Университет ИТМО
Факультет программной инженерии и компьютерной техники

Лабораторная работа №2

По дисциплине «Операционные Системы»

Выполнили:
Студенты групп P3331
Нодири Хисравхон
Вариант: Aging

Преподаватель:
Лисицина Василиса Васильевна

г. Санкт-Петербург
2024 г.

Содержание

1 Задание **3**
1.1 Требования к реализации 3
1.2 Проверка работоспособности 4
1.3 Ограничения 4

2 Листинг исходного кода **5**

3 Данные о работе программы-нагрузчика до и после внедрения своего page cache **12**
3.1 Данные программы-нагрузчика без кэша 12
3.2 Данные программы-нагрузчика с кэшем 13

4 Заключение с анализом результатов и выводом **14**
4.1 Сравнение времени и загрузки процессора 14
4.2 Контекст-переключения и загрузка памяти 14
4.3 Показатели производительности (cycles, instructions) 15
4.4 Системные вызовы (по `strace`) 15
4.5 Общий вывод 15

1 Задание

Для оптимизации работы с блочными устройствами в операционных системах (ОС) существует кэш страниц с данными, используемыми для операций чтения и записи на диск. Такой кэш позволяет избежать высоких задержек при повторном доступе к данным, поскольку операция будет выполнена с использованием данных в оперативной памяти (RAM), а не с диска (вспомним пирамиду памяти).

В данной лабораторной работе необходимо реализовать блочный кэш в пространстве пользователя в виде динамической библиотеки (dll или so). Политику вытеснения страниц и другие элементы задания необходимо получить у преподавателя.

1.1 Требования к реализации

При выполнении работы необходимо реализовать простой API для работы с файлами, предоставляющий пользователю следующие возможности:

- **Открытие файла** по заданному пути файла, доступного для чтения. Процедура возвращает некоторый хэндл на файл. Пример:

```
int lab2_open(const char *path);
```

- **Закрытие файла** по хэндлу. Пример:

```
int lab2_close(int fd);
```

- **Чтение данных** из файла. Пример:

```
ssize_t lab2_read(int fd, void buf[.count], size_t count);
```

- **Запись данных** в файл. Пример:

```
ssize_t lab2_write(int fd, const void buf[.count], size_t count);
```

- **Перестановка позиции указателя** на данные файла. Достаточно поддержать только абсолютные координаты. Пример:

```
off_t lab2_lseek(int fd, off_t offset, int whence);
```

- **Синхронизация данных** из кэша с диском. Пример:

```
int lab2_fsync(int fd);
```

Операции с диском разработанного блочного кэша должны производиться в обход page cache используемой ОС.

1.2 Проверка работоспособности

В рамках проверки работоспособности разработанного блочного кэша необходимо:

1. Адаптировать указанную преподавателем программу-загрузчик из Лабораторной работы №1, добавив использование кэша.
2. Запустить программу и убедиться, что она корректно работает.
3. Сравнить производительность до и после использования блочного кэша.

1.3 Ограничения

- Программа (комплекс программ) должна быть реализована на языке C или C++.
- Если по выданному варианту задана политика вытеснения Optimal, то необходимо предоставить пользователю возможность подсказать page cache, когда будет совершен следующий доступ к данным. Это можно сделать либо добавив параметр в процедуры `read` и `write` (например, `ssize_t lab2_read(int fd, void buf[.count], size_t count, access_hint_t hint)`), либо добавив еще одну функцию в API (например, `int lab2_advice(int fd, off_t offset, access_hint_t hint)`). `access_hint_t` в данном случае – это абсолютное время или временной интервал, по которому разработанное API будет определять время последующего доступа к данным.
- Запрещено использовать высокоуровневые абстракции над системными вызовами. Необходимо использовать, в случае Unix, процедуры `libc`.

