Федеральное государственное автономное образовательное учреждение высшего образования Национальный исследовательский университет ИТМО

Факультет программной инженерии и компьютерной техники

Операционные системы
Лабораторная работа №2
Вариант LRU

Выполнил: студент группы P3308,
Васильев Н. А.

Преподаватель: Максимов Андрей Николаевич

Санкт-Петербург 2025

Текст задания

Для оптимизации работы с блочными устройствами в ОС существует кэш страниц с данными, которыми мы производим операции чтения и записи на диск. Такой кэш позволяет избежать высоких задержек при повторном доступе к данным, так как операция будет выполнена с данными в RAM, а не на диске.

При выполнении работы необходимо реализовать простой API для работы с файлами, предоставляющий пользователю следующие возможности (по аналогии с системным API):

1. Открытие файла по заданному пути файла, доступного для чтения. Процедура возвращает некоторый хэндл на файл. Пример: `int vtpc_open(const char *path)`.
2. Закрытие файла по хэндлу. Пример: `int vtpc_close(int fd)`.
3. Чтение данных из файла. Пример: `ssize_t vtpc_read(int fd, void buf[.count], size_t count)`.
4. Запись данных в файл. Пример: `ssize_t vtpc_write(int fd, const void buf[.count], size_t count)`.
5. Перестановка позиции указателя на данные файла. Достаточно поддержать только абсолютные координаты. Пример: `off_t vtpc_lseek(int fd, off_t offset, int whence)`.
6. Синхронизация данных из кэша с диском. Пример: `int vtpc_fsync(int fd)`.
7. Операции с диском разработанного блочного кеша должны производиться в обход страничного кеша операционной системы.

В рамках проверки работоспособности разработанного блочного кэша необходимо адаптировать указанную преподавателем программу-загрузчик из ЛР 1, добавив использование кэша. Запустите программу и убедитесь, что она корректно работает. Сравните производительность до и после.

Листинг исходного кода всех программ

GitHub: https://github.com/kihort-si/os-course/tree/lab-2/lab/vtpc

Реализация кэша

LRU (Least Recently Used) – это алгоритм вытеснения страниц из кэша/памяти, который при нехватке места удаляет ту страницу, к которой обращались наиболее давно. Идея опирается на принцип локальности: если страницу давно не использовали, с меньшей вероятностью она понадобится в ближайшее время.

Как работает LRU:

Пусть есть кэш фиксированного размера (например, N страниц). Для каждой страницы нужно понимать, насколько «свежим» было последнее обращение.

1. Обращение к странице (hit):

   Если страница уже в кэше, она считается “самой недавно использованной” и её метка/позиция обновляется.

2. Обращение к странице (miss):

   Если страницы нет в кэше:

   - если в кэше есть свободное место — страницу просто добавляют и помечают как “самую свежую”;
   - если кэш заполнен – выбирают жертву для вытеснения: страницу с самым старым временем последнего обращения (она “Least Recently Used”), удаляют её и помещают новую страницу.

Таким образом, кэш всегда хранит страницы, которые использовались относительно недавно. При любом обращении к странице она перемещается в начало, при добавлении новой страницы она также ставится в начало, при переполнении удаляется элемент с конца.

Реализация LRU:

```
typedef struct CachePage CachePage;

struct CachePage {
  int   used;
  int   fd;
  off_t page_index;

  char *data;
  int   dirty;

  CachePage *lru_prev;
  CachePage *lru_next;

  CachePage *hash_next;
};

static size_t     g_page_size = 0;
static FileCtx    g_files[MAX_OPEN_FILES];
static CachePage  g_pages[MAX_CACHE_PAGES];

static CachePage *g_lru_head = NULL;
static CachePage *g_lru_tail = NULL;
static CachePage *g_hash[HASH_SIZE];
```

Кэш хранит данные постранично. Ключ страницы – пара (fd, page_index):

- `data` – буфер размера `g_page_size`.
- `dirty` – признак грязной страницы (в неё писали; перед вытеснением нужно сбросить на диск).
- `lru_prev/lru_next` – двусвязный список LRU:
  - `g_lru_head` – самая недавно использованная (MRU),
  - `g_lru_tail` – самая давно использованная (LRU), кандидат на вытеснение.
- `g_hash[HASH_SIZE]` – хеш-таблица для быстрого поиска страницы по ключу (цепочки коллизий через `hash_next`).
- `g_pages[]` – пул заранее выделенных структур страниц (ёмкость кэша ограничена `MAX_CACHE_PAGES`).

