CS 259 - HW 1

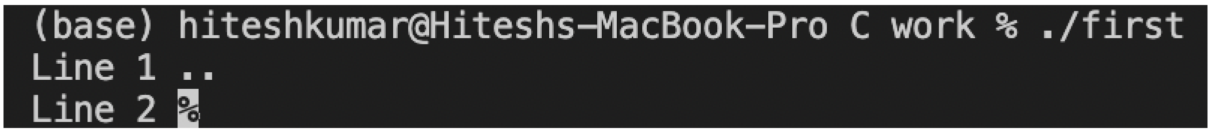
Name : Hitesh Kumar

SJSU ID : 015237989

Solution 1

The Output observed in the **terminal** is below where **Line 1** is **printed before Line 2.** This is because Line buffering has occurred due to the presence of **\n along with printf** function. Line buffers outputs when a new line occurs.

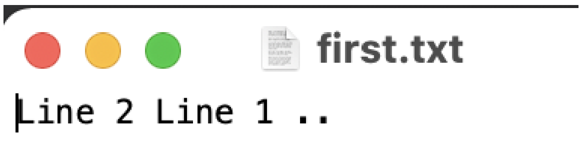
The output is directed to the terminal.



```
(base) hiteshkumar@Hiteshs-MacBook-Pro C work % ./first
Line 1 ..
Line 2 %
```

The Output observed in **target pipe file** is below ; where **Line 2** is printed before **Line 1.** This type of buffering is called **unbuffered** due to present of **write() function**. This method outputs the result no matter we have a new line or not. Its motto is to output as quickly as possible.

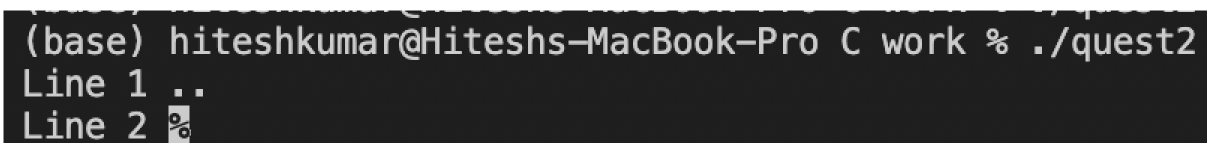
The output generated is written using write function to the desired output file.



```
Line 2 Line 1 ..
```

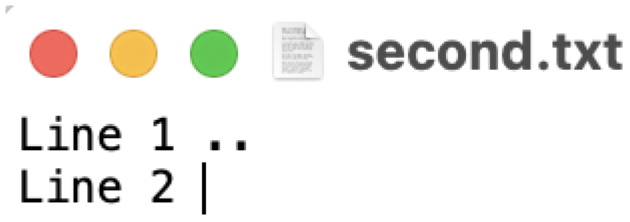
Solution 2

The Output observed in the terminal is below :



```
(base) hiteshkumar@Hiteshs-MacBook-Pro C work % ./quest2
Line 1 ..
Line 2 %
```

The Output observed in file is below :



```
Line 1 ..
Line 2 
```

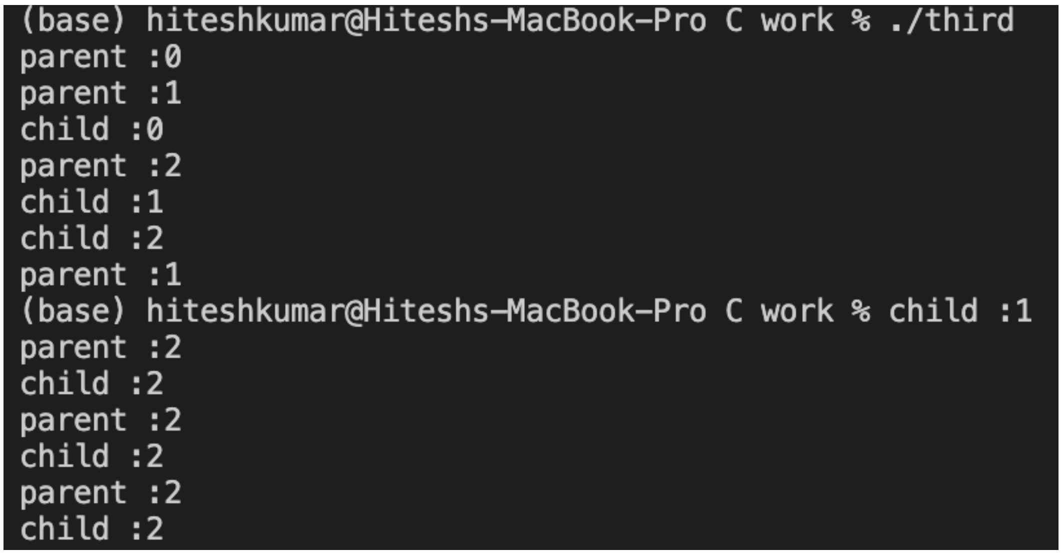
In both cases, **Line 1** is **printed first** This is because the output is flushed using **fflush(stdout) function.** The buffer gets flushed manually by the developer or by the programmer using fflush function. Hence, here both the ouput is same.

Solution 3

The Output observed in the terminal **does not follow any order** every time. The printing order on the terminal of child and parents is **system specific as per its own kernel**. For each system scheduler, the output generated is different with no specific pattern.

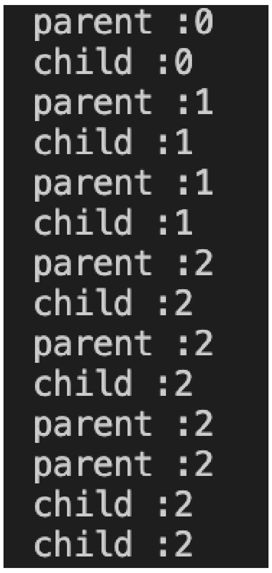
Since we are not using wait() function, the execution happens in order as per the scheduler of the operating system.

Run 1:



```
(base) hiteshkumar@Hiteshs-MacBook-Pro C work % ./third
parent :0
parent :1
child :0
parent :2
child :1
child :2
parent :1
(base) hiteshkumar@Hiteshs-MacBook-Pro C work % child :1
parent :2
child :2
parent :2
child :2
parent :2
child :2
```

Run 2:



```
parent :0
child :0
parent :1
child :1
parent :1
child :1
parent :2
child :2
parent :2
child :2
parent :2
parent :2
child :2
child :2
```