2 Листинг исходного кода

```
#define _GNU_SOURCE
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <fcntl.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <errno.h>

#include "lab2.h"

#define BLOCK_SIZE 4096

#define CACHE_CAPACITY 1024

#define MAX_FILES 128

//
typedef struct {
    off_t block_number;  //                                    (
       block = offset / BLOCK_SIZE)
    char *data;          //
                        (                                  )
    int valid;           // 1,                                ; 0

    int dirty;           // 1,

    unsigned int age;    //                 "                 "
    unsigned int refbit; //                  -       (
                            age)
} cache_block_t;

//
typedef struct {
    int real_fd;     //
                                  (     open)
    off_t offset;    //

    int in_use;      // 1,          fd                        ; 0,

} file_info_t;


static cache_block_t g_cache[CACHE_CAPACITY];

static file_info_t g_fd_table[MAX_FILES];

static int g_cache_initialized = 0;

```

```c
static void *aligned_alloc_block(size_t size) {
    void *ptr = NULL;
    int ret = posix_memalign(&ptr, BLOCK_SIZE, size);
    if (ret != 0) {
        return NULL;
    }
    memset(ptr, 0, size);
    return ptr;
}

static void init_cache_if_needed() {
    if (g_cache_initialized) return;

    for (int i = 0; i < CACHE_CAPACITY; i++) {
        g_cache[i].block_number = -1;
        g_cache[i].valid = 0;
        g_cache[i].dirty = 0;
        g_cache[i].age = 0;
        g_cache[i].refbit = 0;
        g_cache[i].data = (char *)aligned_alloc_block(BLOCK_SIZE);
        if (!g_cache[i].data) {
            fprintf(stderr, "Failed to allocate aligned memory for cache block\n");
        }
    }

    for (int i = 0; i < MAX_FILES; i++) {
        g_fd_table[i].real_fd = -1;
        g_fd_table[i].offset  = 0;
        g_fd_table[i].in_use  = 0;
    }

    g_cache_initialized = 1;
}

static int find_block_in_cache(int fd_idx, off_t block_num) {
    (void)fd_idx;
    for (int i = 0; i < CACHE_CAPACITY; i++) {
        if (g_cache[i].valid && g_cache[i].block_number == block_num) {
            return i;
        }
    }
    return -1;
}

//                    "aging tick"                              ,
//                                          -                        .
//           : age = (age >> 1) | (refbit << (          -
//                         )), refbit                          .
static void perform_aging() {
    for (int i = 0; i < CACHE_CAPACITY; i++) {
        g_cache[i].age >>= 1;
```

```
92          if (g_cache[i].refbit) {
93              g_cache[i].age |= (1U << 31);
94          }
95          g_cache[i].refbit = 0;
96      }
97  }
98
99  static int find_victim_block() {
100     perform_aging();
101
102     unsigned int min_age = (unsigned int)-1;
103     int victim_idx = -1;
104
105     for (int i = 0; i < CACHE_CAPACITY; i++) {
106         if (!g_cache[i].valid) {
107             return i;
108         }
109         if (g_cache[i].age < min_age) {
110             min_age = g_cache[i].age;
111             victim_idx = i;
112         }
113     }
114     return victim_idx;
115 }
116
117 static void flush_block_to_disk(int real_fd, cache_block_t *block) {
118     if (!block->valid || !block->dirty) return;
119
120     off_t offset = block->block_number * BLOCK_SIZE;
121     if (lseek(real_fd, offset, SEEK_SET) == (off_t)-1) {
122         perror("lseek (flush_block_to_disk)");
123         return;
124     }
125
126     ssize_t written = write(real_fd, block->data, BLOCK_SIZE);
127     if (written < 0) {
128         perror("write (flush_block_to_disk)");
129     } else if (written != BLOCK_SIZE) {
130         fprintf(stderr, "Partial write in flush_block_to_disk: %zd\n"
            , written);
131     }
132
133     block->dirty = 0;
134 }
135
136 static void read_block_from_disk(int real_fd, cache_block_t *block,
    off_t block_num) {
137     off_t offset = block_num * BLOCK_SIZE;
138     if (lseek(real_fd, offset, SEEK_SET) == (off_t)-1) {
139         perror("lseek (read_block_from_disk)");
140     }
141
142     ssize_t bytes_read = read(real_fd, block->data, BLOCK_SIZE);
```

```c
    if (bytes_read < 0) {
        perror("read␣(read_block_from_disk)");
    } else if (bytes_read < BLOCK_SIZE) {
        memset(block->data + bytes_read, 0, BLOCK_SIZE - bytes_read);
    }

    block->block_number = block_num;
    block->valid = 1;
    block->dirty = 0;
    block->age = 0;
    block->refbit = 1; //
}

//                                  .
                                 ,                          ,
                                    .
static int get_block_for_read(int fd_idx, off_t block_num) {
    int cache_idx = find_block_in_cache(fd_idx, block_num);
    if (cache_idx >= 0) {
        g_cache[cache_idx].refbit = 1; //
