Project 7: Contiguous Memory Allocation

CS307-Operating System (D), CS356-Operating System Course Design, Chentao Wu, Spring 2020.

Name: 方泓杰(Hongjie Fang) Student ID: 518030910150 Email: galaxies@sjtu.edu.cn

1 Contiguous Memory Allocation

1.1 Requirements

In Section 9.2 in textbook, we presented different algorithms for contiguous memory allocation. This project will involve managing a contiguous region of memory of size MAX where address may range from $0 \cdots (MAX - 1)$. This project requires us to respond to four different requests.

- Request for a contiguous block of memory;
- Release of a contiguous block of memory;
- Compact unused holes of memory into one single block;
- Report the regions of free and allocated memory.

The program will be passed the initial amount of memory at startup, and it will allocate memory using one of the three approaches highlighted in Section 9.2.2, depending on the flag that is passed to `RQ` commands. The flags are:

- `F`: first-fit;
- `B`: best-fit;
- `W`: worst-fit.

This will require the program keep track of the different holes representing available memory. When a request for memory arrives, it will allocate the memory from one of the available holes based on the allocation strategy, It there is insufficient memory to allocate to a request, it will output an error message and reject the request.

The program will also need to keep track of which region of memory has been allocated to which process. This is necessary to support the `STAT` command and is also needed when memory is released via the `RL` command, as the process releasing memory is passed to this command. If a partition being released is adjacent to an existing hole, be sure to combine the two holes into a single hole.

If the user enters `C` command, the program will compact the set of holes into one larger hole. There are several strategies for implementing compaction, one of which is suggested in Section 9.2.3 in textbook. Be sure to update the beginning address of any processes that have been affected by compaction.

1.2 Methods

Here are some specific methods of the contiguous memory allocator program.

- We use the linked list to keep track of every allocated regions of memory, and the rest of the memory is the holes. Since we have the size information about the memory, we can keep track of every free regions of memory according to the data we have. Moreover, the adjacent holes causing by `RL` command is automatically combined, which makes our programming easier.

- For first-fit algorithm, we examine the holes one by one, and if we found a suitable hole, then we will put the process into it.
- For best-fit algorithm and worst-fit algorithm, we examine all holes, and put the process into one suitable hole according to the type of the algorithm.
- For `STAT` command, we will traverse the linked list and print out all information about the memory allocation.
- For compaction command, we put all the allocated memory in the front contiguous region, therefore the holes are automatically compacted into a big one in the end of the memory.

