Contents

List of Figures

1. Introduction

1.1 Background

Pintos is a simple operating system framework for the 80x86 architecture. Developed at the Stanford University, this kernel can be used as a base, by aspiring developers to design their own kernels from scratch to finish. The focus of this project was to build a modern kernel called *burritos* from the primitive pintos in a step by step approach.

1.2 The Journey

The journey 'from pintos to burritos' involved the following stages of development:-

1. <u>Design of the alarm clock and implementation of the threads system.</u>

Synchronization primitives like semaphores, locks and condition variables were designed and the primitive alarm clock of pintos was re-implemented using sleep and wakeup operations to eliminate the inefficient and dangerous 'busy waiting.'

A priority scheduler was built as a replacement for the round robin scheduler of pintos. Priority donation algorithms were designed to fix the priority inversion problem. Then the priority scheduler was further extended to a Multi-level Feedback Queue Scheduler similar to the 4.4 BSD Scheduler.

A deadlock avoidance algorithm was designed so that the threads in burritos never run into deadlocks.

2. <u>User processes and System Calls</u>

The thread system implemented in the previous stage was extended to create a hierarchy of single-threaded processes by the development of process management utilities like process creation and termination.

System calls for process management, memory management and file management were developed using interrupts. Threads in pintos could trespass into each other’s address space. All such attempts will now immediately cause a segmentation fault and terminate the offending process.

3. Virtual Memory implementation

A paging scheme was devised to load segments from executables lazily, that is, only as the kernel intercepts page faults for them. The paging scheme allows for sharing read-only pages among processes executing the same executable. A global page replacement algorithm was implemented as efficient as the 'clock' algorithm.

Memory mapped files were implemented by developing the *mmap* and *munmap* system calls to speed up file operations.

Stack which was originally fixed in size was made to grow past its current size by the allocation of additional pages till the system defined limit.

4. File Systems

A hierarchical namespace was implemented to extend pintos' concept of single directory to subdirectories in burritos.

1.3 Trivia

Burrito is a common Mexican dish and Pinto beans are important ingredients of its recipe. The operating system pintos originated as a replacement for the ‘nachos’ instructional operating system, developed at the University of California, Berkeley. Nachos, too is the name of a common Mexican food.

2. System Architecture

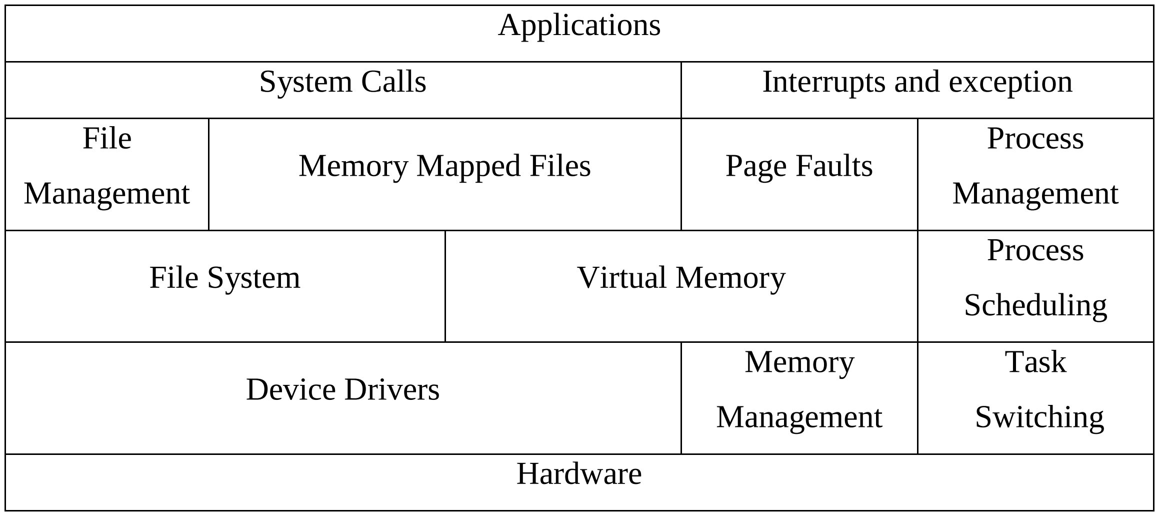

Figure 2.1

User applications, in course of their execution, may need to access many system resources like files on a disk, system memory and other devices. To accomplish this, user applications in burritos trap into the operating system through system calls. Burritos provides several system calls for file management to create/remove, open/close, read/write from or to files that are organized in a hierarchical file system. The file system makes use of the underlying device drivers to access data on the disk. The burritos kernel includes drivers for 8254 timer, VGA display, serial port, IDE disks and keyboard.

System memory is yet another very important resource used by the user applications. Users are allowed to access memory larger than physical memory by the virtual memory subsystem of burritos. The VM subsystem includes lazy loading of processes and swapping in and out of pages, employing global page replacement algorithm. It also interacts with the memory management utilities that draw support from the MMU of underlying architecture for memory addressing. These utilities are also responsible for all the memory allocations for user applications.

File operations in burritos can be speeded up by accessing the contents of the file from the physical memory rather than secondary storage. Burritos provides system calls for mapping and unmapping of files to and from the memory.

The burritos kernel manages user processes when they transit from user mode to kernel mode via interrupts. These user processes are dispatched to CPU one after the other based on the kernel's scheduling policies. The scheduler is similar to the 4.4 BSD scheduler. And the task switching unit of the kernel does the actual switching between the current process and the next process to run, selected by the scheduler.

Pintos came along with an interface to the 80x86 architecture that included device driver support, memory management utilities and task switching. It was extended to burritos by developing all the layers above it.

3. Design of the alarm clock and thread system

3.1 Design of the synchronization primitives

If sharing of resources between threads is not handled in a careful, controlled fashion, the result is usually a big mess. This is especially the case in operating system kernels, where faulty sharing can crash the entire machine. We design several synchronization primitives like semaphores, locks and condition variables to help out.

