Федеральное государственное автономное образовательное учреждение высшего образования «Национальный исследовательский университет ИТМО»

Факультет программной инженерии и компьютерной техники

Отчёт
по лабораторной работе №2

по дисциплине «Операционные системы»

Вариант: *Linux FIFO*

Выполнил:
Рудкевич И. А., группа P3306

Преподаватель:
Тюрин И. Н.

Санкт-Петербург
~ 2024 ~

Оглавление

Задание

В данной лабораторной работе необходимо реализовать блочный кэш в пространстве пользователя в виде динамической библиотеки (.so). Политику вытеснения: Linux FIFO
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 кеша должны производиться в обход page cache используемой ОС.
В рамках проверки работоспособности разработанного блочного кэша необходимо адаптировать программу-загрузчик из ЛР 1, добавив использование кэша. Запустить программу и убедиться, что она корректно работает. Сравнить производительность до и после.

Листинг исходного кода

lib/src/cache.c

```
#include "../include/cache.h"

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

Cache cache = {NULL, NULL, 0};

CacheBlock *find_cache_block(int fd, off_t offset) {
  CacheBlock *block = cache.head;
  while (block != NULL) {
    if (block->fd == fd && block->offset == offset) {
      return block;
    }
    block = block->next;
  }
  return NULL;
}

void add_cache_block(int fd, off_t offset, const char *data) {
  // check if cache is full
  if (cache.current_size >= CACHE_SIZE) {
    CacheBlock *old_block = cache.head;
    if (old_block) {

      // write the data of the oldest block back to disk to persist any changes
      pwrite(old_block->fd, old_block->data, BLOCK_SIZE, old_block->offset);

      // remove the oldest block
      cache.head = old_block->next;
      if (cache.head)
        cache.head->prev = NULL;
      if (cache.tail == old_block)
        cache.tail = NULL;
      free(old_block);
      cache.current_size--;
    }
  }

  // create and add the new block to the tail (most recent)
  CacheBlock *new_block = (CacheBlock *)malloc(sizeof(CacheBlock));
  new_block->fd = fd;
  new_block->offset = offset;
  memcpy(new_block->data, data, BLOCK_SIZE);

  new_block->next = NULL;
  new_block->prev = cache.tail;
  if (cache.tail)
    cache.tail->next = new_block;
  cache.tail = new_block;
  if (cache.head == NULL)
    cache.head = new_block;
  cache.current_size++;
}

ssize_t read_block(int fd, off_t offset, char *buffer) {

  // look for the requested block in the cache
  CacheBlock *block = find_cache_block(fd, offset);
```

```
  if (block != NULL) {
    // if the block is found in the cache, copy its data to the buffer
    memcpy(buffer, block->data, BLOCK_SIZE);
    return BLOCK_SIZE;
  } else {
    // if the block is not found in the cache,
    // read from disk (and save to cache)
    ssize_t bytes_read = pread(fd, buffer, BLOCK_SIZE, offset);
    if (bytes_read > 0)
      add_cache_block(fd, offset, buffer); // add block to the cache
    return bytes_read;
  }
}

ssize_t write_block(int fd, off_t offset, const char *data) {

  // look for the requested block in the cache
  CacheBlock *block = find_cache_block(fd, offset);

  if (block != NULL) {
    // if the block is found in the cache, update its data
    memcpy(block->data, data, BLOCK_SIZE);
    return BLOCK_SIZE;
  } else {
    // add to cache, if the block is not found
    add_cache_block(fd, offset, data);
    return BLOCK_SIZE;
  }
}
```

lib/src/api.c

```
#include "../include/api.h"
#include "../include/cache.h"

#include <errno.h>
#include <fcntl.h>
#include <stdlib.h>
#include <string.h>
#include <sys/fcntl.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <unistd.h>

int c_open(const char *path) {
  int flags = O_RDWR | O_SYNC;

#if defined(__APPLE__)
  flags |= F_NOCACHE;

#elif defined(__linux__)
  flags |= O_DIRECT;
#endif

  // NOTE: F_NOCACHE for macos, O_DIRECT for linux
  return open(path, flags);
}

int c_close(int fd) {
  // remove fd from cache before closing it
  CacheBlock *block = cache.head;
  while (block != NULL) {
    if (block->fd == fd) {

      // if the block is at the head of the cache
      if (block == cache.head) {
        cache.head = block->next;
        if (cache.head)
          cache.head->prev = NULL;
      }

      // if the block is at the tail of the cache
      if (block == cache.tail) {
        cache.tail = block->prev;
        if (cache.tail)
          cache.tail->next = NULL;
      }

      // if the block is somewhere in the middle
      if (block->prev != NULL) {
        block->prev->next = block->next;
      }
      if (block->next != NULL) {
        block->next->prev = block->prev;
      }

      free(block);
      cache.current_size--;
      break;
    }
    block = block->next;
  }

  return close(fd);
}

off_t align_offset(off_t offset) {
  return offset & ~(BLOCK_SIZE - 1);
}
```

```
ssize_t c_read(int fd, void *buf, size_t count) {
  off_t offset = lseek(fd, 0, SEEK_CUR);
  ssize_t bytes_read = 0;
  size_t remaining = count;

  while (remaining > 0) {
    // align the offset to the block boundary
    off_t block_offset = align_offset(offset);
    char block[BLOCK_SIZE];

    ssize_t bytes_in_block = read_block(fd, block_offset, block);
    if (bytes_in_block <= 0)
      return bytes_in_block;

    // copy data from the block to the buffer (either remaining or block size)
    size_t to_copy = remaining < bytes_in_block ? remaining : bytes_in_block;

    memcpy(buf + bytes_read, block + (offset - block_offset), to_copy);

    bytes_read += to_copy;
    remaining -= to_copy;
    offset += to_copy;
  }

  return bytes_read;
}

ssize_t c_write(int fd, const void *buf, size_t count) {
  off_t offset = lseek(fd, 0, SEEK_CUR);
  ssize_t bytes_written = 0;
  size_t remaining = count;

  while (remaining > 0) {
    // align the offset to the block boundary
    off_t block_offset = align_offset(offset);
    char block[BLOCK_SIZE];

    // copy data into block buffer (either remaining or full block size)
    size_t to_copy = remaining < BLOCK_SIZE ? remaining : BLOCK_SIZE;
    memcpy(block, buf + bytes_written, to_copy);

    ssize_t written = write_block(fd, block_offset, block);
    if (written <= 0)
      return written;

    bytes_written += to_copy;
    remaining -= to_copy;
    offset += to_copy;
  }

  return bytes_written;
}

int c_fsync(int fd) {
  // write all blocks back to disk
  CacheBlock *block = cache.head;
  while (block != NULL) {
    pwrite(block->fd, block->data, BLOCK_SIZE, block->offset);
    block = block->next;
  }
  return fsync(fd);
}

off_t c_lseek(int fd, off_t offset, int whence) {
  return lseek(fd, offset, whence);
}
```

