МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
ХАРКІВСЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ РАДІОЕЛЕКТРОНІКИ

Кафедра Програмної інженерії

Звіт
з лабораторної роботи №6
з дисципліни: «Операційні системи»
з теми: «Керування пам'яттю. Частина 2»

Виконали:
ст. гр. ПЗПІ-23-2
Ситник Є. С.
Малишкін. А. С.

Перевірила:
доц. каф. ПІ,
Мельнікова Р. В.,

Харків – 2025

6 КЕРУВАННЯ ПАМ'ЯТТЮ. ЧАСТИНА 2

6.1 Мета роботи

Вивчити особливості захисту критичних даних, використання динамічної та кеш пам’яті.

6.2 Хід роботи

6.2.1 Порівняти алгоритми заміщення сторінок (оптимальний, годинник, LRU).

Заміщення сторінок - це процес, який використовується в операційних системах для керування віртуальною пам’яттю, коли фізичної пам’яті недостатньо. Коли потрібна сторінка, якої немає в оперативній пам’яті, операційна система вибирає одну зі сторінок, що вже знаходяться в пам’яті, і замінює її потрібною сторінкою.

Сучасні операційні системи використовують різні підходи для визначення сторінок, які можна замістити.

Розглянемо деякі алгоритми заміщення сторінок:

а) оптимальний алгоритм (також відомий як алгоритм Беладі) – теоретичний алгоритм, який досягає найменшої кількості промахів сторінок. Він працює, замінюючи сторінку, яка не буде використовуватися найдовший період часу в майбутньому. Хоча цей алгоритм є оптимальним, він не може бути реалізований на практиці, оскільки вимагає знання майбутньої послідовності звернень до сторінок;

б) алгоритм годинник (також відомий як алгоритм «другого шансу») – зберігає список сторінок у пам’яті, кожна з яких має біт використання. Коли відбувається промах сторінки і потрібно замінити сторінку, алгоритм перевіряє біт використання поточної сторінки, на яку вказує покажчик. Якщо біт використання дорівнює 1, він скидається до 0, і покажчик переміщується до наступної сторінки. Цей процес повторюється до тих пір, поки не буде знайдено сторінку з бітом використання, що дорівнює 0, яка і замінюється. Такий підхід

надає «другий шанс» сторінкам, які нещодавно використовувалися, запобігаючи їхньому швидкому витісненню;

в) алгоритм LRU (Least Recently Used) – замінює сторінку, яка найдовше не використовувалася. Він базується на припущенні, що сторінки, які використовувалися нещодавно, ймовірно, будуть використовуватися знову в найближчому майбутньому, а ті, що давно не використовувалися, мають меншу ймовірність повторного звернення. Для реалізації LRU необхідно відстежувати час останнього звернення до кожної сторінки в пам'яті. Коли виникає потреба у заміщенні, вибирається сторінка з найдавнішим часом останнього використання.

paging.cpp

```cpp
int32_t paging::optimal_replacer::run(const std::vector<int32_t> &pages) {
  int32_t faults = 0;
  std::vector<int32_t> frames;
  frames.reserve(static_cast<size_t>(this->_frames));
  for (size_t i : std::views::iota(size_t(0), pages.size())) {
    int32_t p = pages[i];
    if (std::find(frames.begin(), frames.end(), p) != frames.end())
      continue;
    ++faults;
    if (static_cast<int32_t>(frames.size()) < this->_frames) {
      frames.push_back(p);
    } else {
      int32_t idx_to_replace = 0;
      size_t farthest = 0;
      for (int32_t j : std::views::iota(0, this->_frames)) {
        size_t k = i + 1;
        while (k < pages.size() && pages[k] != frames[j])
          ++k;
        size_t dist =
            (k < pages.size() ? k : std::numeric_limits<size_t>::max());
        if (dist > farthest) {
          farthest = dist;
          idx_to_replace = j;
        }
      }
      frames[idx_to_replace] = p;
    }
  }
  return faults;
}
```

Рисунок 6.1 – Оптимальний алгоритм заміщення сторінок

paging.cpp

```cpp
int32_t paging::clock_replacer::run(const std::vector<int32_t> &pages) {
  int32_t faults = 0;
  struct entry {
    int32_t page;
    bool ref;
  };
  std::vector<entry> frames;
  frames.reserve(static_cast<size_t>(this->_frames));
  int32_t hand = 0;

  for (int32_t p : pages) {
    bool hit = false;
    for (auto &e : frames) {
      if (e.page == p) {
        e.ref = true;
        hit = true;
        break;
      }
    }
    if (hit)
      continue;
    ++faults;
    if (static_cast<int32_t>(frames.size()) < this->_frames) {
      frames.push_back({p, true});
    } else {
      while (true) {
        if (!frames[hand].ref) {
          frames[hand] = {p, true};
          hand = (hand + 1) % this->_frames;
          break;
        }
        frames[hand].ref = false;
        hand = (hand + 1) % this->_frames;
      }
    }
  }
  return faults;
}
```