A counting semaphore has been implemented using a non-negative integer variable accompanied with a list of waiters containing the Process Control Blocks (PCBs) of the threads waiting on that semaphore. The Down and Up (P and V) operations on the semaphore variables are performed atomically by disabling interrupts. A broadcast operation has also been implemented to unblock all the threads waiting on the semaphore.

Next, a variant of the counting semaphore, the binary semaphore has been used to construct locks, which further are used to implement the monitor, a higher level synchronization primitive**.**

3.2 The alarm clock

3.2.1 Prelude

Modern operating systems require threads to suspend their execution for a period of time, for a number of applications. The threads in pintos do busy waiting to achieve this. Busy waiting not only wastes CPU cycles, but also may run the busy waiting threads into deadlocks in case of priority inversion. Burritos re-implements this by using sleep and wakeup operations, using the counting semaphore implemented in the previous stage.

3.2.2 Data Structures

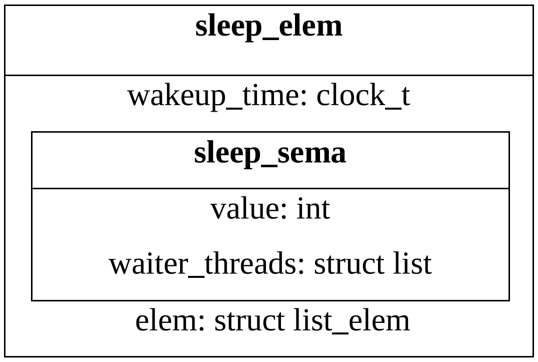

Figure 3.1

A list of sleep elements (sleep list) ordered by wakeup time is maintained. Each sleep element (Figure 3.1) is implemented using a semaphore on which all threads to be woken up at the same time will sleep.

3.2.3 Algorithm

<u>Sleep Operation</u>

First, the wakeup time is calculated using the current time and number of seconds to sleep. A sleep element is then created based on the wakeup time calculated and its semaphore is initialized to zero, so that any thread performing a down operation on it will block. It is then inserted at the appropriate position in the sleep list, if another sleep element with the same wakeup time is not present. Next, a down operation is performed on the semaphore of this sleep element so that the thread is blocked on that semaphore.

<u>Wakeup Operation</u>

When a timer interrupt occurs, the head node of the sleep list is examined to check if the threads blocked on its semaphore have to be woken up, by comparing its wakeup time with the current time. On successful comparison, a broadcast operation is performed on the semaphore, so that all the threads blocked on that semaphore are unblocked.

3.2.4 Synchronization

Race conditions can occur when:-

1. Multiple threads try to sleep simultaneously.
2. A timer interrupt occurs during a sleep operation.

Interrupts are disabled during the sleep operation so that the race conditions listed above can be avoided.

3.2.5 Rationale

Earlier, the design involved inserting the sleep element at the end of the sleep list during the sleep operation; and the wakeup operation involved traversing the sleep list until a sleep element is found whose wakeup time matches with the current time.

The time complexity of the sleep operation should be as minimum as possible, since the operation is done disabling interrupts. The present design has an overhead of maintaining an ordered sleep list, thereby increasing the complexity of the sleep operation. The earlier design did not have this overhead.

Interrupt handlers run asynchronously and thus interrupt other potentially important code, including other interrupt handlers. Therefore, to avoid stalling interrupted code for too long, interrupt handlers need to run as quickly as possible.

The major drawback of the earlier design was that it spent more time traversing the sleep list in the timer interrupt handler, during its wakeup operation. The present design makes a compromise by maintaining an ordered sleep list, so that examining the head node will suffice in the wakeup operation, thereby reducing the time spent in the interrupt handler.

3.3 The priority scheduler

3.3.1 Prelude

The priority scheduler of burritos is based on following policies:-

1. When a thread that has a higher priority than that of the currently running thread, is added to the ready list, the current thread should immediately yield the processor to the new thread.
2. When threads are waiting for a lock, semaphore, or condition variable, the thread with the highest priority should be awakened first.
3. A thread may raise or lower its own priority at any time, but lowering its priority such that it no longer has the highest priority will cause it to immediately yield the CPU.
4. If multiple threads have the same priority, they are scheduled in a round robin fashion.

One issue with priority scheduling is ‘priority inversion’. The following scenario illustrates priority inversion:-
H, M, and L are three threads with high, medium and low priorities respectively. If H needs to wait for L (for instance, for a lock held by L), and M is on the ready list, then H will never get the CPU because the low priority thread will not get any CPU time. A partial fix for this problem is for H to ‘donate’ its priority to L while L is holding the lock, and then to recall the donation once L releases (and thus H acquires) the lock.

We handle multiple donations, in which multiple priorities are donated to a single thread. Nested donation is also handled, wherein H is waiting on a lock that M holds and M is waiting on a lock that L holds, then both M and L should be boosted to H’s priority.

3.3.2 Data Structures

1. A list of all threads ready for execution (ready list) ordered by thread priority.
2. A list of lock elements (lock list.) Figure 3.2 shows the structure of the each lock element.

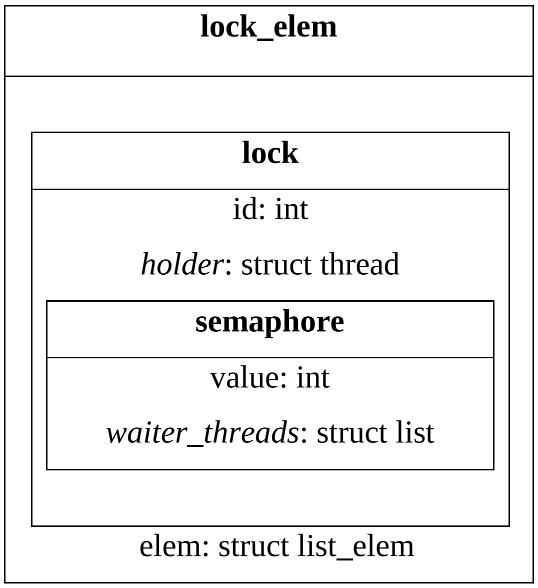

Figure 3.2

3.3.3 Algorithm

The priority scheduler first invokes the priority donation algorithm in case of priority inversion. Then, it schedules the thread with the maximum priority from the ready list.

