MP2: Frame Manager

Vinay Balamurali
UIN: 936002032
CSCE611: Operating System

Assigned Tasks

Main: Completed.

System Design

The machine problem states to create a frame manager. The frame manager is responsible for the allocation and de-allocation of frames using frame pools. The frame pool is a data management structure that has the mapping of free and in-use frames for a certain logical block of memory.

In this machine problem, we are dealing with two frame pools, namely:

- Kernel Frame Pool
- Process Frame Pool

The structure of the memory is as follows:

- 32 MB of total memory.
- Frame size is 4KB
- The Kernel address space is between 0 MB and 4 MB
  - First 1 MB is reserved for global data and memory mapped devices such as the video display.
  - Between 1 MB and 2 MB resides the Kernel code.
  - Between 2 MB and 4 MB is the Kernel Frame Pool.
- The Process address space (pool) is between 4 MB and 32 MB.
  - There is a memory hole between 15 MB and 16 MB which is a reserved space of sorts, specific to the x86 architecture.

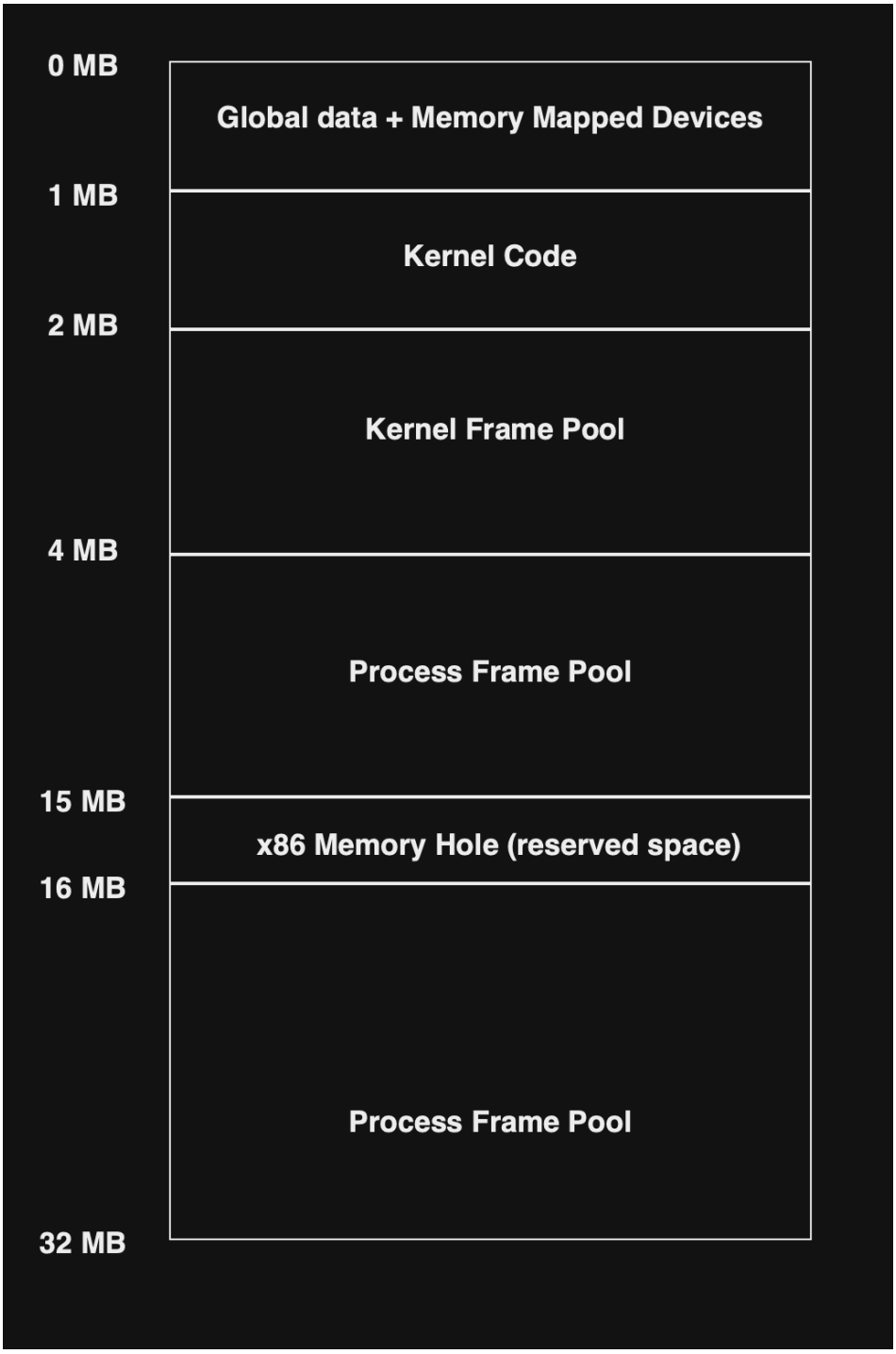


Figure 1: Memory design

Code Description

0.1 Implementation

The implementation was based on the idea that a bitmap would be used to manage frame information. But unlike the Simple Frame Allocator, over here, we would use 2 bits to represent each frame.

The bit representation is as follows:

- Free → 00
- Used → 01
- HoS → 11 (Head of Sequence)

0.2 Files Modified

- cont_frame_pool.H
- cont_frame_pool.C

cont_frame_pool.H : The following private members were introduced similar to the Simple Frame Pool.

- *unsigned char *bitMap* : The bit map having 2 bits to manager each frame.
- *nFreeFrames* : Initial no: of free frames.
- *base_frame_no* : Starting Frame No for this frame pool.
- *nframes* : total size of frame pool.
- *info_frame_no* : management info frame no.
- *ContFramePool *nextPool* : next pointer for a linked list.
- *static ContFramePool *headPool* : Head pointer for this frame pool.

```
/*--------------------------------------------------------------------------*/
/* C o n t F r a m e   P o o l */
