OS - PC2: Contiguous Memory Allocator

Eduardo Sebastián Castro Quispe
Abril Vento Bustamante
David Soto Aguirre

July 2021

1 Explanation

Memory management is an important aspect to consider during the implementation of an operating system. Since main memory is expected to accommodate both the operating system and user processes, one must come up with a way to allocate this space efficiently.

Such a method is the use of **contiguous memory allocation**. According to Silberschatz et al. (2021), in this method, "each process is contained in a single section of memory that is contiguous to the section containing the next process". To do this, it considers aspects from both *memory protection* (having a relocation register and a limit register) as well as *memory allocation* (having the capacity to assign processes to memory partitions in which they fit).

Going more in-depth with this second aspect, the simplest approach is the use of a **variable-partition scheme**. This scheme requires a way to keep track of which parts of memory are available and which are not to "assign processes to variably sized partitions in memory, where each partition may contain exactly one process" (Silbershatz, et al., 2021).

According to Tanenbaum (2014), one way to do so is to **use a double-linked list sorted by addresses** which has entries that represent either a hole or a process, indicating the addresses in which they start, their length and a pointer to the next and previous item. Another option proposed by Tanenbaum (2014), is keeping two separate lists: one for processes and another one for holes, where the latter is sorted on size. This allows for a slight optimization regarding the placement algorithms used for the dynamic storage-allocation problem (or "knowing where to place a request of size n from a list of free holes", according to Silberschatz et al. (2021)), but comes at the cost of a higher complexity and slowdown when deallocating memory (Tanenbaum, 2014).

Because of this drawback, the first approach was picked for this implementation.

```
int LOWEST_ADDRESS = 0;
int HIGHEST_ADDRESS = 256;
int BYTES_PER_ADDRESS = 4;
int INITIAL_ADDRESS = 0;

struct Node {
        bool        available; // true: hole, false: process
        int        address;
        int        length;
        struct Node*        next;
        struct Node*        prev;
};

struct ContiguousMemory {
        int capacity;
        struct Node* head;
        int lowest;
        int highest;
};

struct Node* get_node(bool type, int len) {
        struct Node* n = (struct Node*)malloc(sizeof(struct Node));
        n->available = type;
        n->length = len;
        n->next = NULL;
        n->prev = NULL;
        return n;
}

struct ContiguousMemory* init_mem(int MEMORY_SIZE) {
        struct ContiguousMemory* mem = (struct ContiguousMemory*)
                                        malloc(sizeof(struct ContiguousMemory));
        mem->capacity = MEMORY_SIZE;
        mem->lowest = LOWEST_ADDRESS;
        mem->highest = HIGHEST_ADDRESS;
        mem->head = get_node(true, mem->capacity);
        (mem->head)->address = INITIAL_ADDRESS;
        printf("%s", "Contiguous memory created with capacity: ");
        printf("%d", mem->capacity);
        printf("%s", "\n\n");
        return mem;
}
```

When it comes to placing new processes, the core steps to follow are: searching for a hole that is large enough for the process, once it is found, split it into two pieces – one for the process, one for the unused memory. Since there is usually more than one free block of memory, one must have a decision criterion to know where to place an incoming process. This is the reason why there are a variety of **placement algorithms** available.

Silberschatz, et al. (2021) mention the following three:

- **Best fit:** allocate the hole closest in size to the request so that the partition with unused memory is of minimal size. For the chosen implementation, this requires iterating the entire list. Were the second option been selected, it would only require iterating the list of holes until a block that is larger than the process is found.
- **Worst fit:** allocate the largest hole. For the chosen implementation, this requires iterating the entire list. Were the second option be selected, it would only have required picking the last element in the list, as it would be the largest of all holes.
- **First fit:** starting from the beginning of memory, scan until a free hole that is large enough is found.

Both Stallings (2014) and Tanenbaum (2014) mention **next-fit**. Next fit works similarly to first fit, except it keeps track of the last placement and scans memory from the location of the last suitable hole on.

Tanenbaum (2014) mentions one last allocation algorithm: **quick fit**, which considers one more separate list on top of the one for holes and the one for processes: one that maintains the more common sizes requested. While being highly effective for allocation, it suffers heavily when having to merge hole sizes during deallocation.