Test for Priority Inversion

The test involves checking whether a lock holder has lesser priority than any of its respective waiters, for each lock element in the lock list.

Priority Donation

First, the lock list is sorted in decreasing order of holder priorities. Next, the waiter with the maximum priority is calculated for each lock element. If the priority of such a waiter is greater than that of the holder, the waiter donates its priority to the holder. Before donation, the priority of the holder is saved, so that it can later be restored, when it releases the lock.

The lock list is sorted to handle a situation (nested donation) like the following:-

A, B, and C are three threads with priorities 1, 2 and 3 respectively. Thread C waits for thread B and in turn, thread B waits for thread A. There are 2 possible orders of donating priorities:-

1. B donates to A and then, C donates to B.
2. C donates to B and then, B donates to A.

In the first case, A gets a priority of 2 and B gets a priority of 3; and priority inversion still persists. In the second case, both A and B gets a priority of 3 and thus priority inversion is eliminated. Thus, the lock list is sorted in decreasing order of holder priorities.

3.3.4 Synchronization

There can be a potential race between lowering the priority of a lock holder and priority donation from one of its waiters.

The priority of the holder is saved before donation, so that it can be restored later after it releases the lock. If such a holder tries to lower its priority after donation, a change is made only to the saved priority, leaving the current priority unaltered.

3.3.5 Rationale

Initially, the design included priority donation within the 'lock acquire' operation. The algorithm worked as follows:-

If an attempt is made to acquire a lock held by a lower priority thread, then the thread trying to acquire the lock will donate its priority to the holder and finally wait for the lock.

The earlier design did not involve the overhead of maintaining a list of locks and running the priority donation algorithm. But, it did not handle the case of nested donation. The present design finds a middle ground by implementing nested donation and thereby completely eliminating priority inversion, though it involves the complexity of maintaining the lock list.

3.4 The advanced scheduler

3.4.1 Prelude

The goal of a general-purpose scheduler is to balance threads' different scheduling needs. Threads that perform a lot of I/O require a fast response time to keep input and output devices busy, but need little CPU time. On the other hand, compute-bound threads need to receive a lot of CPU time to finish their work, but have no requirement for fast response time. Other threads lie somewhere in between, with periods of I/O punctuated by periods of computation, and thus have requirements that vary over time. A well-designed scheduler can often accommodate threads with all these requirements simultaneously.

We implement an advanced scheduler that resembles the 4.4 BSD scheduler, which is one example of a multilevel feedback queue scheduler. This type of scheduler maintains several queues of ready-to-run threads, where each queue holds threads with a different priority. At any given time, the scheduler chooses a thread from the highest-priority non-empty queue. If the highest-priority queue contains multiple threads, then they are scheduled in 'round robin' order.

Burritos allows users to choose a scheduling policy at startup time. By default, the priority scheduler is active, but the 4.4BSD scheduler can be chosen with the '-mlfqs' kernel option.

The 4.4BSD scheduler does not include priority donation.

3.4.2 Algorithm

<u>Niceness of a thread</u>

Each thread has an integer nice value that determines how 'nice' the thread should be to the other threads. A nice of zero does not affect thread priority. A positive nice decreases the priority of a thread and causes it to give up some CPU time it would otherwise receive. On the other hand, a negative nice tends to take away CPU time from other threads. The initial value of nice is zero in the first thread created, or the parent's value in other new threads. Nice value and hence the priority of a thread can be updated by a set of thread management functions during the course of its execution.

Calculation of priority

The advanced scheduler has 64 priorities and thus 64 ready queues, numbered 0 (PRI_MIN) through 63 (PRI_MAX.) The priority of a thread is first calculated at thread initialization. It is recalculated once every fourth clock tick, for every thread. In either case, it is determined by the formula:-

priority = PRI_MAX - (recent_cpu / 4) - (nice * 2),

where 'recent_cpu' is an estimate of the CPU time the thread has used recently and nice is the thread's nice value. The calculated priority is always adjusted to lie in the valid range PRI_MIN to PRI_MAX. This formula ensures that a thread that has received CPU time recently is given a lower priority. This is a key to preventing starvation: a thread that has not received any CPU time recently will have a recent_cpu of 0, which barring a high nice value should ensure that it receives CPU time soon.

Calculation of the recent CPU time

The initial thread starts with a recent cpu time of zero. Other threads start with a recent cpu value inherited from their parent thread. On each timer interrupt, the recent cpu time is incremented by 1 for the running thread, unless the idle thread is running. In addition, every second, the recent cpu time is recalculated for every thread (whether running, ready, or blocked), as follows:-

recent_cpu = (2*load_avg) / (2*load_avg + 1) * recent_cpu + nice,

where load_avg is a moving average of the number of threads ready to run. If load_avg is 1, indicating that a single thread, on average, is competing for the CPU, then the current value of recent_cpu decays to a weight of .1 in $\log_{2/3}.1 \approx 6$ seconds; if load_avg is 2, then decay to a weight of .1 takes $\log_{3/4}.1 \approx 8$ seconds. The effect is that recent_cpu estimates the amount of CPU time the thread has received 'recently,' with the rate of decay inversely proportional to the number of threads competing for the CPU.

Calculation of the system load average

The system load average estimates the average number of threads ready to run over the past minute. Unlike priority and recent cpu, the load average is system-wide, not thread specific. At system boot, it is initialized to 0. Every second thereafter, it is updated according to the following formula:-

load_avg = (59/60)*load_avg + (1/60)*ready_threads,

where ready_threads is the number of threads that are either running or ready to run at time of update (not including the idle thread.)

3.4.3 Rationale

In the formulae above, priority, niceness, and number of ready threads are integers, but recent cpu and load average are real numbers. Unfortunately, burritos does not support floating-point arithmetic in the kernel, because it would complicate and slow down the kernel. Real kernels often have the same limitation, for the same reason. Thus, all calculations on real quantities are simulated using integers. In burritos, we use a 17.14 fixed point number representation, i.e, the rightmost 14 bits of the integer represent the fractional part of the real number, that integer represents.

