MP2: Simple File System

Mohammed Sharique
UIN: 837004870
CSCE611: Operating System

02/11/2026

Assigned Tasks

Main: Completed.

System Design

The goal of this machine problem was to build a frame manager and mainly a contiguous frame pool manager that can allocate and de-allocate a contiguous sequence of frames of requested length. We'll be using this in the future to manage memory in our machine.

- My implementation manages a doubly linked list of frame pools that is sorted in increasing order of their base frame addresses. Every time a new contiguous frame pool is created it is added into the list accordingly. I am not managing deletion of a frame pool for now.
- Every frame within a frame pool can be in one of the following three states Free (denoting it is free for allocation), Used (denoting it is allocated) or Head-Of-Sequence (denoting that it is the first frame of an allocated contiguous memory segment).
- The states of each frame of a frame pool is stored in a bitmap. The user has the option to either store this bitmap in a frame outside of the frame pool or we reserve the first frame of the memory segment to store the bitmap.
- When a frame pool gets a request to provide n contiguous frames. It first goes through its bitmap to check if it has n contiguous frames available. If it does, it marks the start of the first frame as the Head-of-Sequence (HoS) and sets the state of all the other n-1 frames as Used and returns the frame address of the first pool otherwise it returns with an error code denoting that contiguous memory of requested length is not available.
- To release a previously allocated memory segment. In the request user provides the frame address of the first frame of the sequence. Using this frame address we traverse through the doubly linked list of the frame pools to determine which frame pool does this frame belongs to. After finding the frame pool of this frame we check if the state of the frame is marked as Head-of-Sequence, if not we let the user now by displaying it in the console. If the state is Head-of-Sequence we mark this frame as Free and we mark all the contiguous frames next to this frame in state Used as Free i.e we stop when we encounter a frame in state Free or Head-Of-Sequence.

Code Description

I changed cont_frame_pool.C, cont_frame_pool.H and kernel.C (to add tests).
Github: https://github.com/sharique-tamu/csce-611-machine-problems
The code should compile by just running make in the folder containing the code.

cont_frame_pool.C: ContFramePool(constructor) : This constructor initializes a new contiguous frame pool, sets up its bitmap, and inserts the pool into a global doubly linked list sorted by increasing base frame number. First, it asserts that the bitmap can fit within a single frame, since each frame requires 2 bits to represent its state. It then initializes the member variables and sets the prev and next pointers to nullptr. The constructor inserts the new pool into the doubly linked list at the correct position to maintain sorted order. If the list is empty, it becomes the head, otherwise it is inserted before or after the appropriate node and the pointers are updated. Next, it determines where to store the bitmap, either in an external frame (info_frame_no) or in the first frame of the pool. All frames are initially marked as Free. If the bitmap is stored in the first frame of the pool, that frame is marked as Head-Of-Sequence to reserve it. Finally, it prints a message indicating that the frame pool has been initialized

```
ContFramePool::ContFramePool(unsigned long _base_frame_no,
                             unsigned long _n_frames,
                             unsigned long _info_frame_no) {
  // Bitmap must fit in a single frame!
  // dividing by 2 as 2 bits will be used per frame
  assert(_n_frames ≤ FRAME_SIZE * 8 / 2);
  assert(needed_info_frames(_n_frames) == 1);

  base_frame_no = _base_frame_no;
  nframes = _n_frames;
  info_frame_no = _info_frame_no;
  prev = nullptr;
  next = nullptr;

  if (!head) {
    head = this;
  } else {
    ContFramePool *tmp = head;
    while (tmp->base_frame_no < base_frame_no && tmp->next) {
      tmp = tmp->next;
    }
    // insert after
    if (tmp->base_frame_no < base_frame_no) {
      prev = tmp;
      next = tmp->next;
      if (tmp->next) {
        tmp->next->prev = this;
      }
      tmp->next = this;
    } else {
      // insert before
      next = tmp;
      if (tmp->prev) {
        prev = tmp->prev;
        tmp->prev->next = this;
        tmp->prev = this;
      } else {
        // inserting before head
        tmp->prev = this;
        head = this;
      }
    }
  }
```

