MP2: Simple File System

Ajay Jagannath
UIN: 236002822
CSCE611: Operating System

Assigned Tasks

Main: Frame Manager (via continuous frame pool) - Completed.
Files modified: cont_frame_pool.C, cont_frame_pool.H and kernel.C

System Design

The goal of machine problem 2 is to implement a simple version of a frame manager that allows for allocation of frames requested and releases specific frames after use. This is done by partitioning the physical memory into frame pools, within which frames can be allotted and returned. The pooling is usually done to partition our kernel and user/process space in physical memory.

Taking inspiration from the simple pooling class given, we do frame management through our objects of our Continuous Frame Pool, where requested frames have to allocated in a contiguous sequence. As additional features, we ensure that certain memory space is marked inaccessible for processes. This is to protect reserved memory space such as the first 2MB in Kernel space and 15MB - 16MB region in user space.

We keep track of frame states by using a bitmap, which is located at certain info frames. For contiguous frame allotment, we need to represent each frame by 2 bits (00 for used, 01 for free, 10 for HoS). For this case each byte of our bitmap maps 4 frames, so a single frame of 4KB size can map up to 16k frames.

Code Description

I have primarily changed 2 files - cont_frame_pool.C and cont_frame_pool.H. The code for this was mainly inspired by simple_frame_pool.C and simple_frame_pool.H, with changes made to meet the requirements given. I will walkthrough each function below.

Kernel.C changed only for testing and all tests commented out (can be uncommented to check again).

I have already generated the kernel.binary file and added it, but that can be done using 1) 'make clean' , followed by 2) 'make'. THen to run the simulation, 3) 'make run' can be used.

Header File (cont_frame_pool.H)

This file contains the ContFramePool class declaration and the different function declarations. I have used this space to declare the multiple data members (all private) of this class that will be used in the function definitions later.

```
private:
    /* -- DEFINE YOUR CONT FRAME POOL DATA STRUCTURE(s) HERE. */
    unsigned char * bitmap;        // We implement the frame pool with a bitmap for 2 bits
    unsigned int    nFreeFrames;   //
    unsigned long   base_frame_no; // Where does the frame pool start in phys mem?
    unsigned long   nframes;       // Size of the frame pool
    unsigned long   info_frame_no; // Where do we store the management information?
    unsigned long   info_frame_total; // total frames used to store bitmap

    static ContFramePool* head; //for storing all the objects (pools) in a linked list
    ContFramePool* next;
```

Figure 1: cont_frame_pool.H

Implementation File (cont_frame_pool.C)

get_state : This method checks the status of a specific memory frame. It takes a frame number as input, locates and reads a two-bit state value from a bitmap, where each byte stores the states of four frames. The function then returns an enumeration indicating whether the frame is Used (00), Free (01), or the Head of Section (HoS, 10).

```
ContFramePool::FrameState ContFramePool::get_state(unsigned long _frame_no) {
    unsigned int bitmap_index = _frame_no / 4; // get the index of the byte containing this frame's map
    unsigned char mask = 0x3 << ((_frame_no % 4) * 2); // moving 00000011 mask to start of this frame's map - (* 2) because each frame covers 2 bits

    unsigned char state = (bitmap[bitmap_index] & mask) >> ((_frame_no % 4) * 2); // isolate frame value and bring them to the first 2 bits

    //return FrameState based on value (Used = 00, Free = 01, HoS = 10)
    if(state == 0){
        return FrameState::Used;
    }
    else if(state == 1){
        return FrameState::Free;
    }
    else{
        return FrameState::HoS;
    }

}
```

Figure 2: get_state

set_state : This method updates the status of a specific memory frame. It takes a frame number and a new state as input. The function first clears the two bits corresponding to that frame in the bitmap (after locating those bits in the bitmap). It then sets the bits to the new value based on whether the frame is Used, Free, or HoS.