1.3 Implementation

Here is the specific implementation of the contiguous memory allocator (`allocator.c`).

```c
# include <stdio.h>
# include <stdlib.h>
# include <string.h>

# define LINE_MAX 256

int memory_limit;

typedef struct memory_node {
  // belong
  char *bel;
  // begin, end
  int beg, end;
  // next
  struct memory_node * nxt;
} mem_node;

mem_node *head = NULL;


// Requested by process [process_name], size of [size], memory allocation type [type].
// If success, return 0; or return 1 with the error message printed in the screen.
int request(char *process_name, int size, char type) {
  if (size <= 0) {
    fprintf(stderr, "[Err] the process size should be positive!\n");
    return 1;
  }

  int name_len = strlen(process_name);
  if (head == NULL) {
    if (size <= memory_limit) {
      head = (mem_node *) malloc (sizeof(mem_node));
      head -> bel = (char *) malloc (sizeof(char) * (name_len + 1));
      strcpy(head -> bel, process_name);
```

```
      head -> beg = 0;
      head -> end = 0 + size - 1;
      head -> nxt = NULL;
      return 0;
    } else {
      // No enough spaces
      fprintf(stderr, "[Err] No enough spaces!\n");
      return 1;
    }
  }

  // First-fit
  if (type == 'F') {
    mem_node *p = head;
    int hole_len;

    hole_len = p -> beg - 0;
    if (size <= hole_len) {
      mem_node *tmp = head;
      head = (mem_node *) malloc (sizeof(mem_node));
      head -> bel = (char *) malloc (sizeof(char) * (name_len + 1));
      strcpy(head -> bel, process_name);
      head -> beg = 0;
      head -> end = 0 + size - 1;
      head -> nxt = tmp;
      return 0;
    }

    while (p -> nxt != NULL) {
      hole_len = p -> nxt -> beg - p -> end - 1;
      if (size <= hole_len) {
        mem_node *tmp = p -> nxt;
        p -> nxt = (mem_node *) malloc (sizeof(mem_node));
        p -> nxt -> bel = (char *) malloc (sizeof(char) * (name_len + 1));
        strcpy(p -> nxt -> bel, process_name);
        p -> nxt -> beg = p -> end + 1;
        p -> nxt -> end = p -> nxt -> beg + size - 1;
        p -> nxt -> nxt = tmp;
        return 0;
      }
      p = p -> nxt;
    }

    hole_len = memory_limit - p -> end - 1;
    if (size <= hole_len) {
      p -> nxt = (mem_node *) malloc (sizeof(mem_node));
      p -> nxt -> bel = (char *) malloc (sizeof(char) * (name_len + 1));
      strcpy(p -> nxt -> bel, process_name);
      p -> nxt -> beg = p -> end + 1;
      p -> nxt -> end = p -> nxt -> beg + size - 1;
```

```
      p -> nxt -> nxt = NULL;
      return 0;
    }

    // No enough spaces
    fprintf(stderr, "[Err] No enough spaces!\n");
    return 1;
  }

  // Best-fit
  if (type == 'B') {
    mem_node *p = head;
    int hole_len;

    // position
    int cur_min = 2e9;
    int pos_flag = 0;
    mem_node *pos;

    hole_len = p -> beg - 0;
    if (size <= hole_len && hole_len < cur_min) {
      cur_min = hole_len;
      pos_flag = 1;
    }

    while (p -> nxt != NULL) {
      hole_len = p -> nxt -> beg - p -> end - 1;
      if (size <= hole_len && hole_len < cur_min) {
        cur_min = hole_len;
        pos_flag = 2;
        pos = p;
      }
      p = p -> nxt;
    }

    hole_len = memory_limit - p -> end - 1;
    if (size <= hole_len && hole_len < cur_min) {
      cur_min = hole_len;
      pos_flag = 3;
    }

    // No enough spaces
    if (pos_flag == 0) {
      fprintf(stderr, "[Err] No enough spaces!\n");
      return 1;
    }

    if (pos_flag == 1) {
      mem_node *tmp = head;
      head = (mem_node *) malloc (sizeof(mem_node));
```

```
      head -> bel = (char *) malloc (sizeof(char) * (name_len + 1));
      strcpy(head -> bel, process_name);
      head -> beg = 0;
      head -> end = 0 + size - 1;
      head -> nxt = tmp;
      return 0;
    }

    if (pos_flag == 2) {