cont_frame_pool.C: get_state : This function returns the state of a given frame by reading its 2-bit entry from the bitmap. Since each byte stores the states of 4 frames, it computes the byte index using _frame_no / 4 and the bit offset using 2 * (_frame_no % 4). It then right-shifts the byte and masks with 0x3 (11 in binary) to extract the 2-bit value, which is mapped to Free, Used, or HoS. If no valid value is found, it defaults to Free.

```
  if (info_frame_no == 0) {
    bitmap = (unsigned char *)(base_frame_no * FRAME_SIZE);
  } else {
    bitmap = (unsigned char *)(info_frame_no * FRAME_SIZE);
  }

  for (int fno = 0; fno < _n_frames; fno++) {
    set_state(fno, FrameState::Free);
  }

  if (_info_frame_no == 0) {
    set_state(0, FrameState::HoS);
  }

  Console::puts("Frame Pool initialized\n");
}
```

```
ContFramePool::FrameState ContFramePool::get_state(unsigned long _frame_no) {
  unsigned int bitmap_index = _frame_no / 4;
  unsigned int shift = 2 * (_frame_no % 4);
  unsigned char current_state = (bitmap[bitmap_index] >> shift) & 0x3;
  switch (current_state) {
  case 0:
    return FrameState::Free;
  case 1:
    return FrameState::Used;
  case 2:
    return FrameState::HoS;
  }
  return FrameState::Free;
}
```

cont_frame_pool.C: set_state : This function sets the state of a given frame in the bitmap. Since each frame is represented using 2 bits and each byte stores 4 frames, it computes the byte index using _frame_no / 4 and the bit offset using 2 * (_frame_no % 4). It first clears the existing 2 bits at that position using a reset mask, then sets the new value based on the given FrameState: Free requires no further action (as 0 is already written), Used sets bit pattern 01, and HoS sets bit pattern 10.

```
void ContFramePool::set_state(unsigned long _frame_no, FrameState _state) {
  unsigned int bitmap_index = _frame_no / 4;
  unsigned int shift = 2 * (_frame_no % 4);
  unsigned char reset_mask = ~(0x3 << shift);

  bitmap[bitmap_index] &= reset_mask;
  switch (_state) {
  case FrameState::Free:
    // As I have already set 0 using the reset mask, nothing more is need to set
    // it as free (enum value = 0).
    break;
  case FrameState::Used:
    bitmap[bitmap_index] |= 0x1 << shift;
    break;
  case FrameState::HoS:
    bitmap[bitmap_index] |= 0x2 << shift;
    break;
  }
}
```

cont_frame_pool.C: get_frames : This function searches the frame pool for _n_frames contiguous free frames and returns the physical frame number of the first frame if successful. It scans sequentially from start_frame, checking whether the next _n_frames frames are all in the Free state using get_state(). If a non-free frame is encountered, it skips ahead to the frame after that position and continues searching. Once a suitable contiguous block is found, it marks the frames as allocated using mark_inaccessible() and returns the corresponding physical frame number (start_frame + base_frame_no). If no such block exists, it returns 0 to indicate failure.

```
unsigned long ContFramePool::get_frames(unsigned int _n_frames) {
  unsigned int start_frame = 0;
  while (start_frame + _n_frames - 1 < nframes) {
    bool found = true;
    for (unsigned int i = 0; i < _n_frames; i++) {
      if (get_state(start_frame + i) != FrameState::Free) {
        found = false;
        start_frame = start_frame + i + 1;
        break;
      }
    }
    if (found) {
      mark_inaccessible(start_frame, _n_frames);
      return start_frame + base_frame_no;
    }
  }
  return 0;
}
```

