HA NOI UNIVERSITY OF SCIENCE AND TECHNOLOGY

A picture containing logo

Description automatically generated

**reports  
cOURSE: operating system**

**sub-project**

SCHOOL OF ELECTRICAL & ELECTRONIC ENGINEERING

**TABLE OF CONTENTS**

Student Name: Nguyen Dinh Thuan

Student ID : 20210827

Instructor: Assoc. Prof. Pham Van Tien

Hanoi University of Science and Technology

No 1 Dai Co Viet Road, Hanoi, Vietnam

1. Introduction……………………………………..…..2
2. Argument Passing……………………….…………3
3. Buffer Cache…………..…………………………..9
4. References………………………………………..13
5. **INTRODUCTION**
6. **Pintos**

* Pintos is an instructional operating system framework used primarily in educational settings to teach students about operating system design and implementation. It was developed at Stanford University by Ben Pfaff and others. The primary objectives of Pintos are to introduce students to the fundamental concepts of operating systems and to provide a hands-on experience in implementing these concepts.
* Objectives:
* Educational Tool: Pintos serves as a hands-on educational tool for teaching operating system concepts in a practical manner. It provides students with a platform to understand the inner workings of an operating system by implementing various components themselves.
* Modular Structure: Pintos is designed with a modular structure, allowing students to implement different parts of the operating system independently. It typically includes components such as threads, scheduling, memory management, file systems, and more, encouraging students to work on each module separately.
* Project-Based Learning: Pintos is organized as a series of programming projects, where students progressively build and expand the functionality of an operating system. These projects often start with simpler components and gradually move towards more complex concepts, allowing students to grasp one concept before moving on to the next.
* Understanding OS Concepts: Through implementing various components of an operating system, students gain a deeper understanding of core concepts such as process management, scheduling algorithms, memory management, file systems, and synchronization.
* Testing and Evaluation: Pintos provides a testing framework that allows students to evaluate their implementations by running test cases. This hands-on evaluation helps students debug their code and understand the expected behavior of different operating system functionalities.
* Real-World Application: While Pintos is a simplified educational tool, the concepts and principles learned through working on it can be applied to real-world operating system design and development, providing a solid foundation for students pursuing careers in systems programming or operating systems.

1. **Argument Passing**

* In an Operating Systems, the study of argument passing in user programs often revolves around how command-line arguments or parameters are passed to programs and how these arguments are accessed by the program during its execution**.**
* Objectives:
* Understanding Command-Line Arguments: To comprehend how command-line arguments are passed to user programs when they are invoked.
* Parameter Passing Mechanisms: Explore different parameter passing mechanisms such as passing arguments via registers, the stack, or a combination of both.
* Parsing and Processing Arguments: Learn how programs access and parse these arguments to utilize them during runtime.
* Security and Error Handling: Address security concerns related to argument passing, including buffer overflows or incorrect usage. Understanding error handling when handling arguments is also crucial.
* Standard Libraries and System Calls: Explore how standard libraries or system calls (like argc, argv in C/C++) are used to access command-line arguments and how they relate to the underlying mechanisms.

1. **Buffer Cache**

* Buffer cache is a crucial component of the file system responsible for caching disk blocks in memory. Its primary function is to improve overall system performance by reducing the number of disk accesses required for reading and writing data.
* Objectives:
* Improving File System Performance: The primary objective of the buffer cache is to enhance overall system performance by reducing the latency associated with disk I/O operations. Caching frequently accessed disk blocks in memory speeds up data retrieval.
* Minimizing Disk Access: By storing frequently used data in memory, the buffer cache minimizes the number of read and write operations performed directly on the slower disk storage, thus improving overall system responsiveness.
* Optimizing Resource Utilization: Efficiently managing memory resources is another objective. The buffer cache allocates memory for caching disk blocks while maintaining a balance between cached data and available system memory.
* Ensuring Data Consistency: A crucial objective is to ensure the consistency of data between the cached blocks in memory and the actual disk storage. This involves proper synchronization and flushing of modified data back to the disk.
* Concurrency and Protection: Guaranteeing safe access to cached data among multiple processes or threads is essential to prevent data corruption and maintain the integrity of the file system.

