Concurrent Systems & Operating Systems.

- Operating System
  provides controlled, secure and assured access to the resources available on a computer.
  - Shares and limits access to the computer and its devices

- Operating System
  Provides a uniform set of facilities for which programs can be developed
  - Hides much of the unimportant detail from application programs
    eg. memory management, communications.
  - provides high level abstractions of low level facilities
    eg. file systems, media handling.

- Shells
  A shell is a program that allows you to give commands to a computer and get responses
  - Common GUI shell is 'finder' for mac and 'File explorer' for windows. Easy to learn but hard to automate.
  - Common UNIX shells are BASH, CSH, TCSH, ASH. These are command line text based programs that are a bit more difficult to learn but very powerful

- Shell Commands
  Generally invoke programs to do the work - eg. 'ls' command. When you write this command the shell looks for a program called 'ls' in a few places. When it finds it, the 'ls' program is executed.

- Pipes
  Most programs work with standard character-based input sources and output sinks 'standard input', 'standard output' and 'standard error'. These are all examples of pipes that can be connected to files, the terminal window, other programs and network feeds etc.

- Simple/Complex
  Most programs do simple things but very well. To do complex things you string programs together.

- Multiple Commands
  To issue a sequence of commands on one line, separate the commands with a semi-colon (";")

- Piping
Using the bar symbol ("|"), you can direct ("pipe"), the standard output of one command into the standard input of the next command.

- Automation
You can write a text file containing a sequence of commands and shell commands and constructs. This can be executed as a shell script.

- POSIX - Portable Operating System Interface
  - for variants of UNIX including LINUX
  - IEEE 1003, ISO/IEC9945

- POSIX
Considered a standard set of facilities and API's for Unix.
  - 1988 onwards
  - Doesn't have to be UNIX
    eg. 'Cygwin' for windows to give it partial POSIX compliance

- POSIX Threads
Also known as 'pthreads'
  - Correspond to Lightweight Processes (LWP) in old literature
  - pthreads live within processes
  - processes have separate memory addresses from eachother, thus inter-process communication and scheduling may be expensive
  - pthreads within a procces share memory space therefore inter-thread communication and scheduling may be cheap.

- POSIX Threads
  Portable threading library across Unix OSes
  - All POSIX-compliant Unixes implement pthreads
    eg. Linux, Mac OS X, Solaris, Free BSD, Open BSD etc.
  - Also on windows
    eg. Open Source : pthreads-Win32
  - By the Way Windows threads are different.

- Creating a pthread.

```
#include <pthreads.h>
int pthread_create(
        pthread_t * thread, //the returned thread ID
      const pthread_attr_t *attr, //starting attributes
    void *(*start_routine)(void*), //the function to
        void *arg);//parameter      //run in the thread
```

- Where ....
  - 'thread' is the ID of the thread
  - 'attr' is the input-only attributes (NULL for standard attributes)
  - 'start_routine' (can be any name) is the function that runs when the thread is started, and which must have the signature
    void * *start_routine (void* arg);
  - 'arg' is the parameter that is sent to the start routine.
  - returns a status code..
    - '0' is good
    - '-1' is bad.

- Wait for a thread to finish

```
int pthread_join(pthread_t thread, void ** value_ptr);
```

- Where ....
  - 'thread' is the ID of the thread you wish to wait on
  - 'value_ptr' is where the thread's exit status will be placed on exit (NULL if you're not interested)
  - NB a thread can only be joined only to one other thread.

- 'Hello World' - Creating Threads

```
int main(int argc, const char * argv[]) {

    pthread_t threads[NUM_THREADS];
    int rc;
    int t;

    for (t=0; t<NUM_THREADS; t++) {

        printf("Creating thread %d\n", t);
        rc= pthread_create(&threads[t], NULL,
                           PrintHello, (void *)t);

        if (rc) {

            printf("ERROR return code from pthread_create();
                                                %d\n", rc);

            exit(-1);
        }
    }
```

- 'Hello World' - waiting for threads to exit

```
//wait for threads to exit
for(t=0; t<NUM_THREADS; t++){

    pthread_join(threads[t], NULL);

}
```

where exit status would be placed but NULL because we don't care about it.

- 'Hello World' - Thread function

```
void *PrintHello (void *threadid){

    printf("\n%d: Hello World!\n", threadid);
    pthread_exit(NULL);

}
```

- Complete all of these parts to Complete the 'Hello World' program with pthreads
  - At Desktop on my mac in Terminal write.

```
cc -o main pthread.c -pthread
./main
```

Runtime Behaviour

Output gave ...

Creating thread 0
Creating thread 1

0: Hello World!
Creating thread 2

1: Hello World!
Creating thread 3

2: Hello World!
Creating thread 4

3: Hello World!
Creating thread 5

4: Hello World!

5: Hello World!

Runtime Behaviour

The Runtime Behaviour is no longer under the control of the program

- The order in which work gets done on the machine is not exactly under control of the program
- It seems to be the price paid for parallelism

But...

- What errors can it introduce?
- Can we prevent them / Protect against them or design them out?

- Massive Parallel problems
  There is almost no interaction between parallel threads calculating independent parts of the solution.