cont_frame_pool.C: mark_inaccessible : This function marks a contiguous sequence of frames as allocated in the bitmap. It sets the first frame of the sequence to Head-Of-Sequence (HoS) to indicate the start of the allocation, and then marks the remaining _n_frames - 1 frames as Used by iteratively calling set_state() on each subsequent frame.

```
void ContFramePool::mark_inaccessible(unsigned long _base_frame_no,
                                      unsigned long _n_frames) {

  set_state(_base_frame_no, FrameState::HoS);
  for (unsigned int i = 1; i < _n_frames; i++) {
    set_state(_base_frame_no + i, FrameState::Used);
  }
}
```

cont_frame_pool.C: release_frames : This function releases a previously allocated contiguous block of frames given the physical number of its first frame. It traverses the global doubly linked list of frame pools to find which pool the frame belongs to. Once found, it computes the relative frame number and checks whether it is marked as Head-Of-Sequence (HoS). If it is, the function marks that frame as Free and continues marking subsequent frames as Free as long as they are in the Used state, stopping when it encounters a Free or another HoS frame. If the given frame is not a HoS, it prints an error message indicating an invalid release request.

```
void ContFramePool::release_frames(unsigned long _first_frame_no) {
  ContFramePool *tmp = head;
  while (tmp && tmp->base_frame_no <= _first_frame_no) {
    if (_first_frame_no <= tmp->base_frame_no + tmp->nframes - 1) {
      unsigned int rel_frame_no = _first_frame_no - tmp->base_frame_no;
      if (tmp->get_state(rel_frame_no) == FrameState::HoS) {
        tmp->set_state(rel_frame_no, FrameState::Free);
        unsigned int fno = rel_frame_no + 1;
        while (fno < tmp->nframes && tmp->get_state(fno) == FrameState::Used) {
          tmp->set_state(fno, FrameState::Free);
          fno++;
        }
      } else {
        // Frame is not a head of sequence
        Console::puts(
            "First Frame requested to release is not a Head of Sequence\n");
      }
      break;
    }
    tmp = tmp->next;
  }
}
```

cont_frame_pool.C: needed_info_frames : This function computes how many frames are required to store the bitmap for _n_frames. Since each frame requires 2 bits in the bitmap, it first calculates the total number of bits needed as _n_frames * 2. It then divides this by the number of bits available in one frame (FRAME_SIZE * 8), rounding up using ceiling division to ensure enough space is allocated. The result is the number of information frames required to store the bitmap.

```
unsigned long ContFramePool::needed_info_frames(unsigned long _n_frames) {
  unsigned int bits_required = _n_frames * 2;
  unsigned int info_frames =
      (bits_required + (FRAME_SIZE * 8) - 1) / (FRAME_SIZE * 8);
  return info_frames;
}
```

Testing

I explicitly ran the bit logic manually in an online C compiler to test if it is working perfectly or not. I add three more test functions to increase the test coverage. Except the error cases all the code is covered by the tests present. The preexisting test does covered the majority usecase of creating the poolframe requesting memory segments within it, writing on the memory segments and at the end verifying if no other frames are writing over it and releasing the frames at the end. I added three more tests:

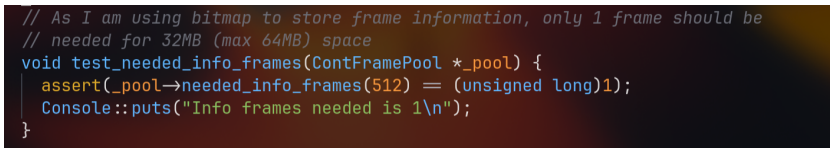
kernel.C: test_max_space : This function tests whether the frame pool can correctly allocate and manage the maximum available contiguous memory. It allocates all usable frames in the pool (except the one reserved for the bitmap), writes known integer values into the entire allocated memory region, and then reads them back to verify that no memory corruption has occurred. If any value differs from what was written, it reports a failure and halts execution. Finally, it releases the allocated frames and prints a message confirming that all frames have been freed.

```
void test_max_space(ContFramePool *_pool, unsigned int max_frames) {
  // Leaving out 1st frame that is used for storing the bitmap.
  unsigned long n_frames = max_frames - 1;
  unsigned long frame =
      _pool->get_frames(n_frames); // we allocate the n_frames from the pool

  Console::puts("All frames allocated.\n");
  int *value_array =
      (int *)(frame * (4 KB)); // we pick a unique number that we want to
                               // write into the memory we just allocated
  for (int i = 0; i < (1 KB) * n_frames;
       i++) { // we write this value int the memory locations
    value_array[i] = i;
  }
  for (int i = 0; i < (1 KB) * n_frames;
       i++) { // We check the values written into the memory before we
              // recursed
    if (value_array[i] != i) { // If the value stored in the memory locations
                               // is not the same that we wrote a few lines
                               // above then somebody overwrote the memory.
      Console::puts("MEMORY TEST FAILED. ERROR IN FRAME POOL\n");
      Console::puts("i =");
      Console::puti(i);
      Console::puts("   v = ");
      Console::puti(value_array[i]);
      Console::puts("   n =");
      Console::puti(i);
      Console::puts("\n");
      for (;;)
        ; // We throw a fit.
    }
  }
  ContFramePool::release_frames(
      frame); // We free the memory that we allocated above.
  Console::puts("All frames freed.\n");
}
```

kernel.C: test_multiple_allocations_and_contiguous_mem : This function tests whether the frame pool correctly handles multiple allocations while preserving contiguous memory and preventing memory corruption. It recursively allocates memory in chunks (up to 10 frames at a time) until the requested number of frames is exhausted, writing unique sequential values into each allocated block. After the recursive allocations complete, it verifies that the previously written values remain unchanged, ensuring that subsequent allocations did not overwrite earlier ones. If any mismatch is detected, it reports a memory error and halts execution.

```
void test_multiple_allocations_and_contiguous_mem(ContFramePool *_pool, unsigned int rem_frames, unsigned long &start) {
  Console::puts("alloc_to_go = ");
  Console::puti(rem_frames);
  Console::puts("\n");
  if (rem_frames == 0)
    return;
  unsigned int n_frames;
  unsigned int frame;
  unsigned long cur_start = start;
  if (rem_frames < 10) {
    frame = _pool->get_frames(rem_frames);
    n_frames = rem_frames;
  } else {
    frame = _pool->get_frames(10);
    n_frames = 10;
  }

  unsigned int *value_array = (unsigned int *)(frame * (4 KB));     ■ Cast to 'unsigned int *' from smaller integer type '
  for (int j = 0; j < (1 KB) * n_frames; j++) {
    value_array[j] = start++;
  }
  test_multiple_allocations_and_contiguous_mem(_pool, rem_frames - n_frames,
                                               start);
  for (int i = 0; i < (1 KB) * n_frames;
       i++) { // We check the values written into the memory before we
              // recursed
    if (value_array[i] !=
        cur_start++) { // If the value stored in the memory locations is not
                       // the same that we wrote a few lines above then
                       // somebody overwrote the memory.
      Console::puts("MEMORY TEST FAILED. ERROR IN FRAME POOL\n");
      Console::puts("i =");
      Console::puti(i);
      Console::puts("   v = ");
      Console::puti(value_array[i]);
      Console::puts("   n =");
      Console::puti(cur_start - 1);
      Console::puts("\n");
      for (;;)
        ; // We throw a fit.
    }
  }
}
```

kernel.C: test_needed_info_frames : This function tests whether the bitmap size calculation is correct. It verifies that for 512 frames, only one information frame is required to store the bitmap (since each frame uses 2 bits). The function asserts that needed_info_frames(512) returns 1, and if successful, prints a confirmation message.



```
// As I am using bitmap to store frame information, only 1 frame should be
// needed for 32MB (max 64MB) space
void test_needed_info_frames(ContFramePool *_pool) {
  assert(_pool->needed_info_frames(512) == (unsigned long)1);
  Console::puts("Info frames needed is 1\n");
}
```