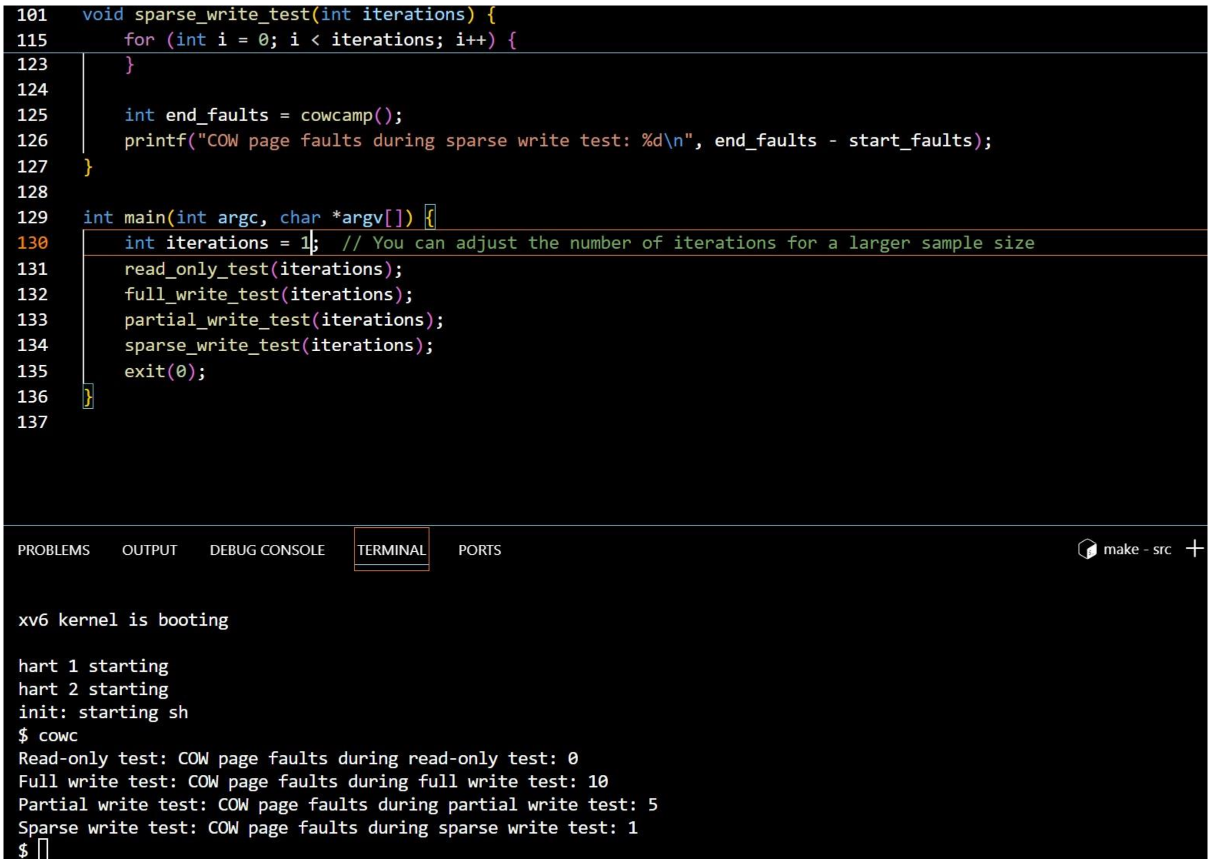
# Copy-on-Write fork Report

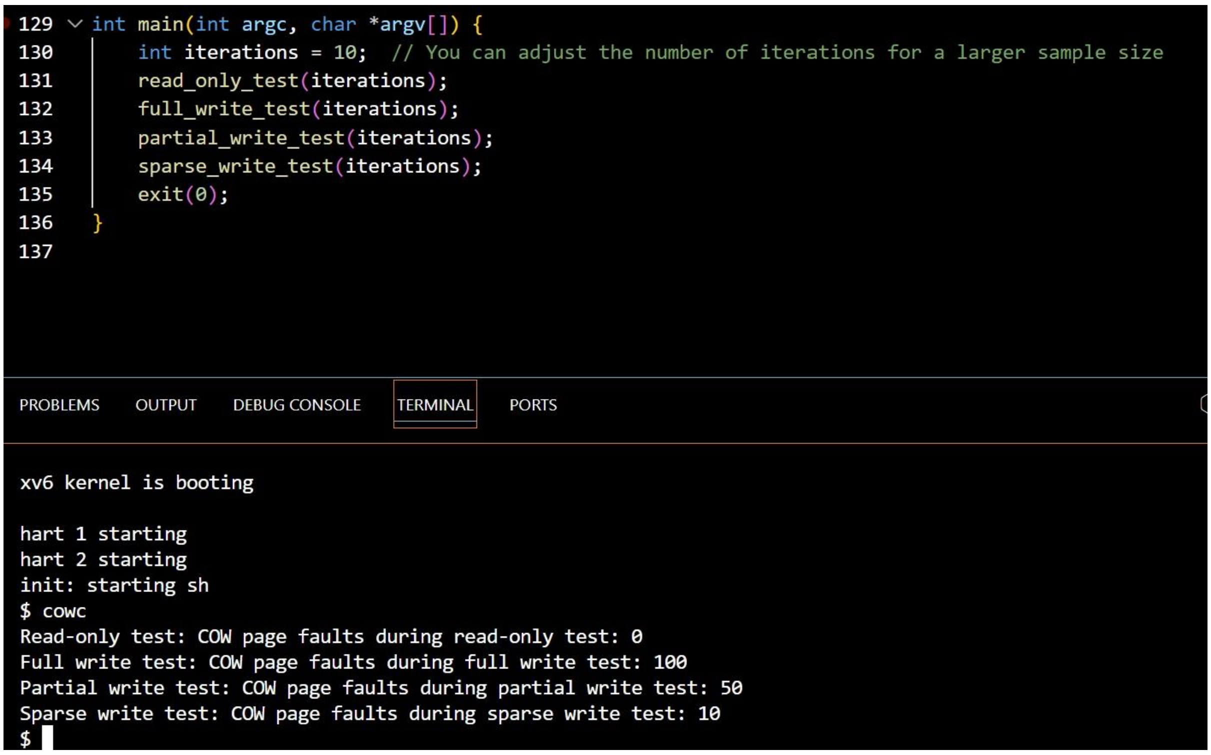
## Page Fault Frequency:

To do so, I made a cowc user program and a cowcamp system call. The system call returns the overall pagefaults that have occurred since the system has booted up which can be used to find pagefaults over a processes's execution by finding this count before and after the process's execution time. The user program does this, with 4 kinds of test, read-test,fullwrite-test,halfwritetest and sparsewrite test all of which have 10 pages allocated to each child, and each of which modify none,all,half and 1 page respectively. Upon running this we can see conclusively that processes which read only cause no pagefaults, and each process which runs and modifies a page triggers as many pagefaults as the number of pages the process modifies.{Note: here iteration refers to the number of child processes that are created by the function (via fork)}

{also note that if the above hypothesis were true we would see number of pagefaults per test as 0*iterations,10*iterations,5*iterations and 1*iterations respectively which we do across iterations upto 100 and farther, evidence for 1,10 and 100 is attached here}