Рисунок 6.2 – Алгоритм заміщення сторінок «Годинник»

paging.cpp

```cpp
int32_t paging::lru_replacer::run(const std::vector<int32_t> &pages) {
  int32_t faults = 0;
  std::vector<int32_t> frames;
  std::unordered_map<int32_t, int32_t> last_used;
  frames.reserve(static_cast<size_t>(this->_frames));

  for (int32_t i : std::views::iota(0, static_cast<int32_t>(pages.size()))) {
    int32_t p = pages[i];
    if (std::find(frames.begin(), frames.end(), p) != frames.end()) {
      last_used[p] = i;
      continue;
    }
    ++faults;
    if (static_cast<int32_t>(frames.size()) < this->_frames) {
      frames.push_back(p);
    } else {
      int32_t lru_page = frames[0];
      int32_t min_idx = last_used[lru_page];
      for (int32_t q : frames) {
        if (last_used[q] < min_idx) {
          min_idx = last_used[q];
          lru_page = q;
        }
      }
      auto it = std::find(frames.begin(), frames.end(), lru_page);
      if (it != frames.end())
        *it = p;
    }
    last_used[p] = i;
  }
  return faults;
}
```

Рисунок 6.3 – Алгоритм заміщення сторінок «LRU»

Протестуємо всі алгоритми із однаковими тестовими даними.

main.cpp

```cpp
std::vector<int> pages = {7, 0, 1, 2, 0, 3, 0, 4, 2, 3, 0, 3};
int frames = 3;

paging::optimal_replacer opt(frames);
paging::clock_replacer clk(frames);
paging::lru_replacer lru(frames);

std::println("Optimal faults: {}", opt.run(pages));
std::println("Clock   faults: {}", clk.run(pages));
std::println("LRU     faults: {}", lru.run(pages));
```

Рисунок 6.4 – Код тестування алгоритмів

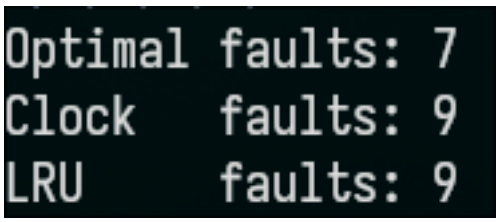

Рисунок 6.5 – Результати тестування

Можемо побачити, що оптимальний алгоритм показує найкращі результати.

6.2.2 Реалізувати алгоритм LRU для роботи з кешем для стандартних параметрів кешу

Кешування - це техніка збереження часто використовуваних даних у швидшій пам’яті (кеші) для зменшення часу доступу. Оскільки кеш має обмежений розмір, при його заповненні виникає потреба у визначенні того, які дані слід видалити для звільнення місця для нових.

cache.cpp

```cpp
bool cache::lru::access(uint64_t address) {
  uint64_t line = (address >> 6) % this->_lines;
  uint64_t tag =
      address >> (6 + static_cast<uint64_t>(std::log2(this->_lines)));

  auto &line_list = this->_cache_lines[line];
  auto &line_map = this->_lookup[line];

  auto it = line_map.find(tag);
  if (it != line_map.end()) {
    line_list.splice(line_list.begin(), line_list, it->second);
    line_map[tag] = line_list.begin();
    return true;
  }

  if (static_cast<int32_t>(line_list.size()) >= this->_assoc) {
    auto last = line_list.back();
    line_map.erase(last.tag);
    line_list.pop_back();
  }
  line_list.push_front({tag});
  line_map[tag] = line_list.begin();
  return false;
}
```

Рисунок 6.6 – Алгоритм «LRU» для доступу до кешу

main.cpp

```cpp
std::println("\nCache LRU Simulation:");
cache::lru cache(128, 4);
std::vector<uint64_t> addresses = {0x0000, 0x0040, 0x0080, 0x0000,
                                   0x0100, 0x0140, 0x0000, 0x0200,
                                   0x0080, 0x0240, 0x0280, 0x0000};
int32_t hits = 0, misses = 0;
for (auto addr : addresses) {
  if (cache.access(addr)) {
    ++hits;
    std::println("Address 0x{:x}:\t HIT", addr);
  } else {
    ++misses;
    std::println("Address 0x{:x}:\t MISS", addr);
  }
}
std::println("Hits: {}, Misses: {}\n", hits, misses);
```

Рисунок 6.7 – Код тестування

```
Cache LRU Simulation:
Address 0x0:     MISS
Address 0x40:    MISS
Address 0x80:    MISS
Address 0x0:     HIT
Address 0x100:   MISS
Address 0x140:   MISS
Address 0x0:     HIT
Address 0x200:   MISS
Address 0x80:    HIT
Address 0x240:   MISS
Address 0x280:   MISS
Address 0x0:     HIT
Hits: 4, Misses: 8
```

Рисунок 6.8 – Результати тестування

6.2.3 Реалізувати функції встановлення паролю та перевірки паролю. При складанні функцій забезпечте безпечне зберігання паролів.