- Interaction
  No thread interaction is shown
  - Most of the issues to do with threads (really, with any kind of parallel processing) arise over unforeseen interaction sequences.

  For example.
  if two threads try to increment the same variable

- Unsafe Interaction Example

Thread 1	Thread 2	G
L = G (4)	-	4
L = L + 10 (14)	-	4
-	L = G (4)	4
G = L	L++ (5)	14
-	G = L	5

G = Global Variable L = Separate Local Variables

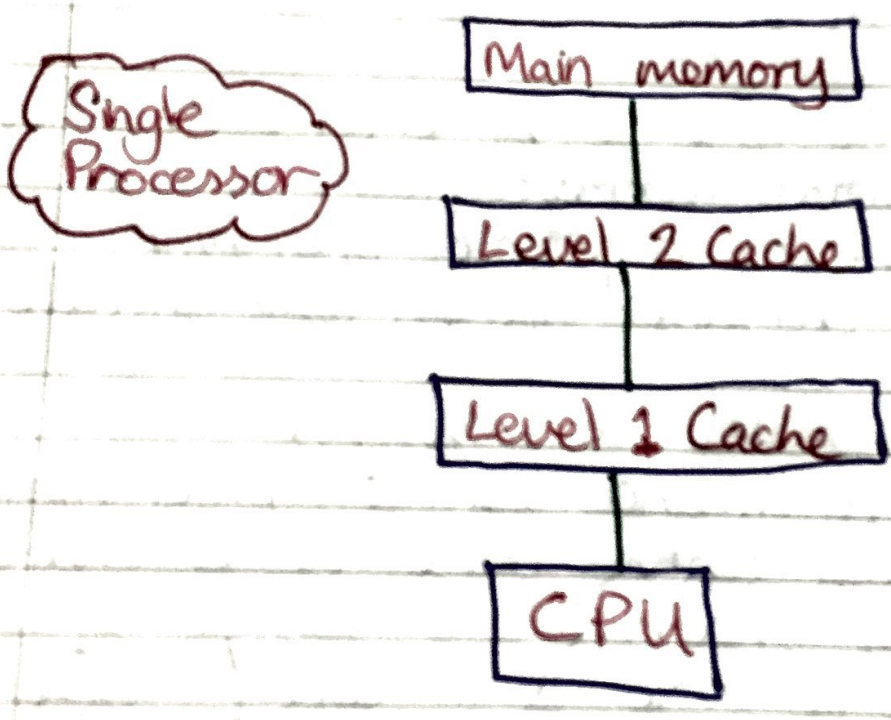
- Interactions and Dependencies
  Interactions occur because of dependencies
  - Interactions can be made safe using a variety of techniques
  - eg. ~~Semaphores~~ Semaphores, condition variables etc.
    We will look at these but they tend to be very expensive.
  - Sometimes we can redesign the code to avoid dependencies.
  - One of the biggest themes in this kind of programming is dependency minimisation.

- We need to use our knowledge of...
  - Sequential imperative programming, in C/C++ on a single processor
  - How programs could communicate.

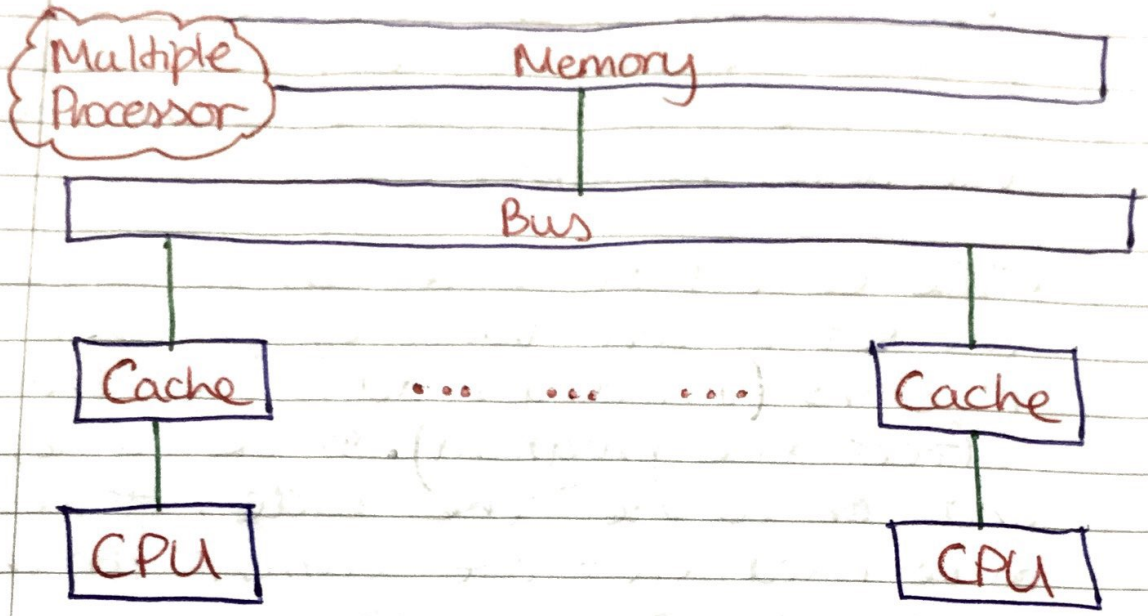
- To do Parallel Programming
  - High Performance Parallel programming
  - We will confine ourselves to shared-memory machines
  - We will use dual-core (maybe quad core) machines for practice.

