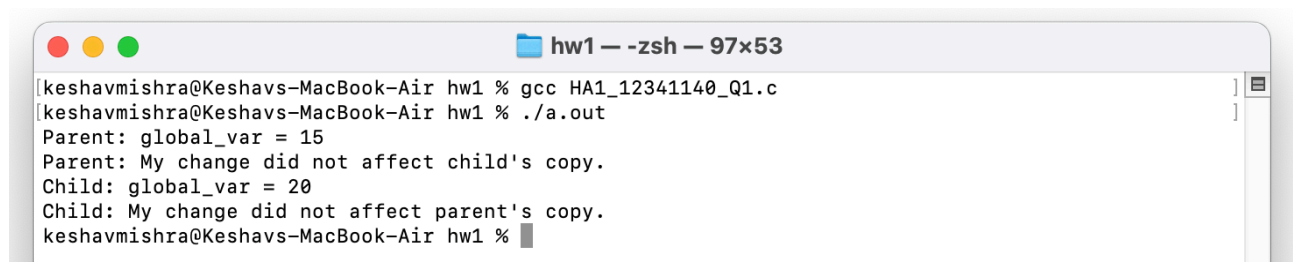


Question 1:



```
hw1 — -zsh — 97x53
keshavmishra@Keshavs-MacBook-Air hw1 % gcc HA1_12341140_Q1.c
keshavmishra@Keshavs-MacBook-Air hw1 % ./a.out
Parent: global_var = 15
Parent: My change did not affect child's copy.
Child: global_var = 20
Child: My change did not affect parent's copy.
keshavmishra@Keshavs-MacBook-Air hw1 %
```

Screenshot of output of HA1_12341140_Q1.c

Explanation for Question 1:

When `fork()` is called, the operating system creates a child process that is an almost exact copy of the parent process. However, memory is not physically duplicated immediately. The OS uses copy-on-write. Initially, both processes share the same physical memory pages. When either process modifies a variable, the OS makes a separate copy of that page for that process.

Because of this, the parent's change to `global_var` (adding 5) and the child's change to `global_var` (adding 10) occur in separate memory copies. These changes do not affect each other.

Memory Segments after `fork()`:

Code segment: Shared between parent and child (read-only).

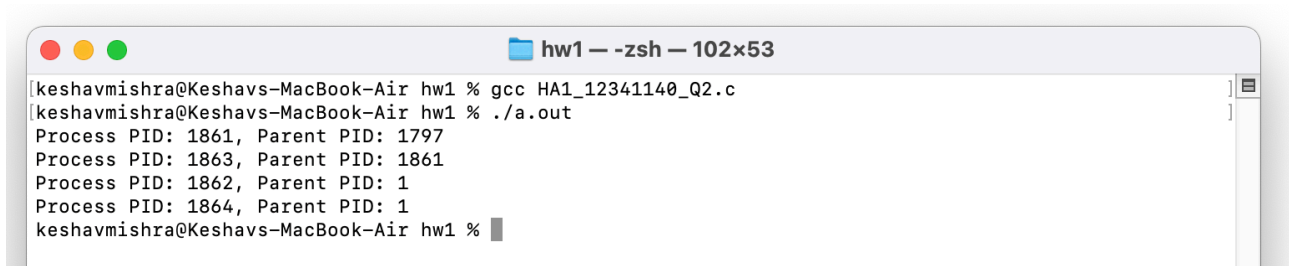
Data segment (globals/statics): Logically copied; copy-on-write ensures independence.

Stack: Duplicated so local variables are separate.

Conclusion:

Changes to global variables in one process do not affect the other process after `fork()` because each process maintains its own independent copy of the variable in memory.

Question 2:



```
hw1 - zsh - 102x53
[keshavmishra@Keshavs-MacBook-Air hw1 % gcc HA1_12341140_Q2.c
[keshavmishra@Keshavs-MacBook-Air hw1 % ./a.out
Process PID: 1861, Parent PID: 1797
Process PID: 1863, Parent PID: 1861
Process PID: 1862, Parent PID: 1
Process PID: 1864, Parent PID: 1
keshavmishra@Keshavs-MacBook-Air hw1 %
```

Screenshot of output of HA1_12341140_Q2.c

Explanation for Question 2:

Initially, there is only 1 process.

First fork(): Creates 1 new process, so now 2 processes exist.

Second fork(): Both of these processes execute the second fork(), each creating a new child. This doubles the total to 4 processes.

Process Tree Structure:

Original Parent

- — — — Child 1 (from first fork)
- — — — — — — — Grandchild 1 (from second fork)
- — — — Child 2 (from second fork by original parent)

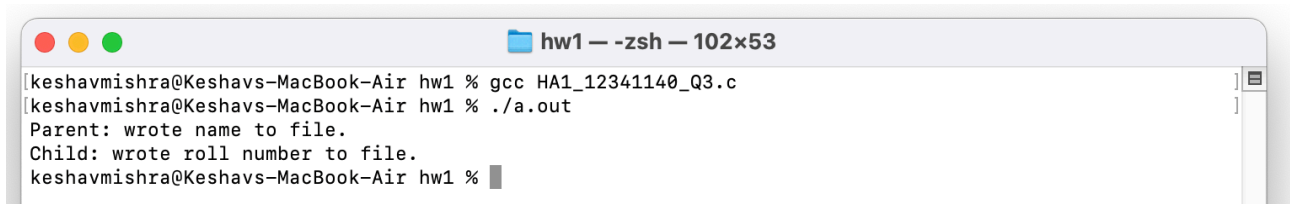
The exact parent-child mapping for the second fork depends on execution order, but the end result is always 4 processes.

