sfork: A fork with flexible shared memory regions

Chinmay Goyal(180206), Prashant Kumar(180801), Rishik Jain(18807615)

1 Objective

Implement a variant of the fork syscall where some part of the memory of the parent can be shared with the child process in a manner such that the update to that memory region from the parent is visible to the child and vice-a-versa. This sharing can be either unidirectional, i.e., only the parent can update, and the child can read, and vice-versa, or bidirectional, i.e., both the parent and child can update and see the updated values. This sharing can also be achieved with the help of the shared memory construct. Hence, compare sfork with shared memory to study the effectiveness .

2 Implementation

Having shared memory is a method of inter process communication for processes. These processes can be parent/ child or can be completely unrelated. Method to allocate shared memory regions already exists, in this project we try to implement the shared memory regions through various implementations and try to compare the efficiency of each implementation.

2.1 Shared Memory Segments (SHM)

- Shared Memory segments are temporary files which are a part of tempfs of the system. There is a process which is the owner or the creator of this shared segment. Rest of the processes sharing this segment are the clients.
- Each shared memory segment can be accessed by any number of processes which have the ID corresponding to it.
- After allocating the shared memory segment each process needs to attach this temporary file to its virtual address space.
- Shared memory segments are inherited across fork which implies that if a shared memory segment is allocated by a parent process and then a fork is called then the parent and the child can easily communicate through this segment.

2.2 Sfork

We implement sfork as a library function call and not a system call. Over the entire course of our implementation we made some changes to the kernel code and mostly used the existing system calls to achieve the desired results. The structure of the function call **sfork** is as follows:

```
int sfork(size_t len, unsigned int flags, void **addr);
```

- *len* represents the size of the shared memory segment.
- *flags* represent the direction of communication needed in the shared memory segment. Available values for *flags* are:
    1. CHILD_WRITE: Flag to allow write access to the child process
    2. PARENT_WRITE: Flag to allow write access to the parent process
    3. SFORK_POPULATE: Allocate the shared memory region with the mmap flag MAP_POPULATE
- *addr* is a pointer to a pointer which will contain the virtual address of the shared memory segment
- The function returns the PID of the child process to the parent and 0 to the child process.

2.2.1 Kernel Code Changes

Some of the changes that we made to kernel code include:

- We implemented a flag VM_SFORK in *vm_area_struct* which takes the value 0x8000000000 (40th bit of sfork)
- We implemented a flag MAP_SFORK in mmap syscall which takes the value 0x200
- *shared_area_struct*: This data structure shared the information about the shared memory segment and is present in memory. We have a pointer to *shared_area_struct* from *vm_area_struct*.

    ```
    struct shared_area_struct {
        atomic_t ref_count;
        spinlock_t sl;
        struct vma_node* head;
    };

    struct vma_node{
        struct vm_area_struct* vma;
        struct vma_node* next;
    };
    ```

- Other changes were made at appropriate places to support the functionality of these data structures.

2.2.2 File Backed Mapping

- We create a temporary file called ”temp” with the *open* syscall by passing O_CREAT flag.

- We use *ftruncate* call to change the size of the temporary file to the desired size
- We then call *unlink* to remove the directory entry of the file and provide the essence that no new file is being created on calling sfork.
- We call mmap with MAP_SHARED flag and pass the file descriptor of the ”temp” file to the mmap call.
- We call fork and return the desired PIDs to the parent and the child process.

2.2.3 SFORK_POPULATE

- In this implementation we call mmap syscall with MAP_POPULATE flag.
- This method allocated all the shared memory before calling fork.
- This is different than allocating pages after calling fork because in this method the page fault has already happened and we only need to take care of the Copy On Write mechanism when a fault occurs on the child process.
- As can be expected this process slows down the process of mmap when a huge chunk of memory is allocated to the shared region.
- Users can use this implementation by calling sfork and passing the SFORK_POPULATE flag in the *flags* parameter.

