Now we look at other problems
Now we look at other problems

Consider our original counting code with a shared variable `count`. A simple solution might be to make `count` non-shared:

```plaintext
1 2
for (i = 0; i < 50; i++) {
    for (j = 50; j < 100; j++) {
        if (val[i] > 0) if (val[j] > 0)
            count1 = count1 + 1;                  count2 = count2 + 1;
    }
}

count = count1 + count2;
```
Now we look at other problems

Consider our original counting code with a shared variable count. A simple solution might be to make count non-shared:

1
for (i = 0; i < 50; i++) {
    if (val[i] > 0)
        count1 = count1 + 1;
}

2
for (j = 50; j < 100; j++) {
    if (val[j] > 0)
        count2 = count2 + 1;
}

count = count1 + count2;

There is now another, different, problem with this code!
The problem now is *when is the* $\text{count} = \text{count1} + \text{count2}$ *executed?*
The problem now is *when is the* \( \text{count} = \text{count1} + \text{count2} \) *executed*?

To be correct, it has to happen after both the loops have finished: any earlier will give a wrong answer.
The problem now is \textit{when is the} $\text{count} = \text{count1} + \text{count2}$ \textit{executed}?

To be correct, it has to happen after both the loops have finished: any earlier will give a wrong answer.

It will definitely happen after loop 1 has finished, but what about loop 2?
Concurrent Primitives

Synchronisation

The problem now is *when is the* count = count1 + count2 *executed?*

To be correct, it has to happen after both the loops have finished: any earlier will give a wrong answer.

It will definitely happen after loop 1 has finished, but what about loop 2?

We can’t rely (in a MIMD architecture) on the two loops on different cores running at the same time and finishing at the same time.
The problem now is *when is the* \( \text{count} = \text{count1} + \text{count2} \) *executed?*

To be correct, it has to happen after both the loops have finished: any earlier will give a wrong answer.

It will definitely happen after loop 1 has finished, but what about loop 2?

We can’t rely (in a MIMD architecture) on the two loops on different cores running at the same time and finishing at the same time.

Timings in the system may have the two loops running in any conceivable arrangement of before, after or overlapped.
Concurrent Primitives

Synchronisation

1 2
for (i = 0; i < 50; i++) {
  if (val[i] > 0)
    count1 = count1 + 1;
}
for (j = 50; j < 100; j++) {
  if (val[j] > 0)
    count2 = count2 + 1;
}
Concurrenty Primitives
Synchronisation

So we must explicitly write code to ensure the final sum only happens when both loops are finished.
So we must explicitly write code to ensure the final sum only happens when both loops are finished.

This is a *synchronisation* between the two threads.
Concurrency Primitives

Synchronisation

So we must explicitly write code to ensure the final sum only happens when both loops are finished

This is a *synchronisation* between the two threads

It may mean thread 1 waiting for thread 2
So we must explicitly write code to ensure the final sum only happens when both loops are finished.

This is a *synchronisation* between the two threads.

It may mean thread 1 waiting for thread 2.

Another sequentialisation!
More subtly: if this code is executed more than once (perhaps counting more than one array), thread 2 ought to wait for thread 1 before starting!
More subtly: if this code is executed more than once (perhaps counting more than one array), thread 2 ought to wait for thread 1 before starting!

It is possible that 1 is slow or paused for some reason, when 2 might do its bit and come around again on the next call to the count code, do the count on some other data, updating count2 as it goes
More subtly: if this code is executed more than once (perhaps counting more than one array), thread 2 ought to wait for thread 1 before starting!

It is possible that 1 is slow or paused for some reason, when 2 might do its bit and come around again on the next call to the count code, do the count on some other data, updating count2 as it goes

Finally 1 awakes and gets the wrong count2
More subtly: if this code is executed more than once (perhaps counting more than one array), thread 2 ought to wait for thread 1 before starting!

It is possible that 1 is slow or paused for some reason, when 2 might do its bit and come around again on the next call to the count code, do the count on some other data, updating count2 as it goes

Finally 1 awakes and gets the wrong count2

This does happen and is a source of bugs
Semaphores can be used for thread synchronisation
Semaphores can be used for thread synchronisation

Typically, we might have some thread that can only continue its work when one (or more) others have finished doing something, maybe computing some inputs for the thread to process
Semaphores can be used for thread synchronisation

Typically, we might have some thread that can only continue its work when one (or more) others have finished doing something, maybe computing some inputs for the thread to process

It can wait on a semaphore, again a simple flag, until another thread sets the flag. Then it knows it can continue
Semaphores can be used for thread synchronisation

Typically, we might have some thread that can only continue its work when one (or more) others have finished doing something, maybe computing some inputs for the thread to process

It can wait on a semaphore, again a simple flag, until another thread sets the flag. Then it knows it can continue

Note that even though both locks and semaphores are flags, they are very different things! Beware it is common for people to confuse the two
Semaphores are manipulated by two atomic operations \( P \) and \( V \) that symbolically act atomically as:

\[
P(s): \text{while } (s == 0) \{ \\
    \text{suspend();} \\
\}
\]

\[
V(s): s = 1; \\
\text{if any process waiting on } s \text{ unblock one}
\]

\[
s = 0;
\]
Concurrency Primitives
Semaphores

On finding $s = 0$ a thread will suspend itself; when awoken it will re-attempt to set the semaphore: and it will often succeed, unless a third thread comes along and gets the semaphore first.
On finding $s = 0$ a thread will suspend itself; when awoken it will re-attempt to set the semaphore: and it will often succeed, unless a third thread comes along and gets the semaphore first.