        return cache_idx;
    }

    //
    cache_idx = find_victim_block();
    if (cache_idx < 0) {
        //
                             ,
        return -1;
    }

    flush_block_to_disk(g_fd_table[fd_idx].real_fd, &g_cache[cache_idx]);

    read_block_from_disk(g_fd_table[fd_idx].real_fd, &g_cache[cache_idx], block_num);

    return cache_idx;
}

static int get_block_for_write(int fd_idx, off_t block_num) {
    int cache_idx = get_block_for_read(fd_idx, block_num);
    return cache_idx;
}

int lab2_open(const char *path) {
    init_cache_if_needed();

    int fd_idx = -1;
    for (int i = 0; i < MAX_FILES; i++) {
        if (!g_fd_table[i].in_use) {
            fd_idx = i;
            break;
```

```
        }
    }
    if (fd_idx < 0) {
        errno = EMFILE; //
        return -1;
    }

    int real_fd = open(path, O_RDWR | O_CREAT | O_DIRECT | O_SYNC,
       0666);
    if (real_fd < 0) {
        perror("open");
        return -1;
    }

    g_fd_table[fd_idx].real_fd = real_fd;
    g_fd_table[fd_idx].offset  = 0;
    g_fd_table[fd_idx].in_use  = 1;

    return fd_idx;
}

int lab2_close(int fd) {
    if (fd < 0 || fd >= MAX_FILES || !g_fd_table[fd].in_use) {
        errno = EBADF;
        return -1;
    }

    lab2_fsync(fd);

    int real_fd = g_fd_table[fd].real_fd;
    close(real_fd);

    g_fd_table[fd].real_fd = -1;
    g_fd_table[fd].offset  = 0;
    g_fd_table[fd].in_use  = 0;

    return 0;
}

ssize_t lab2_read(int fd, void *buf, size_t count) {
    if (fd < 0 || fd >= MAX_FILES || !g_fd_table[fd].in_use) {
        errno = EBADF;
        return -1;
    }

    file_info_t *finfo = &g_fd_table[fd];
    size_t bytes_read_total = 0;

    while (count > 0) {
        off_t block_num = finfo->offset / BLOCK_SIZE;
        size_t block_offset = finfo->offset % BLOCK_SIZE;

        size_t bytes_in_block = BLOCK_SIZE - block_offset;
```

```
243         size_t to_read = (count < bytes_in_block) ? count :
               bytes_in_block;
244
245         int cache_idx = get_block_for_read(fd, block_num);
246         if (cache_idx < 0) {
247             if (bytes_read_total == 0) {
248                 return -1;
249             }
250             break; //                              ,
251         }
252
253         memcpy((char *)buf + bytes_read_total,
254                g_cache[cache_idx].data + block_offset,
255                to_read);
256
257         bytes_read_total += to_read;
258         count -= to_read;
259         finfo->offset += to_read;
260
261     }
262
263     return bytes_read_total;
264 }
265
266 ssize_t lab2_write(int fd, const void *buf, size_t count) {
267     if (fd < 0 || fd >= MAX_FILES || !g_fd_table[fd].in_use) {
268         errno = EBADF;
269         return -1;
270     }
271
272     file_info_t *finfo = &g_fd_table[fd];
273     size_t bytes_written_total = 0;
274
275     while (count > 0) {
276         off_t block_num = finfo->offset / BLOCK_SIZE;
277         size_t block_offset = finfo->offset % BLOCK_SIZE;
278
279         size_t bytes_in_block = BLOCK_SIZE - block_offset;
280         size_t to_write = (count < bytes_in_block) ? count :
               bytes_in_block;
281
282         int cache_idx = get_block_for_write(fd, block_num);
283         if (cache_idx < 0) {
284             if (bytes_written_total == 0) {
285                 return -1;
286             }
287             break;
288         }
289
290         memcpy(g_cache[cache_idx].data + block_offset,
291                (char *)buf + bytes_written_total,
292                to_write);
```

```
293
294             g_cache[cache_idx].dirty = 1;
295             g_cache[cache_idx].refbit = 1;
296
297             bytes_written_total += to_write;
298             count -= to_write;
299             finfo->offset += to_write;
300         }
301
302         return bytes_written_total;
303     }
304
305     off_t lab2_lseek(int fd, off_t offset, int whence) {
306         if (fd < 0 || fd >= MAX_FILES || !g_fd_table[fd].in_use) {
307             errno = EBADF;
308             return (off_t)-1;
309         }
310
311         if (whence != SEEK_SET) {
312             errno = EINVAL;
313             return (off_t)-1;
314         }
315
316         g_fd_table[fd].offset = offset;
317         return offset;
318     }
319
320     int lab2_fsync(int fd) {
321         if (fd < 0 || fd >= MAX_FILES || !g_fd_table[fd].in_use) {
322             errno = EBADF;
323             return -1;
324         }
325
326         int real_fd = g_fd_table[fd].real_fd;
327
328         for (int i = 0; i < CACHE_CAPACITY; i++) {
329             if (g_cache[i].valid && g_cache[i].dirty) {
330                 flush_block_to_disk(real_fd, &g_cache[i]);
331             }
332         }
333
334         if (fsync(real_fd) < 0) {
335             perror("fsync");
336             return -1;
337         }
338
339         return 0;
340     }
```