Given these trade-offs and the design that was picked for the implementation, it was decided that the design to implement would work with first fit.

```
void first_fit(struct ContiguousMemory* mem, struct Node* elem) {
        struct Node* curr;
        struct Node** head_p = &(mem->head);

        if (elem->length > mem->capacity) {
                printf("%s", "ERROR: process is too big for memory.\n\n");
                return;
        }

        if (elem->length <= (*head_p)->length // it fits
```

```
&& (*head_p)->available == true // it's a hole
) {

        //split node
        (*head_p)->length = (*head_p)->length - elem->length;
        elem->address = (*head_p)->address;
        (*head_p)->address = (*head_p)->address
                             + (elem->length/BYTES_PER_ADDRESS);

        //insert node
        elem->next = *head_p;
        elem->next->prev = elem;
        *head_p = elem;

        printf("%s", "Process successfully inserted in head!\n");
}

else {
        curr = *head_p;

        while(curr->next != NULL // end has not been reached
        ) {
                if(curr->available == true // if a hole is found
                && elem->length <= curr->length //and the process fits
                ) {
                        break; // choose this hole
                }
                curr = curr->next;
        }

        if (curr->next == NULL && curr->available == false) {
                printf("%s",
                       "ERROR: No space left for a new process in memory.\n\n");
                return;
        }

        //split
        curr->length = curr->length - elem->length;
        elem->address = curr->address;
        curr->address = curr->address + (elem->length/BYTES_PER_ADDRESS);

        //insert node
        elem->next = curr->next;

        if (curr->next != NULL)
                elem->next->prev = elem;
```

```
                curr->next = elem;
                elem->prev = curr;

                merge_sort(mem->head);

                printf("%s", "Process successfully inserted!\n");
        }
}
```

> Note that a merge sort is used here to maintain the list sorted by addresses. The implementation of said merge sort was modified from **this webpage**https://www.geeksforgeeks.org/merge-sort-for-doubly-linked-list/.

When there is not enough space for arriving processes to be located, there are two main available options (as proposed by Silberschatz, et al. (2021): 1. Reject and display an error message. 2. Place the process in a wait queue which is checked every time memory is released to determine if the memory demands of a waiting process are already met.

Given the textbook does not focus on this aspect, the first option was picked for this implementation, as it is shown in this specific section of the first fit function:

```
if (curr->next == NULL && curr->available == false) {
                        printf("%s",
                        "ERROR: No space left for a new process in memory.\n\n");
                        return;
                }
```

To make space for arriving processes, a way to free memory once processes are terminated or swapped out and organize these free segments must be implemented. Considering a terminating process has two neighbors, Tatenbaum (2014) considers four scenarios when it comes to this: 1. A process between two processes is terminated. 2. A process whose left neighbor is a process and right neighbor is a hole is terminated. 3. A process whose right neighbor is a hole is terminated. 4. A process between two holes is terminated.

In the first scenario, there is nothing left to do. However, in the case of the latter three, **free spaces must be merged**, as a method of compaction to overcome *external fragmentation*. The way this is done is shown as follows:

```
void deallocate(struct ContiguousMemory* mem, int add) {
        struct Node* curr;
        struct Node** head_p = &(mem->head);
        int new_length;
```

```
if(add >= mem->highest || add < mem->lowest) {
        printf("%s", "ERROR: Address unavailable\n\n");
        return;
}

curr = *head_p;

while(curr->address < add
      && curr->next != NULL // end has not been reached
) {
        curr = curr->next;
}

//verification that there is a hole here
if(curr->available == true) {
        printf("%s", "ERROR: There is not a program located here.\n\n");
        return;
}

//set location as available
curr->available = true;

/*==NO MERGING IS NECESSARY==*/

if(
        //if there are no partitions
        (curr->prev == NULL
        && curr->next == NULL
) || (
        curr->prev == NULL //if node is the head
        && (curr->next)->available == false //and is followed by a process
) || (
        curr->next == NULL //if node is the tail
          && (curr->prev)->available == false //and is preceeded by a process
) || (
        //if surrounded by processes
        (curr->prev)->available == false
        && (curr->next)->available == false
        )
) {
        printf("%s", "Memory set as hole\n");
        list_print(mem->head);
        return;
}

/*==MERGING IS NECESSARY==*/
```

```
struct Node* merge_aux;

/*MERGE WITH RIGHT*/
if(
        //if previous is a process or current node is the head
        ((curr->prev)->available == false || curr->prev == NULL)
    && (curr->next)->available == true //but next one is a hole
) {
        new_length = curr->length + (curr->next)->length;
        merge_aux = curr->next;

        if(merge_aux->next != NULL) {
                (merge_aux->next)->prev = curr;
        }
        curr->next = merge_aux->next;

        free(merge_aux);
        curr->length = new_length;

        printf("%s", "Memory hole merged with right\n");
        list_print(mem->head);
        return;
}

/*MERGE WITH LEFT*/

if(
        // if next one is a process or current node is the tail
        ((curr->next)->available == false || curr->next == NULL)
        && (curr->prev)->available == true // but previous is a hole
) {
        new_length = (curr->prev)->length + curr->length;
        merge_aux = curr->prev;

        merge_aux->next = curr->next;
        (curr->next)->prev = curr->prev;

        free(curr);
        merge_aux->length = new_length;

        printf("%s", "Memory hole merged with left\n");
        list_print(mem->head);
        return;
}
```

```
        /*DOUBLE MERGE*/
        // if surrounded by holes

        new_length = (curr->prev)->length +
                                (curr->length) +
                                (curr->next)->length;

        //merge with right hole
        merge_aux = curr->next;
        if(merge_aux->next != NULL) {
                (merge_aux->next)->prev = curr;
        }
        curr->next = merge_aux->next;
        free(merge_aux);

        //merge with left hole
        merge_aux = curr->prev;
        if(curr->next != NULL) {
                (curr->next)->prev = merge_aux;
        }
        (merge_aux)->next = curr->next;
        free(curr);

        //update hole size
        merge_aux->length = new_length;

        printf("%s", "Memory holes from both sides merged\n");
        list_print(mem->head);
}
```