ContFramePool - constructor : This method is the constructor for an object of class ContFramePool that initializes a new contiguous frame pool for memory management. It sets up the pool's boundaries by taking in the base frame and total frames in the pool, calculates the number of frames needed for its bitmap, and initializes the bitmap. The bitmap initialization can be within its memory limits or beyond it as well (if no value specified, we consider the first frame (onwards) for the bitmap). It then marks all frames as Free, except for ones used to store the management information, which are marked as HoS and Used. We also keep track of the number of free frames in this process.

mark_inaccessible : This method marks a range of frames as unavailable for allocation. It takes a base frame number and the number of frames to mark as inaccessible. The first frame in the specified range is set to HoS, and the remaining frames are set to Used. This is for protecting memory that are already in use, such as for kernel data.

```cpp
void ContFramePool::set_state(unsigned long _frame_no, FrameState _state) {
    unsigned int bitmap_index = _frame_no / 4; // get the index of the byte containing this frame's map
    unsigned char mask= 0x3 << ((_frame_no % 4) * 2); // moving 00000011 mask to start of this frame's map - (* 2) because each frame covers 2 bits

    bitmap[bitmap_index] &= ~mask; // clear bits for frame

    switch(_state) {
        case FrameState::Used: // already cleared
            break;
        case FrameState::Free:
            mask= 0x1 << ((_frame_no % 4) * 2); // moving 00000001 mask to start of this frame's map
            bitmap[bitmap_index] |= mask;
            break;
        case FrameState::HoS:
            mask= 0x2 << ((_frame_no % 4) * 2); // moving 00000010 mask to start of this frame's map
            bitmap[bitmap_index] |= mask;
            break;
    }

}
```

Figure 3: set_state

```cpp
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; // initially all frames free
    info_frame_no = _info_frame_no;
    info_frame_total = needed_info_frames(_n_frames); //get total frames needed for info.

    next = head; //link previous pool to new pool
    head = this; // mark new pool as head - going in LIFO order

    // 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); // set address of bitmap as base
    } else {
        bitmap = (unsigned char *) (info_frame_no * FRAME_SIZE); // set address of bitmap as specified info_frame
    }

    // Everything ok. Proceed to mark all frame as free.
    for(unsigned long fno = 0; fno < _n_frames; fno++) {
        set_state(fno, FrameState::Free);
    }

    unsigned long info_start = info_frame_no ? info_frame_no : base_frame_no;

    // To ensure we don't go out of bounds
    if(info_start >= base_frame_no &&
        info_start + info_frame_total <= base_frame_no + nframes) {

        // first frame is HoS, rest are Used
        for (unsigned long i = 0; i < info_frame_total; i++) {

            if (i == 0) {
                set_state(info_start - base_frame_no + i, FrameState::HoS);
            } else {
                set_state(info_start - base_frame_no + i, FrameState::Used);
            }

            nFreeFrames--;        You, 6 minutes ago • Uncommitted changes
        }
    }

    Console::puts("Frame Pool initialized\n");
    // Uncomment to print the frame states after this function
    // print_frame_states("ContFramePool::ContFramePool");
}
```

Figure 4: Constructor

```
void ContFramePool::mark_inaccessible(unsigned long _base_frame_no,
                                      unsigned long _n_frames)
{
    // Mark all frames in the range as being used except for first frame.
    set_state(_base_frame_no - base_frame_no, FrameState::HoS); // relative to base_frame_no
    nFreeFrames--;

    for(unsigned long fno = _base_frame_no + 1; fno < _base_frame_no + _n_frames; fno++){
        set_state(fno - base_frame_no, FrameState::Used);
        nFreeFrames--;
    }

    // Uncomment to print the frame states after this function
    // print_frame_states("ContFramePool::mark_inaccessible");
}
```

