EEE3535-02 **Fall 2024**

Assignment 4: Free List

Due: Sunday, Nov. 17, 2024, 11:59PM

1 Introduction

- The objective of this assignment is to replace the next-fit free list management scheme of xv6-riscv's malloc library with the worst-fit policy.
- The following shows the `malloc()` function in `user/umalloc.c`.

```
void*
malloc(uint nbytes)
{
  Header *p, *prevp;
  uint nunits;

  nunits = (nbytes + sizeof(Header) - 1)/sizeof(Header) + 1;
  if((prevp = freep) == 0){
    base.s.ptr = freep = prevp = &base;
    base.s.size = 0;
  }
  for(p = prevp->s.ptr; ; prevp = p, p = p->s.ptr){
    if(p->s.size >= nunits){
      if(p->s.size == nunits)
        prevp->s.ptr = p->s.ptr;
      else {
        p->s.size -= nunits;
        p += p->s.size;
        p->s.size = nunits;
      }
      freep = prevp;
      return (void*)(p + 1);
    }
    if(p == freep)
      if((p = morecore(nunits)) == 0)
        return 0;
  }
}
```

- At the beginning of `malloc()`, it rounds up `nbytes` (i.e., requested memory allocation size) to the nearest multiple of `sizeof(Header)` and adds one `Header` space. Thus, `nunits` is the memory size in units of `Header` plus 1.
- The following is the definition of `Header`, which is the `union` of `struct s` and `Align x`.

```
typedef long Align;

union header {
  struct {
    union header *ptr;
    uint size;
  } s;
  Align x;
};

typedef union header Header;
```

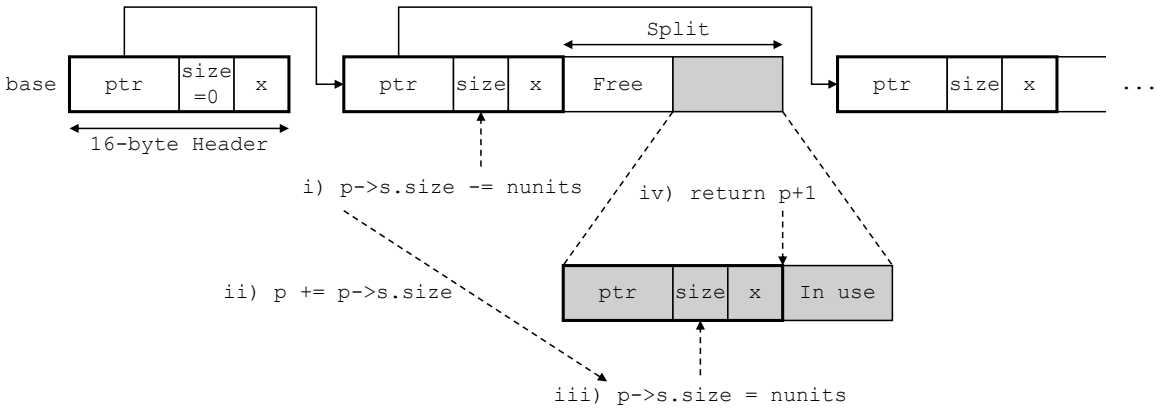

Figure 1: The procedure of splitting a free block to allocate a memory chunk. `malloc()` searches the free list to find a free block whose size is greater than or equal to the memory allocation size. If the free block is greater than the allocation size, it is split to allocate the memory chunk at the end of the free block. The first 16 bytes of the memory chunk are used as `Header`, and the actual starting address of the data space is `p+1`.

- The size of `struct s` is 12 bytes (i.e., 8-byte `ptr` and 4-byte `size`), and `long` (i.e., `Align`) in Linux is 8 bytes.
- The effective size of `union header` is 12 bytes but is rounded up to a multiple of the largest member variable size (i.e., `long`), which makes it align with 8 bytes. Thus, `sizeof(Header)` is 16 bytes.
- The first `if` statement of `malloc()` checks if a free list exists. If not, it creates one with a dummy node, `base`, with a size of zero (i.e., `base.s.size = 0`) and the next pointer pointing to itself (i.e., `base.s.ptr = &base`).
- Once the free list is established, free blocks are organized in the singly linked circular list, where the next pointer of the last free block loops back to `base`.
- The `for` loop searches the linked list starting from the next of `freep` and chooses the first-found free block whose size is greater than or equal to `nunits`. Since `freep` points to the last-accessed free block in the linked list, it works as a next-fit scheme.
- If the free block has the exact size, it is simply taken out from the linked list by making the next of the previous free block point to the next block (i.e., `prevp->s.ptr = p->s.ptr`).
- Otherwise (i.e., free block size greater than `nunits`), it is split to allocate a memory chunk at the rear end of the block. The size of the free block shrinks by `nunits`.
- `p` is incremented by `p->s.size` to point to the new memory chunk, and its size is set to `nunits` (i.e., rounded memory allocation size).
- Then, the memory address `p+1` is returned to the caller program because the first 16 bytes are used as `Header`. `p+1` is the actual starting address of the data space. The splitting procedure is outlined in Fig. 1.
- If `malloc()` fails to find a suitable free block until it loops back to `freep`, it calls `morecore(nunits)` to add a new free block to the linked list.

```
static Header*
morecore(uint nu)
{
  char *p;
  Header *hp;

  if(nu < 4096)
    nu = 4096;
```

```
  p = sbrk(nu * sizeof(Header));
  if(p == (char*)-1)
    return 0;
  hp = (Header*)p;
  hp->s.size = nu;
  free((void*)(hp + 1));
  return freep;
}
```

- To avoid making many syscalls with small memory sizes, `morecore()` allocates at least a 64KB free block at a time (i.e., `4096 * sizeof(Header)`).
- The `morecore()` function uses the `sbrk()` syscall to increase the heap segment size, which is directed to `sys_sbrk()` in `kernel/sysproc.c`.
- `sys_sbrk()` again calls `growproc()` in `kernel/proc.c` to grab the right number of free virtual pages.
- Returning to `morecore()`, it calls `free()` to add the new free block to the linked list and merge it with neighboring free blocks if applicable.
- This assignment only requires modifying the `malloc()` function, and you do not have to touch other functions, including `morecore()`, `free()`, etc. Understanding the rest of the code, such as `free()`, is left for you.

