**Ю.Р. Смокович**

 студент, Державний торгівельно-економічний

університет, м.Київ, Україна

Ю.Ю. Юрченко

старший викладач,

Державний торгівельно-економічний

університет, м.Київ, Україна

 ORCID - 0009-0009-6181-5477

**Ефективне управління пам’яттю мовою асемблера**

Вступ

Ефективне управління пам’яттю є однією з ключових задач системного програмування, оскільки від цього напряму залежить продуктивність програмного забезпечення, зменшення витрат ресурсів та стабільність роботи систем. Мова асемблера, як низькорівневий інструмент програмування, дозволяє отримати прямий контроль над апаратними ресурсами, що відкриває широкі можливості для оптимізації обчислювальних процесів.

Асемблер забезпечує програмісту прямий доступ до адрес пам’яті та апаратних інструкцій, дозволяючи точно визначати, як обробляються дані у пам’яті. Це стає особливо актуальним у задачах, пов’язаних із розробкою операційних систем, драйверів пристроїв та вбудованих систем, де використання високорівневих мов програмування може призводити до неефективного використання ресурсів.

Метою цієї роботи є аналіз методів управління пам’яттю засобами асемблера, виявлення шляхів оптимізації доступу до пам’яті та розробка практичних рекомендацій для зменшення витрат ресурсів. Зокрема, розглядаються питання роботи зі стеком, динамічною пам’яттю та інструменти забезпечення максимальної продуктивності через правильну організацію роботи з пам’яттю.

Принципи управління пам’яттю

Ефективне управління пам’яттю починається з розуміння основних типів пам’яті, доступних у сучасних комп’ютерних системах. У контексті мови асемблера головними складовими є:

* **Сегмент коду (Code Segment)**
Цей сегмент містить інструкції програми, які виконуються процесором. Робота з ним вимагає точного визначення меж пам’яті для уникнення переповнення або порушення захищених областей.
* **Сегмент даних (Data Segment)**
Використовується для зберігання змінних, які мають статичний чи глобальний характер. Управління пам’яттю в цьому сегменті передбачає розуміння специфіки ініціалізованих і неініціалізованих даних.
* **Стек (Stack)**
Стек використовується для зберігання локальних змінних, адрес повернення та контексту виконання функцій. Ефективне управління стеком дозволяє мінімізувати витрати пам’яті та запобігти помилкам переповнення.
* **Купа (Heap)**
Це динамічна пам’ять, яка виділяється під час виконання програми. Вона забезпечує більшу гнучкість у роботі з даними, однак неправильне управління купою може спричинити витоки пам’яті.

Управління пам’яттю також тісно пов’язане з різними типами адресації, такими як:

* **Пряма адресація:** використовується для доступу до конкретних областей пам’яті.
* **Непряма адресація:** забезпечує гнучкіший доступ через регістри.
* **Базово-індексна адресація:** дозволяє ефективно працювати з масивами та структурами.

Асемблер надає інструменти для точного маніпулювання цими сегментами, зокрема інструкції MOV, LEA та інші, які забезпечують доступ до даних та передачу їх між регістрами чи пам’яттю.

MOV AX, DataSegment ; Завантаження адреси сегмента даних

MOV DS, AX ; Встановлення сегмента даних

MOV AL, [1234H] ; Читання байта з пам’яті

Ефективне використання стеку

Стек — це одна з ключових структур пам’яті в мовах низького рівня, яка дозволяє зручно зберігати тимчасові дані, такі як локальні змінні, адреси повернення та параметри функцій. Він працює за принципом *LIFO* (останній зайшов — перший вийшов), що забезпечує швидкість виконання та простоту в управлінні.

Коли викликається функція, стек автоматично використовується для збереження адреси повернення і локальних змінних. Наприклад:

PUSH BP ; Збереження попереднього базового вказівника

MOV BP, SP ; Налаштування нового базового вказівника

SUB SP, 4 ; Виділення місця для локальних змінних

; Тіло функції

MOV SP, BP ; Відновлення стану стеку

POP BP ; Відновлення базового вказівника

RET ; Повернення з функції

Ефективне використання стеку включає уникнення його переповнення, економне використання пам’яті та обережне застосування рекурсії. Наприклад, якщо в одній функції викликаються десятки рекурсивних викликів без належної перевірки, це може призвести до переповнення стеку і аварійного завершення програми.

Стек також зручний для збереження проміжних результатів:

PUSH AX ; Збереження значення AX

MOV AX, BX ; Зміна значення AX

POP AX ; Відновлення попереднього значення

Правильне використання стеку дозволяє оптимізувати програму та уникати зайвих витрат ресурсів.

Купа та динамічна пам’ять

Купа (heap) — це ділянка оперативної пам’яті, яка використовується для динамічного розподілу ресурсів у програмах. На відміну від стеку, де розподіл пам’яті чітко організований і обмежений принципом LIFO, купа дозволяє створювати об'єкти довільного розміру і управляти їхнім життєвим циклом вручну.

