Крупченко Владислав Миколайович,бакалавр,студент

Державний торгівельно-економічний університет м.Київ

<https://orcid.org/0009-0001-8402-870X>

Оптимізація роботи програм через багатопоточність на асемблері

Вступ

З розвитком обчислювальної техніки та поширенням багатоядерних процесорів важливість багатопоточності стала очевидною для ефективного використання доступних ресурсів. Багатопоточність дозволяє виконувати кілька потоків одночасно, що збільшує швидкість виконання програм та забезпечує більш плавний розподіл навантаження між ядрами. Реалізація багатопоточності на низькому рівні за допомогою мови асемблера відкриває унікальні можливості для глибокої оптимізації, хоча й потребує значного рівня знань апаратної архітектури.

# Основи багатопоточності

Потік (або Thread) є незалежною послідовністю виконання команд. Кожен потік має власний стек, контекст виконання та реєстрові значення, але всі потоки процесу спільно використовують пам’ять та ресурси.

Основні компоненти багатопоточності:

- Планування потоків: Механізм розподілу ресурсів між потоками. Це може бути кооперативне планування або витісняюче.

- Синхронізація: Потоки часто взаємодіють через спільну пам’ять, що вимагає захисту критичних секцій (використовуючи м’ютекси, семафори або атомарні операції).

- Управління контекстом: Перемикання між потоками потребує збереження поточного стану (контексту) і завантаження нового, що включає значення регістрів, покажчиків стека та інші дані.

# Реалізація багатопоточності на асемблері

Реалізація багатопоточності на асемблері забезпечує максимальну продуктивність завдяки прямому доступу до апаратних інструкцій процесора та мінімізації накладних витрат. Однак це також ускладнює процес розробки.

1. Створення потоків

Для створення потоків зазвичай використовуються системні виклики. Наприклад, у Linux системний виклик clone дозволяє створювати нові потоки, визначаючи їхній стек та права доступу.

mov rax, 56 ; Код системного виклику clone

mov rdi, 0x120000 ; Прапори для нового потоку

mov rsi, stack\_ptr ; Вказівник на стек нового потоку

syscall ; Виконання системного виклику

2. Перемикання контексту

Перемикання між потоками вимагає збереження поточного стану виконання. Це включає збереження значень регістрів, адреси стека та інструкцій.

save\_context:

push rax

push rbx

push rcx

ret

restore\_context:

pop rcx

pop rbx

pop rax

ret

3. Синхронізація потоків

Для запобігання конфліктам між потоками використовуються механізми синхронізації, такі як м’ютекси та атомарні операції:

lock cmpxchg [mutex], rax ; Атомарна операція порівняння і обміну

# Практичний приклад: Паралельне додавання елементів масиву

Для демонстрації розглянемо завдання паралельного додавання елементів масиву. Кожен потік буде обробляти окремий фрагмент даних, а результати об'єднуватимуться.

section .data

array db 1, 2, 3, 4, 5, 6, 7, 8

array\_size equ 8

result dd 0

section .text

global \_start

\_start:

mov rsi, array ; Вказівник на масив

mov rcx, array\_size ; Розмір масиву

xor rax, rax ; Обнулення результату

parallel\_sum:

mov rdx, [rsi] ; Завантаження елемента

add rax, rdx ; Додавання

add rsi, 4 ; Перехід до наступного елемента

loop parallel\_sum

mov [result], eax ; Збереження результату

mov rax, 60 ; Код exit

xor rdi, rdi ; Код завершення 0

syscall

Цей приклад демонструє базовий підхід до розподілу задач між потоками.

# Оптимізація багатопоточності

Для досягнення максимальної продуктивності слід враховувати наступні аспекти:

- Ефективне використання регістрів: Мінімізуйте звернення до пам'яті, використовуючи регістри для тимчасового збереження даних.

- Розподіл задач: Рівномірно розподіляйте обчислення між потоками, щоб уникнути простою ядер.

- Зменшення блокувань: Використовуйте атомарні операції замість дорогих механізмів блокування, де це можливо.

- Бар’єри пам’яті: Використовуйте інструкції, такі як mfence, для синхронізації доступу до пам’яті.

Всі ці техніки дозволяють створювати високооптимізовані програми.

# Висновок

Реалізація багатопоточності на асемблері надає унікальні можливості для оптимізації роботи програм, дозволяючи максимально ефективно використовувати апаратні ресурси. Водночас цей підхід є складним і потребує глибокого розуміння архітектури процесора, синхронізації потоків та системних викликів. Правильне використання технік багатопоточності забезпечує значне підвищення продуктивності та ефективності програм.