2.2.4 Lazy Implementation

- This implementation varies from the SFORK_POPULATE one by the allocation of pages on demand.
- When the first process tries to access a page that has not been allocated, that process will allocate a physical page for itself. Then it will take a lock on the *shared_area_struct* and allocate the same physical page for the same virtual address for all the *vm_area_struct* that are present as linked list in the *shared_area_struct*.
- This process is time intensive when a large number of processes are accessing the share area, because this process tries to take a lock on the common data structure and then allocates the physical page to all the processes one by one.

3 Source Code

We made two github repositories to complete the entire implementation.

- Kernel Source Code: We used kernel version 5.10.89 as the base source code and make the subsequent changes on it. The github repo containing the changes is this.
- We implement the library to access the sfork implementation as another Github repo which can be found here.

4 Results

4.1 System Configuration

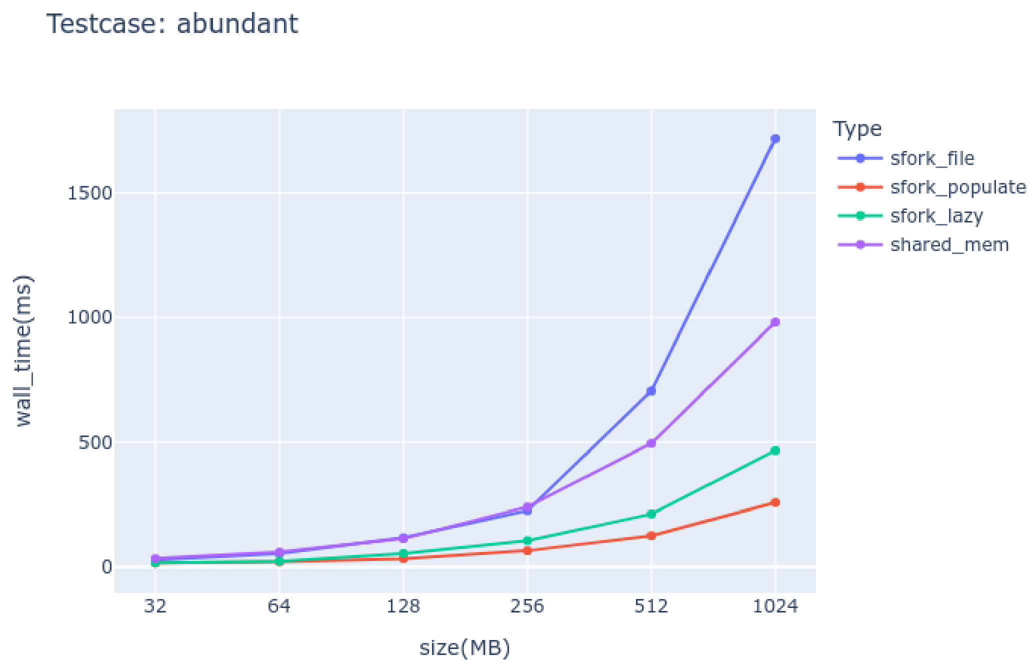
- Operating System: Ubuntu 20.04.4 LTS x86_64
- Kernel Source Code: linux 5.10.89 (modified)
- CPU Qemu Virtual version 2.5+ (8) @2.207GHz
- Memory: 3932 MB

4.2 Comparative Study

While comparing the performances of the different implementations, a certain amount of memory was allocated and then accessed from the corresponding processes.