Безпечність пароля можна визначити за часом, що потрібен для його зламу. Чим більше часу необхідно на злам, тим безпечніший пароль.

Number of Characters	Numbers Only	Lowercase Letters	Upper and Lowercase Letters	Numbers, Upper and Lowercase Letters	Numbers, Upper and Lowercase Letters, Symbols
4	Instantly	Instantly	Instantly	Instantly	Instantly
5	Instantly	Instantly	Instantly	Instantly	Instantly
6	Instantly	Instantly	Instantly	Instantly	24 mins
7	Instantly	Instantly	3 hours	12 hours	1 day
8	Instantly	43 mins	1 weeks	1 month	3 months
9	Instantly	18 hours	1 year	5 years	16 years
10	Instantly	3 weeks	56 years	325 years	1k years
11	20 mins	1 year	2k years	20k years	76k years
12	3 hours	37 years	151k years	1m years	5m years
13	1 day	962 years	7m years	77m years	375m years
14	2 weeks	25k years	409m years	4bn years	26bn years
15	5 months	650k years	21bn years	298bn years	1tn years
16	4 years	16m years	1tn years	18tn years	128tn years
17	39 years	439m years	57tn years	1qd years	9qd years
18	388 years	11bn years	2qd years	71qd years	631qd years

Рисунок 6.9 – Таблиця відповідності складності паролю до часу, що необхідний на злам

Критерії надійності паролю зазначені в таблиці детально обговорюються в дослідженні Hive Systems, яке оновлюється кожен рік починаючи з 2020 (https://www.hivesystems.com/blog/are-your-passwords-in-the-green)

password.cpp

```cpp
sec::strength_t sec::manager::evaluate_strength(const sec::password &pwd) {
  uint8_t score = 0;

  if (has_lower(pwd))         ++score;
  if (has_upper(pwd))         ++score;
  if (has_digit(pwd))         ++score;
  if (has_spec_symbol(pwd))   ++score;
  if (pwd.view().size() ≥ 8) ++score;

  if (score ≠ 0) score--;

  uint8_t max_val = static_cast<uint8_t>(sec::strength_t::very_strong);
  uint8_t idx = std::min(score, max_val);

  return static_cast<sec::strength_t>(idx);
}
```

Рисунок 6.10 – Функція перевірки надійності пароля

Для забезпечення безпеки пароля в пам’яті під час гешування було використано стандартні POSIX функції «mlock» та «munlock».

locked_buffer.hpp

```cpp
locked_buffer() {
  this->size_ = N * sizeof(T);
  this->ptr_ = static_cast<T *>(std::malloc(this->size_));
  if (!this->ptr_)
    throw std::bad_alloc();
  if (mlock(this->ptr_, this->size_) != 0) {
    std::free(this->ptr_);
    throw std::runtime_error("mlock failed");
  }
}

~locked_buffer() noexcept {
  cleanse();
  if (this->ptr_) {
    munlock(this->ptr_, this->size_);
    std::free(this->ptr_);
  }
}
```

Рисунок 6.11 – Функції блокування та розблокування пам’яті

Для гешування паролю ми використали алгоритм «SHA256» із бібліотеки «openssl», яка є стандартним вибором для всього, що так чи інакше пов’язано із криптографією.

password.cpp

```cpp
std::string sec::manager::hash_password(const sec::password &pwd) {
  auto sv = pwd.view();
  unsigned char hash[SHA256_DIGEST_LENGTH];
  SHA256(reinterpret_cast<const unsigned char *>(sv.data()), sv.size(), hash);
  std::ostringstream oss;
  for (int32_t i : std::views::iota(0, SHA256_DIGEST_LENGTH)) {
    oss << std::hex << std::setw(2) << std::setfill('0')
        << static_cast<int32_t>(hash[i]);
  }
  return oss.str();
}
```

Рисунок 6.12 – Функція гешування паролю

6.3 Висновки

Під час даної лабораторної роботи ми вивчили особливості захисту критичних даних, використання динамічної та кеш пам’яті.