MP2: Frame Manager

Manikanta Gudipudi
UIN: 335006924
CSCE611: Operating System

Assigned Tasks

Main: Completed.

System Design

The goal of this machine problem was to implement a contiguous frame pool manager for a demand-paging based virtual memory system. The frame manager is responsible for allocating and releasing contiguous sequences of physical frames, which will be used by the kernel and user processes.

Key Design Decisions:

- **Bitmap-based Management:** Used a bitmap with 2 bits per frame to track four states: Free (0), Used (1), Head-of-Sequence (2), and Reserved (3). This allows efficient tracking of contiguous allocations and prevents accidental release of inaccessible memory.
- **Four-State System:**
  - `Free`: Frame is available for allocation
  - `Used`: Frame is allocated but not the first in a sequence
  - `HoS`: Frame is allocated and is the first frame in a contiguous sequence
  - `Reserved`: Frame is marked as inaccessible and cannot be allocated or released
- **Internal vs External Management:** The frame pool can store its management information either internally (using the first frame of the pool) or externally (using frames from another pool, like the kernel pool for the process pool).
- **Static Pool Tracking:** Used a static array with a configurable maximum size (`Max_pools = 2`) to track all frame pools, enabling the static `release_frames` function to identify which pool owns a given frame. This design supports the typical use case of having one kernel pool and one process pool.
- **First-Fit Allocation:** Implemented a first-fit algorithm to find contiguous sequences of free frames, ensuring efficient memory utilization.

The system supports two frame pools:

- **Kernel Frame Pool:** Manages frames from 2MB to 4MB, uses internal management
- **Process Frame Pool:** Manages frames from 4MB to 32MB, uses external management (stored in kernel pool frames)

Code Description

I modified `cont_frame_pool.H` and `cont_frame_pool.C` to implement the frame pool manager. The kernel uses the frame pool through the test function in `kernel.C`. To compile the code, use `make` in the project directory.

cont_frame_pool.H: Data Structures The header file defines the private data members for the frame pool:

Listing 1: Frame Pool Data Structures

```
private:
    unsigned char * bitmap;        // Bitmap for frame states (2 bits per frame)
    unsigned int    nFreeFrames;   // Number of free frames
    unsigned long   base_frame_no; // First frame number managed by this pool
    unsigned long   nframes;       // Total number of frames in this pool
    unsigned long   info_frame_no; // Frame number for management info (0 = internal)

    // Static list to track all frame pools for release_frames
    static const unsigned int Max_pools = 2;  // Maximum number of frame pools (kernel + process)
    static ContFramePool* frame_pools[];      // Array of frame pool pointers
    static unsigned int num_pools;            // Current number of active pools
```

cont_frame_pool.C: Constructor The constructor initializes the frame pool and sets up the bitmap:

Listing 2: Frame Pool Constructor

```
ContFramePool::ContFramePool(unsigned long _base_frame_no,
                             unsigned long _n_frames,
                             unsigned long _info_frame_no)
{
    base_frame_no = _base_frame_no;
    nframes = _n_frames;
    nFreeFrames = _n_frames;
    info_frame_no = _info_frame_no;

    // If _info_frame_no is zero, use first frame internally
    if (info_frame_no == 0) {
        bitmap = (unsigned char *) (base_frame_no * FRAME_SIZE);
    } else {
        bitmap = (unsigned char *) (info_frame_no * FRAME_SIZE);
    }

    // Initialize all frames as free
    for (unsigned long fno = 0; fno < _n_frames; fno++) {
        set_state(fno, FrameState::Free);
    }

    // If using internal management, mark first frame as used
    if (info_frame_no == 0) {
        set_state(0, FrameState::Used);
        nFreeFrames--;
    }

    // Add this pool to the static list
    if (num_pools < Max_pools) {
        frame_pools[num_pools] = this;
        num_pools++;
    }
}
```

cont_frame_pool.C: get_frames This method allocates contiguous frames using a first-fit algorithm:

Listing 3: Frame Allocation Method

```
unsigned long ContFramePool::get_frames(unsigned int _n_frames)
{
    // Check if we have enough free frames
    if (nFreeFrames < _n_frames) {
        return 0; // Not enough free frames
    }

    // Look for a contiguous sequence of free frames
    for (unsigned long start_frame = 0; start_frame <= nframes - _n_frames;
        start_frame++) {
        bool found_sequence = true;

        // Check if we can fit _n_frames starting at start_frame
        for (unsigned int i = 0; i < _n_frames; i++) {
            FrameState state = get_state(start_frame + i);
            if (state != FrameState::Free) {
                found_sequence = false;
                break;
            }
        }

        if (found_sequence) {
            // Mark first frame as Head-of-Sequence
            set_state(start_frame, FrameState::HoS);
            nFreeFrames--;

            // Mark remaining frames as Used
            for (unsigned int i = 1; i < _n_frames; i++) {
                set_state(start_frame + i, FrameState::Used);
                nFreeFrames--;
            }

            return base_frame_no + start_frame;
        }
    }

    return 0; // No contiguous sequence found
}
```

cont_frame_pool.C: mark_inaccessible This method marks inaccessible memory regions as Reserved to prevent accidental allocation or release:

Listing 4: Mark Inaccessible Method

```
void ContFramePool::mark_inaccessible(unsigned long _base_frame_no,
                                      unsigned long _n_frames)
{
    // Convert absolute frame number to relative frame number
    unsigned long relative_base = _base_frame_no - base_frame_no;

    // Check if the range is within this pool
    if (relative_base >= nframes || relative_base + _n_frames > nframes) {
        return; // Range is outside this pool
    }

    // Mark first frame as Head-of-Sequence
    set_state(relative_base, FrameState::HoS);
    nFreeFrames--;

    // Mark remaining frames as Reserved (not Used, so they can't be
        accidentally released)
    for (unsigned long i = 1; i < _n_frames; i++) {
        set_state(relative_base + i, FrameState::Reserved);
    }
    nFreeFrames -= _n_frames;
}
```

cont_frame_pool.C: release_frames This static method identifies the correct pool and releases the frame sequence, stopping at Reserved frames to prevent accidental release of inaccessible memory:

Listing 5: Static Frame Release Method

```
void ContFramePool::release_frames(unsigned long _first_frame_no)
{
    // Find the pool that owns this frame
    for (unsigned int i = 0; i < num_pools; i++) {
        ContFramePool* pool = frame_pools[i];

        // Check if this frame belongs to this pool
        if (_first_frame_no >= pool->base_frame_no &&
            _first_frame_no < pool->base_frame_no + pool->nframes) {

            // Convert to relative frame number
            unsigned long relative_frame = _first_frame_no - pool->
                base_frame_no;

            // Check if this frame is marked as Head-of-Sequence
            if (pool->get_state(relative_frame) != FrameState::HoS) {
                Console::puts("ERROR: Trying to release frame that is not Head
                    -of-Sequence!\n");
                assert(false);
                return;
            }

            // Release the sequence starting from this frame
            unsigned long current_frame = relative_frame;

            // First, mark the Head-of-Sequence frame as free
            pool->set_state(current_frame, FrameState::Free);
            pool->nFreeFrames++;
            current_frame++;

            // Now release all subsequent frames that are marked as Used
            while (current_frame < pool->nframes) {
                FrameState state = pool->get_state(current_frame);
                if (state == FrameState::Free) {
                    break; // End of sequence - we hit a free frame
                } else if (state == FrameState::HoS) {
                    break; // End of sequence - we hit another Head-of-
                        Sequence
                } else if (state == FrameState::Reserved) {
                    break; // End of sequence - we hit a reserved frame (
                        inaccessible)
                } else if (state == FrameState::Used) {
                    // This frame is part of our sequence, release it
                    pool->set_state(current_frame, FrameState::Free);
                    pool->nFreeFrames++;
                    current_frame++;
                } else {
                    // Unknown state, stop here
                    break;
                }
            }

            return;
        }
    }

    Console::puts("ERROR: Frame not found in any pool!\n");
    assert(false);
}
```

cont_frame_pool.C: needed_info_frames This static method calculates the number of frames needed for management data:

Listing 6: Info Frames Calculation

```
unsigned long ContFramePool::needed_info_frames(unsigned long _n_frames)
{
    // Calculate bitmap size needed (2 bits per frame)
    unsigned long bitmap_size_bytes = (_n_frames * 2 + 7) / 8; // Round up to
        nearest byte

    // Calculate how many frames are needed to store the bitmap
    unsigned long frames_needed = (bitmap_size_bytes + FRAME_SIZE - 1) /
        FRAME_SIZE; // Round up

    return frames_needed;
}
```

cont_frame_pool.C: State Management Helper functions for bitmap manipulation:

Listing 7: State Management Functions

```
ContFramePool::FrameState ContFramePool::get_state(unsigned long _frame_no) {
    // Convert frame number to bitmap index (2 bits per frame)
    unsigned int bitmap_index = _frame_no * 2 / 8;
    unsigned int bit_offset = (_frame_no * 2) % 8;

    unsigned char mask = 0x3 << bit_offset; // 2 bits mask
    unsigned char state_bits = (bitmap[bitmap_index] & mask) >> bit_offset;

    switch(state_bits) {
        case 0: return FrameState::Free;
        case 1: return FrameState::Used;
        case 2: return FrameState::HoS;
        case 3: return FrameState::Reserved;
        default: return FrameState::Free; // Should not happen
    }
}

void ContFramePool::set_state(unsigned long _frame_no, FrameState _state) {
    // Convert frame number to bitmap index (2 bits per frame)
    unsigned int bitmap_index = _frame_no * 2 / 8;
    unsigned int bit_offset = (_frame_no * 2) % 8;

    unsigned char mask = 0x3 << bit_offset; // 2 bits mask
    unsigned char state_bits;

    switch(_state) {
        case FrameState::Free:     state_bits = 0; break;
        case FrameState::Used:     state_bits = 1; break;
        case FrameState::HoS:      state_bits = 2; break;
        case FrameState::Reserved: state_bits = 3; break;
    }

    // Clear the bits first, then set them
    bitmap[bitmap_index] &= ~mask;
    bitmap[bitmap_index] |= (state_bits << bit_offset);
}
```

Reserved State Implementation

To prevent accidental release of inaccessible memory blocks, the implementation includes a `Reserved` state in addition to the original three states. This enhancement addresses a critical safety issue where inaccessible memory regions could be accidentally released.

Key Improvements:

- **Reserved State:** Added `Reserved` (value 3) to the `FrameState` enum to mark inaccessible memory regions
- **Safe mark_inaccessible:** The `mark_inaccessible` function now marks frames as `Reserved` instead of `Used`, preventing accidental release
- **Protected Release:** The `release_frames` function stops when it encounters a `Reserved` frame, ensuring inaccessible memory is never released
- **Allocation Safety:** `Reserved` frames are excluded from allocation, maintaining the integrity of inaccessible regions
- **Configurable Pool Limit:** Added `Max_pools` constant (set to 2) to limit the number of frame pools, supporting the typical kernel + process pool configuration

Benefits:

- Memory safety: Inaccessible regions cannot be accidentally released
- System stability: Prevents corruption of critical memory areas
- Clear separation: Distinguishes between user-allocated and system-reserved memory
- Backward compatibility: All existing functionality continues to work unchanged

Testing

I thoroughly tested the frame pool implementation using multiple approaches:

- **Compilation Testing:** Verified that the code compiles successfully with `make` without any errors or warnings.
- **Memory Test:** The provided recursive memory test in `kernel.C` was used to validate the implementation. This test:
  - Performs 32 recursive allocations with varying frame counts (1-4 frames per allocation)

  - Writes unique values to allocated memory
  - Recursively calls itself to test nested allocations
  - Verifies that memory contents remain intact after recursive calls
  - Releases frames in reverse order of allocation

- **Test Results:** The test completed successfully with "Testing is DONE. We will do nothing forever", indicating no memory corruption or allocation failures.

- **Frame Pool Validation:** Verified that:
  - Only one "ContFramePool initialized" message appears (kernel pool only)
  - Frame allocations are contiguous and non-overlapping
  - Frame releases properly free the correct number of frames
  - No frame reuse occurs before proper release

- **Edge Case Testing:** The implementation handles:
  - Insufficient free frames (returns 0)
  - Invalid frame release (asserts and exits)
  - Frame not found in any pool (asserts and exits)
  - Both internal and external management scenarios
  - Reserved frames are properly excluded from allocation and release
  - Inaccessible memory regions are protected from accidental release
  - Pool limit enforcement (Max_pools = 2) prevents excessive pool creation