Figure 5: mark_inaccessible

release_frames : This method frees up a contiguous block of frames, making them available for use again. It takes the starting frame number of the block, which must be an HoS frame (else we print to notify). The function then iterates through the subsequent Used frames, setting all of them, including the initial HoS frame, back to Free. It also correctly increments the pool's free frame counter. An important point here is that, this is a static function, as this is pool agnostic. We find the exact pool where the starting frame is located by storing all the pools as a static linked list and traversing through them.

```
void ContFramePool::release_frames(unsigned long _first_frame_no)
{
    // finding the pool
    ContFramePool* pool = head;
    while (pool != nullptr) {
        //check if pool has the given frame
        if (pool->base_frame_no <= _first_frame_no &&
            _first_frame_no < (pool->base_frame_no + pool->nframes)) {
            // if the frame is HoS - mark as free and set subsequesnt used frames as free
            if(pool->get_state(_first_frame_no-pool->base_frame_no) == FrameState::HoS){
            pool->set_state(_first_frame_no-pool->base_frame_no, FrameState::Free);
                unsigned long fno = _first_frame_no +1;
                pool->nFreeFrames++; // update free frames
                while((fno < (pool->base_frame_no + pool->nframes)) && // check out of bounds
                (pool->get_state(fno-pool->base_frame_no) == FrameState::Used)){
                    pool->set_state(fno-pool->base_frame_no, FrameState::Free);
                    fno++;
                    pool->nFreeFrames++;
                }
            }
            else{ // assertion for non-HoS first frame
                Console::puts("Error: release_frames called on non-HoS frame!\n");
                assert(false);
            }
            // Uncomment to print the frame states after this function
            // pool->print_frame_states("ContFramePool::release_frames");
            return;
        }
        else
            pool = pool->next; // move to next pool
    }
}
```

Figure 6: release_frames

needed_info_frames : This is a method that calculates how many full frames are required to store the bitmap for a given number of frames. Since each frame's state is stored in two bits, one frame of memory (which is 4KB or 4096 byte) can manage 4096*4=16384 frames, beyond which we need a new frame to be alotted. The function performs a ceiling division to ensure enough space is allocated for the entire bitmap.

```
unsigned long ContFramePool::needed_info_frames(unsigned long _n_frames)
{
    return (_n_frames + (FRAME_SIZE * 4) - 1) / (FRAME_SIZE * 4);  //ceil function implementation

}
```

Figure 7: needed_info_frames

print_frame_states : This helper method is a debugging tool that prints a summary of all allocated memory blocks within the frame pool. It iterates through the bitmap and identifies every block that starts with a Head of Section (HoS) frame. For each block found, it counts all the subsequent Used frames and prints a single line showing the start and end frame numbers of the contiguous block and the total number of frames it contains. It is called at end of every main function defined above.

```
void ContFramePool::print_frame_states(const char* caller) {
    Console::puts("Called from: ");
    Console::puts(caller);
    Console::puts("\n");
    Console::puts("Frame states for pool starting at ");
    Console::puti(base_frame_no);
    Console::puts(":\n");

    for(unsigned long fno = 0; fno < nframes; fno++) {
        if(get_state(fno) == FrameState::HoS) {
            unsigned long start = fno;
            unsigned long count = 1;
            fno++;
            while(fno < nframes && get_state(fno) == FrameState::Used) {
                count++;
                fno++;
            }
            fno--; // adjust back
            Console::puts("HoS + Used frames at "); // print from HoS to last Used frame chunks only
            Console::puti(start + base_frame_no);
            Console::puts("-");
            Console::puti(start + base_frame_no + count - 1);
            Console::puts(" (");
            Console::puti(count);
            Console::puts(" frames)\n");
        }
    }
}
```

Figure 8: print_frame_states

Testing

Pre-defined Test

Testing is done by executing make clean, make to create binary file and running it with make run in the terminal. By executing these commands, we execute the start.asm file, whcih is the entry point to the kernel. This then evokes the kernel.C file that contains a pool setup and initializes the kernel memory pool at 2MB with a reserved 1MB memory hole at 15MB. It tests the frame manager using test memory function, which recursively allocates and releases frames. If the memory is changed unexpectedly, an error is logged, and execution stops. Once the testing is performed, the system enters an infinite loop state, further confirming that the frame manager implemented works as expected.