- Why is it this difficult?
  It's not clear whether it really is more difficult to think about using multiple agents, e.g. multiple processors to solve a problem.
  - maybe we are conditioned into thinking about just one agent.
  - maybe it's natural.
  - maybe it's our notations and toolsets
  - Of course, maybe it really is trickier.

- Hardware



- Hardware



- Shared Memory Machine
  - Each processor has equal access to each main memory location - Uniform Memory Access (UMA)
    - Opposite: Non Uniform Memory access (NUMA)
  - Each processor may have its own cache, or may share caches with others
    - may have problems with cache issues, eg. consistency, line-sharing etc.

Sequential Program

- Sequential Program
  - A Sequential Program has one site of program execution. At any time, there is only one site in the program where execution is under way.
  - The Context of a sequential Program is the set of all variables, processor registers (including hidden ones that can affect the program), stack values, and, of course, the code. Assumed to be fixed, that are current for the single site of execution.
  - The combination of the context and the flow of program execution is often called (informally) a Thread.

Concurrent Program

- Concurrent Program
  - A Concurrent Program has more than one site of execution. That is, if you took a snapshot of a concurrent program, you could understand it by examining the state of execution in a number of different sites in the program.
  - Each site of execution has its own context eg. registers, stack values etc.
  - Thus, a concurrent program has multiple threads of execution.

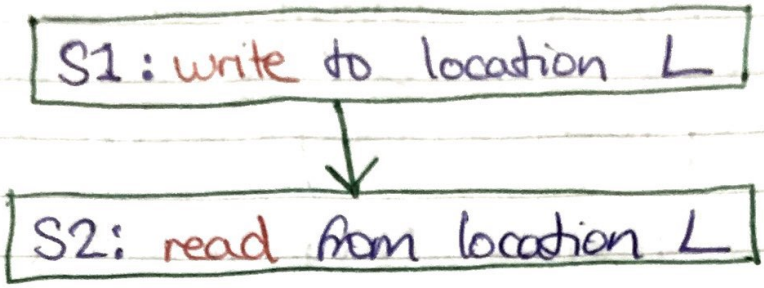
Parallel Program

- Parallel Program
  - A Parallel Program, like a concurrent program, has multiple threads of program execution.
  - The key difference between concurrent and parallel is
    - In Concurrent programs, only one execution agent is assumed to be active. Thus, while there are many execution sites, but only one will be active at any one time.
    - In Parallel programs, multiple execution agents are assumed to be active simultaneously, so many execution sites will be active at any one time.

- Communication
  - A parallel program consists of two or more separate threads of execution, that run independently except when they interact
  - To interact, they must Communicate.
  - Communication is typically implemented by
    - sharing memory
      eg. one thread writes data to memory and the other reads it.
    - passing messages
      eg. one thread sends a message, the other receives it.

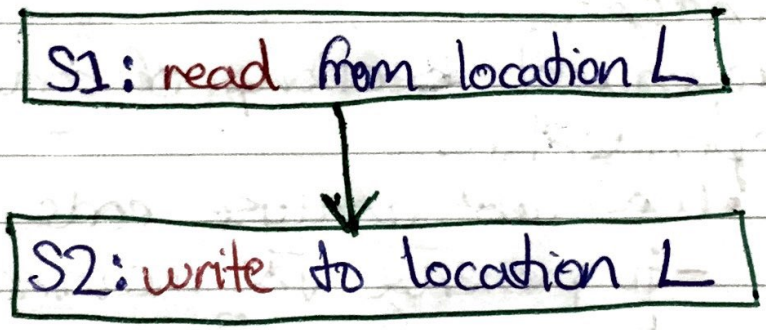
- Dependency
  Independent sections of code can run independently.
  - We can analyse code for dependencies
    - to preserve the meaning of the program.
    - to transform the program to reduce dependencies and improve parallelism.

- Flow Dependence



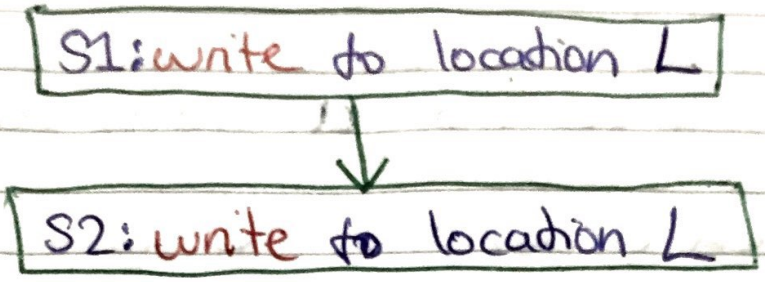
  - Flow dependence:
    - S2 is flow dependent on S1 because S2 reads a location S1 writes to
    - It must be written to S1 before it's read from S2.

