RV Educational Institutions ®

RV College of Engineering ®

Autonomous Institution
Affiliated to Visvesvaraya
Technological University,
Belagavi

Approved by AICTE,
New Delhi

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

OPERATING SYSTEMS - CS235AI

REPORT

Title of the work	Implementation of Copy-on-Write on Fork() system call in xv6
Submitted to	Dr. Jyoti Shetty Department of Computer Science, RV College of Engineering
Group member list	Sundarakrishnan N (1RV22CS210) Tanmay Umesh (1RV22CS217) Tarun Bhupathi (1RV22CS218)

SYNOPSIS

In xv6, the integration of Copy-On-Write (COW) functionality into the fork() system call signifies a substantial enhancement in memory management and process creation methodologies. This implementation enables efficient sharing of memory between parent and child processes, significantly reducing memory duplication overhead. When a fork() call is made, the child process initially shares memory with the parent through Copy on Write, postponing memory copying until one of the processes attempts to modify the shared memory. This delayed copying mechanism optimises resource utilisation and enhances system performance by minimising unnecessary memory duplication.

By leveraging Copy on Write, xv6 achieves improved memory efficiency and scalability without compromising data integrity or security. The Copy On Write mechanism allows multiple processes to access the same memory pages concurrently, preserving the integrity of the data by deferring memory copying until modification is necessary. This approach ensures that changes made by one process do not affect the memory accessed by other processes until divergence occurs. As a result, xv6 can efficiently manage memory resources, enabling the concurrent execution of multiple processes with minimal overhead and memory footprint.

Overall, the integration of Copy-On-Write functionality into the fork() system call within xv6 underscores its commitment to efficient memory management and process creation. This implementation optimises resource utilisation, enhances system performance, and contributes to the overall stability and scalability of the xv6 operating system. Through Copy on Write, xv6 demonstrates its ability to balance memory efficiency with data integrity, making it a robust and reliable platform for educational and research purposes

TABLE OF CONTENTS:

INTRODUCTION

In operating systems, efficient memory management is paramount for optimising system performance and resource utilisation. However, traditional methods of memory allocation and process creation pose challenges in terms of overhead and redundancy. When a parent process forks to create a child process, a common issue arises: the need to duplicate the parent's memory space for the child, consuming both time and resources. This duplication, while necessary for preserving data integrity, becomes inefficient when the child process immediately modifies its memory, resulting in unnecessary copying and wasted memory.

Enter Copy-On-Write (COW), a technique devised to alleviate the inefficiencies of memory duplication during process creation. COW operates on the principle of deferred copying, postponing the actual duplication of memory pages until a write operation is attempted on either the parent or child process's memory. Rather than immediately duplicating memory pages upon fork, COW allows both processes to share the same memory pages initially. When either process attempts to modify a shared page, only then is a copy created, ensuring data integrity while minimising unnecessary copying and resource consumption.

By implementing Copy-On-Write, operating systems can significantly improve memory utilisation and process creation efficiency. The deferred copying mechanism not only conserves system resources by eliminating redundant memory duplication but also enhances system performance by reducing memory overhead and minimising disk I/O operations. Moreover, COW facilitates seamless process creation and memory sharing, enabling faster startup times and smoother execution of applications. Overall, Copy-On-Write stands as a testament to the ingenuity of memory management techniques, offering a pragmatic solution to the challenges posed by traditional memory duplication methods in operating system design.

LITERATURE SURVEY

AUTHORS	PAPER TITLE	INFERENCE
Marten van Steen	Understanding xv6	• It provides a practical and hands-on approach to learning about operating system concepts. • Relevance of COW Fork Discussion: Since the paper is expected to discuss COW fork, it would be important for the author to provide clear explanations of how this technique is implemented in xv6 and its significance in terms of performance and memory management. • The paper is structured to provide a comprehensive overview of xv6's kernel structures, system calls, internal mechanisms, and perhaps focuses on the Copy-On-Write (COW) fork technique, which is a significant feature of xv6.
Jonathan M. Smith, Gerald Q. Maguire Jr.	Effect of copy-on-write memory management on the response time of UNIX fork operations	• The study offers a focused analysis on 'copy-on-write' in UNIX fork(), emphasising memory and execution time aspects. • Execution time dependence on memory copied during fork() is a key finding, highlighting a critical factor in 'copy-on-write' performance. • 'Copy-on-write' proves practically efficient, reducing real-time requirements and demonstrating its potential for enhancing overall system performance • 'Copy-on-write' proves effective, reducing real-time requirements for fork() operations, showcasing practical advantages for system efficiency.