lab2.c

3 Данные о работе программы-нагрузчика до и после внедрения своего page cache

3.1 Данные программы-нагрузчика без кэша

```
# started on Sat Jan 25 20:36:00 2025


 Performance counter stats for './bench 10000':

          1406.01 msec task-clock                #    0.585 CPUs
             utilized
            10009      context-switches          #    7.119 K/sec
                9      cpu-migrations            #    6.401 /sec
               63      page-faults               #   44.808 /sec
        583205436      cycles                    #    0.415 GHz
         32422414      stalled-cycles-frontend   #    5.56% frontend
              cycles idle
         69946877      stalled-cycles-backend    #   11.99% backend
              cycles idle
        201670631      instructions              #    0.35  insn per
             cycle
                                                 #    0.35  stalled
                                                    cycles per insn
         49665468      branches                  #   35.324 M/sec
                0      branch-misses             #    0.00% of all
                   branches

       2.402313142 seconds time elapsed

       0.000000000 seconds user
       1.078255000 seconds sys
```

perf_stat.txt

```
% time     seconds  usecs/call     calls    errors syscall
------ ----------- ----------- --------- --------- ----------------
 86.45    0.178352          17     10001           write
 13.50    0.027849           2     10000           lseek
  0.02    0.000032          10         3           openat
  0.01    0.000017           5         3           brk
  0.01    0.000014          14         1           munmap
  0.00    0.000009           9         1           fsync
  0.00    0.000009           9         1           getrandom
  0.00    0.000008           2         3           newfstatat
  0.00    0.000008           8         1           prlimit64
  0.00    0.000006           2         3           close
  0.00    0.000000           0         1           read
  0.00    0.000000           0         8           mmap
  0.00    0.000000           0         4           mprotect
  0.00    0.000000           0         4           pread64
  0.00    0.000000           0         1         1 access
```

```
  0.00    0.000000           0         1           execve
  0.00    0.000000           0         2         1 arch_prctl
  0.00    0.000000           0         1           set_tid_address
  0.00    0.000000           0         1           set_robust_list
  0.00    0.000000           0         1           rseq
------ ----------- ----------- --------- --------- ----------------
100.00    0.206304          10     20041         2 total
```

strace.txt

3.2 Данные программы-нагрузчика с кэшем

```
# started on Sat Jan 25 20:36:18 2025


 Performance counter stats for './bench_cache 10000':

             31.75 msec task-clock                #    0.243 CPUs
                utilized
               232      context-switches          #    7.306 K/sec
                 1      cpu-migrations            #   31.493 /sec
               197      page-faults               #    6.204 K/sec
          18424567      cycles                    #    0.580 GHz
            782976      stalled-cycles-frontend   #    4.25% frontend
                cycles idle
           1523840      stalled-cycles-backend    #    8.27% backend
              cycles idle
          10721121      instructions              #    0.58  insn per
              cycle
                                                  #    0.14  stalled
                                                     cycles per insn
           2727926      branches                  #   85.911 M/sec
                 0      branch-misses             #    0.00% of all
                    branches

       0.130912331 seconds time elapsed

       0.000000000 seconds user
       0.030871000 seconds sys
```