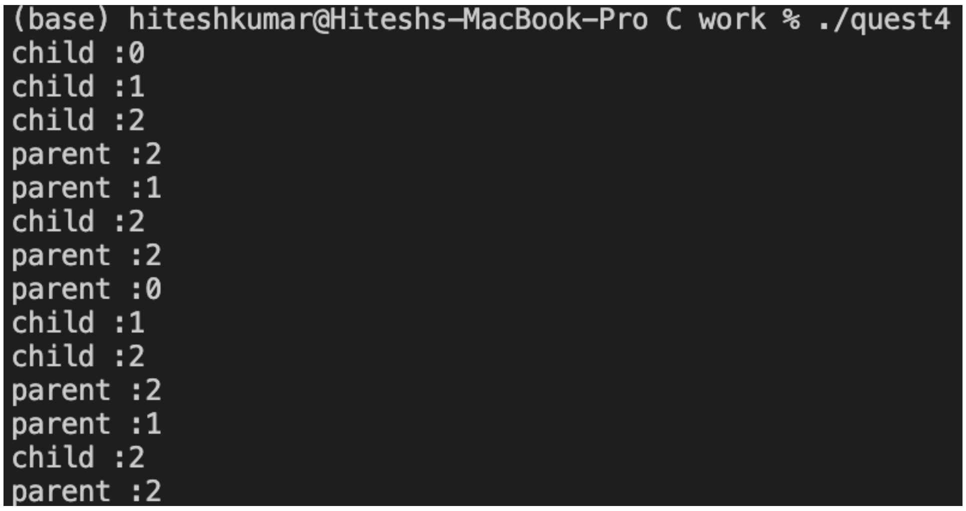
Solution 4(a)

As compared to program 3, we have used wait() function in this question. The wait() function let the child process gets completed first.

Order of traversal : rightmost, depth-first order.

The is because of the wait() function which is introduced after fork. This allows the child process to execute first and after its completion, it get backs to the parents process for its execution.

Terminal output :

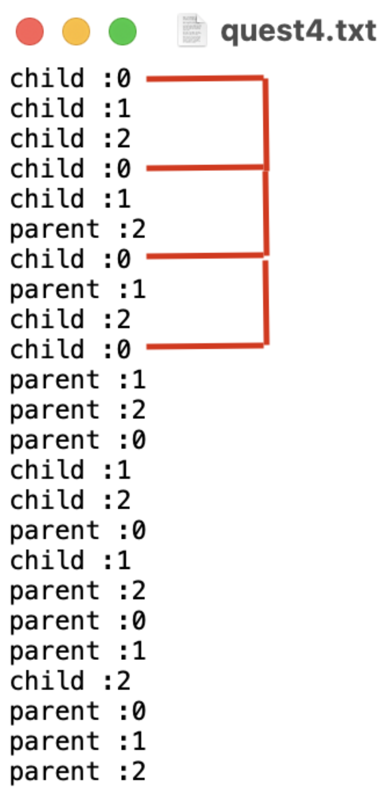


```
(base) hiteshkumar@Hiteshs-MacBook-Pro C work % ./quest4
child :0
child :1
child :2
parent :2
parent :1
child :2
parent :2
parent :0
child :1
child :2
parent :2
parent :1
child :2
parent :2
```

Solution 4(b)

Pipe file output :

Here we see various printf() statements getting repeated. This is because printf() just before forking is in buffer which gets inherited in its child process. hence we can see many printfs getting repeated while traversing the tree. The order is same rightmost-depth first with multiple repeats of printf() statements.

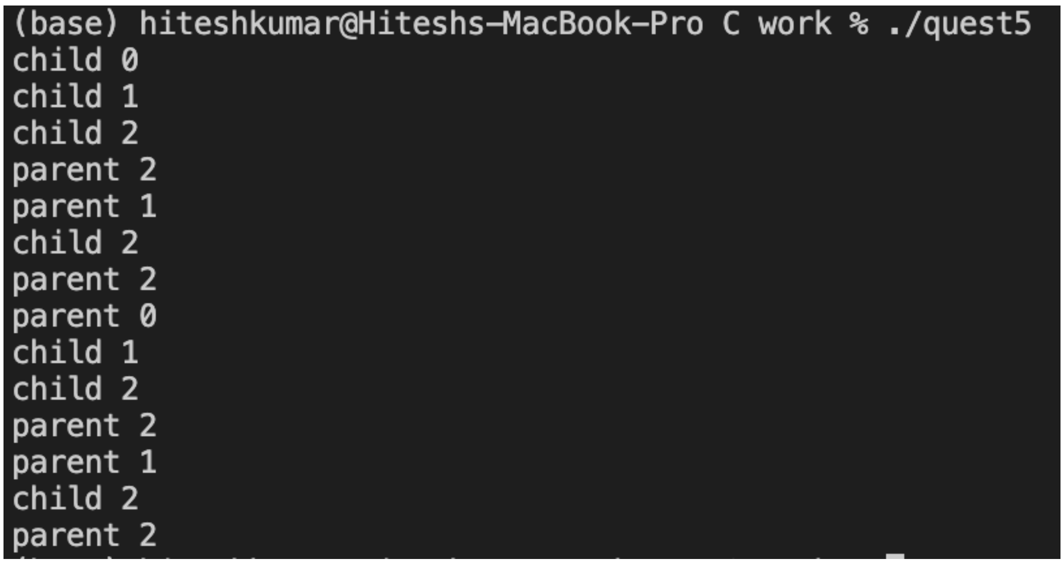

quest4.txt

```
child :0
child :1
child :2
child :0
child :1
parent :2
child :0
parent :1
child :2
child :0
parent :1
parent :2
parent :0
child :1
child :2
parent :0
child :1
parent :2
parent :0
parent :1
child :2
parent :0
parent :1
parent :2
```

Solution 5

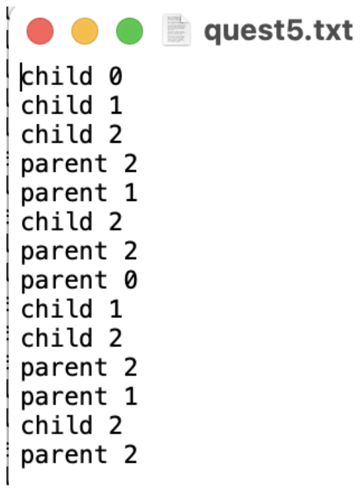
In both the output, whether through terminal interactively or via text file, both the outputs are **same**. This is because we have used **fflush(stdout)** here. With this fully buffering technique, we are not dependent on the kernel; we manually flushes the printf() statements in our code. Hence, the output is same.

Terminal Output :



```
(base) hiteshkumar@Hiteshs-MacBook-Pro C work % ./quest5
child 0
child 1
child 2
parent 2
parent 1
child 2
parent 2
parent 0
child 1
child 2
parent 2
parent 1
child 2
parent 2
```

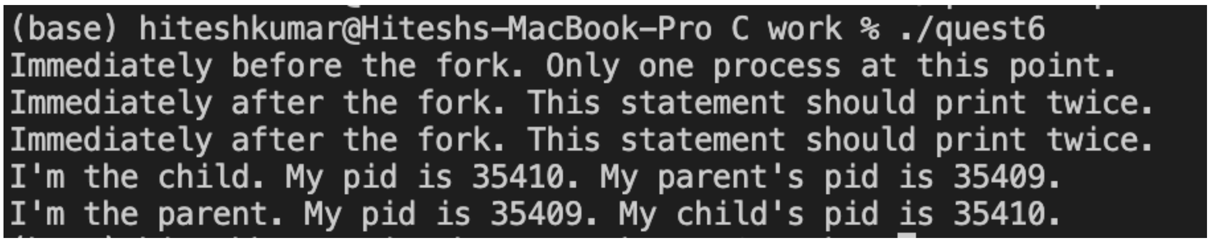
Pipe file Output :



```
child 0
child 1
child 2
parent 2
parent 1
child 2
parent 2
parent 0
child 1
child 2
parent 2
parent 1
child 2
parent 2
```