SYSTEM ARCHITECTURE

xv6 stands as a cornerstone in operating system education, developed with the purpose of imparting foundational knowledge in system programming and OS principles. Its Unix-like structure provides students with a structured yet accessible environment to delve into the inner workings of operating systems. Featuring a streamlined design, xv6 offers an invaluable platform for comprehending critical OS components like process management, memory allocation, and file systems.

The RISC-V iteration of xv6 signifies a notable adaptation, aligning this educational OS with the RISC-V instruction set architecture (ISA). RISC-V, an open-source ISA hailing from UC Berkeley, embodies simplicity, adaptability, and scalability, making it an ideal fit for both educational and practical contexts. By transitioning xv6 to RISC-V, exposure is gained to a contemporary and versatile ISA while retaining the educational advantages inherent in xv6's simplicity and transparency.

It is worth noting that xv6, regardless of its traditional or RISC-V variant, deliberately maintains a minimalist approach, eschewing many features commonly found in commercial operating systems. However, this deliberate choice serves to enhance the learning journey by allowing students to concentrate on fundamental concepts without the distractions of unnecessary complexity. Through its low-level environment, xv6 fosters exploration and experimentation, nurturing a deeper understanding of operating system fundamentals.

Fork operation prior to Copy on Write implementation:

In the xv6 operating system, the fork() system call is used to create a new process, which is a near-identical copy of the calling process.A detailed explanation of how fork() works in xv6:

1. **Invocation:**
   When a process calls fork(), the kernel receives the system call and begins the process of creating a new process.

2. **Allocation of Process Table Entry:**
   The kernel allocates a new entry in the process table for the new process. This entry will store information about the process, such as its state, program counter, stack pointer, etc.

3. **Copy of Address Space:**
   The kernel creates an exact copy of the address space of the parent process for the child process. This includes the entire memory layout of the parent process, including code, data, heap, and stack segments.

4. **Copying File Descriptors:**
   File descriptors are also copied from the parent process to the child process. This ensures that the child process inherits the same open files as the parent.

5. **Setting Return Values:**
   In the parent process, the fork() system call returns the process ID (pid) of the newly created child process. In the child process, fork() returns 0, indicating that it is the child process.

6. **Adjusting Process State:**
   The kernel sets the state of the child process to "Runnable". Both the parent and child processes are now ready to execute.

7. **Parent and Child Processes Resume Execution:**
   After fork() returns, both the parent and child processes resume execution at the instruction immediately following the fork() call. However, since they are separate processes, they have different process IDs (pid) and may have different return values from fork().

8. **Execution Continues Independently:**
   From this point on, the parent and child processes execute independently of each other. Changes made by one process to its memory or file descriptors do not affect the other process.

9. **Wait for Child Process (Optional):**
   If the parent process wishes to wait for the child process to terminate, it can use the wait() system call. This allows the parent process to synchronise its execution with the termination of the child process.

10. **Termination:**

    When the child process terminates, either by calling exit() explicitly or by reaching the end of its main function, the kernel cleans up the resources associated with the child process and updates its process table entry accordingly.

METHODOLOGY

Xv6's fork system call traditionally duplicates all parent process memory. This inefficiency is addressed by Copy-on-Write, which defers memory duplication until a write attempt occurs. This methodology outlines the key steps for implementing Copy on Write in xv6, focusing on modifications to **vm.c** (virtual memory management), **trap.c** (system trap handling), and defs.h (constant and type definitions). Notably, **defs.h** will be modified to introduce a PTE_COW flag bit within the pte_t structure, enabling Copy-on-Write functionality.

1. **Enabling CoW Functionality (defs.h):**

    - Introduce a new flag bit named PTE_COW within the pte_t structure definition in defs.h.
    - This flag serves as a crucial indicator, signifying whether the corresponding page is currently shared (copy-on-write) or privately owned by a process.

