Idea of the monitor is implemented in many languages including C#, Java

Monitor Implementation Using Semaphores

Signal and wait scheme

- an additional binary semaphore `next` is introduced
- `next_count` counts the number of suspended processes

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

Each function F will be replaced by

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

Mutual exclusion within a monitor is ensured

Monitor Implementation - Condition Variables

For each condition variable x, we have:

```
semaphore x_sem; // (initially = 0)
int x_count = 0;
```

`x.wait()`

```
x_count++;
if (next_count > 0)
        signal(next);
else
        signal(mutex);
wait(x_sem);
x_count--;
```

`x.signal()`

```
if (x_count > 0) {
        next_count++;
        signal(x_sem);
        wait(next);
        next_count--;
}
```

Resuming Processes within a Monitor

If several processes queued on condition variable x, and `x.signal()` is executed, which process should be resumed?

First come, first served (FCFS) frequently not adequate

Conditional-wait construct of the form `x.wait(c)`

- Where `c` is priority number
- Process with lowest number (highest priority) is scheduled next

Single Resource Allocation

Allocate a single resource among competing processes using priority numbers that specify the maximum time a process plans to use the resource

```
R.acquire(t);
        ...
    access the resource;
        ...

R.release;
```

Where R is an instance of type `ResourceAllocator`

A Monitor to Allocate Single Resource

```
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;
        }
}
```

Liveness

Processes may have to wait indefinitely while trying to acquire a synchronization tool such as a mutex lock or semaphore.

Waiting indefinitely 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 a liveness failure.

Liveness

Deadlock - two or more processes are waiting indefinitely for an event that can be caused by only one of the waiting processes

Let S and Q be two semaphores initialized to 1

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

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

- If P_0 executes `wait(S)` and P_1 `wait(Q)`. When P_0 executes `wait(Q)`, it must wait until P_1 executes `signal(Q)`
- P_1 is waiting until P_0 execute `signal(S)`.

Since these `signal()` operations will never be executed, P_0 and P_1 are deadlocked.

Liveness - Other forms of deadlock

Starvation - indefinite blocking

- A process may never be removed from the semaphore queue in which it is suspended

Priority Inversion - Scheduling problem when lower-priority process holds a lock needed by higher-priority process

- Solved via priority-inheritance protocol

Priority Inheritance Protocol

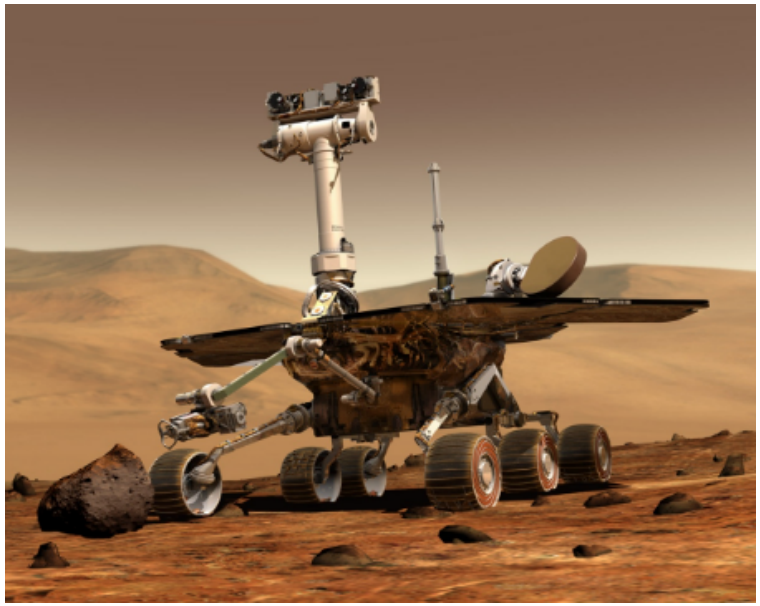
Consider P_1, P_2, and P_3. P_1 has the highest priority, P_2 the next highest, and P_3 the lowest.
- Assume P_3 is assigned a resource R that P_1 wants.
- Thus, P_1 must wait for P_3 to finish using the resource.
- However, P_2 becomes runnable and preempts P_3.

What has happened is that P_2, a process with a lower priority than P_1, has indirectly prevented P_1 from gaining access to the resource.

To prevent this from occurring, a priority inheritance protocol is used - allows the priority of the highest thread (P_1) waiting to access a shared resource to be assigned to the thread currently using the resource (P_3).

Priority Inversion Example

Mars Pathfinder



https://www.rapitasystems.com/blog/what-really-happened-to-the-software-on-the-mars-pathfinder-spacecraft

Thank you !

Operating Systems are among the most complex pieces of software ever developed !

Mutual-exclusion Implementation

```
boolean waiting[0] = false;
boolean waiting[1] = false;
int lock = 0;
```

P0

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

    /* critical section */

   j = (0 + 1) % 2;
   while ((j != 0) && !waiting[j])
      j = (j + 1) % 2;
   if (j == 0)
      lock = 0;
   else
      waiting[j] = false;
   /* remainder section */
}
```

P1

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

    /* critical section */

   j = (1 + 1) % 2;
   while ((j != 1) && !waiting[j])
      j = (j + 1) % 2;
   if (j == 1)
      lock = 0;
   else
      waiting[j] = false;
   /* remainder section */
```