Like locks, semaphores are *not fair* on which thread will be awoken if more than one is waiting.
Concurreny Primitives

Semaphores

On finding \( s = 0 \) a thread will suspend itself; when awoken it will re-attempt to set the semaphore: and it will often succeed, unless a third thread comes along and gets the semaphore first.

Like locks, semaphores are *not fair* on which thread will be awoken if more than one is waiting.

Other names for P are: wait, up, lock, enter, open.

Other names for V are: signal, down, unlock, exit, close.
On finding $s = 0$ a thread will suspend itself; when awoken it will re-attempt to set the semaphore: and it will often succeed, unless a third thread comes along and gets the semaphore first.

Like locks, semaphores are *not fair* on which thread will be awoken if more than one is waiting.

Other names for P are: wait, up, lock, enter, open.

Other names for V are: signal, down, unlock, exit, close.

P stands for “proberen”, V for “verhogen”, which are Dutch for “test” and “increase”.
Semaphores synchronise across threads:

do something
wait(s) prepare data
read data signal(s)
carry on

Thread 1 waits until thread 2 has prepared some data before reading it
Concurrent Primitives

Semaphores

Semaphores synchronise across threads:

- prepare data
- do something
- signal(s)
- wait(s)
- carry on
- read data
- carry on

Thread 1 waits until thread 2 has prepared some data before reading it.

The signal and wait might happen in any order.
The above are called *binary* semaphores as the idea can be trivially extended into *counting* semaphores

\[
\text{P}(s): \text{while } (s == 0) \{ \text{V}(s): s = s + 1; \\
\hspace{1cm} \text{suspend}(); \hspace{1.5cm} \text{if any process waiting on } s \\
\hspace{1cm} \text{unblock one} \\
\} \\
s = s - 1;
\]
The above are called *binary* semaphores as the idea can be trivially extended into *counting* semaphores

\[ \text{P(s): while } (s == 0) \{ \text{V(s): } s = s + 1; \]
\[ \text{ } \text{suspend(); } \text{ if any process waiting on } s \]
\[ \text{ } \text{unblock one} \]
\[ \text{ } s = s - 1; \]

When initialised with the value \( n \), this will allow \( n \) threads to open the semaphore before blocking
Concurrency Primitives
Counting Semaphores

This allows region access control when there can be one than one, but fewer than some limit in the region simultaneously.
Concurrency Primitives

Counting Semaphores

This allows region access control when there can be one than one, but fewer than some limit in the region simultaneously.

For example, if there are 5 places at a dining table we can allow no more than 5 people in the room at a time.
Concurrency Primitives
Counting Semaphores

This allows region access control when there can be one than one, but fewer than some limit in the region simultaneously.

For example, if there are 5 places at a dining table we can allow no more than 5 people in the room at a time.

Or 4 if they are philosophers...
Concurrence Primitives

Semaphores

Mutual exclusion with semaphores happens to be easy:

wait(s);
<CR>
signal(s);

Wait for the semaphore; signal it’s free when you are done
Concurrent Primitives

Semaphores

Mutual exclusion with semaphores happens to be easy:

wait(s);
<CR>
signal(s);

Wait for the semaphore; signal it’s free when you are done

But it’s better to use locks for this: semaphores are more general than locks: they allow a thread to suspend itself and be awoken by another thread when some condition is true
Concurrency Primitives

Semaphores

Mutual exclusion with semaphores happens to be easy:

\[
\text{wait}(s); \\
<\text{CR}> \\
\text{signal}(s);
\]

Wait for the semaphore; signal it’s free when you are done

But it’s better to use locks for this: semaphores are more general than locks: they allow a thread to suspend itself and be awoken by another thread when some condition is true

Semaphores should be used across threads, mutexes must not
Concurrent Primitives

Semaphores

Mutual exclusion with semaphores happens to be easy:

```c
wait(s);
<CR>
signal(s);
```

Wait for the semaphore; signal it’s free when you are done

But it’s better to use locks for this: semaphores are more general than locks: they allow a thread to suspend itself and be awoken by another thread when some condition is true

Semaphores should be used *across* threads, mutexes must not

The locking is in some sense incidental: more useful is using semaphores to synchronise
POSIX semaphores:

```c
#include <semaphore.h>
sem_t sem;
int sem_init(sem_t *sem, int pshared, unsigned int value);
int sem_destroy(sem_t *sem);
int sem_wait(sem_t *sem);
int sem_post(sem_t *sem);
int sem_trywait(sem_t *sem);
```

“post” for signal
Concurrency Primitives
POSIX Semaphores

Exercise. Add a semaphore to the count1/count2 example to get thread 1 to wait for thread 2 before doing the final sum.

Exercise. Then add another semaphore to get thread 2 to wait for thread 1 before starting.