Memory Management in the Scheduler: A Buddy System Implementation

Abdelhady Mohamed 202201172
Ahmed Almuhanna 202202017
Salahdin Ahmed 202201079
Shehab Mohamed 202200285

December 26, 2024

1 Introduction

This report details the memory management aspect of a process scheduler, specifically focusing on the implementation of the Buddy System. In this context, the scheduler needs not only to manage the scheduling of processes but also efficiently allocate and deallocate memory to them. The Buddy System is chosen as a robust strategy for dynamic memory allocation that balances fragmentation and allocation speed.

2 The Buddy System Theory

2.1 Core Concept

The Buddy System is a memory allocation algorithm that divides memory into blocks that are powers of 2. When a request for memory is received, the system attempts to find a block of the appropriate size. If a suitable block is not available, a larger block is recursively split into ”buddies” (equal halves). Similarly, when memory is freed, the system attempts to merge adjacent ”buddy” blocks to form a larger block, avoiding fragmentation.

2.2 Key Principles

- **Power of 2 Blocks:** Memory is organized into blocks with sizes that are powers of 2 (e.g., 1, 2, 4, 8, 16, ... bytes).
- **Recursive Splitting:** If a block of the exact size requested is not available, a larger block is split into two equal halves (buddies). This splitting continues until a block of adequate size is obtained.
- **Coalescing:** When a block is freed, it’s merged with its buddy if the buddy is also free. This coalescing operation is also recursive, combining blocks to create larger contiguous free spaces.
- **Address Calculation:** Addresses of buddy blocks are easy to compute, simplifying the allocation and deallocation process.

2.3 Advantages

- **Reduced External Fragmentation:** By using powers of 2 block sizes and combining free buddies, fragmentation is minimized.
- **Efficient Allocation and Deallocation:** The buddy system has logarithmic time complexity for allocating and deallocating memory, which makes it efficient for frequent memory requests.
- **Simplified Address Calculation:** The use of powers of 2 sizes and bitwise operations makes it quick and easy to find the addresses of buddy blocks.

2.4 Disadvantages

- **Internal Fragmentation:** If a process requests a memory size that is not a power of 2, the system must allocate a block that is the next higher power of 2. This leads to internal fragmentation within the allocated block.

3 Implementation in the Scheduler

3.1 Buddy System Initialization

In the provided ‘scheduler.c‘ code, the buddy system is initialized as follows:

```c
#define MEMORY\_SIZE 1024
struct buddy *memory\_manager;

int main(int argc, char * argv[]) {
    // ...
    memory\_manager = buddy\_new(MEMORY\_SIZE);
    if (!memory\_manager) {
        fprintf(stderr, "Failed to initialize buddy system\n");
        exit(-1);
    }
    // ...
}
```

Listing 1: Buddy System Initialization in scheduler.c

The ‘MEMORY_SIZE‘ is set to 1024, indicating the total memory available to this memory management system. ‘buddy_new()‘ is a function (presumably defined in ‘buddy.c‘ or included in ‘buddy.h‘) that initializes the buddy system structure.

3.2 Memory Allocation

When a process is scheduled to run, the scheduler calls ‘buddy_alloc‘ to find memory for the process. The process’s memory size is specified in the ‘memsize‘ field of the ‘process_t‘ struct.

```c
int startProcess(process\_t * proc) {
    // ...
    int mem\_offset = buddy\_alloc(memory\_manager, proc->memsize);
    if (mem\_offset == -1) {
        printf("Error: Could not allocate memory for process %d\n", proc->id);
        return -1;
    }
    proc->startAddress = mem\_offset;
    // ...
}
```

Listing 2: Memory Allocation using buddy_alloc in scheduler.c

If successful, ‘buddy_alloc()‘ returns the starting address (‘mem_offset‘) of the allocated memory block. If memory cannot be allocated (e.g., not enough space is available), the function returns -1, and the scheduler handles the error by attempting again later. The allocated address is stored in the ‘startAddress‘ field of the ‘process_t‘ for future reference.

3.3 Memory Deallocation

When a process terminates, its allocated memory is released back to the system using ‘buddy_free‘:

```c
void handleProcTerm(int signum) {
    // ...
    if (running.startAddress != -1) {
        logMemoryEvent("freed", running);
        buddy\_free(memory\_manager, running.startAddress);
        running.startAddress = -1;  // Reset memory start
    }
    // ...
}
```

Listing 3: Memory Deallocation using buddy_free in scheduler.c

The scheduler frees memory only if ‘running.startAddress‘ is not -1, which ensures no double deallocations. After deallocation, the ‘startAddress‘ is reset, and the buddy system internally handles any merging of free blocks.

3.4 Memory Logging

The scheduler uses the ‘logMemoryEvent‘ to log the memory-related activities:

```c
void logMemoryEvent(const char *action, process\_t proc) {
    FILE *memoryLog = fopen("memory.log", "a");
    if (memoryLog == NULL) {
        perror("Error opening memory log file");
        return;
    }
    if (strcmp(action,"allocated")==0)
        fprintf(memoryLog, "At time %d allocated %d bytes for process %d from %d to %d\n",
                getClk(), proc.memsize, proc.id, proc.startAddress,proc.startAddress + buddy\
    _size(memory\_manager, proc.startAddress) - 1);
    else
        fprintf(memoryLog, "At time %d freed %d bytes from process %d from %d to %d\n",
                getClk(), proc.memsize, proc.id, proc.startAddress,proc.startAddress + buddy\
    _size(memory\_manager, proc.startAddress) - 1);
    fclose(memoryLog);
}
```

Listing 4: Memory Event Logging in scheduler.c

This function logs the allocation and deallocation events, which provides a track record of the actions taken by the buddy system within the scheduler. The log includes the time of the event, the process ID, the size of the memory, and the start and end addresses of the memory block.

3.5 Error Handling

The scheduler includes some error checks to handle allocation failures:

```c
int startProcess(process\_t * proc) {
    // ...
    int mem\_offset = buddy\_alloc(memory\_manager, proc->memsize);
    if (mem\_offset == -1) {
        printf("Error: Could not allocate memory for process %d\n", proc->id);
        return -1;
    }
    // ...
}
```

Listing 5: Error handling for memory allocation in scheduler.c

If ‘buddy_alloc‘ returns -1, it implies an allocation error, and the scheduler informs the user about this error.

4 Buddy System Structure

The provided code uses a ‘buddy‘ struct for memory management. Here are the basic functions:

- `buddy_new(size)`: Initialized the buddy system
- `buddy_alloc(buddy, size)`: Allocates memory of given size using buddy system
- `buddy_free(buddy, address)`: Frees memory of given address
- `buddy_size(buddy, address)`: Returns the size of memory block at given address
- `buddy_destory(buddy)`: Destroys the buddy system

The functions are assumed to be included in ‘buddy.h‘.

5 Conclusion

The Buddy System provides an effective method for dynamic memory allocation within the scheduler. It efficiently manages memory, minimizing external fragmentation and providing quick allocation and deallocation. The implementation in the provided scheduler code demonstrates how a memory management system can work hand-in-hand with process scheduling, enhancing the overall resource management of the operating system. The integration of a memory logging feature further enhances debugging and monitoring of memory-related issues during the scheduling of multiple processes.