/*--------------------------------------------------------------------------*/

class ContFramePool {

private:
    /* -- DEFINE YOUR CONT FRAME POOL DATA STRUCTURE(s) HERE. */
    unsigned char * bitMap;        // We implement the simple frame pool with a bitmap
    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?

    ContFramePool *nextPool = NULL;
    static ContFramePool *headPool;

    /* ---- STATE MANAGEMENT */

    enum class FrameState {Free, Used, HoS};

    FrameState get_state(unsigned long _frame_no);
    void set_state(unsigned long _frame_no, FrameState _state);
```

Figure 2: Header File

cont_frame_pool.C: ContFramePool::ContFramePool : This is the parameterised constructor. This method is involved with initialising the frame management bit map in the frame pool to 'Free'. It is also responsible for

```
// Parameterized constructor.
ContFramePool::ContFramePool(unsigned long _base_frame_no,
                             unsigned long _n_frames,
                             unsigned long _info_frame_no)
{
    // Bitmap must fit in a single frame!
    assert(_n_frames <= FRAME_SIZE * 8);

    // initialize private members.
    this->base_frame_no = _base_frame_no;
    this->nframes = _n_frames;
    this->info_frame_no = _info_frame_no;
    this->nFreeFrames = this->nframes;

    // Place the Management frame appropriately.
    if (this->info_frame_no == 0)
    {
        this->bitMap = (unsigned char *) (this->base_frame_no * FRAME_SIZE);
    }
    else
    {
        this->bitMap = (unsigned char *) (this->info_frame_no * FRAME_SIZE);
    }

    // Set all frames to free
    for (int frame = 0; frame < this->nframes; frame++)
    {
        set_state(frame, FrameState::Free);
    }

    // Set the info frame as 'used'.
    if (this->info_frame_no == 0)
    {
        set_state(0, FrameState::Used);
        // Don't forget to decrement :)
        this->nFreeFrames--;
    }

    // Linked list stuff.
    // Add a node at the end.
    // This linked is allocatted in the stack currently.
    if (headPool == NULL)
    {
        headPool = this;
        headPool->nextPool = NULL;
    }
    else
```