1. **ARGUMENT PASSING**
2. **Objectives and Algorithm**

Currently, process\_execute() does not support passing arguments to new processes. Argument passing should be implemented by extending process.c file. The objectives are:

* Using strtok\_r(), prototyped in lib/string.h to parse file name into an array of arguments, with the first element is the name of the executable.
* Setting up user stack of the program such that the argument data and address will be placed on top of the stack. To be more specific:
* First, after breaking down the input string into multiple words, we put the word on top of the user stack. Align the stack pointer downs to a multiple of 4 before the first push to enhance performance.
* Then, push the address of each string plus a null pointer sentinel, on the stack, in right-to-left order. These are the elements of argv. The null pointer sentinel ensures that argv[argc] is a null pointer, as required by the C standard. The order ensures that argv[0] is at the lowest virtual address.
* Then, push argv (the address of argv[0]) and argc, in that order. Finally, push a fake "return address": although the entry function will never return, its stack frame must have the same structure as any other.

1. **Code explanation**

**a. Parsing**

In function

bool load (const char \*file\_name\_ori, void (\*\*eip) (void), void \*\*esp),

I initialize three new variables: argv, argc, addresses.

* argc: the number of arguments
* argv: argument data, with a maximum of 32 arguments
* addresses: addresses of the argument in the memory of the program

  int argc = 0;

  char \*argv[32];

  char file\_name\_temp[strlen(file\_name\_ori) + 1];

  char \*file\_name = &file\_name\_temp[0];

  strlcpy (file\_name, file\_name\_ori, strlen(file\_name\_ori) + 1);

  argv[0] = file\_name;

  char \*token, \*save\_ptr;

  for (token = strtok\_r (file\_name, " ", &save\_ptr); token != NULL;

       token = strtok\_r (NULL, " ", &save\_ptr))

    {

      argv[argc] = token;

      argc++;

    }

  char \*addresses[argc + 1];

  addresses[argc] = 0;

Using strtok\_r(), I tokenize the input string into multiple tokens and store them in the argv array.

**b. Setting up user stack**

  for (i = argc - 1; i >= 0; i--)

    {

      \*esp -= (strlen (argv[i]) + 1);

      memcpy (\*esp, argv[i], strlen (argv[i]) + 1);

      addresses[i] = \*esp;

    }

This loop handles the arguments passed to the user program. It iterates through each argument (argv[]) in reverse order and copies each argument string onto the stack pointed to by \*esp. It also stores the address of each argument in the addresses[] array for later reference.

while ( ((long) \*esp) % 4 != 0)

    {

      \*esp -= 1;

      memset (\*esp, 0, 1);

    }

This loop ensures that the stack pointer \*esp is aligned on a 4-byte boundary. It pads the stack with zero bytes if necessary until it reaches a 4-byte alignment. This alignment is often required for correct memory access by many processors.

\*esp -= sizeof (addresses[0]) \* (argc + 1);

  memcpy (\*esp, &addresses[0], sizeof (addresses[0]) \* (argc + 1));

  /\* argv address \*/

  memcpy (\*esp - 4, esp, 4);

  \*esp -= 4;

This line moves the stack pointer \*esp down by the size needed to store the addresses of all the arguments plus an additional entry for the null pointer. It allocates space on the stack for the addresses[] array.

Here, it copies the addresses[] array onto the stack at the current \*esp position. This places all the argument addresses on the stack.

This block is preparing the "argv" address by copying the current value of the stack pointer \*esp (which points to the start of the addresses[] array) to the location just below it on the stack. It also decrements the stack pointer by 4 bytes to account for this stored address.

  \*esp -= 4;

  memcpy (\*esp, &argc, 4);

This section places the argument count (argc) onto the stack. It decrements the stack pointer by 4 bytes and copies the value of argc to that location on the stack.

