

# 写时复制 (Copy-on-Write, COW) 简介

写时复制 (Copy-on-Write, 简称 COW) 是一种优化技术, 其核心思想是延迟资源的实际复制。当多个调用者 (如进程或线程) 共享同一资源时, 资源最初设置为只读并共享。当某个调用者尝试写入该资源时, 系统才会为其创建一个独立的副本, 而其他调用者仍然继续访问原始资源。

## 原有 xv6 的进程创建机制

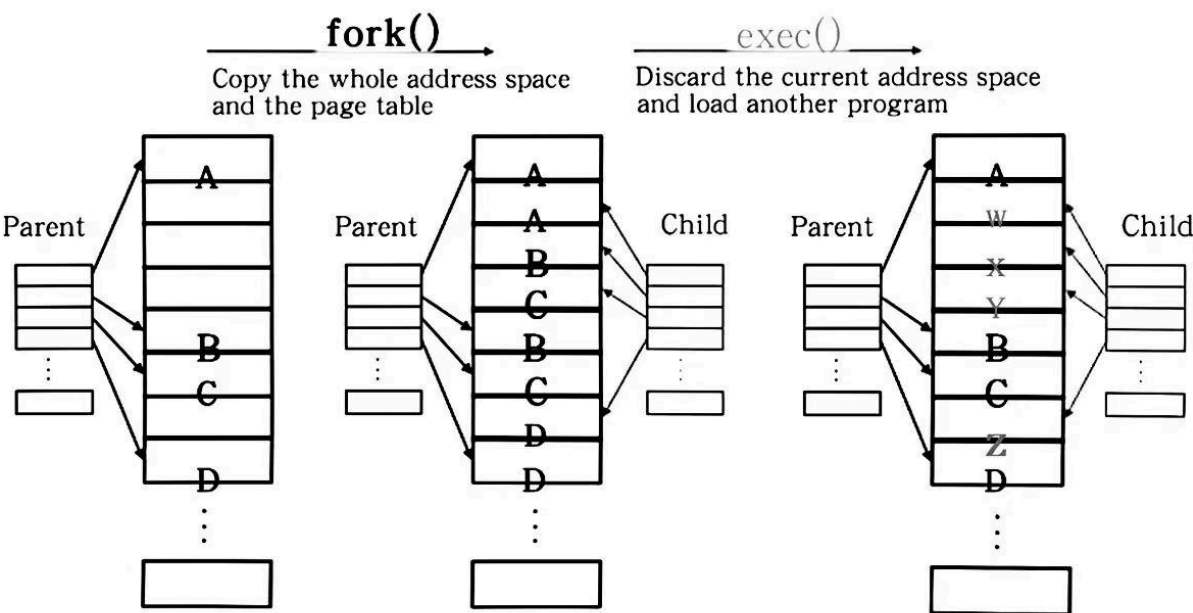
在 xv6 操作系统中, `fork()` 是创建新进程的主要方式。`fork()` 会完整地拷贝父进程的所有内存页面和页表, 如图1所示:

- 父进程与子进程之间的所有资源 (如 A、B、C、D) 都会被复制。
- 无论子进程是否需要这些资源, 都会占用同等的内存和时间。

虽然这种方式保证了父子进程的独立性, 但存在以下缺点:

1. **内存浪费**: 如果子进程立即调用 `exec()` 替换自身, 那么大量的内存拷贝会变得无用。
2. **性能瓶颈**: 在大内存场景下, 拷贝每个页面需要大量时间, 增加了系统开销。
3. **缺乏优化**: 即使父子进程仅有少量数据需要修改, 也会拷贝整个地址空间。

