EI338 Computer Systems Engineering

Project 7 Contiguous Memory Allocation

November 30, 2019

Zhihui Xie, 517030910356

The environment used in this project is **Deepin 15.11**, the latest version of an open source operating system based on Debian's stable branch. The kernel version is **Linux version 4.15.0**.

Exercise

In this exercise, we will manage a contiguous region of memory. Our program is able to respond to four different requests:

1. Request for a contiguous block of memory. This is associated with command "RQ".
2. Release of a contiguous block of memory. This is associated with command "RL".
3. Compact unused holes of memory into one single block. This is associated with command "C".
4. Report the regions of free and allocated memory. This is associated with command "STAT".

Basic Structure

First of all, we will assign some amount of initial memory. Once your program has started, it will present the user with the prompt "allocator>". It will then respond to the following commands: RQ (request), RL (release), C(compact), STAT (status report), and X (exit).

Implementation

Since a lot of previous projects includes parts about how to manage commands, we skip it here to save space. Apart from that, three main functions are implemented:

- *request_memory()*, which is used to request some amount of memory using some specific allocation strategy.
- *release_memory()*, which is used to release some amount of memory occupied by some specific process.
- *compact()*, which is used to compact unused holes of memory into one region.

Contiguous memory is realized by a linked list whose data include the low address, the high address, and the occupying process ("NULL" if unused).

request_memory()

request_memory() allocates memory using one of the three approaches depending on the flag that is passed to the RQ command:

- "F" for first fit
- "B" for best fit
- "W" for worst fit

First Fit For the first-fit strategy, we only need to find the first hole.

```
hole_prev = head;

while (hole_prev->next != NULL) {
    if (!strcmp(hole_prev->next->process, "NULL")
        && hole_prev->next->high - hole_prev->next->low >= space)
        break;

    hole_prev = hole_prev->next;
}
```

Listing 1: Finding the Hole with First-fit Strategy

Best Fit & Worst Fit These two strategies are very similar. We will find the smallest/largest hole to allocate memory.

```
Node *tmp = head;
int best = INT_MAX;

while (tmp->next != NULL) {
    if (!strcmp(tmp->next->process, "NULL")
        && tmp->next->high - tmp->next->low >= space
        && tmp->next->high - tmp->next->low < best) {
        hole_prev = tmp;
        best = tmp->next->high - tmp->next->low;
    }

    tmp = tmp->next;
}
```

Listing 2: Finding the Hole with Best-fit Strategy

```
Node *tmp = head;
int worst = INT_MIN;

while (tmp->next != NULL) {
    if (!strcmp(tmp->next->process, "NULL")
        && tmp->next->high - tmp->next->low >= space
        && tmp->next->high - tmp->next->low > worst) {
        hole_prev = tmp;
        worst = tmp->next->high - tmp->next->low;
    }

    tmp = tmp->next;
}
```

Listing 3: Finding the Hole with Worst-fit Strategy

After finding which hole to allocate memory in, we need to allocate memory in it. This would involve separating it into two nodes and then linking them.

```
// can not find unused memory
if (hole_prev->next == NULL)
    return -1;

// allocate memory
```

```
else {
    Node *allocated = malloc(sizeof(Node));
    Node *remain = malloc(sizeof(Node));

    allocated->low = hole_prev->high;
    allocated->high = allocated->low + space;
    strcpy(allocated->process, process);

    remain->low = allocated->high;
    remain->high = hole_prev->next->high;
    strcpy(remain->process, "NULL");

    allocated->next = remain;
    remain->next = hole_prev->next->next;

    free(hole_prev->next);
    hole_prev->next = allocated;

    return 0;
}
```

Listing 4: Allocating Memory in the Hole

release_memory()

To release memory used by a process, we can simply traverse the linked list and set all occupied memory nodes' member variable **process** to "NULL".

```
Node *partition = head->next;

// find the allocated partition and then release
while (partition != NULL) {
    if (!strcmp(partition->process, process))
        strcpy(partition->process, "NULL");

    partition = partition->next;
}
```

Listing 5: Finding the Occupied Memory

Besides, we merge unused memory nodes to keep our linked list correct.

```
// merge unused memory
Node *tmp_prev = head;

while (tmp_prev->next != NULL) {
    if (!strcmp(tmp_prev->next->process, "NULL")) {
        Node *to_merge_tmp = tmp_prev->next;
        Node *to_delete;
        Node *merge_next = to_merge_tmp->next;

        while (merge_next != NULL && !strcmp(merge_next->process, "NULL")) {
            to_merge_tmp->high += merge_next->high - merge_next->low;
            to_delete = merge_next;
            merge_next = merge_next->next;
```

```
            free(to_delete);
        }

        to_merge_tmp->next = merge_next;
    }

    tmp_prev = tmp_prev->next;
}

// always success
return 0;
```

Listing 6: Merging the Unused Memory

compact()

compact() shuffles the memory contents so as to place all free memory together in one large block. Here we apply the simplest compaction algorithm, moving all holes toward one end of memory. We traverse the linked list, calculate the total unused memory, and update addresses that have been affected by compaction.

```
Node *tmp_prev = head;
Node *to_delete;
int compact_space = 0;

while (tmp_prev->next != NULL) {
    if (!strcmp(tmp_prev->next->process, "NULL")) {
        compact_space += tmp_prev->next->high - tmp_prev->next->low;

        // adjust
        to_delete = tmp_prev->next;
        tmp_prev->next = tmp_prev->next->next;

        if (tmp_prev->next == NULL)
            break;

        free(to_delete);
    }

    if (tmp_prev != NULL) {
        tmp_prev->next->high -= tmp_prev->next->low - tmp_prev->high;
        tmp_prev->next->low = tmp_prev->high;
    }

    else {
        tmp_prev->next->high -= tmp_prev->next->low;
        tmp_prev->next->low = 0;
    }

    tmp_prev = tmp_prev->next;
}
```

Listing 7: Traversing the Linked List

Finally, a merged block of free memory is then added to the end of the linked list.

```
// move toward one end
```

```
Node *new = malloc(sizeof(Node));

new->low = tmp_prev->high;
new->high = new->low + compact_space;
strcpy(new->process, "NULL");
new->next = NULL;

tmp_prev->next = new;
```

Listing 8: Moving Holes Toward One End

Result

To test whether out program is correct, we perform a list of commands check its correctness. The command file is as follows:

```
RQ P0 100 W
RQ P1 50 W
RQ P2 300 W
RQ P3 300 W
STAT
RL P1
RL P3
RQ P4 30 W
RQ P5 20 B
STAT
RL P0
RQ P6 50 F
RQ P7 300 F
STAT
C
STAT
X
```

Listing 9: command.txt

The result of an experiment redirecting file **command.txt** as our input varifies the correctness.

Figure 1: Result