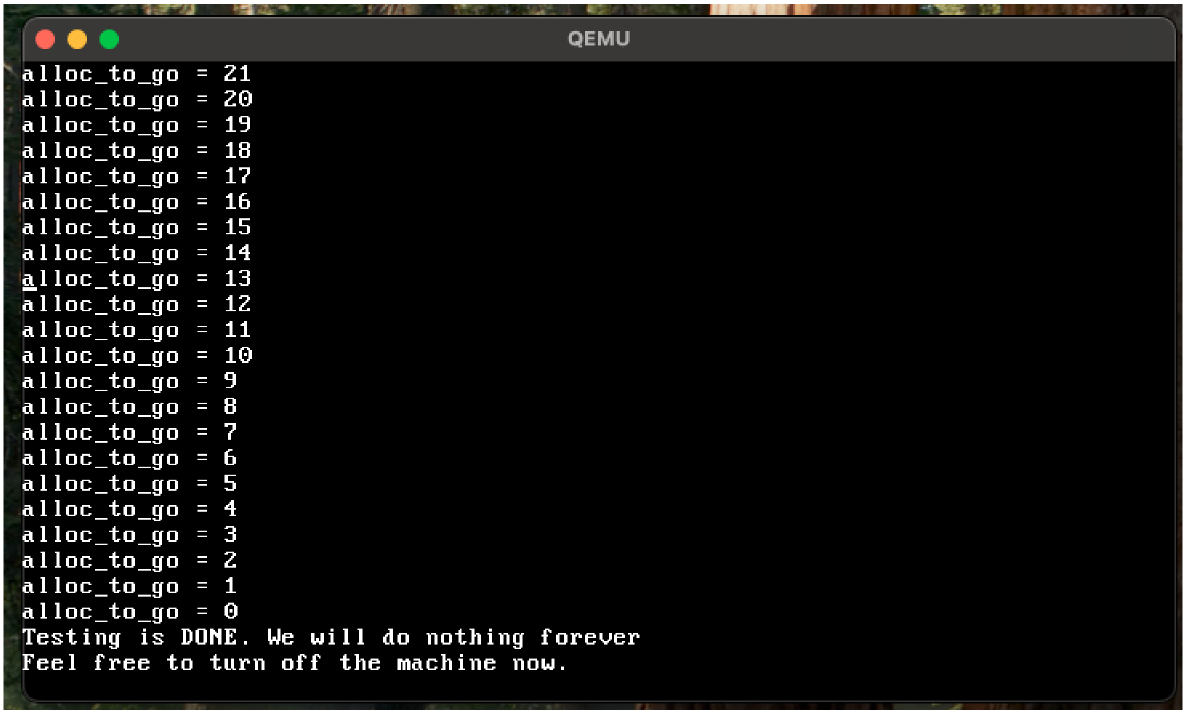

Figure 9: QEMU screen for pre-defined test

Figure 10: Console screen for pre-defined test

Custom Tests

1) Printing Frame States Using the function print_frame_states defined above, we can get all the frames within a pool that get allotted after each function and check for fidelity. We keep the call to this function disabled after this to keep the output simplified, but they can be turned on by uncommenting this call within the main functions.