Застосування асемблера в супутнику Voyager 1

Програмування на низькому рівні, зокрема за допомогою асемблера, є основою для роботи багатьох критичних систем, включаючи космічні апарати. Один із найвідоміших прикладів — міжпланетний зонд Voyager 1, який був запущений у 1977 році та продовжує надсилати дані з міжзоряного простору.

Основна система управління Voyager 1 була створена на основі набору інструкцій асемблера, що дозволило максимально ефективно використовувати обмежені апаратні ресурси зонда. Центральний комп'ютер зонда — CCS (Command and Control Subsystem) — працює із швидкістю близько 250 тисяч інструкцій на секунду, що значно поступається сучасним процесорам. Однак оптимізація коду на рівні асемблера забезпечила стабільну роботу системи вже понад 45 років.

### Методи оптимізації багатопоточності

1. Ефективне використання регістрів

Мінімізація звернень до оперативної пам'яті шляхом використання регістрів для тимчасового збереження даних. Це дозволяє значно скоротити час виконання операцій.

2. Рівномірний розподіл задач між потоками

Важливо уникати простою ядер процесора. Це досягається шляхом розподілу роботи між потоками, беручи до уваги навантаження кожного з них.

3. Зменшення блокувань

Використання атомарних операцій замість м'ютексів або семафорів у критичних секціях дозволяє уникнути дорогих операцій синхронізації.

4. Бар’єри пам’яті

Інструкції, такі як mfence або lfence, забезпечують порядок виконання операцій читання і запису, гарантуючи коректний доступ до спільної пам'яті.

5. Адаптація до апаратної архітектури

Оптимізація коду з урахуванням специфічних особливостей процесора (наприклад, використання SIMD-інструкцій).

Варіанти реалізації багатопоточност

1. Створення потоків

Використання системних викликів, таких як clone у Linux, для запуску нових потоків із визначеними параметрами (наприклад, окремий стек або права доступу).

mov rax, 56 ; Код системного виклику clone

mov rdi, 0x120000 ; Прапори для нового потоку

mov rsi, stack\_ptr ; Вказівник на стек нового потоку

syscall ; Виконання системного виклику

2. Перемикання контексту

Збереження стану виконання поточного потоку (значення регістрів, покажчиків стека тощо) та завантаження нового контексту для активного потоку.

Приклад

save\_context:

push rax

push rbx

push rcx

ret

restore\_context:

pop rcx

pop rbx

pop rax

ret

3. Синхронізація потоків

Використання механізмів, таких як атомарні операції (lock cmpxchg), м'ютекси чи семафори для захисту спільної пам’яті.

4. Розподіл даних між потоками

У завданнях, таких як обробка масивів, дані розбиваються на сегменти, які обробляються паралельно різними потоками. Наприклад, паралельне додавання елементів масиву.

section .data

array db 1, 2, 3, 4, 5, 6, 7, 8

array\_size equ 8

result dd 0

section .text

global \_start

\_start:

mov rsi, array ; Вказівник на масив

mov rcx, array\_size ; Розмір масиву

xor rax, rax ; Обнулення результату

parallel\_sum:

mov rdx, [rsi] ; Завантаження елемента

add rax, rdx ; Додавання

add rsi, 4 ; Перехід до наступного елемента

loop parallel\_sum

mov [result], eax ; Збереження результату

mov rax, 60 ; Код exit

xor rdi, rdi ; Код завершення 0

Syscall

# Наукові джерела

**Офіційна документація процесорів**

Intel® 64 and IA-32 Architectures Software Developer's Manual

AMD64 Architecture Programmer's Manual

**Книги та навчальні посібники**

"Modern Operating Systems" - Andrew S. Tanenbaum

"Programming from the Ground Up" - Jonathan Bartlett

"The Art of Assembly Language" - Randall Hyde

**Матеріали про багатопоточність та синхронізацію**

"Pthreads Programming" - David R. Butenhof

"Concurrency in Action" - Anthony Williams

**Документація для системних викликів**

Linux man pages (особливо секції clone, futex, pthread та ін.)

Kernel documentation (<https://www.kernel.org/doc/>)

**Статті та технічні ресурси**

Intel Developer Zone (<https://www.intel.com/content/www/us/en/developer/overview.html>)

AMD Developer Resources (<https://developer.amd.com/>)

Довідники на платформі Stack Overflow щодо використання асемблера та багатопоточності.

**Навчальні відео та курси**

Відеолекції на YouTube від каналів, присвячених низькорівневому програмуванню (Low-Level Programming, Assembly tutorials).

**Дослідницькі праці про Voyager 1**

* Офіційні дані NASA: <https://voyager.jpl.nasa.gov/>