Идея такая: поиск страницы – через хеш O(1), а выбор жертвы для вытеснения – через хвост LRU-списка O(1).

```
static unsigned long hash_key(int fd, off_t page_index) {
  unsigned long x = (unsigned long)fd;
  unsigned long y = (unsigned long)page_index;
  return (x * 1315423911u ^ y * 2654435761u) % HASH_SIZE;
}

static CachePage* hash_lookup(int fd, off_t page_index) {
  unsigned long idx = hash_key(fd, page_index);
  CachePage* p = g_hash[idx];
  while (p) {
    if (p->fd == fd && p->page_index == page_index)
      return p;
    p = p->hash_next;
  }
  return NULL;
}

static void hash_insert(CachePage* page) {
  unsigned long idx = hash_key(page->fd, page->page_index);
  page->hash_next = g_hash[idx];
  g_hash[idx] = page;
}

static void hash_remove(CachePage* page) {
  unsigned long idx = hash_key(page->fd, page->page_index);
  CachePage** pp = &g_hash[idx];
  while (*pp) {
    if (*pp == page) {
      *pp = page->hash_next;
      page->hash_next = NULL;
      return;
    }
    pp = &(*pp)->hash_next;
  }
}
```

LRU-алгоритму важно быстро отвечать на вопрос: страница уже в кэше или нет? Это делается хеш-таблицей:

- `hash_lookup(fd, page_index)` возвращает указатель на страницу, если она уже закэширована.
- `hash_insert(page)` и `hash_remove(page)` поддерживают консистентность кэша при загрузке новой страницы и при вытеснении/закрытии файла.

Так обеспечивается быстрый доступ без линейного перебора всех страниц.

```
static void lru_insert_front(CachePage* page) {
  page->lru_prev = NULL;
  page->lru_next = g_lru_head;

  if (g_lru_head)
    g_lru_head->lru_prev = page;
  g_lru_head = page;
  if (!g_lru_tail)
    g_lru_tail = page;
}

static void lru_remove(CachePage* page) {
  if (page->lru_prev)
    page->lru_prev->lru_next = page->lru_next;
```

```
  else
    g_lru_head = page->lru_next;

  if (page->lru_next)
    page->lru_next->lru_prev = page->lru_prev;
  else
    g_lru_tail = page->lru_prev;

  page->lru_prev = page->lru_next = NULL;
}

static void lru_move_to_front(CachePage* page) {
  if (g_lru_head == page)
    return;
  lru_remove(page);
  lru_insert_front(page);
}
```

LRU реализован как двусвязный список:

- При каждом обращении к странице (прочитали/записали/попали в кэш) её нужно сделать самой недавно использованной > `lru_move_to_front`.
- Новая загруженная страница также становится “самой свежей” > `lru_insert_front`.
- При вытеснении всегда удаляют хвост списка > `g_lru_tail` (самая старая страница), после чего `lru_remove`.

```
static int flush_page(CachePage* page) {
  if (!page->dirty)
    return 0;

  off_t  offset    = page->page_index * (off_t)g_page_size;
  size_t to_write  = g_page_size;
  size_t written   = 0;

  while (written < to_write) {
    ssize_t n = pwrite(page->fd,
                       page->data + written,
                       to_write - written,
                       offset + (off_t)written);
    if (n < 0) {
      if (errno == EINTR)
        continue;
      return -1;
    }
    if (n == 0) {
      errno = EIO;
      return -1;
    }
    written += (size_t)n;
  }

  page->dirty = 0;
  return 0;
}
```

В данном кэше используется политика “write-back”:

- При `vtpc_write` данные кладутся в кэш и помечаются `dirty=1`, но не пишутся сразу на диск.
- Перед тем как вытеснить страницу (или закрыть файл/сделать fsync), если `dirty=1`, вызывается `flush_page`, которая записывает полный блок `g_page_size` по смещению `page_index * page_size`.