src/ema_search_with_cache.c

```
#include "../lib/include/api.h"
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <time.h>
#include <unistd.h>

#define BUFFER_SIZE 128

int search_substring(const char *buffer, const char *substring) {
  int i, j;
  if (substring[0] == '\0') {
    return -1;
  }
  for (i = 0; buffer[i] != '\0'; i++) {
    for (j = 0; substring[j] != '\0'; j++) {
      if (buffer[i + j] != substring[j]) {
        break;
      }
    }
    if (substring[j] == '\0') {
      return i;
    }
  }
  return -1;
}

int process_file(int fd, const char *substring) {
  off_t offset = 0;
  int iteration = 0;
  char buffer[BUFFER_SIZE];

  while (1) {
    if (c_lseek(fd, offset, SEEK_SET) == (off_t)-1) {
      perror("Error seeking file");
      return EXIT_FAILURE;
    }

    ssize_t bytes_read = c_read(fd, buffer, BUFFER_SIZE);
    if (bytes_read <= 0) {
      break;
    }

    int index = search_substring(buffer, substring);
    if (index != -1) {
      printf("Found substring at position %lld\n", offset + index);
      break;
    }

    printf("Not Found substring at offset %lld\n", offset);

    iteration++;
    offset += bytes_read;
  }

  return EXIT_SUCCESS;
}

int main(int argc, char *argv[]) {
  if (argc < 4) {
    fprintf(stderr, "Usage: %s <filename> <substring> <repeats>\n", argv[0]);
    return EXIT_FAILURE;
  }

  const char *filename = argv[1];
  const char *substring = argv[2];
  size_t repeats = atoi(argv[3]);
```

```
  if (strlen(substring) > BUFFER_SIZE) {
    fprintf(stderr, "Substring is larger than buffer size\n");
    return EXIT_FAILURE;
  }

  int fd = c_open(filename);
  if (fd == -1) {
    perror("Error opening file");
    return EXIT_FAILURE;
  }

  clock_t total_start = clock();
  for (int r = 0; r < repeats; r++) {
    clock_t start = clock();
    if (process_file(fd, substring) != 0) {
      fprintf(stderr, "[%d] AHTUNG AHTUNG SOME ERROR OCCURRED\n", r);
    }
    printf("Execution time for repetition %d: %lf seconds\n\n", r,
           (double)(clock() - start) / CLOCKS_PER_SEC);
  }
  clock_t total_end = clock();

  c_close(fd); // Close the file using c_close instead of fclose

  printf("\n");
  printf(">>> Total execution time: %lf seconds <<<\n\n",
         (double)(total_end - total_start) / CLOCKS_PER_SEC);

  return EXIT_SUCCESS;
}
```

Измерения

Запуск программы-нагрузчика ema_search **без** использования разработанного кэша (одна итерация):

```
>>> Total execution time: 1.058498 seconds <<<
```

Запуск программы-нагрузчика ema_search **с** использованием разработанного кэша (одна итерация):

```
>>> Total execution time: 9.641296 seconds <<<
```

Заключение

Как можно заметить, использование разработанного кэша замедлило выполнение программы-нагрузчика в 9 раз. Такая деградация в производительности может быть обусловлена следующими факторами:

1) Политика вытеснения FIFO
2) Программа-нагрузчик "поиск подстроки во внешней памяти" читает данные по чанкам из внешней памяти. Если в текущем чанке подстрока не найдена, программа переходит к следующему. При этом к предыдущим чанкам повторного обращения нет, что ведёт к постоянным кэш-промахам.

Вывод

В ходе выполнения лабораторной работы я ознакомился с политикой вытеснения Linux FIFO, узнал как можно осуществлять операции с диском в обход page cache на Linux и MacOS. А также научился собирать проект в виде динамической библиотеки.