2 Testing

To test the beforementioned implementation, the following code was used:

```
void insert(struct ContiguousMemory* mem, int size) {
        struct Node* n = get_node(false, size);
        first_fit(mem, n);
        list_print(mem->head);
}


void test() {
        struct ContiguousMemory* mem = init_mem(256);

        //bigger than memory
        insert(mem, 512);

        //filling memory
        insert(mem, 12);
        insert(mem, 20);
        deallocate(mem, 3);
        insert(mem, 48);
        insert(mem, 108);
        deallocate(mem, 15);
        insert(mem, 100);
        deallocate(mem, 3);
        deallocate(mem, 15);
}

int main() {
        test();
        return 0;
}
```

The results are as shown:

```
Contiguous memory created with capacity: 256

ERROR: process is too big for memory.

[ Address: 0, Type: hole, Length: 256] ->

Process successfully inserted in head!
[ Address: 0, Type: process, Length: 12] ->
[ Address: 3, Type: hole, Length: 244] ->

Process successfully inserted!
```

```
[ Address: 0, Type: process, Length: 12] ->
[ Address: 3, Type: process, Length: 20] ->
[ Address: 8, Type: hole, Length: 224] ->

Memory hole merged with right
[ Address: 0, Type: process, Length: 12] ->
[ Address: 3, Type: hole, Length: 244] ->

Process successfully inserted!
[ Address: 0, Type: process, Length: 12] ->
[ Address: 3, Type: process, Length: 48] ->
[ Address: 15, Type: hole, Length: 196] ->

Process successfully inserted!
[ Address: 0, Type: process, Length: 12] ->
[ Address: 3, Type: process, Length: 48] ->
[ Address: 15, Type: process, Length: 108] ->
[ Address: 42, Type: hole, Length: 88] ->

Memory hole merged with right
[ Address: 0, Type: process, Length: 12] ->
[ Address: 3, Type: process, Length: 48] ->
[ Address: 15, Type: hole, Length: 196] ->

Process successfully inserted!
[ Address: 0, Type: process, Length: 12] ->
[ Address: 3, Type: process, Length: 48] ->
[ Address: 15, Type: process, Length: 100] ->
[ Address: 40, Type: hole, Length: 96] ->

Memory set as hole
[ Address: 0, Type: process, Length: 12] ->
[ Address: 3, Type: hole, Length: 48] ->
[ Address: 15, Type: process, Length: 100] ->
[ Address: 40, Type: hole, Length: 96] ->

Memory holes from both sides merged
[ Address: 0, Type: process, Length: 12] ->
[ Address: 3, Type: hole, Length: 244] ->
```

3 Bibliography

- Silberschatz, A., Gagne, G., & Galvin, P. B. (2021). Operating System Concepts (10th ed.). Wiley. p. 357.
- Tanenbaum, A., & Bos, H. (2014). Modern Operating Systems (4th ed.). Pearson.
- Stallings, W. (2014). Operating Systems: Internals and Design Principles (8th Edition) (8th ed.). Pearson.
- GeeksforGeeks (2021). Merge Sort for Doubly Linked List. Recovered from: `https://www.geeksforgeeks.org/merge-sort-for-doubly-linked-list/`

4 Annex

Github repository:
`https://github.com/CEduardoSQUTEC/OS-ContiguousMemoryAllocation`