Таким образом, LRU-вытеснение безопасно: грязные страницы не теряются, а предварительно сбрасываются.

```
static CachePage* alloc_page_struct(void) {
  for (int i = 0; i < MAX_CACHE_PAGES; ++i) {
    CachePage* p = &g_pages[i];
    if (!p->used) {
      p->used      = 1;
      p->fd        = -1;
      p->page_index = 0;
      p->dirty     = 0;
      p->lru_prev  = p->lru_next = NULL;
      p->hash_next = NULL;

      if (!p->data) {
        if (posix_memalign((void**)&p->data, g_page_size, g_page_size) != 0)
{
          p->used = 0;
          return NULL;
        }
      }

      return p;
    }
  }

  CachePage* victim = g_lru_tail;
  if (!victim)
    return NULL;

  if (victim->dirty) {
    if (flush_page(victim) != 0)
      return NULL;
  }

  g_cache_evictions++;
  hash_remove(victim);
  lru_remove(victim);

  victim->fd         = -1;
  victim->page_index = 0;
  victim->dirty      = 0;
  victim->hash_next  = NULL;

  return victim;
}
```

Это ключевая часть LRU-вытеснения.

1. Сначала код пытается найти свободный слот в пуле `g_pages[]`.
2. Если свободных нет, кэш заполнен > выбирается жертва `victim = g_lru_tail`, то есть страница, к которой обращались наиболее давно (LRU).
3. Если жертва грязная, выполняется `flush_page(victim)`.

4. Страница удаляется из двух структур:
    - из хеша (`hash_remove`) – чтобы больше не находилась по ключу,
    - из LRU-списка (`lru_remove`) – чтобы не считалась частью активного кэша.
5. Поля страницы сбрасываются, и структура переиспользуется под новую страницу.

Это даёт классическое поведение LRU: при переполнении вытесняется самая старая страница

```
static CachePage* get_page(int fd, off_t page_index) {
  vtpc_init();

  CachePage* page = hash_lookup(fd, page_index);
  if (page) {
    lru_move_to_front(page);
    g_cache_hits++;
    return page;
  }

  g_cache_misses++;

  page = alloc_page_struct();
  if (!page)
    return NULL;

  page->fd         = fd;
  page->page_index = page_index;
  page->dirty      = 0;

  off_t  offset   = page_index * (off_t)g_page_size;
  size_t to_read  = g_page_size;
  size_t done     = 0;

  while (done < to_read) {
    ssize_t n = pread(fd,
                      page->data + done,
                      to_read - done,
                      offset + (off_t)done);
    if (n < 0) {
      if (errno == EINTR)
        continue;
      page->used = 0;
      return NULL;
    }
    if (n == 0)
      break; /* EOF */
    done += (size_t)n;
  }

  if (done < to_read)
    memset(page->data + done, 0, to_read - done);

  hash_insert(page);
  lru_insert_front(page);

  return page;
}
```

Функция `get_page` – единая точка доступа к странице, в ней и реализуется логика LRU:

- Cache hit: страница найдена в `hash_lookup` > считается использованной сейчас, поэтому перемещается в начало LRU-списка `lru_move_to_front(page)`.
- Cache miss: страницы нет > выделяется слот `alloc_page_struct()` (возможно с вытеснением LRU-tail) и данные читаются с диска `pread`. После загрузки:
    - страница добавляется в хеш `hash_insert(page)`,
    - страница становится MRU `lru_insert_front(page)`.

Именно эти два действия (перемещение при hit и вставка в голову при miss) обеспечивают корректное поддержание LRU-порядка.

```
CachePage* page = get_page(fd, page_index);
if (!page) {
  if (total == 0)
    return -1;
  break;
}

memcpy((char*)buf + total, page->data + offset_in_page, chunk);
```

`vtpc_read` работает через `get_page`. Поэтому:

- при попадании в кэш страница переносится в начало LRU,
- при промахе загружается и также оказывается в начале.

```
CachePage* page = get_page(fd, page_index);
if (!page) {
  if (total == 0)
    return -1;
  break;
}

memcpy(page->data + offset_in_page,
       (const char*)buf + total,
       chunk);

page->dirty = 1;
lru_move_to_front(page);
```

`vtpc_write` тоже получает страницу через `get_page`, а затем:

- изменяет данные в `page->data`,
- помечает страницу `dirty=1` (данные нужно будет сбросить),
- явно делает её MRU через `lru_move_to_front(page)`.