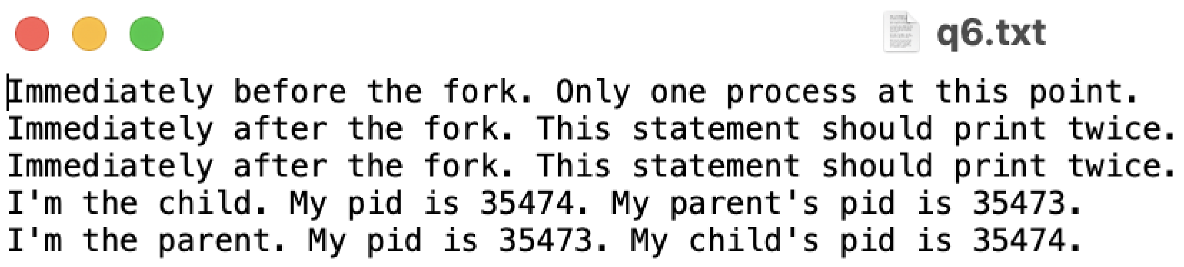
Solution 6

Test Case 1 : Output in console



```
(base) hiteshkumar@Hiteshs-MacBook-Pro C work % ./quest6
Immediately before the fork. Only one process at this point.
Immediately after the fork. This statement should print twice.
Immediately after the fork. This statement should print twice.
I'm the child. My pid is 35410. My parent's pid is 35409.
I'm the parent. My pid is 35409. My child's pid is 35410.
```

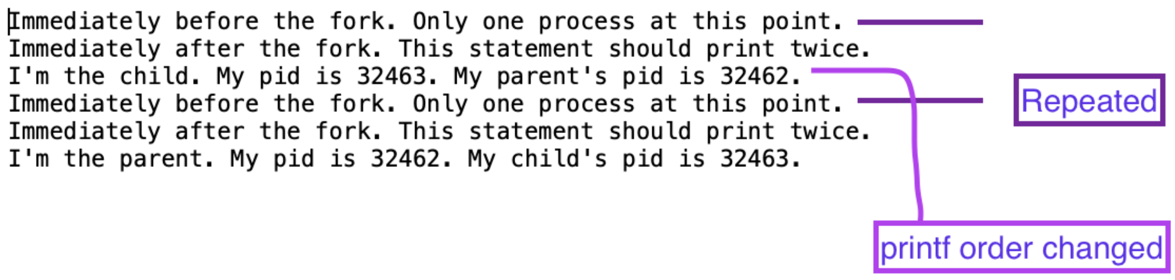
Test Case 2 : Output in text file



```
Immediately before the fork. Only one process at this point.
Immediately after the fork. This statement should print twice.
Immediately after the fork. This statement should print twice.
I'm the child. My pid is 35474. My parent's pid is 35473.
I'm the parent. My pid is 35473. My child's pid is 35474.
```

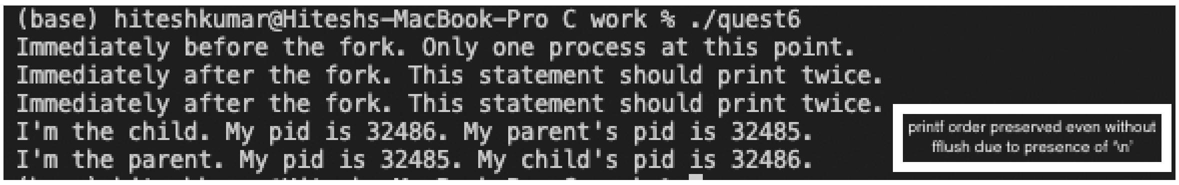
Result : Test Case 1 and Test Case 2 are same. Hence the use of fflush and wait function is correct in the program below.

Test case 3 : removing only fflush(stdout) with wait() function , Output : file



```
Immediately before the fork. Only one process at this point.
Immediately after the fork. This statement should print twice.
I'm the child. My pid is 32463. My parent's pid is 32462.
Immediately before the fork. Only one process at this point.
Immediately after the fork. This statement should print twice.
I'm the parent. My pid is 32462. My child's pid is 32463.
```

Test case 4 : removing only fflush(stdout) with wait() function , Output : TERMINAL



```
(base) hiteshkumar@Hiteshs-MacBook-Pro C work % ./quest6
Immediately before the fork. Only one process at this point.
Immediately after the fork. This statement should print twice.
Immediately after the fork. This statement should print twice.
I'm the child. My pid is 32486. My parent's pid is 32485.
I'm the parent. My pid is 32485. My child's pid is 32486.
```

printf order preserved even without fflush due to presence of '\n'

Source Code :

#author : Hitesh Kumar

#include<stdio.h>

#include<unistd.h>

#include<sys/types.h>

#include<stdlib.h>

int main(){

int pid,A_pid, B_pid,pid2,C_pid,pid3,pid4;

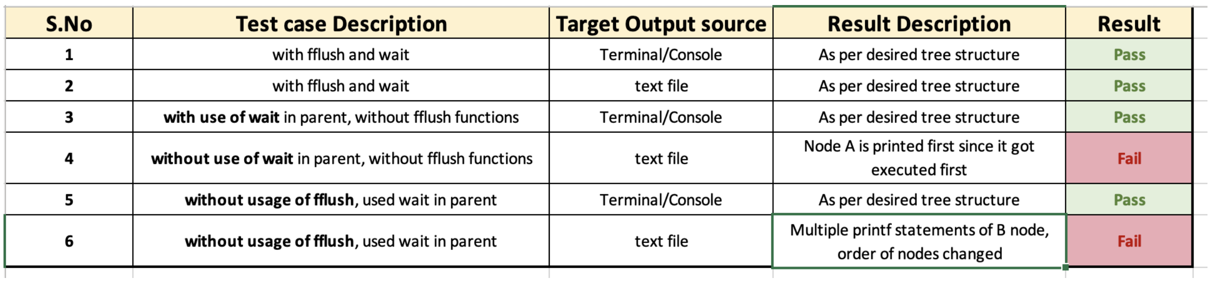
printf("Immediately before the fork. Only one process at this point.\n");

```
fflush(stdout);
pid  = fork();
printf("Immediately after the fork. This statement should print twice.\n");
fflush(stdout);
// To create Nodes A and B
if (pid >0 )  //  A node
{
wait(NULL);
A_pid= getpid();
printf("I'm the parent. My pid is %d. My child's pid is %d.\n",getpid(),pid);
fflush(stdout);
}
else if (pid ==0)  // B node
{
B_pid= getpid();
printf("I'm the child. My pid is %d. My parent's pid is %d.\n",getpid(),getppid());
fflush(stdout);
}
else{
printf("CANT FORK ! \n");
fflush(stdout);
}
}
```

Solution 7

Note : The output has been printed with the following notation :

'In Node X' : means we are traversing that particular node 'X' at that time.



