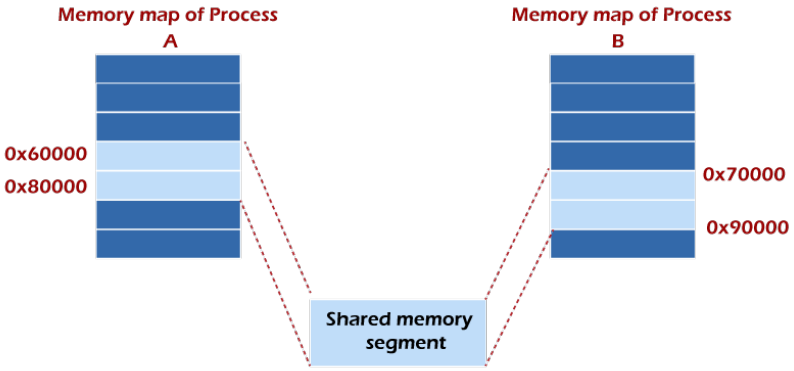
Mutual Exclusion for Cooperating Processes

Motivation

Processes cooperate through shared memory. When multiple processes share memory data, things can easily go wrong. This lecture discusses the following:

- Challenges when maintaining consistency in shared data:
    - Atomicity, Race Condition, mutual exclusion
- Solutions:
    - Locks for Mutual Exclusion
    - Peterson's Algorithm for the Critical-Section Problem
    - Hardware Locks: TestAndSet
    - Semaphores

Interprocess Communication



From https://www.javatpoint.com/ipc-through-shared-memory.

Examples of Challenges to Shared Memory Consistency

- In a single process, your program dictates the order of execution.
- When there are two or more processes, the order of execution is not guaranteed!

Assume two processes share a variable `count`. If Process A and Process B make a call em at about the same time, can you guarantee the outcome?

- Process A: `count++;`;
  Process B: `count--;`
- Process A: `count = 2*count;`
  Process B: `count--;`
- Process A: `x = count; [...]; count = x`;
  Process B: `count--;`

Examples of Challenges to Shared Memory Consistency

Process A: `count++;;`
Process B: `count--;`

The operations are such that the order does not matter. There is no data consistency problem. (As long as $++$ and – are atomic (they don't interleave), see later.)

Process A: `count = 2*count;`
Process B: `count--;`

The order matters! Take for example `count = 4`. Then `count =` 7 if A completes before B, and `count = 6` if B goes before A. (Again assuming the operations are atomic.)

Examples of Challenges to Shared Memory Consistency

Process A: `x = count; count = x + 1;`
Process B: `count--;`

Take a value, for instance `count = 4`. Then the following could happen:

- B completes (`count = 3`), then A completes (`count = 4`)
- A completes (`count = 5`), then B completes (`count = 4`)
- A executes `x = count` (then `count = x = 4`), then B **interleaves** (`count = 3` and `x = 4`), and then A executes `count = x + 1`. Then `count = 5`...!

The execution of Process A is **not atomic**. Actually, even simple operations such as `count++` are actually not atomic once translated into assembly code executed by the CPU! See Section 6.1 in Silberschatz' book.

Critical Section Problem

When the order in which shared data is accessed matters, we say that there is a **Race Condition**. The Operating Sytems kernel is full with potential Race Conditions, since it serves interrupts, preempts processes, and serves many processes at one time.

One main source of race conditions is the interleaving of processes manipulating the data. We want to avoid that multiple processes can access the shared data at the same time.

Approach: each process will have a **Critical Section**. No two processes are allowed to execute in their critical section at the same time.

Critical Section Problem: design a protocol that the processes can use to implement Critical Sections.

Critical-Section Problem

A protocol that solves the Critical Section Problem should guarantee the following:

- **Mutual Exclusion** - If process P_i is in its critical section then no other processes can be in their critical section.
- **Progress** - No indefinite blocking.
- **Bounded Waiting** - There exist a bounded time within which any process will be allowed to enter their critical section.

We assume that each process executes at a nonzero speed. There is no assumption concerning relative speed of the N processes.

Critical-Section Problem

Why is the Critical Section problem complicated? Aren't there some simple solutions:

- **Can't we avoid Preemption in the Operating System.** If the Operating System does not pre-empt processes, then that solves the Critical Section Problem (assuming a single CPU). But that would mean less ability to fairly share resources between processes, and no quick I/O handling for interactive applications.
- **Can't we have a 'token' in Shared Memory that a Process reserves?** Yes, that is exactly what we will aim for in this lecture (we will call the 'token' a 'lock')! But it is more subtle than one would think.