Это логично: страница, в которую только что писали, точно свежая и не должна быть вытеснена первой.

```
for (int i = 0; i < MAX_CACHE_PAGES; ++i) {
  CachePage* p = &g_pages[i];
  if (p->used && p->fd == fd) {
    if (p->dirty) {
      if (flush_page(p) != 0 && rc == 0) {
        rc = -1;
        saved_errno = errno;
```

```
      }
    }
    hash_remove(p);
    lru_remove(p);
    p->used      = 0;
    p->fd        = -1;
    p->page_index = 0;
    p->dirty     = 0;
  }
}
```

`vtpc_close` чистит кэш-страницы, относящиеся к конкретному `fd`:

- сначала гарантирует сохранность данных (`flush_page` для dirty),
- затем удаляет страницу из обеих структур (`hash_remove`, `lru_remove`),
- освобождает слот (`used=0`), чтобы его можно было заново занять.

```
CachePage* p = &g_pages[i];
if (p->used && p->fd == fd && p->dirty) {
  if (flush_page(p) != 0 && rc == 0) {
    rc = -1;
    saved_errno = errno;
  }
}
```

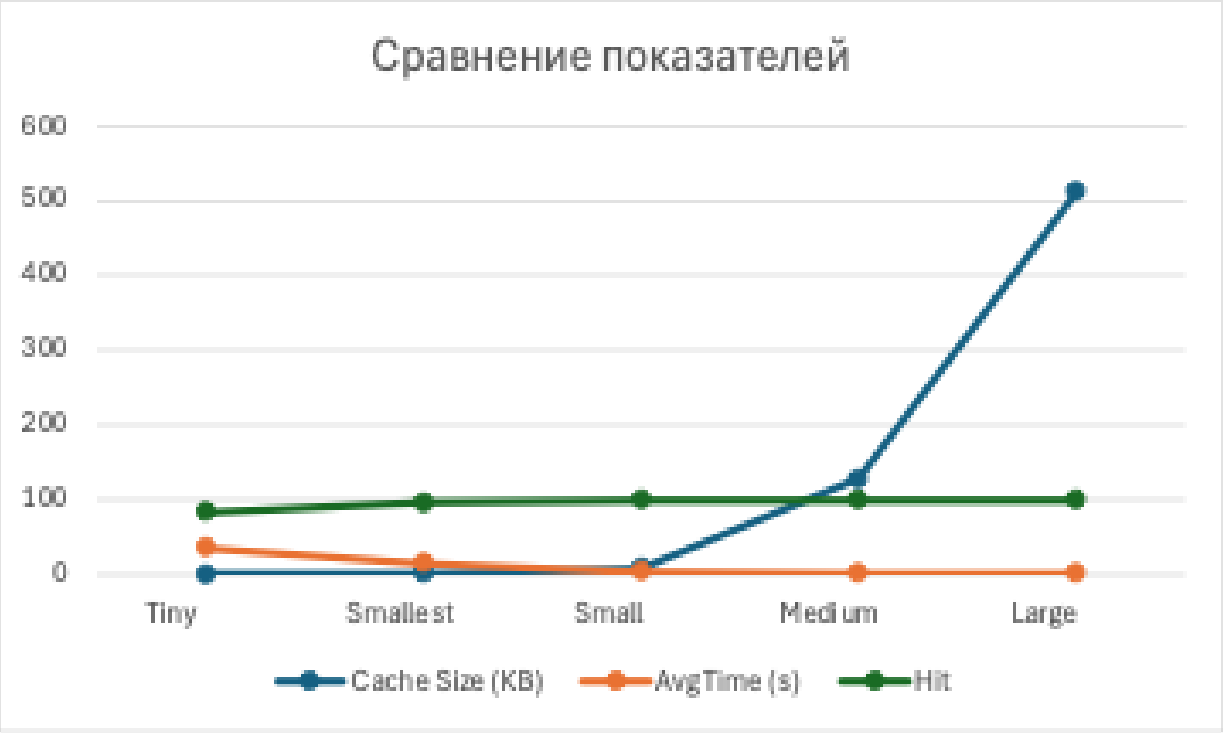
`vtpc_fsync` реализует принудительный сброс кэша на диск, но не меняет порядок LRU: страницы остаются в кэше, просто становятся `dirty=0`.

