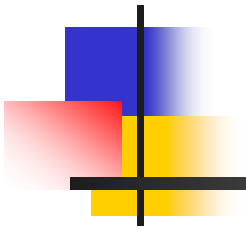


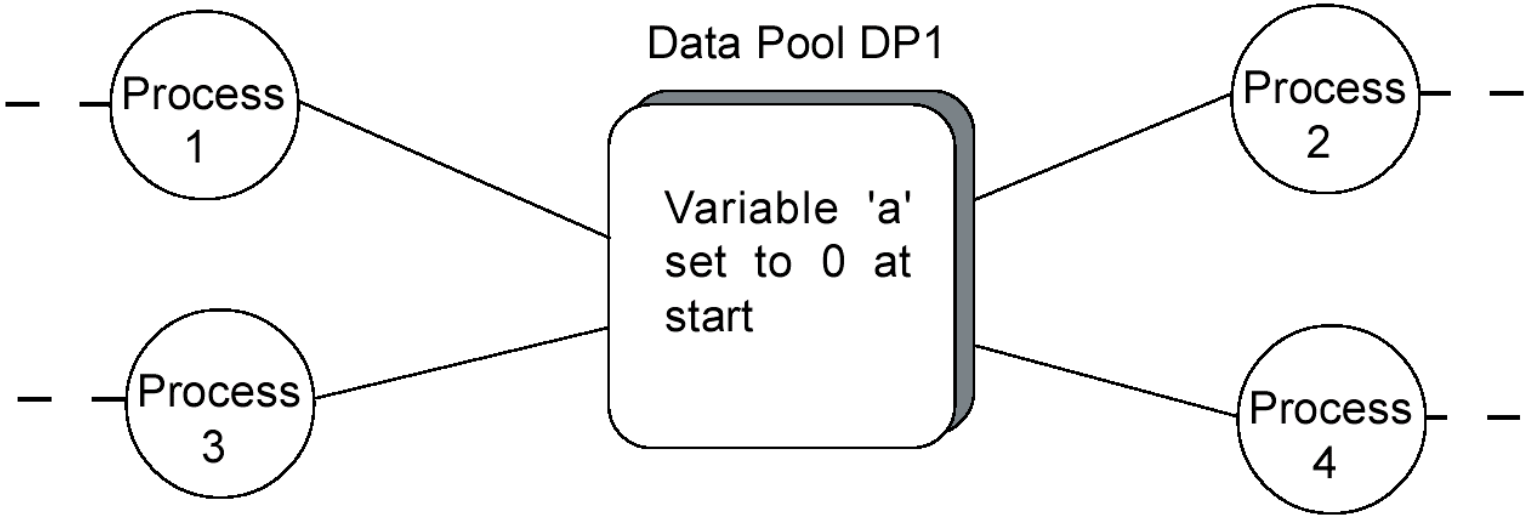
Process and Thread Synchronisation Concepts

Mutual Exclusion

- There exist in all computer systems, certain resources which are inherently 'non-sharable' by two or more processes.
- Here, we use the term non-sharable to mean "at the same time", since simultaneous access to that resource by two to more threads or processes could lead to corruption of the resource.
- In other words the resources are "mutually exclusive".
- Examples of such resources include hardware things such as
  - Printers
  - Memory
  - I/O devices
- And user-generated resources such as
  - datapools
  - global "thread-sharable" variables
- In the case of the 1st group (hardware resources), the operating system takes care of the problem by using print spoolers, memory management units and device drivers to queue up processes and threads so that they don't attempt to share these resource at the same time.
- In the case of the second, the programmer has to take responsibility for synchronising the processes so that only one of them at a time is allowed to use the resource.

