Certainly. Here is a **comprehensive, step-by-step explanation** of how the **xv6 swap space implementation** works, **with code and detailed explanations** for each stage—from page allocation to swapping in and out.

🌐 OVERVIEW

In xv6, **swap space** extends usable memory by allowing pages to be **evicted to disk** when RAM is full. This is implemented through:

- Swap initialization (`swapinit`)
- Page allocation with eviction (`kalloc`)
- Trap handling for swapped pages (`usertrap`)
- Page replacement algorithm (`select_victim`)
- Disk I/O for swap-in/swap-out
- Bitmap to track free swap slots

---

1. 🔧 SWAP STRUCTURE AND INITIALIZATION

📁 `vm.c`

```
struct swap {

    uint block_start;     // Start block on disk (e.g., 1000)

    uint block_count;     // Number of blocks (e.g., 1024)

    uchar *bitmap;        // 1 bit per slot (512 slots for 512KB swap)

    struct spinlock lock;

};

struct swap swapdev;

void swapinit() {

    initlock(&swapdev.lock, "swap");

    swapdev.block_start = 1000;
```

```
    swapdev.block_count = 1024;

    // Allocate bitmap: 1 bit per page, so 512 slots = 64 bytes

    swapdev.bitmap = (uchar*)kalloc();

    memset(swapdev.bitmap, 0, 1024 / 8); // Clear bitmap

}
```

🔍 Explanation

- Swap begins at block 1000.
- Supports 1024 512-byte blocks = 512KB swap space.
- Each page takes 8 blocks (4096 bytes), so 512 slots.
- Bitmap tracks whether each slot is occupied (1) or free (0).

---

2. 📦 PAGE ALLOCATION WITH SWAP EVICTION

📁 `kalloc.c`

```
void* kalloc() {

    acquire(&kmem.lock);

    struct run *r = kmem.freelist;

    if (r)

        kmem.freelist = r->next;

    release(&kmem.lock);

    if (r)

        memset((char*)r, 5, PGSIZE);

    else {

        // 🔁 If no free pages, trigger eviction
```

```
        struct proc *p = myproc();

        uint64 va = select_victim(p);

        swapout(p, va);        // Evict a victim page to disk

        return kalloc();       // Try allocation again

    }

    return (void*)r;

}
```

🔍 Explanation

- Allocates a physical page.
- If memory is exhausted, chooses a victim page to evict using `select_victim`.
- Frees RAM by swapping the victim page to disk.
- Recursively retries allocation.

---

3. ⚙️ PAGE REPLACEMENT — CLOCK ALGORITHM

📁 `vm.c`

```
uint64 select_victim(struct proc *p) {

    static uint64 clock_hand = 0;

    uint64 start_va = p->trapframe->sp & ~(PGSIZE - 1);

    uint64 max_va = MAXVA;

    for (;;) {

        pte_t *pte = walk(p->pagetable, clock_hand, 0);

        if (pte && (*pte & PTE_V)) {

            if (!(*pte & PTE_A)) {

                return clock_hand;
```

```
            } else {
                *pte &= ~PTE_A;  // Clear access bit
            }
        }
        clock_hand = (clock_hand + PGSIZE) % max_va;
    }
}
```

🔍 Explanation

- Walks through virtual addresses.
- Selects the first page with the **access bit cleared**.
- Otherwise clears the access bit and continues.
- Implements second-chance clock algorithm.

---

4. ⬇️ SWAP OUT

📁 `vm.c`

```
int swapout(struct proc *p, uint64 va) {
    pte_t *pte = walk(p->pagetable, va, 0);
    if (!pte || !(*pte & PTE_V)) return -1;
    uint64 pa = PTE2PA(*pte);
    int slot = -1;
    acquire(&swapdev.lock);
    for (int i = 0; i < swapdev.block_count / 8; i++) {
        if ((swapdev.bitmap[i/8] & (1 << (i % 8))) == 0) {
```

```c
            swapdev.bitmap[i/8] |= (1 << (i % 8));
            slot = i;
            break;
        }
    }
    release(&swapdev.lock);
    if (slot == -1) return -1;
    struct buf *b = bread(0, swapdev.block_start + slot * 8);
    memmove(b->data, (char*)pa, PGSIZE);
    bwrite(b);
    brelse(b);
    // Update PTE: mark not present, mark swapped, encode slot
    *pte = (slot << 10) | (PTE_FLAGS(*pte) & ~PTE_V) | PTE_SWAPPED;
    kfree((void*)pa); // Free physical memory
    return 0;
}
```

