CS307 Project7: Contiguous Memory Allocation

Junjie Wang 517021910093

May 23, 2019

1 Programming Thoughts

First store the allocation as blocks into double link list(with head and tail), and all of our operations are done on these blocks.

1.1 Hit Strategies

Three hit strategies often used are: best/first/worst hit strategy. The first hit is the easiest to implement, where we simply move along the link list and stop once we find one block meeting our requirement. For best/worst strategy, we need to traverse the whole double link list.

1.2 Handle Memory Request

When we use one of strategies above and find a block to allocate to the process, what we need to do is split the block. After splitting the block, one part of the block will be allocated to the process, and the other part will be unused.

1.3 Handle Memory Release

When receiving the memory releasing request, we traverse the link list and free the blocks whose "name" is the same as the name given. Note that once the block becomes unused once released, and we have to check whether we can merge it with its previous block and following block.

1.4 Compaction

After a number of iterations of memory allocation and memory freeing, these may be many holes in our memory, which will reduce the memory utilization rate. Solution here is to use memory compaction strategy, which will merge all the holes into only one large hole on

one side of the memory, and place all the processes on the other side. I offer 2 different implementations in my code. One is naive and simply re-allocate a memory space whose size is equal to our original one. Then process the processes and unused memory blocks according to the original one. This is probably **not what happens in real-world scenerio** because of limitation of memory space. So I offer the second implementation, which shuffles the unused memory forward as it moves along the link list(this can be done by copying the allocated block backward, byte by byte, from start to end).

2 Execution Results And Snapshots

We first request 4000 bytes for the process P0 using best hit strategy. Then we request another 4000 bytes for process P0 using worst hit. Now P0 has 8000 bytes memory space, and once we release all the memory which belongs to P0, the expected output should be an completely unused block. As we can see in 1, the real output is correct.
Then we test the compaction logic. The idea is to request 4000 bytes for P0 first and then P1. Then release the memory which belongs to P0. As we can see in 1, the unused memory is shuffled towards one side of the memory.

```
dreamboy@Elon_Mask:~/OSProjects/project7$ ./allocator 10000
allocator>RQ P0 4000 B
allocator>RQ P0 4000 W
allocator>STAT
Addresses       [0: 4000]       Process P0
Addresses       [4000: 8000]    Process P0
Addresses       [8000: 10000]   Unused
allocator>RL P0
allocator>STAT
Addresses       [0: 10000]      Unused
allocator>RQ P0 4000 F
allocator>RQ P1 4000 B
allocator>STAT
Addresses       [0: 4000]       Process P0
Addresses       [4000: 8000]    Process P1
Addresses       [8000: 10000]   Unused
allocator>RL P0
allocator>STAT
Addresses       [0: 4000]       Unused
Addresses       [4000: 8000]    Process P1
Addresses       [8000: 10000]   Unused
allocator>C
allocator>STAT
Addresses       [0: 6000]       Unused
Addresses       [6000: 10000]   Process P1
allocator>
```

Figure 1: first

3 Code Explanation

First we define some data structures which will be used to represent the memory block and form the link list. Listed as follows:

```
/* define the struct to represent the allocation and hole */
typedef struct{
   /* type 1: allocation
   * type 0: hole(namely unused) */
   int type;
   int start;
   int end;
   /* the process id */
   char *name;
} Block;

/* use double link list */
struct node{
   Block *block;
   struct node *prev;
   struct node *next;
};
```

Then we show the code of splitting the node(which will be used in handling memory requests)

```
void splitNode(int size, char *name, struct node *oldNode){
   block = oldNode -> block;
   if(block -> end - block -> start == size){
   /* if we have size == end - start, that's perfect */
   strcpy(block->name, name);
   block->type = 1;
   }else{
   /* else we have to split the node */
   tmp_block = malloc(sizeof(Block));
   newNode = malloc(sizeof(struct node));
   tmp_block -> start = block -> start;
   tmp_block -> end = block -> start + size;
   block -> start = block -> start + size;
   /* since the first one is the allocation
   * its type = 1 */
   tmp_block -> type = 1;
   /* if use strcpy() function,
   * don't forget tmp_block -> name = malloc(...) */
   tmp_block -> name = name;
   newNode -> block = tmp_block;
   oldNode -> prev -> next = newNode;
   newNode -> prev = oldNode -> prev;
   newNode -> next = oldNode;
```

```
    oldNode -> prev = newNode;
    }
}
```

Then we show the logic to merge holes(used when releasing memory space)

```
/* check whether we can merge the node with
* its previous one and its following one */
void mergeHole(struct node *target){
    newNode = malloc(sizeof(struct node));
    block = malloc(sizeof(Block));
    if(target->prev != head && target->prev->block->type == 0){
    /* merge current one with previous one */
    block -> start = target -> prev -> block -> start;
    block -> end = target -> block -> end;
    block -> type = 0;
    newNode -> block = block;
    target -> prev -> prev -> next = newNode;
    newNode -> prev = target -> prev -> prev;
    newNode -> next = target -> next;
    target -> next -> prev = newNode;
    target = NULL;
    }
    if(target == NULL) { target = newNode; newNode = malloc(sizeof(struct
        node)); }

    if(target->next != tail && target->next->block->type == 0){
    /* merge current one with following one */
    block -> start = target -> block -> start;
    block -> end = target -> next -> block -> end;
    block -> type = 0;
    newNode -> block = block;
    target -> next -> next -> prev = newNode;
    newNode -> next = target -> next -> next;
    target -> prev -> next = newNode;
    newNode -> prev = target -> prev;
    }
}
```

At last we show the logic of compaction(the **shuffle** method).

```
/* As we traverse the link list, whenever we meet
* type 0 block, we shuffle that forward. Finally we
* will have all the holes on one side of memory. */
void handleCompactShuffle(){
    struct node *p = head->next, *prev;
    while(p != tail){
        /* only when we meet unused */
```

```
        if(p->block->type == 0 && p->prev != head){
        /* since we merge the holes when
        * allocation, the previous one must
        * be allocated and we can just exchange */
        prev = p->prev;
        /* actually this operation may be implemented as memory copy */
        prev->block->start += (p->block->end - p->block->start);
        prev->block->end += (p->block->end - p->block->start);
        p->block->start -= (prev->block->end - prev->block->start);
        p->block->end -= (prev->block->end - prev->block->start);
        /* now exchange these two */
        prev->prev->next = p;
        p->prev = prev->prev;
        p->next->prev = prev;
        prev->next = p->next;
        p -> next = prev;
        /* merge the the holes */
        mergeHole(p);
        /* as we shift p ahead , add one more p -> next step */
        p = p->next ;
        }
    p = p->next;
    }
}
```