Background

- Concurrent access to shared data may result in data inconsistency
- Maintaining data consistency requires mechanisms to ensure the orderly execution of cooperating processes

Classic Problem: Finite buffer shared by producer and consumer

- Producer produces a stream of items and puts them into buffer
- Consumer takes out stream of items

Have to deal with producers waiting for consumers when buffer is full, and consumers waiting for producers when buffer is empty.

Producer Process Code

```
// Produce items until the cows come home.
while (TRUE) {
  // Produce an item.
  Item next_produced = [some new item to add to buffer];

  // Wait one place behind the next item to be consumed - so we don't
  // write to items that have yet to be consumed.
  while (((in + 1) % BUFFER_SIZE) == out); // <- Spin on condition

  // Store the new item and increment the 'in' index.
  buffer[in] = next_produced;
  in = (in + 1) % BUFFER_SIZE;
}
```

Consumer Process Code

```
while (true) {
  // Wait until there is something to consume.
  while (in == out); // <- Spin on condition
  // Get the next item from the buffer
  Item next_item = buffer[out];
  [process next_item]

  // Increment the out index.
  out = (out + 1) % BUFFER_SIZE;
}
```

Improving the Bounded Buffer

- Suppose that we wanted to provide a solution to the consumer-producer problem that fills all the buffer slots, rather than `BUFFER_SIZE-1`.
  - We can do so by having an integer `count` that keeps track of the number of full buffers. Initially, `count` is set to `0`. It is incremented by the producer after it produces a new item and is decremented by the consumer after it consumes an item.

Producer

```
while (true) {
  /* produce an item and put in nextProduced */
  while (count == BUFFER_SIZE); // wait if buffer full
  buffer [in] = nextProduced; // store new item
  in = (in + 1) % BUFFER_SIZE; // increment IN pointer.
  count++; // Increment counter
}
```

Consumer

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

Race Condition

There is something wrong with this code!

- `count++` could be compiled as a sequence of CPU instructions:
  - `register1 = count`
  - `register1 = register1 + 1`
  - `count = register1`
- `count--` could be compiled likewise:
  - `register2 = count`
  - `register2 = register2 - 1`
  - `count = register2`

Race Condition

- Consider that the producer and consumer execute the `count++` and `count--` around the same time, such that the CPU instructions are interleved as follows (with `count = 5` initially):
  - Prod.: `register1 = count` {`register1` $\rightarrow$ `5`}
  - Prod.: `register1 = register1 + 1` {`register1` $\rightarrow$ `6`}
  - Cons.: `register2 = count` {`register2` $\rightarrow$ `5`}
  - Cons.: `register2 = register2 - 1` {`register2` $\rightarrow$ `4`}
  - Prod.: `count = register1` {`count` $\rightarrow$ `6`}
  - Cons.: `count = register2` {`count` $\rightarrow$ `4`}
- With an increment and a decrement, in whatever order, we would expect no change to the original value (5), but here we have ended up with 4?!?!.

Solution Criteria to Critical-Section Problem

- So we must find a solution to protect against concurrent modification of data in what is termed the *critical section* with the following criteria:
    - **Mutual Exclusion** - If process P_i is executing in its critical section (*i.e.* where shared variables could be altered inconsistently), then no other processes can be executing in their critical sections.
    - **Progress** - no process outside of the critical section (*i.e.* in the *remainder section*) should block a process waiting to enter.
    - **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 (*i.e.* it must be fair, so one poor process is not always waiting behind the others).
- Assume that each process executes at a nonzero speed.
- No assumption concerning relative speed of the N processes.

Peterson's Solution

- Two process solution.
- Assume that the CPU's register LOAD and STORE instructions are atomic; that is, cannot be interrupted.
    - With the extreme optimisation of modern CPU architectures this assumption no longer holds, but this algorithm is of historical relevance.
- 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!

But firstly: A Naïve Algorithm for Process P_i

```
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);
```

What's wrong with this?

Peterson's Algorithm for Process P_i

```
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 for Process P_i

- Interesting, but:
  - Aside from the question of CPU instruction atomicity, how can we support more than two competing processes?
  - Also, if we were being pedantic, we could argue that a process may be left to wait unnecessarily if a context switch occurs only after another process leaves the remainder section but before it politely offers the turn to our first waiting process.

Synchronisation Hardware

- Many systems provide hardware support for critical section code.
- 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

Solution to Critical-section Problem Using Locks

- The general pattern for using locks is:

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

TestAndndSet 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.
- This is useful because, 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.

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.

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);
```

But There's a Bit of a Problem...