The Data Pool as a Non-Sharable Resource

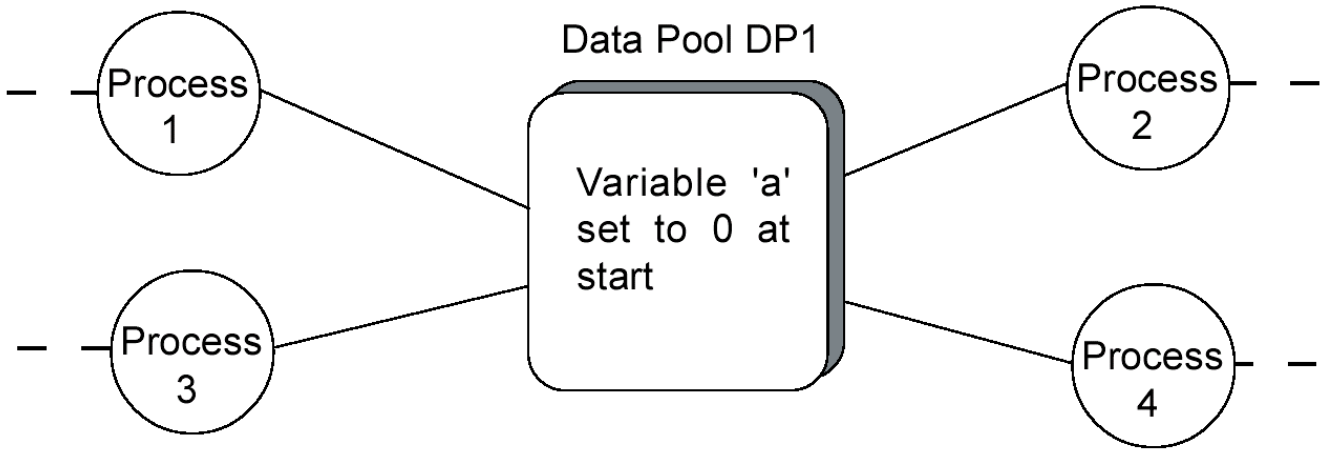
- Now although the datapool has been described previously as a randomly accessible communication medium capable of being shared by several processes, this is only true if the processes do not update the datapool at the same instant in time.
- To see why this is the case let us consider a typical arrangement, whereby we have more than one process/thread accessing a datapool for the purpose of updating some or all of the information contained within.
- Suppose then we have 4 processes in a system, each having access to a data pool DP1 which is non-sharable (See diagram below).
- Inside the data pool let us assume there is exists a single variable 'a' whose value has been set to 0 when the system was brought to life.
- Now suppose each process is doing something very simple, such as counting the number of carriage returns entered on a keyboard that is being monitored by that process.



- The variable ‘a’ in the data pool thus acts as a system wide, or global count of all carriage returns entered from any keyboard being monitored by any of the processes.
- Buried somewhere deep inside each process will reside the code that detects when a carriage return key has been pressed on its associated keyboard and updates the variable ‘a’ in the data pool DP1 with an operation similar too this C/C++ statement (ignoring the needs for pointers to access ‘a’ within the data pool)

a = a + 1 ;

- Now although this statement looks harmless enough and even appears to work, it is ultimately destined to fail if two or more processes detect their carriage returns on their keyboards at or very near the same time. Let us see why.



Analysis of the Problem

- The less than obvious problem here lies with the harmless looking statement

  a = a + 1 ;