Locks

All solutions to the Critical Section problem require a **lock.**

A process **acquires a lock** before entering, and **releases the lock** when it exits the Critical Section.

The general pattern for using locks is:

```
do {
  [acquire lock]
    [critical section]
  [release lock]
    [remainder section]
} while (TRUE);
```

Locks

Observations about locks:

- As long as only one party has the lock at any one time, **Mutual Exclusion is guaranteed**.
- At the same time, there is always a **performance penalty**: locks slow down completion of processes.
- Lock-based protocols or algorithms are often quite subtle.
- More complex if there are less assumptions (e.g., assuming atomicity of executing certain instructions simplifies the protocol).
- Yet more complex if time and delays are considered (realm of synchronous and asynchronous protocols)
- Yet more complex if faults or attacks are considered

This is the area of **Distributed Systems** or **Distributed Algorithms**, including **Fault Tolerance** and **Byzantine Fault Tolerance**.

A Naïve Locking Algorithm

- Let's try a simple algorithm to grab the lock, for two processes.
- In these algorithms, we always write in terms of process P_i and P_j (and others if the algorithm is designed for more processes). Doing so allows us to write the multi-partys algorithms as succinct as possible. You may find it useful to change i and j into A and B when you study the algorithms.
- What's wrong with the following algorithm?

```
is_in[i] = FALSE; // I'm not in the section
do {
  while (is_in[j]); // Wait whilst j is in the critical section
  is_in[i] = TRUE; // I'm in the critical section now.
    [critical section]
  is_in[i] = FALSE; // I've finished in the critical section now.
    [remainder section]
} while (TRUE);
```

A Naïve Locking Algorithm

Directly after a process escapes the while-loop *both* `is_in[i]` and `is_in[j]` are FALSE. At that point, both processes may go into the Critical Section (the while-loop may evaluate FALSE for both).

Therefore we introduce Peterson's Algorithm, which uses the shared variable `turn`–you give the turn to the other party when you decide you want to enter. Therefore, the while loops in Peterson's algorithm never evaluate to FALSE for *both* processes at the same time. That is, there is no chance for the two processes to enter the Critical Section together.

Peterson's Algorithm for Critical Section Problem

- Two process solution.
- Assume that the CPU's register LOAD and STORE instructions are atomic (*not realistic* on modern CPUs, educational example).
- The two processes share two variables:
  - `int turn;`
  - `Boolean wants_in[2];`
- The variable `turn` indicates whose turn it is to enter the critical section.
- The `wants_in` array is used to indicate if a process is ready to enter the critical section. `wants_in[i] = true` implies that process P_i is ready.
- `turn` and `wants_in` together implement a locking mechanism (to assure mutual exclusion), and also assure progress.

Peterson's Algorithm

```
do {
  wants_in[i] = TRUE; // I want access...
  turn = j; // but, please, you go first
  while (wants_in[j] && turn == j); // if you are waiting and it is
                                    // your turn, I will wait.
    [critical section]
  wants_in[i] = FALSE; // I no longer want access.
    [remainder section]
} while (TRUE);
```

When both processes are interested, they achieve fairness through the `turn` variable, which causes their access to alternate.

Peterson's Algorithm

Peterson's Algorithm is interesting, but:

- We made assumptions about CPU instruction atomicity.
- Only support two competing processes. (BTW, more that two can be done, see exercise Chapter 6 Silberschatz' book.)
- A process may wait unnecessarily if a context switch occurs as follows: after another process leaves the remainder section but before it politely offers the turn to our waiting process.
- It needs to be implemented in software, and different processes may run different software that need to be compatible.
- There could be hardware or software errors, even malicious attacks, that cannot be handled gracefully – one would require a fault tolerant or Byzantine protocol.

Synchronisation Hardware

Synchronisation Hardware

- Many systems provide hardware support for critical section
- Uniprocessors - could disable interrupts
  - Currently running code would execute without preemption
  - Generally too inefficient on multiprocessor systems
  - Delay in one processor telling others to disable their interrupts
- Modern machines provide the special atomic hardware instructions (Atomic = non-interruptible) `TestAndSet` or `Swap` which achieve the same goal:
  - `TestAndSet`: Test memory address (*i.e.* read it) and set it in one instruction
  - `Swap`: Swap contents of two memory addresses in one instruction
- We can use these to implement simple locks to realise mutual exclusion

TestAndSet Instruction

- High-level definition of the atomic CPU instruction:

```
boolean TestAndSet (boolean *target) {
  boolean original = *target; // Store the original value
  *target = TRUE; // Set the variable to TRUE
  return original: // Return the original value
}
```

- In a nutshell: this single CPU instruction sets a variable to TRUE and returns the original value.
- If it returns FALSE, we know that only our thread has changed the value from FALSE to TRUE; if it returns TRUE, we know we haven't changed the value.
- In other words, with `TestAndSet` we can express interest in the lock and capture the lock. If it returns FALSE we have the lock. If it returns TRUE we know someone else is using the lock.