2. **Modifying Memory Mapping (vm.c):**

    - Enhance the **kvmmap** function in vm.c to incorporate Copy on Write which includes identification of specific memory regions (e.g., shared libraries, read-only data segments) during process memory mapping.
    - For these identified regions, explicitly set the newly introduced PTE_COW flag in the corresponding PTEs. This explicitly marks these memory regions as shared between the parent and child processes upon a fork event.

3. **Implementing CoW-Specific Page Fault Handling (trap.c & vm.c)**:

    - Modifying the existing page fault handling mechanism to address Copy on Write violations by following the steps below
        - When a page fault occurs, the page fault handler meticulously examines the faulted PTE in trap.c.
        - If the PTE_COW flag is set within the PTE (identified in trap.c), this signifies an attempt to write to a shared page.
        - Upon detection of a CoW violation in trap.c, trigger the CoW operation residing potentially in vm.c.

4. **Copy on Write Operation Details (vm.c):**

    - Upon detection of a Copy on Write violation, a dedicated Copy on Write operation is initiated (potentially in vm.c) by employing the kalloc function where a new page is allocated from the physical memory pool
    - The contents of the original shared page, identified by the faulted PTE, are meticulously copied to the newly allocated page. This guarantees that both the parent and child processes possess private copies of the modified data

5. **Updating PTEs (vm.c):**

    - Child Process PTE Update: The child process's PTE corresponding to the faulted virtual address is modified to point to the newly allocated page, ensuring the child operates on its private copy.
    - PTE_COW Flag Management: The PTE_COW flag is cleared within the child process's PTE, signifying that the child now owns a private copy of the data.

```
{
  else if(r_scause()== 15){
    if ((*my_pte&PTE_COW)){

      uint flags;
      char *mem;


      flags = PTE_FLAGS(*my_pte);
      flags |= PTE_W;
      flags &= ~PTE_COW;

      if((mem = kalloc()) == 0){

        p->killed=1;
        exit(-1);
      }


      memmove(mem, (char*)pa, PGSIZE);


      *my_pte = PA2PTE(mem) | flags;

      kfree((char*)pa);
      if((*my_pte&PTE_COW)&&reference_counter[pa/PGSIZE]==1)
      {

      *my_pte &= ~PTE_COW;
      *my_pte |= PTE_W;
      }
      p->trapframe->epc = r_sepc();
```

```
      if((pte = walk(old, 1, 0)) == 0)
      if((*pte & PTE_V) == 0)
        panic("uvmcopy: page not present");

      *pte &= ~PTE_W;


      *pte |= PTE_COW;



      pa = PTE2PA(*pte);


      acquire(&reflock);
      reference_counter[pa/PGSIZE]+=1;
      release(&reflock);

      flags = PTE_FLAGS(*pte);
```

The two above images show the modification that was done at the kernel level to implement Copy on Write on fork().

SYSTEM CALLS USED

System calls are fundamental interfaces between user programs and the operating system, enabling crucial operations such as process management, file I/O, and memory management. Some of the system calls that are implemented here are as follows

1. **fork()**
   Creates a new child process by duplicating the current process. The child process inherits the address space of the parent process through Copy-On-Write (COW). It returns different values in parent and child processes: 0 in the child and the child's PID in the parent.

2. **exit()**
   It terminates the current process and returns its resources to the operating system. It also optionally returns an exit status that can be retrieved by the parent process using wait(). Any open file descriptors belonging to the process are closed upon termination.

3. **wait()**
   It suspends execution of the calling process until one of its child processes terminates. It allows the parent process to synchronise its execution with the termination of its child processes. It retrieves the exit status of the terminated child process, enabling the parent to handle it appropriately.

4. **brk()**
   It sets the end of the data segment for the calling process to the specified value. It is used to adjust the amount of memory allocated to the process dynamically. It returns the previous end of the data segment, allowing the process to query or adjust its memory usage.

5. **sbrk()**
   It Increases or decreases the program's data segment by a specified increment. It provides a more flexible alternative to `brk()` by allowing incremental adjustments to the data segment.It returns the starting address of the newly allocated memory region or -1 if the operation fails.

These are the major core system calls that are implemented in xv6. Using these various functionalities are created that are then used to implement features of xv6.

