The Problem of Concurrency

- The problem is simply sharing resources.
  - Several threads/processes running at the same time.
  - Using the same resources - accessing the same data structures/objects/devices
  - Some resources can only be safely used by one thread at a time.
    - e.g. readers accessing shared data while a writer is changing it
    - or writers changing a resource simultaneously
- Race condition

  Any situation where the order of execution of threads can cause different results.

Our programs must control the *non-deterministic* nature of thread scheduling.

Example (contention.c)

```
#include <stdio.h>
#include <pthread.h>

volatile int counter = 0;
_Atomic int acounter = 0;

void increment_counter(void *id) {
     for (int i = 0; i < 1000000; i++){
          counter++;
          acounter++;
     }
     printf("thread: %ld counter: %d\n", (long)id, counter);
     printf("thread: %ld acounter: %d\n", (long)id, acounter);
}

int main() {
        int const num_threads = 10;
        pthread_t thread_refs[num_threads];

        for (long i = 0; i < num_threads; i++)
                pthread_create(&thread_refs[i], NULL,
                                 (void *)increment_counter, (void *)i);
        for (int i = 0; i < num_threads; i++)
                pthread_join(thread_refs[i], NULL);
}
```

Critical sections

- An area of code in which we only want one thread to be active at a time.
- Providing this is known as *mutual exclusion*.
- We need:
  1. a way of locking threads out of critical sections
  2. to guarantee threads are not kept waiting forever - starvation
- Starvation can be caused in different ways
  - deadlock
  - indefinite postponement - priority too low or just unlucky

Software solutions

- We want something like this:

```
lock
  critical section
unlock
```

- We have a boolean variable `locked` which is true if the critical section is being used by a thread.
  Initially `locked` is false.
- Attempt 1

  Our first lock procedure:

```
while locked
end
locked = true
```

  And the unlock:

```
locked = false
```

- Locks like this are known as *spin-locks* or *busy waits*.
- What is wrong with this lock? At least 3 different things

Another attempt - Peterson’s Solution

- This only works on shared memory multiprocessors if instruction reordering can be turned off. Otherwise we need hardware help with memory barriers.
- Two writes don't get interleaved at some minimum write size, the hardware allows only one processor access at a time.
- Java note: all primitives except double and long are guaranteed to be written atomically
- Software solutions to locking critical regions require this level of hardware assistance.

- A two thread solution.

```
flag = [false, false]
# both false initially
turn = 0
```

- lock: performed by thread i; j is the other thread

```
    flag[i] = true
    turn = j
    while (flag[j] && turn == j)
    end
```

- unlock: performed by thread i

```
    flag[i] = false
```

https://en.wikipedia.org/wiki/Peterson%27s_algorithm

Bakery algorithm and hardware help

- The previous method works but does not solve the general case.
- The bakery algorithm:
  - Each thread is given a number indicating when it requests the lock.
  - These are not unique so some other method of ordering e.g. pid is necessary as well.

Interrupt priority level

- We could just raise the interrupt priority level to stop any other process (which might affect the area) from running while the lock is being tested.
- Disadvantages
  - heavy-handed - not all processes at the current interrupt priority level need to be stopped
  - doesn't work efficiently on multiprocessors
  - a message requesting the IPL change must be sent to all processors, in some circumstances all other processors must wait.

Test and Set

- Or equivalent *atomic* or indivisible instructions
- they appear uninterruptible - once started no other process can interfere until completed

```
testAndSet(lockVariable)
```

- returns the current value of the lockVariable and sets the lockVariable to true
- With this our lock can become

```
while (testAndSet(locked))
end
```

- unlock:

```
locked = false
```

- The textbook has a C definition in Figure 6.5.

Getting out of the spin

- Our lock is a spin lock or busy wait. A waiting thread keeps running trying to get the resource even though it is not available.
  It is also not fair. (See the textbook's progress and bounded waiting requirements.)

Fairness

- Without priorities:
  - Each thread shouldn't have to wait while another thread gets access to the resource more than once.
  - Each thread should get access before any other thread which requests it later.
  - Otherwise indefinite postponement is possible.
  - i.e. a queue would help.
- But with priorities:
  - Threads with higher priorities - should they get prior access to resources?
    - Makes the priority mechanism more effective.
    - Increases the chance of indefinite postponement.
    - Priority mechanism can still work when selecting next runnable thread.

Priority inversion

- When you have priorities on processes and a locking mechanism you can get priority inversion.
- Lower priority processes with a lock can force higher priority processes to wait. But because they are low priority they may not run very frequently.
- Particularly important in real-time systems.
- Solved with priority inheritance – when a higher priority process blocks waiting for a resource the process with the resource is temporarily given the priority of the blocked process. The high priority process will now only wait during the critical section.

Placing in a queue

- When a thread must wait we put it on a queue and stop it running. This solves two problems:
  1. fairness
  2. wasting processor cycles

- Other advantages:
  - possibly frees pages for other processes
  - we know how many threads are waiting for this resource

- It is subtle, however. What could go wrong with the following? (the lock and unlock are on the next page)

```
def suspend
   enqueue(thisThread) # put on the queue
   reschedule # start another thread
end
```

- like yield but the current thread is now waiting rather than runnable

```
def awaken
   first = dequeue # head of the queue
   makeRunnable(first) # to run eventually
end
```

Placing in a queue (cont.)

- and our lock and unlock are:

```
def lock
  if (testAndSet(locked))
    suspend
end

def unlock
  if (!emptyQueue) # something in the queue
    awaken
  else
    locked = false
end
```

Semaphores

- Edsger Dijkstra (1965)
- A semaphore is an integer count, two indivisible (atomic) operations and an initialization.
- S a semaphore - the indivisible operations are:

```
V(S):
   S = S + 1

P(S):
   wait until S > 0
   S = S - 1
```

- The count tells how many of a certain resource are available.

Binary semaphores

- The semaphore is initialised to 1.
- To get a resource the thread calls P on the semaphore.
  To return the resource the thread calls V.

Implementing semaphores

- Rather than calling the operations P and V we will call them wait and signal.

```
signal(S):
    if anything waiting on S then
     start the first process on the S queue
    else
     S = S + 1
wait(S):
    if S < 1 then
     put this process on the S queue
    else
     S = S - 1
```

- another common alternative is:

```
signal(S):
    S = S + 1
    if S < 1 then
     start the first process on the S queue
wait(S):
    S = S - 1
    if S < 0 then
     put this process on the S queue
```

Producer/Consumer problem

```
require 'semaphore'

number_received = Semaphore.new(?)
number_deposited = Semaphore.new(?)

buffer = 0

producer = Thread.new do
      loop do
            next_result = rand
            number_received.wait()
            buffer = next_result
            number_deposited.signal()
      end
end

consumer = Thread.new do
      loop do
            number_deposited.wait()
            next_result = buffer
            number_received.signal()
            puts next_result
      end
end

consumer.join
```

- A thread producing data, a thread consuming the data.
  - We don't want to lose any data.
  - We don't want to use any data more than once.

What values for the "?"?

Before next time

- Read from the textbook

  7.1.2 The Readers-Writers Problem

  6.7 Monitors

  7.3 POSIX Synchronization