S.No	Test case Description	Target Output source	Result Description	Result
1	with fflush and wait	Terminal/Console	As per desired tree structure	Pass
2	with fflush and wait	text file	As per desired tree structure	Pass
3	**with use of wait** in parent, without fflush functions	Terminal/Console	As per desired tree structure	Pass
4	**without use of wait** in parent, without fflush functions	text file	Node A is printed first since it got executed first	Fail
5	**without usage of fflush**, used wait in parent	Terminal/Console	As per desired tree structure	Pass
6	**without usage of fflush**, used wait in parent	text file	Multiple printf statements of B node, order of nodes changed	Fail

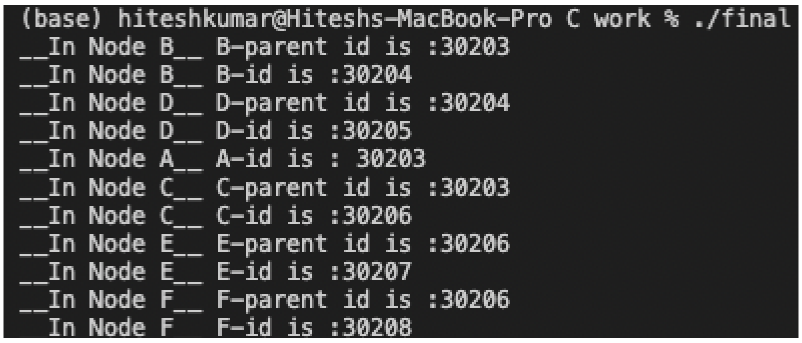
Test Case 1 : with flush and wait function - output : Terminal

Highlighted Desired Terminal Output :

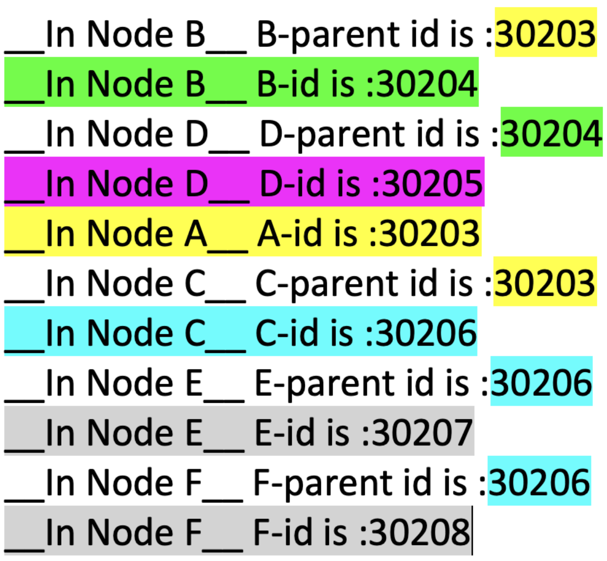
For Explanation, Same coloring has been used for nodes with their IDs and Parent ids.

Input : Initially A node is created by forking the current running process.

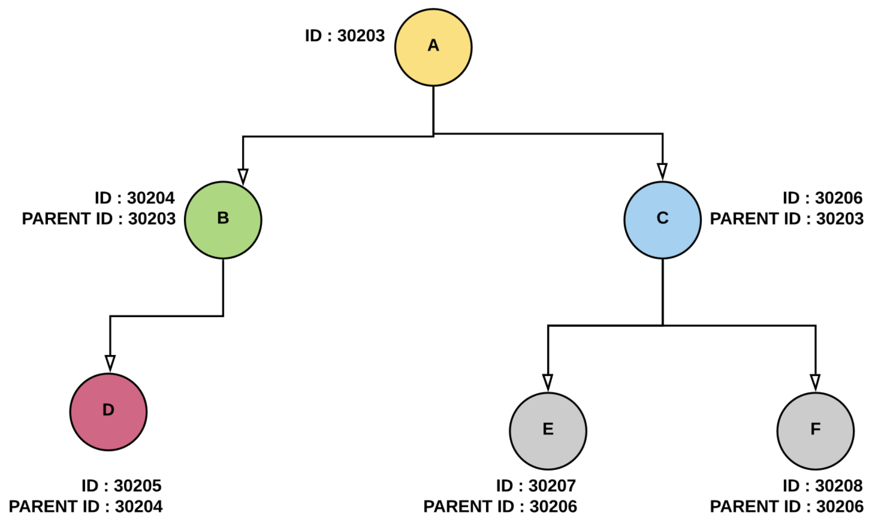
By using **getpid()** function as well as using **getppid()** function, we can handle the child and current parent block and fork again to create new child and parents.



```
(base) hiteshkumar@Hiteshs-MacBook-Pro C work % ./final
__In Node B__ B-parent id is :30203
__In Node B__ B-id is :30204
__In Node D__ D-parent id is :30204
__In Node D__ D-id is :30205
__In Node A__ A-id is : 30203
__In Node C__ C-parent id is :30203
__In Node C__ C-id is :30206
__In Node E__ E-parent id is :30206
__In Node E__ E-id is :30207
__In Node F__ F-parent id is :30206
__In Node F__ F-id is :30208
```



```
__In Node B__ B-parent id is :30203
__In Node B__ B-id is :30204
__In Node D__ D-parent id is :30204
__In Node D__ D-id is :30205
__In Node A__ A-id is :30203
__In Node C__ C-parent id is :30203
__In Node C__ C-id is :30206
__In Node E__ E-parent id is :30206
__In Node E__ E-id is :30207
__In Node F__ F-parent id is :30206
__In Node F__ F-id is :30208
```

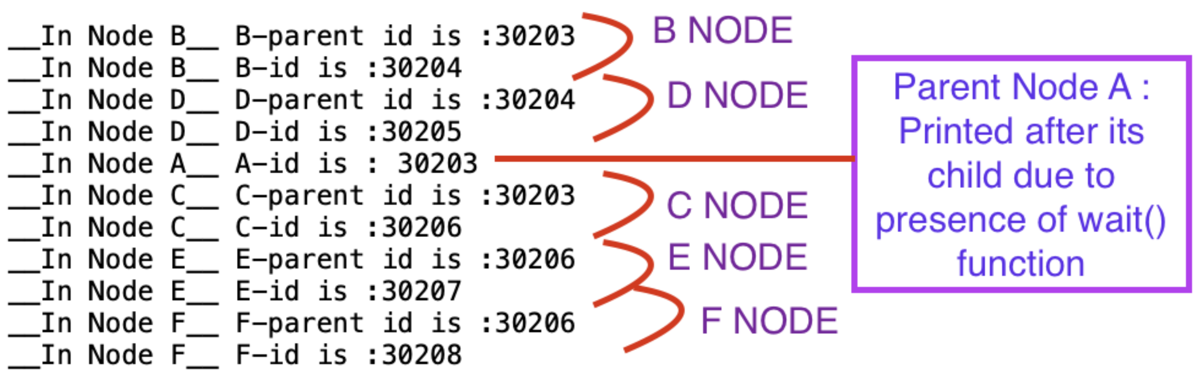


Test Case 2 : with flush and wait function - output : Text File

```
__In Node B__ B-parent id is :30203
__In Node B__ B-id is :30204
__In Node D__ D-parent id is :30204
__In Node D__ D-id is :30205
__In Node A__ A-id is : 30203
__In Node C__ C-parent id is :30203
__In Node C__ C-id is :30206
__In Node E__ E-parent id is :30206
__In Node E__ E-id is :30207
__In Node F__ F-parent id is :30206
__In Node F__ F-id is :30208
```