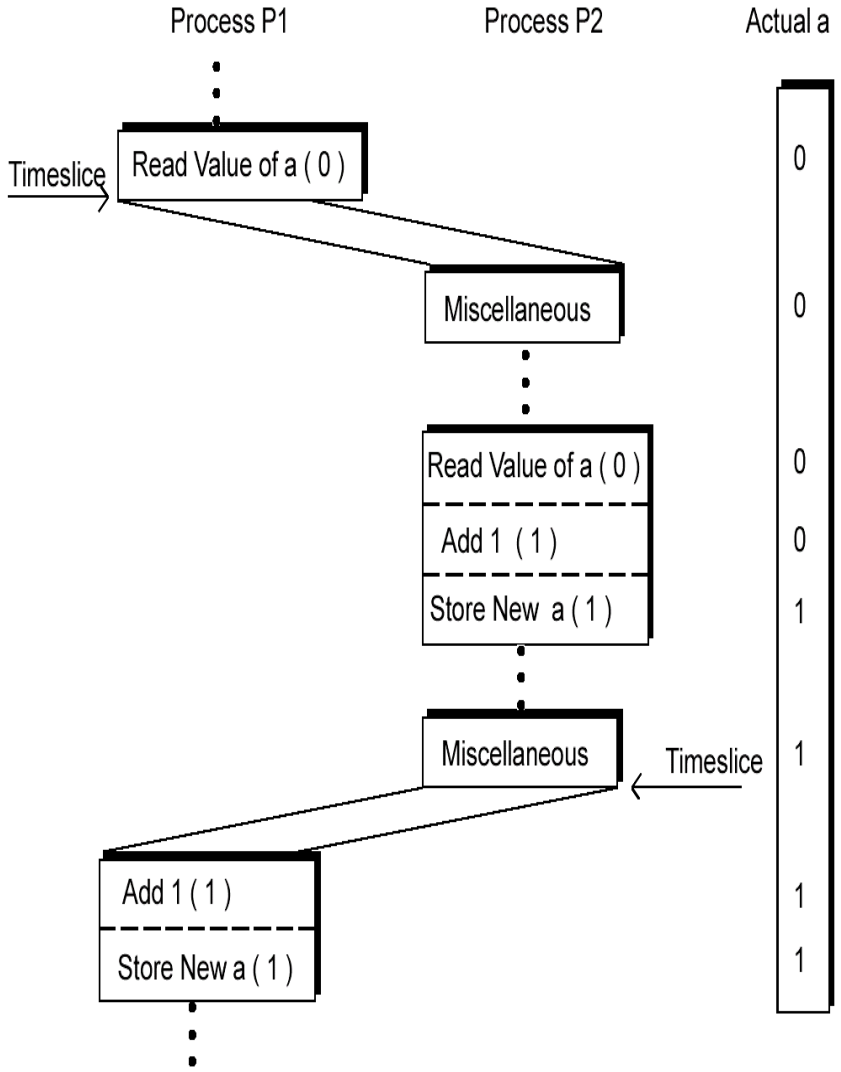
- Now although this looks to us like a single 'C/C++' statement to add 1 to 'a', we have to remember that the CPU in our computer does not understand 'C/C++' directly and the above statement will have to be compiled to machine code in order to run on our computer, an herein lies the problem.
- When our compiler translates the above statement, it generates not one but three machine code instructions for the above 'C/C++' code.
- An example Pseudo code translation along with sample 68000 assembly code for these three instructions is outlines below, where address register a0 points to 'a' in the datapool

Example Pseudo Code	68000 Assembly code
Extract the value of 'a' from DP1.	move.l (a0), d0
Add 1 to it	addi.l #1, d0
Store new value of 'a' back into DP1	move.l d0,(a0)

- (**Note**: The use of a specific increment instruction rather than an addition of 1 does not fundamentally alter the problems, although it might help in the specific example.)
- The problem, described here is that the above three instructions represent a critical operation and is not guaranteed to execute indivisibly in a time-sliced system.
- That is, it cannot be guaranteed that the above three instructions, once started, will be allowed to complete, without interruption or interference from other processes, since a time slice could take place at any instant during the execution of this 'critical section' of code. Let's see how