- Anti-dependence



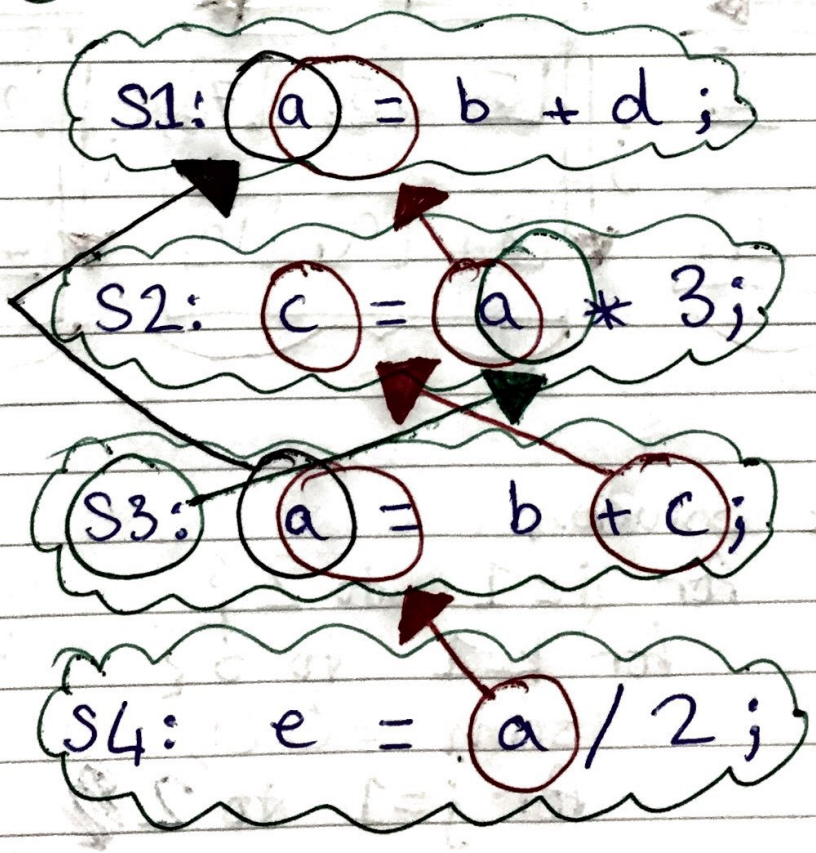
  - Anti-dependence
    - S2 is anti-dependent on S1 because S2 writes to a location S1 reads from
    - It must be read from S1 before it can be written to S2.

- Output Dependence

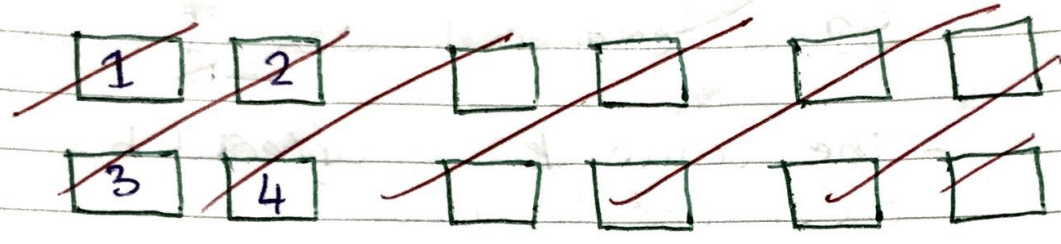


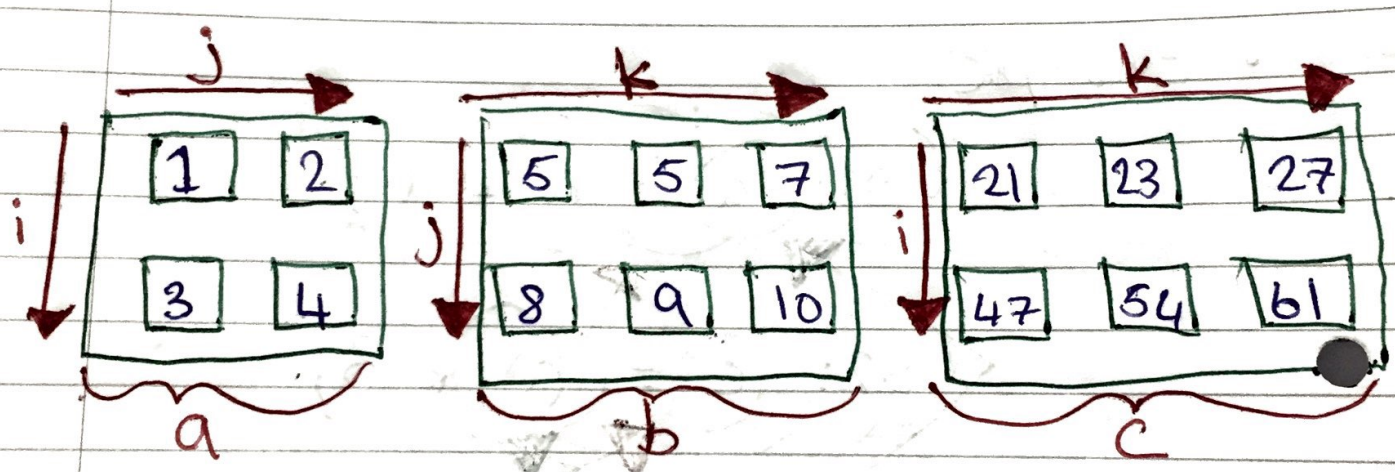
- Output dependence
  - S2 is output dependent on S1 because S2 writes to a location S1 writes to
  - The value L is affected by the order of S1 and S2