Result : Test Case 1 and Test Case 2 are same. Hence the use of fflush(stdout) is correct

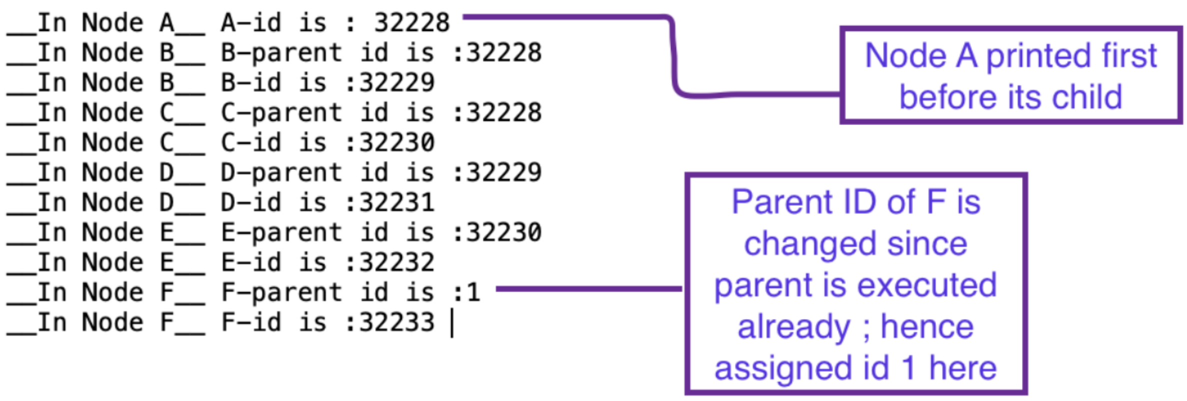
Test Case 3 : Use of wait() function in Parent A



```
__In Node B__  B-parent id is :30203
__In Node B__  B-id is :30204
__In Node D__  D-parent id is :30204
__In Node D__  D-id is :30205
__In Node A__  A-id is : 30203
__In Node C__  C-parent id is :30203
__In Node C__  C-id is :30206
__In Node E__  E-parent id is :30206
__In Node E__  E-id is :30207
__In Node F__  F-parent id is :30206
__In Node F__  F-id is :30208
```

B NODE
D NODE
C NODE
E NODE
F NODE
Parent Node A : Printed after its child due to presence of wait() function

Test Case 4 : Removal of wait() function in Parent A



```
__In Node A__  A-id is : 32228
__In Node B__  B-parent id is :32228
__In Node B__  B-id is :32229
__In Node C__  C-parent id is :32228
__In Node C__  C-id is :32230
__In Node D__  D-parent id is :32229
__In Node D__  D-id is :32231
__In Node E__  E-parent id is :32230
__In Node E__  E-id is :32232
__In Node F__  F-parent id is :1
__In Node F__  F-id is :32233
```

Node A printed first before its child
Parent ID of F is changed since parent is executed already ; hence assigned id 1 here

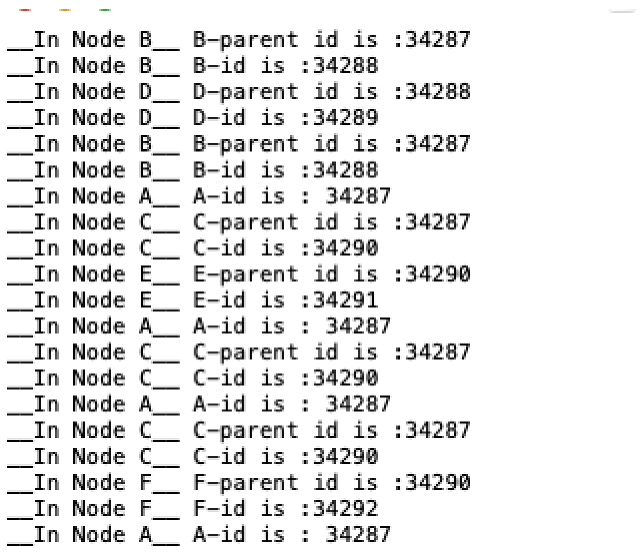
Result : **The Test case 3 and 4 are different due to wait() function. Even the order of printf() statements are also changed. Therefore, a wait function changes the order of printf() statements.**

Test Case 5 : Removal of fflush() function , keeping wait function in parents, output : console/terminal

Result : Desired Tree structure as test case 1

Test Case 6 : Removal of fflush() function , keeping wait function in parents,output : text file

Result : Multiple printf statements of B node, order of nodes changed


```
__In Node B__ B-parent id is :34287
__In Node B__ B-id is :34288
__In Node D__ D-parent id is :34288
__In Node D__ D-id is :34289
__In Node B__ B-parent id is :34287
__In Node B__ B-id is :34288
__In Node A__ A-id is : 34287
__In Node C__ C-parent id is :34287
__In Node C__ C-id is :34290
__In Node E__ E-parent id is :34290
__In Node E__ E-id is :34291
__In Node A__ A-id is : 34287
__In Node C__ C-parent id is :34287
__In Node C__ C-id is :34290
__In Node A__ A-id is : 34287
__In Node C__ C-parent id is :34287
__In Node C__ C-id is :34290
__In Node F__ F-parent id is :34290
__In Node F__ F-id is :34292
__In Node A__ A-id is : 34287
```

Source Code :

```
// Author : Hitesh Kumar

// OS: MAC

// Editor : VS CODE , gcc

#include<stdio.h>

#include<unistd.h>

#include<sys/types.h>

#include<stdlib.h>

int main(){

int pid,A_pid, B_pid,pid2,C_pid,pid3,pid4;

pid  = fork();

fflush(stdout);

// To create Nodes A and B

if (pid >0 )  //  A node

{

wait(NULL);

A_pid= getpid();
```