## | Process Creation— `fork()`



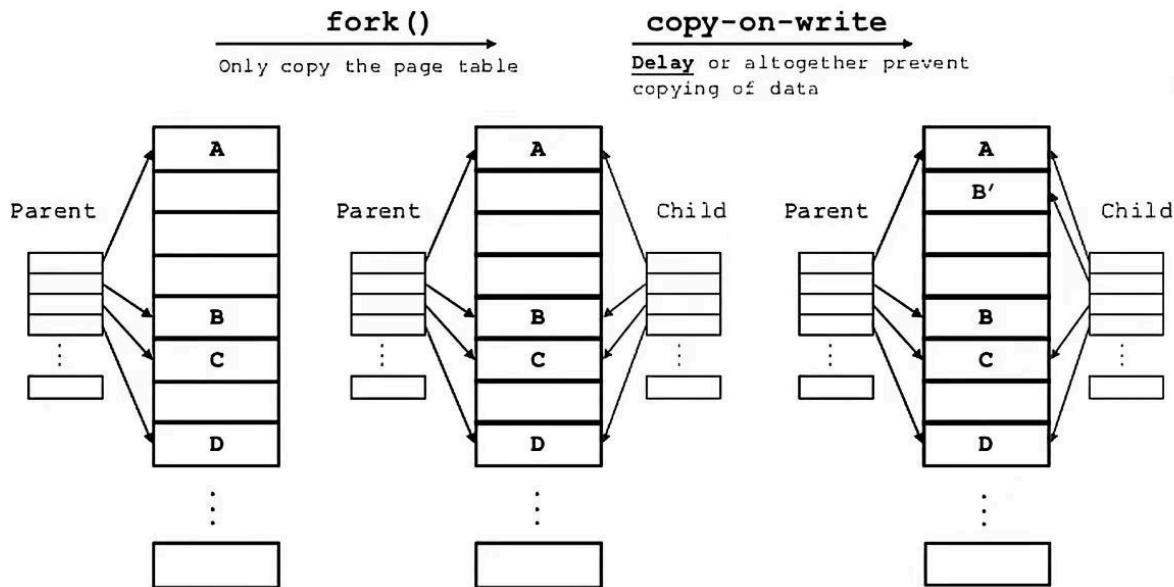
## 改进后的 COW 实现

引入写时复制 (COW) 后, xv6 的 `fork()` 被优化为:

1. **共享页面**: 父子进程最初共享所有内存页面 (如图2), 避免不必要的复制。
2. **只读保护**: 通过硬件页表标记页面为只读 (清除 `PTE_W` 标志并设置 `PTE_COW` 标志)。
3. **按需复制**: 当父子进程中的任意一个尝试写入共享页面时, 触发页错误 (Page Fault)。系统为该进程分配新的物理页面, 并将原始数据复制到新页面。

4. 引用计数管理：通过引用计数追踪页面的共享状态，只有当页面不再被任何进程使用时才释放。

# Process Creation – Copy-on-Write



如图2所示，COW 的过程如下：

1. 父子进程开始时共享页面 A、B、C、D，所有页面标记为只读。
2. 当子进程尝试写入页面 B 时，系统检测到页错误（Page Fault）。
3. 系统分配一个新的页面 B'，将原页面 B 的数据复制到 B'，并解除子进程对 B' 的只读限制。
4. 父进程仍然使用原页面 B，子进程使用新的页面 B'。

## 改进后 COW 的优点

1. **大幅节省内存：**父子进程共享内存页面，只有在需要修改时才进行实际复制。
2. **提升性能：**避免了对整个地址空间的冗余拷贝，尤其在子进程调用 `exec()` 时显著减少资源浪费。
3. **按需分配资源：**减少了物理内存的分配开销，有效提升了多进程并发性能。

## 总结

通过引入写时复制（COW），xv6 的 `fork()` 机制得到了显著优化，既减少了内存开销，又提升了性能。COW 利用了硬件的页保护和引用计数机制，是一种简单而高效的优化策略，在现代操作系统中广泛应用。

## 1. 定义 COW 标志位

文件: `mmu.h`

逻辑:

- 需要在页表条目中添加一个标志位 `PTE_COW`，用于标记页面为写时复制页面。

改动:

```
// bit 9-11 软件可用位, 标记 COW 页面
#define PTE_COW    (1UL << 9)
```

## 2. 管理页面引用计数

文件: `qq3_mm.h`

逻辑:

- 引入引用计数机制，用于追踪共享页面的引用。
- 在分配、释放页面时更新引用计数。

改动:

```
static inline void inc_page_ref(struct page *page) {
    acquire(&kmem.lock); // 加锁保护引用计数
    page->_count++;
    release(&kmem.lock);
}

static inline void dec_page_ref(struct page *page) {
    acquire(&kmem.lock); // 加锁保护引用计数
    if (--page->_count == 0) { // 如果引用计数为 0, 则释放页面
        kfree((void *)page2va(page));
    }
    release(&kmem.lock);
}
```

文件: `qq3_mm.c`

逻辑:

- 初始化页面引用计数和内存管理结构。

改动:

```

#define PGNUM_MAX 1024 * 128 // 最大页面数
struct page _mem_map[PGNUM_MAX];

void qq3_mm_init(unsigned pa, unsigned end_pa) {
    mem_map = _mem_map;
    memset(mem_map, 0, PGNUM_MAX * sizeof(struct page));

    for (int i = 0; i < (end_pa - pa) / PGSIZE; i++) {
        mem_map[i]._count = 1; // 初始化引用计数
    }

    cprintf("Total RAM size: %dMB\n", end_pa / 1024 / 1024);
}

```

### 3. 捕获和处理页错误

文件: `trap.c`

逻辑:

- 捕获页错误, 判断是否是 `COW` 页面写入引发的错误。
- 如果是, 分配新页面并将原页面的数据复制到新页面。

改动:

```

case T_PGFLT: { // 页错误处理
    uint err_va = rcr2(); // 获取引发错误的虚拟地址
    pte_t *pte = walkpgdir(myproc()->pgdir, (void *)err_va, 0);

    if (pte && (*pte & PTE_COW)) {
        // 处理 COW 页面
        char *new_page = kalloc();
        if (new_page == 0) {
            cprintf("COW: kalloc failed\n");
            myproc()->killed = 1;
            break;
        }
        // 复制原页面数据到新页面
        memmove(new_page, (char *)P2V(PTE_ADDR(*pte)), PGSIZE);

        // 更新页表
        uint old_pa = PTE_ADDR(*pte);
        *pte = V2P(new_page) | PTE_P | PTE_W | PTE_U;

        // 更新引用计数
        acquire(&kmem.lock);
        dec_page_ref(pte2page(old_pa)); // 旧页面引用计数减一
        release(&kmem.lock);

        // 刷新 TLB
        tcr3(V2P(myproc()->pgdir));
    } else {
        // 非 COW 页错误
        cprintf("Page fault at %x\n", err_va);
    }
}

```

```
    myproc()->killed = 1;
}
break;
}
```

## 4. 扩展 fork 支持 COW

文件: `proc.h`

逻辑:

- 定义 `xfork` 系统调用接口和扩展的 `copyvm_ex`。

改动:

```
// 定义 xfork 策略
#define FORK_POLICY_OLD      1 // 传统 fork
#define FORK_POLICY_COW     2 // Copy-On-Write
#define FORK_POLICY_THREAD  3 // 线程共享

int xfork(int policy); // 新增系统调用
pde_t *copyvm_ex(pde_t *pgdir, uint sz, int policy); // 扩展内存复制
```

## 5. 实现 copyvm\_ex

文件: `vim.c`

逻辑:

- 根据策略选择是使用 COW 还是传统拷贝方式。
- 对 COW 页面, 设置 `PTE_COW` 标志并清除写权限。

改动:

```
pde_t *copyvm_ex(pde_t *pgdir, uint sz, int policy) {
    pde_t *pgdir2;
    pte_t *pte;
    uint pa, i, flags;

    if ((pgdir2 = setupkvm()) == 0)
        return 0;

    for (i = 0; i < sz; i += PGSIZE) {
        if ((pte = walkpgdir(pgdir, (void *)i, 0)) == 0)
            panic("copyvm: pte should exist");
        if (!(*pte & PTE_P))
            panic("copyvm: page not present");

        pa = PTE_ADDR(*pte);
        flags = PTE_FLAGS(*pte);

        if (policy == FORK_POLICY_COW) {
            flags &= ~PTE_W; // 清除写标志
        }
    }
}
```

```

        flags |= PTE_COW; // 设置 COW 标志
        inc_page_ref(pte2page(pa)); // 增加引用计数
    } else {
        // 传统拷贝策略
        char *mem = kalloc();
        if (mem == 0) {
            freevm(pgdir2);
            return 0;
        }
        memmove(mem, (char *)P2V(pa), PGSIZE);
        pa = V2P(mem);
    }

    // 映射新页面
    if (mappages(pgdir2, (void *)i, PGSIZE, pa, flags) < 0) {
        freevm(pgdir2);
        return 0;
    }
}
return pgdir2;
}

```

## 6. 实现 xfork 系统调用

文件: `proc.c`

逻辑:

- 调用 `copyvm_ex` 创建子进程的地址空间。

改动:

```

int xfork(int policy) {
    struct proc *np;
    struct proc *curproc = myproc();

    // 分配新的进程结构
    if ((np = allocproc()) == 0) {
        return -1;
    }

    // 根据策略复制父进程地址空间
    if ((np->pgdir = copyvm_ex(curproc->pgdir, curproc->sz, policy)) == 0) {
        kfree(np->kstack);
        np->kstack = 0;
        np->state = UNUSED;
        return -1;
    }

    np->sz = curproc->sz;
    np->parent = curproc;
    *np->tf = *curproc->tf;

    // 子进程返回 0
}

```

```

np->tf->eax = 0;

for (int i = 0; i < NOFILE; i++)
    if (curproc->ofile[i])
        np->ofile[i] = filedup(curproc->ofile[i]);
np->cwd = idup(curproc->cwd);

safestrncpy(np->name, curproc->name, sizeof(curproc->name));
np->state = RUNNABLE;

return np->pid;
}

```

## 7. 注册系统调用

文件: `sysproc.c`

逻辑:

- 调用 `xfork` 接口, 实现策略选择。

改动:

```

int sys_xfork(void) {
    int policy;
    if (argint(0, &policy) < 0) {
        return -1;
    }
    return xfork(policy);
}

```

文件: `syscall.h`

逻辑:

- 为 `xfork` 分配系统调用号。

改动:

```
#define SYS_xfork 22
```

文件: `syscall.c`

逻辑:

- 在系统调用表中注册 `xfork`。

改动:

```
extern int sys_xfork(void);

static int (*syscalls[])(void) = {
    ...
    [SYS_xfork] sys_xfork,
};
```

## 8. 测试用例

文件: `testfork.c`

逻辑:

- 测试不同策略的行为, 验证 `COW` 是否生效。

改动:

```
#include "types.h"
#include "stat.h"
#include "user.h"

#define DATA_SEG_SIZE 1024 * 1024 * 128 // 数据段大小

char data_seg[DATA_SEG_SIZE]; // 用于测试的数据段

void walk_pages() {
    char *p = data_seg;
    for (int i = 0; i < DATA_SEG_SIZE; i += 4096) {
        p[i] = 1; // 修改页面内容
    }
}

int main(void) {
    printf(1, "Test xfork with COW:\n");

    int pid = xfork(FORK_POLICY_COW); // 使用 COW 策略创建子进程
    if (pid == 0) {
        printf(1, "Child process\n");
        walk_pages(); // 修改共享页面
    } else {
        wait(); // 等待子进程完成
        printf(1, "Parent process\n");
    }
    exit();
}
```

## 整体逻辑顺序总结:

1. 定义 `PTE_COW` 标志位。
2. 引入引用计数, 管理页面共享。
3. 捕获页错误, 处理 COW 逻辑。



4. 扩展 `fork` 支持 COW 策略。
5. 实现 `copyvm_ex` , 创建支持 COW 的地址空间。
6. 实现 `xfork` 系统调用。
7. 注册系统调用。
8. 编写测试用例验证 COW 行为。