3.5 Deadlock Avoidance

3.5.1 Prelude

Modern computers consist of a wide variety of resources that can be used by one process at a time. Consequently, operating systems must have the ability to grant exclusive access to certain resources. Exclusive access to resources, in burritos, is accomplished with the help of *locks*. That is, if a thread has to access any resource, it must first acquire a lock (associated with that resource), complete its job with that resource, and then release the lock, so that any other thread waiting on the lock can now acquire it and start using the resource.

During its course of execution, a thread needs exclusive access to not just one resource, but several. Thus, a thread currently holding resources granted earlier can make a request for a new resource. The thread may have to wait for the new resource, if it is currently held by any other thread. If there is a circular chain of such threads which currently hold a resource (and hence the lock associated with it) and wait for other resources (or locks), a situation arises where no thread can run to completion, and such a situation is called a *deadlock*. Burritos avoids such situations by allowing threads to acquire a lock only when it does not result in a circular dependency.

3.5.2 Data Structures

The following lists are maintained:-

1. The lock list (described in section 3.3.)
2. A list of all living threads (ready, running or blocked.)

These lists are used to construct a temporary data structure, the wait-for-graph (WFG.)
The WFG has the following properties:-

1. Each vertex in the graph represents a thread in the system (ready, running or blocked.)
2. There is an edge from vertex v_i to vertex v_j if the thread represented by v_i is waiting for the thread represented by v_j.

3.5.3 Algorithm

When a thread W makes a request for a lock held by a thread H, the following operations are performed before blocking W on that lock:-

1. A WFG is constructed with the help of the global lock list and thread list.
2. An edge from the thread W to the thread H is added to the WFG.
3. The WFG is then fed into a cycle detection algorithm (the source removal algorithm is employed here.)
4. If a cycle is detected, the request for the lock is denied, since a circular dependency exists. Otherwise, the thread W is blocked on that lock.

3.5.4 Synchronization

A race condition can occur when the lock list gets updated (as a result of lock initialization, acquire or release) when a circular dependency check is being made while processing a lock request. This is avoided by disabling interrupts during the circular dependency check.

If the holder of a lock terminates itself after completing its execution (or is abnormally terminated by the kernel) without releasing the lock it holds, then all the waiters of the lock can never run to completion. Even worse, if one of these waiters holds some other lock, then the waiters for that lock also cannot run to completion. Thus, all locks held by a thread getting terminated, are released before it is completely destroyed.

3.5.5 Rationale

The lock list, by itself, was sufficient for detecting a deadlock. But, the complexity of the cycle detection algorithm would have been significant in such a design. The present design trades off space for time by using a temporary data structure WFG, so that there can be a significant reduction in the complexity of the cycle detection algorithm. Since the WFG is implemented using a 2-D array (a common data structure used to represent a graph), standard algorithms such as the ‘source removal’ or the DFS algorithm can be used for cycle detection.

4. User Processes and System Calls

4.1 Design of the Process Management Utilities

4.1.1 Prelude

All of the code that ran under pintos had been a part of the operating system kernel and thereby had full access to privileged parts of the system. But once user programs are allowed to run on the top of the operating system, this is no longer true. Burritos extends the thread system developed in the previous stage to allow users to load programs from the disk and run a number of such programs at a time.

The thread management functions of pintos are developed into sophisticated process management utilities, which include process creation and termination. Processes in burritos are single threaded. The first process, init, has the process ID 1, and is created by burritos during boot time. All other processes of the system are descendants of the init process.

4.1.2 Data Structures

The Process Control Block (PCB) of a process is one of the most important data structures used for process management in an operating system. Figure 4.1 shows the structure of the PCB of a process in burritos.

4.1.3 Algorithm

<u>Process Creation</u>

Process creation in burritos involves the following steps:-

1. Create a new thread.
2. Insert the newly created thread into the parent's list of children.
3. Initialize the 'zombie' semaphore of the thread to 0.

thread
tid: tid_t
status: enum thread_status
name: char[16]
stack: uint8_t*
old_priority: int
priority: int
elem: struct list_elem
pagedir: uint32_t*
user_stack_size: int
children: struct list
load: struct semaphore
wait: struct semaphore
zombie: struct semaphore
executable: struct file*
load_status: int
exit_status: int
fd_list: struct list
nice: int64_t
recent_cpu: int64_t
magic: unsigned
current_directory: struct dir*

Figure 4.1

4. Allocate and activate the thread’s page tables.
5. Set the thread’s kernel stack for use in processing interrupts.
6. Receive the string containing the name of the ELF (Executable and Linking Format) executable and the arguments passed to the newly created process by the parent process.
7. Parse the string for arguments and push them into the stack of the process.
8. Load the ELF executable into the process’s address space.
9. Deny write permission to the executable.

10. Return the load status to the parent process.
11. Start the user process by simulating a return from an interrupt.

The Process Wait Operation

The parent process may wait for one of its children to die through the .process wait operation. In particular, when a parent waits for a child, it suspends its execution until the child completes its execution and its exit status is retrieved by the parent. The parent suspends its execution by sleeping on the 'wait' semaphore which is a part of the child's PCB. The child thread after completing its execution awakens the parent and blocks on its zombie semaphore. The parent process then retrieves the exit status of the child and unblocks the child, which finally destroys itself completely. The child is then removed from the parent's list of children.

Process Termination

Process termination in burritos involves the following steps:-

1. Close the executable being run by the process so that writes to the executable is re-enabled.
2. Close all the open files.
3. Release all the locks currently held by the process.
4. Perform an up operation on the 'zombie' semaphores of the process's children that are alive.
5. Activate the base page directory (of the init process) and destroy the process's page directory.
6. Signal the waiting parent.
7. Perform a down operation on the process's zombie semaphore.

4.1.4 Synchronization

The parent process should retrieve the load status of the child to determine whether the executable was successfully loaded. So, it should not return before the executable has completed loading. This is ensured by blocking the parent on the child's 'load' semaphore, and unblocking it only after the child completes loading.

There can be a potential race between the parent waiting for the child and the child getting terminated.