perf_stat_cache.txt

```
% time     seconds  usecs/call     calls    errors syscall
------ ----------- ----------- --------- --------- ----------------
 54.57    0.002984          45        65           write
 23.04    0.001260           9       128           lseek
 21.62    0.001182          17        66           read
  0.51    0.000028          14         2           fsync
  0.13    0.000007           1         4           close
  0.13    0.000007           0        33        27 newfstatat
  0.00    0.000000           0        13           mmap
  0.00    0.000000           0         5           mprotect
```

```
  0.00    0.000000           0         1           munmap
  0.00    0.000000           0         6           brk
  0.00    0.000000           0         4           pread64
  0.00    0.000000           0         1         1 access
  0.00    0.000000           0         1           execve
  0.00    0.000000           0         2         1 arch_prctl
  0.00    0.000000           0         1           set_tid_address
  0.00    0.000000           0        33        29 openat
  0.00    0.000000           0         1           set_robust_list
  0.00    0.000000           0         1           prlimit64
  0.00    0.000000           0         1           getrandom
  0.00    0.000000           0         1           rseq
------ ----------- ----------- --------- --------- ----------------
100.00    0.005468          14       369        58 total
```

strace_cache.txt

4 Заключение с анализом результатов и выводом

4.1 Сравнение времени и загрузки процессора

- **Без кэша (обычный `bench`):**
  - *task-clock:* 1406.01 мс
  - *Общее реальное время:* около 2.40 с
  - *Использование CPU:* 0.585
- **С кэшем (Aging, `bench_cache`):**
  - *task-clock:* 31.75 мс
  - *Общее реальное время:* около 0.13 с
  - *Использование CPU:* 0.243

Использование блочного кэша с алгоритмом Aging позволило сократить время выполнения с 1406 мс до 31 мс (более чем в 40 раз).

4.2 Контекст-переключения и загрузка памяти

- **Контекст-переключения:**
  - Без кэша: 10 009 переключений
  - С кэшем: 232 переключения

  Уменьшение числа переключений контекста указывает на более эффективное взаимодействие с ОС при наличии кэша.

- **Пробеги по памяти (Page Faults):**
  - Без кэша: 63
  - С кэшем: 197

Хоть и Page Faults с кэшем больше, общее время выполнения всё равно ниже т.к операция Fault быстро разрешается. enditemize

4.3 Показатели производительности (cycles, instructions)

- **Без кэша:**
  - 583 млн тактов при 0.415 ГГц
  - 201 млн инструкций
  - 0.35 инструкций за такт
- **С кэшем:**
  - 18 млн тактов при 0.580 ГГц
  - 10.7 млн инструкций
  - 0.58 инструкций за такт

4.4 Системные вызовы (по `strace`)

- **Без кэша:**
  - Основные вызовы: `write` (86.45% времени) и `lseek` (13.50%).
  - Всего системных вызовов: 20 041.
- **С кэшем (Aging):**
  - Основные вызовы: `write` (54.57%), `lseek` (23.04%), `read` (21.62%).
  - Всего системных вызовов: 369.

Резкое снижение количества системных вызовов (`write` и `lseek`) означает, что большая часть операций выполняется непосредственно в пользовательском кэше, а обращения к диску (т.е. "дорогие"операции) происходят реже.

4.5 Общий вывод

Алгоритм Aging в блочном кэше позволил достичь:

1. **Значительного уменьшения времени выполнения** — более чем в 40 раз.
2. **Сокращения числа контекст-переключений и обращений к диску**, благодаря чему снизилась нагрузка на системные вызовы.

3. **Повышения эффективности исполнения** (инструкций на цикл).

Внедрение блочного кэша с механизмом Aging существенно улучшает производительность программы за счёт меньшего количества операций чтения/записи на диск и более рационального использования оперативной памяти.