OUTPUT/RESULTS

To check for the correct implementation of Copy on Write we use a cowtest. A "cowtest" within operating systems, such as xv6, serves as a validation method to ensure the accurate implementation and functionality of Copy-On-Write (CoW) memory management. Copy on Write is a memory optimization technique where memory pages are shared between parent and child processes until one attempts to modify the shared memory.

During the cowtest, memory pages are initially allocated to a parent process, and subsequently, a child process is created via forking. Both processes are expected to share the same memory pages. The test involves modifying the memory contents in one process while verifying that the changes do not affect the other. Successful completion of the cowtest indicates the correct implementation of Copy on Write, which offers benefits such as reduced memory consumption, improved performance through delayed memory copying, and enhanced system stability by ensuring data integrity between parent and child processes.

```
xv6 kernel is booting

hart 2 starting
hart 1 starting
init: starting sh
$ cowtest
ok
 COW TEST PASSED
$
```

```c
#include "kernel/types.h"
#include "kernel/stat.h"
#include "kernel/memlayout.h"
#include "user/user.h"

void
cwtest()
{
  uint64 phys_size = PHYSTOP - KERNBASE;
  int sz = phys_size / 4;
  int pid1, pid2;


  char *p = sbrk(sz);
  if(p == (char*)0xffffffffffffffffL){
    printf("sbrk(%d) failed\n", sz);
    exit(-1);
  }

  pid1 = fork();
  if(pid1 < 0){
    printf("fork failed\n");
    exit(-1);
  }
  if(pid1 == 0){
    pid2 = fork();
```

CONCLUSION

In conclusion, the introduction of Copy-on-Write (CoW) presents a demonstrably effective technique for optimising memory management. This mechanism hinges on the principle of delaying memory duplication until a write attempt is detected. Consequently, CoW fosters significant memory usage advantages for read-only data shared between parent and child processes. This approach not only minimises memory consumption during the initial fork event, but also extends these benefits throughout the processes' lifecycles.

The ramifications of CoW extend beyond immediate memory savings. By minimising unnecessary memory duplication, xv6 achieves a demonstrably more scalable and resource-conscious operating system. This enhanced efficiency allows xv6 to effectively manage a larger number of processes within the constraints of available memory resources. This translates to demonstrably improved system performance and resource utilisation, particularly for applications that heavily rely on process creation and the sharing of read-only data.

The implementation of CoW in xv6 serves as a compelling illustration of the benefits derived from leveraging architectural features, such as RISC-V's memory management capabilities, for efficient process creation. This integration contributes to a more robust and scalable xv6 system, ultimately paving the way for the development and execution of a wider range of applications and system functionalities within this operating system.

ACKNOWLEDGEMENT

We would like to extend our sincere gratitude to everyone who helped finish this project on implementation of Copy-on-Write on fork() in xv6.

First and foremost, We would like to thank Dr.Jyoti Shetty, our project guide, for giving us the required direction and assistance throughout the project. Ma'am's insightful comments and recommendations were very helpful in forming this endeavour.

Additionally, We would like to thank our peers and coworkers for their unwavering encouragement and support throughout the project. We were able to make a number of improvements to the undertaking thanks to their comments and suggestions.

We also would like to thank all of the helpful sources and references we used to complete this endeavour.

Finally, We would like to express my heartfelt thanks to our family and friends for their unwavering support and motivation during the project.

Sundarakrishnan N
Tanmay Umesh
Tarun Bhupathi

REFERENCES

1] Jonathan M. Smith and John Ioannidis, **“Implementing remote fork() with checkpoint/restart,”** IEEE Technical Committee on Operating Systems Newsletter,(February, 1989)

[2] **Effects of copy-on-write memory management on the response time of UNIX fork operations** Jonathan M. Smith,Gerald Q. Maguire, Jr.Computer Science Department, Columbia University, New York, NY 10027

[3]Remzi H. Arpaci-Dusseau and Andrea C. Arpaci-Dusseau, **"Operating Systems: Three Easy Pieces"** includes a section on xv6.

[4]**“CCoW: Optimising Copy-on-Write Considering the Spatial Locality in Workloads”**,Minjong Ha,Sang-Hoon Kim,23 December 2021,

[5] **"Copy-on-Write"** by Ousterhout, et al.

[6] **"Copy-on-Write: Principles and Performance"** by D. R. Engler and M. F. Kaashoek.