The wait operation includes the following semaphore operations:-

1. Down on ‘wait’ semaphore.
2. Up on ‘zombie’ semaphore.

The exit operation includes the following semaphore operations:-

1. Up on ‘wait’ semaphore.
2. Down on ‘zombie’ semaphore.

Each down (or up) operation in wait has a corresponding up (or down) operation in exit and vice versa. Thus, the order of execution of wait and exit is not of any significance; thereby eliminating the race between them.

Another potential race is between the parent getting terminated without waiting for the child and the child getting terminated. If the parent gets terminated without waiting for its children, it performs an up operation on the ‘zombie’ semaphore of each of its children. The child performs a down operation on its ‘zombie’ semaphore irrespective of its parent waiting for it. Again, the down operation on ‘zombie’ has a corresponding up operation. Thus, the order of termination of parent and child is not of any significance; thereby, eliminating the race between them.

4.1.5 Rationale

An alternative design can be, to include a link to the parent’s PCB, in the child’s PCB, rather than maintaining a list of children in the parent’s PCB. This design requires a list of all threads in the system to be maintained, so that whenever a process P tries to wait on a process C, the list can be used to obtain the PCB of C. The present design reduces this overhead by requiring the parent to search for the child in its own list of children, rather than searching for the child in the list of all threads which is potentially much longer.

4.2 System Calls

4.2.1 Prelude

System calls are programmer's functional interface to the kernel. They are subroutines that reside inside the burritos kernel, and support basic system functions such as halt, exit, exec, wait, create, remove, open, close, filesize, read, write, seek and tell.

In burritos, user programs invoke 'int $0x30' to make a system call. The system call number and any additional arguments are expected to be pushed on the stack before invoking the interrupt. Thus, when the system call handler gets control, the system call number is in the 32-bit word at the caller's stack pointer, the first argument is in the 32-bit word at the next higher address and so on.

4.2.2 Data Structures

file_desc
fd: int
file: struct file*
elem: struct list_elem

Figure 4.2

A list of file descriptors is maintained, per process. The structure of each file descriptor is shown in Figure 4.2.

4.2.3 Algorithm

The exec, exit and wait system calls use the kernel's process management utilities developed earlier (Section 3.1.) The create, remove, open, close, filesize, read, write, seek and tell system calls use file management utilities of pintos (Pintos came along with a flat filesystem.) The halt system call simply powers off the operating system. Additional system calls are implemented in the stages that follow.

User Memory Access

As part of the system call, the kernel must often access memory through pointers provided by a user program. The kernel must be very careful about doing so, because the user can pass a null pointer, a pointer to unmapped virtual memory, or a pointer to kernel virtual address space (above virtual address PHYS_BASE.) All of these types of invalid pointers must be rejected without harm to kernel or other running process, by terminating the offending process and freeing its resources.

Our algorithm checks that a user pointer points below PHYS_BASE and then dereferences it. An invalid user pointer will cause a page fault. The page fault handler then returns an error code to the system call handler, if it finds that the faulting address is invalid. The system call handler then terminates the process.

4.2.4 Rationale

Invalid user pointers can be rejected by the kernel through an alternate design, wherein the validity of a user pointer is verified and then dereferenced. This design is much simpler than the present design in which an invalid pointer causes a page fault. However the technique used in the present design is normally faster because it takes advantage of the processor’s MMU.

5. Virtual Memory

5.1 Memory terminology

5.1.1 Page

A page, sometimes called a virtual page, is a continuous region of virtual memory, 4,096 bytes (the page size) in length. A 32-bit virtual address can be divided into a 20-bit page number and a 12-bit page offset, as shown below:-

31 12	11 0
Page Number	Offset

Virtual Address

Figure 5.1

5.1.2 Frame

A frame is a continuous region of physical memory. Like pages, frames must be page-size and page-aligned. Thus, a 32-bit physical address can be divided into a 20-bit frame number and a 12-bit frame offset as shown below:-

31 12	11 0
Frame Number	Offset

Physical Address

Figure 5.2

5.1.3 Page Table

In Burritos, a page table is a data structure that the CPU uses to translate a virtual address to a physical address, that is, from a page to a frame. The page table format is dictated by the 80x86 architecture.

The diagram below illustrates the relationship between pages and frames. The virtual address, on the left, consists of a page number and an offset. The page table translates the page number into a frame number, which is combined with the unmodified offset to obtain the physical address, on the right.

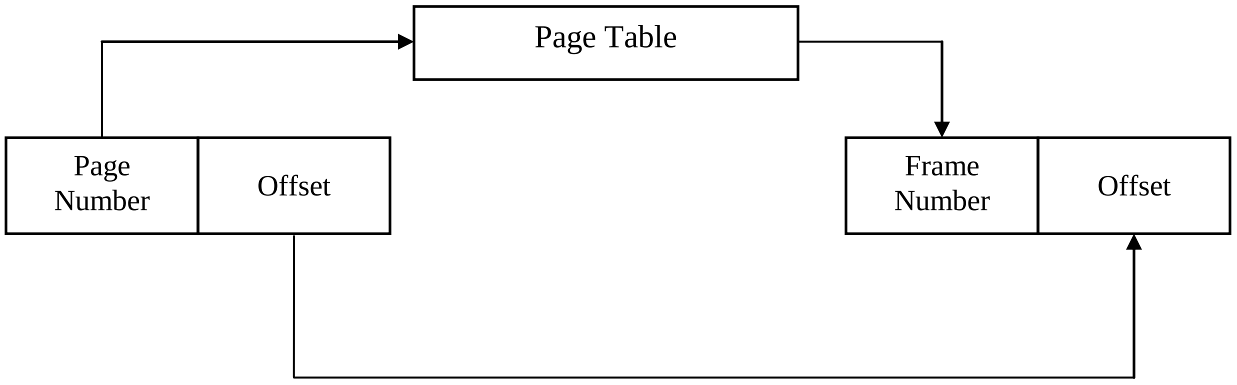

Figure 5.3

5.2 Frame Table Management

5.2.1 Prelude

The frame table contains an entry for each frame that contains a user page. Each entry in the frame table contains a list of pointers to the page table entries of all the processes sharing the frame. The frame table allows burritos to efficiently implement an eviction policy, by choosing a page to evict when no frames are free.

