Trong hệ thống quản lý bộ nhớ, danh sách liên kết đơn có thể được sử dụng để theo dõi các **khối bộ nhớ trống (free blocks)**. Mỗi nút trong danh sách sẽ lưu thông tin về một khối bộ nhớ trống, bao gồm:

- **Địa chỉ bắt đầu** của khối.
- **Kích thước** của khối.
- **Con trỏ** trỏ tới khối trống tiếp theo.

Khi chương trình yêu cầu cấp phát bộ nhớ (malloc), hệ thống sẽ duyệt danh sách để tìm khối đủ lớn. Khi giải phóng bộ nhớ (free), khối được trả về danh sách và có thể được hợp nhất với các khối liền kề để tránh phân mảnh.

Ưu điểm và Nhược điểm của Quản lý Bộ nhớ Động bằng Danh sách Liên kết Đơn

1. Ưu điểm

✅ **Linh hoạt trong cấp phát và giải phóng**

- Dễ dàng **thêm/xóa** các khối bộ nhớ mà không cần dịch chuyển dữ liệu (khác với mảng).
- Không cần biết trước kích thước bộ nhớ cần dùng.

✅ **Tiết kiệm bộ nhớ**

- Chỉ sử dụng bộ nhớ cho các khối đã được cấp phát, không lãng phí như cấp phát tĩnh.
- Có thể **tái sử dụng** các khối đã giải phóng.

✅ **Hỗ trợ hợp nhất khối liền kề (Coalescing)**

- Khi giải phóng bộ nhớ, có thể **ghép các khối trống liền kề** thành một khối lớn hơn, giảm phân mảnh.

✅ **Phù hợp với bộ nhớ không liên tục**

- Có thể quản lý các vùng nhớ **phân tán** trên RAM mà không cần yêu cầu vùng nhớ liên tục.

✅ **Dễ triển khai các thuật toán cấp phát**

- Có thể áp dụng **First Fit, Best Fit, Worst Fit** để tối ưu hiệu suất.

---

2. Nhược điểm

❌ **Truy cập chậm do phải duyệt tuyến tính**

- Để tìm khối trống phù hợp, phải duyệt từ đầu danh sách → Độ phức tạp **O(n)**.
- Không hỗ trợ truy cập ngẫu nhiên (random access) như mảng.

❌ **Tốn bộ nhớ cho metadata**

- Mỗi khối nhớ phải lưu thêm con trỏ next và thông tin kích thước → **Overhead bộ nhớ**.
- Ví dụ: Nếu mỗi nút tốn 16 byte (8 byte con trỏ + 8 byte size), với nhiều khối nhỏ, phần metadata có thể chiếm đáng kể.

❌ **Phân mảnh bộ nhớ (Fragmentation)**

- **Phân mảnh ngoại (External Fragmentation):** Các khối trống nhỏ nằm rải rác, không đủ để cấp phát dù tổng bộ nhớ trống đủ.
- **Phân mảnh nội (Internal Fragmentation):** Nếu khối được cấp lớn hơn yêu cầu, phần dư bị lãng phí.

❌ **Khó debug khi xảy ra lỗi**

- Nếu con trỏ bị hỏng (ví dụ: giải phóng sai địa chỉ), danh sách có thể bị **đứt gãy** → Rò rỉ bộ nhớ hoặc crash.

❌ **Hiệu suất kém với danh sách lớn**

- Khi số khối trống tăng lên, thời gian tìm kiếm và hợp nhất khối trống tăng theo.

Định nghĩa 1 nút

```c
// Cấu trúc nút quản lý khối bộ nhớ trống
typedef struct MemoryBlock {
    void* start_addr;          // Địa chỉ bắt đầu
    size_t size;               // Kích thước khối
    struct MemoryBlock* next;  // Trỏ tới khối tiếp theo
} MemoryBlock;
```

Khởi tạo vùng nhớ trống ban đầu

```c
MemoryBlock* free_list = NULL; // Danh sách các khối trống

// Hàm khởi tạo một vùng nhớ ban đầu
void initialize_memory(void* base_addr, size_t total_size) {
    free_list = (MemoryBlock*)malloc(sizeof(MemoryBlock));
    free_list->start_addr = base_addr;
    free_list->size = total_size;
    free_list->next = NULL;
}
```

Hàm in ra danh sách các khối đang trống

```c
// In danh sách các khối trống
void print_free_list() {
    MemoryBlock* current = free_list;
    printf("Free Memory Blocks:\n");
    while (current != NULL) {
        printf("  Address: %p, Size: %zu\n", current->start_addr, current->size);
        current = current->next;
```

```
    }
}
```

Hãy hoàn thiện các hàm sau

```
// Hàm cấp phát bộ nhớ (đơn giản - First Fit)
void* firstfit_malloc(size_t size)

Dùng vùng nhớ trống đầu tiên cấp phát được (có kích thước >= kích thước cần cấp phát)
```

```
// Hàm cấp phát bộ nhớ (đơn giản - Best Fit)
void* bestfit_malloc(size_t size)

Dùng nhớ có kích thước nhỏ nhất lớn hơn hoặc bằng vùng nhớ xin cấp phát.
```

```
// Hàm cấp phát bộ nhớ (Worst Fit)
void* worstfit_malloc(size_t size)

Dùng vùng nhớ có kích thước lớn nhất để cấp phát.
```

```
// Hàm cấp phát bộ nhớ (Next Fit)
void* nextfit_malloc(size_t size)

Dùng vùng nhớ trống tiếp theo (tiếp tục từ vị trí cấp phát trước) có kích thước đủ lớn để cấp phát.
Khác với First Fit là luôn quay lại duyệt từ đầu.
```

```
// Hàm cấp phát bộ nhớ (Buddy System)
void* buddysystem_malloc(size_t size)

Bộ nhớ được chia thành các khối có kích thước là lũy thừa của 2 (2^n). Khi có yêu cầu cấp phát,
hệ thống chia khối nhớ lớn thành các "buddy" nhỏ hơn cho đến khi tìm được kích thước vừa đủ.
VD. nếu kích tước trống là 64KB, và cần cấp phát 20KB thì 64KB sẽ chia nhỏ thành 32 + 32KB và
cấp phát khối 32KB cho yêu cầu 20KB (do không thể chia nhỏ 32KB hơn nữa).
Khi tiến hành hợp nhất, sẽ hợp nhất với nửa đã chia ban đầu nếu có thể.
```

```
// Hàm giải phóng bộ nhớ và hợp nhất khối liền kề (nếu có)
void free_mem(void* addr, size_t size)
```

Một số ví dụ dùng hàm

```
initialize_memory(memory_pool, 2048);

print_free_list();

// Cấp phát 256 bytes
void* ptr1 = firstfit_malloc(256);
printf("\nAfter allocating 256 bytes:\n");
print_free_list();

// Cấp phát thêm 128 bytes
void* ptr2 = bestfit_malloc(128);
printf("\nAfter allocating 128 bytes:\n");
```

```
print_free_list();

// Giải phóng ptr1 (256 bytes)
free_mem(ptr1, 256);
printf("\nAfter freeing 256 bytes:\n");
print_free_list();
```

Input, đơn vị là KB

```
initialize_memory 2048
print_free_list
ptr1  = firstfit_malloc 256
ptr2  = bestfit_malloc 50
free_mem(ptr1, 256)
ptr3  = nextfit_malloc(1000)
ptr4 = bestfit_malloc(500)
free_mem(ptr3, 1000)
print_free_list
```