Результаты измерений при разных конфигурациях кэша

```
Running tests with different cache configurations...

========================================

=== Configuration: Tiny ===
Block Size: 32 bytes
Cache Capacity: 8 pages
Total Cache Size: 0 KB

Running external sort test...
Iteration 1/5...
Iteration 1 completed in 33.709 seconds
Iteration 2/5...
Iteration 2 completed in 31.888 seconds
Iteration 3/5...
Iteration 3 completed in 31.804 seconds
Iteration 4/5...
Iteration 4 completed in 31.187 seconds
Iteration 5/5...
Iteration 5 completed in 46.042 seconds
All 5 iterations completed in 174.630 seconds (avg: 34.926 sec)
Cache Statistics:
  Hits: 122458165
  Misses: 23791835
  Evictions: 23791475
  Hit Rate: 83.73%

----------------------------------------
```

```
=== Configuration: Smallest ===
Block Size: 128 bytes
Cache Capacity: 8 pages
Total Cache Size: 1 KB

Running external sort test...
Iteration 1/5...
Iteration 1 completed in 15.040 seconds
Iteration 2/5...
Iteration 2 completed in 16.944 seconds
Iteration 3/5...
Iteration 3 completed in 15.411 seconds
Iteration 4/5...
Iteration 4 completed in 16.080 seconds
Iteration 5/5...
Iteration 5 completed in 12.578 seconds
All 5 iterations completed in 76.053 seconds (avg: 15.211 sec)
Cache Statistics:
  Hits: 135625000
  Misses: 5937500
  Evictions: 5937140
  Hit Rate: 95.81%

----------------------------------------

=== Configuration: Small ===
Block Size: 512 bytes
Cache Capacity: 16 pages
Total Cache Size: 8 KB

Running external sort test...
Iteration 1/5...
Iteration 1 completed in 3.562 seconds
Iteration 2/5...
Iteration 2 completed in 3.972 seconds
Iteration 3/5...
Iteration 3 completed in 3.533 seconds
Iteration 4/5...
Iteration 4 completed in 3.511 seconds
Iteration 5/5...
Iteration 5 completed in 3.557 seconds
All 5 iterations completed in 18.136 seconds (avg: 3.627 sec)
Cache Statistics:
  Hits: 138915140
  Misses: 1484470
  Evictions: 1483750
  Hit Rate: 98.94%

----------------------------------------

=== Configuration: Medium ===
Block Size: 2048 bytes
Cache Capacity: 64 pages
Total Cache Size: 128 KB

Running external sort test...
Iteration 1/5...
Iteration 1 completed in 1.572 seconds
Iteration 2/5...
Iteration 2 completed in 1.394 seconds
Iteration 3/5...
Iteration 3 completed in 1.835 seconds
Iteration 4/5...
Iteration 4 completed in 1.574 seconds
```

```
Iteration 5/5...
Iteration 5 completed in 1.512 seconds
All 5 iterations completed in 7.886 seconds (avg: 1.577 sec)
Cache Statistics:
  Hits: 139789160
  Misses: 371260
  Evictions: 368380
  Hit Rate: 99.74%

----------------------------------------

=== Configuration: Large ===
Block Size: 4096 bytes
Cache Capacity: 128 pages
Total Cache Size: 512 KB

Running external sort test...
Iteration 1/5...
Iteration 1 completed in 1.045 seconds
Iteration 2/5...
Iteration 2 completed in 1.056 seconds
Iteration 3/5...
Iteration 3 completed in 1.045 seconds
Iteration 4/5...
Iteration 4 completed in 1.338 seconds
Iteration 5/5...
Iteration 5 completed in 1.164 seconds
All 5 iterations completed in 5.647 seconds (avg: 1.129 sec)
Cache Statistics:
  Hits: 139925940
  Misses: 185640
  Evictions: 179880
  Hit Rate: 99.87%

----------------------------------------

=== Test Summary ===

Configuration Comparison:

Config          Block Size   Capacity     Cache Size (KB) Avg Time (s) Hit
Rate (%)
--------------------------------------------------------------------------
-----------
Tiny            32           8            0               34.926       83.73
Smallest        128          8            1               15.211       95.81
Small           512          16           8               3.627        98.94
Medium          2048         64           128             1.577        99.74
Large           4096         128          512             1.129        99.87
```



Результаты показывают, что производительность внешней сортировки сильно зависит от размера страницы (block size) и ёмкости кэша (числа страниц): при малом кэше возникает почти постоянное вытеснение (thrashing), что резко увеличивает число промахов и время выполнения.

LRU-кэш эффективно ускоряет external sort, но только при достаточном размере страницы и/или ёмкости. Переход от Tiny к Medium/Large уменьшает число промахов на порядки и ускоряет выполнение до ~31× (34.93 с > 1.13 с). При этом Medium выглядит лучшим компромиссом: почти максимальный hit rate и время близкое к Large, но при в 4 раза меньшем размере кэша.