After n successive fork() calls, if each process continues execution and calls the next fork(), the total number of processes is $= 2^n$

For $n = 2$:

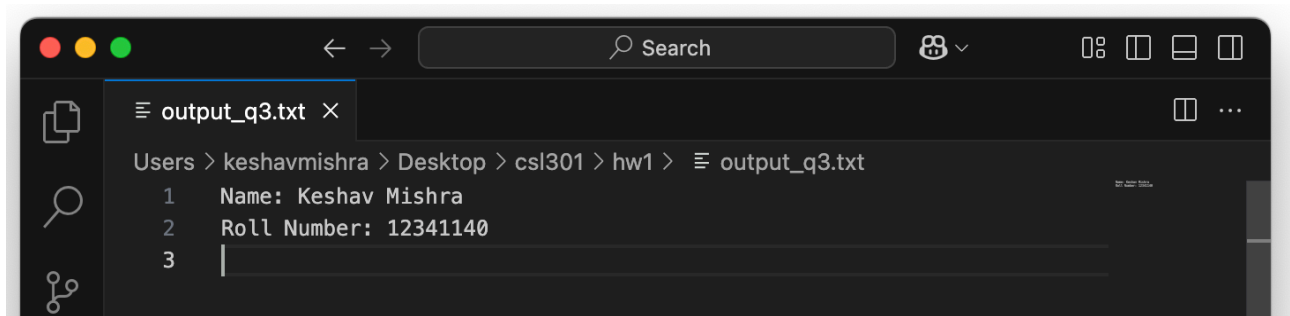
$$2^n = 2^2 = 4 \text{ processes}$$

Question 3:



```
hw1 — zsh — 102x53
[keshavmishra@Keshavs-MacBook-Air hw1 % gcc HA1_12341140_Q3.c
[keshavmishra@Keshavs-MacBook-Air hw1 % ./a.out
Parent: wrote name to file.
Child: wrote roll number to file.
keshavmishra@Keshavs-MacBook-Air hw1 %
```

Screenshot of output of HA1_12341140_Q3.c



```
output_q3.txt x
Users > keshavmishra > Desktop > csl301 > hw1 > output_q3.txt
1 Name: Keshav Mishra
2 Roll Number: 12341140
3 |
```

Screenshot of the output_q3.txt file made by HA1_12341140_Q3.c

Explanation for Question 3:

When a process opens a file, the OS assigns it a file descriptor, which refers to an entry in the system wide open file table. This entry contains the current file offset and file status flags.

When `fork()` is called:

The child process inherits a copy of the parent's file descriptor table.

Both parent and child file descriptors point to the same open file table entry.

Therefore, they share the same file offset.

When both processes write to the file:

Writes are made at the shared file offset, and the offset advances after each write.

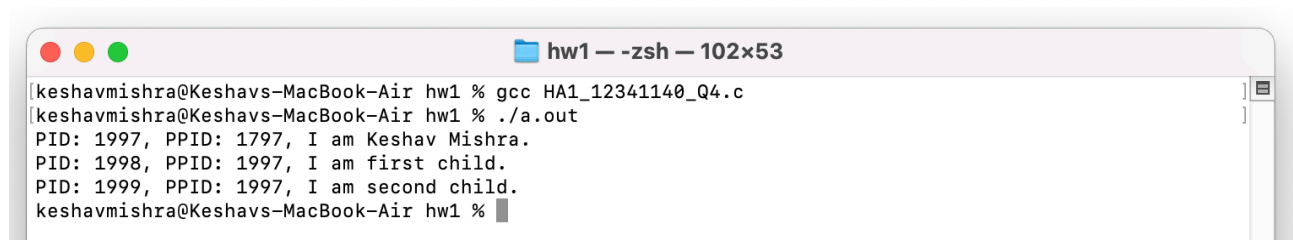
If the writes are small and atomic (like in my code), the output will contain both messages without overlap, but the order depends on process scheduling.

Larger or non-atomic writes can result in interleaving of bytes.

Conclusion:

After `fork()`, parent and child share the same open file description, meaning writes affect the same file offset. This leads to a combined output file containing both processes' messages, though the order may vary.

Question 4:



```
keshavmishra@Keshavs-MacBook-Air hw1 % gcc HA1_12341140_Q4.c
keshavmishra@Keshavs-MacBook-Air hw1 % ./a.out
PID: 1997, PPID: 1797, I am Keshav Mishra.
PID: 1998, PPID: 1997, I am first child.
PID: 1999, PPID: 1997, I am second child.
keshavmishra@Keshavs-MacBook-Air hw1 %
```

Screenshot of output of HA1_12341140_Q4.c

Explanation for Question 4:

When the program starts, there is only 1 parent process.

First fork(): Creates the first child. Now there are 2 processes — parent and first child.

The first child executes its own printf() and does not execute the second fork.

Second fork(): Executed only by the original parent, creating the second child. Now there are 3 processes in total:

Parent (original)

First child (from first fork)

Second child (from second fork)

Process hierarchy:

Parent (Keshav Mishra)

— — — — First Child

— — — — Second Child

PID relationships in output:

getpid() returns the current process's PID.

getppid() returns the PID of the process's parent.

Both children show the parent's PID as the original parent's PID.

The parent's parent PID (getppid()) is the shell or terminal process.

Conclusion:

Sequential forking creates one child at a time under the same parent, resulting in a flat process tree with all children at the same hierarchy level. The total number of processes is 1 (parent) + number_of_children.