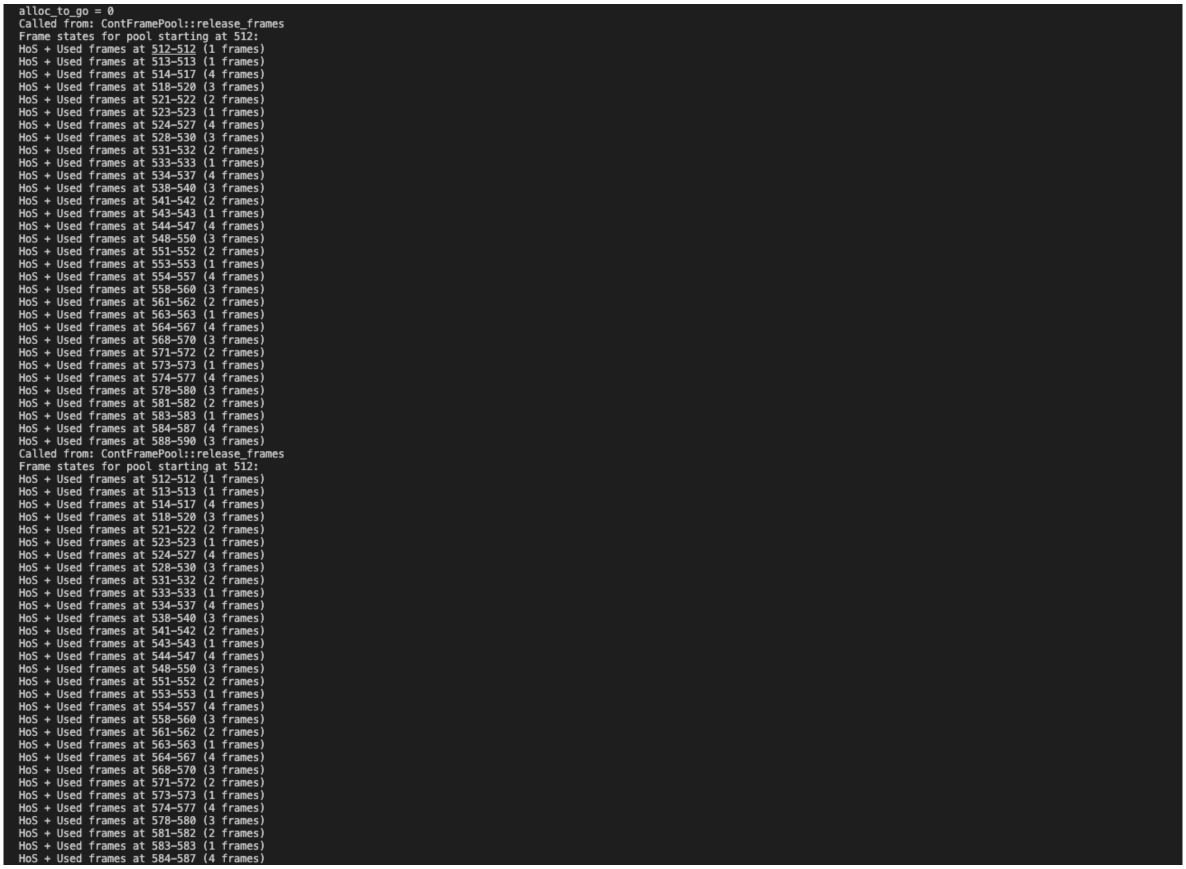

Figure 11: Frame states: They are allotted and removed in a stack like LIFO fashion

2) Testing Mark_inaccessible + more than one info_frame (+ allotted at fixed frame number) Allotted more frames to pool (20k) than the info_frame size (so need 2 frames), and info frame location set to custom value. Now checked the frame states. Also marked some frames as inaccessible and checked frame states. Commented out this in Kernel.C - can be uncommented and checked.

```
unsigned long test_pool_size = 20000; // large enough for needed_info_frames > 1
unsigned long info_frame_no = KERNEL_POOL_START_FRAME + 10; // within pool, not 0
ContFramePool test_pool(KERNEL_POOL_START_FRAME, test_pool_size, info_frame_no);
test_pool.print_frame_states("After initializing test_pool");
test_pool.mark_inaccessible(KERNEL_POOL_START_FRAME + 100, 50); // mark some frames as inaccessible
test_pool.print_frame_states("After Marking Inaccessible");
```

3) Process Pool Allocation We can test the functionality of two pools enabling the process pools and requesting frames from it. We can also check the linked list functionality by releasing frames from the process pool.