- Example



- Removing Unneeded Dependencies in Loops
  - Example, we know, intuitively, we can parallelise this a great deal:
  - each element in 'c' is independant





- Sample Solution

```
For i=1 to 2{
  for k=1 to 3{
    sum =0.0;
    for j=1 to 2
      sum = sum + a[i,j]* b[j,k];
    c[i,k]=sum;
  }
}
```

- We can work out how to transform it by locating the dependencies in it.
- Some dependencies are intrinsic to the solution, but
- Some are artefacts of the way we are solving the program.
  - if we can identify them, perhaps we can modify or remove them.

- Try 3 execution agents.

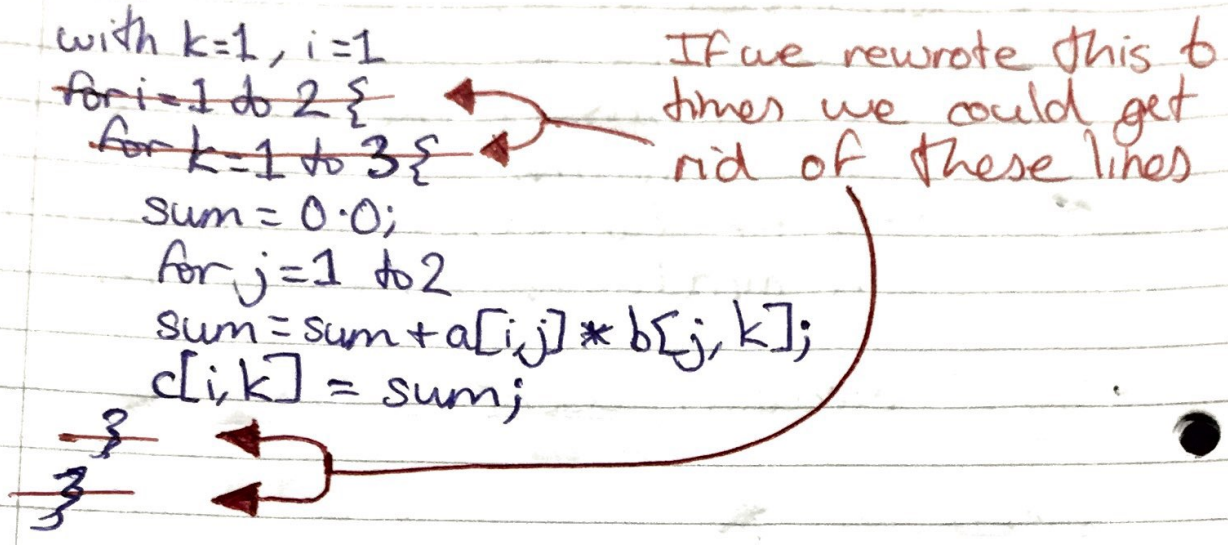
```
with k=1
for i=1 to 2{
~~for k=1 to 3{~~
    sum=0.0;
    for j=1 to 2
    sum = sum+a[i,j]* b[j,k];
    c[i,k]=sum;
~~}~~
}
```

If we rewrote this 3 times we could get rid of this line and we could get rid of this line

- Issues
  - The variable sum, as written, is common to all 3 programs.

  Solution
  - Make sum private to each program to avoid this dependency.