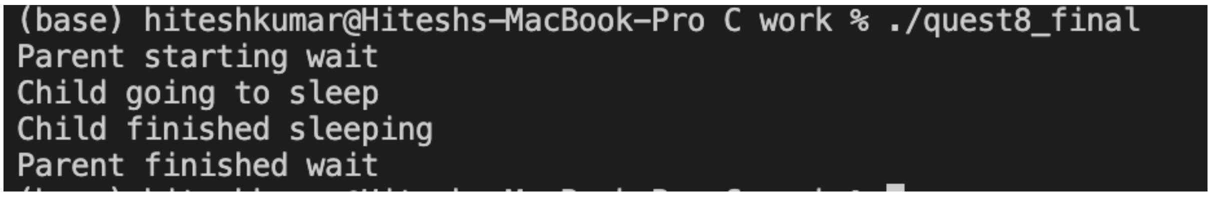
```
printf("__In Node A__ A-id is : %d \n",A_pid);

fflush(stdout);

}

else if (pid ==0)  // B node

{

B_pid= getpid();

printf("__In Node B__ B-parent id is :%d \n",getppid());

fflush(stdout);

printf("__In Node B__ B-id is :%d \n",getpid());

fflush(stdout);

}

else{

printf("CANT FORK ! \n");

fflush(stdout);

}

pid2= fork();

if(pid2>0)

{

wait(NULL);

}

else if ( pid2 ==0 && getppid()==A_pid){

C_pid= getpid();

printf("__In Node C__ C-parent id is :%d \n",getppid());

fflush(stdout);

printf("__In Node C__ C-id is :%d \n",getpid());

fflush(stdout);

// To create Nodes E

pid3= fork ();

if(pid3 ==0){
```

```
printf("__In Node E__ E-parent id is :%d \n",getppid());
fflush(stdout);
printf("__In Node E__ E-id is :%d \n",getpid());
fflush(stdout);
}
else if(pid3 >0){
wait(NULL);
pid4= fork();
if(pid4 ==0){
// To create Nodes F
printf("__In Node F__ F-parent id is :%d \n",getppid());
fflush(stdout);
printf("__In Node F__ F-id is :%d \n",getpid());
fflush(stdout);
}
}
}
else if ( pid2 ==0 && getppid()==B_pid){
printf("__In Node D__ D-parent id is :%d \n",getppid());
fflush(stdout);
printf("__In Node D__ D-id is :%d \n",getpid());
fflush(stdout);
}
} // Main end
```

Solution 8

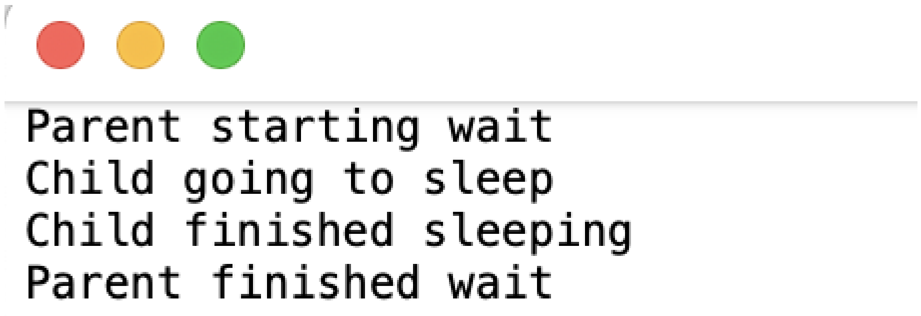
1. *with wait(&status) function*

Test Case 1: output in console



```
(base) hiteshkumar@Hiteshs-MacBook-Pro C work % ./quest8_final
Parent starting wait
Child going to sleep
Child finished sleeping
Parent finished wait
```

Test Case 2: output in txt file



```
Parent starting wait
Child going to sleep
Child finished sleeping
Parent finished wait
```

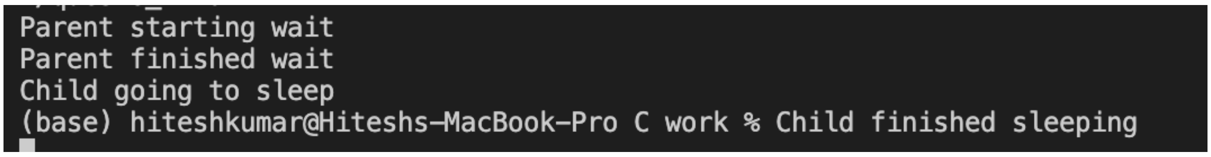
Test Case 1 and Test case 2 are same.

Source Code (*with wait(&status) function*) :

```
#include<stdio.h>

#include<unistd.h>

#include<sys/types.h>

#include<stdlib.h>

#include <unistd.h>

int main(){

int pid,A_pid, B_pid,pid2,C_pid,pid3,pid4;

pid  = fork();

if (pid >0 )

{

printf("Parent starting wait \n");
fflush(stdout);

int t= 5;

wait(&t);

printf("Parent finished wait \n");

fflush(stdout);

}

else if (pid ==0)

{

printf("Child going to sleep \n");

fflush(stdout);

sleep(5);

printf("Child finished sleeping \n");

fflush(stdout);

}

else{

printf("CANT FORK ! \n");

//fflush(stdout);

}

}
```

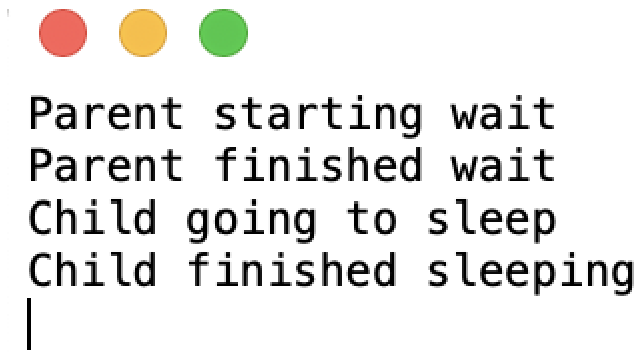
2. *without wait(&status) function*

Test Case 1: output in console



```
Parent starting wait
Parent finished wait
Child going to sleep
(base) hiteshkumar@Hiteshs-MacBook-Pro C work % Child finished sleeping
```

Test Case 2: output in txt file



```
Parent starting wait
Parent finished wait
Child going to sleep
Child finished sleeping
```

Source Code (*without wait(&status) function*) :

```
#include<stdio.h>

#include<unistd.h>

#include<sys/types.h>

#include<stdlib.h>

#include <unistd.h>

int main(){

int pid,A_pid, B_pid,pid2,C_pid,pid3,pid4;

pid  = fork();

if (pid >0 )

{

printf("Parent starting wait \n");

fflush(stdout);

//int t= 5;

//wait(&t);

printf("Parent finished wait \n");

fflush(stdout);

}

else if (pid ==0)

{

printf("Child going to sleep \n");
fflush(stdout);
sleep(5);
printf("Child finished sleeping \n");
fflush(stdout);
}
else{
printf("CANT FORK ! \n");
//fflush(stdout);
}
}
```

Test Case 1 and Test case 2 are same.

Solution 9

The better Cost per performance ration will be achieved when we will use it in a **parallel processing** fashion. Let's assume the cost per 1 MIPS performance is $1 and cost per 2 MIPS performance is $3 (as per graph)

We also assume that we are in the optimal utilization region of the technology curve.