5.2.2 Data Structures

frame_elem
frame_addr: uintptr_t
pte_list: struct list
flags: int
sector_no: disk_sector_t
read_bytes: size_t
elem: struct list_elem

Figure 5.4

The frame table is constructed using a list of frame elements. The structure of each frame element is shown above. Each frame element contains the address of the frame it refers to, a list of page table entries of all processes sharing the frame, flags indicating the type and present status of the frame (can be a combination of flags listed in table 5.1), sector number of the swap device, and number of non-zero bytes (read bytes) .

Flag	**Represents**
FRAME_MMAP	A frame of a memory mapped file.
FRAME_EXEC	A frame of an executable.
FRAME_SWAP	A swapped out frame.
FRAME_DIRTY	A dirty frame (modified.)
FRAME_ACCESSED	A recently referenced frame.
FRAME_IO	A frame being read/written from/into the disk.

Table 5.1

5.2.3 Algorithm

<u>Construction of the frame table</u>

A new entry in the frame table is made when:-

1. The executable of a process is being loaded.
2. A file is mapped onto memory (mmap.)
3. The stack segment of a process grows beyond a page.

In the former two cases, the newly created frame element is marked as 'swapped out' with its 'sector number' attribute indicating the sector at which the corresponding data is found, so that these pages can be fetched on demand from the respective sector on the disk and loaded onto the physical frames. Whereas, in the latter case, the frame element will contain the address of the newly allocated physical frame to which the stack page is mapped onto.

The procedure described above is carried out only when the frame being referred to, by the frame element, is not in the memory or any of the swap devices, and is a writable segment of an executable. In all other cases, the frame has to be shared, that is, the frame element would have already been created, and only a new entry will be made in its list of page table entries; and the page table entry itself will be updated to contain the physical address, through the 'frame address' attribute of the frame element.

<u>Destruction of a process's Page Directory</u>

The page directory of a process is destroyed when it ceases to exist, that is, all the memory allocated to the process including the page directory itself, is reclaimed. This involves the following steps:-

1. Retrieve the frame element corresponding to each page table entry of the process.
2. Remove the page table entry (PTE) from the frame element.
3. If the frame is not shared by any other process (the list of PTEs is empty), then:-
    a. Free the sectors allocated to the frame, on the disk, if it is a data or a stack page that has been swapped out.
    b. Write the frame back to the disk, if it is a frame of a memory mapped file and has been modified; and discard it otherwise.

Frame Eviction

The most important operation on the frame table is obtaining an unused frame. This is trivial when a frame is free. When none is free, a frame is made available by evicting a victim frame. The eviction algorithm employed in burritos is the 'clock' algorithm.

If such a victim frame is found dirty (modified), then following checks are performed:-

1. If the victim's frame element still indicates that it is an all-zero frame, then the number of non-zero bytes is recomputed.
2. If its frame element indicates that, it is a page from an executable, reset the corresponding flag to suggest that, it should no longer be read from the executable.

Next, the victim frame is swapped out.

A swapped out frame or a frame involved in disk I/O, is not a candidate for eviction.

Syncing Aliases

Aliases are two (or more) pages that refer to the same frame. When an aliased frame is accessed, the accessed and dirty bits are updated by the CPU, in only one page table entry (the one for the page used for access.) The operating system takes care of syncing the dirty and access bits of all aliases before any operation (insert/delete/update) is performed on the frame table.

5.2.4 Synchronization

Race conditions can occur when two (or more) user processes access the frame table simultaneously. The kernel keeps the frame table consistent and avoids races by requiring a process to acquire a lock before it:-

1. Obtains a new frame.
2. Faults on some page.
3. Relinquishes its frames, when it destroys its page directory after completing execution

5.2.5 Rationale

Scope of the Frame Table

A design using a global frame table is prone to many race conditions, since multiple processes can access the frame table simultaneously. A local (per-process) frame table in each process makes synchronization easy and thereby, avoids such races.

When two (or more) processes share their pages, the entries corresponding to the shared pages, in the frame table of each process, need to be consistent. This can be a significant overhead. In the global frame table implemented in the present design, each entry contains a list of page table entries of all processes sharing that frame. This design, by its nature, maintains a single entry in the frame table for all processes, sharing that frame, which completely eliminates the consistency requirement associated with a design that uses per-process frame table.

Placement of the Frame Table

To simplify the design, the frame table is stored in non-pageable memory, which ensures that no part of the frame table ever gets swapped out.

5.3 The Page Fault Handler

5.3.1 Algorithm

The kernel raises a page fault exception when a process accesses a page that is not in memory. The page fault is serviced as follows:-

1. The page fault handler first retrieves the address of the faulting page from one of the CPU control registers.
2. If the page is unmapped, that is, if there's no data there, or if the page lies within kernel virtual memory, or if the access is an attempt to write to a read-only page, then the access is invalid. Any invalid access terminates the process and thereby frees all of its resources.

3. If the memory reference is valid, the frame table entry associated with the page that has faulted is obtained.
4. The frame table entry is used to locate the data that goes in the page, which might be in the file system, or in a swap slot, or it might simply be an all-zero page, or it might already be swapped into the memory by another process, which shares that page.
5. If the frame is already present in the memory (swapped in by a process that shares the page), then the frame only needs to be located. Otherwise, a new frame is obtained to store the page. And when all the memory has been exhausted, a frame must be made available by evicting some victim frame, chosen by the eviction algorithm.
6. Next, the data is fetched into the frame, by
    a. Reading it from the file system or a swap slot if the frame is on the disk.
    b. Zeroing it out if the frame table entry indicates that it is an all-zero page.

    If the frame is already available in memory as a page of another process sharing this page, then no action is necessary in this step.
7. The page table entry (PTE) of the page and all its aliases will be updated to indicate that the page and its aliases are now available in memory.
8. The process is put back into the ready queue, so that it can resume execution from the faulting instruction, when it is next scheduled to run.