  \*esp -= 4;

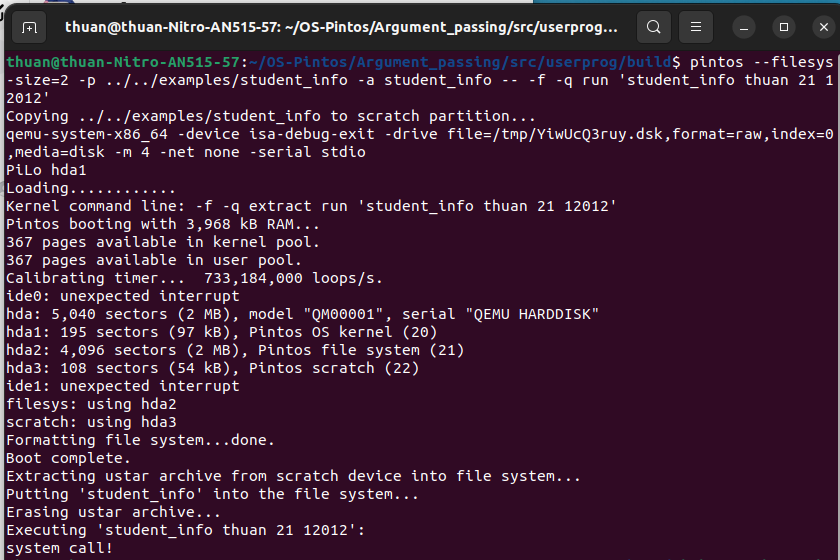
  memset (\*esp, 0, 4);

Finally, it prepares the "return address" by decrementing the stack pointer by 4 bytes and setting the memory at that location to zero. This is typically used for the return value when a function finishes execution.

**3. Result**

After running custom-made student\_info program, which takes 3 arguments: name, age and id, this is the result.

Since system calls haven’t been implemented, the message ‘system call!’ is printed, indicating the implementation of argument passing is successful



1. **BUFFER CACHE**

**1. Objectives and Algorithm**

Create a cache\_block struct to represent the buffer cache. The buffer cache should contain 64 cache blocks.

Using the clock algorithm to decide which block to evict when the cache is full.

**2. Coding explanation**

**a. Addition**

**cache.c**

struct buffer\_cache\_entry\_t {

  bool occupied;  // true only if this entry is valid cache entry

  block\_sector\_t disk\_sector;

uint8\_t buffer[BLOCK\_SECTOR\_SIZE];

  bool dirty;     // dirty bit

  bool access;    // reference bit, for clock algorithm

};

Each buffer cache entry has the following attributes:

* bool occupied: true if the cache block is being occupied by a block sector
* bool dirty: true if the data of the cache block has been modified
* bool access: reference bit for the clock algorithm
* block\_sector\_t disk\_sector: the sector the cache block is currently holding, if exists
* uint8\_t buffer[BLOCK\_SECTOR\_SIZE]; the array to store the block sector data

void buffer\_cache\_init (void)

{

  lock\_init (&buffer\_cache\_lock);

  // initialize entries

  size\_t i;

  for (i = 0; i < BUFFER\_CACHE\_SIZE; ++ i)

  {

    cache[i].occupied = false;

  }

}

The initialization method for the buffer cache. Called when filesys\_init ()

static void buffer\_cache\_flush (struct buffer\_cache\_entry\_t \*entry)

{

  ASSERT (lock\_held\_by\_current\_thread(&buffer\_cache\_lock));

  ASSERT (entry != NULL && entry->occupied == true);

  if (entry->dirty) {

    block\_write (fs\_device, entry->disk\_sector, entry->buffer);

    entry->dirty = false;

  }

}

A method to write back an entry cache if its dirty flag is true

void buffer\_cache\_close (void)

{

  // flush buffer cache entries

  lock\_acquire (&buffer\_cache\_lock);

  size\_t i;

  for (i = 0; i < BUFFER\_CACHE\_SIZE; ++ i)

  {

    if (cache[i].occupied == false) continue;

    buffer\_cache\_flush( &(cache[i]) );

  }

  lock\_release (&buffer\_cache\_lock);