Figure 3: Constructor

cont_frame_pool.C: get_state : The *get_state* method creates the appropriate bit mask based on the frame number. It then fetches the state as an *enum class* based on one of the three values.

```
// Method to return one of the 3 states of the bit map.
ContFramePool::FrameState ContFramePool::get_state(unsigned long _frame_no)
{
    unsigned int bitMapIndex = _frame_no / 4;
    unsigned int shift = ((_frame_no % 4) * 2);
    unsigned char mask = 0x03 << shift;
    unsigned char res = ((bitMap[bitMapIndex] & mask) >> shift);
    // Console::puts("get_state = "); Console::puti(res); Console::puts("\n");

    if (res == 0x03)
    {
        Console::puts("Frame is Head of Sequence!\n");
        return FrameState::HoS;
    }
    else if (res == 0x01)
    {
        Console::puts("Frame is Used!\n");
        return FrameState::Used;
    }
    else if (res == 0x00)
    {
        Console::puts("Frame is Free!\n");
        return FrameState::Free;
    }

    // Console::puts("Frame state unknown\n");
    return FrameState::Free;

} // ContFramePool::get_state */
```

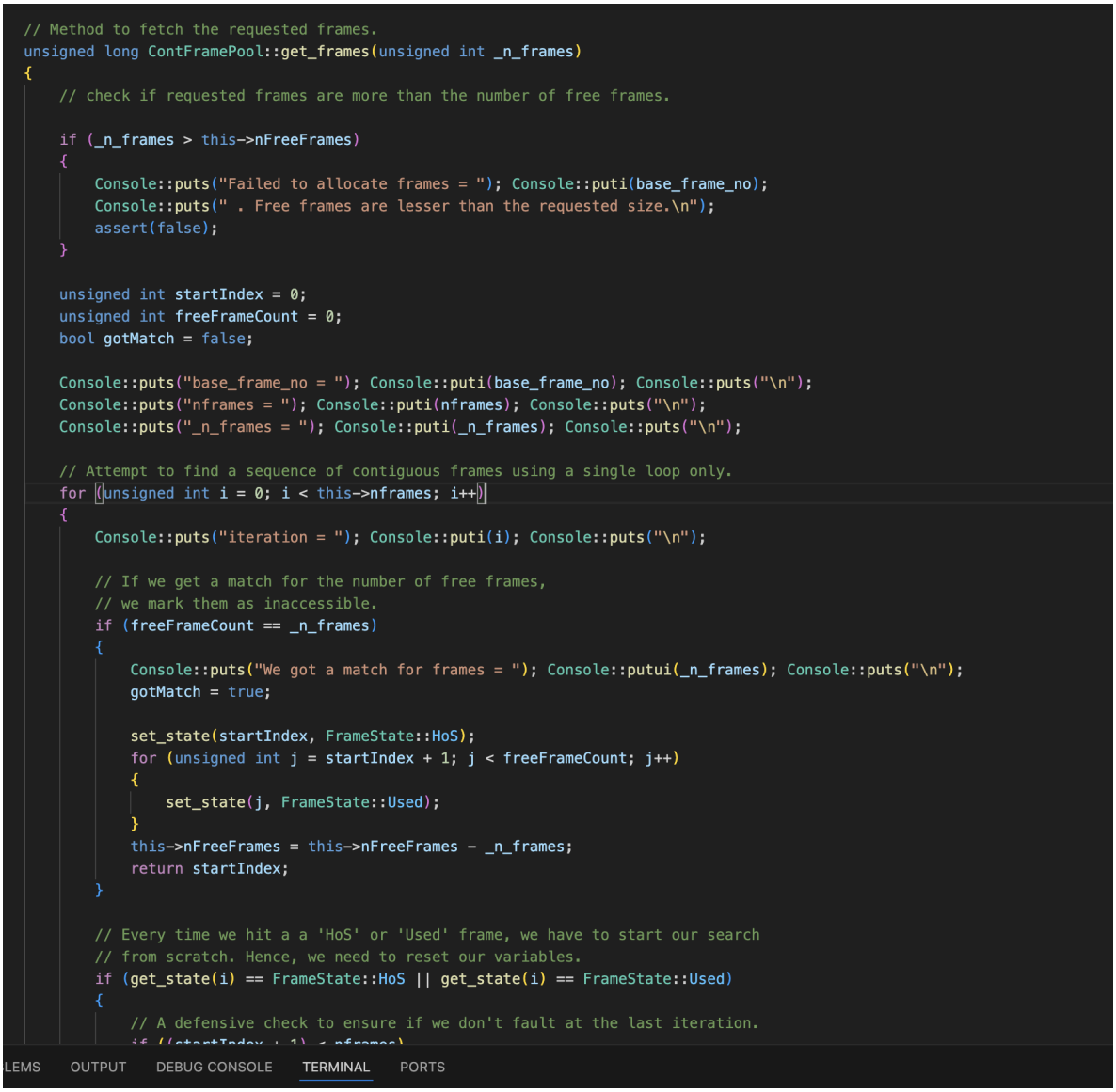
Figure 4: Get State method

cont_frame_pool.C: set_state : The *set_state* method sets the state of a given char in the bit map. This is done by providing appropriate bit masks. Generally, AND with NOT is used to clear. OR is used to set.

```
// Method to set the state for a given frame no.
// Options are as follows:
// Free: 00
// Used: 01
// Hos:  11
// Hos --> Head of Sequence.
// With 2 bits, we have 4 options, but we are only making use of 3.
void ContFramePool::set_state(unsigned long _frame_no, FrameState _state)
{
    // Console::puts("Setting state now!\n");
    // Parse out the index and exact position of the frame management
    // bits within the index.
    unsigned int bitMapIndex = _frame_no / 4;
    unsigned int shift = ((_frame_no % 4) * 2);
    // Console::puts("_frame_no = "); Console::putui(_frame_no); Console::puts("\n");
    // Console::puts("bitMapIndex = "); Console::putui(bitMapIndex); Console::puts("\n");
    // Console::puts("shift = "); Console::putui(shift); Console::puts("\n");

    // Console::puts("bitmap value before = "); Console::puti(bitMap[bitMapIndex]); Console::puts("\n");
    Console::puts("setting state for frame no = "); Console::putui(_frame_no); Console::puts("\n");
    switch (_state)
    {
        case FrameState::HoS:
            Console::puts("Setting state to HoS\n");
            bitMap[bitMapIndex] |= (0x03 << shift);
            break;

        case FrameState::Used:
            Console::puts("Setting state to used\n");
            bitMap[bitMapIndex] ^= (0x01 << shift);
            break;

