

MP2: Frame Manager

Naveen Babu Krishnasamy Jeyakumar

UIN: 335007146

CSCE611: Operating System

Assigned Tasks:

Main: Frame Manager - Completed.

Files Modified: cont_frame_pool.H and cont_frame_pool.C

System Design:

The goal of Machine Problem 2 is to create a frame pool manager that handles the allocation and release of frames for both the kernel frame pool (2MB) and the process frame pool (28MB). Additionally, it should mark certain frames as inaccessible (15MB), making them off-limits to users.

Contiguous Frame Pool Design: The contiguous frame pool manages frames for the kernel pool between 2MB and 4MB, and frames for the process pool from 4MB to 32MB. It uses a bitmap methodology where each frame is represented by two bits indicating its state (11 - Frame is used, 00 - Frame is free, 01 - Frame is HoS and allocated). Since two bits are used per frame, one byte of the bitmap can store information for 4 frames. Given that the frame size is 4KB, a bitmap can store a total of 16K frames (4KB * 4).

Code Description:

Contiguous Frame Pool Constructor: Initializes the data structures for the contiguous frame pool from the starting base frame number to base frame number + number of frames. It also uses an info frame number to store the state of the frame in a bitmap. If the info frame number is 0, the first base frame number is used as the info frame.

```
ContFramePool::ContFramePool(unsigned long _base_frame_no,
                             unsigned long _nframes,
                             unsigned long _info_frame_no)
{
    assert(_nframes <= FRAME_SIZE * 4);

    int num_bytes;
    base_frame_no = _base_frame_no;
    nframes = _nframes;
    nFreeFrames = _nframes;
    info_frame_no = _info_frame_no;

    /* If _info_frame_no is zero then we keep management info in the first
     * frame, else we use the provided frame to keep management info */
    if(info_frame_no == 0) {
        bitmap = (unsigned char *) (base_frame_no * FRAME_SIZE);
    } else {
        bitmap = (unsigned char *) (base_frame_no + (info_frame_no * FRAME_SIZE));
    }

    assert ((nframes % 4) == 0);

    /* Everything ok. Proceed to mark all frame as free and set bitmap to 0
     * which indicates that all the frames are free */
    for(unsigned long fno = 0; fno < _nframes; fno++) {
        set_state(fno, FrameState::Free);
    }

    // Mark the first frame as being used if it is being used
    if(_info_frame_no == 0) {
        set_state(0, FrameState::HoS);
        nFreeFrames--;
    } else {
        set_state(0, FrameState::HoS);
        nFreeFrames--;
        for (int i = 0; i < info_frame_no; i++) {
            set_state(i, FrameState::Used);
            nFreeFrames--;
        }
    }

    if (ContFramePool::frame_pool_head == NULL) {
        ContFramePool::frame_pool_head = this;
        ContFramePool::frame_pool_tail = this;
    } else {
        ContFramePool::frame_pool_tail->frame_pool_next = this;
        ContFramePool::frame_pool_tail = this;
    }
    frame_pool_next = NULL;

    Console::puts("ContFramePool::Constructor initialized\n");
}
```

Figure 1: Contiguous frame pool constructor API code

get_frames API: Takes the number of frames requested as an argument and returns the head frame number. This function searches for free frames in the bitmap, marks the first free frame as the head

frame, and the rest as used (allocated) frames. It then returns the head frame number. This API allocates only contiguous frames and skips non-contiguous frames even if they are available.

```

unsigned long ContFramePool::get_frame(unsigned int _n_frames)
{
    // Any frames left to allocate?
    if (_n_frames > nFreeFrames) {
        Console::puts("There may free frames are not available!");
        Console::puts("nFreeFrames = "); Console::puti(nFreeFrames); Console::puts("\n");
        Console::puti(_n_frames); Console::puti(_n_frames); Console::puts("\n");
    }

    unsigned int i = 0; count = 0; cont_n_avail = 0;
    while ((i < ContFramePool::nFrames) && (count < _n_frames)) {
        if (get_state(i) == FrameState::Free) {
            count++;
        } else {
            count = 0;
        }

        if (count == _n_frames) {
            cont_n_avail = i;
            break;
        }

        i++;
    }

    if (cont_n_avail == 0) {
        Console::puts("Continuous memory not available!\n");
        return 0;
    }

    unsigned long first_free_frame = i - _n_frames + 1;
    for (i = first_free_frame; i < (first_free_frame + _n_frames); i++) {
        if (i == first_free_frame) {
            set_state(i, FrameState::HOS);
        } else {
            set_state(i, FrameState::Used);
        }
        nFreeFrames--;
    }

    return (first_free_frame + base_frame_no);
}