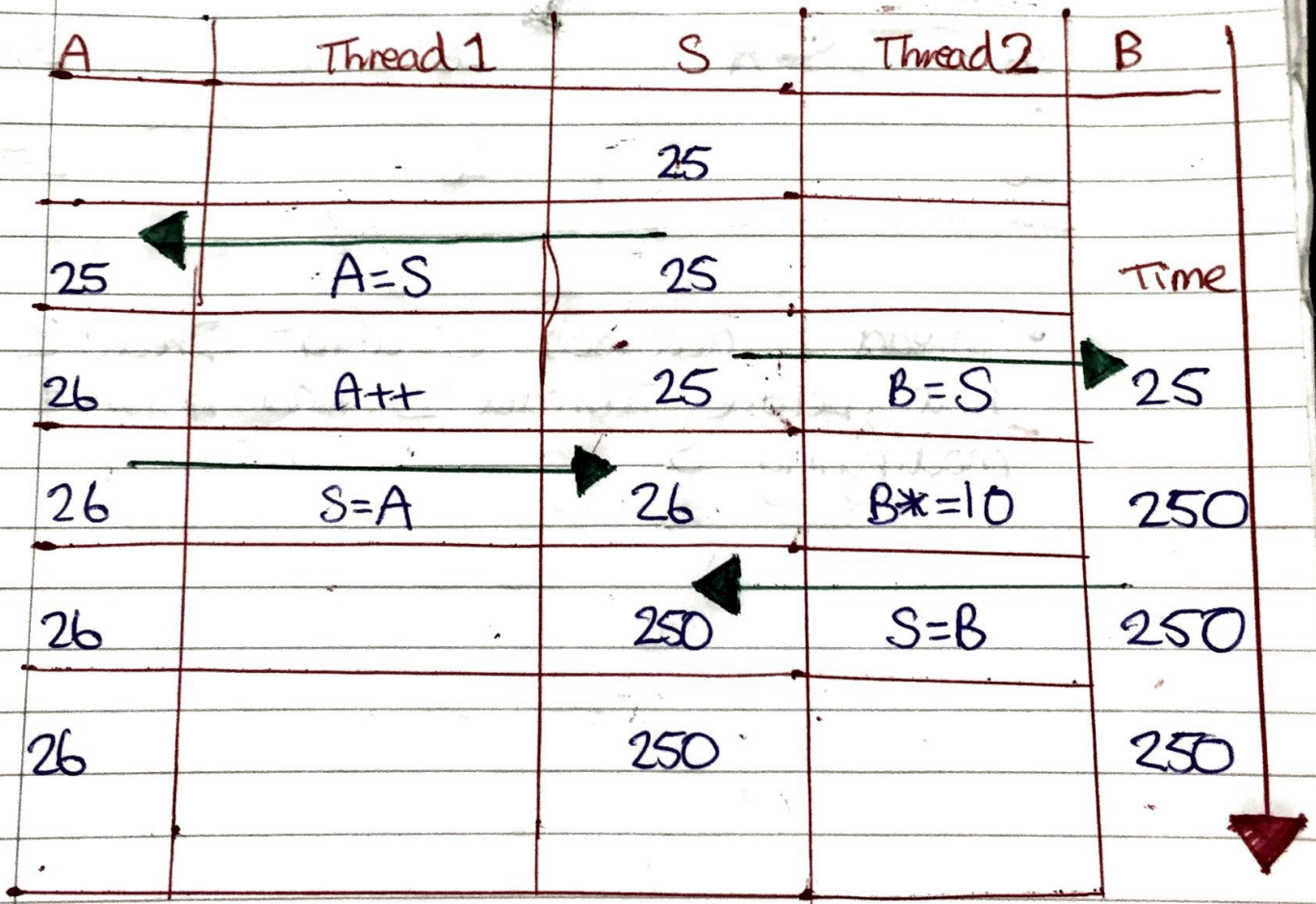
- Try 6 execution agents.



- Summarising
  - We could parallelise the original algorithm with some care:
    - Private Variables to avoid unnecessary dependencies

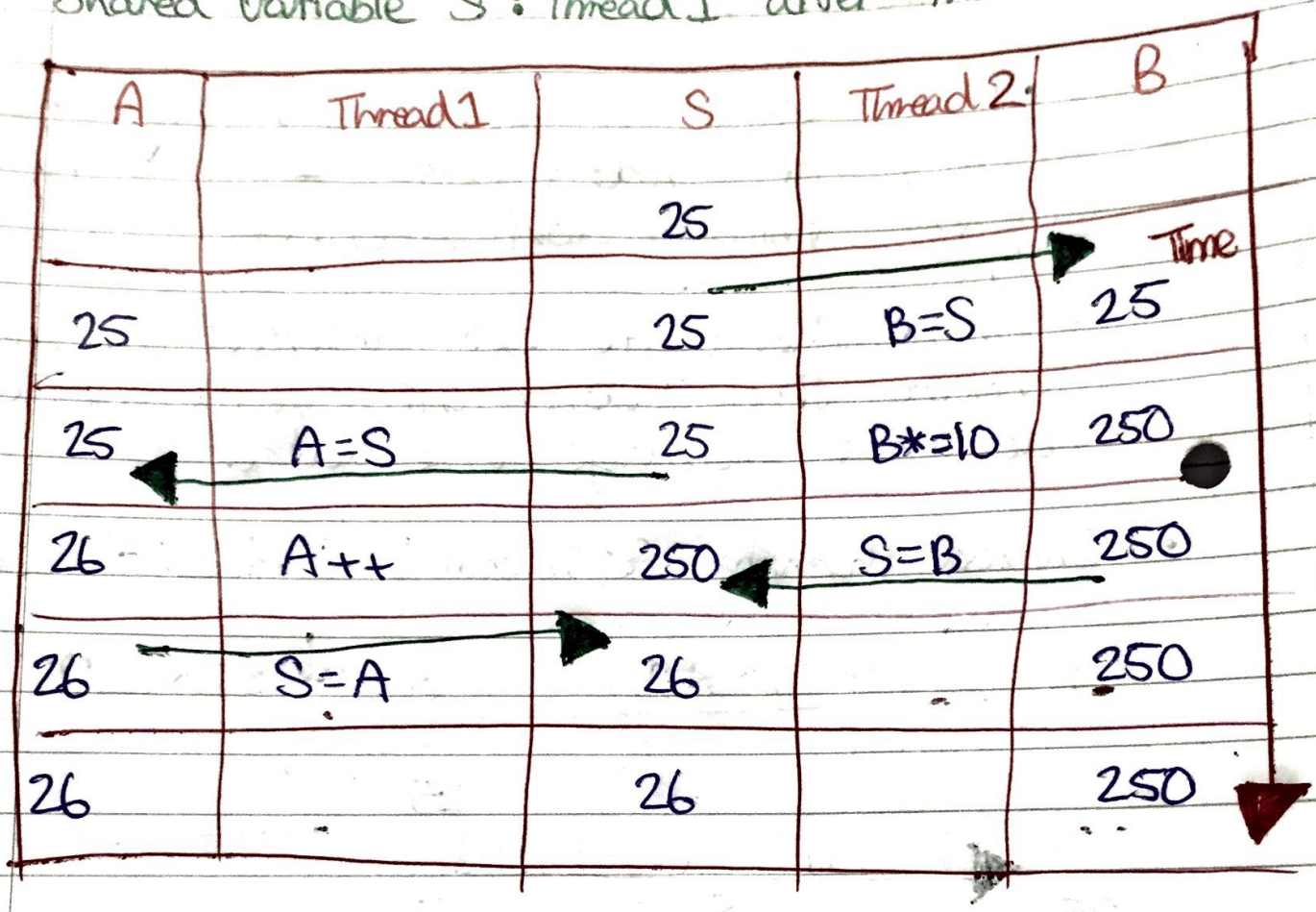
  - Actually we could break this 'Embarrasingly Parallel' problem into tiny steps/pieces;
    maybe too small
    (How will we know?)