        case FrameState::Free:
            Console::puts("Setting state to free\n");
            bitMap[bitMapIndex] &= ~(0x03 << shift);
            break;
    }

    // Console::puts("bit map value after = "); Console::puti(bitMap[bitMapIndex]); Console::puts("\n");

} // ContFramePool::set_state
```

Figure 5: Set State method

cont_frame_pool.C: get_frames : The *get_frames* method iterates through the list of frames in the given frame pool. It starts from 0 and iterates until it hits a *HoS* or *Used* frame. The idea is to get a contiguous set of free frames before a *HoS* or *Used* frame is hit. If 'x' no of free frames are able to be allocated, then the first frame in that sequence is marked as 'HoS' and the remaining as 'Used'.



```
// Method to fetch the requested frames.
unsigned long ContFramePool::get_frames(unsigned int _n_frames)
{
    // check if requested frames are more than the number of free frames.

    if (_n_frames > this->nFreeFrames)
    {
        Console::puts("Failed to allocate frames = "); Console::puti(base_frame_no);
        Console::puts(" . Free frames are lesser than the requested size.\n");
        assert(false);
    }

    unsigned int startIndex = 0;
    unsigned int freeFrameCount = 0;
    bool gotMatch = false;

    Console::puts("base_frame_no = "); Console::puti(base_frame_no); Console::puts("\n");
    Console::puts("nframes = "); Console::puti(nframes); Console::puts("\n");
    Console::puts("_n_frames = "); Console::puti(_n_frames); Console::puts("\n");

    // Attempt to find a sequence of contiguous frames using a single loop only.
    for (unsigned int i = 0; i < this->nframes; i++)
    {
        Console::puts("iteration = "); Console::puti(i); Console::puts("\n");

        // If we get a match for the number of free frames,
        // we mark them as inaccessible.
        if (freeFrameCount == _n_frames)
        {
            Console::puts("We got a match for frames = "); Console::putui(_n_frames); Console::puts("\n");
            gotMatch = true;

            set_state(startIndex, FrameState::HoS);
            for (unsigned int j = startIndex + 1; j < freeFrameCount; j++)
            {
                set_state(j, FrameState::Used);
            }
            this->nFreeFrames = this->nFreeFrames - _n_frames;
            return startIndex;
        }

        // Every time we hit a a 'HoS' or 'Used' frame, we have to start our search
        // from scratch. Hence, we need to reset our variables.
        if (get_state(i) == FrameState::HoS || get_state(i) == FrameState::Used)
        {
            // A defensive check to ensure if we don't fault at the last iteration.
            if ((startIndex + 1) < nframes)
```

LEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS

Figure 6: Get Frames method

cont_frame_pool.C: mark_inaccessible : The *mark_inaccessible* sets as a given set of frames as 'Used'. The first frame in the sequence is marked as 'HoS'. It is typically used to reserve some space beforehand.

```
// Mark a series as frames as inaccessible.
// This is done by setting the 'Head of Sequence' as 'HoS'.
// The rest of the frames are marked 'Used'.
void ContFramePool::mark_inaccessible(unsigned long _base_frame_no,
                                      unsigned long _n_frames)
{
    // Limit check;
    if ((_base_frame_no < this->base_frame_no) ||
        ((this->base_frame_no + this->nframes) < (_base_frame_no + _n_frames)))
    {
        Console::puts("Out of bounds.\n");
        assert(false);
    }
    // Mark the firt frame as 'Head of Sequence'.
    set_state(_base_frame_no, FrameState::HoS);

    // Mark rest of the frames in the range as being used.
    for (int fno = _base_frame_no + 1; fno < _base_frame_no + _n_frames; fno++)
    {
        set_state(fno, FrameState::Used);
    }

} // ContFramePool::mark_inaccessible
```

Figure 7: Mark Inaccessible method

cont_frame_pool.C: release_frames : The *release_frames* releases a set of frames from the given frame pool. It typically starts from the given *_frame_no* as input and releases all the frames until a 'HoS' is hit.

```
// Method to release a set of frames back to the frame pool.
void ContFramePool::release_frames(unsigned long _first_frame_no)
{
    ContFramePool *node = headPool;
    do
    {
        unsigned int frameValue = _first_frame_no - node->base_frame_no;
        if ((_first_frame_no < node->base_frame_no) ||
            (_first_frame_no > (node->base_frame_no + node->nframes)))
        {
            Console::puts("Requested frame does not belong to this pool! Check next.\n");
            continue;
        }
        // some basic checks.
        if (node->get_state(frameValue) != FrameState::HoS)
        {
            Console::puts("Incorrect release request!\n");
            assert(false);
        }

        unsigned int index = frameValue + 1;
        while (node->get_state(index) == FrameState::Used)
        {
            node->set_state(index, FrameState::Free);
            index++;
        }

    } while (node->nextPool != NULL);

} // ContFramePool::release_frames
```

Figure 8: Release frames method

cont_frame_pool.C: needed_info_frames : The *needed_info_frames* just returns the total numbers of management frames required to manage a frame pool. This calculation is based on the page size converted to bits. If the required number of frames is returned as a decimal value, then the value is

rounded up before returning.

```
// Method to calculate the number of frames required to manage a
// frame pool of size '_n_frames'.
unsigned long ContFramePool::needed_info_frames(unsigned long _n_frames)
{
    // Since we use 2 bits per frame.
    unsigned long bitsUsed = (_n_frames * 2);
    // 4096 --> 4kB (1024 * 4)
    // multiply by 8 since we need the value in bits.
    unsigned long frameSize = (4096 * 8);

    // divide and get the quotient.
    // Add the remainder if any.
    return ((bitsUsed / frameSize) + (bitsUsed % frameSize) > 0 ? 1 : 0);

} // ContFramePool::needed_info_frames
```