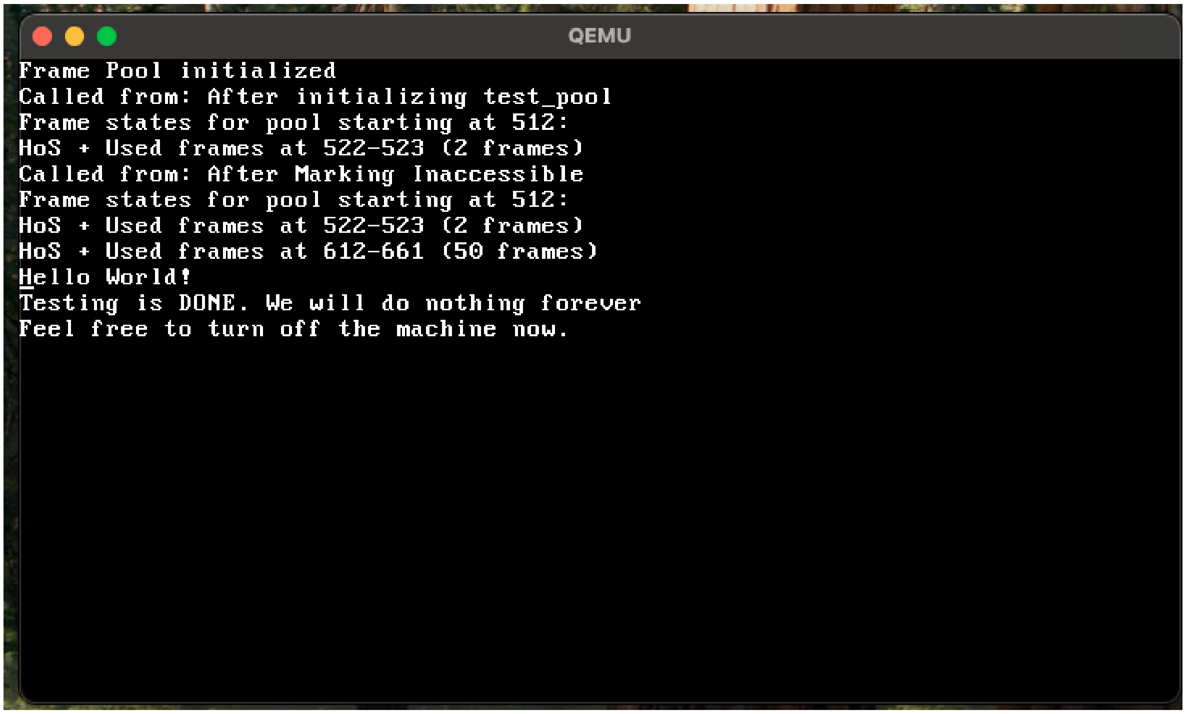

Figure 12: Test results for info_frames and mark_inaccessible

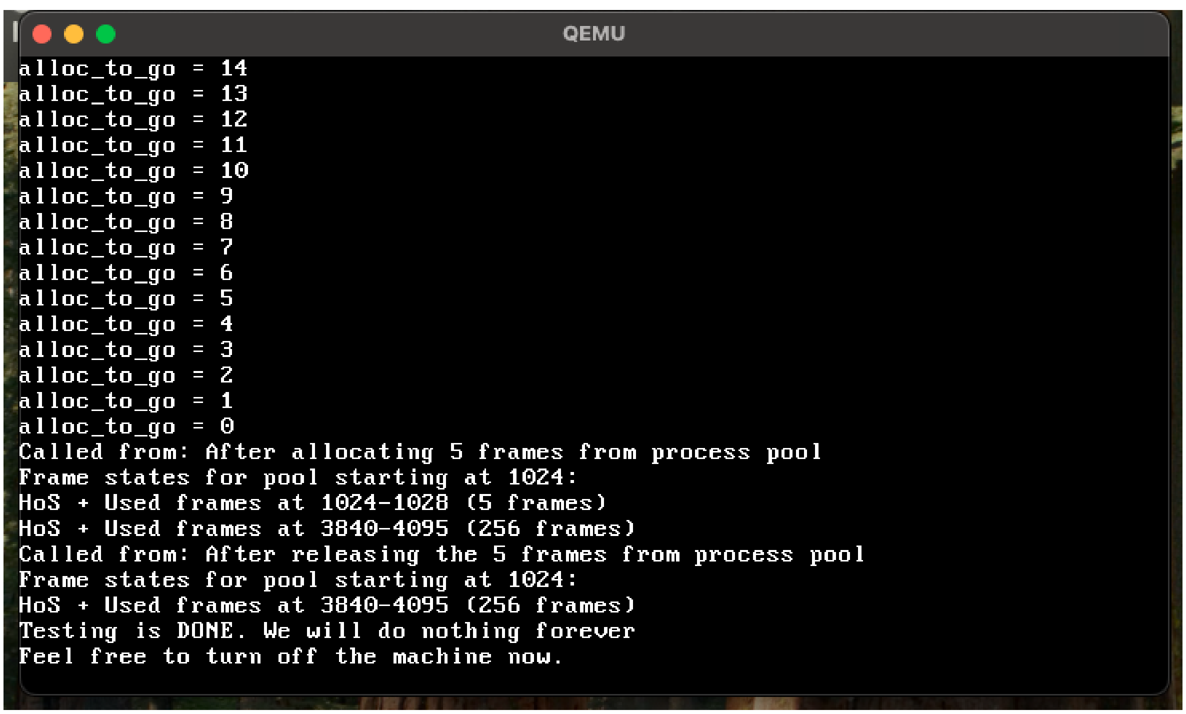

Figure 13: Test results for process pool access and release