У мовах низького рівня, таких як асемблер, доступ до купи здійснюється через системні виклики операційної системи або спеціальні функції, як-от malloc і free в C. Наприклад, у багатьох процесорах є регістри, що використовуються для управління адресами динамічної пам'яті.

Основні аспекти роботи з купою:

* **Ручне керування:** програміст повинен сам виділяти та звільняти пам’ять, що може призводити до помилок, таких як витоки пам’яті.
* **Фрагментація:** при активному використанні купи можуть виникати області невикористаної пам’яті, що знижує ефективність роботи програми.

Простий приклад розподілу пам’яті в купі:

MOV AX, size ; Розмір пам'яті, яку потрібно виділити

INT 21h ; Виклик системної функції для виділення пам'яті

MOV BX, AX ; Збереження адреси виділеної пам'яті

Оптимізація роботи з купою включає використання спеціальних алокаторів пам’яті, зменшення кількості звернень до неї і розумне управління життєвим циклом даних. Особливо важливо уникати ситуацій, коли пам'ять виділяється, але не звільняється, що може викликати перевантаження системи.

Оптимізація роботи з великими обчисленнями

Робота з великими обчисленнями в асемблері вимагає обережного управління як пам’яттю, так і регістрами процесора, щоб досягти максимальної ефективності. Основні стратегії включають:

1. **Мінімізація кількості операцій:** використання інструкцій процесора, які можуть виконувати декілька дій за одну команду (наприклад, множення з додаванням).
2. **Оптимізація використання регістрів:** більшість процесорів мають обмежену кількість регістрів, тому важливо мінімізувати використання проміжних змінних у пам’яті.
3. **Уникнення переповнення:** у великих обчисленнях результат може не поміститися у відведений регістр. Наприклад, якщо виконується множення великих чисел, необхідно заздалегідь обробити старші та молодші біти.

Приклад множення двох великих чисел:

MOV AX, high\_part ; Старша частина числа

MUL BX ; Множення старших частин

MOV DX, result\_high ; Збереження старшого результату

MOV AX, low\_part ; Молодша частина числа

MUL BX ; Множення молодших частин

ADD DX, AX ; Додавання результатів

Також корисно застосовувати алгоритми, які дозволяють розбити обчислення на менші частини, наприклад, метод Карацуби для множення або використання спеціалізованих бібліотек.

Робота з апаратними ресурсами

Одна з головних переваг програмування на асемблері полягає в прямому доступі до апаратних ресурсів системи, таких як регістри, порти вводу-виводу, таймери та контролери переривань.

**Робота з портами вводу-виводу**

Прямий доступ до периферійних пристроїв, таких як клавіатура, миша чи принтер, здійснюється через порти вводу-виводу. Наприклад, у x86 для цього використовується команда IN для читання і OUT для запису.

MOV DX, 03F8h ; Адреса COM-порту

MOV AL, 'A' ; Дані для передачі

OUT DX, AL ; Передача даних у порт

**Контроль переривань**

Переривання дозволяють процесору виконувати обробку подій від апаратних пристроїв. Програміст може налаштовувати їх, використовуючи таблицю векторів переривань. Наприклад, встановлення власної обробки переривань:

MOV AX, offset MyHandler ; Адреса обробника

MOV [int\_table+10h], AX ; Заміна стандартного обробника

**Оптимізація роботи апаратних ресурсів**

Ефективне використання ресурсів включає:

* Зниження частоти звернень до апаратури.
* Паралельну обробку подій через переривання.
* Використання DMA (Direct Memory Access) для прискорення передачі даних.

Таке управління дозволяє значно підвищити продуктивність і зменшити навантаження на процесор.

Висновок

У роботі було здійснено аналіз методів ефективного управління пам’яттю в програмуванні мовою асемблера, що дозволяє оптимізувати використання ресурсів комп’ютера. Підкреслено важливість правильного управління стеком і купою, оптимізації обчислень та апаратних ресурсів для підвищення продуктивності програмного забезпечення. Практична значущість полягає у можливості застосування цих методів у розробці операційних систем, драйверів та вбудованих систем.

Література

1. Абрамович С. А. *Архітектура комп’ютера та системне програмування*. — К.: Наукова думка, 2019.
2. Таненбаум Е. *Структура та проектування операційних систем*. — М.: Вільямс, 2020.
3. Кнут Д. Е. *Мистецтво програмування: Низькорівневе програмування*. — М.: БІНОМ, 2018.
4. Сивухин Д. В. *Основи системного програмування*. — СПб.: Пітер, 2021.
5. Офіційна документація Intel: *Intel® 64 and IA-32 Architectures Software Developer Manuals*. — Intel Corporation.
6. Stallings W. *Operating Systems: Internals and Design Principles*. — Pearson, 2020.
7. Fog A. *Optimizing Assembly Code for x86 Processors*. — Copenhagen University, 2021.