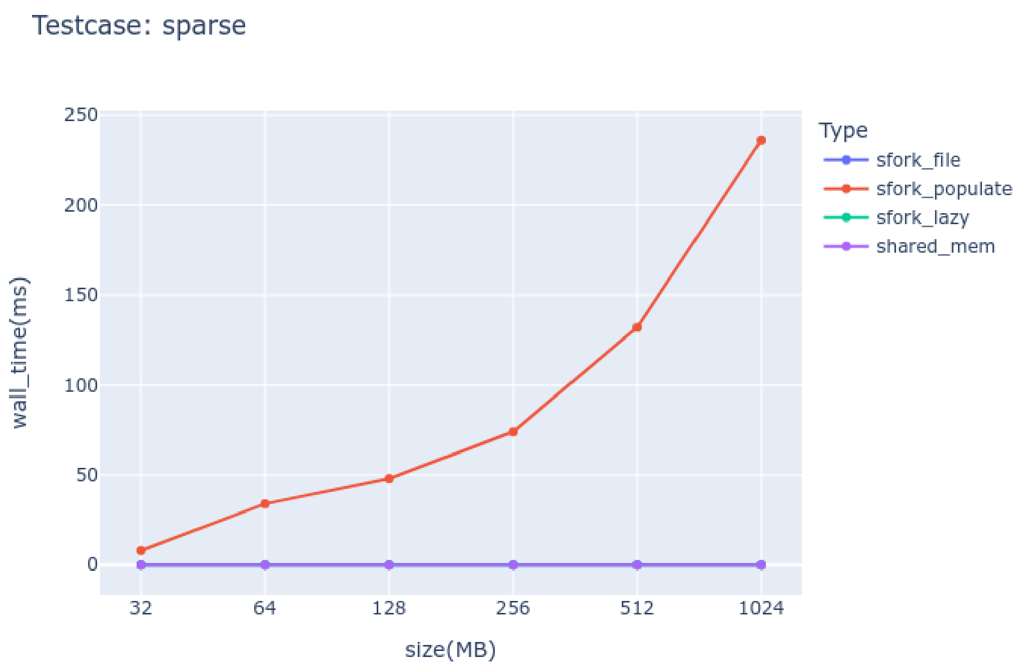
4.2.1 Abundant Access

In this test case, all the pages that have been allocated are accessed by the child and the parent processes. Only 2 processes exist in this case.



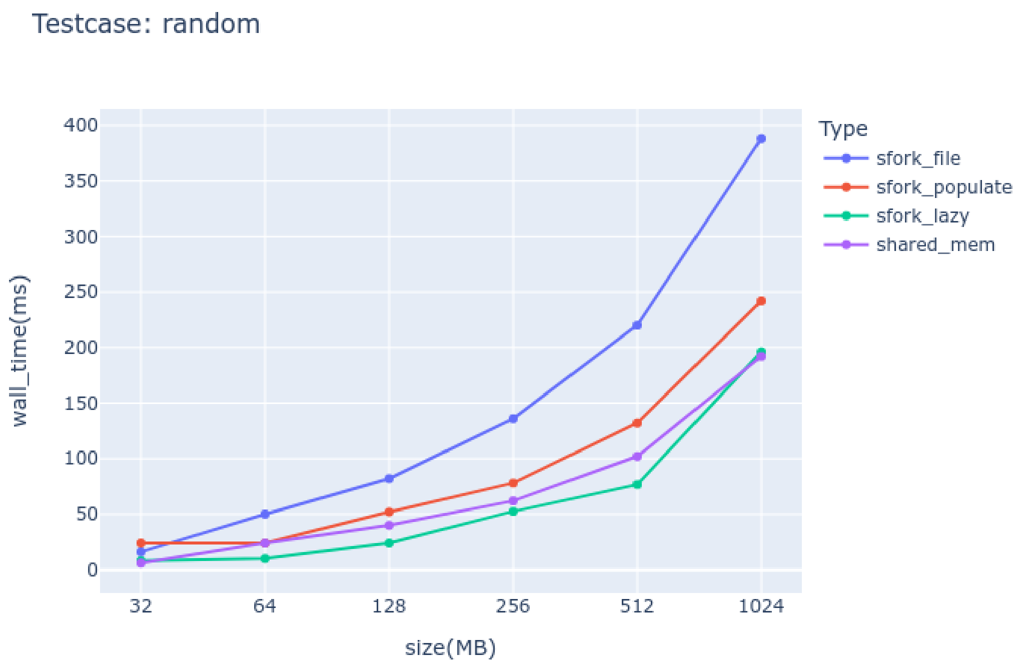
4.2.2 Sparse Access

In this test case, only 6 (3 each for child and parent) pages are accessed by the child and the parent processes. Only 2 processes exist in this case.