      p = pos;
      mem_node *tmp = p -> nxt;
      p -> nxt = (mem_node *) malloc (sizeof(mem_node));
      p -> nxt -> bel = (char *) malloc (sizeof(char) * (name_len + 1));
      strcpy(p -> nxt -> bel, process_name);
      p -> nxt -> beg = p -> end + 1;
      p -> nxt -> end = p -> nxt -> beg + size - 1;
      p -> nxt -> nxt = tmp;
      return 0;
    }

    if (pos_flag == 3) {
      p -> nxt = (mem_node *) malloc (sizeof(mem_node));
      p -> nxt -> bel = (char *) malloc (sizeof(char) * (name_len + 1));
      strcpy(p -> nxt -> bel, process_name);
      p -> nxt -> beg = p -> end + 1;
      p -> nxt -> end = p -> nxt -> beg + size - 1;
      p -> nxt -> nxt = NULL;
      return 0;
    }
  }

  // Worst-fit
  if (type == 'W') {
    mem_node *p = head;
    int hole_len;

    // position
    int cur_max = -2e9;
    int pos_flag = 0;
    mem_node *pos;

    hole_len = p -> beg - 0;
    if (size <= hole_len && hole_len > cur_max) {
      cur_max = hole_len;
      pos_flag = 1;
    }

    while (p -> nxt != NULL) {
      hole_len = p -> nxt -> beg - p -> end - 1;
      if (size <= hole_len && hole_len > cur_max) {
```

```
        cur_max = hole_len;
        pos_flag = 2;
        pos = p;
      }
      p = p -> nxt;
    }

    hole_len = memory_limit - p -> end - 1;
    if (size <= hole_len && hole_len > cur_max) {
      cur_max = hole_len;
      pos_flag = 3;
    }

    // No enough spaces
    if (pos_flag == 0) {
      fprintf(stderr, "[Err] No enough spaces!\n");
      return 1;
    }

    if (pos_flag == 1) {
      mem_node *tmp = head;
      head = (mem_node *) malloc (sizeof(mem_node));
      head -> bel = (char *) malloc (sizeof(char) * (name_len + 1));
      strcpy(head -> bel, process_name);
      head -> beg = 0;
      head -> end = 0 + size - 1;
      head -> nxt = tmp;
      return 0;
    }

    if (pos_flag == 2) {
      p = pos;
      mem_node *tmp = p -> nxt;
      p -> nxt = (mem_node *) malloc (sizeof(mem_node));
      p -> nxt -> bel = (char *) malloc (sizeof(char) * (name_len + 1));
      strcpy(p -> nxt -> bel, process_name);
      p -> nxt -> beg = p -> end + 1;
      p -> nxt -> end = p -> nxt -> beg + size - 1;
      p -> nxt -> nxt = tmp;
      return 0;
    }

    if (pos_flag == 3) {
      p -> nxt = (mem_node *) malloc (sizeof(mem_node));
      p -> nxt -> bel = (char *) malloc (sizeof(char) * (name_len + 1));
      strcpy(p -> nxt -> bel, process_name);
      p -> nxt -> beg = p -> end + 1;
      p -> nxt -> end = p -> nxt -> beg + size - 1;
      p -> nxt -> nxt = NULL;
      return 0;
```

```
    }
  }

  fprintf(stderr, "[Err] Argument Error!\n");
  return 1;
}


// Release the resources allocated to process [process_name].
// If success, return 0; or return 1 with the error message printed in the screen
    .
int release(char *process_name) {
  if (head == NULL) {
    fprintf(stderr, "[Err] No such process!\n");
    return 1;
  }
  if (strcmp(head -> bel, process_name) == 0) {
    mem_node *tmp = head;
    head = head -> nxt;
    free(tmp -> bel);
    free(tmp);
    return 0;
  }

  mem_node *p = head;
  while (p -> nxt != NULL) {
    if (strcmp(p -> nxt -> bel, process_name) == 0) {
      mem_node *tmp = p -> nxt;
      p -> nxt = p -> nxt -> nxt;
      free(tmp -> bel);
      free(tmp);
      return 0;
    }
    p = p -> nxt;
  }

  fprintf(stderr, "[Err] No such process!\n");
  return 1;
}


// Compact the holes.
void compact() {
  int cur = 0;
  mem_node *p = head;
  while (p != NULL) {
    int size = p -> end - p -> beg + 1;
    p -> beg = cur;
    p -> end = cur + size - 1;
    cur += size;
```

```
    p = p -> nxt;
  }
  return ;
}


// Display the statistics.
void display() {
  if (head == NULL) {
    fprintf(stdout, "Address [0 : %d] Unused\n", memory_limit - 1);