Figure 9: Needed Info method

Testing

Testing was performed using the functionality provided in *kernel.C* wherein we allocate a kernel frame pool. As per the given method we have up to 32 allocations in random order. All the test cases have passed as per the basic test functionality.

```
QEMU
Requested frame does not belong to this pool! Check next.
Requested frame does not belong to this pool! Check next.
Requested frame does not belong to this pool! Check next.
Requested frame does not belong to this pool! Check next.
Requested frame does not belong to this pool! Check next.
Requested frame does not belong to this pool! Check next.
Requested frame does not belong to this pool! Check next.
Requested frame does not belong to this pool! Check next.
Requested frame does not belong to this pool! Check next.
Requested frame does not belong to this pool! Check next.
Requested frame does not belong to this pool! Check next.
Requested frame does not belong to this pool! Check next.
Requested frame does not belong to this pool! Check next.
Requested frame does not belong to this pool! Check next.
Requested frame does not belong to this pool! Check next.
Requested frame does not belong to this pool! Check next.
Requested frame does not belong to this pool! Check next.
Requested frame does not belong to this pool! Check next.
Requested frame does not belong to this pool! Check next.
Requested frame does not belong to this pool! Check next.
Requested frame does not belong to this pool! Check next.
Requested frame does not belong to this pool! Check next.
Testing is DONE. We will do nothing forever
Feel free to turn off the machine now.
```

Figure 10: QEMU output

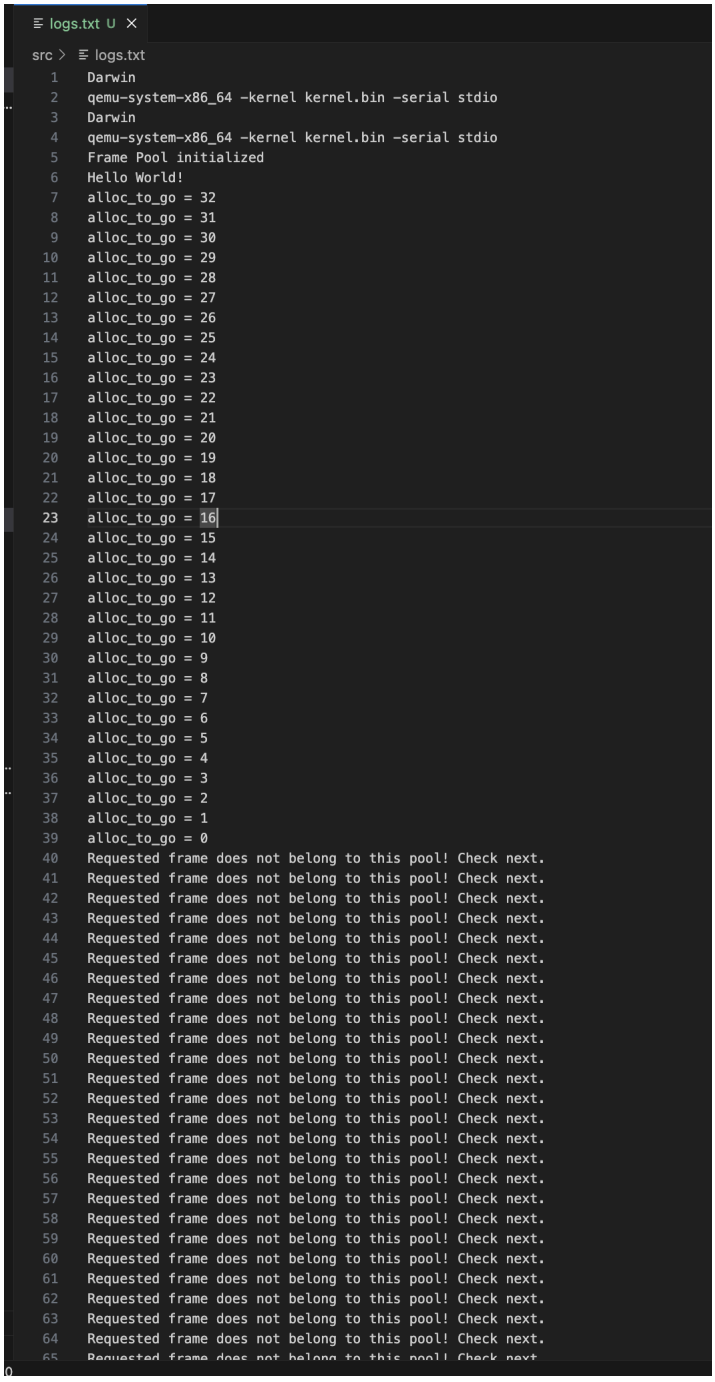


Figure 11: Console Logs

Limitations

In my opinion, we have to test the two pools simultaneously to vet the working of the frame manager. This needs to be done by commenting out the given code section where the process frame pools and the memory hole has been allocated.