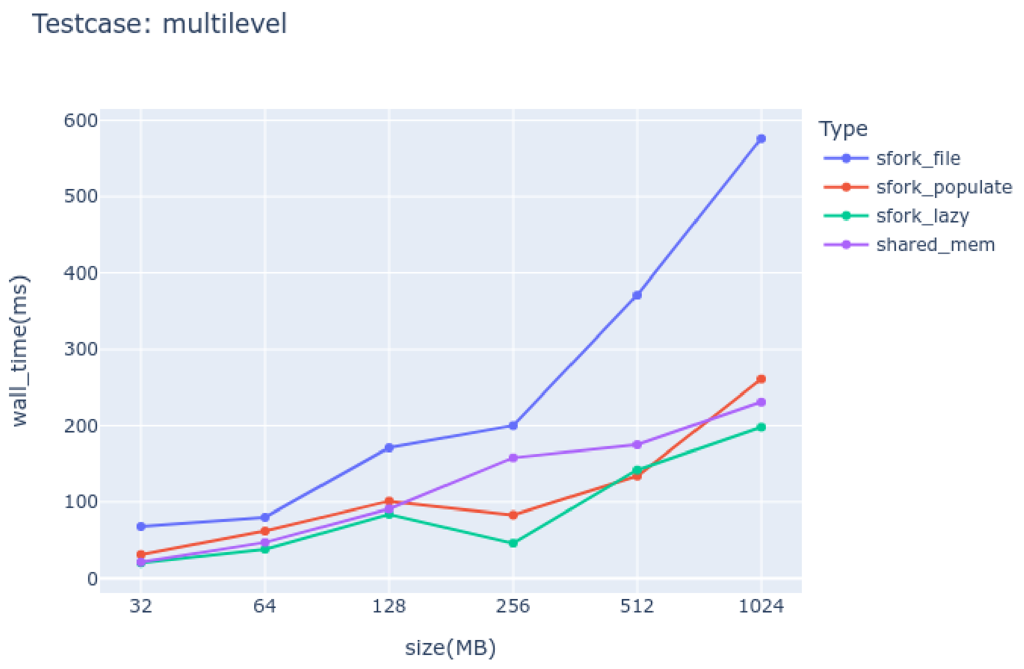
4.2.3 Random Access

In this test case, $\frac{1}{4}^{th}$ pages are randomly accessed by the child and the parent processes. Only 2 processes exist in this case.



4.2.4 Multiprocess Implementation

$\frac{1}{4}^{th}$ pages are randomly accessed by 5 processes. This test case checks the wait time due to the additional lock implemented in shared_area_struct.



4.3 Conclusion

It is evident from the study done above that sfork has a better performance in comparison to shm. Moreover, as expected SFORK_POPULATE performs better when the processes access a major chunk of the memory that they allocate. File backed implementation of sfork is comparatively slower in two test cases, which shows that file backed mappings have some unnecessary overhead which can be avoided by using Lazy implementation and the POPULATE implementation of sfork. As the performance of lazy implementation is poor in comparison to SFORK_POPULATE when major chunk of the memory is accessed, the usage of on-demand allocation of memory can be left to the discretion of the user when calling sfork. The test case of multilevel processes show that the overhead added by spinlock is not large.

5 Future Work

- The project can be extended to include the shared memory segments even after exec call. We tried to implement this but ran into troubles and could not make a working implementation
- Optimizations to the TLB for the shared memory segments. As the physical mapping of the virtual addresses would be same across the processes sharing the memory regions, the TLB does not need to be flushed for these entries.

6 Acknowledgement

- We looked at the kernel code at Elixir Bootlin website.
- We thank Prof. Debadatta Mishra for guiding us through the project.

7 Contributions

- All the group members contributed equally to the project
- The kernel code changes were made on the system owned by Prashant Kumar and the other two members would ssh into the system to make changes and compile the kernel, hence github commits/insights do not reflect the real contribution of all the members.