}

A method to flush all entries in the buffer cache. Called when filesys\_done() and machine shutdown

static struct buffer\_cache\_entry\_t\*

buffer\_cache\_lookup (block\_sector\_t sector)

{

  size\_t i;

  for (i = 0; i < BUFFER\_CACHE\_SIZE; ++ i)

  {

    if (cache[i].occupied == false) continue;

    if (cache[i].disk\_sector == sector) {

      // cache hit.

      return &(cache[i]);

    }

  }

  return NULL; // cache miss

}

Lookup the cache entry, and returns the pointer of buffer\_cache\_entry\_t or NULL in case of cache miss. (simply traverse the cache entries)

static struct buffer\_cache\_entry\_t\*

buffer\_cache\_evict (void)

{

  ASSERT (lock\_held\_by\_current\_thread(&buffer\_cache\_lock));

  // clock algorithm

  static size\_t clock = 0;

  while (true) {

    if (cache[clock].occupied == false) {

      // found an empty slot -- use it

      return &(cache[clock]);

    }

    if (cache[clock].access) {

      // give a second chance

      cache[clock].access = false;

    }

    else break;

    clock ++;

    clock %= BUFFER\_CACHE\_SIZE;

  }

  // evict cache[clock]

  struct buffer\_cache\_entry\_t \*slot = &cache[clock];

  if (slot->dirty) {

    // write back into disk

    buffer\_cache\_flush (slot);

  }

  slot->occupied = false;

  return slot;

}

Obtain a free cache entry slot. If there is an unoccupied slot already, return it. Otherwise, some entry should be evicted by the clock algorithm.

Void buffer\_cache\_read (block\_sector\_t sector, void \*target)

{

  lock\_acquire (&buffer\_cache\_lock);

  struct buffer\_cache\_entry\_t \*slot = buffer\_cache\_lookup (sector);

  if (slot == NULL) {

    // cache miss: need eviction.

    slot = buffer\_cache\_evict ();

    ASSERT (slot != NULL && slot->occupied == false);

    // fill in the cache entry.

    slot->occupied = true;

    slot->disk\_sector = sector;

    slot->dirty = false;

    block\_read (fs\_device, sector, slot->buffer);

  }

  // copy the buffer data into memory.

  slot->access = true;

  memcpy (target, slot->buffer, BLOCK\_SECTOR\_SIZE);

  lock\_release (&buffer\_cache\_lock);

}

A method to read data from the cache block, which is loaded from the disk memory.

void

buffer\_cache\_write (block\_sector\_t sector, const void \*source)

{

  lock\_acquire (&buffer\_cache\_lock);

  struct buffer\_cache\_entry\_t \*slot = buffer\_cache\_lookup (sector);

  if (slot == NULL) {

    // cache miss: need eviction.

    slot = buffer\_cache\_evict ();

    ASSERT (slot != NULL && slot->occupied == false);

    // fill in the cache entry.

    slot->occupied = true;

    slot->disk\_sector = sector;

    slot->dirty = false;

    block\_read (fs\_device, sector, slot->buffer);

  }

  // copy the data form memory into the buffer cache.

  slot->access = true;

  slot->dirty = true;

  memcpy (slot->buffer, source, BLOCK\_SECTOR\_SIZE);

  lock\_release (&buffer\_cache\_lock);

}

A method to write data to the cache block, which is loaded to the corresponding disk memory.

**b. Modification**

**inode.c**

Substitute all `block\_read` & `block\_write` to `buffer\_cache\_read` & `buffer\_cache\_write`

**filesys.c**

void filesys\_init (bool format)

{

...

buffer\_cache\_init();

...

}

void filesys\_done (void)

{

...

/\* Write back dirty blocks when filesys is done \*/

Buffer\_cache\_close();

...

}

**3. Result**

Unfortunately, since system calls haven’t been implemented, especially system calls for write and read, buffer cache cannot be tested in anyway meaningfully. At its best, it will print “system call!” like in argument passing implementation

1. **REFERENCE**

[1] [CS140-Pintos project](https://www.scs.stanford.edu/23wi-cs212/labs/project.html)