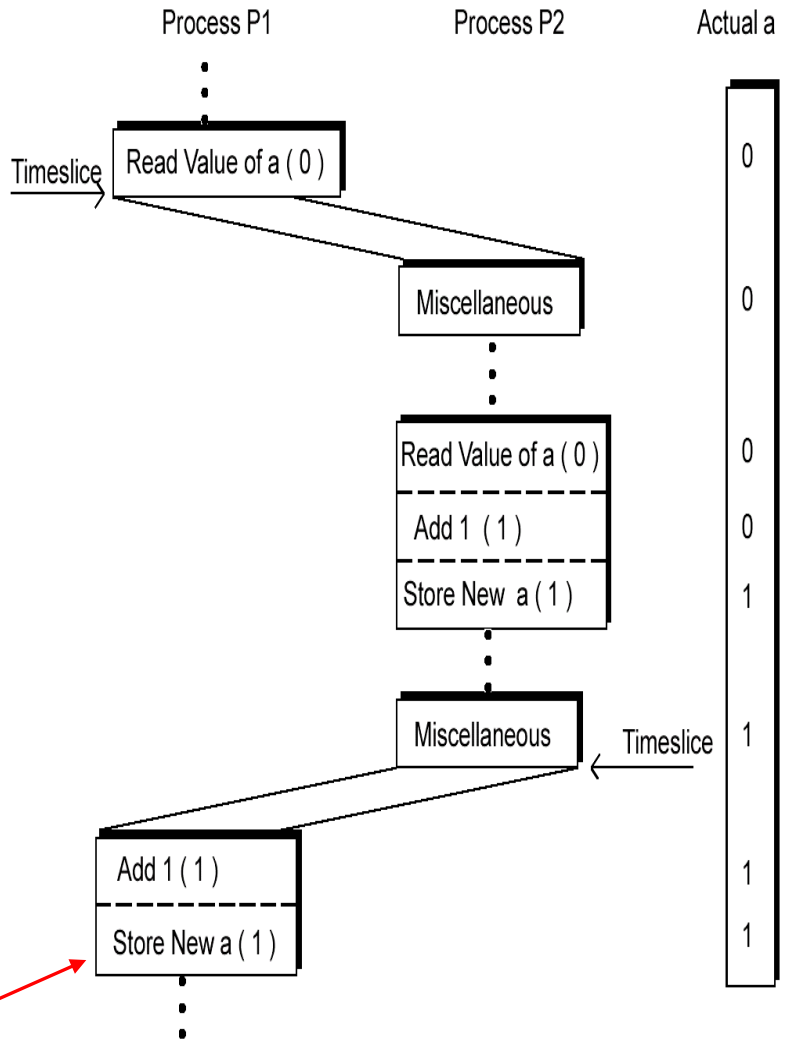
Analysis

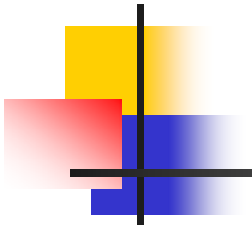
- Suppose the value of 'a' in the pool is initially '0'.
- Suppose also that some time into the life of this system, Process P1 comes along, having detected a carriage return and attempts to add 1 to the value of 'a' in the datapool.
- Thus it commences execution of the 'C' code statement a = a + 1.
- In other words
  - It reads the actual value of 'a' from the datapool into a CPU register (such as d0)
  - It is just about to adds 1 to that copy of 'a' inside its register and store it back, when the Real-time clock comes along and interrupts the execution of this critical section of code and forces a process swap.
  - At this point, the variable 'a' still has the value 0 in the pool, since process P1 has not yet completed updating it.



Process and Thread Synchronisation Concepts

- Let us suppose at this critical point in time (i.e. while P1 is suspended part through updating the datapool), process P2 also detects a carriage return on its keyboard and also decides to update the variable 'a'
- Thus it commences execution of the 'C' code statement a = a + 1. In other words
  - It reads the actual value of 'a' from the datapool into a CPU register (its value is still 0)
  - It adds 1 to that copy inside its register and manages to store back a new value for 'a' without being interrupted by the RTC. thus the actual value of 'a' in the datapool is now '1'.
- Sometime later P2 is time sliced out and P1 resumes processing where it left off, with the next instruction.
  - It has no idea that the value of 'a' in the data pool has changed (how could it, it was asleep at the time it was changing) and continues to increment and store back the value 1 back into 'a', overwriting the value stored by P1.
- At this point the system has now failed, since the variable 'a', which started off with the value 0, has been incremented by each of the two processes but has only achieved the value 1, not 2.





- You might argue that this set of circumstances only occurs very rarely, i.e. when both processes detect the carriage return at the same time and one of them happens to get time sliced part way through updating ‘a’.
- However, this is a best-case scenario. Consider extending the problem slightly to include a process updating thousands of variables, or where the updating process is very complex and certainly takes more time than a single time slice.
- The risk of corrupted data then increases considerably.
- Tutorial Question Q8 Demonstrates mutual exclusion in a meaningful way.