Solution using TestAndSet

- Shared boolean variable `lock`, initialized to false.
- Solution:

```
do {
  while (TestAndSet(&lock)) ; // wait until we successfully
                              // change lock from false to true
    [critical section]
  lock = FALSE; // Release lock
    [remainder section]
} while (TRUE);
```

- Note, though, that this achieves mutual exclusion but not *bounded waiting* - one process could potentially wait for ever due to the unpredictability of context switching.

Bounded-waiting Mutual Exclusion with TestAndSet()

- All data structures are initialised to FALSE.
- `wants_in[]` is an array of waiting flags, one for each process.
- `lock` is a boolean variable used to lock the critical section.

The protocol on the next page achieves Mutual Exclusion as well as Bounded Waiting. We won't prove this, nor do we go into too much detail in the lecture.

Bounded-waiting Mutual Exclusion with TestAndSet()

```
boolean wants_in[], key = FALSE, lock = FALSE; //all false to begin with

do {
  wants_in[i] = TRUE; // I (process i) am waiting to get in.
  key = TRUE; // Assume another has the key.
  while (wants_in[i] && key) { // Wait until I get the lock
                                // (key will become false)
    key = TestAndSet(&lock); // Try to get the lock
  }
  wants_in[i] = FALSE; // I am no longer waiting: I'm in now

  [*** critical section ***]

  // Next 2 lines: get ID of next waiting process, if any.
  j = (i + 1) % NO_PROCESSES; // Set j to my right-hand process.
  while ((j != i) && !wants_in[j]) { j = (j + 1) % NO_PROCESSES };

  if (j == i) { // If no other process is waiting...
    lock = FALSE; // Release the lock
  } else { // else ....
    wants_in[j] = FALSE; // Allow j to slip in through the 'back door'
  }
  [*** remainder section ***]
} while (TRUE);
```

Spinlock

- Consider the simple mutual exclusion mechanism we just saw:

```
do {
  while (TestAndSet(&lock)) ; // wait until we successfully
                             // change lock from false to true
    [critical section]
  lock = FALSE; // Release lock
    [remainder section]
} while (TRUE);
```

- The while loop spins constantly (using up CPU cycles) until it manages to enter the critical section.
- This is called a **Spinlock**.
- Such waste of CPU cycles is in general not acceptable, particularly for user processes where the critical section may be occupied for some time.

Sleep and Wakeup

- Rather than having a process spin around and around, checking if it can proceed into the critical section, suppose we implement some mechanism whereby it sends itself to sleep and then is awoken only when there is a chance it can proceed
- Functions such as `sleep()` and `wakeup()` are often available via a threading library or as kernel service calls.

Mutual Exclusion with sleep()

- If constant spinning is a problem, suppose a process puts itself to sleep in the body of the spin.
- Then whoever won the competition (and therefore entered the critical section) will wake all processes before releasing the lock - so they can all try again.

```
do {
  while (TestAndSet(&lock)) { // If we can't get the lock, sleep.
    sleep();
  }
    [critical section]
  wake_up(all); // Wake all sleeping threads.
  lock = FALSE; // Release lock
    [remainder section]
} while (TRUE);
```

Still some remaining Data Consistency Challenges

- Somehow, we need to make sure that when a process decides that it will go to sleep (if it failed to get the lock) it actually goes to sleep without interruption, so the wake-up signal is not missed by the not-yet-sleeping process
- In other words, we need to make the check-if-need-to-sleep and go-to-sleep operations happen atomically with respect to other threads in the process.
- It turns out that is essentially not possible to do within the process code. Instead, Operating System must force the release of locks prior to serving an interrupt or doing a context switch. This piece of kernel code is called a **semaphore**, after the Dijkstra's solution we discuss now.