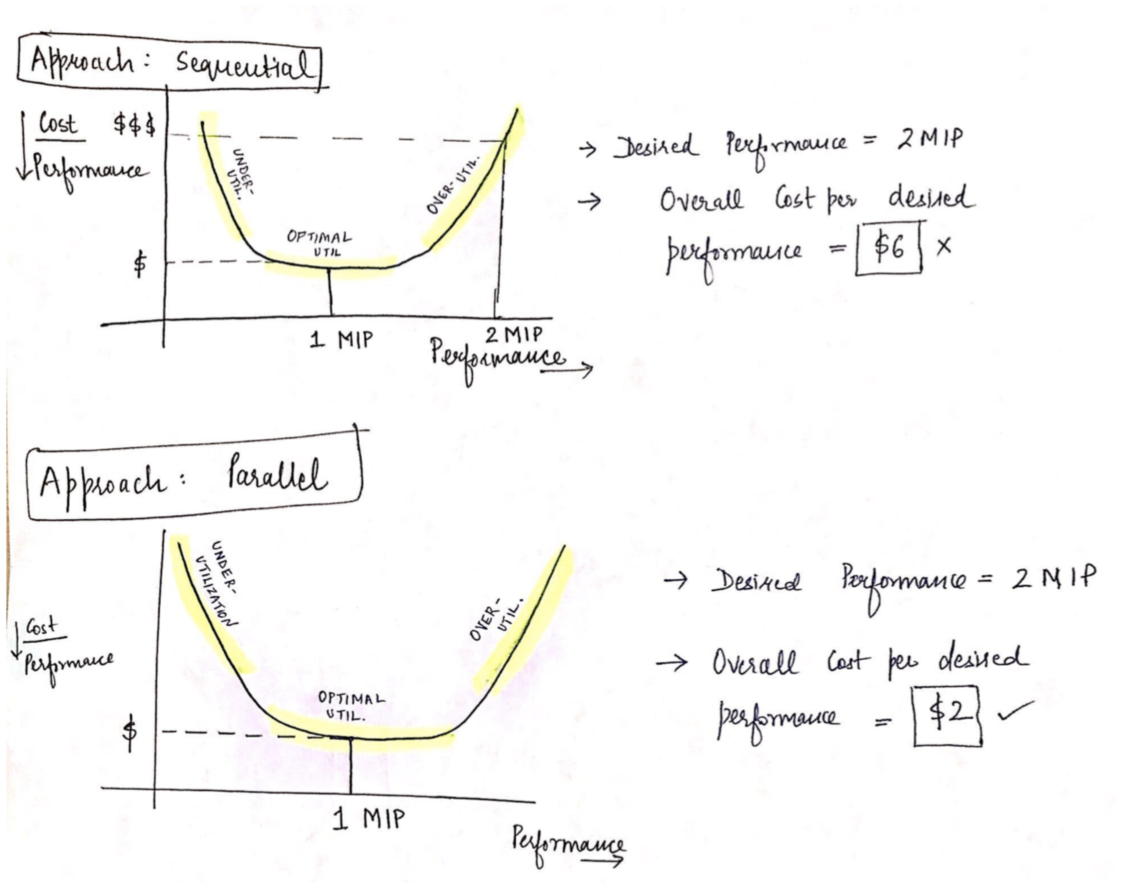
If we choose sequential approach, we will end up with overall $6.

If we parallel processing approach, we will use each processor of 1 MPS each in a parallel fashion, then the overall cost will be $2. (1 MIP and 1MIP in parallel).

Parallel approach is way cheaper per performance ratio as compared to the sequential approach.

If we use Parallel approach, we will save $4.

Below Diagram for better explanation :



Solution 10

There are majorly 2 limitations to ILP parallel processing:

1. Decrease marginal rates as more Functional Units are added which leads to OOO (Out of Order) Execution of instructions.
2. High degree of lookahead needs to utilize large number of Functional Units ; which leads to occurrence of different types of **hazards** ; most commonly **Data Hazard**. Hence, their management becomes an overhead for the operating system.

<u>How it happens :</u>

let's assume we have **2 adders** and we have a degree of single instruction lookahead.

For example:

R1 = A + B................. S1

A = A + Z................. S2

Adder 1: R1= A + B

Adder 2: ** Cant be used **

Reason : Since S2 has a dependency on S1 we cant use the second adder due to detection of Data Hazard.

Therefore scheduling and detection goes hand in hand which effects the overall efficacy.

3. Hazards Issue

Having an aggressive lookahead creates data , control or Resource hazard, which stalls the pipelines and effects the overall performance.

Solution 11

Hyperthreading :

It is a process introduced by Intel to speedup context switching using an extra Hardware i.e. using an extra register. It is observed that by introducing an extra Hardware , it incurs an extra cost of 5% with speed gains of 15%. The switching between two threads without any delay is achieved via hyperthreading. When a single thread is doing an I/O operation, the system can switch to the other thread.

Hyperthreading is transparent to Operating System and software and application level.

To gain peak performance by programmers :

The number of active threads in the program should be equal to the number of logical processors in the system; and the program should be capable of doing multi-threading.

For example : If we have a dual core with each core having two logical processors, and we have two threads T1 and T2, the OS should map T1 to Core 1 and T2 to Core 2 to balance the workload.

Solution 12

At Micro - level, the speedup by Hardware got hindered by the Frequency vs Power Heat**(Heat & Power Wall**) issue where Power is directly proportional to third power of heat. Hence the **maximum clock frequency** it could reach was 3.7 MHz. Hence there was a shift from Single Core to Multi-core to cope up the heating problem towards software side.

Moreover, For Speedup , **Instruction level Parallelism** have been used at the hardware level but it has reached its limit too due to increased complexity and overhead.

At Macroscopic view, The usage of computers follows below path :

personal computer —> cluster computer —>Grid Computer —> Cloud Computer.

<u>Challenges associated with directly exposing the underlying parallelism to the software layer and programmers :</u>

The movement of Single Core to Dual core introduces parallelism which requires a new framework of programming language to adjust this change. The programmers need to have a thorough understanding of hardware to cope with the memory and processor working together. Therefore multiple challenges arise which includes design & re-engineering , profiling, debugging , scaling and testing. As exposing parallelism to programers adds temporal dimension to problem, its hard to think for a human to take into account all the possible combination of cases which arises due to parallelism.

Solution 13

Flynn Architecture Classification is followed as per Instruction and Data being Single or Multiple:

1. **Single** Instruction **Single** Data (SISD)
2. **Single** Instruction **Multiple** Data (SIMD)
3. **Multiple** Instruction **Single** Data (MISD)
4. **Multiple** Instruction **Multiple** Data (MIMD)

1. **Single** Instruction **Single** Data (SISD)

One instruction is applied to one piece of data in a sequential fashion.

Control Unit → Arithmetic Unit. **[1 cycle]**

if there is a looping statement for 20 iterations, there will be 20 cycles of execution.

2. **Single** Instruction **Multiple** Data (SIMD)

Here we have a single instruction executing on multiple data; since we have multiple processing units. At a particular instance, there will be 'n' processing units working on multiple data. Its faster than SISD.