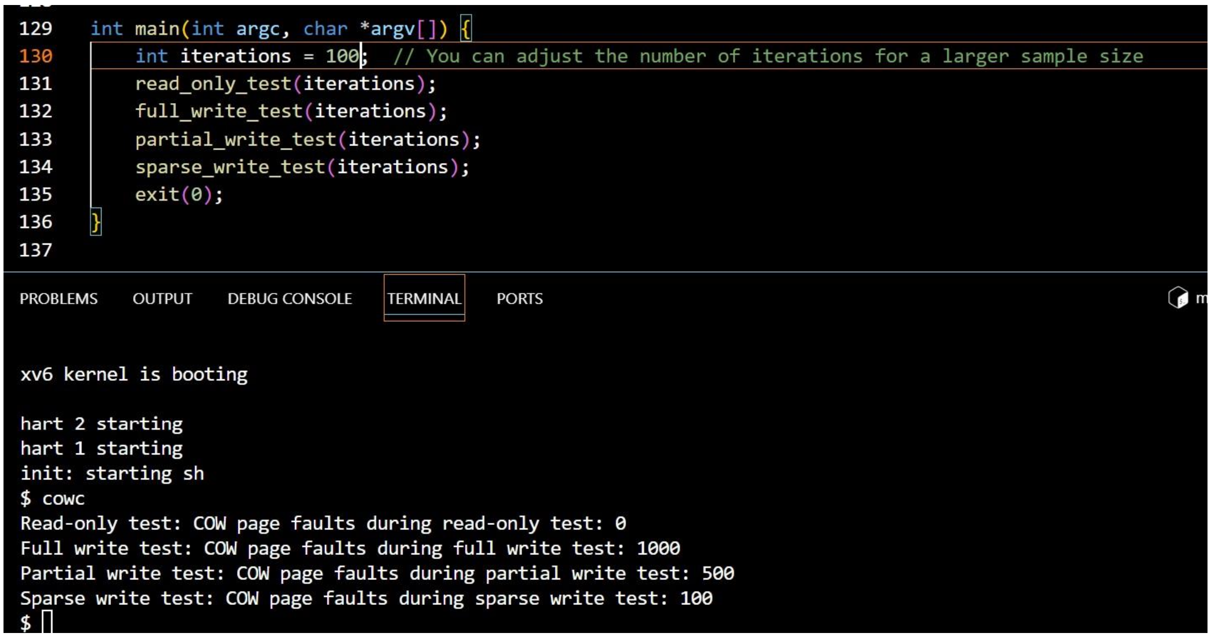

```c
void sparse_write_test(int iterations) {
    for (int i = 0; i < iterations; i++) {
    }

    int end_faults = cowcamp();
    printf("COW page faults during sparse write test: %d\n", end_faults - start_faults);
}

int main(int argc, char *argv[]) {
    int iterations = 1;  // You can adjust the number of iterations for a larger sample size
    read_only_test(iterations);
    full_write_test(iterations);
    partial_write_test(iterations);
    sparse_write_test(iterations);
    exit(0);
}
```

```
xv6 kernel is booting

hart 1 starting
hart 2 starting
init: starting sh
$ cowc
Read-only test: COW page faults during read-only test: 0
Full write test: COW page faults during full write test: 10
Partial write test: COW page faults during partial write test: 5
Sparse write test: COW page faults during sparse write test: 1
$ 
```



```c
int main(int argc, char *argv[]) {
    int iterations = 10;  // You can adjust the number of iterations for a larger sample size
    read_only_test(iterations);
    full_write_test(iterations);
    partial_write_test(iterations);
    sparse_write_test(iterations);
    exit(0);
}
```

```
xv6 kernel is booting

hart 1 starting
hart 2 starting
init: starting sh
$ cowc
Read-only test: COW page faults during read-only test: 0
Full write test: COW page faults during full write test: 100
Partial write test: COW page faults during partial write test: 50
Sparse write test: COW page faults during sparse write test: 10
$
```



```c
int main(int argc, char *argv[]) {
    int iterations = 100;  // You can adjust the number of iterations for a larger sample size
    read_only_test(iterations);
    full_write_test(iterations);
    partial_write_test(iterations);
    sparse_write_test(iterations);
    exit(0);
}
```

```
xv6 kernel is booting

hart 2 starting
hart 1 starting
init: starting sh
$ cowc
Read-only test: COW page faults during read-only test: 0
Full write test: COW page faults during full write test: 1000
Partial write test: COW page faults during partial write test: 500
Sparse write test: COW page faults during sparse write test: 100
$
```

## Brief Analysis:

CoW conserves memory very obviously by only duplicating when there is a specific need for it. Instead of duplicating all pages of the parent immediately upon a fork, which is pretty memory inefficient especially for processes which never actually modify said memory thus never actually needing the extra memory to begin with. CoW attacks this specifically by only allocating the memory when a modification is made, thus reducing the overhead of duplicating upfront, both memory-wise and CPU-wise thus causing a substantial improvement in performance and memory overhead.

CoW is thus very efficient, but there may be a way to take this idea further. We can apply the same line of reasoning to deduce that copying a whole page for processes which only modify locations of memory significantly smaller than a page could again be considered as overhead that can be attacked. Thus a possible optimisation is to apply a similar idea to memory smaller than pages, something like a sub page CoW fork. This could help further streamline memory usage and reduce CPU overhead, provided size of memory block chosen as standard for copying on write is chosen appropriately. Another more experimental approach is to apply some predictive reasoning/analysis of processes in real-time could be used to streamline further, by detecting processes with high frequency of writes and then using spatial and temporal locality to pre allocate pages, allowing us to potentially reduce the overhead of handling various pagefaults in quick succession, which could be helpful.