5.3.2 Synchronization

Whenever a process Q’s frame is evicted, the page table entries (PTEs) pointing to this frame are updated to indicate that the frame is no longer available in memory, so that all accesses to the frame by Q will result in a page fault. Interrupts are disabled during this operation in order to achieve atomicity, so that when a page fault in process P causes another process Q's frame to be evicted, Q will not be able to access or modify the page during the eviction process.

But, there can still be a race between a process P evicting another process Q’s frame and Q faulting on the page. Such a race is avoided by prohibiting more than one process to execute in the page fault handler simultaneously, by using a lock.

Suppose a process P faults on some page, reads it from the file system or swap, and is put back into the ready queue; it must be guaranteed that a second process Q should not evict the frame, while P is still in the ready queue, and until it has successfully resumed execution at the faulting instruction. This is ensured by forbidding the frame from participating in eviction until the page fault has been completely serviced and the process successfully restarts the faulting instruction.

5.3.3 Rationale

Using many locks in the VM synchronization design allows for high parallelism, but complicates synchronization and raises the possibility for a deadlock. The present design uses a single lock to make synchronization easy, but makes a compromise on parallelism.

5.4 Memory Mapped Files

The file system is most commonly accessed with ‘read’ and ‘write’ system calls. A secondary interface is to *map* the file into virtual pages, using the ‘mmap’ system call. The program can then use memory instructions directly on the file data.

The mmap system call

The mmap system call maps an open file into the process’s virtual address space. The entire file is mapped into consecutive virtual pages starting at the virtual address provided by the user.

The VM system loads pages on demand into the memory mapped (mmap) regions and uses the mmap’d file itself, as a backing store for the mapping. That is, evicting a page mapped by mmap writes it back to the file it was mapped from.

If the file’s length is not a multiple of size of a page, then some bytes in the final mapped page stick out beyond the end of the file. These bytes are set to zero when the page is faulted in from disk, and are discarded when the page is written back to disk.

If successful, the system call returns the virtual address at which the file is mapped. On failure, it returns -1. A call to mmap may fail if:-

1. The file opened has a length of zero bytes.
2. The virtual address at which the file is mapped is not page-aligned.
3. The range of pages mapped overlaps any existing set of mapped pages, including the stack or pages loaded from the executable.
4. The virtual address at which the file is mapped is 0, because the kernel assumes that virtual page 0 is not mapped.
5. The file being mapped has a file descriptor 0 or 1, representing console input and output.

The munmap system call

The munmap system call unmaps the mapping present at the virtual address given by the user, which must be an address returned by a previous call to mmap by the same process. When a mapping is unmapped, all pages written to by the process are written back to the file. The frame table entries corresponding to the pages in the mmap'd region are removed from the frame table and the memory utilized by those pages is reclaimed. All mappings are implicitly unmapped when a process exits.

Closing or removing a file does not unmap any of its mappings. Once created, a mapping is valid until munmap is called or the process exits, following the UNIX convention of removing an open file.

5.5 Stack Growth

In pintos, user processes were limited to a page long stack present at the top of the user virtual address space. But, the stack in burritos is made to grow past its current size by allocation of additional pages as necessary. Additional pages are allocated only if they 'appear' to be stack accesses. Access to any address lying between the current user stack boundary and the predefined stack limit, is a stack access. The first stack page is not allocated lazily. That is, it is allocated and initialized with the command line arguments at load time, with no need to wait for it to be faulted in.

6. File System

Implementation of a hierarchical namespace

In the basic file system, all files live in a single directory. A hierarchical file system modifies this to allow directory entries to point to files or to other directories.

A separate current directory is maintained for each process. At startup, the root directory is set as the initial process's current directory. When one process starts another with the exec system call, the child process inherits its parent's current directory. After that, the two processes' current directories are independent, so that either changing its own current directory has no effect on the other.

The file management utilities of pintos have been updated so that, anywhere a filename is provided by the caller, an absolute or a relative path name may be used. The open and close system calls are modified to open and close directories. New system calls chdir, mkdir, readdir, isdir and inumber are implemented for directory management.

7. Test Documentation

All the functionalities implemented throughout this work have been tested against an extensive set of test cases provided by the Stanford University. A subset of those test cases has been listed below:-

7.1 Alarm Clock

For each of the following tests, N threads are created, each of which sleeps for a different, fixed duration, M times.

Name of the Test	Description
alarm-multiple	Thread 0 sleeps for 10 sec, thread 1 sleeps for 20 sec and so on. Expected Result: Product of iteration count and sleep duration will appear in increasing order.
alarm-simultaneous	Each thread sleeps equal number of ticks in each iteration. Expected Result: Within an iteration, all threads should wake up on the same tick.

7.2 Priority Scheduler

Name of the Test	Description
priority-change	Verifies that lowering a thread's priority so that it is no longer the highest-priority thread in the system causes it to yield immediately.
priority-fifo	Creates several threads all at the same priority and ensures that they consistently run in the same round-robin order.
priority-donate-multiple	Checks that when multiple threads donate their priorities to a single thread, in effect only the highest priority thread should have donated.

Name of the Test	Description
priority-donate-chain	Handles the case of nested donation. Creates several threads that wait for each other in a chain in decreasing order of priorities, along with a few interloper threads that do not wait for any lock. Expected Result: All threads run to completion in the order of their priorities.
priority-donate-lower	Verifies that attempts to lower the priority will not take effect until donation is released.
priority-donate-sema	Lower priority thread L acquires a lock, then blocks downing a semaphore. Medium priority thread M then blocks waiting on the same semaphore. Next, high priority thread H attempts to acquire the lock, donating its priority to L. Next, the main thread ups the semaphore, waking up L. L releases the lock, which wakes up H. H ups the semaphore, waking up M. H terminates, then M, then L, and finally the main thread.