```

Figure 2: `get_frames` code

release_frames API: Takes the head frame number as input and returns nothing. This function first checks whether the frame is inside the process pool or kernel pool, marks the head frame as free, and marks the rest of the allocated frames as free until it reaches another head frame or free frame.

[illegible]

Figure 3: release_frames API code

mark_inaccessible API: Takes the base frame number and number of frames as input and returns nothing. This function marks the frames from the base frame number to base frame number + number of frames as head frames to make them inaccessible.

```
void ContFramePool::mark_inaccessible(unsigned long _base_frame_no,
                                     unsigned long _nframes)
{
    // Mark all frames in the range as being used.
    for(unsigned long i = _base_frame_no; i < _base_frame_no + _nframes; i++) {
        set_state(i - this->base_frame_no, FrameState::HoS);
        nfreeFrames--;
    }
}
```

Figure 4: mark_inaccessible API code

needed_info_frames API: Takes the total number of frames as an argument and returns the number of info frames required to store the total frames. In a simple pool, 1 bit is used for addressing the frame, so the total number of frames that can be stored in the bitmap is 32K. Similarly, in a contiguous pool, 2 bits per frame are used, so the total frames that can be stored in the bitmap are 16K. Hence, the total number of info frames is: $\text{total frames} / 16K + (\text{total frames} / 32K)$.

```

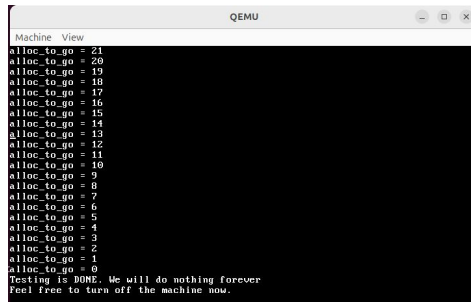
unsigned long ContFramePool::needed_info_frames(unsigned long _nframes)
{
    if (_nframes > FRAME_SIZE * 4) {
        Console::puts("Frame number out of range");
        return 0;
    }
    unsigned long max_bits_in_frame = 8 * ContFramePool::FRAME_SIZE;
    return _nframes / max_bits_in_frame + (_nframes % max_bits_in_frame > 0 ? 1 : 0);
}

```

Figure 5: needed_info_frames API code

Testing:

The testing is conducted using Bochs, where the test case recursively allocates memory and stores a unique value in each memory location. After recursion, the values in memory are checked against what was originally stored. If any value has changed, the test fails, indicating a memory overwrite. The test also provides the memory address and allocation number where the failure occurred, aiding in debugging.



```
Machine View
alloc.to.go = 21
alloc.to.go = 20
alloc.to.go = 19
alloc.to.go = 18
alloc.to.go = 17
alloc.to.go = 16
alloc.to.go = 15
alloc.to.go = 14
alloc.to.go = 13
alloc.to.go = 12
alloc.to.go = 11
alloc.to.go = 10
alloc.to.go = 9
alloc.to.go = 8
alloc.to.go = 7
alloc.to.go = 6
alloc.to.go = 5
alloc.to.go = 4
alloc.to.go = 3
alloc.to.go = 2
alloc.to.go = 1
alloc.to.go = 0
Testing is DONE. We will do nothing forever
Feel free to turn off the machine now.
```

Figure 6: Output