- Consider the simple mutual exclusion mechanism we saw in the last lecture:

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

- This guarantees mutual exclusion but with a high cost: that while loop spins constantly (using up CPU cycles) until it manages to enter the critical section.
- This is a huge waste of CPU cycles and is 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.
- Let's explore this idea

Mutual Exclusion with sleep(): A first Attempt

- 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);
```

Mutual Exclusion with sleep(): A first Attempt

- Is this a satisfactory solution?
- Does it achieve mutual exclusion?
- Does it allow progress (*i.e.* stop non-interested processes from blocking those that want to get it)?
- Does it have the property of bounded waiting?

Towards Solving The Missing Wakeup Problem

- 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.
- Perhaps we could do this by making use of another lock variable, say `deciding_to_sleep`.
- Since we now have two locks, for clarity, let's rename `lock` to `mutex_lock`.

A Possible Solution?

```
while (True) {
  // Spinning entry loop.
  while (True) {
    // Get this lock, so a wake signal cannot be raised before we actually sleep.
    while(TestAndSet(&deciding_to_sleep));

    // Now decide whether or not to sleep on the lock.
    if (TestAndSet(&mutex_lock)) {
      sleep();
    } else {
      // We are going in to critical section.
      deciding_to_sleep = False;
      break;
    }
    deciding_to_sleep = False; // Release the sleep mutex for next attempt.
  }
    [critical section]
  while(TestAndSet(&deciding_to_sleep)); // Don't issue 'wake' if a thread is
                                         // deciding whether or not to sleep.
  mutex_lock = False; // Open up the lock for the next guy.
  wake_up(all); // Wake all sleeping threads so they may compete for entry.
  deciding_to_sleep = False;
    [remainder section]
}
```

- Have we, perhaps, overlooked something here?

Sleeping with the Lock

- We've encountered an ugly problem — in fact, there are several problems with the previous example that are all related.
    - As we said before, we need to decide to sleep and then sleep in an atomic action (with respect to other threads/processes)
    - But in the previous example, when a thread goes to sleep it keeps hold of the `deciding_to_sleep` lock which sooner or later will result in deadlock!
    - And if we release the `deciding_to_sleep` lock immediately before sleeping, then we have not solved the original problem.
- What to do, what to do...

Sleeping with the Lock

- The solution to this problem implemented in modern operating systems, such as Linux, is to release the lock *during* the kernel service call `sleep()`, such that it is released prior to the context switch, with a guarantee there will be no interruption.
- The lock is then reacquired upon wakeup, prior to the return from the `sleep()` call.
- Since we have seen how this can all get very complicated when we introduce the idea of sleeping (to avoid wasteful spinning), kernels often implement a sleeping lock, called a *semaphore*, which hides the gore from us.
- See `http://tomoyo.sourceforge.jp/cgi-bin/lxr/source/kernel/semaphore.c#L75`

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: State and Wait

- Note that, here, we decrement the wait counter before blocking (unlike the previous description).
- This does not alter functionality but has the useful side-effect that the negative count value reveals how many processes are currently blocked on the semaphore.

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);
  }
}
```

But we have to be Careful: 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
  - Good account of this from NASA programme: `http://research.microsoft.com/en-us/um/people/mbj/mars_pathfinder/authoritative_account.html`

Classical Problems of Synchronisation and Semaphore Solutions

- Bounded-Buffer Problem
- Readers and Writers Problem
- Boring as it may be to visit (and teach about) the same problems over and over again, a solid understanding of these seemingly-insignificant problems will directly help you to spot and solve many related real-life synchronisation issues.

Our Old Friend: The Bounded-Buffer Problem

- The solution set-up:
  - A buffer to hold N items, each can hold one item
  - Semaphore `mutex` initialized to the value 1
  - Semaphore `full_slots` initialized to the value 0
  - Semaphore `empty_slots` initialized to the value N.

Producer

```
while(True) {
  wait(empty_slots); // Wait for, then claim, an empty slot.
  wait(mutex);
  // add item to buffer
  signal(mutex);
  signal(full_slots); // Signal that there is one more full slot.
}
```

Consumer

```
while(True) {
  wait(full_slots); // Wait for, then claim, a full slot
  wait(mutex);
  // consumer item from buffer
  signal(mutex);
  signal(empty_slots); // Signal that now there is one more empty slot.
}
```

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
}
```