——— Processing Unit 1 ————

——— Processing Unit 2 ————

CPU ——— Processing Unit 3 ———— **Interconnected**

——— Processing Unit 4 ————

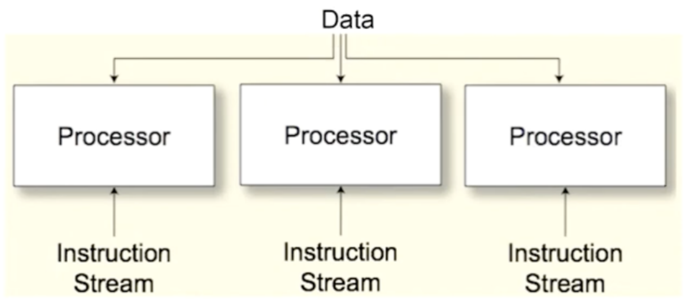
——— Processing Unit n ————

3. **Multiple** Instruction **Single** Data (MISD)

In this classification, we have multiple instruction on a single Data (M,N). If we have three processors, Processor 1 will do Operation-1 on (M,N) ,Processor-2 will do Operation-2 and Processor 3 will do Operation-3.

Traditionally, this is considered **non-sensible** as same data inputted is resulting different results. There was no usage at that time for this configuration since the processors were limited and wasting of cycles could not be afforded. But now, there is a scope of speculative computation where this configuration will be useful such as video gaming where the map rendering happens.

Moreover, in machine learning too, where we need to split the data using cross validation, this approach is quite useful.



4. **Multiple** Instruction **Multiple** Data (MISD)

Many different sets of instructions executes on different sets of data. This configuration is very useful as each processor works on a different problem/subproblem. This architecture provides huge scope for parallel processing and opens areas of shared memory and distributed memory.

Solution 14 (a)

Lets assume, we have 4 stage per pipeline :

Stage 1 - 20 ms

Stage 2 - 25 ms

Stage 3 - 20 ms

Stage 4 - 20 ms

If we go in a sequential fashion , we will finish off our instruction in (20+25+20+20) = 85 ms

If we have N instructions, total time = 85 x N

if we go in pipelined fashion, we will first assign the maximum stage time to each stage and then at every iteration, we will get out output reedy.

Stage 1 - 25 ms

Stage 2 - 25 ms

Stage 3 - 25 ms

Stage 4 - 25 ms

interstage delay per stage : 5 ms

Total Time for one instruction : 120

But After 120+25(N-1),we will have out instruction getting executed.

Although, our per stage time is increased but the overall throughput increases.

Solution 14 (b)

The performance of pipeline is directly proportional to number of pipelines stages(K) , the latch overhead and total number of instructions (N).

Throughput(pipeline)= Throughput(non-pipeline) x Number_of_pipe_stages

Pipeline performance is also limited by instruction stride.

Extreme case 1 , High Latch overhead : when K is too large. Then Latch overhead >work done per stage.

Extreme case 2 , when N is too small. Then we can't assume that N >>K ; hence the throughout of the pipeline gets effected.

Example : When N =5 and K = 8

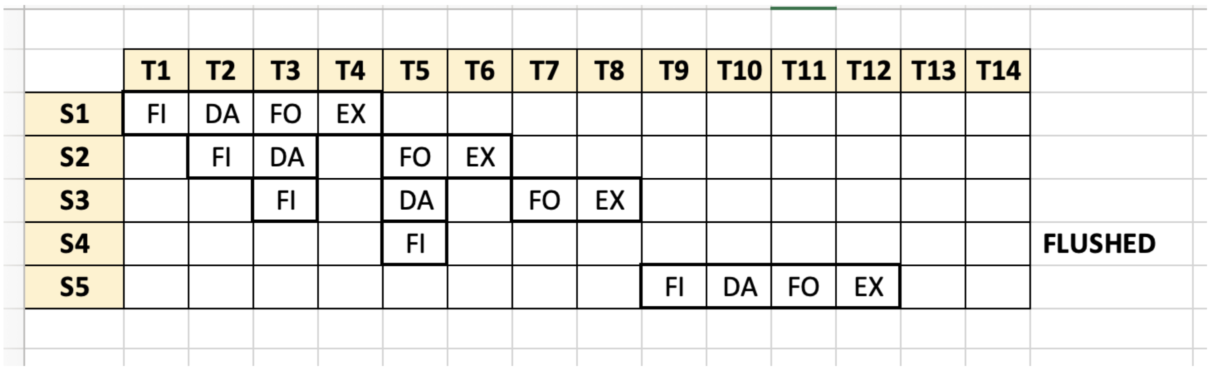
Then as per formula =(N x K) / (K+N-1)

= (4 x 8) /(8+4-1)

= 2.90

Ideally, it should have been 8 times since K =8.

Solution 15



	T1	T2	T3	T4	T5	T6	T7	T8	T9	T10	T11	T12	T13	T14	
S1	FI	DA	FO	EX											
S2		FI	DA		FO	EX									
S3			FI		DA		FO	EX							
S4					FI										**FLUSHED**
S5									FI	DA	FO	EX			

Solution 16

16 (a)

Time	1	2	3	4	5
Adder 1	A = B + C		Z = H + G		
Adder 2	D = E + F				
Multiplier		G = A * Y	W = I * J	F = W * Z	H = K * L
Hazard :	S3: A (data hazard)	S4:G (data hazard)	S6:W,Z (data hazard), Multiplier (Resource hazard)	S7: Multiplier (Resource hazard)	

16 (b)

Time	1	2	3	4	5
Adder 1	A = B + C		Z = H + G		
Adder 2	D = E + F				
Multiplier 1		G = A * Y	W = I * J	F = W * Z	
Multiplier 2				H = K * L	
Hazard :	S3: A (data hazard)	S4:G (data hazard)	S6:W,Z (data hazard)		

16 (c)

Time	1	2	3	4	5
Adder 1	A = B + C		Z = H + G		
Adder 2	D = E + F				
Adder 3					
Adder 4					
Multiplier 1		G = A * Y	W = I * J	F = W * Z	
Multiplier 2				H = K * L	
Multiplier 3					
Multiplier 4					
Hazard :	S3: A (data hazard)	S4:G (data hazard)	S6:W,Z (data hazard)		

Solution 17

non- interleaved

	0.2	0.4	0.6	0.8	1	1.2	1.4	1.6	1.8	2	2.2	2.4	2.6	2.8	3	3.2	3.4	3.6	3.8	4
CPU Prepare	A				B				C				D							
Memory		A-access		A-wait		B-access		B-wait		C-access		C-wait		D-access		D-wait				
CPU Handle				A				B				C				D				

Total Time	3.2
CPU Utilization	50

2 way interleaved

	0.2	0.4	0.6	0.8	1	1.2	1.4	1.6	1.8	2	2.2	2.4	2.6	2.8	3	3.2	3.4	3.6	3.8	4
CPU Prepare	A	B				C	D													
Memory 1		A-access		A-wait			C-access		C-wait											
Memory 2			B-access		B-wait			D-access		D-wait										
CPU Handle				A	B				C	D										

Total Time	2
CPU Utilization	80%
Speedup	1.6

4 way interleaved

	0.2	0.4	0.6	0.8	1	1.2	1.4	1.6	1.8	2	2.2	2.4	2.6	2.8	3	3.2	3.4	3.6	3.8	4
CPU Prepare	A	B	C	D																
Memory 1		A-access		A-wait																
Memory 2			B-access		B-wait															
Memory 3				C-access		C-wait														
Memory 4					D-access		D-wait													
CPU Handle					A	B	C	D												

Total Time	1.6
CPU Utilization	100%
Speedup	2