Solving the Problem of Mutual Exclusion in Time Sliced Systems

- When initially asked to come up with a solution to this problem, most programmers could be forgiven for attempting to solve it using a simple flag, i.e. a single byte (or bit) variable that could be used to indicate whether the non-sharable resource (in this case our data pool) is free, (flag = 0) or is busy (flag = 1).
- Such a solution would involve each re-writing each process to test the flag to establish whether the resource is free (flag == 0), and if so, proceed to mark it as busy (flag = 1) before updating the resource. The example pseudo code solution to this problem is described below.

```
                while( flag == BUSY )
Time slice ?            ;                   // poll the flag until resource becomes free
                flag = BUSY;                // mark the resource as busy
                        Read the value of 'a'   // code to update 'a' as before
                        Adding one to it
                        and storing it back
                flag = FREE ;               // release resource for another process
```

- Unfortunately this is not the solution either, since the actions of testing the flag (part of the 'while' loop), and marking it busy (flag = BUSY) is not guaranteed to be indivisible either, since a time slice could occur between these two operations. For example
- Both processes (interleaved by time slices) could both test the flag (as part of their 'while' loop), both see it as free and both mark it as busy before entering the resource at the same time.
- In other words the problem has shifted from updating the resource to updating the flag protecting it

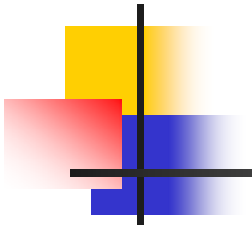
Solving Mutual exclusion in Time sliced systems: Masking Interrupts

- By analysing the problem, we see that it is the action of the real-time clock time-slicing a process part way through some critical section of code that is causing the problem.
- One solution then would be to disable or mask the RTC interrupts during the critical section of testing the flag and setting it, as shown below.
- Notice how interrupts have been enabled twice. Can you think why?

```
while( attempting to access pool)                    {
        disable interrupts
        if ( flag = = FREE)      {
                flag = BUSY;
                enable interrupts;
                break;
        }
        else
                enable interrupts ;
        SLEEP(10) ;                    // give up CPU to reduce wastage
}

Read the value of 'a'                  // code to update 'a' as before
Add one
Storing it back

flag = FREE ;                          // release resource for use by another process
```