🔍 Explanation

- Finds a free swap slot in the bitmap.
- Writes page content from RAM to disk using `bread`/`bwrite`.
- Updates the page table entry:
    - Clears valid bit (`PTE_V`)
    - Sets `PTE_SWAPPED`
    - Encodes slot number in unused PTE bits

- Frees the physical page.

---

5. ⬆️ SWAP IN

📁 `vm.c`

```
int swapin(struct proc *p, uint64 va) {

    pte_t *pte = walk(p->pagetable, va, 0);

    if (!pte || !(*pte & PTE_SWAPPED)) return -1;

    int slot = (*pte >> 10) & 0x3FFFFF;

    char *pa = kalloc();

    if (!pa) return -1;

    struct buf *b = bread(0, swapdev.block_start + slot * 8);

    memmove(pa, b->data, PGSIZE);

    brelse(b);

    acquire(&swapdev.lock);

    swapdev.bitmap[slot / 8] &= ~(1 << (slot % 8));

    release(&swapdev.lock);

    *pte = PA2PTE(pa) | PTE_FLAGS(*pte) | PTE_V;

    *pte &= ~PTE_SWAPPED;

    return 0;

}
```

🔍 Explanation

- Extracts swap slot from the PTE.

- Allocates a new physical page.
- Reads page from disk using `bread()`.
- Updates page table:
    - Marks valid (`PTE_V`)
    - Clears `PTE_SWAPPED`
- Frees the swap slot in the bitmap.

---

6. ⚠️ PAGE FAULT HANDLING

📁 `trap.c`

```
void usertrap(void) {

   if (r_scause() == 13 || r_scause() == 15) {

      uint64 va = r_stval();

      pte_t *pte = walk(p->pagetable, PGROUNDDOWN(va), 0);

      if (pte && (*pte & PTE_SWAPPED)) {

         if (swapin(p, PGROUNDDOWN(va)) < 0)

            panic("swapin failed");

         return;

      }

      panic("page fault");

   }

}
```

🔍 Explanation

- Handles **load/store page faults**.

- Checks if the PTE has `PTE_SWAPPED`.
- If yes, calls `swapin()` to bring page back into RAM.
- If `swapin` fails, kernel panics.
- If fault isn't due to swapping, panics by default.

---

7. 📊 BITMAP MANAGEMENT

- Each bit in `bitmap[i]` represents one **swap slot**.
- `bitmap[i] & (1 << j)` checks if slot is used.
- Setting a bit: `bitmap[i] |= (1 << j)`
- Clearing a bit: `bitmap[i] &= ~(1 << j)`

📈 Slot Mapping:

- 512KB swap / 4KB page = **128 pages**
- So bitmap is **128 bits = 16 bytes**

---

8. 💿 DISK I/O: `virtio_disk.c`

void virtio_disk_rw(struct buf *b, int write) {

// Uses DMA and descriptors to transfer data

// Sets VIRTIO_BLK_T_OUT (write) or VIRTIO_BLK_T_IN (read)

// Not directly called by swap.c but used via bread()/bwrite()

}

Disk I/O is performed through `bread()`/`bwrite()` which internally call `virtio_disk_rw()`.

✅ END-TO-END FLOW SUMMARY

malloc() -> kalloc() -> No free page?

-> select_victim()

-> swapout(victim)

-> free page -> retry kalloc()

User accesses swapped-out page -> Page fault

-> usertrap()

-> swapin(va)

-> page loaded into RAM

If you want, I can draw a **flowchart** or generate an annotated code map showing interconnections. Would that help clarify the control flow?