Проверим значения при одинаковых значениях страниц в кэше и разных размерах блоков.

При 8 страницах:

Config	Block Size	Capacity	Cache Size (KB)	Avg Time (s)	Hit Rate (%)
Tiny	32	8	0	44.960	83.73
Smallest	128	8	1	10.377	95.81
Small	512	8	4	3.747	98.94
Medium	2048	8	16	1.555	99.74
Large	4096	8	32	1.017	99.87

При 16 страницах:

Config	Block Size	Capacity	Cache Size (KB)	Avg Time (s)	Hit Rate (%)
Tiny	32	16	0	35.226	83.76
Smallest	128	16	2	10.248	95.81
Small	512	16	8	3.641	98.94

```
Medium          2048         16           32              1.484        99.74
Large           4096         16           64              1.073        99.87
```

При 64 страницах:

```
Config          Block Size   Capacity     Cache Size (KB) Avg Time (s) Hit
Rate (%)
-------------------------------------------------------------------------
-----------
Tiny            32           64           2               46.339       83.76
Smallest        128          64           8               9.410        95.81
Small           512          64           32              3.633        98.94
Medium          2048         64           128             1.605        99.74
Large           4096         64           256             1.096        99.87
```

При 128 страницах:

```
Config          Block Size   Capacity     Cache Size (KB) Avg Time (s) Hit
Rate (%)
-------------------------------------------------------------------------
-----------
Tiny            32           128          4               34.260       83.76
Smallest        128          128          16              10.722       95.81
Small           512          128          64              3.598        98.94
Medium          2048         128          256             1.565        99.74
Large           4096         128          512             1.031        99.87
```

В данном внешнем сортировании характер обращений такой, что увеличение глубины кэша (числа страниц) даёт небольшой эффект, а вот увеличение размера страницы резко снижает количество обращений/промахов и системных операций.

Теперь наоборот посмотрим на показатель при одинаковых размерах блока, но разных размерах страниц.

При размере 32 байта:

```
Config          Block Size   Capacity     Cache Size (KB) Avg Time (s) Hit
Rate (%)
-------------------------------------------------------------------------
-----------
Tiny            32           8            0               41.104       83.73
Smallest        32           8            0               41.331       83.73
Small           32           16           0               34.810       83.76
Medium          32           64           2               37.958       83.76
Large           32           128          4               37.862       83.76
```

При размере 128 байтов:

```
Config          Block Size   Capacity     Cache Size (KB) Avg Time (s) Hit
Rate (%)
-------------------------------------------------------------------------
-----------
Tiny            128          8            1               9.970        95.81
Smallest        128          8            1               10.301       95.81
Small           128          16           2               9.714        95.81
Medium          128          64           8               10.163       95.81
Large           128          128          16              10.315       95.81
```

При размере 512 байтов:

Config	Block Size	Capacity	Cache Size (KB)	Avg Time (s)	Hit Rate (%)
Tiny	512	8	4	3.615	98.94
Smallest	512	8	4	3.648	98.94
Small	512	16	8	3.542	98.94
Medium	512	64	32	3.493	98.94
Large	512	128	64	3.645	98.94

При размере 2048 байтов:

Config	Block Size	Capacity	Cache Size (KB)	Avg Time (s)	Hit Rate (%)
Tiny	2048	8	16	1.526	99.74
Smallest	2048	8	16	1.587	99.74
Small	2048	16	32	1.535	99.74
Medium	2048	64	128	1.575	99.74
Large	2048	128	256	1.481	99.74

При размере 4096 байтов:

Config	Block Size	Capacity	Cache Size (KB)	Avg Time (s)	Hit Rate (%)
Tiny	4096	8	32	1.103	99.87
Smallest	4096	8	32	1.056	99.87
Small	4096	16	64	1.036	99.87
Medium	4096	64	256	1.069	99.87
Large	4096	128	512	1.112	99.87

Вывод

В ходе лабораторной работы была разработана собственная командная оболочка с поддержкой запуска внешних программ и логических операторов, а также реализованы и исследованы различные типы нагрузчиков на подсистемы CPU и ввода-вывода. Проведённый анализ показал, как различаются поведение и эффективность многопроцессной, многопоточной и оптимизированной реализации в зависимости от характера нагрузки. Полученные результаты подтвердили ожидания и позволили глубже понять работу операционной системы под нагрузкой.