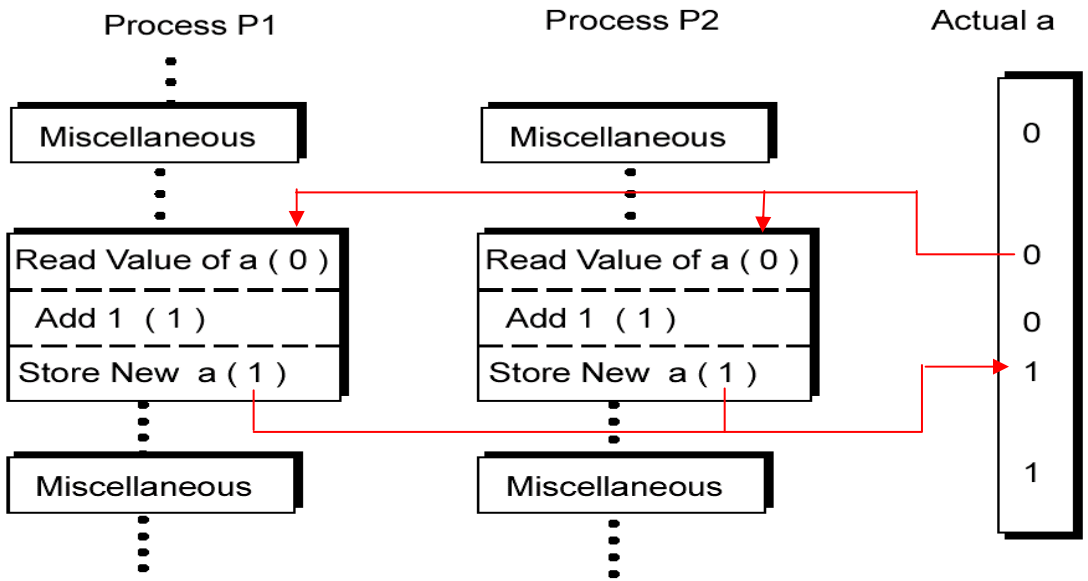
Problems with this Approach

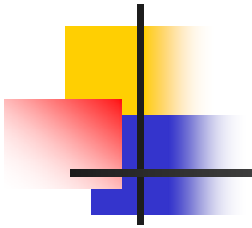
- It requires that each programmer writes a copy of some rather nasty and very critical code within each process wanting to use a non-sharable resource. OK so we could turn it into a subroutine and let the process call that, so it's not a major problem.
- However, disabling the RTC or interrupts should never be left to the discretion of a users process, since simply having access to that ability would invite programmers to abuse it.

  After all, who is to say whether your process is the most important one in the system. If a process disables the RTC it effectively becomes 'god' having life and death privileges over other processes and even external interrupt driven events in the system.
- More important is the fact that a process which is unable to gain immediate access to the resource will enter a 'busy-waiting' loop, wasting large amounts of CPU time but achieving nothing. Such an approach dramatically affects the performance of a concurrent system, since the CPU still has to schedule the process even though to all intents and purposes, it is achieving nothing.
- More serious still is the fact that with such a scheme, there is a possibility of indefinite lockout of a process, since there is no natural queuing mechanism to order the processes and release them in the order they arrive. In effect, a process could always be polling the flag and always find it busy as other processes leave and enter the resource will it was time sliced out.
- This last point would be totally unacceptable in a real-time system, as it means that the system become non – deterministic, with non calculatable response times.

Mutual Exclusion Problems in Multi-Processor Systems

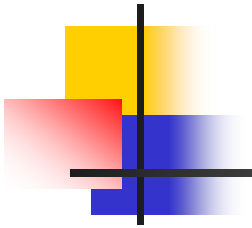
- Mutual exclusion is also a problem in multi-processor systems, where individual processes/threads run in true parallel on individual CPU's (i.e. no time-slicing).
- Now if two processes/threads running on separate CPUs both attempted to access the same resource, such as the variable 'a' in the data pool discussed previously, then the following could happen
- As you can see, both processes (CPU's) read 'a' and see it as 0, both increment their copy of 'a' and then store it back.
- The problem once again is that 'a' has been incremented twice, but its true value does not reflect this





Solving Mutual Exclusion in a Multi-CPU System

- The solution here is for all CPU's to co-operate when using the resource by first executing a special CPU instruction, guaranteed to test and set a flag as one indivisible or atomic operation, i.e. one which does not release control of the address bus to another CPU while the testing and setting operation is in progress.
- This last point is critical, since it means that another CPU cannot gain access to the same flag variable while the 1st CPU is testing and updating it.
- On the 68000 Family of CPUs, this instruction is the TAS or Test-and-Set instruction whose operation is outlined below.
  - Read the value of a flag at a specified address.
  - Set the value of the flag to logic 1 (regardless of what was before the instruction).
  - Return the previous value of the flag (before this process set it).
- Crucially this is not the same as three separate instructions that together test and set the flag.
- That arrangement would leave the back door open for a 2nd CPU, interleaved with the 1st, to both test, and set the flag at the same time, thus negating the solution.



Operation and Explanation of TAS

- If the flag was already set when the TAS instruction was executed, indicating that another CPU was using the resource being protected by the flag, then setting it again with TAS makes no difference and the value returned by the TAS instruction will be 1
- If it was not set, indicating that no CPUs were using the resource, it will be set and the value returned by the TAS instruction will be 0.
- Thus, you could arrange for all your processes to include the following assembly language code into their programs before attempting to gain access to the resource

```
loop  TAS   $10000      Test and set the flag at location 10000
      bne   loop        jump back and try again if resource busy

      .....             Got resource, do what I like now !!!

      clr.b $10000      Clear the flag to release resource.
```

- Again executing this form of busy-waiting loop (known as a spin-lock) for a resource to become free is very wasteful of CPU time and would be unacceptable in a time slicing single CPU system. However it's use in multiple CPU systems is often deemed acceptable because it doesn't block or slow down other processes running on separate CPUs