Semaphores

Semaphores

- Synchronisation tool, proposed by E. W. Dijkstra (1965), that
  - Simplifies synchronisation for the programmer
  - Does not require (much) busy waiting: We don't busy-wait for the critical section, usually only to achieve atomicity to check if we need to sleep or not, *etc.*
  - Can guarantee *bounded waiting time* and *progress*.
- Consists of:
  - A semaphore type `S`, that records a list of waiting processes and an integer
  - Two standard atomic ($\leftarrow$ very important) operations by which to modify `S`: `wait()` and `signal()`
  - Originally called `P()` and `V()` based on the equivalent Dutch words

Semaphores

- Works like this:
  - The semaphore is initialised with a count value of the maximum number of processes allowed in the critical section at the same time.
  - When a process calls `wait()`, if count is zero, it adds itself to the list of sleepers and blocks, else it decrements count and enters the critical section
  - When a process exits the critical section it calls `signal()`, which increments count and issues a wakeup call to the process at the head of the list, if there is such a process
    - It is the use of ordered wake-ups (*e.g.* FIFO) that makes semaphores support bounded (*i.e.* fair) waiting.

Semaphore as General Synchronisation Tool

- We can describe a particular semaphore as:
    - A Counting semaphore - integer value can range over an unrestricted domain (*e.g.* allow at most N threads to read a database, *etc.*)
    - A Binary semaphore - integer value can range only between 0 and 1
        - Also known as mutex locks, since ensure mutual exclusion.
        - Basically, it is a counting semaphore initialised to 1

Critical Section Solution with Semaphore

```
Semaphore mutex; // Initialized to 1
do {
  wait(mutex); // Unlike the pure spin-lock, we are blocking here.
    [critical section]
  signal(mutex);
    [remainder section]
} while (TRUE);
```

Semaphore Implementation: State and Wait

We can implement a semaphore within the kernel as follows (note that the functions must be atomic, which our kernel must ensure):

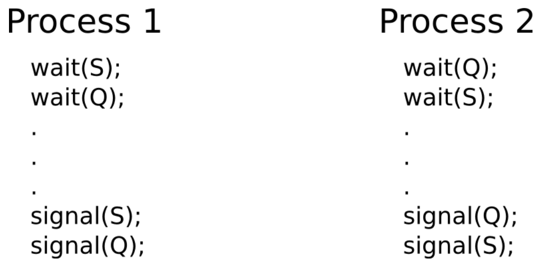
```
typedef struct {
  int count;
  process_list; // Hold a list of waiting processes/threads
} Semaphore;

void wait(Semaphore *S) {
  S->count--;
  if (S->count < 0) {
    add process to S->process_list;
    sleep();
  }
}
```

Semaphore Implementation: Signal

```
void signal(Semaphore *S) {
  S->count++;
  if (S->count <= 0) { // If at least one waiting process, let him in.
    remove next process, P, from S->process_list
    wakeup(P);
  }
}
```

Deadlock and Priority Inversion

- **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



```
Process 1                Process 2

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

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

Classical Problems of Synchronisation and Semaphore Solutions

- Readers and Writers Problem
- Semaphores in Linux

Readers-Writers Problem

- A data set is shared among a number of concurrent processes
    - Readers - only read the data set; they do not perform any updates
    - Writers - can both read and write
- Problem:
    - Allow multiple readers to read at the same time if there is no writer in there.
    - Only one writer can access the shared data at the same time

Readers-Writers Problem

- The solution set-up:
  - Semaphore `mutex` initialized to 1
  - Semaphore `wrt` initialized to 1
  - Integer `readcount` initialized to 0

Writer

```
while(True) {
  wait(wrt); // Wait for write lock.
  // perform writing
  signal(wrt); // Release write lock.
}
```

Reader

```
while(True) {
  wait(mutex) // Wait for mutex to change read_count.
  read_count++;
  if (read_count == 1) // If we are first reader, lock out writers.
    wait(wrt)
  signal(mutex) // Release mutex so other readers can enter.

  // perform reading

  wait(mutex) // Decrement read_count as we leave.
  read_count--;
  if (read_count == 0)
    signal(wrt) // If we are the last reader to
  signal(mutex) // leave, release write lock
}
```

Linux kernel representation of semaphores

- `wait(mutex)` is `mutex_lock` in the kernel
  `signal(mutex)` is `mutex_unlock` in the kernel
  This is a binary semaphore
- `down_read` and `down_write` in the kernel are read-write binary semaphores
- Have also counting semaphores in the linux kernel

Summary

- Need to ensure that certain parts of the code (critical sections) are executed in a specified order
- Software solutions exist, but complexity very high
- Need atomic test-and-set operation, supported by hardware
- Can build synchronisation primitives (semaphores, locks) on top of test-and-set operation
- Linux kernel implements these primitives