    return ;
  } else if (head -> beg != 0) {
    fprintf(stdout, "Address [0 : %d] Unused\n", head -> beg - 1);
  }

  mem_node *p = head;
  int hole_len;

  while (p -> nxt != NULL) {
    fprintf(stdout, "Address [%d : %d] Process %s\n", p -> beg, p -> end, p -> bel
      );
    hole_len = p -> nxt -> beg - p -> end - 1;
    if (hole_len > 0)
      fprintf(stdout, "Address [%d : %d] Unused\n", p -> end + 1, p -> nxt -> beg
        - 1);
    p = p -> nxt;
  }

  fprintf(stdout, "Address [%d : %d] Process %s\n", p -> beg, p -> end, p -> bel);
  hole_len = memory_limit - p -> end - 1;
  if (hole_len > 0)
    fprintf(stdout, "Address [%d : %d] Unused\n", p -> end + 1, memory_limit - 1);
}


// Standardlize input.
void standardlize_input(char *op) {
  char tmp[LINE_MAX];
  int no_need = 1, tmpn = 0;

  for (int i = 0; op[i]; ++ i) {
    if (op[i] == ' ' || op[i] == '\t' || op[i] == '\n') {
      if (no_need == 0) {
        no_need = 1;
        tmp[tmpn ++] = ' ';
      }
    } else {
      tmp[tmpn ++] = op[i];
      no_need = 0;
    }
```

```
  }

  if (tmpn > 0 && tmp[tmpn - 1] == ' ') tmpn --;

  for (int i = 0; i < tmpn; ++ i) op[i] = tmp[i];
  op[tmpn] = 0;
}


int main(int argc, char *argv[]) {
  if (argc != 2) {
    fprintf(stderr, "[Err] Arguments Error!\n");
    exit(1);
  }

  memory_limit = atoi(argv[1]);

  char op[LINE_MAX], oper[LINE_MAX];
  char process_name[LINE_MAX];
  while(1) {
    for (int i = 0; i < LINE_MAX; ++ i) {
      op[i] = 0; oper[i] = 0;
      process_name[i] = 0;
    }

    fprintf(stdout, "allocator> ");
    fgets(op, LINE_MAX, stdin);

    standardlize_input(op);

    if (strcmp(op, "EXIT") == 0) break;

    if (strcmp(op, "C") == 0) {
      compact();
      continue;
    }

    if (strcmp(op, "STAT") == 0) {
      display();
      continue;
    }

    // parse the inputs.
    int pos = 0;
    for (; op[pos]; ++ pos)
      if(op[pos] == ' ') break;

    for (int i = 0; i < pos; ++ i)
      oper[i] = op[i];
    oper[pos] = 0;
```

```
    if (strcmp(oper, "RQ") == 0) {
      if (op[pos] == 0) {
        fprintf(stderr, "[Err] Invalid input!\n");
        continue;
      }

      int i;
      for (i = pos + 1; op[i]; ++ i) {
        if (op[i] == ' ') break;
        process_name[i - pos - 1] = op[i];
      }

      if (op[i] == 0) {
        fprintf(stderr, "[Err] Invalid input!\n");
        continue;
      }

      int invalid_input = 0;
      int size = 0;
      pos = i;
      for (i = pos + 1; op[i]; ++ i) {
        if(op[i] == ' ') break;
        if(op[i] < '0' || op[i] > '9') {
          invalid_input = 1;
          break;
        }
        size = size * 10 + op[i] - '0';
      }

      if (invalid_input || op[i] == 0) {
        fprintf(stderr, "[Err] Invalid input!\n");
        continue;
      }

      if ((i - pos - 1) >= 10) {
        fprintf(stderr, "[Err] Size too large!\n");
        continue;
      }

      pos = i;
      for (i = pos + 1; op[i]; ++ i) {
        if (op[i] == ' ') {
          invalid_input = 1;
          break;
        }
      }

      if (invalid_input || (i - pos - 1) != 1) {
        fprintf(stderr, "[Err] Invalid input!\n");
```

```
        continue;
      }

      request(process_name, size, op[pos + 1]);
    } else if (strcmp(oper, "RL") == 0) {
      if (op[pos] == 0) {
        fprintf(stderr, "[Err] Invalid input!\n");
        continue;
      }

      int invalid_input = 0;
      for (int i = pos + 1; op[i]; ++ i) {
        if (op[i] == ' ') {
          invalid_input = 1;
          break;
        }
        process_name[i - pos - 1] = op[i];
      }

      if (invalid_input) fprintf(stderr, "[Err] Invalid input!\n");
      else release(process_name);
    } else
      fprintf(stderr, "[Err] Invalid input!\n");
  }

  return 0;
}
```