2 Implementation

- To start the assignment, go to the `xv6-riscv/` directory, download `freelist.sh`, and run the script.

```
$ cd xv6-riscv/
$ wget https://casl.yonsei.ac.kr/teaching/eee3535/freelist.sh
$ chmod +x freelist.sh
$ ./freelist.sh
```

- If the update is successful, you can run the new user program called `malloctest`.

```
$ make qemu

...

qemu-system-riscv64 -machine virt -bios none -kernel kernel/kernel -m 128M -smp 1
-nographic -global virtio-mmio.force-legacy=false -drive file=fs.img,if=none,
format=raw,id=x0 -device virtio-blk-device,drive=x0,bus=virtio-mmio-bus.0

EEE3535 Operating Systems: booting xv6-riscv kernel
EEE3535 Operating Systems: starting sh
$ malloctest
Test #1: malloc(49997 bytes)
Test #2: malloc(19837 bytes)
Test #3: malloc(17581 bytes)

...

Free list:
[1] p = 0x0000000000004000, p->s.size = 32112 bytes, p->s.ptr = 0x0000000000019310
[2] p = 0x0000000000019310, p->s.size = 8432 bytes, p->s.ptr = 0x0000000000024000
[3] p = 0x0000000000024000, p->s.size = 15616 bytes, p->s.ptr = 0x0000000000044000
[4] p = 0x0000000000044000, p->s.size = 36896 bytes, p->s.ptr = 0x0000000000001020
```

2.1 Worst-Fit Policy

- The worst-fit policy requires searching the entire free list to find the largest free block for each memory allocation. You can either start the search from `base` or maintain `freep` similar to the default scheme.
- If the free list has multiple candidates with the same condition, you may choose any one of them.

2.2 Tracking Free List

- To help you debug the malloc library, a new function called `freelist()` was added to `user/umalloc.c`, which traverses the linked list and prints the information of each free block, including its address, size, and the next pointer.
- In the following example, `freelist()` shows that there are two free blocks, one located at `0x4000` and another at `0x19310`. The first free block has a size of 32112 bytes (or 2007 `Header` units), and its next pointer points to the second free block. The second free block is 8432 bytes in size (or 527 `Header` units), and its next pointer is `0x1020`, which is the address of `base`.

```
Free list:
[1] p = 0x0000000000004000, p->s.size = 32112 bytes, p->s.ptr = 0x0000000000019310
[2] p = 0x0000000000019310, p->s.size = 8432 bytes, p->s.ptr = 0x0000000000001020
```

- `freelist()` can be used in both `user/umalloc.c` and `user/malloctest.c`.
- The `malloctest` program calls `freelist()` every five tests, but you may want to print the state of the free blocks after every `malloc` or `free()` to debug your implementation.
- It is fine to modify the `malloctest()` and `freelist()` functions during your assignment, but they must revert to the original code before submission.

3 Validation

- `malloctest` calls `freelist()` every five tests. Your assignment will be graded based on the linked list states of the free blocks.

```
$ malloctest
Test #1: malloc(49997 bytes)
Test #2: malloc(19837 bytes)
Test #3: malloc(17581 bytes)
Test #4: malloc(6788 bytes)
Test #5: free(19837 bytes)
Free list:
[1] p = 0x0000000000004000, p->s.size ...
```

4 Submission

- In the `xv6-riscv/` directory, run the `tar.sh` script to create a tar file named after your student ID (e.g., `2024143535`).

```
$ ./tar.sh
$ ls
2024143535.tar  kernel  LICENSE  Makefile  mkfs  README  tar.sh  user
```

- Upload this tar file (e.g., `2024143535.tar`) on LearnUs. Do not rename the file.

5 Grading Rules

- The following is the general guideline for grading. A 30-point scale will be used for this assignment. The minimum score is zero, and negative scores will not be given. Grading rules are subject to change; a grader may add extra rules without notice for the fair evaluation of students' efforts.

  -5 points: The tar file includes extra tags such as a student name, `hw4`, etc.

  -5 points: The code has insufficient comments. Comments in the skeleton code do not count. You must clearly explain what each part of your code does.

  -5 points: Do not modify `user/malloctest.c` and `freelist()` in `user/umalloc.c`. They must revert to the original state before submission.

  -5 points: Do not print unasked debugging messages.

  -5 points each: `malloctest` calls `freelist()` six times total. Each incorrect `freelist()` output will lose 5 points; no partial credits per `freelist()` call.

  -30 points: No or late submission.

  Final grade = F: The submitted tar file is copied from someone else. All students involved in the incidents will get "F" for the final grade.

- Your teaching assistant (TA) will grade your assignments. If you think your assignment score is incorrect, discuss your concerns with the TA. Always be courteous when contacting the TA. If no agreement is reached between you and the TA, escalate the case to the instructor to review your assignment. Refer to the course website for the contact information of the TA and the instructor: `https://casl.yonsei.ac.kr/eee3535`

- Arguing for partial credits for no valid reasons will be considered as a cheating attempt and lose the assignment score.