7.3 Advanced Scheduler

Name of the Test	Description
mlfqs-fair	Measures the correctness of the ‘nice’ implementation. The ‘fair’ tests run either 2 or 20 threads all niced to 0. The threads should all receive approximately the same number of ticks. Each test runs for 30 seconds, so the ticks should also sum to approximately 30 * 100 == 3000 ticks.
mlfqs-load-1	Verifies that a single busy thread raises the load average to 0.5 in 38 to 45 seconds. The expected time is 42 seconds. Then, verifies that 10 seconds of inactivity drop the load average back below 0.5 again.
mlfqs-recent-1	Checks that the recent cpu time is calculated properly for the case of a single ready process.

7.4 Deadlock Avoidance

Name of the Test	Description
deadlock-simple	Thread A holds lock 0 and thread B holds lock 1. Thread B tries to acquire lock 0 and blocks. Thread A now makes a request for lock 1. This should be denied.
deadlock-nest	Thread A waits for thread B and thread B waits for thread C. Now, all requests by thread C for locks held by thread A have to be denied.

7.5 User Processes

Name of the Test	Description
args-*	Verifies that the command line string has been correctly parsed and the arguments are properly pushed onto the stack.
exec-multiple	Execs and waits for multiple child processes.
exec-missing	Tries to execute a non existent process. Exec should return -1.
rox-multichild	Ensures that the executable of a running process cannot be modified, even in presence of multiple children.
wait-simple	Wait for a subprocess to finish.
wait-twice	Wait for a subprocess to finish, twice. The first call must wait in the usual way and return the exit code. The second wait call must return -1 immediately.
multi-oom	Recursively executes itself until the child fails to execute. It is expected that at least 30 copies can run. The number of children the kernel was able to execute is counted before it fails to start a new process. It is required that, if a process doesn't actually get to start, exec() must return -1. This process is repeated 10 times, checking that the kernel allows for the same level of depth every time. In addition, some processes will spawn children that terminate abnormally after allocating some resources.

Name of the Test	Description
multi-child-fd	Opens a file and then runs a subprocess that tries to close the file. (Burritos does not have inheritance of file handles, so this must fail.) The parent process then attempts to use the file handle, which must succeed.

7.6 User Memory Access

Name of the Test	Description
bad-jump/bad-read/ bad-write	Ensures that all attempts to access unmapped addresses or the kernel virtual addresses will cause a user process to be immediately terminated with an exit code of -1.

7.7 Paging

Each of the following test cases was made to run with a limited memory of only 2MB, so that key features of our virtual memory system such as handling of page faults, swapping in and swapping out of pages, frame eviction and sharing of pages are rigorously tested.

Name of the Test	Description
page-linear	Encrypts, then decrypts, 2MB of memory and verifies that the values are as they should be.
page-parallel	Runs 4 child processes at once. Each child process encrypts 1MB of zeroes, then decrypts it, and ensures that the zeroes are back. Since the job of each child process is the same, they run the same executable, and hence read only pages of each process are shared. It is always possible that these shared pages get evicted because of limited memory. Hence, consistency must be maintained when swapping in and swapping out such shared pages.
page-merge-seq / page-merge-par	Generates 1 MB of data that is then divided into 16 chunks. A separate sub-process sorts each chunk in sequence/parallel. We then merge the chunks and verify that the data is intact.

7.8 Memory Mapped Files

Name of the Test	Description
mmap-write	Writes to a file through a mapping, and unmaps the file, then reads the data in the file back using the read system call to verify.
mmap-twice	Maps the same file into memory twice and verifies that the same data is readable in both.
mmap-clean	Verifies that mmap'd regions are only written back on munmap if the data was actually modified in memory.
mmap-exit	Executes a child process that mmaps a file and writes to it via the mmap'ing, then exits without calling munmap. The data in the mapped region must be written out at program termination. The parent verifies that the writes that should have occurred really did
mmap-remove	Deletes and closes file that is mapped into memory and verifies that it can still be read through the mapping.

7.9 Stack Growth

Name of the Test	Description
pt-grow-stack	Demonstrates that the stack can grow and this must succeed.
pt-grow-pusha	Expands the stack by 32 bytes all at once using the PUSHA instruction. This must succeed.
pt-grow-bad	Reads from an address 4,096 bytes below the stack pointer. The process must be terminated with an exit code of -1.

7.10 Hierarchical file system

Name of the Test	Description
dir-mk-tree	Creates directories /0/0/0 through /3/2/2 and creates files in the leaf directories.
dir-open	Opens a directory, then tries to write to it, which must fail.
dir-over-file/ dir-under-file	Tries to create a file with the same name as an existing directory (or vice-versa), which must return failure.

8. Future Work

This focus of this project was to get a hands-on learning experience by developing a primitive kernel like pintos into a kernel similar to that of the present day operating systems like UNIX. While many lower level features in the key areas of process, memory and file management have been designed and implemented in this work, we still have a long way to go in the process of modernizing pintos. Potential future works include:

- Multi-threaded Processes. Processes in burritos have only one thread of execution. A single threaded process is not well suited for many applications such as those used in the internet, where speed is imperative. The process management utilities of burritos can be further strengthened to make processes multi-threaded and improve their performance.

- Network Support. The existing burritos kernel does not have enough support for communication over a network or even for communication among processes running on the same machine. Like most modern operating systems, burritos can be extended to support message passing, remote procedure calls and remote login.

- File Synchronization. In order to support database applications, where only one process should be allowed to access a file at any instant of time, a file locking mechanism can be employed in burritos, which currently does nothing to prevent concurrent accesses to files.

- Buffer/Page Cache. File access can be speeded up by the provision of a buffer cache and a page cache. A buffer cache can be implemented to cache recently accessed disk blocks, so that subsequent accesses can be made from the cache rather than the disk.

Bibliography

1. Robert Love, *Linux Kernel Development*, Pearson Education, 2005.

2. Barry B. Brey, *The Intel Microprocessors – Architecture, Programming, and Interfacing*, Prentice Hall of India, 2006.

3. Andrew S. Tanenbaum, Albert S Woodhull, *Operating Systems – Design and Implementation*, Pearson Education, 2005.

4. W. Richard Stevens, *Advanced Programming in the UNIX Environment*, Pearson Education, 2005.

5. All the support materials provided by the Stanford University – Pintos Documentation and Lecture Notes.