1.4 Testing

I write a `Makefile` file to help testing the program. We only need to enter the following instructions in the terminal and we can begin testing, and the argument of the `allocator` program stands for the memory limit.

```
make
./allocator 1048576
RQ P1 10000 B
RQ P2 20000 B
RQ P3 30000 B
RQ P4 40000 B
STAT
RL P2
RQ P5 20001 F
STAT
RQ P6 10000 B
RQ P7 10000 W
RQ P8 10000 F
STAT
RL P6
RL P3
```

```
RL P5
STAT
C
STAT
EXIT
```

Here is the execution result of the memory allocation program (Fig. 1).

```
galaxies@ubuntu:~/CS307-Projects/Project7$ ./allocator 1048576
allocator> RQ P1 10000 B
allocator> RQ P2 20000 B
allocator> RQ P3 30000 B
allocator> RQ P4 40000 B
allocator> STAT
Address [0 : 9999] Process P1
Address [10000 : 29999] Process P2
Address [30000 : 59999] Process P3
Address [60000 : 99999] Process P4
Address [100000 : 1048575] Unused
allocator> RL P2
allocator> RQ P5 20001 F
allocator> STAT
Address [0 : 9999] Process P1
Address [10000 : 29999] Unused
Address [30000 : 59999] Process P3
Address [60000 : 99999] Process P4
Address [100000 : 120000] Process P5
Address [120001 : 1048575] Unused
allocator> RQ P6 10000 B
allocator> RQ P7 10000 W
allocator> RQ P8 10000 F
allocator> STAT
Address [0 : 9999] Process P1
Address [10000 : 19999] Process P6
Address [20000 : 29999] Process P8
Address [30000 : 59999] Process P3
Address [60000 : 99999] Process P4
Address [100000 : 120000] Process P5
Address [120001 : 130000] Process P7
Address [130001 : 1048575] Unused
allocator> RL P6
allocator> RL P3
allocator> RL P5
allocator> STAT
Address [0 : 9999] Process P1
Address [10000 : 19999] Unused
Address [20000 : 29999] Process P8
Address [30000 : 59999] Unused
Address [60000 : 99999] Process P4
Address [100000 : 120000] Unused
Address [120001 : 130000] Process P7
Address [130001 : 1048575] Unused
allocator> C
allocator> STAT
Address [0 : 9999] Process P1
Address [10000 : 19999] Process P8
Address [20000 : 59999] Process P4
Address [60000 : 69999] Process P7
Address [70000 : 1048575] Unused
allocator> EXIT
```

Figure 1: The execution result of the memory allocation program

2 Personal Thoughts

The project helps me understand the contiguous memory allocation algorithm better, including the first-fit algorithm, the best-fit algorithm and the worst-fit algorithm. The project also improves my understanding of memory compaction. The project trains our coding skills since we also need to implement parsing process and the shell-like windows except the algorithm. The implementation of the algorithms is quite simple but it needs patience. I enjoy the process of writing this program.

By the way, you can find all the source codes in the "src" folder. You can also refer to my github to see my codes of this project, and they are in the `Project7` folder.