- Synchronisation
  - We've looked at situations where the threads can operate independently - the 'write sets' of the threads don't intersect.
  - Where the write sets intersect, we must ensure that independent thread writes do not damage the data.
- Shared Variable S: Thread 1 before Thread 2



- Thread 2 accesses S after Thread 1 has finished updating S

- Shared variable S : Thread 1 after Thread 2



- Thread 1 accesses S after Thread 2 but before Thread 2 has finished modifying S

- Synchronisation
  - Problem: Access to shared resources can be dangerous
    - These are so-called 'critical' accesses.
  - Solution: Critical accesses should be made exclusive. Thus, all critical accesses to a resource are mutually exclusive.
  - In the example, both threads should have asked for exclusive access before making their updates.
    - Depending on timing, one or the other would get exclusive access first. The other would have to wait to get any kind of access.
- Mutual Exclusion in pthreads
  - Multiple exclusion is accomplished using 'mutex' variables.
  - Mutex variables are used to protect access to a resource.

- Accessing a protected Resource.

- To access a mutex-protected resource, an agent must acquire a lock on the mutex variable.
  - if the mutex variable is unlocked, it is immediately locked and the agent has acquired it. When finished, the agent must unlock it.
  - if the mutex variable is already locked, the agent has failed to acquire the lock - the protected resource is in exclusive use by someone else.
    - The agent is usually blocked until lock is acquired.
    - A non-blocking version of lock acquisition is available.

- Shared Variable S Protected by Mutex M

A	Thread 1	S	Thread 2	B	M
	Lock(M)	25			locked
25	A = S	25	lock(M)		locked
26	A++	25	[Thread Blocked]		locked
26	S = A	26			locked
26	Unlock(M)	26			locked
26		26	B = S	26	locked
		26	B* = 10	260	locked
		260	S = B	260	locked
		260	Unlock(M)		locked
		260			unlocked

Time

- Create Pthread Mutex Variable
  - Static:
    - pthread_mutex_t m = PTHREAD_MUTEX_INITILISER;
    - initially locked.
  - Dynamic:
    - pthread_mutex_init(<reference to mutex variable>, <attributes>)
- Lock and Unlock Mutex
  - pthread_mutex_lock(<mutex variable reference>);
    - acquire lock or block while waiting
  - pthread_mutex_trylock(<mutex variable reference>);
    - non blocking; check returned code
  - pthread_mutex_unlock(<mutex variable reference>);

- Two Long Examples shown in the notes on Threads
  - Example (1 of 2) - Thread
    and
  - Example (2 of 2) - ~~Thread~~ Main

- Problems
  - Voluntary
    - Mutexes 'protect' code
    - Other programmers don't have to use them to get access to the variables the code accesses
      - This is a part of the trade-off. Use processes rather than threads if you want better protection.
  - Unfair
    - If multiple threads are blocked on a mutex, the order in which they waken up is not guaranteed to be any particular order.

- Producers and Consumers
  - Essentially a "pipeline" of separate sequential programs
    - eg. concatenation of unix commands.
  - Programs communicate via buffers.
    - Implement in different ways but for example....
      - Shared memory, flags, semaphores etc.
      - Message passing over a network.
  - Flow of data is essentially one-way.
- Imagine a Situation in which...
  - A 'Consumer' is waiting for a buffer to become non-empty to take an item from it, decrementing it's counter.
  - A 'Producer' adds items to the buffer from time to time, incrementing its counter.
  - We could use a mutex to control access to the counter.
  - How do we organise the 'consumer'?

- Condition Variables.
  - A Generalised Synchronisation construct.
  - Allows you to acquire mutex lock when a "condition" relying on a shared variable is true and to sleep otherwise.
  - The "Condition Variable" is used to manage the mutex lock and the signalling.
  - This can be slightly tricky...
- Condition Variables
  - Dramatis Personae
    - A Mutex variable 'M'
    - A shared resource 'R' to be guarded by 'M'
    - A Condition variable 'V'
    - A Condition 'C'
    - A thread 'U' that wishes to use 'R', protected by 'M', on condition that 'C' is true
    - A thread 'S' that will signal 'V', presumably when it has done something that might indirectly change the value of 'C'.

- From Thread 'L1's Point Of View
  - Acquire mutex 'M'
    - This controls access to 'R'
    - But must also control access to any shared variables that 'C' depends on
  - If 'C' is true, continue - the mutex is available unfettered.
    - i.e. if 'C' is true, the mutex can be used as normal
    - i.e. if 'C' is false, wait for condition variable 'V' to be signalled.
  - This is tricky, as access to 'R' is controlled by 'M'
    - So mutex 'M' is unlocked,
    - The thread sleeps, to be woken when 'V' is signalled.
    - Then 'M' is re-acquired.
      - No Guarantee you'll get it right away - maybe another thread will.
  - So now if 'V' is signalled..
    - Since 'V' has been signalled there is now a chance that the condition 'C' is true.
    - Re-evaluate 'C':
      - If true, then the mutex is available for you to use,
      - If false back to arrow.
  - Our 'if' needs to be replaced by a 'while'.

- From Thread U's Point of View

• Acquire Mutex M	pthread_lock (&m)
• While !C	while (!C)
- Unlock M	pthread_cond_wait (&v, &m)
- Wait for V to be signaled	or
- Lock M	pthread_cond_timedwait (&v, &m, <time>)
• Continue	continue...
• Unlock M (Finished)	pthread_unlock (&m)

- Thread S

- Thread U is the 'user' of the condition variable 'V'.

- If the condition is not true, U unlocks V's mutex M and sleeps, waiting for some other thread S to signal V and thereby waking U to check the condition again.

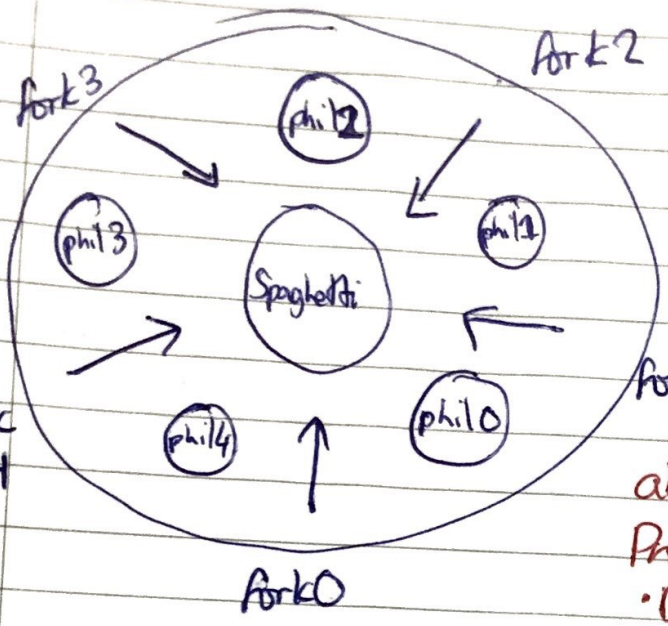
- Thread S (2)

• Acquire Mutex M	pthread_lock(&m)
• Do Something	Do something...
• Unlock Mutex M	pthread_unlock(&m)
• Signal the Condition Variable	pthread_cond_signal(&v)
- This will awaken a thread that is waiting on the condition variable	or
	pthread_cond_broadcast(&v)

- Semaphores
  - Semaphore = sema[signal] + photos [bearer] .. (from Greek)
  - Historically, semaphores were used for long distance optical signalling e.g. lanterns, flags, coloured arms etc
    - E.g. Semaphores ("The Chappe System") were used in France from late 1700's to mid 1800's for military and governmental use.
  - Computer Semaphores invented by Edsger W. Dijkstra, named after signalling Semaphores.
  - Can be thought of as a special case of condition Variable, a non-negative counter value.
  - Producer and Consumer Threads.
  - Useful to think of the variable as a record of how many units of a particular resource are available.
  - This is coupled with operations, based on a mutex to adjust that record as units are required or become free, and to wait until a unit of the resource becomes availab
  - Binary Semaphores: values are just 0 and 1
  - Counting Semaphore: values are 0, 1, 2, 3, 4, 5... etc

- Weak and Strong Semaphores
  - A Weak Semaphore stores a set of blocked processes.
  - A Strong Semaphore stores a queue of blocked processes. Processes are unblocked in the order they were blocked
    - Guarantee starvation free
    - First in First out Queue (FIFO)
  - Binary Semaphores
  - General ("Counting") Semaphores
  - Weak Semaphores
  - Strong Semaphores

- Dining Philosophers problem



Philosophers want to repeatedly think and then eat. Each needs two forks. Infinite supply of Pasta. To Ensure they wont Starve

A model problem for thinking about problems in Concurrent Programming:
- Deadlock
- Livelock
- Starvation / Fairness
- Data Corruption

- Features:
  - A Philosopher eats only if he/she has two forks
  - No two philosophers may hold the same fork simultaneously

- Characteristics of Desired Solution:
  - Freedom from deadlock
  - Freedom from Starvation
  - Efficient Behaviour generally.